Adv. Radio Sci., 12, 75–81, 2014
www.adv-radio-sci.net/12/75/2014/
doi:10.5194/ars-12-75-2014

Advances in Radio Science
Open Access Proceedings

On parallel random number generation for accelerating simulations of communication systems

C. Brugger, S. Weithoffer, C. de Schryver, U. Wasenmüller, and N. Wehn

Microelectronic Systems Design Research Group, University of Kaiserslautern, 67663 Kaiserslautern, Germany

Correspondence to: C. Brugger (brugger@eit.uni-kl.de)

Received: 20 December 2013 – Accepted: 5 February 2014 – Published: 10 November 2014

Abstract. Powerful compute clusters and multi-core systems have become widely available in research and industry nowadays. This boost in utilizable computational power tempts people to run compute-intensive tasks on those clusters, either for speed or accuracy reasons. Especially Monte Carlo simulations with their inherent parallelism promise very high speedups. Nevertheless, the quality of Monte Carlo simulations strongly depends on the quality of the employed random numbers. In this work we present a comprehensive analysis of state-of-the-art pseudo random number generators like the MT19937 or the WELL generator used for parallel stream generation in different settings. These random number generators can be realized in hardware as well as in software and help to accelerate the analysis (or simulation) of communications systems. We show that it is possible to generate high-quality parallel random number streams with both generators, as long as some configuration constraints are met. We furthermore depict that distributed simulations with those generator types are viable even to very high degrees of parallelism.

1 Introduction

The ongoing research activities and technological progress over the last years have led to a tremendous increase in the complexity of current systems and scientific challenges. More and more realistic and highly sophisticated models require numerical computations, since analytical solutions are not available in many cases. For example in the analysis of communications systems, independent random numbers are required in a lot of places, e.g. for the generation of payload data, additive white gaussian channel noise, or the modeling of fading channels. Furthermore, we can mention modeling carrier offsets and timing offsets, and the multiple channels between transmit and receive antennas in multiple-input and multiple-output (MIMO) systems.

Monte Carlo (MC) simulations remain a widely employed universal method to solve problems when other approaches are not applicable (Korn et al., 2010). It is important to note that the result quality of a MC simulation strongly depends on the quality of the RNs used. When only one random number generator (RNG) is used for the complete simulation (for example for distributing one random number (RN) stream to only one or a smaller number of processing elements (PEs)), it is sufficient to instantiate a single generator with the desired quality properties.

Nevertheless, in order to obtain the results fast and with the desired accuracy, parallel simulations are mandatory. MC simulations are perfect candidates for parallel simulation, since they rely on a large number of independent experiments that can be carried out simultaneously. However, the generation of high-quality independent and identically distributed (i.i.d.) RNs is a big challenge. With huge computing clusters and the *Cloud* becoming accessible for more and more researchers and professionals, it is time to have a closer look on possible pitfalls and quality issues arising from highly parallel simulations.

Various approaches for creating i.i.d. parallel RN streams have been proposed up to now that we study in detail in this paper. We give a short summary about available generation methods for parallel RN streams and how they can be tested, together with an overview about related work. For the famous MT19937 Mersenne Twister (MT) and the well-equidistributed long-period linear (WELL) generator proposed by Pierre L'Ecuyer, we show that randomly seeding parallel RNGs is safe, also for very high degrees of parallelism. We have carried out a comprehensive analysis of

possible inter-stream correlations and substreams in the popular MT19937 pseudo random number generator (PRNG) using the TestU01 test suite, a popular test framework for RN quality analysis. We show that for a well-defined seeding schemes no peculiarities are observed. In summary we demonstrate that the use of standard state-of-the-art PRNGs like the MT and the WELL is safe, also for parallel simulations.

2 Background

The generation of high-quality RNs in high volumes has long been a hot topic in research. In general, three types of RNs exist:

- *True RNs*,
- *quasi RNs*, and
- *pseudo RNs*.

