CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2013; **25**:1427–1442
Published online 14 October 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.2942

SPECIAL ISSUE PAPER

Distribution of random streams for simulation practitioners[‡]

David R. C. Hill[1,2,3,*,†], Claude Mazel[1,2,3], Jonathan Passerat-Palmbach[1,2,3] and Mamadou K. Traore[1,2,3]

[1]*Clermont Université, BP 10448, F-63000, Clermont-Ferrand, France*
[2]*CNRS, UMR 6158, Université Blaise Pascal, LIMOS, F-63173, Aubiere, France*
[3]*ISIMA, Computer Science & Modeling Institute, BP 10125, F-63177, Aubiere, France*

SUMMARY

There is an increasing interest in the distribution of parallel random number streams in the high-performance computing community particularly, with the manycore shift. Even if we have at our disposal statistically sound random number generators according to the latest and thorough testing libraries, their parallelization can still be a delicate problem. Indeed, a set of recent publications shows it still has to be mastered by the scientific community. With the arrival of multi-core and manycore processor architectures on the scientist desktop, modelers who are non-specialists in parallelizing stochastic simulations need help and advice in distributing rigorously their experimental plans and replications according to the state of the art in pseudo-random numbers parallelization techniques. In this paper, we discuss the different partitioning techniques currently in use to provide independent streams with their corresponding software. In addition to the classical approaches in use to parallelize stochastic simulations on regular processors, this paper also presents recent advances in pseudo-random number generation for general-purpose graphical processing units. The state of the art given in this paper is written for simulation practitioners. Copyright © 2012 John Wiley & Sons, Ltd.

Received 25 January 2011; Revised 13 June 2012; Accepted 7 September 2012

KEY WORDS: parallel random numbers; parallel random streams; high-performance computing; random number generation; stochastic simulation

1. INTRODUCTION

With the arrival of multi-core and manycore processor architectures on the scientist desktop, modelers who are non-specialists in parallelizing stochastic simulations need help and advice in distributing rigorously their experimental plans and replications according to the state of the art in pseudo-random numbers partitioning techniques. New programming models and platforms are now available to a larger audience, particularly with the arrival of very powerful general-purpose graphics processing units (GP-GPUs). Stochastic simulation is now an essential tool for many research domains. Many applications are concerned, particularly when dealing with complex system science. Random number generators (RNGs) are mainly used in simulation to model the stochastic phenomena that appear in the formulation of a problem or to apply an iterative simulation-based problem solving technique (such as Monte Carlo simulation [1]) or coupling between simulation and operations research optimization tools. Pseudo-random number generators (PRNGs) use deterministic algorithms to produce sequences of random numbers and have been studied in pertinent reviews, many from Michael Mascagni and Pierre l'Ecuyer, and also from other

*Correspondence to: CNRS, UMR 6158, Université Blaise Pascal, LIMOS, F-63173 Aubiere.
[†]E-mail: David.Hill@univ-bpclermont.fr

[‡]This article was published online on 14 October 2012. An error was subsequently identified. This notice is included in the online and print versions to indicate that both have been corrected 25 October 2012.

authors, including the well-known first edition of Knuth's Art of Computer Programming, volume 2 and its re-editions [2–6]. If some applications can cope with 'bad' or poor randomness according to current standards, we cannot afford producing biased results for nuclear physics or medicine for instance [7–10]. For parallel stochastic simulations, there is an even more critical need for a large number of parallel and independent random sequences, each with good statistical properties (approximating as close as possible a truly random sequence). We cannot give a mathematical proof of independence between two random sequences, so each time this word is used, it should be considered as (pseudo) independence according to the best accepted practices in mathematics and statistics.

A category of random numbers called quasi-random numbers can improve the convergence speed (and accuracy) of Monte Carlo integration and also of some Markovian analysis [11]. However, these numbers are not independent and can only be used for specific applications. Random numbers produced by deterministic algorithms are called pseudo-random numbers by simulationists and by most researchers, whereas this notion of 'pseudo'-random refers to polynomial-time unpredictability properties in the area of cryptology. In this paper, we do not consider RNGs for cryptographic usage but only for scientific simulations. Truly random numbers could be a very interesting source, but they are not reproducible [6]. They can be supplied for instance by a physical external source such as a radioactive decay, but there are also many drawbacks in their use. True random numbers can be clumsy to produce and use, some devices are subject to partial breakdowns (their production has to be regularly statistically checked), some have biases that have to be corrected, and most of the time, we need to store the sequences to be able to reproduce exactly the same sequence several times. This is not possible for many high-performance computing (HPC) applications. New devices named Quantis, proposed by 'Id Quantique', have generated a strong interest over the last few years. Their principle is based on an optical quantum process as source of randomness, and they are fairly easy to use through PCI cards or USB sticks. This source can deliver a sequence of bits at a high bit rate of 4–16 Mbits/s. This sequence is then post-processed by the Quantis processing unit to remove the potential biases. Although for a supercomputer we could consider the acquisition through many devices, this is not possible with world wide computing grids (such as the European Grid Initiative, EGI) with heterogeneous computing centers. Once again, the major problem in our opinion is that sequences are not reproducible. For small-scale simulations, we can store the sequences or use a central server approach (detailed in Section 3.1.1) similar with [12]. Still, HPC applications, such as nuclear medicine simulations for instance, cannot download, or store the hundreds of thousands of billions of numbers needed to reproduce the experiments [7–10]. If we consider smaller needs in terms of numbers of draws, optimizations using memory mapping and unrolling of pre-stored numbers can however be very efficient for various applications [13]. Reproducing experiments is the essence of science, and even if this is not always necessary, in the case of stochastic simulation, the need for reproducible generators is essential for various purposes, including the analysis of results. To investigate and understand the results, we have to reproduce the same scenarios and find the same confidence intervals every time we run the same stochastic experiment. When debugging parallel stochastic applications, we need to reproduce the same control flow and the same result to correct an anomalous behavior. Reproducibility is also necessary for variance reduction techniques, for sensitivity analysis and many other statistical techniques [14]. In addition, for rigorous scientific applications, we want to obtain the same results if we run the application in parallel or in serial. Consequently, software random number generation remains the prevailing method for HPC, and we will see that specialists are warning us to be particularly careful when dealing with parallel stochastic simulations [15–18].

In this paper, we focus our contribution on the following aspects:

- We point out that parallelization of random streams has still to be mastered by the scientific community. Although a set of fine techniques exists, recent publications show that more attention has to be given to this aspect (Section 2).
- We survey the parallelization methods and their design, and we propose a possible classification for simulation practitioners (Section 3). Reliable initializations of PRNGs are also discussed.
- In Section 4, we present the major PRNG parallelization software and libraries, and then we discuss the empirical testing approaches (Section 5).
- In the last 2 sections, we present considerations raised by the emergence of hybrid computing with GP-GPUs.

