Efficient Use of Parallel PRNGs on Heterogeneous Servers

André Pereira[*] and Alberto Proença[*]
[*]Algoritmi Centre, University of Minho, Campus de Gualtar, 4710-057 Braga, Portugal
{ampereira, aproenca}@di.uminho.pt

***Abstract*—Scientific code often requires very large amounts of independent streams of random numbers with a Gaussian distribution for stochastic simulations. This code typically uses high statistical quality Pseudo-Random Number Generators (PRNGs) and distribution transformation algorithm. Scientific data analyses developed for the ATLAS Experiment at CERN were used to test and validate our approaches to an efficient parallel PRNG, with a Gaussian distribution in a multicore server with a GPU accelerator. This paper evaluates the performance of our approaches in ATLAS data analyses and presents a comparative performance evaluation among different implementations of popular PRNG algorithms available in ROOT, MKL, and PCG libraries, on an heterogeneous compute server, showing the positive impact of a GPU accelerator to generate large amounts of PRN streams.**

***Index Terms*—Pseudo-Random Number Generation, Efficient Parallel PRNG, Scientific Computing, Code Execution Efficiency, Performance Analysis, High Performance Computing.**

I. Introduction

Scientific data analysis usually aim to test hypothesis and theories or simulate phenomena. Computing tasks of these applications often require sampling of measured values within their margin of error, Monte Carlo algorithms or randomness associated with specific scientific phenomena, which may account for a significant portion of the overall execution time. This need for randomness in the deterministic environment of computer science created the demand for algorithms that provide seemingly random numbers.

Pseudo-random number generation, PRNG, the process of generating apparently random numbers on digital chips, is a well studied topic, with the first computer based algorithms being suggested as early as 1951 [1]. There are several PRNGs available with excellent statistical quality, as well as implementations on various programming environments with reasonable performance. However, the generator performance is often overlooked by non-computer scientists, which may lead to a significant application performance degradation.

Three main aspects should be considered when selecting a PRNG for a scientific application: the statistical quality, which is out of the scope in this work, the computational performance of the algorithm/implementation and the way that a given implementation is used in the code. These aspects are critical specially for parallel code executing on multicore and manycore compute servers, where algorithmic and computational inefficiencies may lead to significant performance and scalability bottlenecks.

This paper presents a performance evaluation of different implementations of a popular PRNG, the Mersenne Twister [2], as well as Gaussian distribution transformation algorithms, such as Inverse Transform Sampling and Box-Muller [3], available in popular scientific libraries. It also provides an insight on the best way to use these implementations, comparing three different approaches on two real parallel applications with different amounts of pseudo-random numbers (PRNs), related to the search of the Higgs boson [4]. A PRNG from the permuted congruential generator (PCG) family [5] was also evaluated, as the authors claim that it performs better than any other algorithm, although it is not yet fully accepted by the scientific community.

This paper is structured as follows: section II presents the two case studies used to evaluate the different PRNGs and their implementations; section III contextualises the generation of random numbers, presenting the most popular PRNGs, the distribution transformations and the different approaches to use them in a parallel environment; section IV evaluates the different PRNG implementations in the case studies; section V makes a critical analysis of the developed work with suggestions for further improvements.

II. Pipelined Scientific Data Analyses

A scientific data analysis is a process that converts raw scientific data (often from experimental measurements) into useful information to answer questions, test hypotheses or prove theories. When dealing with large amounts of experimental data, data is read from one or more files in variable sized chunks or datasets, and placed into an adequate data structure.

Parallel implementations of these analyses, where concurrent threads process different dataset elements, are often used in pipelined scientific data analyses, with few data dependencies among dataset elements. The overall analysis performance can be improved with an adequate balance between reading data from disk and data processing, exploiting some pipeline features and identifying and minimising code bottlenecks. Among these latter, the use of PRNGs often play a significant role in performance degradation and this is the key subject of this paper.

