Random numbers for large-scale distributed Monte Carlo simulations

Heiko Bauke*
Institut für Theoretische Physik, Universität Magdeburg, Universitätsplatz 2, 39016 Magdeburg, Germany
and Rudolf Peierls Centre for Theoretical Physics, University of Oxford, 1 Keble Road, Oxford, OX1 3NP, United Kingdom

Stephan Mertens†
Institut für Theoretische Physik, Universität Magdeburg, Universitätsplatz 2, 39016 Magdeburg, Germany
(Received 19 September 2006; published 6 June 2007)

Monte Carlo simulations are one of the major tools in statistical physics, complex system science, and other fields, and an increasing number of these simulations is run on distributed systems like clusters or grids. This raises the issue of generating random numbers in a parallel, distributed environment. In this contribution we demonstrate that multiple linear recurrences in finite fields are an ideal method to produce high quality pseudorandom numbers in sequential and parallel algorithms. Their known weakness (failure of sampling points in high dimensions) can be overcome by an appropriate delinearization that preserves all desirable properties of the underlying linear sequence.

DOI: 10.1103/PhysRevE.75.066701 PACS number(s): 02.70.−c, 02.70.Rr, 05.10.Ln

I. INTRODUCTION

The Monte Carlo method is a major industry and random numbers are its key resource. In contrast to most commodities, quantity and quality of randomness have an inverse relation: The more randomness you consume, the better it has to be [1]. The quality issue arises because simulations only *approximate* randomness by generating a stream of deterministic numbers, named *pseudo*random numbers (PRNs), with successive calls to a pseudorandom number generator (PRNG). Considering the ever increasing computing power, however, the quality of PRNGs is a moving target. Simulations that consume 10^{10} pseudorandom numbers were considered large-scale Monte Carlo simulation a few years ago. On a present-day desktop machine this is a ten minute run.

More and more large-scale simulations are run on parallel systems like networked workstations, clusters, or "the grid." In a parallel environment the quality of a PRNG is even more important, to some extent because feasible sample sizes are easily $10,\ldots,10^5$ times larger than on a sequential machine. The main problem is the parallelization of the PRNG itself, however. Some good generators are not parallelizable at all, others lose their efficiency, their quality, or even both when being parallelized [2,3].

In this contribution we discuss linear recurrences in finite fields. They excel as sources of pseudorandomness because they have a solid mathematical foundation and they can easily be parallelized. Their linear structure, however, may cause problems in experiments that sample random points in high dimensional space. This is a known issue ("random numbers fall mainly in the planes" [4]), but it can be overcome by a proper delinearization of the sequence.

The paper is organized as follows. In Sec. II we briefly review some properties of linear recurrences in finite fields, and we motivate their use as PRNGs. In Sec. III we discuss various techniques for parallelizing a PRNG. We will see that only two of these techniques allow for parallel simulations whose results *do not* necessarily depend on the number of processes, and we will see how linear recurrences support these parallelization techniques. Section IV addresses the problem of linear sequences to sample points in high dimensions, measured by the spectral test. We show that with a proper transformation (delinearization) we can generate a new sequence that shares all nice properties with the original sequence (like parallelizability, equidistribution properties, etc.), but passes many statistical tests that are sensitive to the hyperplane structure of points of linear sequences in high dimensions. In the last two sections we discuss the quality of the linear and nonlinear sequences in parallel Monte Carlo applications and how to implement these generators efficiently.

II. RECURRENT RANDOMNESS

Almost all PRNGs produce a sequence $(r)=r_1,r_2,\ldots$ of pseudorandom numbers by a recurrence

$$r_i = f(r_{i-1}, r_{i-2}, \ldots, r_{i-k}), \tag{1}$$

and the art of random number generation lies in the design of the function f. Numerous recipes for f have been discussed in the literature [5,6]. One of the oldest and most popular PRNGs is the linear congruential generator (LCG)

$$r_i = a r_{i-1} + b \bmod m \tag{2}$$

introduced by Lehmer [7]. The additive constant b may be zero. Knuth [8] proposed a generalization of Lehmer's method known as multiple recursive generator (MRG) [6,9,10]

$$r_i = a_1 r_{i-1} + a_2 r_{i-2} + \cdots + a_n r_{i-n} \bmod m. \tag{3}$$

Surprisingly, on a computer *all* recurrences are essentially linear. This is due to the simple fact that computers operate on numbers of finite precision; let us say integers between 0

*Email address: heiko.bauke@physics.ox.ac.uk
†Email address: stephan.mertens@physik.uni-magdeburg.de

and $r_{\max}$. This has two important consequences.

(1) Every recurrence (1) must be periodic.

(2) We can embed the sequence (r) into the finite field $\mathbb{F}_m$, the field of integers with arithmetic modulo m, where m is a prime number larger than the largest value in the sequence (r).

The relevance of linear sequences is then based on the following fact ([11], corollary 6.2.3):

Theorem 1. All finite length sequences and all periodic sequences over a finite field $\mathbb{F}_m$ can be generated by a linear recurrence (3).

In the theory of finite fields, a sequence of type (3) is called *linear feedback shift register sequence*, or LFSR sequence for short. Note that a LFSR sequence is fully determined by specifying n coefficients $(a_1, a_2, \ldots, a_n)$ plus n initial values $(r_1, r_2, \ldots, r_n)$.

According to Theorem 1 all finite length sequences and all periodic sequences are LFSR sequences. The *linear complexity* of a sequence is the minimum order of a linear recurrence that generates this sequence. The Berlekamp-Massey algorithm [11,12] is an efficient procedure for finding this shortest linear recurrence. The linear complexity of a *random* sequence of length T on average equals $T/2$ [13]. As a consequence a random sequence cannot be compressed by coding it as a linear recurrence, because $T/2$ coefficients plus $T/2$ initial values have to be specified. Typically the linear complexity of a nonlinear sequence (1) is of the same order of magnitude as the period of the sequence. Since the designers of PRNGs strive for "astronomically" large periods, implementing nonlinear generators as a linear recurrence is not tractable. As a matter of principle, however, all we need to discuss are linear recurrences, and any nonlinear recurrence can be seen as an efficient implementation of a large order linear recurrence. The popular PRNG "Mersenne Twister" [14], for example, is an efficient implementation of a linear recurrence in $\mathbb{F}_2$ of order 19 937.

There is a wealth of rigorous results on LFSR sequences that can (and should) be used to construct a good PRNG. Here we only discuss a few but important facts without proofs. A detailed discussion of LFSR sequences including proofs can be found in [11,15–19].

Since the all zero tuple (0,0,…,0) is a fixed point of Eq. (3), the maximum period of a LFSR sequence cannot exceed m^n-1. The following theorem tells us precisely how to choose the coefficients $(a_1, a_2, \ldots, a_n)$ to achieve this period [5].

Theorem 2. The LFSR sequence (3) over $\mathbb{F}_m$ has period m^n-1, if and only if the characteristic polynomial

$$f(x) = x^n - a_1 x^{n-1} - a_2 x^{n-2} - \cdots - a_n \tag{4}$$

is *primitive* modulo m.

A monic polynomial $f(x)$ of degree n over $\mathbb{F}_m$ is primitive modulo m, if and only if it is irreducible (i.e., cannot be factorized over $\mathbb{F}_m$), and if it has a primitive element of the extension field $\mathbb{F}_{m^n}$ as one of its roots. The number of primitive polynomials of degree n modulo m is equal to $\phi(m^n-1)/n = O(m^n/(n \ln(n \ln m)))$ [20], where $\phi(x)$ denotes Euler's totient function. As a consequence a random polynomial of degree n is primitive modulo m with probability $\simeq 1/(n \ln(n \ln m))$, and finding primitive polynomials reduces to testing whether a given polynomial is primitive. The latter can be done efficiently, if the factorization of m^n-1 is known [11], and most computer algebra systems offer a procedure for this.

Theorem 3. Let (r) be an LFSR sequence (3) with a primitive characteristic polynomial. Then each k-tuple $(r_{i+1}, \ldots, r_{i+k}) \in \{0, 1, \ldots, m-1\}^k$ occurs m^{n-k} times per period for $k \leqslant n$ (except the all zero tuple for $k=n$, which never occurs).

From this theorem it follows that, if a tuple of k successive terms $k \leqslant n$ is drawn from a random position of a LFSR sequence, the outcome is uniformly distributed over all possible k-tuples in $\mathbb{F}_m$. This is exactly what one would expect from a truly random sequence. In terms of Compagner's ensemble theory, tuples of length $k \leqslant n$ of a LFSR sequence with primitive characteristic polynomial are indistinguishable from truly random tuples [21,22].

Theorem 4. Let (r) be an LFSR sequence (3) with period $T=m^n-1$ and let α be a complex mth root of unity and $\bar{\alpha}$ its complex conjugate. Then

$$C(h) := \sum_{i=1}^{T} \alpha^{r_i} \bar{\alpha}^{r_{i+h}} = \begin{cases} T & \text{if } h = 0 \bmod T \\ -1 & \text{if } h \neq 0 \bmod T. \end{cases} \tag{5}$$

$C(h)$ can be interpreted as the autocorrelation function of the sequence, and Theorem 4 tells us that LFSR sequences with maximum period have autocorrelations that are very similar to the autocorrelations of a random sequence with period T. Together with the nice equidistribution properties (Theorem 3) this qualifies LFSR sequences with maximum period as *pseudonoise sequences*, a term originally coined by Golomb for binary sequences [11,15].