At a first glance, *true RNs* seem to be most desirable, since they are really random and can not predicted in any way. However, true RNs come with some drawbacks: Firstly, their generation typically relies on a physical process and therefore needs a special device that is in general not available in generic computing systems like standard infrastructure as a service (IaaS) or platform as a service (PaaS) clusters or data centers. Secondly, it is quite likely that the untreated output of a true RNG is biased. Therefore the output stream needs to be monitored continuously, and a bias correction is needed that is adjusted continuously (Hill et al., 2013). Thirdly, true RNs are not repeatable, and forbid re-executing a simulation several times with generating exactly the same output – a knock-out criterion for science in general.

Quasi RNs are not random at all, but try to cover a (multi dimensional) space with certain characteristics. Prominent examples are Sobol or Halton sequences used for MC simulations (Korn et al., 2010). We do not consider true and quasi RNs further in this work, details can be found in literature (Knuth, 1997; Korn et al., 2010).

For the main reason of keeping simulations and calculations reproducible, PRNGs are the most favorable choice for simulation purposes. They try to generate a sequence looking like "good" RNs by a deterministic algorithm, this means i.i.d. from a statistical viewpoint (L'Ecuyer, 2007). PRNGs have a number of advantages: they can be run on different execution platforms, require no additional hardware support, and produce exactly the same output streams when seeded with equal values. In the following we will therefore focus on PRNGs only.

2.1 Generating parallel random numbers

A comprehensive summary about generation methods for parallel RN streams has been given recently by Hill et al. (2013).

This section will give a brief overview and comments on some of the methods for generating multiple independent RN streams. We focus on approaches that can be realized in hardware by either utilizing multiple RNGs or a fast, parallelized RNG and use WELL and MTs generators as examples. Parallel RN streams can be generated from:

- Multiple identical RNGs starting from different positions in the period,
- multiple RNGs of different type,
- a single parallelized RNG, or
- combinations of the above.

Implications of different approaches on the quality of statistics are discussed in detail in Sects. 4.1 to 4.3.

2.1.1 Multiple generators of the same type

Using identical RNGs and starting them from a different position in the generator period corresponds to partitioning a single RN stream into blocks (*Blocking*) and is suitable when the number of consumed RNs for each of the parallel streams is known and bounded. Multiple instances of one RNG type can then be initialized to a precomputed state. This so-called *jump-ahead* technique (Haramoto et al., 2008) allows for defined distances between the streams of the RNGs (w.r.t. positions in the RNG period). The initialization of the multiple generators can also be done by employing other RNGs. WELL and MT generators, for example, are commonly initialized (seeded) by other small RNGs (seed generators). This approach is called *Random Spacing*.

As these also have to be seeded, we realize a random seeding of the WELL and MT by seeding the seed generators differently.

2.1.2 Multiple generators of different type

The straight-forward way of getting RNGs of different type is to use a different algorithm for each of the RNGs. Because of the limited number of known algorithms with each having varying statistical properties and implementation complexities, such an approach poses strict limits on the number and quality of RNs generated in parallel for obvious reasons and is not considered here. The WELL and MT generators, however, allow the use of the same algorithm with different parameter sets that correspond to different underlying recurrence polynomials. Correlations between the RN streams generated by following this approach are considered highly unlikely. As the parameter sets for the WELL and MT generator types have to be found by extensive search, parallelism by using this so-called *Parametrization* is also limited (Tian and Benkrid, 2009).

2.1.3 Single, parallelized generator

Generating parallel streams of RNs by using a single parallelized RNG corresponds to parsing a single RN stream into substreams. If the RNs from a single stream are distributed according to a round-robin scheme (i.e. the first RN to the first substream, the second RN to the second substream, ...), the resulting partitioning is called *Leap Frog* partitioning (see Hill et al., 2013).