High energy physics scientists at the ATLAS Experiment [6] at CERN developed a pipelined scientific data analysis code, the $t\bar{t}H$ analysis, to study the associated production of top quarks with the Higgs boson, following head-on proton-proton collisions (known as events) at the Large Hadron Collider (LHC). The final state of an event is recorded by the ATLAS particle detector, which measures the characteristics of the bottom quarks (detected as jets of particles due to a hadronization process) and leptons (both muons and electrons), but not the neutrinos, as they do not interact with the detector sensors. This final state is presented in figure 1.

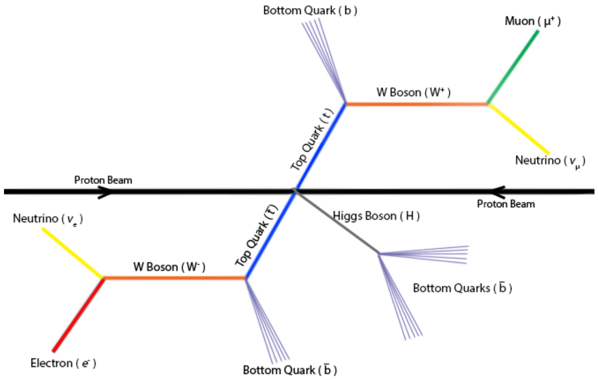

Fig. 1. Schematic representation of the $t\bar{t}$ system and Higgs boson decay.

$t\bar{t}H$ analytically computes the characteristics of the neutrinos with known information, to reconstruct at the end of the data processing pipeline both top quarks and the Higgs boson [4]. This process, known as kinematic reconstruction, tests every combination of bottom quarks and leptons, which are stored in a specific structure in predefined files provided by the experiments at the LHC.

Two compute-bound variants of the $t\bar{t}H$ analysis were considered as representative case studies:

- one reconstructs the top quarks and the Higgs boson, the `ttH_sci` (*sensors with a confidence interval*), assuming $\pm 1\%$ accuracy of the sensors in the ATLAS detector; it performs an extensive sampling within the 99% confidence interval in the kinematic reconstruction, from which only the best reconstruction is considered; this version works with 1024 samples, where each requires the generation of 30 different PRNs, to a total of 30Ki PRNs per event;
- the other is similar to the previous, but where two pipeline stages were replaced, `ttH_scinp` (`sci` *with a new pipeline*), to perform different operations on each data element, maintaining the same overall inter-stage dependencies and a similar sampling of the confidence interval of `ttH_sci`; this version requires less PRNs, 10Ki per event.

The PRNG used by default by these data analyses is the Mersenne Twister implementation provided by the ROOT framework [7], transforming the uniform distribution of the PRNs into a Gaussian distribution by the Box-Muller algorithm.

III. Random Number Generation

Random numbers are used in a wide spectrum of applications where unpredictability is required, including statistical data sampling, scientific computing, gaming and cryptography. Different applications often require specific properties from a random number, for which different random number generators may be used. In the context of computer science, these can be broadly classified as True Random Number Generators (TRNGs) or Pseudo-Random Number Generators (PRNGs).

TRNGs are based in physical random processes to generate random bit strings, which may have to be preprocessed to remove possible bias. The most common example of a TRNG is the coin toss of a symmetrical coin, where one can expect either heads or tails with a 50% certainty. A set of coins, or a series of coin tosses, can be used to generate a random sequence of bits. However, coins are not perfectly balanced and there is a small probability of landing on its side, slightly deviating the 50-50 chances of expecting heads or tails. Post-processing may be used to remove the bias of these processes. There are no correlations among generated numbers but these generators are usually slow, not suited for large scale computing and their results cannot be replicated, which makes debugging code harder.