III. PARALLELIZATION

A. General parallelization techniques

In parallel applications we need to generate streams $t_{j,i}$ of random numbers, where $j=0,1,\ldots,p-1$ numbers the streams for each of the p processes. We require statistical independence of the $t_{j,i}$ within each stream and between the streams. Four different parallelization techniques are used in practice.

Random seeding. All processes use the same PRNG but a different "random" seed. The hope is that they will generate nonoverlapping and uncorrelated subsequences of the original PRNG. This hope, however, has no theoretical foundation. Random seeding is a violation of Donald Knuth's advice, "Random number generators should not be chosen at random" [5].

Parametrization. All processes use the same type of generator but with different parameters for each processor. Example: linear congruential generators with additive constant b_j for the jth stream [23]

$$t_{j,i} = a t_{j,i-1} + b_j \bmod m, \tag{6}$$

where the b_j's are different prime numbers just below $\sqrt{m/2}$. Another variant uses different multipliers a for different

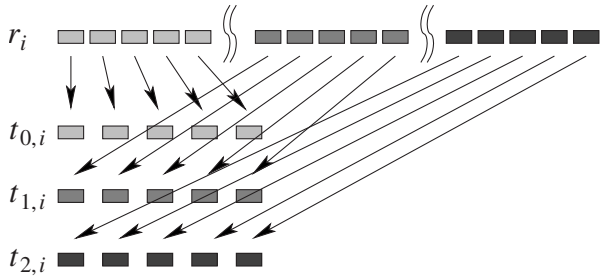

FIG. 1. Parallelization by block splitting.

streams [24]. The theoretical foundation of these methods is weak, and empirical tests have revealed serious correlations between streams [25]. On a massive parallel system you may need thousands of parallel streams, and it is not trivial to find a type of PRNG with thousands of "well tested" parameter sets.

Block splitting. Let M be the maximum number of calls to a PRNG by each processor, and let p be the number of processes. Then we can split the sequence (r) of a sequential PRNG into consecutive blocks of length M such that

$$\begin{aligned} t_{0,i} &= r_i, \\ t_{1,i} &= r_{i+M}, \\ &\vdots \\ t_{p-1,i} &= r_{i+M(p-1)}. \end{aligned} \tag{7}$$

This method works only if we know M in advance or can at least safely estimate an upper bound for M. To apply block splitting it is necessary to jump from the ith random number to the $(i+M)$th number without calculating the numbers in-between, which cannot be done efficiently for many PRNGs. A potential disadvantage of this method is that long range correlations, usually not observed in sequential simulations, may become short range correlations between substreams [26,27]. Block splitting is illustrated in Fig. 1.

Leapfrog. The leapfrog method distributes a sequence (r) of random numbers over p processes by decimating this base sequence such that

$$\begin{aligned} t_{0,i} &= r_{pi}, \\ t_{1,i} &= r_{pi+1}, \\ &\vdots \\ t_{p-1,i} &= r_{pi+(p-1)}. \end{aligned} \tag{8}$$

Leapfrogging is illustrated in Fig. 2. It is the most versatile and robust method for parallelization and it does not require an *a priori* estimate of how many random numbers will be consumed by each processor. An efficient implementation requires a PRNG that can be modified to generate directly only every kth element of the original sequence. Again this excludes many popular PRNGs.

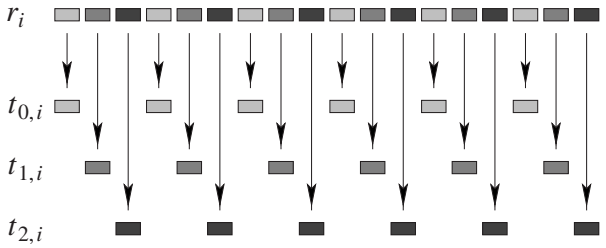

FIG. 2. Parallelization by leapfrogging.

Note that for a periodic sequence (r) the substreams derived from block splitting are cyclic shifts of the original sequence (r), and for p not dividing the period of (r), the leapfrog sequences are cyclic shifts of each other. Hence the leapfrog method is equivalent to block splitting on a different base sequence.

B. Playing fair

We say that a parallel Monte Carlo simulation *plays fair*, if its outcome is strictly independent of the underlying hardware, where strictly means deterministically, not just statistically. Fair play implies the use of the same PRNs in the same context, independently of the number of parallel processes. It is mandatory for debugging, especially in parallel environments where the number of parallel processes varies from run to run, but another benefit of playing fair is even more important: Fair play guarantees that the quality of a PRNG with respect to an application does not depend on the degree of parallelization.

Obviously, parametrization or random seeding as discussed above prevent a simulation from playing fair. Leapfrog and block splitting, on the other hand, do allow one to use the same PRNs within the same context independently of the number of parallel streams.

Consider the site percolation problem. A site in a lattice of size N is occupied with some probability, and the occupancy is determined by a PRN. M random configurations are generated. A naive parallel simulation on p processes could split a base sequence into p leapfrog streams and having each process generate $\approx M/p$ lattice configurations, independently of the other processes. Obviously, this parallel simulation is not equivalent to its sequential version that consumes PRNs from the base sequence to generate one lattice configuration after another. The effective shape of the resulting lattice configurations depends on the number of processes. This parallel algorithm does not play fair.

We can turn the site percolation simulation into a fair playing algorithm by leapfrogging on the level of lattice configurations. Here each process consumes distinct contiguous blocks of PRNs from the sequence (r), and the workload is spread over p processors in such a way, that each process analyzes each pth lattice. If we number the processes by their rank i from 0 to $p-1$ and the lattices from 0 to $M-1$, each process starts with a lattice whose number equals its own rank. That means process i has to skip iN PRNs before the first lattice configuration is generated. Thereafter each process can skip $p-1$ lattices, i.e., $(p-1)N$ PRNs and continue with the next lattice.

Organizing simulation algorithms such that they play fair is not always as easy as in the above example, but with a little effort one can achieve fair play in more complicated situations, too. This may require the combination of block splitting and the leapfrog method, or iterated leapfrogging. Sometimes it is also necessary to use more than one stream of PRNs per process, e.g., in the Swendsen Wang cluster algorithm one may use one PRNG to construct the bond percolation clusters and another PRNG to decide to flip the clusters.

C. Parallelization of LFSR sequences

As a matter of fact, LFSR sequences do support leapfrog and block splitting very well. Block splitting means basically jumping ahead in a PRN sequence. In the case of LFSR sequences this can be done quite efficiently. Note, that by introducing a companion matrix A the linear recurrence (3) can be written as a vector matrix product [28].

$$\begin{pmatrix} r_{i-(n-1)} \\ \vdots \\ r_{i-1} \\ r_i \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \\ a_n & a_{n-1} & \dots & a_1 \end{pmatrix}}_{A} \begin{pmatrix} r_{i-n} \\ \vdots \\ r_{i-2} \\ r_{i-1} \end{pmatrix} \bmod m. \tag{9}$$

From this formula it follows immediately that the M-fold successive iteration of Eq. (3) may be written as

$$\begin{pmatrix} r_{i-(n-1)} \\ \vdots \\ r_{i-1} \\ r_i \end{pmatrix} = A^M \begin{pmatrix} r_{i-M-(n-1)} \\ \vdots \\ r_{i-M-1} \\ r_{i-M} \end{pmatrix} \bmod m. \tag{10}$$

Matrix exponentiation can be accomplished in $O(n^3 \ln M)$ steps via binary exponentiation (also known as exponentiation by squaring).

Implementing leapfrogging efficiently is less straightforward. Calculating $t_{j,i}=r_{pi+j}$ via

$$\begin{pmatrix} r_{pi+j-(n-1)} \\ \vdots \\ r_{pi+j-1} \\ r_{pi+j} \end{pmatrix} = A^p \begin{pmatrix} r_{p(i-1)+j-(n-1)} \\ \vdots \\ r_{p(i-1)+j-1} \\ r_{p(i-1)+j} \end{pmatrix} \bmod m \tag{11}$$

is no option, because A^p is usually a dense matrix, in which case calculating a new element from the leapfrog sequence requires $O(n^2)$ operations instead of $O(n)$ operations in the base sequence.

The following theorem assures that the leapfrog subsequences of LFSR sequences are again LFSR sequences [11]. This will provide us with a very efficient way to generate leapfrog sequences.

Theorem 5. Let (r) be a LFSR sequence based on a primitive polynomial of degree n with period m^n-1 (pseudonoise sequence) over $\mathbb{F}_m$, and let (t) be the decimated sequence with lag $p>0$ and offset j, e.g.,

$$t_{j,i} = r_{pi+j}. \tag{12}$$

Then (t_j) is a LFSR sequence based on a primitive polynomial of degree n, too, if and only if p and m^n-1 are coprime, e.g., greatest common divisor $(\gcd)(m^n-1,p)=1$. In addition, (r) and (t_j) are not just cyclic shifts of each other, except when

$$p = m^h \tag{13}$$

for some $0 \leqslant h < n$. If $\gcd(m^n-1,p)>1$ the sequence (t_j) is still a LFSR sequence, but not a pseudonoise sequence.

