Novel PRNG Schemes (EC-based and/or parallel) and their Automated Testing

M. Gonçalves[†], S. Konchenko[†], L. Trestioreanu [†],
S. Varrette[*], V. Plugaru[*] and Pascal Bouvry[*]
[†]*Univerity of Luxembourg, MICS*
[*]*Computer Science and Communications (CSC) Research Unit*
2, avenue de l'Université, L-4365 Esch-sur-Alzette, Luxembourg
`Firstname.Name@uni.lu`

***Abstract*—This paper describes the development of an automated framework that allows to test the randomness of a Pseudo-Random Number Generator (treated as a black-box) using well-known statistical tests like the Dieharder tests and the NIST test suite. Another objective is to review the state-of-the-art in regards to the random number generators and their parallelization.**

1. Introduction

PRNG (Pseudo-Random Number Generators) play an important role in a variety of applications like computer simulations and modeling, industrial applications including statistical sampling and cryptography. PRNG is a deterministic algorithm that produces seemingly random numbers with certain statistical properties following some distribution (e.g. uniform).

Different tests are conducted to express the quality of any given PRNG. These tests can be divided into two distinct groups: empirical (statistical and application-based) tests and theoretical tests. Theoretical tests can be considered as priori tests in the sense that they require a knowledge of the used PRNG structure. In this paper, the main focus lies on the statistical tests.

We start in Chapter 2 with the motivations by pointing out real-world use cases for PRNG implementations. We then continue with Chapter 3 where we briefly describe the two main PRNG test suits that we are going to make use of. The methodology for our tests will be described at Chapter 4, following by its results in Chapter 5. Finally, we give some references to related works in Chapter 6 and we conclude this paper in Chapter 7.

2. Context & Motivations

Random numbers are used in many computational science fields (e.g. Monte Carlo simulations in physics, chemistry, bioinformatics and finance) and are essential to cryptography. Indeed, Monte Carlo simulation methods always require massive random numbers and the serial approach (consisting of dedicating a single process to the exclusive task of generating the random numbers) does not scale. Yet parallel PRNG requires some special attention, typically to ensure that the individual seeds have different sequence numbers in each processor. In general, one of the basic requirements for parallel random number streams is their mutual independence and lack of intercorrelation.

3. PRNG testing

Generating good random numbers is a difficult process, with the generators falling in two categories: Hardware-based Random Number Generators (HRNG) that are non-deterministic and Pseudo-Random Number Generators (PRNG) that are deterministic. The quality of the latter is generally tested by checking that: (1) the generated numbers are uniformly distributed in all dimensions, (2) there is no correlation between successive numbers, (3) the PRNG has a very long period for all seed states.

3.1. Dieharder Tests

Dieharder [2] [3] is a random number test suite. It is an improvement over the older Diehard implementation which was limited to only read random values from a previously generated binary file. Since PRNG weaknesses may only start to become apparent after having processed a substantial amount of random numbers, this approach leads to gigantic files and high I/O (input/output) loads and is therefore less than ideal. Dieharder, on the other hand, is able to take as input a continuous stream of random numbers, directly provided by the PRNG that we want to test.

It includes multiple test cases that each try to analyze how likely the observed random numbers may have originated from a random source. Since everything is based on probabilities, it is in fact possible that a 'good' PRNG occasionally fails one of the tests. A 'bad' PRNG can still be detected since it usually fails on many tests simultaneously.

Table 1 shows all available Dieharder tests, some of which are executed multiple times while running the tool.

Test Number	Test Name	Test Reliability
0	Diehard Birthdays	Good
1	Diehard OPERM5	Suspect
2	Diehard 32x32 Binary Rank	Good
3	Diehard 6x8 Binary Rank	Good
4	Diehard Bitstream	Good
5	Diehard OPSO	Good
6	Diehard OQSO	Good
7	Diehard DNA	Good
8	Diehard Count the 1s (stream)	Good
9	Diehard Count the 1s (byte)	Good
10	Diehard Parking Lot	Good
11	Diehard Minimum Distance (2d Circle)	Good
12	Diehard 3d Sphere (Minimum Distance)	Good
13	Diehard Squeeze	Good
14	Diehard Sums	Do Not Use
15	Diehard Runs	Good
16	Diehard Craps	Good
17	Marsaglia and Tsang GCD	Good
100	STS Monobit	Good
101	STS Runs	Good
102	STS Serial (Generalized)	Good
200	RGB Bit Distribution	Good
201	RGB Generalized Minimum Distance	Good
202	RGB Permutations	Good
203	RGB Lagged Sum	Good
204	RGB Kolmogorov-Smirnov	Good

TABLE 1. DIEHARDER TESTS [2]

3.2. NIST Test Suite

NIST [4] is yet another statistical test suite for random and pseudo-random number generators conceived for cryptographic applications. Contrarily to Dieharder, NIST is limited to only work on pre-generated binary files filled with random numbers.