PRNGs attempt to approximate the properties of truly random numbers, such as no repetition of sequence of values for a long period and no correlation between generated numbers. The generated values are not truly random as they are determined by an initial value (seed). A proper mathematical analysis of the generator algorithm is required to assess its quality and if they are close enough to truly random for the specific use that they were designed for. The main benefit of this type of random generator is the performance, which, depending on the algorithm, may be able to speedup with the amount of available cores. The use of a seed also eases the process of debugging code. With an proper algorithm, this type of generator is mostly used for scientific applications due to its higher performance and adequate mathematical properties. However, most PRNGs can only generate sets of uniformly distributed PRNs, which may require a transformation algorithm to convert them into another PRN distribution.

A short introduction to the most popular PRNGs and distribution transformations follows through the next subsections, as well as the most used libraries by the scientific community.

A. *Popular PRNG Algorithms*

There is a wide range of algorithms to generate PRNs currently available, each with strengths and weaknesses that may make them best suited for different uses. The quality of a PRNG randomness is usually evaluated by a set of benchmarks, such as the Diehard [8] and TestU01 [9] suites. An ideal PRNG has an infinite period, covers the entire range of possible PRNs (usually 32/64-bit numbers), and has no correlation between generated PRNs. Other mathematical characteristics may be equally important, but are not as relevant in the scientific community when choosing a PRNG.

The scientific community has been using several PRNG algorithms, such as the r1279 and Wichmann-Hill PRNG available in GSL [10], MKL [11], and NAG [12], but one stands above all other in popularity: the Mersenne Twister [2]. This algorithm was developed in 1997 and features a period of $2^{19337} - 1$, passes most statistical tests, and it is extremely fast to generate both 32 and 64-bit numbers. This generator is also implemented in most languages and available in most scientific computing libraries. There are limitations, such as low throughput, but they are often overcome by alternative implementations of this algorithm, which take advantage of vector/SIMD instructions, GPU architectures and multithreaded environments.

Recently, a PCG family of PRNGs was proposed [5], claiming better statistical quality and performance, for both single and multithreaded environments. Even though it is not yet fully accepted by the scientific community, the PCG RXS-M-XS 64 generator (a Linear Congruential Generator, LCG) will be included in our performance evaluation, alongside Mersenne Twister, in section IV, as the authors claims it is one of the best performing PRNGs currently available. Since the PCG generators only generate uniformly distributed numbers, they will be paired with an efficient implementation of the Box-Muller algorithm available in the ROOT framework [7].

B. Transforming Uniformly Distributed PRNs

PRNs are usually generated in an uniform distribution, but other distributions may be required. Gaussian distributed PRNs are often used in scientific computing, so having PRNG implementations that support that functionality is crucial.

Since most algorithms only generate uniformly distributed PRNs, this distribution may require post processing. One of the most common algorithms is the Box-Muller transformation [3], which generates a pair of independent Gaussian distributed PRNs based on a set of uniformly distributed numbers. It is not one of the most computationally efficient transformations, due to its iterative nature and reliance on square roots, logarithmic and trigonometric functions.

The Inverse Transform Sampling[1] is a method that transforms uniformly distributed numbers into any distribution, given its Cumulative Distribution Function (CDF). The CDF maps a PRN into a probability between 0 and 1 and then inverts this function, providing the final non-uniformly distributed number. This number can be adjusted to a specific mean and standard deviation afterwards, as required by a Gaussian distribution. The lack of an analytic CDF for the Gaussian distribution may affect the algorithm performance, favouring other transformations such as the Box-Muller. However, current implementations, widely accepted by the scientific community, use an extremely accurate approximation of the Gaussian CDF, which is faster than most transformations.

The computational performance of both Box-Muller and Inverse Transform Sampling methods (with the CDF approximation) will be assessed and evaluated on real scientific case studies. Other methods could be used, such as the Ziggurat transformation [13], but are not in the scope of this work as they are not used as often by the scientific community.

C. PRNG Libraries

Most scientific computing libraries and frameworks provide efficient implementations of a wide variety of PRNGs.

In the context of the particle physics community, related to the case study presented in section II, the most popular scientific libraries are provided in the ROOT framework. This framework only offers the Mersenne Twister PRNG with the Box-Muller transformation and is used by default in the two case study variants.