DOI: 10.1002/cpe

2. PARALLEL STOCHASTIC SIMULATIONS AND RANDOM NUMBERS GENERATION

Stochastic simulations can require a huge computational capacity particularly when we design experiments to explore large parameter spaces [14]. New technologies in HPC and networking, including hybrid computing, have very significantly improved our computing capacities. The major consequence is the increased interest for parallel and distributed simulation [19]. Major problems arise when dealing with model partitioning and with the stochastic aspects, but in many cases, effective solutions can be developed. Communications protocols (synchronous simulation versus asynchronous) have to be examined seriously to avoid deadlocks and to preserve the causality and determinism principles. When dealing with stochastic parallel simulation, we have to mainly consider two aspects: the quality of the PRNG and the parallelization technique. If stochastic sequential simulations always require a statistically sound PRNG, in the case of parallel simulations, this requirement is even more crucial. As a consequence, simple linear congruential generators (LCGs) should be definitively banned from modern HPC applications [6]. The parallelization technique should ideally generate pseudo-random numbers in parallel, that is, each processing element (PE) should autonomously obtain either its own random sequence or its own subsequence of a global sequence (partitioning of a main sequence). If such independence is not guaranteed, the parallelism is affected. The sequences assigned to each PE must not depend on the number of processors: if the latter varies, we cannot reproduce the simulation experiments. For example, in the case of the central server approach, the PE access to the central RNG in an order that relies on scheduling policies and network communications, which does not ensure the same PE will have the same numbers every time. Such a configuration disables reproducibility, and it also often creates a bottleneck. Designers of parallel stochastic simulations always have to reply to this fundamental question: how can we make a safe RNG repartition to keep, on the one hand, efficiency, and on the other hand, a sound statistical quality of the simulation to obtain credible results? Indeed, the validation of such parallel simulations is a critical issue. Paul Coddington precisely states: 'Random number generators, particularly for parallel computers, should not be trusted' [20]. Much research has been undertaken to design good sequential RNGs. Whatever the parallelization technique is, we have to rely on a 'good' generator according to a set of main principles proposed by specialists [20–22]. Among the best generators currently available, we can cite Mersenne Twister (MT hereafter) [23] introduced in 1997. SFMT [24] is currently less known. It is an SIMD-oriented version of the original Mersenne Twister generator with the following improvements: speed (twice as fast as MT), a better equidistribution and a quicker recovery from bad initialization (zero-excess in the initial state). The periods being supported by SFMT are incredibly large, ranging from $2^{607}-1$ to $2^{216091}-1$. The WELL generators (Well Equidistributed Long-period Linear), on the basis of similar principles (generalized feedback shift register), have been produced by Panneton in collaboration with L'Ecuyer and Matsumoto [25]. L'Ecuyer suggests that multiple recurrence generators (MRGs) with much smaller periods (above 2^{100} but less than 2^{200}), such as MRG32k3a [26], can also have very interesting statistical properties, and are easier to parallelize according to our current knowledge. At the end of the 1980s, as the interest for distributed simulation increased, numerous research works took on to design parallel RNGs [27–33]. Assessing the quality of random streams remains a hard problem, and many widely used partition techniques have been shown to be inadequate for some specific applications. For instance, recent studies in networking and telecommunication question the relevance of many stochastic simulations because of the use of poor quality RNGs [34, 18, 35], as well as the mediocre parallelization techniques used in some software [36]. The next section presents the main advantages and drawbacks of partitioning techniques.

3. DESIGN OF PARALLEL AND DISTRIBUTED RANDOM STREAMS

There are many approaches to obtain parallel random numbers streams either by partitioning the main sequence (stream) of a given generator into subsequences (substreams) or by parameterizing one generator to have several sequences (multiple streams). Different techniques have been surveyed in [37] and more recently in [38]. This section tries to give a thorough overview of the distribution techniques found in the literature. Here, we propose taxonomy of distribution techniques, depending on the type of

DOI: 10.1002/cpe

original random streams they target. We also try to identify the scope of each of these techniques, so that any practitioner can refer to this section to figure out which technique best fits his application.

3.1. Partitioning a unique original stream

Any PRNG whose state vector is n-bits wide generates, from an initial state, a periodic random number sequence whose period is inevitably less than 2^n. Random numbers distribution techniques introduced in this section share a common principle referred to as cycle division in [39]. It splits random numbers from the considered cycle between the different PEs. Techniques detailed hereafter differ from each other only by the method they use to split the main stream. Basically, we consider four main partitioning techniques.

3.1.1. Central server technique. This first approach is not truly parallel. It consists in using a central server, running an RNG and providing on-demand pseudo-random numbers to different PEs. This approach displays two major drawbacks. First, a simulation using this technique will not be reproducible because of scheduling policies that might change the order in which numbers will be provided to PEs. This is very problematic when trying to debug a simulation because the same numbers will not necessarily be assigned to the same PE at each run. Moreover, the results of a simulation cannot be checked by other scientists for the same reason: there are few chances that the same random sequence will be issued twice. Second, the central server approach will create a bottleneck if too many PEs are considered. As a consequence, this is suitable for serious games with limited parallelism but not for scientific applications.

3.1.2. Sequence splitting. The sequence splitting method is also known as 'blocking' or 'regular spacing'. It consists in allocating non-overlapping, contiguous and equally sized blocks from the original random stream to form substream. When partitioning a sequence $\{x_i,\ i=0,\ 1,\ \ldots\}$ into N streams, the jth stream is $\{x_{k+(j-1)m},\ k=0,\ \ldots,\ m-1\}$, where m is the length of each stream; m must be chosen so that each stream is long enough to achieve the stochastic simulation performed by the corresponding process. For instance in [40], Hechenleitner showed that in the OMNeT++, the spacing between sequences set to 1 million draws led to biased results (due to inter-sequence correlations) for processes using more random numbers. However, the determination of a good value for m is not the only difficulty. If overlapping can be easily avoided, long-range correlations in the initial RNG can lead to small-range correlations between the potential substreams [27, 41]. The impact of this kind of correlation is problematic as shown in [10, 42].

Figure 1 represents an original stream chunked through sequence splitting for 2 and 3 PEs. The schema also insists on the fact that an original stream is split in equally sized parts whose length may vary from one application to another.

3.1.3. Random spacing. The random spacing or indexed sequences method builds a partition of n streams by initializing the same generator with n random statuses. In the case of old LCGs, it was named random seeding. For modern generators with a more complex status, the random statuses are generated with another RNG, and this technique is interesting when generators have a huge period. This technique is easy to set up. In 2006, Wu and Huang [43] showed that the minimum distance between n statuses generated in this way is on average $1/n^2$ multiplied by the period length. The risk is of course to have a bad initialization linked to the fact that two random statuses could be too close to

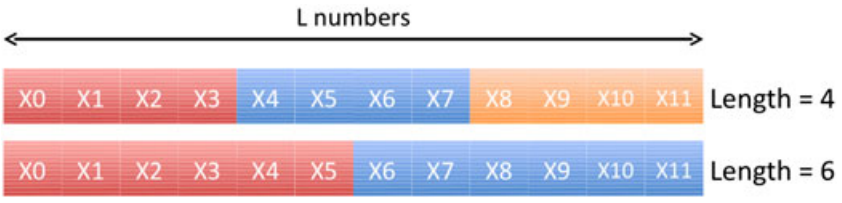

Figure 1. Sequence splitting in a unique original stream considering 2 processing elements (PEs), then 3 PEs.

DOI: 10.1002/cpe

each other, implying an overlapping of corresponding sequences. For a PRNG with a period P, the probability that n sequences of length, L, generated by a random spacing technique will overlap is equal to $1-(1-nL/(\mathrm{P}-1))^{n-1}$, which is equivalent to $n\,(n-1)\,L/\mathrm{P}$ when $n\,L/\mathrm{P}$ is in the neighborhood of 0. A situation implying 3 PEs is sketched in Figure 2 where we can notice the non-equal gaps between two random numbers ranges:

Overlapping risks become sizeable with short-period PRNGs. All the PRNGs tested by Pierre L'Ecuyer and Richard Simard in [44] display periods P from 2^{24} to 2^{131072}. Most of them have $\log_2(\mathrm{P})$ of the order of a few dozens. Now, given that nowadays, longest simulations can consume up to several thousands of billions of random numbers [7–9], ($L=10^{12}$), a hundred of such replications ($n=100$) makes $n\,(n-1)\,L$ on the order of 10^{16} (i.e., more than 2^{53}). The probability to see an overlapping between two subsequences issued from a PRNG of period P far bigger than 2^{53} becomes negligible. Nonetheless, from 20 LCGs PRNGs tested in [44], 14 laid out an overlapping probability greater than 99.9% (period P such that $\log_2(\mathrm{P})<50.4$). Widely spread PRNGs, such as Comblec88 of period $\mathrm{P}=2^{61}$ (combined LCGs), are on the acceptance borderline for this technique (with $n=100$ and $L=10^{12}$, the overlapping probability is equal to 0.43%). They are shipped with several renowned software packages though, including RANLIB, CERNLIB, Boost, Octave and Scilab [44]. Our advice would be however to avoid such LCGs or combined LCGs for modern simulations because of their structural weaknesses [6].

