TinyMT Pseudo Random Number Generator for Erlang

Kenji Rikitake

Institute for Information Management and Communication (IIMC), Kyoto University

kenji.rikitake@acm.org

Abstract

This paper is a case study of implementing Tiny Mersenne Twister (TinyMT) pseudo random number generator (PRNG) for Erlang. TinyMT has a longer generation period ($2^{127} - 1$) than the stock implementation of Erlang/OTP *random* module. TinyMT can generate multiple independent number streams by choosing different generation parameters, which is suitable for parallel generation.

Our test results of the pure Erlang implementation show the execution time of RNG generating integers with TinyMT is approximately two to six times slower of that with the stock *random* module. Additional implementation with Native Interface Functions (NIFs) improved the execution speed to approximately three times as faster than that of the *random* module. The results suggest TinyMT will be a good candidate as an alternative PRNG for Erlang, regarding the increased period of the RNG and the benefit of generating parallel independent random number streams.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; D.3.2 [*Programming Languages*]: Language Classifications—Erlang; G.3 [*Probability and Statistics*]: Random Number Generation

General Terms Algorithms, Performance

Keywords Erlang, Pseudo Random Number Generator, Mersenne Twister, TinyMT

1. Introduction

Random number generators (RNGs) is an essential component of modern programming languages, providing the necessary randomness required for generating unpredictable results. In this paper, we focus on Pseudo RNGs (PRNGs), which are generated through computational algorithms. We will exclude discussing cryptographic safety of each PRNG in this paper.

PRNG algorithms for massively parallel execution environments are of active research, for both the quality and performance aspects, with the details as follows:

Speed Large-scale simulation programs demand fast random number generation, as the execution speed increases.

Length of period Period of each PRNG should be long enough to guarantee the randomness. The longer one is always better than the shorter one, if the other characteristics are not largely affected.

Size of internal state High-speed memory is expensive, especially in cache-dependent architectures which is common among modern computers. Reducing the overhead time of copying the state is also essential for a fast RNG.

Concurrent/parallel generation Capability of generating non-predictable and independent multiple sequences for concurrent or parallel execution from the same algorithm is essential. Making an RNG as a shared state between multiple processes or threads should be avoided as possible since the execution speed of RNG will restrict the speed of all the related processes or threads.

The stock *random* module of Erlang/OTP uses an algorithm called AS183 [1], which has a short period length of $\sim 2^{43}$ [2]. While the internal state size is small (three 15-bit integers), it does not have the capability of generating multiple independent multiple sequences.

This paper is a case study of addressing implementation of PRNGs for Erlang which can maximize the efficiency at a massive parallel execution environment. In this paper, we present an Erlang implementation of algorithm called Tiny Mersenne Twister (TinyMT) [17], called *tinymt-erlang*, as an alternative to our previous SIMD-oriented Fast Mersenne Twister (SFMT) [15] implementation for Erlang.

An outline of this paper's topics is as follows:

- implementation issues of SFMT for Erlang (Section 2);
- the goals and implementation details of *tinymt-erlang* (Section 3);
- performance test results and evaluation of *tinymt-erlang* and comparison with the *random* module (Section 4);
- related work (Section 5); and
- concluding remarks (Section 6).

2. Implementation issues of sfmt-erlang

We present an Erlang PRNG implementation of SFMT called *sfmt-erlang* [13]. The goals of the implementation are to provide a modern and well-known PRNG for Erlang which is suitable for large-scale simulation, while retaining the speed fast enough so that the users would see the incentives to choose it. MT has been chosen among open source language systems such as Python [11] and R [12], not only because of the RNG characteristics, but also due to the choice of the BSD license [19], which allows the proprietary and commercial use of the by-products. One of the real-world usage examples of *sfmt-erlang* is for an Erlang testing tool PropEr [10]; users can choose which PRNG to use between the *random* module and *sfmt-erlang* [21] at the tool building.

SFMT, however, has its own issues which should be solved as an RNG for concurrent/parallel execution environment. We have found that SFMT requires a batch generation of random numbers

at once which fills in the internal state table, and that the execution time of the batch generation is proportional to the internal state table length. Using a NIF means blocking a CPU core of the Erlang virtual machine (BEAM) while the execution, so the execution time of the batch generation should be considered to avoid degrading the performance of BEAM by causing jitter on the scheduling activity. During the development of *sfmt-erlang*, we conclude that the period length of $2^{19937}-1$ gives the best trade-off between the period length and the performance [13], which keeps the execution time of each batch generation $\leq 50\mu$s.