MKL is one of the most popular scientific computing libraries which offers a wide range of relevant functionalities. It features several PRNGs, from which only the Mersenne Twister will be considered, as it would be the most likely to be used by the scientific community. The Box-Muller and ICDF (Inverse Transform Sampling) transformations are available in this library and will be used to convert uniformly distributed PRNs into a Gaussian distribution. MKL also provides the option to generate a batch of PRNs, which will also be tested in section IV.

The fastest PRNG available in the PCG family, the RXS-M-XS 64 (LCG), will be coupled to the Box-Muller implementation available in ROOT to provide Gaussian distributed pseudo-random numbers.

To offload the PRNG to the CPU accelerator the NVidia CUDA toolkit includes a library of PRNGs, cuRAND [14]. It provides an efficient implementation of the Mersenne Twister algorithm and the Box-Muller transformation.

A request for a new PRN in a parallel environment can follow several approaches:

- a single PRNG to feed all concurrent threads, where each PRNG execution is atomic; results would not be reproducible as PRNs consumed by each thread varies between runs [15]; it does not support concurrent execution of the PRNG;
- a single PRNG to feed each stream request using a transition function to guarantee that there are no correlations among streams, known as leapfrog; used in the cuRAND implementation of Mersenne Twister [16]; it supports concurrent execution;
- a single PRNG to feed all concurrent threads with a different a precomputed seed for each stream, causing the generated PRNs to be equally spaced in the overall PRN sequence, which may be slow as shown in [17], known as splitting; it supports concurrent execution;
- an independent PRNG per compute thread initialised with different sets of parameters; if these parameters are not adequate, streams may not be truly independent, as referenced in [18]; the most common and portable approach used in scientific code.

An implementation of a PRNG on a library is as important to the overall scientific application performance as the approach used to interact with the PRNG itself in the

[1] Available at https://en.wikipedia.org/wiki/Inverse_transform_sampling.

code. For instance, one can generate all PRNs upfront, or request a PRN when needed, and these approaches will have a different performance impact depending on the application and execution environment, specially in parallelized code where there may be multiple threads accessing shared PRNs and/or PRNGs.

The parallel implementation of the two variants of our case study was modified to evaluate three different approaches to manage the generation of PRNs:

- to call the PRNG whenever a single PRN is needed;
- to generate a batch of PRNs and store the result in a thread private buffer; when a PRN is needed the compute thread removes it from the buffer; when the buffer is empty, a new batch is requested;
- to generate a batch of PRNs and store the result in a thread private dual-buffer; while the one buffer is being consumed, the other is being filled.

Preliminary results showed that sharing buffers among compute threads degrades performance, due to contention when accessing shared resources.

This dual-buffer approach minimises the overhead of the PRNGs. Figure 2 illustrates this approach. In the case of the PRNG on GPU, this approach also hides the costs of memory transfers between the CPU and GPU memory. The case studies that were implemented with a single thread do not need a PRNG management thread.

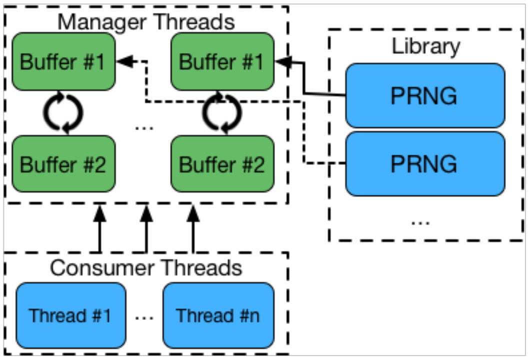

Fig. 2. Dual buffer implementation in the PRNG management threads.

To generate a single PRN at a time the cuRAND PRNG was not used, since the lack of parallelism and the overhead on memory transfers would greatly affect the performance. In the parallel implementations, each PRNG management thread in the multicore devices uses a different stream to launch kernels on the GPU and perform the memory transfers, ensuring that concurrent management threads can simultaneously generate and receive PRNs. Preliminary tests showed that for 24 computing threads (and associated management threads) the GPU device was not fully utilised, meaning that it could scale for a greater number of multicore threads.

