Parallel streams of pseudorandom numbers for Monte Carlo simulations: Using most reliable algorithms and applying parallelism of modern CPUs and GPUs

L.Yu. Barash

Landau Institute for Theoretical Physics RAS

November 30 2012, Universität Leipzig, CompPhys12

Outline

1. Methods for pseudorandom number generation
2. Methods for generation of parallel streams of pseudorandom numbers
3. Fast algorithms for skipping over terms in the generators and for initialization of independent sequences
4. Library PRAND: single-threaded parallel algorithms for pseudorandom number generators and their usage
5. Library PRAND: multithreaded parallel algorithms for pseudorandom number generators. Speeding up the calculations. Efficiency of using GPU capabilities by multi-threaded routines in PRAND library.
6. Multi-GPU multidimensional Monte-Carlo integration with VEGAS algorithm. Efficiency of parallelization.
7. Comparing with other PRND libraries which allow to generate parallel pseudorandom number streams.

Requirements for a PRNG and its implementation in a subroutine library

1. Statistical robustness
2. Unpredictability
3. Long period
4. Efficiency
5. Theoretical support
6. Repeatability
7. Portability
8. Skipping over terms (jumping ahead)
9. Proper initialization

Major methods for random number generation

1. Linear Congruential Generator (LCG)
2. Generalized feedback shift register (GFSR)

Examples of modern modifications and generalizations to the LCG and GFSR methods:

- Mersenne Twister (Matsumoto, Tishimura, 1998), WELL (2006)
- Combined LCG generators (L'Ecuyer, 1999)
- Combined Tausworthe generators (L'Ecuyer, 1996; L'Ecuyer, 1999).

Linear congruential: $x_{n+1} = (ax_n + c)(\text{mod } M)$.

Shift register sequence: $x_n = (a_1 x_{n-1} + \cdots + a_k x_{n-k})(\text{mod } 2)$,

$$u_n = \sum_{i=1}^{L} x_{ns+i-1} 2^{-i}, \qquad P(z) = z^k - a_1 z^{k-1} - \cdots - a_k.$$

Mersenne Twister: $\mathbf{x}_{k+n} := \mathbf{x}_{k+m} \oplus (\mathbf{x}_k^u | \mathbf{x}_{k+1}^l)A$.

Combined linear congruential MRG32K3A:

$a = 1403580, \ b = -810728, \quad c = 527612, \quad d = -1370589,$

$m_1 = 2^{32} - 209, \ \ m_2 = 2^{32} - 22853$

$$x_n = (ax_{n-2} + bx_{n-3})(\text{mod } m_1)$$
$$y_n = (cy_{n-1} + dy_{n-3})(\text{mod } m_2),$$
$$z_n = (x_n + y_n)(\text{mod } m_1).$$

Combined Tausworthe generator LFSR113 is a combination of four shift registers:

- $x_n = x_{n-31} \oplus x_{n-25}, \quad s = 18,$
- $x_n = x_{n-29} \oplus x_{n-27}, \quad s = 2,$
- $x_n = x_{n-28} \oplus x_{n-15}, \quad s = 7,$
- $x_n = x_{n-25} \oplus x_{n-22}, \quad s = 13.$

Method for pseudorandom number generation based on parallel evolution of toral automorphisms

L.Yu. B., Europhysics Letters (EPL) 95, 10003 (2011).
L.Yu. B., L.N. Shchur, Computer Physics Communications, 182 (7), 1518-1527 (2011).
L.Yu. B., L.N. Shchur, Phys.Rev. E 73 , 036701 (2006).

The main highlights of the method are:

Ensemble of transformations: the ensemble of MRG-transformations is used and not the single system

Hidden variables: only part of the generated information goes to the output of the generator. This helps to suppress correlations, complicates deciphering and results in other good properties.

Period length: The period equals p^2-1 if one uses the mesh $p \times p$, where p is prime integer. The period equals $3 \cdot 2^{m-2}$ for the mesh $2^m \times 2^m$
The period can be as large as required.

Description of the method: the basic unit of the generator

Set of states: $R = L^s$, $\qquad L = \{0, 1, \ldots, g-1\} \times \{0, 1, \ldots, g-1\}$

In practice , $g = 2^t$ or $g = p$ or $g = p \cdot 2^t$, (p is a prime integer).