4. Implementation and Experimental Setup

Our goal is to create a framework that, provided with a black-box PRNG, is able to produce and output random binary data, which in turns is automatically fed to a RNG test suite of choice. Finally, a report will be generated once all tests have been run. The results of our experiments will be depicted and described in Chapter 5.

We propose a wrapper implementation, written in C++ and compiled using CMake [13], that continuously generates new random values and outputs them to the standard output stream in form of binary data. The implementation is open-source and can be viewed on our Github[1] repository. As a proof-of-concept, the following well-known PRNGs have been chosen to be tested and analyzed in further detail:

- **g0: std::rand** [12]
  C++ Standard Library Random Number Generator. The exact properties are implementation dependent and thus may differ from one system to another.
  - Period: usually 2^{32}
  - Generated range: usually $[0; 2^{32}]$

- **g1: MT19937** [8]
  Mersenne Twister is a PRNG implementation named by the fact that its period length is a `Mersenne prime`.
  - Period: $2^{19937} - 1$
  - Generated range: $[0; 1]$
  - Good k-distribution property
  - 6024 32-bit parameters
  - Efficient use of memory
  - High speed (typically 4 times faster than `rand()` in C)

- **g2: MRG32k3a** [5]
  Multiple Recursive Generator, proposed by Pierre L'Ecuyer in 1998.
  - Period: 2^{191}
  - Generated range: $[-2^{53}; 2^{53}]$
  - Uniformly distributed

1. Github repository containing both the proposed proof-of-concept implementation and test results: https://github.com/IceYoshi/PRNG

Table 2 displays all available options when executing our wrapper. For instance, one may want to specify a upper limit for the number of random bytes to be generated. Leaving this at its default value of 0 results in it continuously generating new random numbers indefinitely. The type of generator can also be altered upon launch, with:

- `-g 0` (default) standing for std::rand,
- `-g 1` for MT19937, and
- `-g 2` for MRG32k3a.

Option	Description
-h [--help]	Display all available options
-n [--streamlength]	Number of bytes to be generated
-g [--generator]	Type of generator
-m [–mode]	Method for parallelization
-v [–verbose]	Human-readable stream output

TABLE 2. OPTIONS FOR PRNG WRAPPER IMPLEMENTATION

In order to accelerate the output of a given PRNG, one may try to distribute the execution of the generator over multiple processes in a parallelized environment. For this, we adapt the previous sequential wrapper implementation and propose the use of MPI [11] (short for Message Passing Interface) for any message exchanges between processes. The generated binary output of each process is funneled to a common output stream like illustrated in Figure 1.

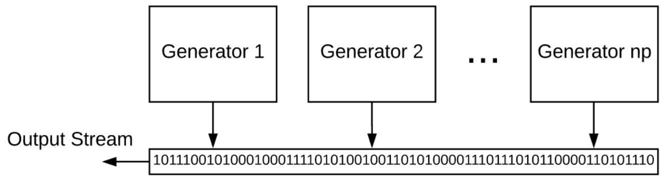

Figure 1. Multiple concurrent PRNG processes outputting random bits on a common stream

4.1. Challenges to PRNG Parallelization

There are some precautions that have to be dealt with when moving to to a scalable parallel execution scheme in order to avoid possible correlations between the produced outputs. For instance, one important aspect is the seed choice when initialization the individual PRNGs for each process. Picking the same seed value for every process will cause their output to be identical to each other. There are multiple approaches to solve this issue, some of which will be described below.

- **m0: Different parameter sets**
  PRNGs first have to be initialized using a `Seed`. Internally, this `Seed` is used to assign values to the PRNG's so-called parameter set. The generated output is produced by performing certain arithmetic operations on those parameters. This is also the reason why PRNGs are referred to as only being pseudo-random, since every step in this process is completely deterministic and only depends on the current parameter set state. Although PRNGs do not have any direct access to true randomness, they still manage to generate numbers that follow a desired distribution and that come with certain statistical properties that make them 'look' random.

  Since we are dealing with exactly the same implementation of a PRNG over many simultaneously executed processes, one may want to assign a distinct `Seed` to every individual PRNG in order to avoid any correlation between the generated outputs. For this to work in a reliable way, the provided PRNG has to come with a sufficiently long period such that the probability for any overlap between different `Seeds` can be neglected. Moreover, `Seeds` may not have the same effect over the output as hashing algorithms do. In other words, depending on the used PRNG implementation, a small difference in the `Seed` may always result in a small difference on the output. Even though the generated values usually diverge rather quickly over time, this is still an unwanted effect that should be avoided at any cost.
  Here are some suggestions to assign seeds to individual generators:
  - Choose random seed. For this to work, there has to be another generator that generates the seed. This is therefore only moving the problem to another place instead of solving it.
  - Start with the value of the current Unix time and increment it by the id of the given process. We assume that the process id is unique within the parallelized system. However, this approach results in seeds that are close by, which may cause similar random number outputs across the generators at first. This can be avoided by using a hash function on top of the given value.
  - Start once again with the Unix time and multiply it by (ID + 1). We want to avoid assigning the value 0 to a seed, and since IDs usually tend to start with 0, we increment it by one in order to avoid a zero multiplication.