IV. Results and Discussion

The testbed used for the quantitative evaluation of the PRNGs was a dual socket server with 12-core Intel Xeon E5-2695v2 Ivy Bridge devices, at 2.4 GHz [19] with 64 GiB RAM, coupled with one NVidia Tesla K20 with 2496 CUDA cores and 5GB of GDDR5 memory (Kepler architecture).

The two variants of the case study code, as described in section II, were `ttH_sci` and `ttH_scinp`. A k-best measurement heuristic was used to ensure that the results can be replicated, with $k = 5$ with a 5% tolerance, a minimum/maximum of 15/25 measurements. The multithreaded tests used 24 cores in the Xeon multicore devices, with 1 computing thread per core. The $t\bar{t}H$ analyses were tested with 128 files, each with $\pm 6,000$ events (dataset elements). In multithreaded tests each PRNG management thread uses an independent PRNG per compute thread.

The `ttH_sci` application, which is the most PRNG intensive, spent around 90% of the execution time calling the ROOT framework PRNG, while the application that required less PRNs, `ttH_scinp`, spent around 50% of the execution time.

Figure 3 compares the execution times to generate 10^6 PRNs of various implementations of the Mersenne Twister with the ROOT and MKL libraries, the former coupled with the Box-Muller (*BM*) transformation and the latter with the two available Box-Muller implementations and the Inverse Transform Sampling (*BM*, *BM2*, and *I*). MKL also offers implementations optimised to generate batches of PRNs (*A*). The PCG was coupled with the Box-Muller transformation.

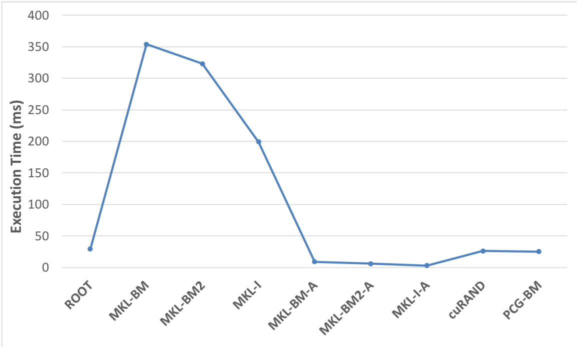

Fig. 3. Execution time of each PRNG to generate 10^6 PRNs.

This test shows that there is a small difference between the ROOT and PCG generator, with 29 and 25 milliseconds respectively, but the MKL batch generator using the Inverse Transform Sampling was able to generate these numbers in 3 milliseconds. This generator will be used as the default MKL generator on the next tests with the case studies. Offloading the PRNG to the GPU accelerator led to a similar performance to the PCG generator, considering PRNG initialisation, generation and memory allocation and transfers between the GPU and the multicore memories.

Figure 4 shows the speedup of the two encoded sequential versions of the data analyses (`ttH_sci` and `ttH_scinp`) with the selected PRNG algorithms and the different approaches detailed in subsection III-C compared to the original ROOT single number PRNG. For the `ttH_sci` application, approaches that use a single or dual PRN buffers are noted as *SB* and *DB*, respectively. In all Mersenne Twister PRNGs in multicore devices, the single and dual buffer approaches provide a slight performance improvement over generating a single pseudo-random number at a time. Offloading the PRNG to GPU provided speedups up to 3.8x, similar to the PCG PRNG. This approach benefits from a dual buffer approach the most as it hides the cost of memory transfers between CPU and GPU devices. The `ttH_scinp` application has a similar behaviour to `ttH_sci`, as it requires a high quantity of pseudo-random numbers. However, the PCG algorithm has a higher performance improvement when larger amounts of PRNs are required, and the same applies when offloading the PRNG into the GPU.