It is not hard to find prime numbers m such that m^n-1 has very few (and large) prime factors. For such numbers, the leapfrog method yields pseudonoise sequences for any reasonable number of parallel streams (see also Sec. III D and the Appendix).

While Theorem 5 ensures that leapfrog sequences are not just cyclic shifts of the base sequence [unless Eq. (13) holds], the leapfrog sequences are cyclic shifts of *each other*. This is true for general sequences, not just LFSR sequences. Consider an arbitrary sequence (r) with period T. If $\gcd(T,p)=1$, all leapfrog sequences $(t_1), (t_2), \dots, (t_p)$ are cyclic shifts of each other, i.e., for every pair of leapfrog sequences (t_{j_1}) and (t_{j_2}) of a common base sequence (r) with period T there is a constant s, such that $t_{j_1,i}=t_{j_2,i+s}$ for all i, and s is at least $\lfloor T/p \rfloor$. Furthermore, if $\gcd(T,p)=d>1$, the period of each leapfrog sequence equals T/d and there are d classes of leapfrog sequences. Within a class of leapfrog sequences there are p/d sequences, each sequence is just a cyclic shift of another and the size of the shift is at least $\lfloor T/p \rfloor$.

Theorem 5 tells us that all leapfrog sequences of a LFSR sequence of degree n can be generated by another LFSR of degree n or less. The following theorem [[11], Theorem 6.6.1] gives us a recipe to calculate the coefficients $(b_1,b_2,\dots,b_n)$ of the corresponding leapfrog feedback polynomial.

Theorem 6. Let (t) be a (periodic) LFSR sequence over the field $\mathbb{F}_m$ and $f(x)$ its characteristic polynomial of degree n. Then the coefficients $(b_1,b_2,\dots,b_n)$ of $f(x)$ can be computed from $2n$ successive elements of (t) by solving the linear system

$$\begin{pmatrix} t_{i+n} \\ t_{i+n+1} \\ \vdots \\ t_{i+2n-1} \end{pmatrix} = \begin{pmatrix} t_{i+n-1} & \dots & t_{i+1} & t_i \\ t_{i+n} & \dots & t_{i+2} & t_{i+1} \\ \vdots & \ddots & \vdots & \vdots \\ t_{i+2n-2} & \dots & t_{i+n} & t_{i+n-1} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \tag{14}$$

over $\mathbb{F}_m$.

Starting from the base sequence we determine $2n$ values of the sequence (t) by applying the leapfrog rule. Then we solve Eq. (14) by Gaussian elimination to get the characteristic polynomial for a new LFSR generator that yields the elements of the leapfrog sequence directly with each call. If the matrix in Eq. (14) is singular, the linear system has more than one solution, and it is sufficient to pick one of them. In this case it is always possible to generate the leapfrog sequence by a LFSR of degree less than the degree of the original sequence.

D. Choice of modulus

LFSR sequences over $\mathbb{F}_2$ with sparse feedback polynomials are popular sources of PRNs [5,29,30] and generators of this type can be found in various software libraries. This is due to the fact that multiplication in $\mathbb{F}_2$ is trivial, addition reduces to exclusive-OR and the mod operation comes for free. As a result, generators that operate in $\mathbb{F}_2$ are extremely fast. Unfortunately, these generators suffer from serious statistical defects [30–33] that can be blamed quite generally on the small size of the underlying field [34] (see also Sec. VI A). In parallel applications we have the additional drawback, that, if the leapfrog method is applied to a LFSR sequence with sparse characteristic polynomial, the new sequence will have a dense polynomial. The computational complexity of generating values of the LFSR sequence grows from $O(1)$ to $O(n)$. Remember that for generators in $\mathbb{F}_2$, n is typically of order 1000 or even larger to get a long period 2^n-1.

The theorems and parallelization techniques we have presented so far do apply to LFSR sequences over any finite field $\mathbb{F}_m$. Therefore we are free to choose the prime modulus m. In order to get maximum entropy on the macrostate level [35] m should be as large as possible. A good choice is to set m to a value that is of the order of the largest representable integer of the computer. If the computer deals with e-bit registers, we may write the modulus as $m=2^e-k$, with k reasonably small. In fact if $k(k+2)\leqslant m$ modular reduction can be done reasonably fast by a few bit-shifts, additions, and multiplications (see Sec. V A). Furthermore a large modulus allows us to restrict the degree of the LFSR to rather small values, e.g., $n\approx 4$, while the PRNG has a large period and good statistical properties.

In accordance with Theorem 5 a leapfrog sequence of a pseudonoise sequence is a pseudonoise sequence, too, if and only if its period m^n-1 and the lag p are coprime. For that reason m^n-1 should have a small number of prime factors [36]. It can be shown that m^n-1 has at least three prime factors and if the number of prime factors does not exceed three, then m is necessarily a Sophie-Germain prime and n a prime larger than two (see the Appendix for details).

To sum up, the modulus m of a LFSR sequence should be a Sophie-Germain prime, such that m^n-1 has no more than three prime factors and such that $m=2^e-k$ and $k(k+2)\leqslant m$ for some integers e and k.

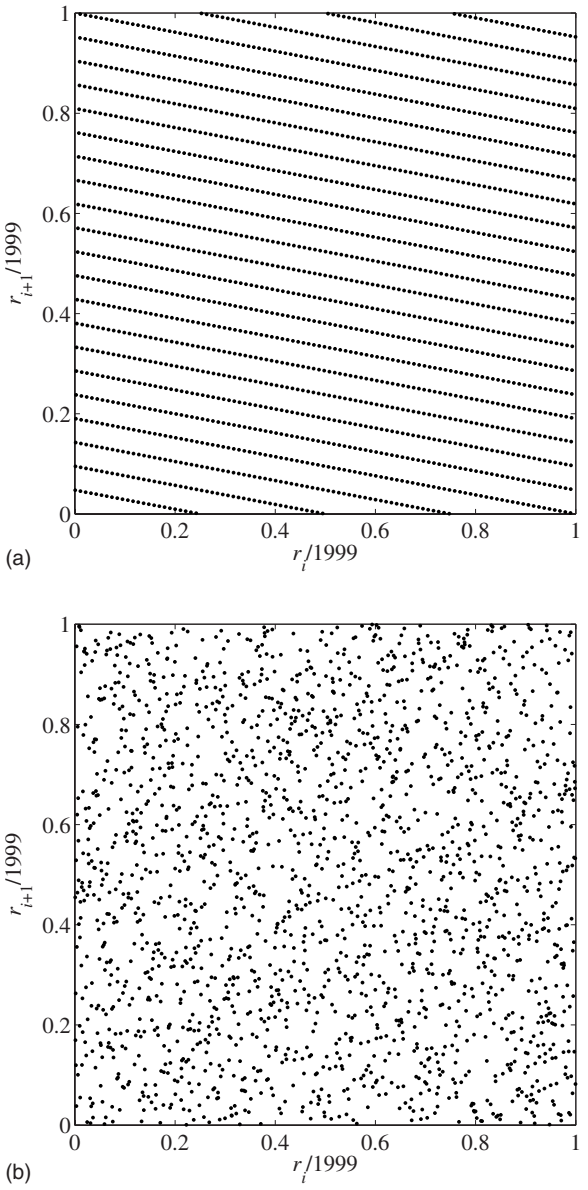

FIG. 3. Exponentiation of a generating element in a prime field is an effective way to destroy the linear structures of LFSR sequences. Both pictures show the full period of the generator. Top: $r_i=95r_{i-i} \bmod 1999$. Bottom: $r_i=1099^{q_i} \bmod 1999$ with $q_i=95q_{i-i} \bmod 1999$.

IV. DELINEARIZATION

LFSR sequences over prime fields with a large prime modulus seem to be ideally suited as PRNGs, especially in parallel environments. They have, however, a well known weakness. When used to sample coordinates in d-dimensional space, pseudonoise sequences cover every point for $d<n$, and every point except (0,0,…,0) for $d=n$. For $d>n$ the set of positions generated is obviously sparse, and the linearity of the production rule (3) leads to the concentration of the sampling points on n-dimensional hyperplanes [37,38] (see also the top of Fig. 3). This phenomenon, first noticed by Marsaglia in 1968 [4], motivates one of the well known tests for PRNGs, the so-called spectral test [5]. The spectral test checks the behavior of a generator when its outputs are used to form d tuples. LFSR sequences can fail the spectral test in dimensions larger than the register size n. Closely related to this mechanism are the observed correlations in other empirical tests such as the birthday spacings test and the collision test [39,40]. Nonlinear generators do quite well in all these tests, but compared to LFSR sequences they have much less nice and *provable* properties and they are not suited for fair playing parallelization.