3.1.4. Leap frog. The leap frog (LF) is the way to partition a random stream in the manner of dealing cards to several players. Random numbers are allocated in turn to PE, in the same way as cards are dealt to players. Pragmatically, let each processor hold an i identifier. Every such PE will build a Y_i substream from an X original random stream such as $Y_i=\{X_i, X_{i+N}, \ldots, X_{i+kN}\}$, with N equal to the number of processors [45].

Given the period P of the global sequence, the period of each stream is P/N. As with the splitting technique, the long-range correlations in the initial RNG can lead to small-range correlations between the potential substreams, particularly if we have a large number of PEs. In addition, Wu and Huang [43] showed that depending on the interval used (i.e., the number of PEs and the length of the random sequences), poor spectral values could be observed. A case where the quality of the original RNG is seriously affected by the LF technique is shown in [16]. In addition, when this technique is used without jump ahead (Section 3.4), performances are divided by the number of PEs because of the bottleneck problem appearing in the central server technique.

This technique needs to be used with caution depending on the environment it is set up in. Thus, to preserve reproducibility between two executions of the same simulation, one should not use this technique when the parallelism grain is not fixed yet. As a matter of fact, different random streams will be assigned to PEs depending on the chosen grain. This situation is illustrated in Figure 3.

For the classical LF approach, one must ensure that the original status can be shared between all the PEs when the underlying PRNG algorithm relies on a linear recurrence to produce the next numbers of the sequence. This limits the target architectures of the classic LF technique to shared memory architectures, where a PRNG state can be stored and accessed efficiently by the PEs. For distributed memory architectures, a smart implementation of the LF technique can be used when the number of PEs is a power of 2. In this case, using a unique Lagged Fibonacci generator, each PE can implement its own version of the generator that performs the right jump corresponding to the PE id [46]. This approach is also discussed in Section 4, focusing on GP-GPU section.

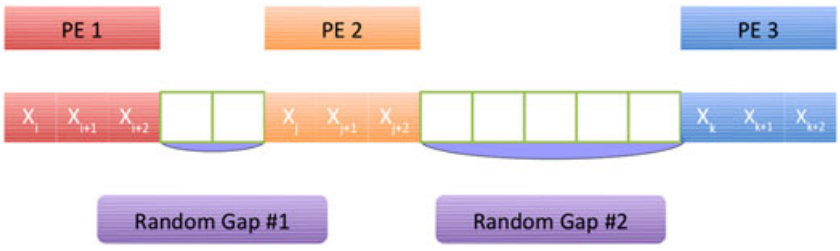

Figure 2. Random spacing applied to 3 processing elements (PEs).

DOI: 10.1002/cpe

Figure 3. Leap frog in a unique original stream considering 2 processing elements (PEs), then 4 PEs.

3.2. Partitioning multiple streams: parameterization

Techniques presented so far tried to split a single stream into several substreams. Another approach consists in using several declinations of the same PRNG: each generator has the same structure and generation mechanism with a unique parameter set, called parameterized status hereafter.

Although no mathematical proof can establish this independence, some implementations of parameterization are safe according to the current state of the art [39]. We especially think to the dynamic creator (DC) algorithm [47] coming along with most of the generators from the Mersenne Twister family. DC integrates a unique identifier, which belongs to the Parameterized Status of the PRNG. This identifier becomes a part of the characteristic polynomial of the matrix that defines the recurrence of the PRNG. Two identifiers will consequently lead to two different Parameterized Statuses. Furthermore, DC ensures that the characteristic polynomials we obtain are mutually prime, and the authors assert that the random sequences generated with such distinct Parameterized Statuses will be highly independent, even if, as mentioned before, this fact cannot be mathematically proven. An example of parameterization can be found in Figure 4, which states three independent parameterized PRNGs.

In the case of LCGs and multiplicative congruential generators, this can rapidly lead to poor results [41, 43] even when the parameters are very carefully checked. For instance Mascagni and Chi proposed that the modulus be Mersenne or Sophie Germain prime numbers [48] to improve the case of LCGs parallelization.

3.3. Other techniques

Now that the main partitioning methods have been presented, we can indulge in a bit of history of other approaches because the parallelization of pseudo-random numbers has been under study for at least 30 years.

Variants have been developed on the basis of classical methods such as the shuffling LF. Its principle is to parameterize both the seed and recursive functions of LCGs. One of the main contributions to this variant is that it results in a scalable period, that is, the number of different random numbers that can be used increases with the number of parallel streams. A parameterization method was used in [32] to obtain parallel streams from a LCG, by choosing for the jth stream a multiplier $a(j)$ and an additive

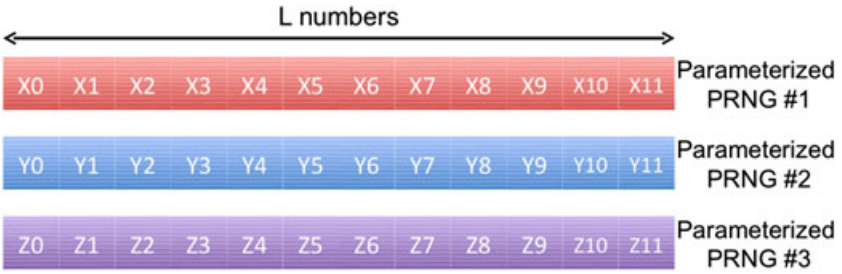

Figure 4. Example of Parameterization for 3 processing elements.

DOI: 10.1002/cpe

constant $c(j)$. It is shown in this case that good results can be obtained if $c(j)$ is the j^{th} prime number less than $\sqrt{(m/2)}$ where m is the modulus. A specific way to generalize the partitioning methods was proposed in the 1980s [49]. It results in what has been called a 'Lehmer tree'. But the suggestion is apparently limited to LCGs, which we still discourage the use.

We can also think of hybrid approaches, but they are not distribution techniques in their own right. Yet they combine several standard distribution techniques to take advantage of their respective features. Please note that the resulting random stream of this combined approach needs to pass through the same test batteries than its parents to be validated. Indeed, combining several distribution techniques might not preserve the statistical quality of the original random stream.

In the end, let us add that Boolean Cellular Automata have been considered to generate parallel pseudo-random numbers by Sipper and Tomassini [50–52], who tested the generated sequences for distributed environments. This technique is rather slow, and [53] has considered its hardware implementation in programmable chips. We are not aware of any evaluation of this technique by any state-of-the-art testing battery; thus, we cannot advise the use of this technique.