Our implementation also has the following new issues which shows that *sfmt-erlang* is not suitable as a direct replacement of the *random* module:

- The internal state data are large (2496 bytes for the period of $2^{19937}-1$) and have to be placed into the shared heap of BEAM.
- Only single sequence can be generated from the same piece of code. The users can choose multiple generation parameters and theoretically dynamic creation of the parameters is possible [8], but practically it is too slow for a long period due to the observation that average CPU time to find each parameter set increases exponentially to the exponent p where the generation period is $2^p - 1$ [8, Table 2].
- The algorithm is much slower without NIFs, especially when generating a set of random numbers at once. 624 random numbers have to be generated sequentially at once for the period of $2^{19937} - 1$.
- The usage of NIFs should not be mandated if the code is provided as a general library for multiple operating systems, since maintaining the NIFs for the Windows environment is far more difficult due to the non-availability of the C compilers. One of the users of *sfmt-erlang* requested us to release and support the pure Erlang implementation of SFMT since it would not crash BEAM [4]. We accepted the request and now support the pure Erlang code which is functionally equivalent to the NIF code.

3. TinyMT implementation

3.1 TinyMT algorithm

TinyMT is a variant of Mersenne Twister (MT) [9] proposed by Saito and Matsumoto [16], specifically designed for a small memory footprint. On TinyMT, the users can generate multiple independent sequences when choosing different sequence parameter sets with the Dynamic Creator (DC) [8]. The seed jumping function, which calculates the internal state of TinyMT after an arbitrary steps of the recursive state transitions, is also provided to make multiple non-overlapping sequences from the same sequence parameter sets. TinyMT is licensed under the BSD License as well as the other MT variants.

TinyMT is a combination of two different functions: the state transition function and the output function. Two different output functions with 32-bit and 64-bit tempering parameters are proposed. In this paper, we focus on the algorithm with the 32-bit tempering parameter.

The size of TinyMT internal state with generation parameters for the 32-bit tempering parameter is 28 bytes including the 127-bit internal state and three 32-bit generation parameters. The period of each generated number sequences is $2^{127} - 1$. Saito and Matsumoto [16] showed that TinyMT passed the BigCrush tests of TestU01 [6], and estimated that the total number of generation parameter sets which could be generated by the TinyMTDC, a variation of MT DC algorithm for TinyMT, is $\sim 2^{58}$ regarding the number of computed irreducible polynomials. TinyMTDC is written in C++ and depends on Number Theory Library (NTL) [18].

```
%% Internal state definition
-type uint32() :: 0..16#ffffffff.
-record(intstate32,
        {status0 :: uint32(), status1 :: uint32(),
         status2 :: uint32(), status3 :: uint32(),
         mat1 :: uint32(), mat2 :: uint32(),
         tmat :: uint32()}).
-define(TINYMT32_SH0, 1).
-define(TINYMT32_SH1, 10).
-define(TINYMT32_SH8, 8).
-define(TINYMT32_MASK, 16#7fffffff).
-define(TINYMT32_UINT32, 16#ffffffff).

%% Initial internal state and tempering parameters
-spec seed0() -> #intstate32{}.
seed0() ->
    #intstate32{
      status0 = 297425621, status1 = 2108342699,
      status2 = 4290625991, status3 = 2232209075,
      mat1 = 2406486510, mat2 = 4235788063,
      tmat = 932445695}.

%% Computing internal state
-spec next_state(#intstate32{}) -> #intstate32{}.
next_state(R) ->
    Y0 = R#intstate32.status3,
    X0 = (R#intstate32.status0 band ?TINYMT32_MASK)
          bxor R#intstate32.status1
          bxor R#intstate32.status2,
    X1 = (X0 bxor (X0 bsl ?TINYMT32_SH0))
          band ?TINYMT32_UINT32,
    Y1 = Y0 bxor (Y0 bsr ?TINYMT32_SH0) bxor X1,
    S0 = R#intstate32.status1,
    S10 = R#intstate32.status2,
    S20 = (X1 bxor (Y1 bsl ?TINYMT32_SH1))
           band ?TINYMT32_UINT32,
    S3 = Y1,
    Y1M = (-(Y1 band 1)) band ?TINYMT32_UINT32,
    S1 = S10 bxor (R#intstate32.mat1 band Y1M),
    S2 = S20 bxor (R#intstate32.mat2 band Y1M),
    R#intstate32{status0 = S0, status1 = S1,
                 status2 = S2, status3 = S3}.

%% Outputting a 32-bit integer from the internal state
-spec temper(#intstate32{}) -> uint32().
temper(R) ->
    T0 = R#intstate32.status3,
    T1 = (R#intstate32.status0 +
          (R#intstate32.status2 bsr ?TINYMT32_SH8))
          band ?TINYMT32_UINT32,
    T2 = T0 bxor T1,
    T1M = (-(T1 band 1)) band ?TINYMT32_UINT32,
    T2 bxor (R#intstate32.tmat band T1M).
```