2.2 Quality issues

As stated before, PRNGs do not produce real RNs, but try to create a "good" output streams that looks random. Evaluating the quality of PRNGs has been an active topic for a long time. To a special degree, this research was driven by Pierre L'Ecuyer, Michael Mascagni, and others who investigated novel RNG approaches. For example, L'Ecuyer has formulated several key criteria for "good" RNGs: a long period, repeatable outputs, portability to different execution platforms, and the ability to split the output sequence into multiple independent blocks (what means they should implement an efficient jump-ahead strategy) and each block again into substreams with the leap frog approach (L'Ecuyer, 2007; L'Ecuyer and Panneton, 2005). D. E. Knuth has summarized the basic test strategies for single-stream RNGs in his famous book "The Art of Computer Programming" (Knuth, 1997), with the first edition already released in 1969. Nowadays, the TestU01 suite developed by L'Ecuyer in 2007 (L'Ecuyer and Simard, 2007) is considered the most comprehensive one and is preferred by many researchers (Salmon et al., 2011; Hill et al., 2013).

A first requirement for a good parallel PRNG is that it also has to be a good sequential PRNG (Srinivasan et al., 2003). Obviously, when testing different substreams of one PRNG, it is mandatory to ensure that exactly the same substream can be outputted again to allow debugging and reproducibility of the test results (Hill et al., 2013).

In 2003, Srinivasan et al. have stressed the importance of test concepts for parallel RNGs, in particular for high-performance computing (HPC) applications (Srinivasan et al., 2003). They have introduced the terms *intra-stream* and *inter-stream correlations* to highlight that both characteristics are crucial for a "good" parallel RNG: the quality of each substream itself, and the independence of all substreams from each other. They also presented first test strategies for parallel RNs.

In order to test inter-stream correlations, Salmon et al. (2011) have concatenated blocks of multiple substreams with a round-robin scheme to a single test stream that they have fed into the TestU01 suite (Salmon et al., 2011). Due to the correlation tests included in TestU01, correlations between the substreams coming from different generators are detected. We also follow this approach in our work, but show that there may be problems when not carefully selecting the right assembly configuration (see Sect. 4.3).

Table 1. Ressource consumption of implementations for FPGAs.

RNG	BRAMs	Slices	max. Freq. [MHz]	Throughput [Gbps]
MT19937[1]	1	57	100.0	3.20
TinyMT[1]	0	65	100.0	3.20
MT19937[2]	2	330	24.2	0.77
3-CP MT[3]	2	207	258.3	24.03
4-IP MT[3]	4	290	277.7	35.54
8-IP MT[3]	8	566	283.5	72.58

[1] On a MaxWorkstation (Virtex-6) from Maxeler Technologies Inc.
[2] (Chandrasekaran and Amira, 2008) on a Virtex-E.
[3] (Dalal and Stefan, 2008) on a Virtex-4.

3 Related work

A hardware implementation of the MT algorithm has, for example, been reported by (Chandrasekaran and Amira, 2008).

Parallel RN generation by multiple MT generator cores with different parameter sets was described by (Tian and Benkrid, 2009). They reported their design to outperform CPU and GPU implementations of the MT algorithm by a factor of 25× and 9× respectively w.r.t. throughput. An investigation of the statistical properties of the generated numbers, however, was not performed.

Dalal and Stefan presented *Interleaved Parallelization* (IP) and *Chunked Parallelization* (CP), two methodologies for parallelizing RNGs of MT and WELL type (Dalal and Stefan, 2008). There, the state vector of the RNG is partitioned in a way that multiple numbers can be generated in parallel. A corner case of this scheme, a fully parallel implementation of the MT, was reported by Sriram and Kearney (Sriram and Kearney, 2009). Both parallelization schemes result in a *Leap Frog* partitioning at the outputs of the parallelized RNGs. However, they are only practical for grades of parallelism that are a divider of the state vector size (that is 624 for the MT19337).

Table 1 lists the resource consumption of selected MT implementations on field programmable gate array (FPGA). Even if the different target FPGAs are taken into account it becomes clear that IP and CP are more hardware efficient than using multiple instances of the respective serial generator. We have been able to confirm these numbers in our own investigations. While BRAMs can become a critical resource in highly parallel RNG architectures we also considered the TinyMT with an internal state of 127 bits, far smaller than the MT with 19937 bits. The TinyMT only uses registers for the state storing, but is characterized by a smaller period.