3.4. A tool for partitioning: jump-ahead algorithms

A jump-ahead algorithm is a tool used by distribution techniques such as sequence splitting and LF to deal numbers efficiently. This technique enables the analytical computing of the generator state in advance after a huge number of cycles (generations) and corresponds to a jump ahead in the random stream. Knowing the recurrence formula $s_{n+1} = f(s_n)$, where s_{n+1} is the generator status, the difficulty is to determine $f^{(n)} = f \circ f \circ \ldots \circ f$ ($n-1$ compositions of f by itself) because $s_n = f^{(n)}(s_0)$.

Such a technique is interesting with generators that have very large periods. Among widespread generators, MT and WELL generators, on the basis of linear recurrences modulo 2 (F_2-linear generators), embed an efficient software providing jump-ahead facilities. Matsumoto and L'Ecuyer teams joined [54] to develop a viable jump-ahead algorithm running in few milliseconds on current processors. In this case, f is a linear application in F_2^w, with a matrix A with w lines and w columns, where w represents the size of the state vector. The problem is to compute A^n, and this can be achieved, thanks to the characteristic polynomial of A matrix. When an F_2 linear generator proposes a full period $(2^w - 1)$ and when the size of the w state vector is big, the computing of the characteristic polynomial will take time to compute [6].

More efficient algorithms exist, such as MRG32k3a from [26]. However, this generator is twice to three times as slow as MT on common 32-bit computers and also has a much shorter period length, although it can be sufficient for modern applications.

3.5. Joint use of the partitioning of a single stream and parameterization

The two main techniques of random streams distribution, presented in Sections 3.1 and 3.2, are not exclusive.

To simplify, we can say that any number x_n of a sequence provided by a conventional PRNG is the result of a call $x_n = g(s_n)$, where s_n is the internal state of the PRNG and g a function usually very simple. The statistical quality of these generators is therefore only in the complexity of the state transition function f that allows us to deduce the following state: $s_{n+1} = f(s_n)$. However, the downside of this complexity is that it induces a sequential dependence between the successive states of the PRNG. The direct and rapid access to a state s_n (required by the sequence splitting or LF techniques, for example) is possible only if one has a jump-ahead function (Section 3.4).

With counter-based PRNGs [55], so named because $f(s_n) = (s_n + 1) \bmod 2^p$ (where p is the size of the state vector), this sequential dependence disappears because the numbers are generated by calls such as $x_n = g(n)$ where n is the state reduced to a simple counter. The parallelism inherent in the latter relationship allows immediate use of single stream partitioning techniques (Section 3.1).

The statistical quality of counter-based PRNGs are therefore due to the complexity of the g functions used (for PRNGs used in simulation, these functions are simplifications of cryptographic block ciphers, such as AES or Threefish, used in cryptographically secure PRNGs). These functions are indexed by keys, allowing a natural distribution of pseudo-random numbers by parameterization over the key space (Section 3.2).

DOI: 10.1002/cpe

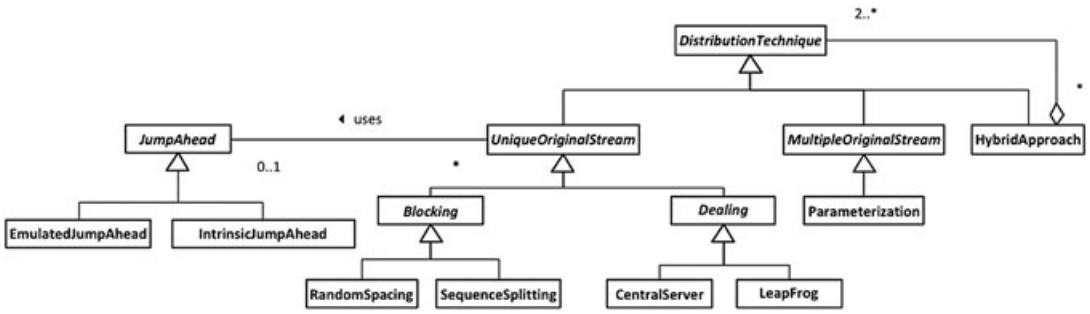

Figure 5. Taxonomy of distribution techniques.

Finally, the interest of the counter-based PRNGs is that the generation of numbers $x_n = g_k(n)$ can be parallelized either by partitioning each stream according to the values of the counter n (Section 3.1) or by following a parameterization approach (Section 3.2) on the basis of the k keys. When the size of counter space and key space is sufficient (up to 2^{128} and 2^{64}, respectively, for counter-based PRNGs presented in [55]), being able to use both techniques presented previously independently even allows, within each PE, to associate a PRNG to each software entity. For instance, in the case of an individual-based stochastic simulation with a very large number of individuals, the sequence splitting technique can be applied in the counter space to distribute the pseudo-random sequences on different replications of the simulation while setting on the keys used to assign each individual its own PRNG.

3.6. Discussions and taxonomy of random streams distribution techniques

For techniques such as sequence splitting and random spacing, we have seen that a common problem is overlapping, but we also have to consider the potential impact of the random initialization on the quality of the underlying PRNG. Recent history has shown that even some of the best RNG algorithms could fail when badly initialized. In the first version of the Mersenne Twister generator, if two initial states were too close with respect to the Hamming distance, then the corresponding output sequences were close to each other. Improvements have been proposed to overcome this problem.[‡] For the initialization problem, the remaining technique is to run empirical or statistical tests such as TestU01. In 2008, Reuillon proposed 1 million statuses for the first Mersenne Twister of period $2^{19937} - 1$: he used a RNG with cryptographic qualities to propose independent and well-balanced bit statuses, knowing that when MT had a zero-excess initialization status, it could take a quite long number of draws to recover good statistical properties [10].

In Figure 5, we propose a Unified Modeling Language class diagram of the main parallelization techniques. The latter are basically ordered depending on the use of either a single stream or multiple streams. Then, the taxonomy is refined considering the strategies set up by distribution techniques: for instance, we distinguish techniques issuing blocks of contiguous numbers in opposition to those dealing numbers. Now, we have seen that techniques based upon unique original random streams might take advantage of a jump-ahead algorithm to improve their generation speed. Basically, this feature concerns the LF and sequence splitting approaches. However, we do not consider jump ahead as a relevant criterion to sort distribution techniques. That is why we decided to allow any UniqueOriginalStream instance to make use of jump ahead. Finally, please note that we chose not to distinguish a particular hybrid approach. Indeed, we consider as hybrid any technique combining at least two of its counterparts. This has been carried out, thanks to the composite design pattern on the right side of Figure 5, which allows us to describe the Hybrid Approach as a class that represents the combination of at least two other distribution techniques.

In a previous paper [56], we have presented the DistMe software toolkit designed to help with the distribution of large parallel stochastic applications. It is a concrete implementation of the techniques and advice provided along this paper and is a concrete implementation of our experience in dealing with pseudo-random numbers for distributed simulations. This paper also presents examples of

[‡]http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html

DOI: 10.1002/cpe

stochastic simulations for life science research where thousands of billions of pseudo-random numbers have been used. The evolution of this software toolkit (OpenMOLE) has been presented in [57]. The preliminary design of the DistMe toolkit was achieved when tackling the distribution of a nuclear medicine application using the largest European computing grid (EGI) [7–9]. At that time, we used sequence splitting and the dynamic creation of Mersenne Twister algorithms [58]. Thanks to the EGI grid, the equivalent of a few years of computing was achieved in a few days. An interesting example of LF usage is given in [46], and efficient usage of the jump-ahead technique is given by simulation software using scalable library for pseudo-random number generation (sprng) [59], SSJ [60] and the variants of Rstream [26].

3.7. Choosing the right distribution technique

To sum up the suggestions stated in this section, Table I proposes another taxonomy of distribution techniques. Here, we have focused on the environment, where the simulation is supposed to run, and the underlying PRNG, which original stream has to be split. This table should help practitioners to figure out which distribution technique they can use, provided their own environment constraints.