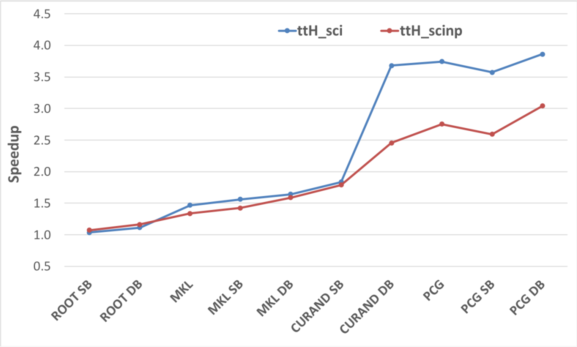

Fig. 4. Speedup of the sequential 2 data analyses with different PRNG algorithms and approaches *vs* the original ROOT single number PRNG.

Figure 5 shows the speedup of the two 24-threaded versions of the data analyses with the selected PRNG algorithms (one per physical core of the Xeon devices) and the different approaches for PRNG concurrent execution, compared to the original ROOT single number PRNG.

For the `ttH_sci`, which requires the most PRNs, the use of single or dual buffers approaches provide larger performance improvements than on the sequential code using only the multicore devices, specially for the PCG generator with a speedup improvement from 42x to 48x. The offload of the PRNG to the GPU devices, with dual buffers to hide the PRNG execution time and memory transfers, provides a performance improvement up to 70x over the original application with the ROOT single number PRNG. This speedup is due to the efficiency of the batch generation of pseudo-random numbers on GPUs, but also due to the higher availability of the Xeon cores since they were freed from generating PRNs, and in a compute-bound code this makes a difference. Worth mentioning is the fact that the GPU is not being fully used, which means that the codes can reach larger speedups if they require larger amounts of PRNs.

The `ttH_scinp` behaves similarly to `ttH_sci` but with smaller speedups, up to 20x using only the multicore devices. The use of a GPU device as a computing accelerator improves the performance up to 10x, but the overhead of the memory transfers and the less amount of pseudo-random numbers required by this application restricts the efficiency of this approach, when compared to the PCG PRNG with a dual buffer.

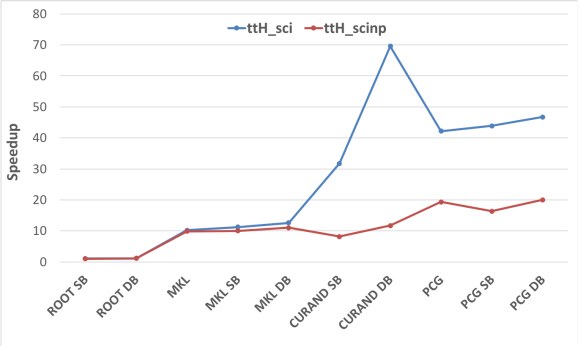

Fig. 5. Speedup of the 24-threaded data analyses with different PRNG algorithms and approaches *vs* the original ROOT single number PRNG.

The overhead of using the Box-Muller transformation with the PCG PRNG accounts for only 30.1% of the overall PRNG time for the `ttH_sci` application with 24 computing threads. However, it was not possible to profile MKL as the available libraries were not compiled with debugging symbols.

Both figures 4 and 5 prove that applications that require a huge amount of PRNs can greatly benefit by the use of efficient implementations of PRNG approaches. While the efficient use of the MKL library can provide significant performance improvements, the PCG PRNG tested was the best performing on multicore devices by a large margin. However, this may be considered an unfair comparison, since this PRNG algorithm is faster than the Mersenne Twister. It is the responsibility of the end user to assess if this PRNG should be used over other traditional PRNGs, which are well accepted and extensively tested by the mathematics's community.

V. Conclusions and Future Work

This paper presents an evaluation of the computational performance of different versions of a popular PRNG, the Mersenne Twister, available on MKL and ROOT libraries, comparing different implementations available in those libraries. It aims to provide an insight into the best way to use these generators with real software codes, evaluating the performance of their implementations. A PRNG of the PCG family was also included in this comparison with Mersenne Twister implementations, since the authors claim it has the best statistical quality and performance.