L'Ecuyer and Panneton have compared throughput and jump-ahead computation times for LFSRs, MT, and the

Table 2. Upper bound probability of a collision for random number generators running at 5 GHz for 100 years.

RNG	2 generators	Chip with 1000 RNGs	System of 1000 of such chips	System of one million chips
TinyMT	$< 10^{-18}$	$< 10^{-13}$	$< 10^{-7}$	$< 10^{-1}$
MT19937	$< 10^{-5982}$	$< 10^{-5976}$	$< 10^{-5970}$	$< 10^{-5964}$
WELL44497b	$< 10^{-13375}$	$< 10^{-13369}$	$< 10^{-13363}$	$< 10^{-13357}$

WELL generator in 2005 (L'Ecuyer and Panneton, 2005). They have shown that the jump-ahead computation time exponentially increases with the number of internal states in the PRNG.

Hill et al. (2013) have carried out an analysis on various Mersenne Twister for Graphic Processors (MTGP) configurations with the TestU01 Big Crush battery in 2012 and have found only a few weaknesses (Hill et al., 2013). They concluded that the use of the MTGP with longer periods therefore is safe.

At SC11, Salom et al. (2011) have presented an evaluation of so-called *counter-based PRNGs*. They are highly scalable, can be implemented on central processing unit (CPU), graphics processor unit (GPU), and FPGA based architectures, and therefore seem to be very promising for distributed HPC simulations. The presented PRNGs pass the Big Crust test battery of the TestU01 suite without failures and even beat the MT19337. At the same time, the counter-based generators can achieve around the same throughput as the MT19337 in software. However, counter-based PRNGs have not been considered for hardware implementation up to now, this is ongoing research due to their higher implementation complexity.

4 New investigations

We have seen that a lot of approaches have been followed to generate and test parallel RNs. However, we had difficulties to evaluate the scalability and efficiency of these methods for highly parallel settings. For example, the leap frog approach does not scale above the dimensionality of the PRNG, i.e. 627 for the MT19937 (Hill et al., 2013). Splitting the RN stream into blocks in advance with the jump-ahead algorithm is critical if it cannot be ensured that the amount of consumed numbers never exceeds the defined block sizes. Furthermore, also the jump-ahead computations are quite compute intensive and do not scale very well (Hill et al., 2013). Instantiating multiple generators with different parameters sounds promising, but also in this case the available number of configuration is limited (Tian and Benkrid, 2009).

Therefore we have investigated partitioning schemes that aim at high degrees of parallel RN streams, in particular random seeding approaches with standard generators. The results are given in the following sections.

4.1 Mersenne twister substream analysis

Since the MT19937 is one of the most popular PRNGs and implemented in many available libraries and software tools, we have analyzed the quality of MT19937 substreams generated with the leap frog method. We have used the MT19937 C implementation as provided by Matsumoto and Nishimura, refined by Richard Wagner in 2009[1]. The seed was fixed to 0x00001571.

In total we have run 88 test on a selection of substreams from up to 64 total substreams. Due to the high runtimes even on the employed compute cluster especially for higher numbers of substreams, we were not able to check all configurations. In 58 of 88 cases, only tests number 80 and 81 (that are always failed by the MT19937) have failed. In 27 cases one additional test was suspicious, and only in 3 cases two additional tests showed suspicious outputs. In summary, the obtained results have shown no suspicious values at all. Therefore we conclude that it is safe to split standard MT19937 streams into up to 64 substreams, what should be sufficient for supplying up to 64 PEs with one generator instance.

4.2 Investigating seed collisions

One method of generating independent random number generators is to use multiple instances of the same generator with different seeds. Each obtained output stream is then a subsequence of one long period sequence, while the starting point depends on the seed. For random streams, the distance between two streams is unknown and they might even overlap. Overlapping streams are highly correlated and have to be avoided. Before considering this, let us investigate the probability it is that two substreams overlap.