4. RANDOM NUMBERS PARALLELIZATION SOFTWARE

4.1. Techniques designed for CPUs

In the early 1990s, at the European Simulation Symposium, Pawlikowski and his colleagues proposed an interesting methodology for the parallelization and for the automatic partitioning of stochastic parallel simulations [61, 62]. As in [63] and [18], Li and Mascagni gave advice to achieve safe multiple replications in parallel [64], but to our knowledge, the first reliable and sound library that takes into account the parallelization of pseudo-random numbers is SPRNG presented in [65]. We can find in the latter paper an overview of the mathematical grounds for random number generators, and there are also implementations of various parameterization techniques for different families of generators, which were presented in [59]. This library has been proved reliable for parallel Monte Carlo computations and also proposes a small test suite. An example of multiple replications in parallel is also given in [66] dealing with performance evaluation studies of modern multimedia telecommunication networks. In [67], we find a short and interesting discussion around Monte Carlo tools for HPC at the end of the last millennium. As stated previously in the parameterization section, Matsumoto and Nishimura proposed the Dynamic Creator software to generate mutually independent Mersenne twister generators for parallel computing [47]. This kind of approach is still a very good parallelization technique for MT generators even if we consider the recent development of efficient jump-ahead techniques for linear recurrences modulo 2 (MT and WELL generators) [54]. At the beginning of the millennium, L'Ecuyer and his team started to propose a package able to produce many long streams and substreams in C, C++, Java, and FORTRAN [26]; a version for R was proposed in 2005 [68]. In 2004, Coddington and Newell released a Java parallel random number library for HPC for the Java language [69]. This library proposes three good generators (two by L'Ecuyer), parallelized by the sequence splitting or the indexed sequence technique. To our knowledge, the library is limited to shared memory machines. In our opinion, the current best library for Java is SSJ [60]. The SSJ package provides a RandomStream Java interface that defines the basic structures to handle multiple streams of uniform random numbers. Each stream of random numbers is a Java object for which the original sequence from a given period can be cut into adjacent streams (or segments) as in the sequence splitting approach. Efficient methods are proposed to move around streams.

4.2. The case of GP-GPU

Dealing with GP-GPU, we have only found a limited number of techniques and few high quality generators [70]. The purpose here is not to list current PRNG implementations for GPUs. We have presented in [71] a survey of this kind, and in most cases, the purpose was to improve the generation speed, thanks to a GPU accelerator. If this point is interesting, it is not what

DOI: 10.1002/cpe

Table I. Summary of the potential uses of distribution techniques.

Distribution technique	Where to use it?	Where to avoid it?	Compliant families of PRNGs
Central server	Serious games	Scientific applications	Any
Sequence splitting	Limited memory environments such as GPUs	When long-range correlations in the initial RNG can lead to small-range correlations between the potential substreams.	Any jump-ahead enabled PRNG (MRG32k3a, counter-based, . . .)
Random spacing	Limited memory environments such as GPUs	When the period of the RNG is smaller than 2^{60} and when a single simulation is consuming more than 10^{11} random numbers.	Large-period PRNGs (WELL, Mersenne Twister, . . .)
Leap frog	Fixed-grained parallelism	When the leap frog is implemented with a Lagged Fibonacci generator, we have to avoid distributed environments where the number of processing elements is not a power of 2	Any
Parameterization	Distributed environments	Limited memory environments (GP-GPUs, . . .)	PRNGs issued with a parameterization algorithm (MTfamily, counter-based,. . .)

PRNGs, Pseudo-random number generators; GP-GPUs, general-purpose graphics processing units.

DOI: 10.1002/cpe

simulationists are looking for. Our main concern is to be able to massively run independent stochastic functions on GP-GPUs (named 'kernels' in the NVIDIA CUDA terminology).

At the time of writing, we find interesting propositions through libraries, mainly CURAND and Thrust::random. CURAND has been introduced in the latest version of the CUDA framework and is designed to generate random numbers in a straightforward way on CUDA-enabled GPUs. The main advantage of CURAND is that it is able to produce both quasi-random and pseudo-random sequences either on GPU or on CPU. In this library, we find a Sobol quasi-random number generator, the pseudo-random number generator implemented being XorShift proposed by Marsaglia [72]. This kind of generator is fast, but it also presents statistical flaws as explained by Panneton in his PhD thesis [73]. The interest of CURAND is that the API of the library stays the same no matter which kind of random sequence you select and the platform you run your application on. The second library found is named Thrust::random. It is part of a GPU-enabled general-purpose library called Thrust. This open-source project intends to provide a GPU-enabled library equivalent to classic general-purpose C++ libraries such as STL or Boost. Classes are split through several namespaces, which Thrust::random is an example. The latter contains all classes and methods related to random numbers generation on GP-GPU. Thrust::random implements three PRNGs, each through a different C++ template class. We find an LCG, a linear feedback shift register and a subtract with borrow [74, 75]. By the way, although the latter PRNG is mentioned as subtract with carry in Thrust::random documentation, the original Marsaglia's proposition is known as subtract with borrow. It is surprising that modern libraries propose such old generators, even if the library offers simple ways to combine them into better quality randomness sources, such as L'Ecuyer's Tausworthe combined generators [76]. In [46], we can see a smart usage of the LF technique adapted to GP-GPU for particle filtering. This paper presents a Lagged Fibonacci generator adapted to the LF approach when the number of PEs is a power of 2. An efficient GP-GPU implementation is shown and used in many computer vision algorithms.

For modern GP-GPUs (starting from the introduction of NVIDIA Tesla boards), our opinion is to consider only recent generators that have already shown statistical strength according to modern and thorough testing. To propose multiple stochastic streams through a GP-GPU architecture, we suggest the use of the DC dedicated to the new MTGP generator proposed by Saito (as a GP-GPU implementation of Mersenne Twister [77]). This careful implementation benefits from GP-GPU shared and constant memory to improve generation speed. This is not the only aspect of the generator that retained our attention: its availability for both CPUs and GPUs architectures is very important for us. Indeed, MTGP has been designed to run on GPUs, but we can also find a CPU version on Matsumoto's Home Page (Saito MTGP part). We were able to test the PRNG on CPU-based hosts with reliable and well-known tools, and this will be discussed in the next section. This PRNG can be well suited for stochastic simulations following the hybrid computing paradigm: multi-scale and holistic simulation running parts of an application on a CPU host (possible 'manycores'), while the massive parallel side is executed on GP-GPU boards. With MTGP dynamic creation, we can rely on the same parallelization technique for both the CPU and the GPU. We have started to test this GPU version of Mersenne Twister, and it will be discussed in the next section. Still, this PRNG relies on a parameterization technique to provide independent random streams. When used on GPU, this technique needs to be coupled with another one because we cannot afford to store a Parameterized Status per thread. This drawback is the main reason why a 'tiny' version of MT has recently been released. It is promising for GP-GPU because it has a small status (16 bytes). Tested successfully with the BigCrush test battery (from TestU01) described in the next section, this generator comes with its DC software and also with an efficient jump-ahead implementation. This efficient implementation is achieved, thanks to its shorter period compared with the usual huge periods of MT family generators; this period is however large enough for random spacing ($2^{127} - 1$).