- **m1: Block-splitting**
  In case every PRNG needs to be initialized with the same `Seed`, one may want to try the block-splitting (or skipping-ahead) approach which consists in splitting the original sequence of the to be generated numbers into non-overlapping blocks that can be simultaneously executed independently from each other on different processes (see Figure 2). The concatenation of all output streams should be identical, except for the ordering, to the one that would be produced if there was only one process running. However, this method is only applicable if the total number of random numbers to be generated is known at the beginning of the execution since the block sizes are static and cannot be changed after the fact. Block-splitting is therefore not appropriate if the goal is to create a open-ended continuous random number stream.

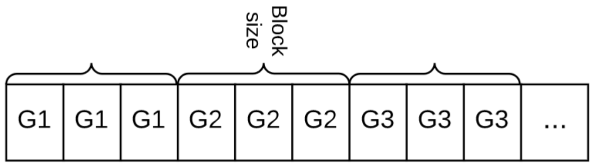

Figure 2. Block-splitting illustration with block size of 3

- **m2: Leap-frogging**
  Leap-frogging is a variation of the block-splitting. Let np be the number of processes running in parallel. Instead of defining sequential block boundaries for every process, each PRNG starts on a different head offset and jumps np positions after every generated value. The offsets are chosen in a way such that there won't be any overlap between two distinct processes and that the total space of the PRNG is covered (see Figure 3). With this method, it is possible to continue the number generation indefinitely, thus eliminating the previous drawback of block-splitting. However, this still does not support adding more processes on the fly, since the value np would have to be adapted accordingly on every process in a synchronized manner.

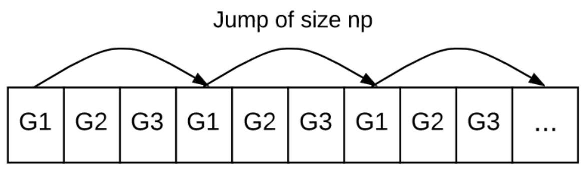

Figure 3. Leap-frogging with 3 concurrent process generators

In both block-splitting and leap-frogging, we are talking about advancing the current PRNG head to a given position.

There may, however, not be any more efficient way of doing so than to generate all the intermediate numbers. The explanation lies underneath the understanding of how a PRNG works internally. Initially, the parameter set of a PRNG is assigned using the provided `Seed`. During the computation of the next number, those parameters change. This explains why the subsequent generated numbers are able to differ from the previous ones. Since the parameters depends on the previous parameter set state, we do not always have a general formula that allows us to directly compute their values for a given position.

For instance, there exists a more efficient method to skip a big block section when using block-splitting with the MT19937 and MRG32k3a generators, but there is none for leap-frogging [10]. One could also imagine a hybrid implementation that makes both use of block-splitting and leap-frogging. The main idea consists into dynamically assigning smaller blocks to each individual process and make them jump to the starting position of the next block once they reach the end of the previous one, thus drastically reducing the number of required micro-jumps present when using the leap-frogging technique only, while avoiding the previously stated drawback, present in block-splitting, where the user must specify in advance the upper limit of to be generated numbers.

4.2. GPU Accelerated PRNGs

GPUs are ideal when it comes to highly parallelized work loads due to their sheer number of cores. For this section we have used as main reference the website Massively Parallel Random Number Generators [14].

We executed the GPU raw performance and Dieharder tests on the following PRNGs: `md5`, `tea`, `drand48gpu`, `park_miller`, `tt800`, `lfsr113`, `combinedLCGTaus`, `kiss07`, `mtgp`, `ranecu` and `precompute`. For Dieharder, a single stream test was also performed. The code was executed locally on Linux/CUDA 9 suite, using the following commands:

- **make** - help overview of the tests, RNGs, and reports
- **make bench_raw_performance-md5** - raw performance test on the `md5` PRNG
- **make bench_raw_performance-all** - raw performance test on all PRNGs
- **make report-raw_performance.data** - create data for raw performance test
- **make report-raw_performance.ps** - create graph from the raw performance test results
- **make report-dieharder_single_stream.html** - generate html report from the Dieharder test results
- **make dieharder_single_stream-ranecu** - execute Dieharder single stream test on the `ranecu` PRNG
- **make -B report-dieharder.html** - generate html report for all Dieharder tests

4.3. PRNG Validation - Dieharder Tests

In order to test the quality of PRNG implementations using Dieharder, we can execute the following commands:

```
$ ./rng_seq -g 0 | dieharder -g 200 -a
                    > dieharder_seq_g0.txt
$ srun -n $SLURM_NTASKS ./rng_mpi
        -g 0 -m 0 | dieharder -g 200 -a
                > dieharder_mpi_g0_m0.txt
```

Listing 1. Sample commands for running Dieharder tests using both sequential and MPI versions of the PRNG wrapper

Where `-g 200` refers to the universal `stdin_input_raw` interface (expects continuous stream of presumably random bits). `-a` runs all available tests.