To get the best of both worlds we propose a delinearization that preserves all the nice properties of linear pseudonoise sequences:

Theorem 7. Let (q) be a pseudonoise sequence in $\mathbb{F}_m$, and

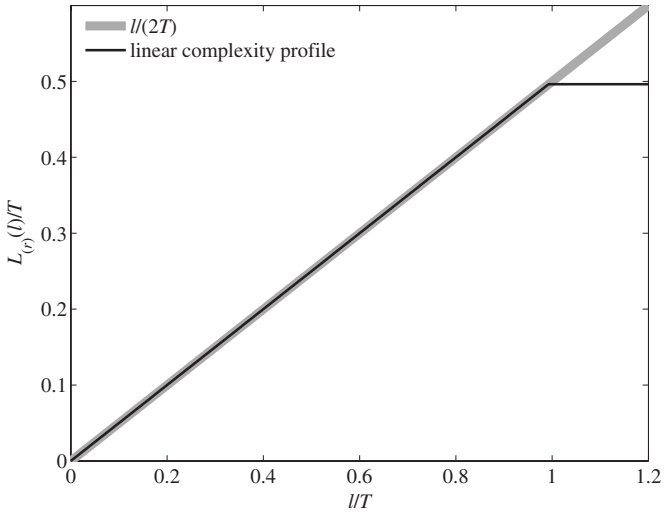

FIG. 4. Linear complexity profile $L_{(r)}(l)$ of a YARN sequence, produced by the recurrence $q_i=173q_{i-1}+219q_{i-2} \bmod 317$ and $r_i=151^{q_i} \bmod 317$. The period of this sequence equals $T=317^2-1$.

let g be a generating element of the multiplicative group $\mathbb{F}_m^*$. Then the sequence (r) with

$$r_i = \begin{cases} g^{q_i} \bmod m & \text{if } q_i > 0 \\ 0 & \text{if } q_i = 0 \end{cases} \tag{15}$$

is a pseudonoise sequence, too.

The proof of this theorem is trivial: since g is a generator of $\mathbb{F}_m^*$, the map (15) is bijective. We call delinearized generators based on Theorem 7 YARN generators (yet another random number).

The linearity is completely destroyed by the map (15) (see Fig. 3). Let $L_{(r)}(l)$ denote the linear complexity of the subsequence $(r_1, r_2, \ldots, r_l)$. This function is known as the linear complexity profile of (r). For a truly random sequence it grows on average like $l/2$. Figure 4 shows the linear complexity profile $L_{(r)}(l)$ of a typical YARN sequence. It shows the same growth rate as a truly random sequence up to the point where more than 99% of the period has been considered. Sharing the linear complexity profile with a truly random sequence, we may say that the YARN generator is as nonlinear as it can get.

V. IMPLEMENTATION AND EFFICIENCY

LFSR sequences over prime fields larger than $\mathbb{F}_2$ have been proposed as PRNGs in the literature [5,28,37]. An efficient implementation of these recurrences requires some care, however.

We assume that all integer arithmetic is done in w-bit registers and $m<2^{w-1}$. Under this condition addition modulo m can be done without overflow problems. But multiplying two $(w-1)$-bit integers modulo m is not straightforward because the intermediate product has $2(w-1)$ significant bits and cannot be stored in a w-bit register. For the special case $a_k<\sqrt{m}$ Schrage [41] showed how to calculate $a_k r_{i-k} \bmod m$ without overflow. Based on this technique a portable implementation of LFSR sequences with coefficients $a_k<\sqrt{m}$ is presented in [10]. For parallel PRNGs this method does not apply because the leapfrog method may yield coefficients that violate this condition. Knuth ([5], Sec. 3.2.1.1) proposed a generalization of Schrage's method for arbitrary positive factors less than m, but this method requires up to twelve multiplications and divisions and is therefore not very efficient.

The only way to implement Eq. (3) without additional measures to circumvent overflow problems is to restrict m to $m<2^{w/2}$. On machines with 32-bit registers, 16 random bits per number is not enough for some applications. Fortunately today's C compilers provide fast 64-bit arithmetic even on 32-bit CPUs and genuine 64-bit CPUs become more and more common. This allows us to increase m to 32.

A. Efficient modular reduction

Since the modulo operation in Eq. (3) is usually slower than other integer operations such as addition, multiplication, boolean operations, or shifting, it has a significant impact on the total performance of PRNGs based on LFSR sequences. If the modulus is a Mersenne prime $m=2^e-1$, however, the modulo operation can be done using only a few additions, boolean operations, and shift operations [42].

A summand $s=a_k r_{i-k}$ in Eq. (3) will never exceed $(m-1)^2=(2^e-2)^2$ and for each positive integer $s\in[0,(2^e-1)^2]$ there is a unique decomposition of s into

$$s = r2^e + q \quad \text{with } 0 \leqslant q < 2^e. \tag{16}$$

From this decomposition we conclude

$$s - r2^e = q,$$

$$s - r(2^e - 1) = q + r,$$

$$s \bmod (2^e - 1) = q + r \bmod (2^e - 1),$$

and r and q are bounded from above by

$$q < 2^e \quad \text{and} \quad r \leqslant \lfloor (2^e-2)^2/2^e \rfloor < 2^e - 2,$$

respectively, and therefore

$$q + r < 2^e + 2^e - 2 = 2m.$$

So if $m=2^e-1$ and $s\leqslant(m-1)^2$, $x=s \bmod m$ can be calculated solely by shift operations, boolean operations, and addition, viz.,

$$x = (s \bmod 2^e) + \lfloor s/2^e \rfloor. \tag{17}$$

If Eq. (17) yields a value $x\geqslant m$ we simply subtract m.

From a computational point of view, Mersenne prime moduli are optimal and we propose to choose the modulus $m=2^{31}-1$. This is the largest positive integer that can be represented by a signed 32-bit integer variable, and it is also a Mersenne prime. On the other hand, our theoretical considerations favor Sophie-Germain prime moduli, for which Eq. (17) does not apply directly. But one can generalize Eq. (17) to moduli 2^e-k [43]. Again we start from a decomposition of s into

$$s = r2^e + q \quad \text{with} \quad 0 \leqslant q < 2^e, \tag{18}$$

and conclude

$$s - r2^e = q,$$

$$s - r(2^e - k) = q + kr,$$

$$s \bmod (2^e - k) = q + kr \bmod (2^e - k).$$

The sum $s' = q + kr$ exceeds the modulus at most by a factor $k+1$, because by applying

$$q < 2^e \quad \text{and} \quad r \leqslant \lfloor (2^e - k - 1)^2 / 2^e \rfloor < 2^e - k - 1,$$

we get the bound

$$q + kr < 2^e + k(2^e - k - 1) = (k+1)m.$$

In addition, by the decomposition of $s' = q + kr$,

$$s' = r'2^e + q' \quad \text{with } 0 \leqslant q' < 2^e,$$

it follows

$$s \bmod (2^e - k) = s' \bmod (2^e - k) = q' + kr' \bmod (2^e - k),$$

and this time the bounds

$$q' < 2^e \quad \text{and } r' \leqslant \lfloor (k+1)(2^e - k)/2^e \rfloor < k + 1$$

and

$$q' + kr' < 2^e + k(k+1) = m + k(k+2)$$

hold. Therefore if $m = 2^e - k$, $s \leqslant (m-k)^2$ and $k(k+2) \leqslant m$, $x = s \bmod m$ can be calculated solely by shift operations, boolean operations, and addition, viz.,

$$s' = (s \bmod 2^e) + k\lfloor s/2^e \rfloor,$$

$$x = (s' \bmod 2^e) + k\lfloor s'/2^e \rfloor. \tag{19}$$

If Eq. (19) yields a value $x \geqslant m$, a single subtraction of m will complete the modular reduction. To carry out Eq. (19), twice as many operations as for Eq. (17) are needed, but Eq. (19) applies for all moduli $m = 2^e - k$ with $k(k+2) \leqslant m$. Note that on systems with big enough word size (such as 64-bit architectures), just doing the mod operation might well be faster than using Eq. (19).

B. Fast delinearization

YARN generators hide linear structures of LFSR sequences (q) by raising a generating element g to the power $g^{q_i} \bmod m$. This can be done efficiently by binary exponentiation, which takes $O(\log m)$ steps. But considering LFSR sequences with only a few feedback taps ($n \leqslant 6$) and $m \approx 2^{31}$ even fast exponentiation is significantly more expensive than a single iteration of Eq. (3). Therefore we propose to implement exponentiation by table lookup. If m is a $2e'$-bit number we apply the decomposition

$$q_i = q_{i,1} 2^{e'} + q_{i,0}$$

with

TABLE I. Generators of the TRNG library. All PRNGs denoted by trng::mrg $n\{s\}$ are pseudonoise sequences over $\mathbb{F}_m$ with n feedback terms, trng::yarn $n\{s\}$ denotes its delinearized counterpart. The prime modulus $2^{31}-1$ is a Mersenne prime, whereas all other moduli are Sophie-Germain primes.