Figure 1. TinyMT main function definition in Erlang.

We assume that the characteristics of TinyMT suggest:

- the internal state size is smaller than SFMT, so the internal state is faster to be moved around between Erlang functions;
- multiple sequences can be easily generated from the same piece of code by modifying the generation parameters; and
- the algorithm is inherently faster than SFMT and will be suitable for the implementation with pure Erlang code.

3.2 TinyMT implementation in pure Erlang

The design goals of our TinyMT implementation on Erlang are as follows:

- running fast enough to run as a pure Erlang piece of code;

Function name	Description
`next_state/1`	Performs a state transition from a given internal state with a given set of generation parameters
`temper/1`	Outputs a 32-bit integer random number of $[0, 2^{32} - 1]$ range from a given internal state
`temper_float/1`	Outputs a float random number of $[0, 1)$ range from a given internal state
`uniform/{0,1}`	Generates an integer random number of $[1, N]$ range for a given positive integer N (compatible with the *random* module)
`uniform_s/{1,2}`	Generate a float random number (compatible with the *random* module)

(All above functions are provided in both pure Erlang and as NIFs)

Table 1. List of *tinymt-erlang* major exported functions.

- retaining readability while optimized for the speed; and
- providing full compatibility functions to the *random* module, so that the implementation can be used as a drop-in replacement.

Table 1 shows a list of *tinymt-erlang* major exported functions referred in this paper. Figure 1 shows the Erlang code of the 32-bit version of the TinyMT algorithm. The following is a list of how this example code is designed:

- This example code does not use any floating point calculation or branching with Erlang `case` expression for higher performance.
- The state and generation parameters are all inside the record `#intstate32{}`.
- Function `seed0/0` represents an example set of the initial state and generation parameters.
- Function `next_state/1` performs the recursive state transition of the internal state.
- Function `temper/1` outputs a 32-bit random number from the internal state with the tempering matrix. We also provide the function `temper_float/1` to output a float random number.
- We should note that the state transition and the output functions are independent with each other; this means the functions `next_state/1` and `temper/1` should be executed one after the other to generate an output *and* to forward the internal state.

We also provide the seeding functions as specified in the TinyMT reference C source code [17], which employs similar internal state initialization function to the code of *sfmt-erlang*.

We added the following compatibility functions as specified in the *random* module of Erlang/OTP:

- generating float random numbers of $[0, 1)$ (`uniform/0` and `uniform_s/1`);
- generating integer random numbers within the range of $[1, N]$ for a given positive integer N (`uniform/1` and `uniform_s/2`);
- saving the internal state and generation parameters in the process dictionary without explicitly holding the internal state as a variable (`uniform/0` and `uniform/1`); and
- seeding functions using the process dictionary, including the one which initializes the seed with three integer arguments and with a tuple of three integer elements (`seed0/0`, `seed/{0,1,3}`).

Total execution time of 10^6 executions [ms]			
function name	**laurel**	**minimax**	**reseaux**
Without HiPE			
`random:uniform_s/1`	504	367	2499
`random:uniform_s/2`	506	366	2494
`tinymt32:uniform_s/1`	418	320	7273
`tinymt32:uniform_s/2`	542	349	8156
With HiPE `o3` option			
`random:uniform_s/1`	458	390	2414
`random:uniform_s/2`	456	392	2406
`tinymt32:uniform_s/1`	132	125	5806
`tinymt32:uniform_s/2`	150	147	6550

(measured by the difference of `statistics(runtime)`, with pure Erlang functions without NIFs.)