The transition function of the generator is defined as an action of the map

$$\begin{pmatrix} x_i^{(n)} \\ y_i^{(n)} \end{pmatrix} = M \begin{pmatrix} x_i^{(n-1)} \\ y_i^{(n-1)} \end{pmatrix} \mod g,$$

where s points $(i = 0, 1, \ldots, s-1)$ are transformed at each step.

Equivalent description with the recurrence relation:

$$x^{(n)} = kx^{(n-1)} - qx^{(n-2)} (\mathrm{mod}\ g)$$
$$y^{(n)} = ky^{(n-1)} - qy^{(n-2)} (\mathrm{mod}\ g)$$

Here $k = \mathrm{Tr}\ (\mathrm{M}),\ \mathrm{q} = \det\ \mathrm{M}$

Characteristic polynomial of the recurrence relation:

$$f(x) = x^2 - kx + q.$$

Description of the method: the basic unit of the generator

Let $\alpha_i^{(n)}$ denote the high-order bit $x_i^{(n)}$: $\alpha_i^{(n)} = \lfloor 2x_i^{(n)}/g \rfloor$.

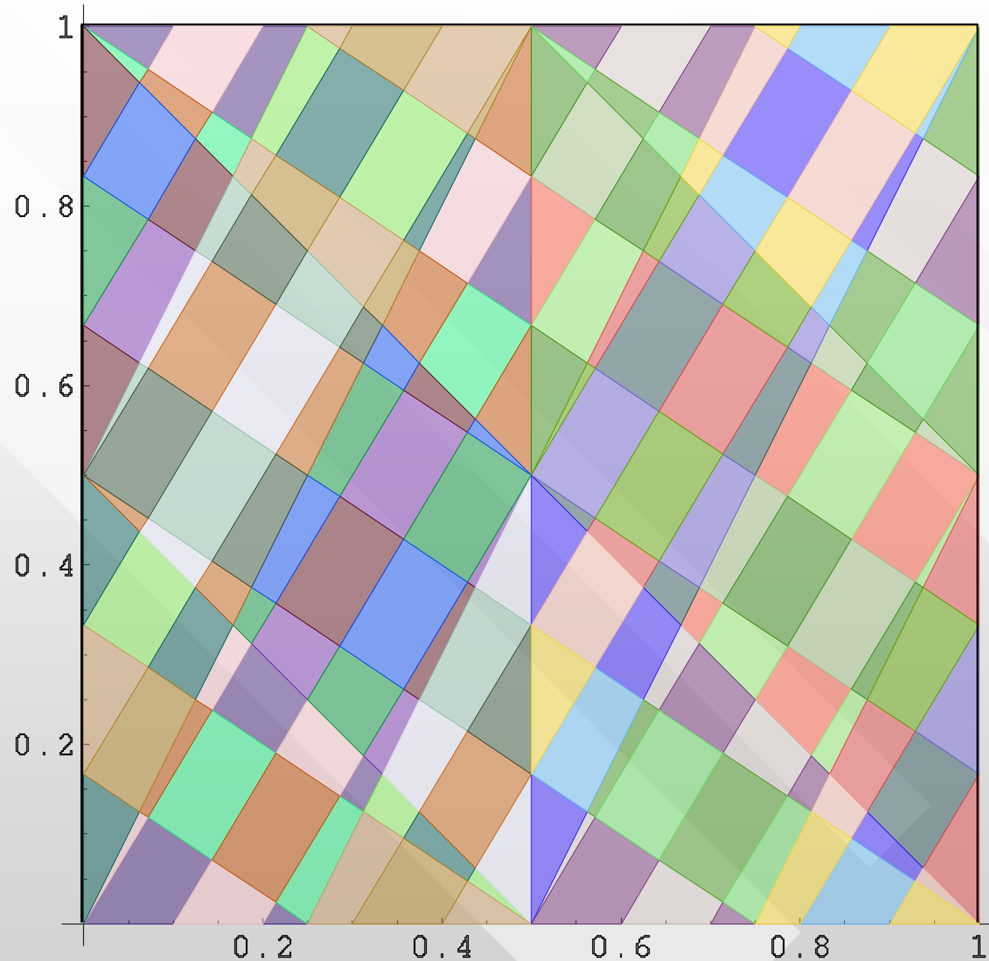
The output function of the generator $G : L^s \to \{0, 1, \ldots, 2^s - 1\}$

is defined as follows: $a_n = \sum_{i=0}^{s-1} \alpha_i^{(n)} \cdot 2^i$.