Some useful Dieharder options (`x` stands for any natural number):

- Replace `-a` with `-d x` in order to only execute a particular test. `-d -1` in order to list all available tests.
- Specify `-m x` in order to test with a bigger pool of random numbers. Note that the test will also take `x`-times longer to run.

4.4. PRNG Validation - NIST Test Suite

Since the NIST test suite does not support an input stream coming directly from a random source, we can prepare a binary file in advance, e.g. by executing the following commands:

```
$ ./rng_seq -g 0 -n 1000000000
                        > data/seq_g0.dat
$ srun -n $SLURM_NTASKS ./rng_mpi -g 1
    -m 0 -n 1000000000 > data/mpi_m0.dat
```

Listing 2. Sample commands for generating random binary files using both sequential and MPI versions of the PRNG wrapper for later use within the NIST test suite.

Now one can start the experiment with `./assess <bitStreamLength>` and following the instructions. We propose the following options:

Input file(0) → data/seq_g0.dat → All(1) → Default(0) → Bitstreams(10) → Binary(1)

After successful completion, the report can be found under *experiments/AlgorithmTesting/finalAnalysisReport.txt*

5. Validation and Experimental Results

The data shown in this section only represent the summary of all the performed tests. The reader is advised to take a look at the raw values that have been submitted together with this paper.

5.1. Speed Benchmark Results

We start by comparing the number generation speed of the different PRNGs using the three previously mentioned architectures: (1) sequential, (2) parallel using OpenMPI, and (3) GPU accelerated ones. For that, we let the generators output 10^{11} random 32-bit numbers and measure the real-world execution time. For the parallel part, the method `m0` (different parameter sets) has been chosen since it introduces the least amount of overhead compared to `m1` (block-splitting) and `m2` (leap-frogging).

In order to be statistically significant, we repeated every run 10 times and computed the average value. While both the sequential and OpenMPI tests have been done on the HPC Platform [1] using the Iris cluster, the tests involving GPU have been done locally due to time restraints. SLURM launchers have been written in order to facilitate and automate the job batch submissions. Listing 3 and 4 represent some small command snippets taken from said launchers. Since there's no need for the actual output of the PRNGs during this test, it is being redirected to `/dev/null`. Finally, some buffering in the code level of the wrapper has been applied to the output in order to reduce its performance impact. In fact, flushing the output stream to the console takes many orders of magnitude longer than the generation of numbers themselves.

Table 3 shows the results of our conducted test. In turns of speed, we observe that the `MT19937` generator performs about twice as fast as `std::rand`. `MRG32k3a` falls in last place, being the slowest PRNG of the three.

Comparing the architectures, we see a clear boost in speed when going from sequential to parallel processing and finally to GPU accelerated computing, with observed difference that occasionally exceed an order of magnitude.

Architecture	**Generator**	**Time** $[s]$	**Speed** $[s^{-1}]$
Sequential	g0: std::rand	1647.502	$6.07 \cdot 10^7$
	g1: MT19937	491.299	$2.035 \cdot 10^8$
	g2: MRG32k3a	2420.335	$4.132 \cdot 10^7$
OpenMPI $n = 56$	g0: std::rand	35.4	$2.825 \cdot 10^9$
	g1: MT19937	20.696	$4.832 \cdot 10^9$
	g2: MRG32k3a	52.125	$1.918 \cdot 10^9$
GPU GeForce GTX 950M	md5	49.596	$2.016 \cdot 10^9$
	kiss007	2.82	$3.546 \cdot 10^{10}$
	drand48gpu	2.264	$4.416 \cdot 10^{10}$

TABLE 3. EXECUTION TIME IN ORDER TO GENERATE 10^{11} RANDOM NUMBERS

```
$ TIMEFORMAT=%R
$ time ./rng_seq -g 0 -n 100000000000
                    &> /dev/null

1647.502
```

Listing 3. Command for sequential speed benchmark

```
$ TIMEFORMAT=%R
$ time srun -n $SLURM_NTASKS ./rng_mpi
  -g 0 -m 0 -n 100000000000 &> /dev/null

35.4
```