Name	m	Period
trng::mrg2, trng::yarn2	$2^{31}-1$	$\approx 2^{62}$
trng::mrg3, trng::yarn3	$2^{31}-1$	$\approx 2^{93}$
trng::mrg3s, trng::yarn3s	$2^{31}-21\,069$	$\approx 2^{93}$
trng::mrg4, trng::yarn4	$2^{31}-1$	$\approx 2^{124}$
trng::mrg5, trng::yarn5	$2^{31}-1$	$\approx 2^{155}$
trng::yarn5s, trng::yarn5s	$2^{31}-22\,641$	$\approx 2^{155}$

$$q_{i,1} = \lfloor q_i / 2^{e'} \rfloor, \quad q_{i,0} = q_i \bmod 2^{e'}, \tag{20}$$

and use the identity

$$r_i = g^{q_i} \bmod m = (g^{2^{e'}})^{q_{i,1}} g^{q_{i,0}} \bmod m \tag{21}$$

to calculate $g^{q_i} \bmod m$ by two table lookups and one multiplication modulo m. If $m < 2^{31}$ the tables for $(g^{2^{e'}})^{q_{i,1}} \bmod m$ and $g^{q_{i,0}} \bmod m$ have 2^{16} and 2^{15} entries, respectively. These 384 kbytes of data easily fit into the cache of modern CPUs.

C. TRNG

LFSR sequences and their nonlinear counterparts have been implemented in the TRNG software package, a portable and highly optimized library of parallelizable PRNGs. Parallelization by block splitting as well as by leapfrogging are supported by all generators. TRNG is publicly available for download [44]. Its design is based on a proposal for the next revision of the ISO C++ standard [48]. TRNG uses 64-bit arithmetic, fast modular reduction (17) and (19), and exponentiation by table lookup (21) to implement PRNGs based on LFSR sequences over prime fields, with Mersenne or Sophie-Germain prime modulus. Table I describes the generators of the software package.

Table II shows some benchmark results. Apparently the performance of both types of PRNGs competes well with popular PRNGs such as the Mersenne Twister or RANLUX. Absolute as well as relative timings in Table II depend on compiler and CPU architecture, but the table gives a rough performance measure of our PRNGs.

VI. QUALITY

It is not possible to *prove* that a given PRNG will work well in any simulation, but one can subject a PRNG to a battery of tests that mimic typical applications. One distinguishes empirical and theoretical test procedures. Empirical tests take a finite sequence of PRNs and compute certain statistics to judge the generator as "random" or not. While empirical tests focus only on the statistical properties of a finite stream of PRNs and ignore all the details of the underlying PRNG algorithm, theoretical tests analyze the PRNG algorithm itself by number-theoretic methods and establish

TABLE II. Performance of various PRNGs from the TRNG library version 4.0 [44] and the GNU Scientific Library (GSL) version 1.8 [45]. The test program was compiled using the Intel C++ compiler version 9.1 with high optimization (option −03) and was executed on a Pentium IV 3 GHz.

TRNG		GSL	
Generator	PRNs per second	Generator	PRNs per second
trng::mrg2	48×10^6	mt19937[a]	41×10^6
trng::mrg3	43×10^6	r250[b]	85×10^6
trng::mrg3s	36×10^6	gfsr4[c]	77×10^6
trng::mrg4	38×10^6	ran2[d]	27×10^6
trng::mrg5	38×10^6	ranlux[e]	8×10^6
trng::mrg5s	23×10^6	ranlux389[f]	5×10^6
trng::yarn2	26×10^6		
trng::yarn3	23×10^6		
trng::yarn3s	16×10^6		
trng::yarn4	20×10^6		
trng::yarn5	22×10^6		
trng::yarn5s	13×10^6		

[a]Mersenne twister [14].
[b]Shift-register sequence over $\mathbb{F}_2$, $r_i=r_{i-103}\oplus r_{i-250}$ [29] (see also Sec. VI A).
[c]Shift-register sequence over $\mathbb{F}_2$, $r_i=r_{i-471}\oplus r_{i-1586}\oplus r_{i-6988}\oplus r_{i-9689}$ [30].
[d]Numerical recipes generator ran2 [46].
[e]RANLUX generator with default luxury level [47].
[f]RANLUX generator with highest luxury level [47].

a priori characteristics of the PRN sequence. These *a priori* characteristics may be used to choose good parameter sets for certain classes of PRNGs. The parameters of the LFSR sequences used in TRNG, for example, were found by extensive computer search [38] to give good results in the spectral test [5].

In this section we apply statistical tests to the generators shown in Table I to investigate their statistical properties and those of its leapfrog substreams. The test procedures are motivated by problems that occur frequently in physics, and that are known as sensitive tests for PRNGs, namely, the cluster Monte Carlo simulation of the two-dimensional Ising model and random walks. Furthermore we analyze the effect of the exponentiation step of the YARN generator in the birthday spacings test. Results of additional empirical tests are presented in the documentation of the software library trng [44].

A. Cluster Monte Carlo simulations

In [33] it is reported that some "high quality" generators produce systematically wrong results in a simulation of the two-dimensional Ising model at a critical temperature of the infinite system by the Wolff cluster flipping algorithm [49]. Deviations are due to the bad mixing properties of the generators (low macrostate entropy) [35]. An infamous example for such a bad PRNG is the r250 generator [29]. It combines e LFSR sequences over $\mathbb{F}_2$ to produce a stream of e-bit integers via

$$r_i = r_{i-103} \oplus r_{i-250}, \tag{22}$$

where $\oplus$ denotes the bitwise exclusive-OR operation (addition in $\mathbb{F}_2$).

Figure 5 shows the results of cluster Monte Carlo simulations of the Ising model on a 16×16 square lattice with cyclic boundary conditions for the generator r250 and its leapfrog substreams. Each simulation was done ten times at the critical temperature applying the Wolff cluster algorithm. The mean of the internal energy E_{MC}, the specific heat c_{MC}, and the empirical standard deviation of both quantities have been measured. The gray scale in Fig. 5 codes the deviation from the exact values [50]. Black bars indicate deviations larger than four standard deviations from the exact values. The original r250 sequence and its leapfrog sequences yield bad results if the number of processes is a power of two. Since the statistical error decreases as the number of Monte Carlo updates increases, these systematic deviations become more and more significant and a PRNG with bad statistical properties will reveal its defects. Figure 5 visualizes the inverse relation between quantity and quality of streams of PRNs mentioned in the Introduction.

The apparent dependency of the quality on the leapfrog parameter p is easily understood: If p is a power of two, the leapfrog sequence is just a shifted version of the original sequence [15,30]. On the other hand, if p is not a power of two, the leapfrog sequence corresponds to a LFSR sequence with a denser characteristic polynomial. For $p=3$, e. g., the recursion reads

$$r_i = r_{i-103} \oplus r_{i-152} \oplus r_{i-201} \oplus r_{i-250},$$

and for $p=7$,

$$r_i = r_{i-103} \oplus r_{i-124} \oplus r_{i-145} \oplus r_{i-166} \oplus r_{i-187} \oplus r_{i-208} \oplus r_{i-229} \oplus r_{i-250},$$

respectively. The quality of a LFSR generator usually increases with the number of nonzero coefficients in its characteristic polynomial [9,51,52], a phenomenon that can be easily explained theoretically [34], and that also holds for LFSR sequences over general prime fields [35].

While LFSR sequences over $\mathbb{F}_2$ with sparse characteristic polynomial are known to fail in cluster Monte Carlo simulations, LFSR sequences over large prime fields with dense characteristic polynomial perform very well in these tests. This can be seen in the lower half of Fig. 5 for the trng::mrg3s generator; other generators from Table I perform likewise.

B. Random walks

The simulation of the two-dimensional Ising model in the previous section measured the quality within distinct substreams. Potential interstream correlations are not taken into account. In [53] some tests are proposed that are designed to detect correlations between substreams. One of them is the so-called S_N test.