It is helpful to visualize the period as a circle in which each instance of a random number generator draws consecutive numbers on this circle, see Fig. 1. When two seeds are too close together on this circle, the streams of random numbers overlap and we have a stream collision. In the following setup we consider n independent RNGs with a period length of L. For each RNG we draw l consecutive RNs. Let us consider two RNGs. The probability for the second one to not collide

[1] http://www.comp.nus.edu.sg/~noi/tasks/2010/PKMATCH/MersenneTwister.h, last access: 5 February 2014.

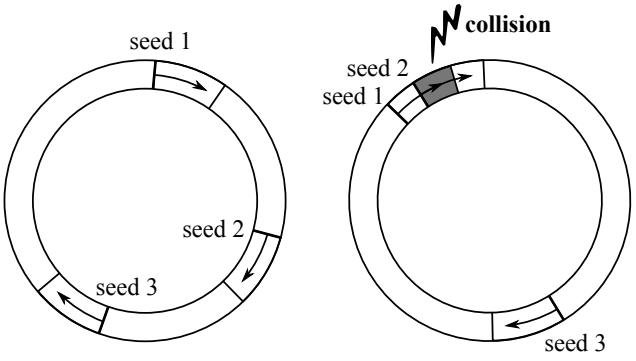

Figure 1. The circles represent the whole period of the random number generator. We consider three instances of the same random number generator. Each generator is seeded with a random number and a specific amount of random numbers are drawn from it. In the left case the streams do not overlap, while on the right side the first random number generator collides with the second.

with the first one is:

$$P_2 = \frac{L - 2l + 1}{L}. \tag{1}$$

When we add one more, the probability of this third generator not colliding with the first two depends on their actual position. The probability is the least when both are far apart. In general it is:

$$P_3 \geq P_2 \frac{L - 2(2l-1)}{L}. \tag{2}$$

We can extend this series and get an upper bound estimate for the collision probability $\hat{P}_n$:

$$\hat{P}_n = 1 - P_n \leq 1 - \prod_{i=1}^{n} \frac{L - (i-1)(2l-1)}{L}. \tag{3}$$

We assume that our random number generators run with 5 GHz, generating one RN per clock cycle, running for 100 years. This results in an l of about 10^{18}. For the MT the state vector has 19 937 bits resulting in a period of about 10^{6002}. We consider chips with 1000 of such generators. We are interested in the collision rate for such chips and huge system containing up to one millions of them. To calculate the fractions containing such large numbers we have used the *fractions* package in Python. To reduce the number of multiplications we estimate the collision rate with an even higher upper bound as:

$$\hat{P}_n \leq 1 - \left(\frac{L - (n-1)(2l-1)}{L} \right)^{n-1}. \tag{4}$$

This power can be efficiently computed with only $\log_2(n-1)$ multiplications instead of $n-1$ required for Eq. (3). For $n = 10^6$ these are 20 instead of 999 999 multiplications.

Table 3. Test failing for our robust parallel seeding procedure.

number of RNGs	failing tests for block size of 1	4	8	12	30	1 k	10 k	100 k	1 M
1	2	2	2	2	2	2	2	2	2
2	2	2	–	–	–	–	–	2	2
4	2	–	–	–	–	–	–	2	2
8	2	–	–	–	–	–	–	2	2
16	–	–	–	–	–	–	–	2	2
64	–	–	–	–	–	–	–	2	2
256	–	–	–	–	–	–	–	2	2
1024	–	–	–	–	–	–	–	2	2

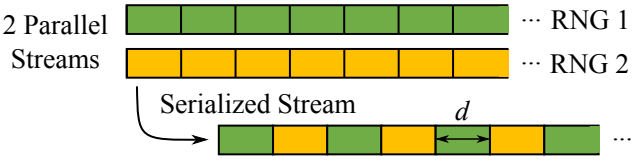

Figure 2. Showing how parallel streams of two independent RNGs can be serialized when run on a serial processor. In the shown case the parallel program operates on equal blocks d of random numbers.