Listing 4. Command for MPI speed benchmark on 56 cores

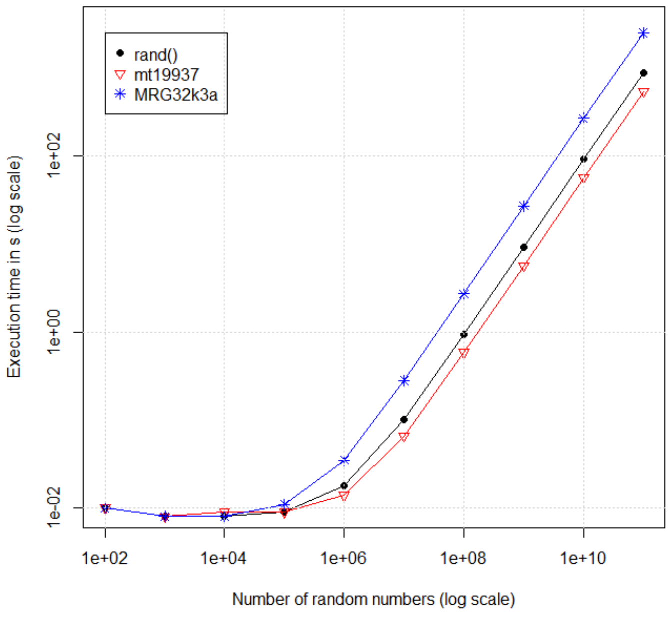

Figure 4. Execution speed between different sequential PRNGs

The generation rate of PRNGs stays more or less constant throughout their execution (shown in Figure 4 by the linear lines), with the exception of the beginning due to the initialization phase taking some time.

Figure 5 shows the execution speed of the MPI code depending on the number of cores it is ran on. The Iris cluster, where this test has been conducted, consists of many nodes, each containing 28 cores. Therefore, the graph has been split into 2 parts:

- 1 to 28 cores: Within the same node ($N = 1$)
- 28 to 56 cores: Split on two separate nodes ($N = 2$)

We once again are able to visually see the difference in speed between the three PRNGs. We observe a sudden change in performance when transitioning from one to two nodes. This is to be expected, since there is some communication overhead introduced between nodes, even if the connections themselves are purposefully build to guarantee high throughput and low latency using InfiniBand. Overall, we can deduce that the generation speed is scaling quite well with the number of cores. There is some anomaly observed with the `MRG32k3a` generator not performing as expected between 15 and 28 cores.

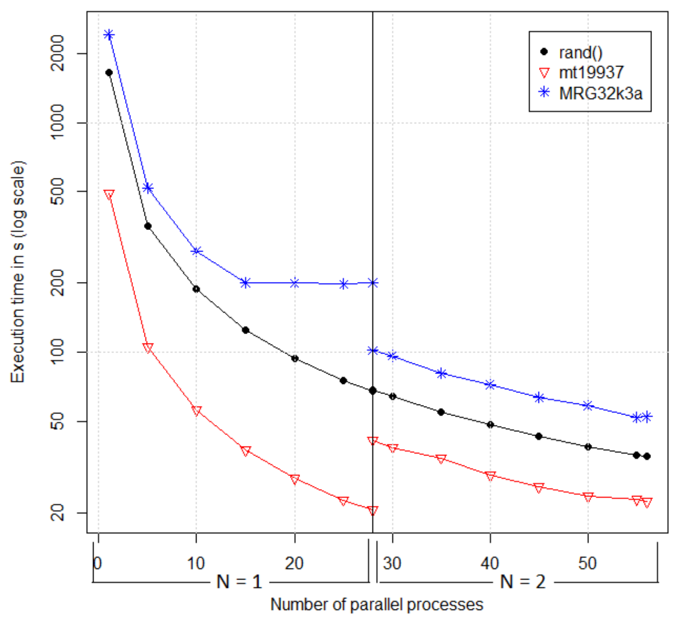

Figure 5. Execution speed comparison between different PRNGs with constant $m = 0$ and 10^{11} random numbers (OpenMPI)

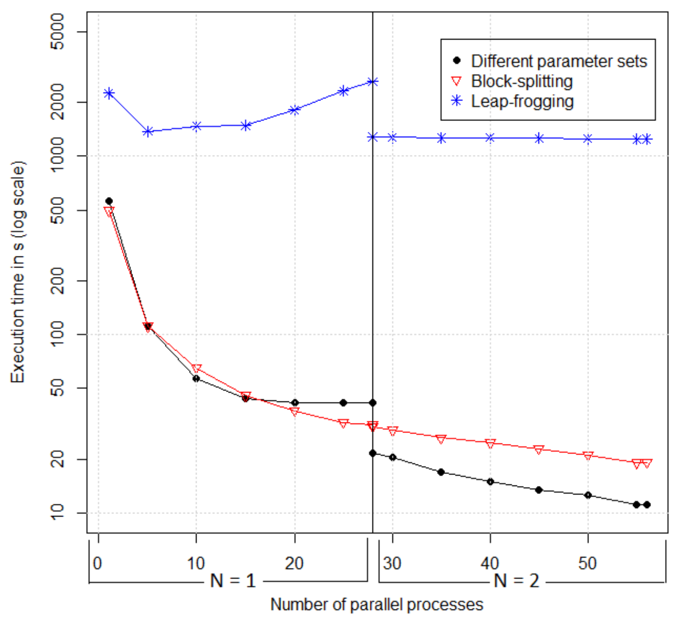

Figure 6. Execution speed comparison between different parallelism methods for MT19937 generator and 10^{11} random numbers (Without output stream overhead)

Finally, Figure 6 shows the performance comparison between the three previously presented methods of parallelization on the `MT19937` generator. Once again, we observe some performance jumps when transitioning from one to two nodes. The approach of choosing different parameter sets for each individual PRNG introduces a initialization overhead of complexity $O(1)$ and thus does not negatively affect the scaling performance curve when adding multiple processes. On the other hand, we are looking at a linear complexity for the PRNG initialization when making use of block-splitting. After the longer initialization part, however, the generation rate matches exactly the one observed from the previous method. With the proposed implementation of leap-frogging, there is no clear advantage in going from a sequential to a parallelized environment, since every PRNG process needs to generate the exact same sequence of random numbers.