In other words, a_n – is an s-bit integer consisting of the bits $\alpha_0^{(n)}, \alpha_1^{(n)}, \ldots, \alpha_{s-1}^{(n)}$. In the case $g = 2^m$, a_n contains precisely the high-order bits of the integers $x_0^{(n)}, x_1^{(n)}, \ldots, x_{s-1}^{(n)}$.

The constructed RNG has much hidden information.
Indeed, $s(m-1)$ bits of each state are the hidden variables;
these are the bits that are not involved in constructing
of the output value a_n.

The regions on the torus and five-bit sequences generated by the toral automorphism



L.Yu. B., Europhysics Letters (EPL) 95, 10003 (2011).
L.Yu. B., L.N. Shchur, Phys.Rev. E 73 , 036701 (2006).

Parameters of the new generators

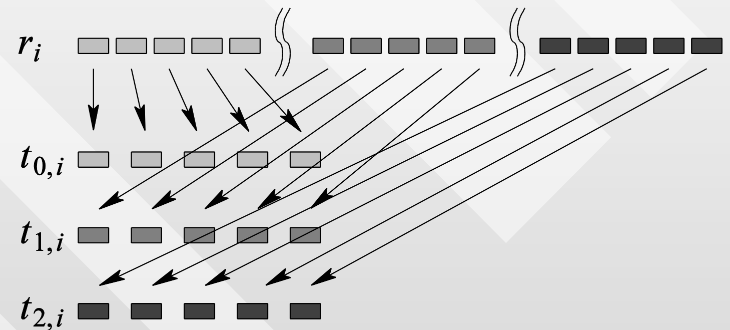
Generator	g	k	q	v	Period
GM19	$2^{19}-1$	15	28	1	$2.7\cdot 10^{11}$
GM31	$2^{31}-1$	11	14	1	$4.6\cdot 10^{18}$
GM61	$2^{61}-1$	24	74	1	$5.3\cdot 10^{36}$
GM29.1-SSE	$2^{29}-3$	4	2	1	$=2.8\cdot 10^{17}$
GM55.4-SSE	$16(2^{51}-129)$	256	176	4	$\geq 5.1\cdot 10^{30}$
GQ58.1-SSE	$2^{29}(2^{29}-3)$	8	48	1	$\geq 2.8\cdot 10^{17}$
GQ58.3-SSE	$2^{29}(2^{29}-3)$	8	48	3	$\geq 2.8\cdot 10^{17}$
GQ58.4-SSE	$2^{29}(2^{29}-3)$	8	48	4	$\geq 2.8\cdot 10^{17}$

Results of statistical testing with the TestU01 batteries of tests.

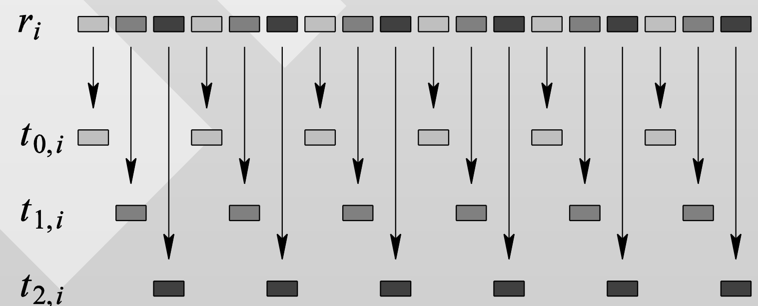
Generator	SmallCrush	Diehard	Crush	Bigcrush
MRG32k3a	0, 0, 0	0, 0, 0	0, 0, 0	0, 0, 0
LFSR113	0, 0, 0	1, 0, 0	6, 6, 6	6, 6, 6
MT19937	0, 0, 0	0, 0, 0	2, 2, 2	2, 2, 2
GM29-SSE	0, 0, 0	0, 0, 0	0, 0, 0	0, 0, 0
GM55.4-SSE	0, 0, 0	0, 0, 0	0, 0, 0	0, 0, 0
GQ58.1-SSE	0, 0, 0	0, 0, 0	0, 0, 0	0, 0, 0
GQ58.3-SSE	0, 0, 0	0, 0, 0	0, 0, 0	0, 0, 0
GQ58.4-SSE	0, 0, 0	0, 0, 0	0, 0, 0	0, 0, 0