Table 2. Time comparison of total execution time by wall clocks for the random number generation of 10^6 integers (`uniform_s/1`) and floats (`uniform_s/2`).

For the output functions of ranged integers of $[1, N]$, we guarantee all numbers between the range will appear in the same probability, by applying the following algorithm:

a) Let R be a 32-bit random integer in $[0, 2^{32} - 1]$, obtained as a return value of function `temper/1`;

b) Compute Q where $0 = Q \bmod N$, and $0 \le (2^{32} - Q) \le (N - 1)$ (i.e., Q is the closest multiple of N to 2^{32});

c) If $R > Q$ then try **a)** again; else compute the result $O \in [1, N]$ where $O = (R \bmod N) + 1$.

4. Test results and evaluation

4.1 Performance test results

We conducted a performance comparison test between *random* and *tinymt-erlang* modules with the following three execution environments, all running Erlang/OTP R15B01 [1] and the stock GNU C compilers [2].

laurel Intel Xeon E5-2670 dual-core CPU ×8 (16 CPU cores), 2.6GHz CPU clock, OS: RedHat Enterprise Linux 6 of x86_64 architecture[3].

minimax Intel Core i5-2410M quad-core CPU, 2.3GHz CPU clock, OS: FreeBSD/amd64 9.0-STABLE; and

reseaux Intel Atom N270 dual-core CPU, 1.6GHz CPU clock, OS: FreeBSD/i386 8.3-RELEASE.

Table 2 shows the test results of each function call between the 32-bit TinyMT (*tinymt32*) and stock *random* modules. We observe that TinyMT functions are faster than the stock functions in the overall wall-clock time for generating 10^6 numbers on 64-bit (Linux/x86_64 and FreeBSD/amd64) architectures, while TinyMT functions are $3 \sim 4$ times slower than the stock functions on 32-bit (FreeBSD/i386).

We have also tested *tinymt32* modules with HiPE option `o3` enabled on all the three systems. We have found that the overall execution speed measured from the wall-clock time gets faster to those of the non-HiPE version at the rates of ×1.25 on FreeBSD/i386 and $\times 2.4 \sim 3.6$ on Linux/x86_64 and FreeBSD/amd64.

[1] With `-enable-hipe` and `-disable-native-libs` compilation options.

[2] gcc 4.2.1 for FreeBSD (**minimax**/**reseaux**), gcc 4.4.6 for **laurel**.

[3] On Kyoto University ACCMS Supercomputer System B cluster; the test programs were executed from the batch queuing system to assign individual run-time nodes to reduce the interference of other running programs.

Accumulated execution time of 1000 executions [ms]			
function name	avg (ratio)	min	max
`random:uniform_s/1`	1.18 (1.00)	1.01	1.23
`random:uniform_s/2`	3.44 (2.92)	3.01	3.56
`tinymt32:uniform_s/1`	7.32 (6.20)	7.01	7.91
`tinymt32:uniform_s/2`	6.50 (5.51)	6.01	6.83
`tinymt32_nif:uniform_s/1`	1.27 (1.08)	1.00	1.51
`tinymt32_nif:uniform_s/2`	1.09 (0.92)	1.00	1.14

(measured by *fprof* with Erlang R15B01 without HiPE at the batch execution nodes of **laurel**.)

Table 3. Accumulated time comparison of per-call execution time for the random number generation of 10^6 integers (`uniform_s/1`) and floats (`uniform_s/2`).

```
-spec uniform_s(#intstate32{}) ->
        {float(), #intstate32{}}.

%% 0.0 <= value < 1.0
uniform_s(R0) ->
    R1 = next_state(R0),
    {temper_float(R1), R1}.
```

Figure 2. Definition of `uniform_s/1` in Erlang.

After the detailed profiling with the *fprof* facility of Erlang/OTP (see Table 3, however, we observe the accumulated [4] execution time of each *tinymt32* function measured by *fprof* test tool of Erlang/OTP takes $2 \sim 6$ times of that with the stock *random* module.