Keep in mind that the overhead introduced by outputting the generated bits into a common stream has been neglected during this test. If this overhead is significantly bigger than the time needed to generate random numbers, then leap-frogging suddenly wields some better results. This is illustrated in Figure 7.

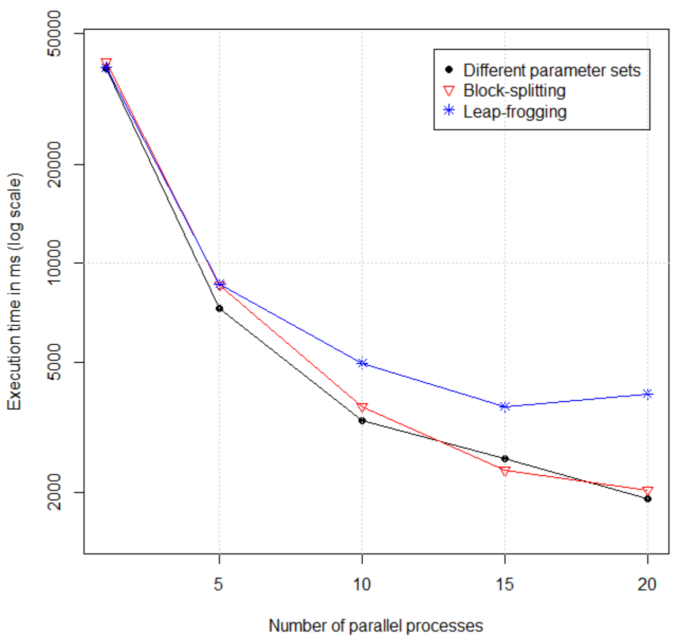

Figure 7. Execution speed comparison with between different parallelism methods for MT19937 generator and 10^8 random numbers (With output stream overhead)

The characteristics of the used GPU (GeForce GTX 950M):

- multiprocessor count: 5
- stream processor count: 128 (total 640)
- warp size: 32
- max threads per block: 1024
- max block dimensions: 1024 x 1024 x 64
- max grid dimensions: 2147483647 x 65535 x 65535

The Dieharder tests were run using the following flags:

```
DIE_HARDER_FLAGS = -g 200 -a -c ','
-D 33272
```

To obtain the 335544320 random numbers used in the raw performance test presented in Figure 10, we used 10 program calls, 10 kernel calls, 16384 threads and 2048 random numbers calculated in each thread in each kernel call. The resulting amount of random numbers per program launch is thus the multiplication of:

kernel_calls · num_threads · num_randoms_per_thread

In Figure 8, we have plotted the evolution of the execution time versus the quantity of random numbers generated, for all studied PRNG.

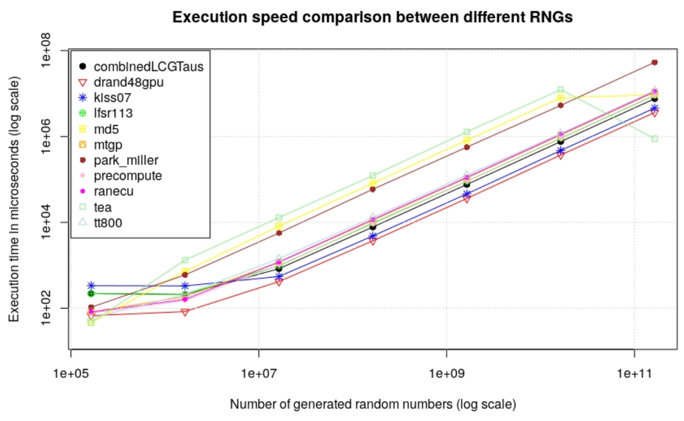

Figure 8. Execution speed comparison between GPU accelerated PRNGs

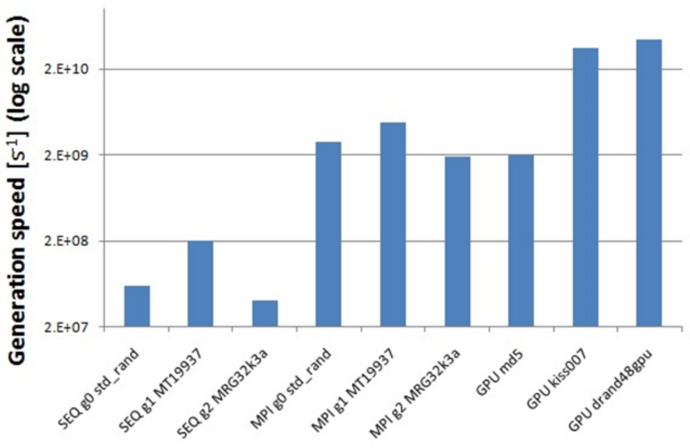

Figure 9. Execution speed comparison between different PRNGs

5.2. PRNG Validation Results

Conducted experiments:

- **e1**: Sequential `std::rand` generator
- **e2**: Sequential `MT19937` generator
- **e3**: Sequential `MRG32k3a` generator
- **e4.1**: Different parameter sets, random seeds, using MPI `MRG32k3a` generator
- **e4.2**: Different parameter sets, fixed seeds, using MPI `MRG32k3a` generator
- **e5**: Block-splitting, using MPI `MRG32k3a` generator
- **e6**: Leap-frogging, using MPI `MRG32k3a` generator

The results can be seen in Table 4 and 5.

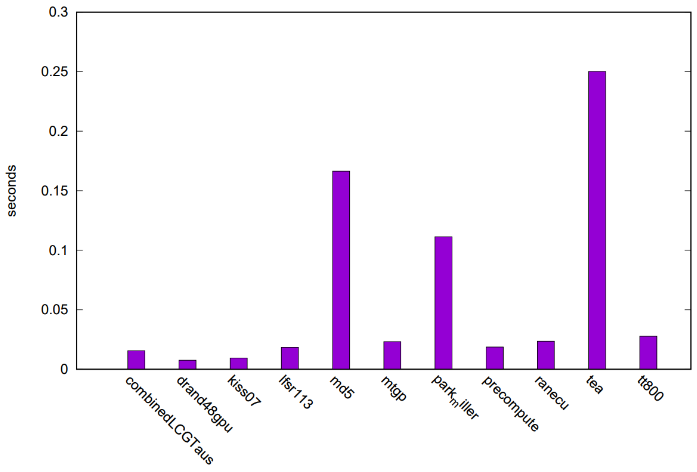

Figure 10. Raw Performance, GeForce GTX 950M, 335544320 Random Numbers

	Test results [passed/total]	
Experiment	**Dieharder**	**NIST**
e1	68/114	80/188
e2	114/114	188/188
e3	114/114	188/188
e4.1	114/114	188/188
e4.2	69/114	35/188
e5	114/114	188/188
e6	114/114	188/188

TABLE 4. PRNG TEST RESULTS

Table 5 combines the results of the Dieharder and raw performance tests. We notice that the slowest generators are not necessarily also the ones with the best quality and vice-versa.

On the GPU side, the following generators performed the best: `lfsr113`, `mtgp` and `kiss07`, with `lfsr113` passing all tests, and `kiss07` being the second fastest algorithm tested. Finally, `Ranecu`, `drand48gpu`, `park_miller` and `precompute` should be avoided as they systematically failed the statistical tests.

6. Related Work

The Middle Square Method, described by John Von Neumann in the early 1950s, was the first attempt to implement a PRNG, which was proven to be a poor source of random numbers with short periodicity [15] by Donald Knuth in 1969. Later in the second edition, the author proposed several empirical tests (called Knuth's tests) and used them to test the quality of PRNGs.

Metropolis 38-bits has been used as a common test for a long time, for which exact results are known [16]. Later on, other cluster algorithms such as Swendsen-Wang and Wolff were conceived which gave satisfactory results. [17] [18] Another paper [19] extends the above mentioned algorithms to test PRNG in the context of parallel execution.

The problem of testing parallel PRNGs was described in yet another paper [20]. The authors describe the imple-

	Test results			
RNG	**Passed**	**Failed**	**Weak**	**Speed**
lfsr113	112	0	0	reference
mtgp	111	0	1	reference
kiss07	111	0	1	faster
md5	110	0	2	slower
combinedLCGTaus	110	0	2	faster
tea	109	0	3	slower
tt800	108	0	4	reference
ranecu	56	**50**	6	reference
drand48gpu	54	**49**	9	fastest
park_miller	26	**85**	1	slower
precompute	0	**112**	0	reference

TABLE 5. GPU DIEHARDER PRNG TEST RESULTS

mentation of several tests in the SPRNG (Scalable Parallel Random Number Generators) test suite. [21]

The DIEHARD [22] suite of statistical tests developed by George Marsaglia consists of fifteen statistical tests. Later this test suite was extended to test suite named Dieharder [2] and includes 26 fully implemented empirical tests.

The NIST Statistical Test Suite [4] was published in 2001 and its last edition was in 2014.

The paper [23] proposed a new test suite named TestU01, implemented in ANSI C. It offers several packages of tests including ”Small Crush” (consisting of 10 tests), ”Crush” (96 tests), and ”Big Crush” (160 tests).