Methods for generation of parallel streams of pseudorandom numbers

1. Random seeding
2. Parameterization
3. Block splitting



4. Leapfrog



(figures are taken from H. Bauke, S. Mertens, Phys. Rev. E 75, 066701 (2007))

Efficient algorithms to jump ahead and to initialize subsequences for the chosen generators

1. Jumping ahead for the generators GM19, GM31, GM61, GM29.1, GM55.4, GQ58.1, GQ58.3, GQ58.4, where the state transforms with the recurrence relation $x^{(n)} = kx^{(n-1)} - qx^{(n-2)} \pmod{g}$.

   One can show that the following equation holds

$$x^{(2n)} = k_n x^{(n)} - q_n x^{(0)} \pmod{g},$$

where $k_1 = k; \quad q_1 = q; \quad k_{2n} = k_n^2 - 2q_n \pmod{g}; \quad q_{2n} = q_n^2 \pmod{g}$.

This allows to quickly calculate the coefficients which are necessary to jump ahead 2^i elements for any integer i. In order to efficiently skip over a sequence of arbitrary length n one has to skip over terms of length $2^{i_0}, \ldots, 2^{i_m}$, where $n = 2^{i_0} + 2^{i_1} + \cdots + 2^{i_m}$ is the binary representation of integer n. Also, one can show that

$$k_0 = 2; \quad k_1 = k; \quad k_{n+1} = kk_n - qk_{n-1} \pmod{g}; \quad q_n = q^n \pmod{g}.$$

2. Jumping ahead for the generator MRG32K3A.

Consider $\mathbf{X}_n = \begin{pmatrix} x_n \\ x_{n-1} \\ x_{n-2} \end{pmatrix}$, $\mathbf{Y}_n = \begin{pmatrix} y_n \\ y_{n-1} \\ y_{n-2} \end{pmatrix}$, $A = \begin{pmatrix} 0 & a & b \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$, $B = \begin{pmatrix} c & 0 & d \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$.

This allows to write the state transformation function for MRG32K3A as

$$\begin{aligned} \mathbf{X}_{n+1} &= A\,\mathbf{X}_n (\mathrm{mod}\ m_1), \\ \mathbf{Y}_{n+1} &= B\,\mathbf{Y}_n (\mathrm{mod}\ m_2). \end{aligned}$$

The state of the generator is a pair of vectors $\mathbf{X}_n$, $\mathbf{Y}_n$. Therefore, in order to skip over n numbers, one has to find n-th power of matrices A and B.

Let $n = 2^{i_0} + 2^{i_1} + \cdots + 2^{i_m}$ be the binary representation of integer n, then $A^n = A^{2^{i_0}} A^{2^{i_1}} \ldots A^{2^{i_m}} (\mathrm{mod}\ m_1)$, $B^n = B^{2^{i_0}} B^{2^{i_1}} \ldots B^{2^{i_m}} (\mathrm{mod}\ m_2)$, i.e. jumping ahead of a block of length n can be carried out with $O(\log n)$ operations.

Function `MRG32K3A_init_sequence` allows to initialize up to 10^{19} independent parallel sequences of pseudorandom numbers, with length of each sequence up to 10^{38}.

3. Jumping ahead for the generator LFSR113.

Algorithm to jump ahead for the combined generator reduces to jumping ahead for every particular shift register sequence.

- $x_n = x_{n-31} \oplus x_{n-25}, \quad s = 18,$
- $x_n = x_{n-29} \oplus x_{n-27}, \quad s = 2,$
- $x_n = x_{n-28} \oplus x_{n-15}, \quad s = 7,$
- $x_n = x_{n-25} \oplus x_{n-22}, \quad s = 13.$

$$x_n = (x_{n-p} + x_{n-p+q}) (\mathrm{mod}\ 2), \qquad u_n = \sum_{l=1}^{32} x_{is+l-1} 2^{-l}$$

Then the following holds: $x_n = (x_{n-2^e p} + x_{n-2^e p+2^e q}) (\mathrm{mod}\ 2)$.

In order to jump ahead of a block of length 2^e, we calculate the bits $x_{2^n p}, x_{2^n (p+1)}, \ldots, x_{2^n (2p-1)}$ for $n = 0, 1, \ldots, e$ using the known values of the bits $x_0, x_1, \ldots, x_{31}$.