The resulting probabilities for various RNGs are shown in Table 2. For RNGs with reasonably large state vectors the probability of a seed collision is below 10^{-5964} even for incredibly large systems containing one billion RNGs running at 5 GHz for 100 years. We can conclude that when the seed is chosen randomly a collision is negligibly unlikely for the MT19937 and WELL44497b. However, for the TinyMT more advanced parallelization methods should be utilized, like e.g. parameterization.

4.3 Inter-stream correlations

In the last section we have seen that stream collisions are highly unlikely for different seeds. Based on this insight it seems reasonable to realize parallel random number generation by using independent RNGs each with a random seeds. However then question remains what the quality of these streams is. The generator might for example show long-term correlations resulting in inter-stream correlations, as discussed in Sect. 2.2.

We now derive a methodology on how to analyze inter-stream correlations. For this it is helpful to keep the application in mind that uses the parallel RNG. In general it is possible to execute a parallel program on a serial processor for which only one instruction is executed at a specific time. That means a single stream of random numbers exists that is equal to how the parallel RNGs are accessed during execution. Thus we can analyze parallel RNGs also with the standard test suites for single streams by using this serialized stream.

While in general the serialized stream is an arbitrarily interleaved stream from the parallel streams, it is helpful to

Table 4. Test failing with Tausworthe seeding.

RNG count:	1	2	4	8	16	32	64	128
MT19937	2	2	2	41	64	79	100	106
WELL44497b	2	2	2	102	88*	89*	89*	

* Tested with smaller test-suite Crush instead of Big Crush.

consider some representative cases. We consider that the program operates on blocks of random numbers of the same size d. This is clearly the case for Monte Carlo simulations, used very widely in communication system simulations. Then we can serialize the parallel streams as shown in Fig. 2.

4.4 Random seeding

Random seeding requires the use of a RNG itself to generate numbers of all the seed values. In Sect. 4.2 the probabilities for seed collisions are only valid when all starting points are equally likely. This means the seeding procedure has to be carefully validated, such that it does not favor specific starting points.

For our robust seeding procedure, we use the seeding algorithm integrated in the MT19937 C implementation (version 2002) and seed the RNG n with $\text{seed}_0 + n$. We have tested the serialized stream for eight independent seed_0 and calculated the mean of failing tests. All investigations have been made with the Big Crush battery from the TestU01 suite v1.2.3. The result is shown in Table 3.

For one generator already two test are failing, for which the MT is known for. However, for increasing numbers of RNGs and block sizes, the serialized stream gets even better. For block sizes of 100 k and higher the same TestU01 results are as for single MT stream. This is due to the limited analysis window of the TestU01 suite as sample sizes of the tests that have been conducted n have been in the order of $10^9 \geq n \geq 10^6$.

As a counterexample we have implemented a naive seeding procedure based on the Tausworthe 88 RNG. It seeds the complete state vector n of the MT with the first values from a Tausworthe RNG. This Tausworthe 88 is itself seeded with the vector [0x88cb47c9 + 2 ($\text{seed}_0 + n$), 0x8a9cdf65 + 4 ($\text{seed}_0 + n$), 0xcaf40ed9 + 8 ($\text{seed}_0 + n$)]. In this case only a block size d of one has been considered for eight independent seed_0. The number of failing test are shown in Table 4. Starting from eight RNGs the number of failing tests dramatically increases, rendering this naive procedure useless. This stresses the importance of the seeding method.

For the robust seeding procedure we can conclude that no inter-stream correlations are observable. Further the use of independent MT generators even improves their quality.