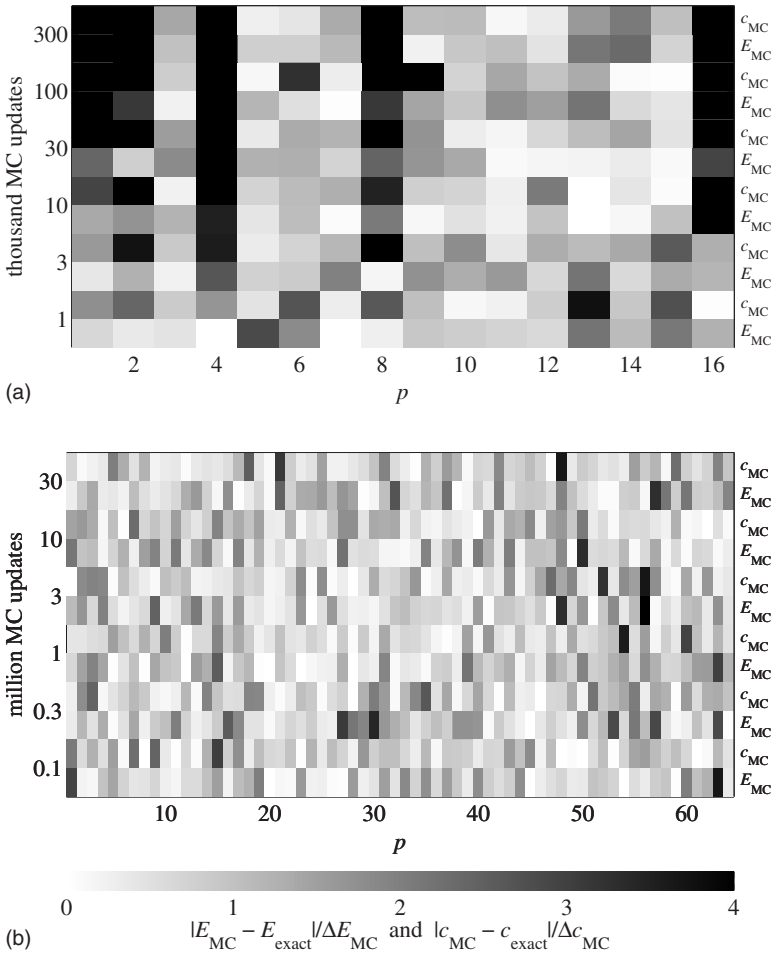

FIG. 5. Results of Monte Carlo simulations of the two-dimensional Ising model at the critical temperature using the Wolff cluster flipping algorithm and the generator r250 (top) and trng::mrg3s (bottom). Only each pth random number of the sequence was used during the simulations. See the text for details.

For the S_N test N random walkers are considered. They move simultaneously and independently on the one-dimensional lattice, i.e., on the set of all integers. At each time step t each walker moves one step to the left or to the right with equal probability. For large times t the expected number $S_N(t)$ of sites that have been visited by at least one walker reaches the asymptotic form $S_N(t) \sim t^\gamma$ with $\gamma = 1/2$. In the S_N test the exponent γ is measured and compared to the asymptotic value $\gamma = 1/2$.

Let p be the number of parallel streams or walkers. The motion of each random walker is determined by a leapfrog stream $t_{i,j}$ based on the generators from Table I. The exponent γ is determined for up to 16 simultaneous random walkers as a function of time t. After about 1000 time steps the exponent has converged and fluctuates around its mean value.

Figure 6 shows some typical plots of the exponent γ against time t for generator trng::yarn3. The exponent γ was determined by averaging over 10^8 random walks. Results for the other generators in Table I look similar and are omitted here. Figure 7 presents the asymptotic values of γ as a function of the number of walkers for generator trng::mrg5. The mean value of γ averaged over the interval $1500 < t < 2500$ is plotted and the error bars indicate the maximum and the minimum value of γ in $1500 < t < 2500$. Almost all numerical results are in good agreement with the theoretical prediction $\gamma = 1/2$ and for other generators we get qualitatively the same results. Therefore we conclude that there are no interstream correlations detectable by the S_N test.

C. Birthday spacings test

The birthday spacings test is an empirical test procedure that was proposed by Marsaglia [5,54]. It is specifically designed to detect the hyperplane structures of LFSR sequences [39].

For the birthday spacings test N days $0 \leqslant d_i < M$ are uniformly and randomly chosen in a year of M days. After sorting these independent identically distributed random numbers into nondecreasing order $b_{(1)}, b_{(2)}, \ldots, b_{(N)}$, N spacings

$$s_1 = b_{(2)} - b_{(1)},$$

$$s_2 = b_{(3)} - b_{(2)},$$

$$\vdots$$

$$s_N = b_{(1)} + M - b_{(N)}$$

are defined. The birthday spacings test examines the distribution of these spacings by sorting them into nondecreasing order $s_{(1)}, s_{(2)}, \ldots, s_{(N)}$ and counts the number R of equal spacings. More precisely, R is the number of indices j such that $1 \leqslant j < N$ and $s_{(j)} = s_{(j+1)}$. The test statistics R is a random number with distribution

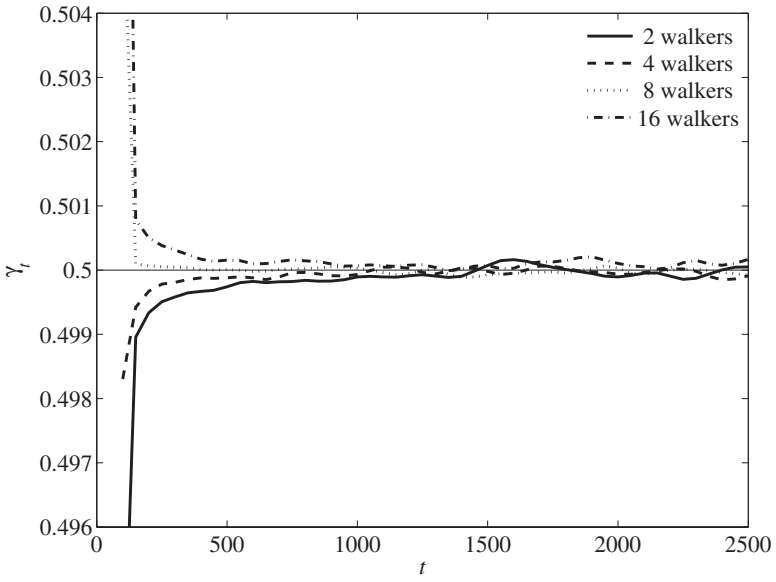

FIG. 6. The number of sites visited by N random walkers after t time steps is $S_N(t) \sim t^{\gamma}$. The plot shows the exponent γ against time for generator trng ::yarn3.

$$p_R(r) = \frac{[N^3/(4M)]^r}{e^{N^3/(4M)} r!} + O\left(\frac{1}{N}\right). \tag{23}$$

For $N \to \infty$ and $M \to \infty$ such that $N^3/(4M) \to \lambda = \text{const}$, $p_R(r)$ converges to a Poisson distribution with mean λ. After repeating the birthday spacings test several times one can apply a chi-square test to compare the empirical distribution of R to the correct distribution (23). If the p value of the chi-square test is very close to one or zero, the birthday spacings test has to be considered failed [5].

In order to make the birthday spacings test sensitive to the hyperplane structure of linear sequences, the birthdays are arranged in a d-dimensional cube of length l, i.e., $M = l^d$. Each day b_i is determined by a d tuple of consecutive PRNs $(r_{di}, r_{di+1}, \ldots, r_{di+d-1})$,

$$b_i = \sum_{j=0}^{d-1} \left\lfloor \frac{l r_{di+j}}{m} \right\rfloor l^j. \tag{24}$$

This bijective mapping transforms points in a d-dimensional space $[0,1,\ldots,l-1]^d$ to the linear space $[0,1,\ldots,M-1]$ and points on regular hyperplanes are transformed into regular spacings between points in $[0,1,\ldots,M-1]$.

To demonstrate the failure of LFSR sequences in the birthday spacings test we applied the test to the LFSR sequence

$$r_i = 17\,384 r_{i-1} + 12\,391 r_{i-2} \bmod 65\,521, \tag{25}$$

and its YARN counterpart

$$r_i = \begin{cases} 20\,009^{q_i} \bmod 65\,521 & \text{if } q_i > 0 \\ 0 & \text{if } q_i = 0, \end{cases} \tag{26}$$

with

$$q_i = 17\,384 q_{i-1} + 12\,391 q_{i-2} \bmod 65\,521,$$

with the test parameters $\lambda = N^3/(4M) \approx 1$ and $d = 6$. Beyond a certain value of M the LFSR sequence starts to fail the birthday spacings test due to its hyperplane structure (see Fig 8). The hyperplane structure of LFSR sequences give rise to birthday spacings that are much more regular than randomly chosen birthdays. As a consequence, we observe a value R that is too large (see the top of Fig. 8).

The nonlinear transformation (15) in the YARN sequences destroys the hyperplane structure. In fact, even those YARN sequences whose underlying linear part fails the birthday spacings test, passes the same test with flying colors (Fig. 8). YARN generators start to fail the birthday spacings test only

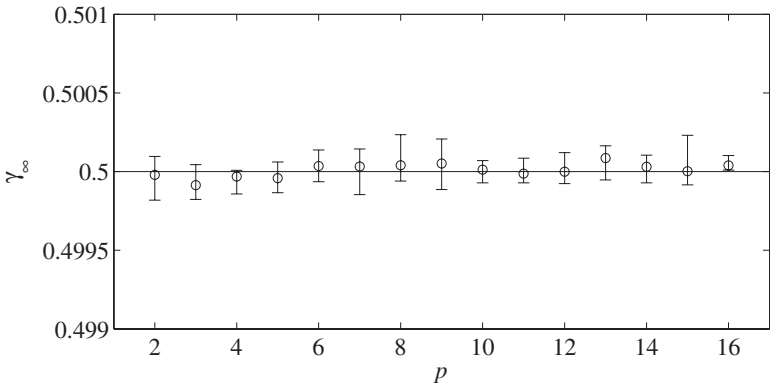

FIG. 7. Results of the S_N test for generator trng::mrg5; corresponding results for other generators in Table I look similar. See the text for details.