Function `LFSR113_init_sequence` allows to initialize $4 \cdot 10^{18}$ independent sequences of pseudorandom numbers, with length of each sequence up to 10^{10};

Function `LFSR113_init_long_sequence` allows to initialize $4 \cdot 10^9$ independent sequences of pseudorandom numbers, with length of each sequence up to $3 \cdot 10^{25}$.

4. Jumping ahead for the generator MT19937.

The generator MT19937 has a linear structure in its algorithm, which can be written as $\mathbf{Y}_{n+1} = A\mathbf{Y}_n \pmod 2$, where $\mathbf{Y}_n$ is the generator state, the size of matrix A is 19937×19937, calculation of A^n would be extremely slow and would require a lot of memory.

Another method [*]: for every $v \in \mathbb{N}$ the following relation holds

$$A^v\mathbf{Y}_0 = g_v(A)\mathbf{Y}_0 = a_k\mathbf{Y}_{k-1} + a_{k-1}\mathbf{Y}_{k-2} + \cdots + a_2\mathbf{Y}_1 + a_1\mathbf{Y}_0, \qquad (1)$$

where $k = 19937$, coefficients $a_i \in \{0, 1\}, i = 1, \ldots, k$, and polynomial $g_v(x) = a_k x^{k-1} + \cdots + a_2 x + a_1$ in the field $\mathbb{F}_2$ depend on v. [*] contains the method of calculation of the polynomial g_v for arbitraru fixed v.

Calculation of (1) can be carried out using massive parallelism of GPU in order to speed up the calculations.

[*] Haramoto et.al., INFORMS Journal on Computing 20 (3), 385-390 (2008).

The library PRAND: single-threaded parallel algorithms for PRNGs and their usage

Interface:

```
__device__ void RNG_init (RNG_state* state);

__device__ void RNG_init_sequence (RNG_state* state,
                                   unsigned SequenceNumber);

__device__ void RNG_SkipAhead (RNG_state* state,
                               unsigned long long offset);

__device__ unsigned int RNG_generate (RNG_state* state);

__device__ float RNG_generate_uniform_float (RNG_state* state);
```

Name of each actual fuction in PRAND library contains a particular PRNG name instead of `RNG`.

The library PRAND: multithreaded parallel algorithms for PRNGs in order to speed up the calculations

Interface:

```
void RNG_initialize (RNG_state* state);

void RNG_initialize_sequence  (RNG_state* state,
                               unsigned SequenceNumber);

void RNG_skip_ahead (RNG_State* state,
                     unsigned long long offset);

void RNG_generate_array (RNG_state* state,
                         unsigned int * out,
                         unsigned int length);

void RNG_generate_uniform_float_array (RNG_state* state,
                                      float * out,
                                      unsigned int length);
```

Name of each actual fuction in PRAND library contains a particular PRNG name instead of `RNG`.

Parallel algorithms for GM19,GM31,GM61,GM55.4,GQ58.1,GQ58.3,GQ58.4

Algorithm «one generator per *s* threads»

Each thread calculates the values of $x_i^{(n)} = kx_i^{(n-1)} - qx_i^{(n-2)} (\mathrm{mod}\ g)$ and $a_i^{(n)} = \lfloor 2^v x_i^{(n)}/g \rfloor \cdot 2^{iv}$ for one of the $i = 0, 1, \ldots, s-1$. Then we apply the sinchronization of threads inside a block which includes waiting for the completion of all calculations. Then, if the tread number is divisible by s, it sums all the calculated values of $a_i^{(n)}$, i.e., calculates the output value $a^{(n)} = \sum_{i=0}^{s-1} a_i^{(n)}$.

Every s threads generate a single sequence of length `length`. Each set of s threads has its own initial conditions $\left(x_0^{(i)}, x_1^{(i)}\right)$. The length of output sequence should be `length*N/s`, where `N` – total number of threads used in the calculation.

Prior to generating pseudorandom numbers, each thread carries out a corresponding jumping ahead to the start of it's sequence.

The number of threads in each block should be divisible by s.