Besides Mersenne Twister-like and WELL algorithms, MRG32k3a [26] from L'Ecuyer has also retained our attention for a GP-GPU use. In fact, it is designed to support natively jump ahead, making MRG32k3a a perfect match to partition a random stream following the sequence splitting strategy. In addition, these PRNG internal data structures display a small memory footprint (24 bytes) making it a definitely good candidate for GP-GPU-enabled stochastic applications. Please note that a CUDA version of this algorithm is described in [70], although a valid implementation is

DOI: 10.1002/cpe

available on our software forge.[§] The counter-based random number generators are also recently available and very efficient for GP-GPUs [55].

5. STATISTICAL AND EMPIRICAL TESTING SOFTWARE FOR RANDOM STREAMS

Knuth proposed a set of statistical tests for random streams [3]. Marsaglia designed a testing suite, named DieHard, highly regarded for many years [78]. The statistical test suite developed by the National Institute for Standards and Technology is also interesting, particularly when cryptographic qualities are required. As mentioned in the previous section, SPRNG is also providing a set of statistical tests. A detailed description of the main statistical tests for pseudo-random numbers was given in Rütti's thesis [79], and he also proposed a testing suite with some colleagues [80]. The DieHarder testing suite is proposed and updated by Brown *et al*. [81]. Although DieHarder is also an interesting testing suite, we highly recommend the TestU01 software library, which currently offers the most complete collection of utilities for the empirical statistical testing of uniform random number generators [44]. In addition to the classical statistical tests for RNGs and the other tests previously cited and proposed in the literature, TestU01 proposes new original tests as well as predefined tests suites (Crush and BigCrush with more than a hundred tests). Test of bit sequences are included, and TestU01 also considers the size of the sample according to the period length and to the kind of test considered. The TestU01 software also proposes interleaving of random streams; this is particularly interesting in the context of parallel simulation, to test the influence of parallel substreams inter-correlations. Empirical techniques to test parallel random number generators have been proposed by Coddington and Ko [82]. In 2008, we considered the testing with TestU01 (Crush) of 65536 independent sequences created by DC [47]. This was achieved using the former EGEE (known as EGI) computing grid [58]. In addition to testing software designed for sequential generators, Coddington proposed a set of criterion for 'parallel generators' [20]: the generator should be able to work on any number of processors, the sequential sequence in use for each processor should satisfy the current statistical tests, the parallel sequences should be reproducible and lastly, as stated by many authors, parallel sequences or streams should be uncorrelated. To this set of criteria, we add the fact that each PE should possess its own sequence or subsequence to ensure the reproducibility of execution on each PE independent of the hardware architecture (PEs can be: threads, processes or processors). For instance, if we detect a bug during the execution on a specific PE, we have to be able to reproduce it with the same sequence and on any architecture. The need for mutual independence between the parallel sequences or streams has to be checked carefully to avoid long-range correlations [27, 28, 41]. The problem of long-range correlations has been identified in various simulation applications for many years, and techniques have been proposed to avoid them as far as possible because to our knowledge, we cannot have rigorous proofs for this problem [27, 83–85]. De Matteis and Pagnutti have proposed interesting approaches to control correlations in [15].

We achieved an analysis of MTGP by using the dynamic creation of this PRNG and facing the resulting generators up to the current most stringent TestU01 test battery: BigCrush. Few weaknesses were identified during these experiments when compared with the original MT. The purpose of our test was to obtain a large set of fine parameterized statuses allowing the initialization of MTGP without introducing any potential bias with a bad status. Only statuses that passed BigCrush were kept in this study [71]. However, problematic statuses are mostly embarrassed by the same two tests. We observed that in some cases with relatively small periods (2^{3217}), 30% of the parameters generated by dynamic creation led to MTGPs with failure to test 35 and 100 of TestU01. Thanks to precisions given by Makoto Matsumoto and Mutsuo Saito, we now know that test 35 discards the most significant 25 bits from the 32-bit words and then uses the next 5-bits. Matsumoto and Saito were extremely reactive and have already corrected the GPU version of MTGPDC to take this into account. Nonetheless, our study has shown that MTGP was particularly safe with longer periods, according to TestU01 criteria. Yet, the longer the period, the more space it needs to store its internal state vector used by its algorithm. And this point is currently noticeable because even the best current GP-GPU architectures propose a small

[§]http://forge.clermont-universite.fr/projects/shoverand

DOI: 10.1002/cpe

size of fast shared memory. We have selected about 6000 fine MTGP parameterized statuses, with the lowest available period 2^{3217} to reduce their GPU shared memory footprint. They have been provided to Matsumoto and Saito, and they will be publicly available.

6. CONCLUSION

We have tackled the use of random number generation for HPC and presented the main partitioning methods for stochastic parallel simulations. The current 'best' generators according to the latest test libraries have been discussed. We have also considered various existing tools because the use of random numbers in parallel stochastic simulations is still a challenging technical problem. A 'good' generator is one where statistical defaults are well hidden, although they are not inexistent because even the 'best' known generators can fail for a particular application or when they are badly initialized. Parallel simulations must first consider the choice of a nice sequential generator, but as usual, no statistical or practical test is universal because we cannot prove the statistical soundness of a generator. Test batteries are the only empirical way to ensure that the generator in use fits for a wide set of applications. Such batteries can be used to independently test parallel streams. In addition, for very sensitive simulations, it can be a good practice to test different generators and partitioning techniques to increase confidence in simulation results. In the particular case of using parallel random number streams, additional care should be taken to ensure that we avoid long-range correlations, even though it often implies an additional computing cost. In our opinion, it would be interesting to include random generator partitioning and advanced random streams testing facilities in HPC middleware for the scientific end users. An attempt to do so was proposed in the PhD thesis of a student of ours with the DistMe, DistRNG and DistTest toolkits [86]. Another practical perspective could be to formalize an assessment process on the basis of the following tuple: the nature of the application, the random number generator selected, the partitioning method and the results of current test methods when available. A test method can be viewed as a rough model for a particular set of applications. Systematic testing of parallel sequences can be achieved once and for all for a particular generator, thanks to the current level of computing power available. As said previously, Reuillon has recently used the EGI Grid to achieve a test of 65536 parallel random streams for Mersenne Twister 19937 [58]. The initial statuses have been generated with the DC proposed by Matsumoto and Nishimura. In the context of GP-GPUs, we have also recently tested the dynamic creation of GPU dedicated Mersenne Twister generators (MTGP), and the results presented in [71] led to a collaboration with Matsumoto's team. The use of GP-GPUs for intensive independent stochastic hybrid computing can be limited by the current hardware constraints, such as the small size of fast shared memory area, as well as the predominant SIMD paradigm. We have recently noticed the promising 'TinyMT' that is carefully crafted for the limited GP-GPU fast shared memory size. Furthermore, the new hybrid architectures proposed are precisely working on the two aforementioned hardware limitations. In conjunction with the increase of the number of 'cores' in the near future (Kepler and Maxwell boards from NVIDIA) and the improvement of GP-GPU programming friendliness, this could change the way many scientists will consider the use of 'desktop' HPC.

REFERENCES

1. Gentle JE. Random Number Generation and Monte Carlo Methods, 2nd Edition. Springer Verlag: New York, 2003.
2. James F. A review of pseudorandom number generators. *Computer Physics Communications* 1990; **60**:329–344.
3. Knuth DE. The Art of Computer Programming, Vol. 2, 2nd edition. Seminumerical Algorithms. Addison-Wesley: Reading, MA, 1981.
4. Lagarias JC. Pseudorandom numbers. *Statistical Science* 1993; **8**:31–39.
5. Mascagni M. Random number generation. CRC Standard Mathematical Tables and Formulae, 31st Edition, Zwillinger D (ed.). Chapman and Hall/CRC: Boca Raton, 2003; 644–649.
6. L'Ecuyer P. Pseudorandom number generators. In Encyclopedia of Quantitative Finance, Rama Cont, Ed., in volume Simulation Methods in Financial Engineering, Platen E, Jaeckel P (eds.). Wiley: Chichester, UK, 2010.
7. El Bitar Z, Lazaro D, Breton V, Hill DRC, Buvat I. Fully 3D Monte Carlo image reconstruction in SPECT using functional regions. *Nuclear Instruments and Methods in Physics Research* 2006; **569**:399–403.