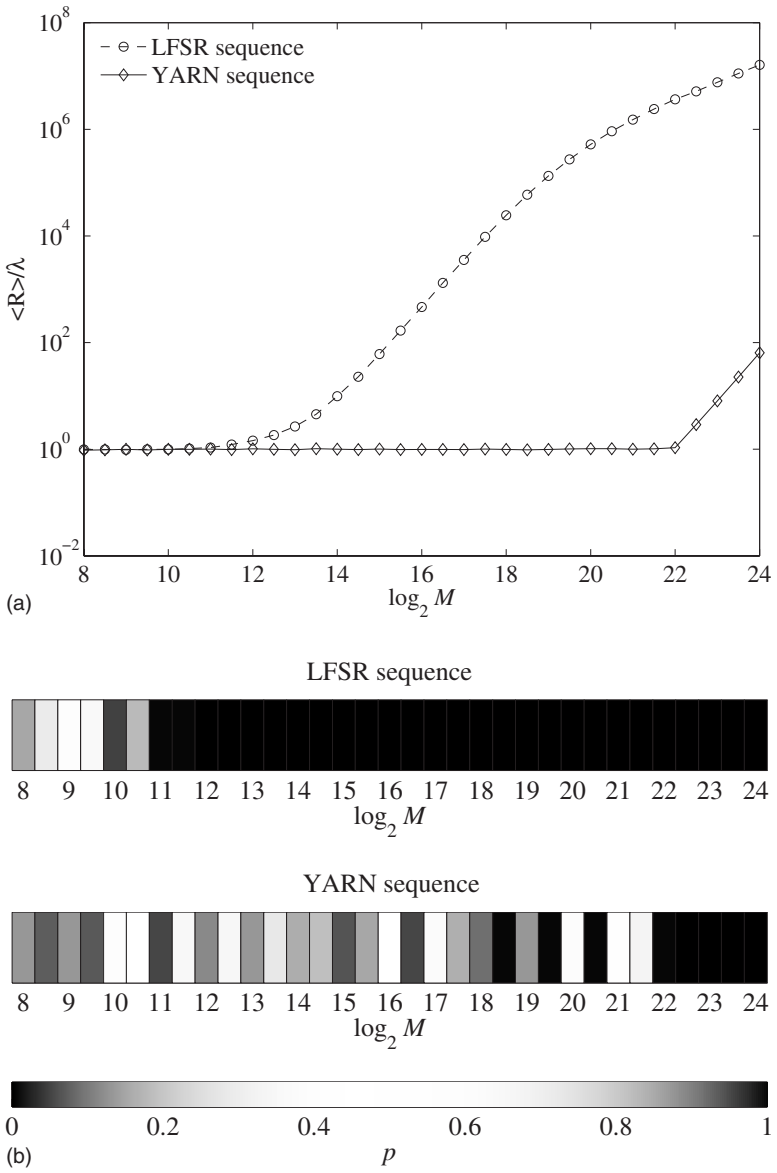

FIG. 8. Due to their lattice structure, LFSR generators fail the birthday spacings test. The nonlinear mapping of YARN sequences is an effective technique to destroy these lattice structures. Top: mean number $\langle R\rangle$ of equal spacings between M randomly chosen birthdays. For a good PRNG $\langle R\rangle/\lambda=1$ is expected. Bottom: gray coded p value of a chi-square test for the distribution of R.

for trivial reasons, i.e., when M gets close to the period of the generator. For the experiment shown in Fig. 8 we averaged over 5000 realizations of the M birthdays, and each birthday was determined by $d=6$ random numbers. Then we expect that the YARN generator starts to fail if $M6\times 5000 \gtrapprox 65\,521^2-1$, i.e., $M \gtrapprox 2^{17}$.

Of course the period of the PRNGs (25) and (26) is artificially small and the birthday spacings test was carried out with these particular PRNGs to make the effect of hyperplane structures and its elimination by delinearization as explicit as possible. "Industrially sized" LFSR and YARN generators like those in Table I share the same structural features, however. So the results for PRNGs (25) and (26) can be assumed to present generic results for LFSR and YARN sequences, respectively.

VII. CONCLUSIONS

We have reviewed the problem of generating pseudorandom numbers in parallel environments. Theoretical and practical considerations suggest to focus on linear recurrences in prime number fields, also known as LFSR sequences. These sequences can be efficiently parallelized by block splitting and leapfrogging, and both methods enable Monte Carlo simulations to play fair, i.e., to yield results that are independent of the degree of parallelization. We also discussed theoretical and practical criteria for the choice of parameters in LFSR sequences. We then introduced YARN sequences, which are derived from LFSR sequences by a bijective nonlinear mapping. A YARN sequence inherits the pseudonoise properties from its underlying LFSR sequence, but its linear complexity is that of a true random sequence. YARN sequences share all the advantages of LFSR sequences, but they pass all tests that LFSR sequences tend to fail due to their low linear complexity.

All our results have been incorporated into TRNG, a publicly available software library of portable, parallelizable pseudorandom number generators. TRNG complies with the proposal for the next revision of the ISO C++ standard [48].

ACKNOWLEDGMENTS

This work was sponsored by the European Community's FP6 Information Society Technologies program under Contract No. IST-001935, EVERGROW, and by the German Science Council DFG under Grant No. ME2044/1-1.

APPENDIX: SOPHIE-GERMAIN PRIME MODULI

The maximal period of a LFSR sequence (3) over the prime field $\mathbb{F}_m$ is given by m^n-1. We are looking for primes m such that m^n-1 has for a fixed n a small number of prime factors.

For $m>2$ the factor $m-1$ is even and the smallest possible number of prime factors of m^n-1 is three.

$$m^n - 1 = 2\frac{m-1}{2}(1 + m + m^2 + \cdots + m^{m-1}). \quad \text{(A1)}$$

If $(m-1)/2$ is also prime, m is called a Sophie-Germain prime or safe prime. For $m=2$ there exist values for n such that 2^n-1 is also prime. Primes of form 2^n-1 are called Mersenne primes, e.g., 2^2-1 and $2^{86\,243}-1$ are prime.

If n is composite and $m>2$ the period T is a product of at least four prime factors.

$$\begin{aligned} m^{kl} - 1 &= (m^k - 1)[(m^k)^{l-1} + (m^k)^{l-2} + \cdots + 1] \\ &= (m-1)(m^{k-1} + m^{k-2} + \cdots + 1) \\ &\quad \times [(m^k)^{l-1} + (m^k)^{l-2} + \cdots + 1]. \end{aligned} \quad \text{(A2)}$$

Actually, the number of prime factors for composite n is even larger. The factorization of m^n-1 is related to the factorization of the polynomial

$$f_n(x) = x^n - 1 \quad \text{(A3)}$$

over $\mathbb{Z}$. This polynomial can be factored into cyclotomic polynomials $\Phi_k(x)$ [55],

$$f_n(x) = x^n - 1 = \prod_{d|n} \Phi_d(x). \quad \text{(A4)}$$

The coefficients of the cyclotomic polynomials are all in $\mathbb{Z}$; so the number of prime factors of the factorization of m^n-1 is bounded from below by the number of cyclotomic polynomials in the factorization of x^n-1. This number equals the number of natural numbers that divide n. If n is prime then $f_n(x)$ is just the product $\Phi_1(x)\Phi_n(x)$.

If m is a prime larger than two, from $\Phi_1(x)=x-1$ it follows that m^n-1 can be factorized into at least three factors, namely,

$$m^n - 1 = 2\frac{m-1}{2} \prod_{d>1;d|n} \Phi_d(m). \quad \text{(A5)}$$

The period of a maximal period LFSR sequence with linear complexity n over $\mathbb{F}_m$ is a product of exactly three factors, if and only if m is a Sophie-Germain prime, n is prime, and $\Phi_n(m)$ is prime also. Note, $\Phi_2(m)=m+1$ is never prime, if m is an odd prime. Let us investigate some special cases in more detail.