Finally, a paper [24] provides a review of existing CUDA variants of RNG, presents the CUDA implementation of a new high-quality PRNG and describe used test suite.

7. Conclusion

In regards to CPU performance, all three used methods yielded good results with leap-frogging performing slightly worse in terms of speed and scalability. As expected, the GPU execution was significantly faster than parallel CPU, which was itself faster than a sequential CPU execution. Overall, `MT19937` performed the fastest of the three analyzed generators. Using the same seeds in order to initialize the generators across all processes results in them outputting the same random numbers, thus the need to additionally use methods like block-splitting or leap-frogging. The `std::rand` generator should be avoided for any critical applications like cryptography.

On the GPU side, the following generators performed the best: `lfsr113`, `mtgp` and `kiss07`, with `lfsr113` passing all tests, and `kiss07` being the second fastest algorithm tested. Finally, `Ranecu`, `drand48gpu`, `park_miller` and `precompute` should be avoided as they systematically failed the statistical tests.

Acknowledgments

The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg [1] – see http://hpc.uni.lu.

References

[1] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, “Management of an Academic HPC Cluster: The UL Experience,” in Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014). Bologna, Italy: IEEE, July 2014, pp. 959–967.

[2] Robert G. Brown. A random number test suite. http://www.phy.duke.edu/~rgb/General/dieharder.php. Last accessed: 2018.

[3] Jochen V. A modified version of Robert G. Brown’s ”dieharder” tests for random number generators. https://github.com/seehuhn/dieharder. Last accessed: 2018.

[4] Lawrence E. Bassham, III, Andrew L. Rukhin, Juan Soto, James R. Nechvatal, Miles E. Smid, Elaine B. Barker, Stefan D. Leigh, Mark Levenson, Mark Vangel, David L. Banks, Nathanael Alan Heckert, James F. Dray, and San Vo. Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, Gaithersburg, MD, United States, 2010.

[5] Pierre L’Ecuyer. Combined multiple recursive random number generators. Operations Research, 44(5):816–822, 1996.

[6] Michael Mascagni and Ashok Srinivasan. Algorithm 806: Sprng: A scalable library for pseudorandom number generation. ACM Trans. Math. Softw., 26(3):436–461, September 2000.

[7] Makoto Matsumoto and Takuji Nishimura. Dynamic creation of pseudorandom number generators. Monte Carlo and Quasi-Monte Carlo Methods, 2000:56–69, 1998.

[8] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul., 8(1):3–30, January 1998.

[9] S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. Commun. ACM, 31(10):1192–1201, October 1988.

[10] Intel. Independent Streams. Leapfrogging and Block-Splitting. https://software.intel.com/en-us/node/590372. Last accessed: 2018.

[11] Open MPI. A High Performance Message Passing Library. https://www.open-mpi.org/. Last accessed: 2018.

[12] std::rand. http://www.cplusplus.com/reference/cstdlib/rand/. Last accessed: 2018.

[13] CMake. Build, Test and Package Your Software With CMake. https://cmake.org/. Last accessed: 2018.

[14] Holger Dammertz, Christoph Schied and Hendrik Lensch. Ulm University. Massively Parallel Random Number Generators. http://mprng.sourceforge.net/. Last accessed: 2018.

[15] D. E. Knuth. The Art of Computer Programming, volume 2: Seminumerical Algorithms 2nd ed. Addison Wesley, Reading, Mass., 1981.

[16] H. Gould and J. Tobochnik. An Introduction to Computer Simulation Methods, Vol. 2, (Addison-Wesley, Reading, Mass., 1988).

[17] Ferrenberg AM, Landau DP, Wong YJ. Monte Carlo simulations: Hidden errors from ”good” random number generators. Phys Rev Lett. 1992 Dec 7;69(23):3382-3384.

[18] I. Vattulainen, T. Ala-Nissila, and K. Kankaala. Physical Tests for Random Numbers in Simulations. Phys. Rev. Lett. 73, 2513 (1994)

[19] P. D. Coddington. Tests of random number generators using Ising model simulations. International Journal of Modern Physics C 7(03). 1996

[20] A Srinivasan, M Mascagni, D Ceperley. Testing parallel random number generators. Parallel Computing 29 (1), 69-94, 2003.

[21] The Scalable Parallel Random Number Generators Library (SPRNG) http://www.sprng.org. Last accessed: 2018.

[22] ”The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness”. Florida State University. 1995.

[23] P. L’Ecuyer, R. Simard. TestU01: A C Library for Empirical Testing of Random Number Generators, ACM Trans. Math. Softw. 33 (2007) 22:1–22:40.

[24] Manssen, M., Weigel, M., Hartmann, A.K. Random number generators for massively parallel simulations on GPU. Eur. Phys. J. Spec. Top. (2012) 210: 53.https://doi.org/10.1140/epjst/e2012-01637-8