Parallel algorithm for MT19937: one generator per 227 threads.

$$\mathbf{x}_{k+n} := \mathbf{x}_{k+m} \oplus (\mathbf{x}_k^u | \mathbf{x}_{k+1}^l) A.$$

The algorithm with a significant speed-up: using in parallel $n - m = 227$ threads in order to update the generator state. Indeed, in order to calculate the value of x_{k+n} one needs to know the values of x_k, x_{k+1}, x_{k+m}, therefore the calculation of $x_n, x_{n+1}, \ldots, x_{2n-m-1}$ can be carried out independently using the values of $x_0, x_1, \ldots, x_{n-1}$, and in order to calculate the value of x_{2n-m} one needs to know x_n.

For the further speed-up, we will use sets of 227 threads in order to fill in different sections of the output array. As it's first step each set of threads jumps ahead prior to staring the calculations in order to go to the beggining of it's own section. The size of each section of the array is selected experimentally, such that the time needed for jumping ahead would be only a small part of the time needed to generate pseudorandom numbers.

Efficiency of using GPU capabilities by multi-threaded routines in PRAND library

CPU/OC	Generator	Number of threads per block	Speed-up factor (time needed for multi-threaded calculation diveded by time needed for single-threaded calculation)	
			Multi-threaded = 1 block	Multi-threaded = 64 blocks
Intel Xeon E5630 2.53 GHz/ CentOS 6.1 («Lomonosov» supercomputer in Moscow State University)	GM19	1024	31.9	259.8
	GM31	1024	96.8	827.0
	GM61	1024	169.5	1424.0
	GM29.1	1024	30.8	252.1
	GM55.4	1024	263.9	1462.3
	GQ58.1	1024	99.8	828.1
	GQ58.3	1023	175.7	1410.9
	GQ58.4	1024	258.4	1469.3
	MRG32K3A	1024	82.4	133.8
	LFSR113	1024	17.2	20.1
	MT19937	227	72.4	191.8

Examples of using PRAND library in applications, where employing hybrid parallel computational systems results in significant economic effect

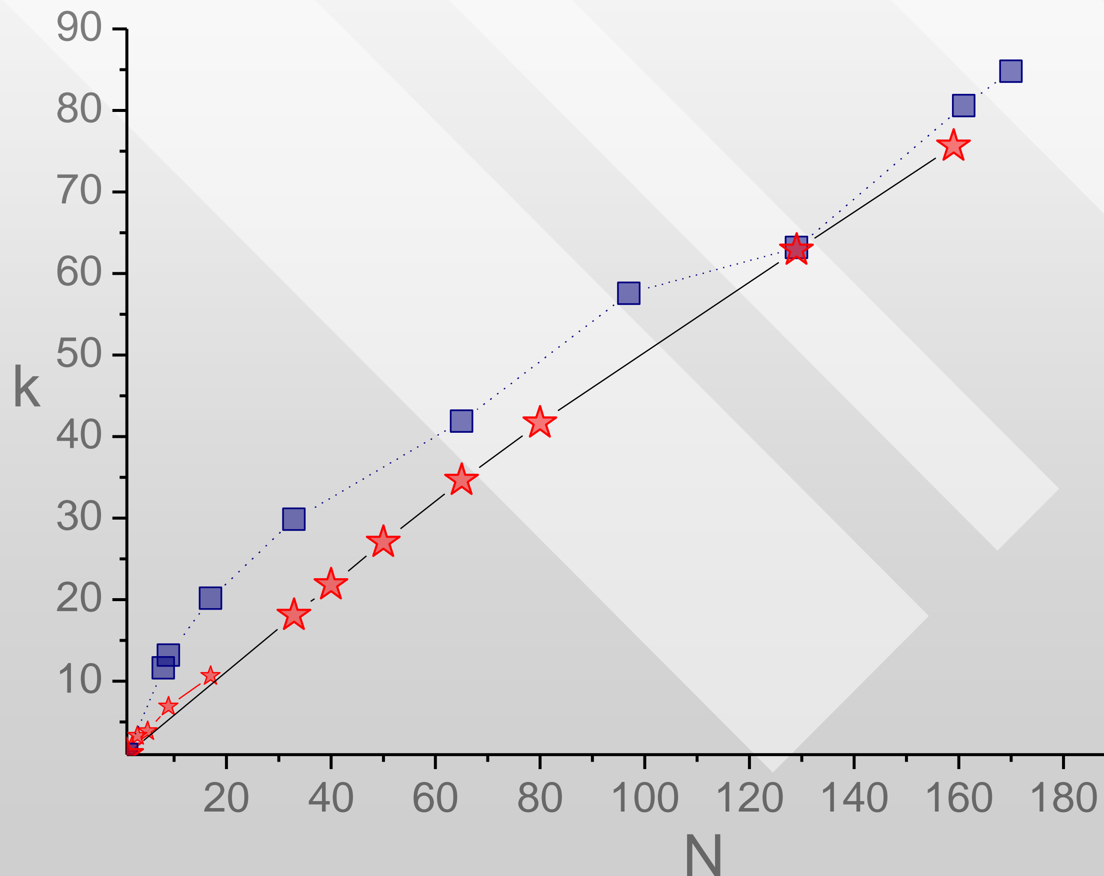
1. Growth of two-dimensional structures within the model of diffusion limited aggregation