DOI: 10.1002/cpe

8. Lazaro D, El Bitar Z, Breton V, Hill DRC, Buvat I. Fully 3D Monte Carlo reconstruction in SPECT: a feasibility study. *Physics in Medicine and Biology* 2005; **50**:3739–3754.
9. Maigne L, Hill DRC, Calvat P, Breton V, Reuillon R, Lazaro D, Legré Y, Donnarieix D. Parallelization of Monte Carlo simulations and submission to a grid environment. *Parallel Processing Letter* 2004; **14**:177–196.
10. Reuillon R, Hill DRC, El Bitar Z, Breton V. Rigorous distribution of stochastic simulations using the DistMe toolkit. *IEEE Transactions on Nuclear Science* 2008; **55**(1):595–603.
11. Niederreiter H. Random Number Generation and Quasi Monte Carlo Methods. SIAM: Philadelphia, 1992.
12. Stevanovic R, Topic G, Skala K, Stipcevic M, Rogina BM. Quantum Random Bit Generator Service for Monte Carlo and Other Stochastic Simulations, Lirkov I, Margenov S, Wasniewski J (eds.). Lecture Notes in Computer Science. Springer-Verlag: Berlin Heidelberg, 2008; **4818**:508–515.
13. Hill DRC. URNG: a portable optimization technique for software application requiring pseudorandom numbers. *Simulation Modelling Practice and Theory* 2002; **11**:643–654.
14. Kleijnen JPC. Statistical Tools for Simulation Practitioners. Dekker: New-York, 1987.
15. De Matteis A, Pagnutti S. Controlling correlations in parallel Monte Carlo. *Parallel Computing* 1995; 73–84.
16. Hellekalek P. Don't trust parallel Monte Carlo. *Proceedings of the twelfth workshop on Parallel and distributed simulation*, Alberta, Canada, 1998; 82–89.
17. Pawlikowski K. Do not trust all simulation studies of telecommunication networks. *Proceeding of International Conference on Information Networking*, *ICOIN'03*, Jeju Island, Korea, 2003; 899–908.
18. Pawlikowski K. Towards credible and fast quantitative stochastic simulation. *Proceedings of International SCS Conference on Design*, *Analysis and Simulation of Distributed Systems*, *DASD'03*, 2003.
19. Fujimoto RM. Parallel and Distributed Simulation Systems. Wiley Interscience: New-York, January, 2000.
20. Coddington PD. Random Number Generators for Parallel Computers. NHSE Review, 2nd Issue, 1996.
21. Hellekalek P. Good random number generators are (not so) easy to find. *Mathematics and Computers in Simulation* 1998; 485–505.
22. L'Ecuyer P. Software for uniform random number generation: distinguishing the good and the bad. Proceedings of the 2001 Winter Simulation Conference. IEEE Press: Pistacaway NJ, 2001; 95–105.
23. Matsumoto M, Nishimura T. Mersenne Twister: a 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation* 1998; **8**:3–30.
24. Saito M, Matsumoto M. SIMD-oriented Fast Mersenne Twister: a 128-bit pseudorandom number generator. Proceedings of Monte Carlo and Quasi-Monte Carlo Methods 2006. Springer, 2008; 607–622.
25. Panneton F, L'Ecuyer P, Matsumoto M. Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software* 2006; **32**(1):1–16.
26. L'Ecuyer P, Simard R, Chen EJ, Kelton WD. An object-oriented random-number package with many long streams and substreams. *Operations Research* 2002; **50**:1073–1075.
27. De Matteis A, Pagnutti S. Parallelization of random number generators and long-range correlations. *Numerische Mathematik* 1988; **53**:595–608.
28. De Matteis A, Pagnutti S. A class of parallel random number generators. *Parallel Computing* 1990; **13**(2):193–198.
29. Durst MJ. Using linear congruential generators for parallel random number generation. Proceedings of the Winter Simulation Conference, MacNair EA, Musselman KJ, Heidelberger P (eds.). ACM: New-York, 1989; 462–466.
30. Eddy WF. Random number generators for parallel processors. *Journal of Computational and Applied Mathematics* 1990; **31**:63–71.
31. Entacher K, Uhl A, Wegenkittl S. Linear congruential generators for parallel Monte Carlo: the leap-frog case. *Monte-Carlo Methods and Applications* 1998; **4**(1):1–16.
32. Percus OE, Kalos MH. Random number generators for MIMD parallel processors. *Journal of Parallel and Distributed Computing* 1999; **6**:477–497.
33. Srinivasan A. Introduction to parallel RNG, 1998. http://sprng.cs.fsu.edu/Version1.0/paper/index.html [2012].
34. Hechenleitner B, Entacher K. On shortcomings of the NS-2 random number generator. In Communication Networks and Distributed Systems Modeling and Simulation (CNDS), Znati T, McDonald B (eds.). SCS, 2002; 71–77.
35. Pawlikowski K, Jeong HDJ, Lee JSR. On credibility of simulation studies of telecommunication networks. *IEEE Communications Magazine* 2002; **40**(1):132–139.
36. Entacher K, Hechenleitner B. Pitfalls when using parallel streams in OMNET++ simulations. *Proceedings of Inter-domain Performance and Simulation* (*IPS*) *Workshop*, Salzburg, Austria, 2003; 11–20.
37. Traore M, Hill DRC. The use of random number generation for stochastic distributed simulation: application to ecological modeling. *Proceedings of the 13th European Simulation Symposium*, 1991; 555–559.
38. Bauke H, Mertens S. Random numbers for large scale distributed Monte Carlo simulations. *Physical Review E* 2007; **75**(6):701–714.
39. Mascagni M, Srinivasan A. Parameterizing parallel multiplicative lagged-Fibonacci generators. *Parallel Computing* 2004; **30**:899–916.
40. Hechenleitner B. Defects in random number routines of well-known network simulators and appropriate improvements. *PhD Thesis*, School of Scientific Computing of the University of Salzburg, 2004.
41. De Matteis A, Pagnutti S. Long-range correlations in linear and non-linear random number generators. *Parallel Computing* 1990; **14**(2):207–210.
42. Srinivasan A, Mascagni M, Ceperley DM. Testing parallel random number generators. *Parallel Computing* 2003; **29**(1):69–94.