5 Conclusion

The recent increase in available computation power on highly parallel computing clusters or the *Cloud* have enabled researchers and professionals to run distributed Monte Carlo simulations easier than ever before. However, those parallel simulations require high-quality parallel random number streams. In this work we have analysed to which extent state-of-the-art PRNG qualify for their employment in parallel settings. We show that the popular MT19937 Mersenne Twister and the superior WELL random number generator are able to create high-quality parallel random number streams, in particular with the random seeding method. Furthermore we reveal that seed collisions for those two generator types are very unlikely, even for high degrees of parallelism, and that no inter-stream correlations are observable. Nevertheless, the parallel seeding method has to be carefully selected. We present a working method and show how to test it. All in all, we conclude that applications involving parallel random number streams need special attention on this point, though available generators are well-applicable for these tasks.

Acknowledgements. We gratefully acknowledge the partial financial support from the Center of Mathematical and Computational Modelling $(CM)^2$ of the University of Kaiserslautern and from the German Federal Ministry of Education and Research under grant number 01LY1202D. The authors alone are responsible for the content of this paper.

Edited by: J. Götze
Reviewed by: M. Lechtenberg and one anonymous referee

References

Chandrasekaran, S. and Amira, A.: High Performance FPGA Implementation of the Mersenne Twister, in: Electronic Design, Test and Applications, DELTA 2008, 4th IEEE International Symposium on, 482–485, doi:10.1109/DELTA.2008.113, 2008.

Dalal, I. L. and Stefan, D.: A Hardware Framework for the Fast Generation of Multiple Long-period Random Number Streams, in: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays, FPGA '08, 245–254, ACM, New York, NY, USA, doi:10.1145/1344671.1344707, 2008.

Haramoto, H., Matsumoto, M., Nishimura, T., Panneton, F., and L'Ecuyer, P.: Efficient Jump Ahead for F2-Linear Random Number Generators, INFORMS J. Comput., 20, 385–390, doi:10.1287/ijoc.1070.0251, 2008.

Hill, D. R. C., Mazel, C., Passerat-Palmbach, J., and Traore, M. K.: Distribution of random streams for simulation practitioners, Concurr. Comp.-Pract. E., 25, 1427–1442, doi:10.1002/cpe.2942, 2013.

Knuth, D. E.: Seminumerical Algorithms, vol. 2 of The Art of Computer Programming, Addison-Wesley, Reading, Massachusetts, 3 Edn., 1997.

Korn, R., Korn, E., and Kroisandt, G.: Monte Carlo Methods and Models in Finance and Insurance, Boca Raton, FL: CRC Press, 2010.

L'Ecuyer, P.: Pseudorandom Number Generators, Tech. rep., DIRO, Université de Montreal, C.P. 6128, Succ. Centre-Ville Montréal (Québec), Canada, H3C 3J7, available at: http://www.iro.umontreal.ca/~lecuyer/myftp/papers/eqf.pdf (last access: 26 August 2014), 2007.

L'Ecuyer, P. and Panneton, F.: Fast random number generators based on linear recurrences modulo 2: overview and comparison, in: Simulation Conference, 2005 Proceedings of the Winter, 110–119, doi:10.1109/WSC.2005.1574244, 2005.

L'Ecuyer, P. and Simard, R.: TestU01: A C library for empirical testing of random number generators, ACM Trans. Math. Software, 33, 22-1–22-40, doi:10.1145/1268776.1268777, 2007.

Salmon, J., Moraes, M., Dror, R., and Shaw, D.: Parallel random numbers: As easy as 1, 2, 3, in: High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for, 1–12, 2011.

Srinivasan, A., Mascagni, M., and Ceperley, D.: Testing Parallel Random Number Generators, Parallel Comput., 29, 69–94, doi:10.1016/S0167-8191(02)00163-1, 2003.

Sriram, V. and Kearney, D.: An FPGA Implementation of a Parallelized MT19937 Uniform Random Number Generator, EURASIP Journal on Embedded Systems, 2009, 507426, doi:10.1155/2009/507426, 2009.

Tian, X. and Benkrid, K.: Mersenne Twister Random Number Generation on FPGA, CPU and GPU, in: Adaptive Hardware and Systems, 2009. AHS 2009, NASA/ESA Conference on, 460–464, doi:10.1109/AHS.2009.11, 2009.