4.2 NIF implementation for TinyMT

We decide to make a NIF implementation of TinyMT called *tinymt32_nif* with the same function set of *tinymt32* as listed in Table 1, according to the following observations:

- Function calls from BEAM to NIFs have the own overhead time and the number of calls should be minimized, especially when each function does not take more time to execute. For example, merging sequential calls of `next_state/1` and `temper/1` inside `uniform_s/1` roughly cut the per-function execution time in half, since the two functions have to be executed together for generating a new output and updating the internal state (Fig. 2).
- 32-bit integer operations will be much efficiently implemented in C than in Erlang. We observed 32-bit integer operation on BEAM became slow when the BEAM runs on a 32-bit architecture such as i386 (See Table 2). Erlang Efficiency Guide [3, Section 10.1] tells the 32-bit BEAM only accepts 28-bit integers as the *Small Integers* which fit in a single word; integers required more bits are stored and computed as *Big Integers*. This problem does not occur on the 64-bit architecture BEAM.
- NIF implementation is only effective for the functions called many times. We decided to leave the seeding and initialization functions of TinyMT as the pure Erlang code.

Table 3 shows the test results of each function calls between the 32-bit TinyMT (*tinymt32*, *tinymt32_nif*) and stock *random* modules. We consider the test results are a logical consequence regarding the difference of the complexity of algorithm between TinyMT and AS183. We also observe the similar behavior under testing on **minimax** and **reseaux** execution environments, though the accumulated time results under FreeBSD interactive execution environments do not converge well as they did in a Linux batch execution environment, presumably due to the difference of OS scheduling and external disturbance by other interactive jobs.

[4] Shown as `ACC` time in the *fprof* results.

random	*sfmt-erlang*	*tinymt-erlang*
Algorithm		
AS183	SFMT	TinyMT
RNG generation period		
$\sim 2^{43}$	$2^{19937} - 1$	$2^{127} - 1$
Internal state in bits		
45	19968 (624×32)	223 (including 96 for the generation parameters)
Multiple generation parameters		
No	Yes (static only) Recompilation required	Yes (dynamic) TinyMTDC available
Accumulated time ratio of `uniform_s/1` at **laurel**		
1.00	5.88 (in pure Erlang) 0.61 (with NIFs)	6.20 (in pure Erlang) 0.93 (with NIFs)

Table 4. Comparison of specification and test results between *random*, *sfmt-erlang*, and *tinymt-erlang* implementations.

We should note that on *sfmt-erlang* the batch generation of the random number by `gen_rand_all/1` took ~ 2.5 [ms/1000 execs], and the conversion between the NIF Erlang binary format and the list format of the internal state by `intstate_to_list_max/2` took ~ 5.0 [ms/1000 execs] in **laurel** environment. These numbers are larger than those of *tinymt-erlang*, which is ≤ 1.51 [ms/1000 execs].

4.3 Evaluation and discussions

Table 4 shows a list of comparison between Erlang PRNG implementations discussed in this paper. The list suggests the following characteristics:

- For `uniform_s/1` in pure Erlang, *sfmt-erlang* and *tinymt-erlang* were both ~ 6 times slower than the *random* module.
- For `uniform_s/1` with NIFs, *sfmt-erlang* was $\times 1.52$ faster than *tinymt-erlang*, based on the average execution time. This is a logical consequence considering the fact that the batch generation of multiple random numbers as a list is performed in *sfmt-erlang*, while on *tinymt-erlang* only one random number is generated for each function call.
- Introducing the NIFs made the code execution speed $7 \sim 10$ faster than the base pure Erlang code running on the BEAM interpreter.

Table 2 suggests HiPE compilation is effective on reducing overall execution time. We think this is a logical consequence because the TinyMT algorithm largely depends on bit-manipulating instructions [5, 7].

Two out of three initial assumptions about the characteristics of TinyMT at Section 3.1 are proven false, comparing the performance test results of *sfmt-erlang* and *tinymt-erlang*:

- The internal state size is mostly irrelevant to the performance.
- While TinyMT is inherently *simpler* than SFMT, that does not necessarily mean TinyMT is *faster* for Erlang. We think this is due to the execution overhead time of each function in BEAM.

On the other hand, the performance test results also suggest that both the pure Erlang and the NIF versions of *sfmt-erlang* and *tinymt-erlang* have roughly the same execution time and equally useful. For an application which requires a very long period and no

concurrent/parallel random number generation, *sfmt-erlang* is the choice.