DOI: 10.1002/cpe

43. Wu P, Huang K. Parallel use of multiplicative congruential random number generators. *Computer Physics Communications* 2006; **175**(1):25–29.
44. L'Ecuyer P, Simard R. TestU01: a C library for empirical testing of random number generator. *ACM Transactions on Mathematical Software* 2007; **33**(4):1–40.
45. Aluru S. Lagged Fibonacci random number generators for distributed memory parallel computers. *Journal of Parallel and Distributed Computing* 1997; **45**(1):1–12.
46. Janowczyk A, Chandran S, Aluru S. Fast, processor-cardinality agnostic PRNG with a tracking application. Computer Vision, Graphics & Image Processing, 2008. ICVGIP '08, December 16–19; 2008; 171–178.
47. Matsumoto M, Nishimura T. Dynamic creation of pseudorandom number generators. Monte Carlo and Quasi-Monte Carlo Methods conference 1998. Springer, 2000; 56–69.
48. Mascagni M, Chi H. Parallel linear congruential generators with Sophie-Germain moduli. *Parallel Computing* 2004; **30**(11):1217–1231.
49. Frederickson P, Hiromoto R, Jordan TL, Smith B, Warnock T. Pseudo-random trees in Monte Carlo. *Parallel Computing* 1984; **1**(2):175–180.
50. Sipper M. Generating parallel random number generators by cellular programming. *International Journal of Modern Physics C* 1996; **7**(2):181–190.
51. Tomassini M, Sipper M, Zolla M, Perrenoud M. Generating high-quality random numbers in parallel by cellular automata. *Future Generation Computer Systems* 1999; **16**(2–3):291–305.
52. Tomassini M. Parallel and distributed evolutionary algorithms: a review. Evolutionary Algorithms in Engineering and Computer Science—Recent Advances in Genetic Algorithms, Evolution Strategies, Evolutionary Programming, Genetic Programming and Industrial Applications, Miettinen K, Mäkelä MM, Neittaanmäki P, Périaux J (eds). John Wiley & Sons, 1999; 113–133.
53. Ackermann J. Parallel random number generator for inexpensive configurable hardware cells. *Computer Physics Communications* 2001; **140**(3):293–302.
54. Haramoto H, Matsumoto M, Nishimura T, Panneton F, L'Ecuyer P. Efficient jump ahead for F_2-linear random number generators. *INFORMS Journal on Computing* 2008; **20**(3):385–390.
55. Salmon JK, Moraes MA, Dror RO, Shaw DE. Parallel random numbers: as easy as 1, 2, 3. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SuperComputing 2011 – SC11). ACM: New York, NY, 2011; 1–12.
56. Reuillon R, Traore MK, Passerat-Palmbach J, Hill DRC. Parallel stochastic simulations with rigorous distribution of pseudo-random numbers with DistMe: application to life science simulations. *Concurrency and Computation: Practice and Experience* 2011, accepted 31 August 2011, published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1883, in press, 2011.
57. Reuillon R, Chuffart F, Leclaire M, Faure T, Dumoulin N, Hill DRC. Declarative task delegation in OpenMOLE. Proceedings of the 2010 International Conference on High Performance Computing and Simulation, Smari WW (ed.). 2010; 55-62.
58. Reuillon R. Testing 65536 parallel pseudo-random number streams. EGEE Grid User Forum 2008, Poster session, Clermont-Ferrand, February 11-14th, Abstract 1 p., 2008.
59. Mascagni M. Some methods of parallel pseudorandom number generation. Algorithms for Parallel Processing, Schreiber R, Heath M, Ranade A (eds). Springer Verlag: New York, Berlin, 1997; 277–288.
60. L'Ecuyer P, Buist E. Simulation in Java with SSJ. *Proceedings of the 2005 Winter Simulation Conference*, 2005; 611–620.
61. Pawlikowski K, McNickle D. Speeding up stochastic discrete-event simulation. *Proceedings of European Simulation Symposium*, *ESS'01*, Marseille, France, Oct. 18–20, 2001; 132–138.
62. Pawlikowski K, Yau V. On automatic partitioning, run-time control and output analysis methodology for massively parallel simulations. *Proceedings of the European Simulation Symposium*, *ESS'92*, 1992; 135–139.
63. Hill DRC. Object-oriented pattern for distributed simulation of large scale ecosystems. *Proceedings of the SCS Summer Computer Simulation Conference*, 1997; 945–950.
64. Li Y, Mascagni M. Improving performance via computational replication on a large computational grid. In 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003; 442–448.
65. Mascagni M, Ceperley D, Srinivasan A. SPRNG: a scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software* 2000; **26**:618–619.
66. Ewing G, Pawlikowski K, McNickle D. Akaroa-2: exploiting network computing by distributed stochastic simulation. *13th European Simulation Multiconference*, Warsaw, Poland, SCSC, June 1999; 175–181.
67. Mascagni M. High-performance Monte Carlo tools. *IEEE Computational Science and Engineering* 1998; **5**(2):97–98.
68. L'Ecuyer P, Leydold J. Rstream: streams of random numbers for stochastic simulation. *The newsletter of the R project* 2005; **5**:16–19.
69. Coddington PD, Newell AJ. JAPARA—a Java parallel random number library for high-performance computing. *Proceedings of the 18th International Parallel and Distributed Processing Symposium* (*IPDPS'04*) *- Workshop 5*, 2004; 156–166.
70. Bradley T, du Toit J, Tong R, Giles M, Woodhams P. Parallelization techniques for random numbers generators, Chapter 16. GPU Computing Gems, Corporation N, Hwu WW (eds.), Emerald Edition. Elsevier, 2011; 231–246.
71. Passerat-Palmbach J, Mazel C, Mahul A, Hill DRC. Reliable initialization of GPU-enabled parallel stochastic simulations using Mersenne Twister for graphics processors. ESM 2010, European Simulation, October 25–27, Hasselt University, Belgium, 2010; 187–195.

DOI: 10.1002/cpe

72. Marsaglia G. Xorshift RNGs. *Journal of Statistical Software* 2003; **8**(14):1–6.
73. Panneton F. Construction d'ensembles de points basée sur des récurrences linéaires dans un corps fini de caractéristique 2 pour la simulation Monte Carlo et l'intégration quasi-Monte Carlo. Thèse de l'Université de Montréal, Département d'Informatique et de Recherche Opérationnelle, 2004.
74. Marsaglia G, Zaman A. A new class of random number generators. *The Annals of Applied Probability* 1991; **3**(3):462–480.
75. Marsaglia G, Narasimhan B, Zaman A. A random number generator for PC's. *Computer Physics Communications* 1990; **60**(3):345–349.
76. L'Ecuyer P. Maximally equidistributed combined Tausworthe generators. *Mathematics of Computation* 1996; **65**(213):203–213.
77. Saito M. A variant of Mersenne Twister suitable for graphics processors. Available online, submitted 2010. http://arxiv.org/abs/1005.4973 [September 2012].
78. Marsaglia G. DIEHARD: a battery of tests of randomness, 1996. http://www.stat.fsu.edu/pub/diehard/ [September 2012].
79. Rütti M. A random number generator test suite for the C++ standard. Diploma Thesis, 2004.
80. Rütti M, Troyer M, Petersen WP. A generic random number generator test suite, 2004. http://arxiv.org/abs/math/0410385 [September 2012].
81. Brown RG, Eddelbuettel D, Bauer D. Dieharder: a random number test suite. Version 3.29.5b. Available at: http://www.phy.duke.edu/~rgb/General/dieharder.php [September 2012].
82. Coddington PD, Ko SH. Techniques for empirical testing of parallel random number generators. Proceedings of the 12th international Conference on Supercomputing, ICS '98, Melbourne, Australia. ACM Press: New York, 1998; 282–288.
83. De Matteis A, Eichenauer-Herrmann J, Grothe H. Computation of critical distances within multiplicative congruential pseudorandom number sequences. *Journal of Computational and Applied Mathematics* 1992; **39**(1):49–55.
84. Eichenauer-Herrmann J, Grothe H. A remark on long-range correlations in multiplicative congruential pseudo random number generators. *Numerische Mathematik* 1989; **56**(6):609–611.
85. Entacher K, Uhl A, Wegenkittl S. Parallel Random Number Generation: Long-Range Correlations Among Multiple Processors. Lecture Notes in Computer Science 1999; 107–116.
86. Reuillon R. Simulations stochastiques en environnements distribués, application aux grilles de calcul. *Ph.D. Thesis*, Blaise Pascal University, 2008.

DOI: 10.1002/cpe