2. Multidimensional numerical integration

3. Modeling of magnetism in materials using spin models

Multi-GPU multidimensional Monte-Carlo integration with VEGAS algorithm. **Efficiency of parallelization.**

Dependence of efficiency of parallelization k=T(1)/T(N) on N,
where T(N) is time needed for the calculation employing N CPU/GPU.
Stars denote calculations with «Lomonosov» supercomputer in Moscow State University.
Squares denote calculations with «K-100» supercomputer in
Keldysh Institute for Applied Mathematics, Russian Academy of Sciences.
Parallel software was developed using the technologies MPI and CUDA.



Other PRND libraries which allow to generate parallel pseudorandom number streams. Comparison.

1. Tina's Random Number Generation Library (TRNG), (Universität Magdeburg, Germany)

Among the generators presented in the library, jumping ahead and initialization of parallel streams are included only for linear congruential generators, for the MRG generators and for the YARN generators, i.e., only for insufficiently reliable PRNGs.

2. NAG Numerical Routines for GPU, (Oxford, UK) released in the end of 2011.

The library contains parallel algorithms for MRG32K3A and MT19937. The library is commercial, source code is not available.

3. cuRAND library, NVIDIA CUDA Toolkit 4.1, actual version is released in February 2012.

The library contains the generators MTGP Mersenne Twister, MRG32K3A, XORWOW. For MTGP Mersenne Twister initialization is carried out with the parameterization method, without convincing theoretical support. The source code is not available.

Other libraries for generation of pseudorandom numbers

GNU Scientific Library http://www.gnu.org/software/gsl/
18 standard PRNGs, including old ones and some modern ones,
standard realization for CPU with C language. No realizations for parallel calculations.

Intel MKL Library http://software.intel.com/en-us/articles/intel-mkl/
7 standard PRNGs, efficient realizations for CPU using SIMD , i.e. SSE processor instructions and 128-bit XMM-registers. No parallelization.

RNGSSELIB *L.Yu. B., L.N. Shchur, Computer Physics Communications, 182 (7), 1518-1527 (2011).*

6 modern and reliable PRNGs. Efficient realizations for CPU using SIMD , i.e. SSE processor instructions and 128-bit XMM-registers. The realizations are even more efficient, than Intel MKL. No parallelization.

SPRNG http://sprng.cs.fsu.edu/
Standard PRNGs (old ones). Parallelization with the parameterization method, without convincing theoretical support. (Florida State University, США)

Conclusion

1. Program library PRAND for parallel pseudorandom number generation is developed. It contains realizations for the generators, which are based on the parallel evolution of toral automorphisms (GM19, GM31, GM61, GM29.1, GM55.4, GQ58.1, GQ58.3, GQ58.4), also the generators MRG32K3A, LFSR113 and MT19937, i.e. the most reliable modern PRNG algorithms.

2. For each of the generators the library includes realizations for:
   - the ability to initialize up to 10^{19} independent streams with the block splitting method;
   - efficient versions for CPU which employ SIMD parallelism of modern CPUs and corresponding 128-bit XMM-registers and SSE-commands;
   - single-threaded realizations for GPU, which can be used in Monte Carlo calculations, where the computational threads and nodes can be used in any way chosen by an application;
   - multi-threaded realizations for GPU – employing many GPU threads in order to substantially speed up the calculations.

3. The multi-GPU realization for multidimensional Monte-Carlo integration algorithm VEGAS is developed. The performance of numerical integration is substantially higher when using a single GPU compared to using a single CPU. Also, the performance of numerical integration substantially and linearly increases with increasing the number of CPU/GPU nodes involved in the calculation.