For an application which requires multiple independent streams for concurrent/parallel processes, however, *tinymt-erlang* still has the advantage to *sfmt-erlang* of being able to guarantee the generation of the independent streams simply by choosing different sequence parameter sets for each stream and/or applying the seed jumping function for the same sequence parameter for each stream so that each stream will not overlap with each other. Using independent pre-computed sequence parameters [5] for each CPU core or Erlang process will automate the procedure of assignment of the sequence parameters while retaining the reproducibility of mathematical simulations and random number generation speed by each CPU core or Erlang process.

5. Related work

Saito and Matsumoto [14] publish Mersenne Twister for Graphic Processor (MTGP), an implementation of MT for GPU CUDA environment. They also compare MTGP and TinyMT at a GPU environment and showed TinyMT takes $\times 1.07$ execution time than that of MTGP, though MTGP fails on some tests of TestU01.

We conducted generating 2^{28} sets of TinyMT generation parameters with TinyMTDC on both 32-bit and 64-bit tempering parameters. We performed the computation on Kyoto University ACCMS Supercomputer Thin cluster [6] with two nodes of 32 CPU cores in total. Each core was assigned independent computation space to maximize the efficiency of the parallel execution. The total computation time was $\sim$ 32 days, where the computation time ratio between the 32-bit and 64-bit tempering algorithms was 1:5. The result showed $18 \sim 19$ generation parameter sets with 32-bit tempering parameters were computed per second for each core, which is roughly equal to the results of Saito and Matsumoto [16].

Wichmann-Hill 2006 algorithm [22] is designed as the succeeding algorithm to AS183. The algorithm has the internal state of four 31-bit integers, and the generation period is $\sim 2^{120}$. The algorithm also has a seed generation method for generating non-overlapping random number sequences in parallel, though the proof of output independency has not been given as firm as the one for TinyMT. Truog [20] implemented a Big Integer version of this algorithm for Erlang by directly representing the internal state as a 124-bit integer.

6. Conclusion and future works

We have presented *tinymt-erlang*, an implementation of TinyMT PRNG for Erlang and the test results in pure Erlang code. The results suggest TinyMT will be a good candidate as an alternative PRNG for Erlang as well as *sfmt-erlang* is, regarding the increased period of the RNG and the benefit of generating parallel independent random number streams. The performance issues of TinyMT can be practically solved by assigning multiple CPU cores for the PRNGs and implementing NIFs for batch generation of multiple random numbers.

We plan the following future works for *tinymt-erlang*:

Performance improvement More execution speed improvement by applying NIFs for batch generation of multiple random numbers. Since batch generation will consume more time in the NIFs, the trade-off between the speed improvement and the absolute execution time of each NIF should be considered.

Multiple generation parameters Providing functions for supplying the sequence generation parameters, either from a pre-computed list or calling an external DC program. Supplying affordable number of the parameter sets is practical for applications which do not require massive number of sets ($< 10^4$). The speed of TinyMTDC is $\sim$ 19 sets per second on a 64-bit server-class computer such as **laurel**, and the code will not be easily translated into Erlang while preserving the execution speed, so calling TinyMTDC C++ code through an Erlang port will be a feasible solution.

Seed jumping Providing functions for the seed jumping, either by calling an external program, or as an Erlang piece of code. Seed jumping is useful when the number of usable generation parameter sets is limited. We consider calling a seed jumping program written in C through an Erlang port will be a more feasible solution than writing it in Erlang and the NIFs, regarding the complexity of the seed jumping algorithm.

Source code availability

The source code and documentation of *tinymt-erlang* is available at `https://github.com/jj1bdx/tinymt-erlang/` on GitHub. It is provided under the BSD License.

Acknowledgments

We thank Mutsuo Saito for his constructive comments to the development of the *tinymt-erlang* software. We also acknowledge feedback from anonymous reviewers.

We used the supercomputer service provided by Academic Center for Computing and Media Studies (ACCMS), Kyoto University for writing this paper.

References

[1] B. A. Wichmann and I. D. Hill. Algorithm AS 183: An Efficient and Portable Pseudo-Random Number Generator. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 31(2):188–190, 1982.

[2] B. A. Wichmann and I. D. Hill. Correction: Algorithm AS 183: An Efficient and Portable Pseudo-Random Number Generator. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 33(1):123, 1984.

[3] Ericsson AB. Erlang Efficiency Guide. `http://www.erlang.org/doc/efficiency_guide/advanced.html`.

[4] M. Gebetsroither. Please add sfmt_pure.erl as tested alternative — sfmt-erlang issue #5. `https://github.com/jj1bdx/sfmt-erlang/issues/5`.