TABLE III. A collection of Sophie-Germain primes m for which m^n-1 has a minimal number of prime factors.

n	m
1	$2^{31}-525$
	$2^{31}-69$
	$2^{63}-5781$
	$2^{63}-4569$
2	$2^{31}-37\,485$
	$2^{31}-2085$
	$2^{63}-927\,861$
	$2^{63}-156\,981$
3	$2^{31}-43\,725$
	$2^{31}-21\,069$
	$2^{63}-275\,025$
	$2^{63}-21\,129$
4	$2^{31}-305\,829$
	$2^{31}-119\,565$
	$2^{63}-3\,228\,621$
	$2^{63}-156\,981$
5	$2^{31}-46\,365$
	$2^{31}-22\,641$
	$2^{63}-594\,981$
	$2^{63}-19\,581$
6	$2^{31}-4\,398\,621$
	$2^{31}-1\,120\,941$
	$2^{63}-122\,358\,381$
	$2^{63}-29\,342\,085$
7	$2^{31}-50\,949$
	$2^{31}-6489$
	$2^{63}-92\,181$
	$2^{63}-52\,425$

Case $n=2$. The period m^2-1 is a product of $2^3\times 3$ and at least two other factors. Using the sieve of Eratosthenes modulo 12 it can be shown, that each large enough prime m can be written as $m=12k+c$, where k and c are integers such that $\gcd(12,c)=1$. Factoring the period $m^2-1=(12k+c)^2-1$ we find

$$\begin{aligned} (12k+1)^2 - 1 &= 2^3 \times 3k(6k+1) \\ (12k+5)^2 - 1 &= 2^3 \times 3(2k+1)(3k+1) \\ (12k+7)^2 - 1 &= 2^3 \times 3(3k+2)(2k+1) \\ (12k+11)^2 - 1 &= 2^3 \times 3(k+1)(6k+5). \end{aligned} \quad \text{(A6)}$$

Case $n=4$. The period m^4-1 is a product of $2^4\times 3\times 5$ and at least three other factors. Using the sieve of Eratosthenes modulo 60 it can be shown, that each large enough prime m can be written as $m=60k+c$, where k and c are integers such that $\gcd(60,c)=1$. Factoring the period $m^4-1=(60k+c)^2-1$ we find

$$
\begin{aligned}
(60k+1)^4 - 1 &= 2^4 \times 3 \times 5k(30k+1)(1800k^2+60k+1) \\
(60k+7)^4 - 1 &= 2^4 \times 3 \times 5(10k+1)(15k+2) \\
&\quad \times (360k^2+84k+5) \\
(60k+11)^4 - 1 &= 2^4 \times 3 \times 5(5k+1)(6k+1) \\
&\quad \times (1800k^2+660k+61) \\
(60k+13)^4 - 1 &= 2^4 \times 3 \times 5(5k+1)(30k+7) \\
&\quad \times (360k^2+156k+17);
\end{aligned}
\tag{A7}
$$

and so on ….

Case $n=6$. This case is similar to the $n=2$ and $n=4$ cases. Applying the sieve of Erastosthenes modulo 84 it can be shown that m^6-1 is a product of $2^3\times 3^2\times 7$ and at least four other factors.

Cases $n=3$, $n=5$, $n=7$. Here n is prime, and therefore the period m^n-1 has at least three factors. The number of factors of m^n-1 will not exceed three, if m is a Sophie-Germain prime and $\Phi_n(m)$ is prime, too.

In Table III we present a collection of Sophie-Germain primes m, for which m^n-1 has a minimal number of prime factors. For n prime an extended table can be found in [56]. If n is prime, its factorization can be found by Eq. (A4), and if $n=2$ or $n=4$ by Eqs. (A6) and (A7), respectively. All these primes are good candidates for moduli of LFSR sequences as PRNGs in parallel applications. Note that the knowledge of the factorization of m^n-1 is essential for an efficient test of the primitivity of characteristic polynomials.

[1] B. Hayes, Am. Sci. **89**, 300 (2001).
[2] P. D. Coddington, NHSE Review (1996), http://nhse.cs.rice.edu/NHSEreview/RNG/
[3] S. L. Anderson, SIAM Rev. **32**, 221 (1990).
[4] G. Marsaglia, Proc. Natl. Acad. Sci. U.S.A. **61**, 25 (1968).
[5] D. E. Knuth, *The Art of Computer Programming*, 3rd ed. (Addison Wesley Professional, Reading, MA, 1998), Vol. 2.
[6] P. L'Ecuyer, in *Handbook of Computational Statistics*, edited by J. E. Gentle, W. Härdle, and Y. Mori (Springer, New York, 2004).
[7] D. H. Lehmer, *Proceedings of the 2nd Symposium on Large-Scale Digital Calculating Machinery, Cambridge, MA* (Harvard University Press, Cambridge, MA, 1951), pp. 141–146.
[8] D. E. Knuth, *The Art of Computer Programming*, 1st ed. (Addison Wesley Professional, Reading, MA, 1969), Vol. 2.
[9] S. Tezuka, *Uniform Random Numbers: Theory and Practice* (Kluwer Academic, Boston, 1995).
[10] P. L'Ecuyer, F. Blouin, and R. Couture, ACM Trans. Model. Comput. Simul. **3**, 87 (1993).
[11] D. Jungnickel, *Finite Fields: Structure and Arithmetics* (Bibliographisches Institut, Mannheim, 1993).
[12] J. Massey, IEEE Trans. Inf. Theory **15**, 122 (1969).
[13] A. J. Menezes, *Handbook of Applied Cryptography*, The CRC Press Series on Discrete Mathematics and its Applications (CRC, Boca Raton, FL, 1996).
[14] M. Matsumoto and T. Nishimura, ACM Trans. Model. Comput. Simul. **8**, 3 (1998).
[15] S. W. Golomb, *Shift Register Sequences*, revised ed. (Aegan Park, Laguna Hills, CA, 1982).
[16] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and their Applications*, 2nd ed. (Cambridge University Press, Cambridge, England, 1994).
[17] R. Lidl and H. Niederreiter, *Encyclopedia of Mathematics and its Applications*, 2nd ed. (Cambridge University Press, Cambridge, England, 1997), Vol. 20.
[18] J. Fillmore and M. Marx, SIAM Rev. **10**, 342 (1968).
[19] N. Zierler, J. Soc. Ind. Appl. Math. **7**, 31 (1959).
[20] Z.-X. Wan, *Lectures on Finite Fields and Galois Rings* (World Scientific, Singapore, 2003).
[21] A. Compagner, Am. J. Phys. **59**, 700 (1991).
[22] A. Compagner, J. Stat. Phys. **63**, 883 (1991).
[23] O. E. Percus and M. H. Kalos, J. Parallel Distrib. Comput. **6**, 477 (1989).
[24] M. Mascagni, Parallel Comput. **24**, 923 (1998).
[25] A. D. Matteis and S. Pagnutti, Parallel Comput. **13**, 193 (1990).
[26] A. D. Matteis and S. Pagnutti, Parallel Comput. **14**, 207 (1990).
[27] J. Eichenauer-Herrmann and H. Grothe, Numer. Math. **56**, 609 (1989).
[28] P. L'Ecuyer, Commun. ACM **33**, 85 (1990).
[29] S. Kirkpatrick and E. P. Stoll, J. Comput. Phys. **40**, 517 (1981).
[30] R. M. Ziff, Comput. Phys. **12**, 385 (1998).
[31] P. Grassberger, Phys. Lett. A **181**, 43 (1993).
[32] L. N. Shchur, J. R. Heringa, and H. W. J. Blöte, Physica A **241**, 579 (1997).
[33] A. M. Ferrenberg, D. P. Landau, and Y. J. Wong, Phys. Rev. Lett. **69**, 3382 (1992).
[34] H. Bauke and S. Mertens, J. Stat. Phys. **114**, 1149 (2004).
[35] S. Mertens and H. Bauke, Phys. Rev. E **69**, 055702(R) (2004).
[36] P. L'Ecuyer, Oper. Res. **47**, 159 (1999).
[37] A. Grube, Z. Angew. Math. Mech. **53**, T223 (1973).
[38] P. L'Ecuyer, ACM Trans. Model. Comput. Simul. **3**, 87 (1993).
[39] P. L'Ecuyer, *Proceedings of the 2001 Winter Simulation Conference* (IEEE Press, Washington, DC, 2001), pp. 95–105.
[40] P. L'Ecuyer and P. Hellekalek, *Random and Quasi-Random Point Sets*, Lecture Notes in Statistics Vol. 138 (Springer, New York, 1998), pp. 223–266.
[41] L. Schrage, ACM Trans. Math. Softw. **5**, 132 (1979).
[42] W. H. Payne, J. R. Rabung, and T. P. Bogyo, Commun. ACM **12**, 85 (1969).
[43] M. Mascagni and H. Chi, Parallel Comput. **30**, 1217 (2004).
[44] *Tina's Random Number Generator Library*, http://tina.nat.uni-magdeburg.de/trng/
[45] *GNU Scientific Library*, http://www.gnu.org/software/gsl/
[46] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flan-

nery, *Numerical Recipes in C*, 2nd ed. (Cambridge University Press, Cambridge, England, 1992).
[47] M. Lüscher, Comput. Phys. Commun. **79**, 100 (1994).
[48] W. E. Brown, M. Fischler, J. Kowalkowski, and M. Paterno, *Random Number Generation in C++0X: A Comprehensive Proposal, version 2* (2006); http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2032.pdf
[49] U. Wolff, Phys. Rev. Lett. **62**, 361 (1989).
[50] A. E. Ferdinand and M. E. Fisher, Phys. Rev. **185**, 832 (1969).
[51] D. Wang and A. Compagner, Math. Comput. **60**, 363 (1993).
[52] M. Matsumoto and Y. Kurita, ACM Trans. Model. Comput. Simul. **6**, 99 (1996).
[53] I. Vattulainen, Phys. Rev. E **59**, 7200 (1999).
[54] G. Marsaglia, in *Computer Science and Statistics: Proceedings of the 16th Symposium on the Interface, Atlanta, Georgia, 1984*, edited by L. Billard (North Holland, Amsterdam, 1985).
[55] Z.-X. Wan, *Lectures on Finite Fields and Galois Rings* (World Scientific, Singapore, 2003).
[56] P. L'Ecuyer, Oper. Res. **47**, 159 (1999).