[5] P. Gustafsson and K. Sagonas. Native code compilation of Erlang's bit syntax. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, ERLANG '02, pages 6–15, New York, NY, USA, 2002. ACM. ISBN 1-58113-592-0. doi: 10.1145/592849.592851. URL `http://doi.acm.org/10.1145/592849.592851`.

[6] P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33 (4), Aug. 2007. ISSN 0098-3500. doi: 10.1145/1268776.1268777. URL `http://doi.acm.org/10.1145/1268776.1268777`.

[7] D. Luna, M. Pettersson, and K. Sagonas. HiPE on AMD64. In *Proceedings of the 2004 ACM SIGPLAN workshop on Erlang*, ERLANG '04, pages 38–47, New York, NY, USA, 2004. ACM. ISBN 1-58113-918-7. doi: 10.1145/1022471.1022478. URL `http://doi.acm.org/10.1145/1022471.1022478`.

[8] M. Matsumoto and T. Nishimura. Dynamic creation of pseudorandom number generators. In *Monte Carlo and Quasi-Monte Carlo Methods*

[5] As we describe in Section 5, we have computed $2^{28} \simeq 2.68 \times 10^8$ sets of the sequence parameters for TinyMT, which is plausibly sufficiently large for the cases where the number of CPU cores are no more than 10^7.

[6] The Thin cluster had the following execution environments for each node: AMD Opteron 8350 $\times 4$ (16 CPU cores), 2.3GHz CPU clock, RedHat Enterprise Linux AS V4 of x86_64 architecture. TinyMTDC was compiled by gcc 4.4.6 and NTL was compiled by gcc 3.4.6. The cluster was running until March 2012 and replaced by the **laurel** cluster in May 2012.

1998, pages 56–69. Springer, 2000. URL http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/articles.html.

[9] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, January 1998. ISSN 1049-3301. doi: http://doi.acm.org/10.1145/272991.272995. URL http://doi.acm.org/10.1145/272991.272995.

[10] M. Papadakis and K. Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, Erlang '11, pages 39–50, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0859-5. doi: 10.1145/2034654.2034663. URL http://doi.acm.org/10.1145/2034654.2034663.

[11] Python Software Foundation. Python Programming Language. http://www.python.org/.

[12] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2012. URL http://www.R-project.org/. ISBN 3-900051-07-0.

[13] K. Rikitake. SFMT pseudo random number generator for Erlang. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, Erlang '11, pages 78–83, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0859-5. doi: 10.1145/2034654.2034669. URL http://doi.acm.org/10.1145/2034654.2034669.

[14] M. Saito and M. Matsumoto. A Variant of Mersenne Twister Suitable for Graphic Processors. *CoRR*, abs/1005.4973, 2010. http://arxiv.org/abs/1005.4973.

[15] M. Saito and M. Matsumoto. SIMD-Oriented Fast Mersenne Twister: a 128-bit Pseudorandom Number Generator. In A. Keller, S. Heinrich, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2006*, pages 607–622. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-74496-2.

[16] M. Saito and M. Matsumoto. A high quality pseudo random number generator with small internal state. *IPSJ SIG Notes*, 2011(3):1–6, Oct. 2011. URL http://ci.nii.ac.jp/naid/110008620834/en/.

[17] M. Saito and M. Matsumoto. Tiny Mersenne Twister (TinyMT). http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/TINYMT/index.html.

[18] V. Shoup. NTL: A Library for doing Number Theory. http://www.shoup.net/ntl/.

[19] The Open Source Initiative. The BSD 3-Clause License. http://www.opensource.org/licenses/bsd-3-clause.

[20] M. Truog. big-integers implementation of Wichmann-Hill 2006 algorithm — sfmt-erlang issue #3. https://github.com/jj1bdx/sfmt-erlang/issues/3.

[21] User thomasc at GitHub. Use sfmt-erlang for random number generation — proper issue #34. https://github.com/manopapad/proper/pull/34.

[22] B. A. Wichmann and I. D. Hill. Generating good pseudo-random numbers. *Comput. Stat. Data Anal.*, 51:1614–1622, December 2006. ISSN 0167-9473. doi: 10.1016/j.csda.2006.05.019. URL http://portal.acm.org/citation.cfm?id=1219162.1219278.