Two variations of a real scientific data analysis were used as case studies: `ttH_sci` and `ttH_scinp`, both compute-bound codes. `ttH_scinp` and `ttH_sci` require 10Ki and 30Ki PRNs per event (dataset element), with the tested dataset containing around 800K events. The PRNGs were used in these case studies as a single number generator, a batch generator to a thread-private buffer, and to a thread-private dual buffer, where one is filled while the other is being consumed. In both single and dual buffer approaches, the PRNG is managed by an additional thread per computing thread, so that data processing and PRNG can be concurrently executed.

An initial test of generating 10^6 PRNs with a Gaussian distribution of the various PRNG implementations available on MKL, ROOT, and PCG libraries, showed a clear disadvantage of using MKL PRNGs that only return a single number, rather than a batch generator. The batch MKL generator with the Inverse Transform Sampling method was 8x faster than the PCG and cuRAND generators, and 10x faster than ROOT. This implementation was used as the PRNG representative of the MKL library in the tests with the scientific applications. There was no significant performance difference between using GNU or Intel compilers.

The first set of tests only used sequential versions of the real scientific data analyses. The data analysis with the PCG PRNG with the dual buffer approach was the faster one, almost 4x faster than the original single number ROOT generator for the sequential version of the `ttH_sci` application, closely followed by cuRAND. The code with MKL PRNG did no beat any of the other codes. A similar behaviour is observed for `ttH_scinp`, but with smaller speedups, as this code variant requires fewer PRNs. For `ttH_sci` with 24 computing threads, the use of the GPU accelerator provided a speedup up to 70x with the dual buffer implementation, due to its faster PRNG and by freeing CPU resources to be used by the computing threads. Code with the PCG PRNG was again the faster one, 47x faster than ROOT, while MKL only displayed a 11x performance improvement.

Three main conclusions can be extracted from this analysis:

- the choice of an efficient implementation of a given algorithm is a key issue: both ROOT and MKL implement the Mersenne Twister but MKL is, at least, 10x faster;
- a performance analysis of these algorithms should not be made with a synthetic code, but rather with real code that requires these generators; although this may change with the applications, the fastest PRNGs on the initial tests were not the best when used in our real scientific case studies;
- the way these PRNGs are used in each application may have a significant impact on performance: the cuRAND dual buffer was 2.3x faster than the single buffer implementation.

This work focused mainly on the popular Mersenne Twister algorithm, but others could benefit from this analysis, such as cryptographycally secure PRNGs. The SIMD-oriented Mersenne Twister [20] could also be tested, but it is expected that an efficient SIMD implementation for Intel CPU devices is provided by MKL, as proved by the initial benchmark results. Other, possibly faster, distributions and distribution transformations could be tested, such as the Ziggurat (which was not available in the tested libraries), since the measured Box-Muller execution time takes up to 30% of the overall PCG PRNG execution time.

Acknowledgments

This work has been supported by COMPETE POCI-01-0145-FEDER-007043 and FCT - Fundação para a Ciência e a Tecnologia within the Project Scope UID/CEC/00319/2013 and by Search-ON2: Revitalization of HPC infrastructure of UMinho, (NORTE-07-0162-FEDER-000086), co-funded by ON.2-O Novo Norte under NSRF through ERDF. The first author is also sponsored by FCT grant SFRH/BD/119398/2016.

References

[1] J. Von Neumann, “Various Techniques Used in Connection With Random Digits,” *Appl. Math Ser*, vol. 12, no. 36-38, p. 3, 1951.

[2] M. Matsumoto and T. Nishimura, “Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator,” *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[3] E. Golder and J. Settle, “The Box-Muller Method for Generating Pseudo-Random Normal Deviates,” *Applied Statistics*, pp. 12–20, 1976.

[4] ATLAS Collaboration, “Observation of a New Particle in the Search for the Standard Model Higgs Boson with the ATLAS Detector at the LHC,” *Phys.Lett.*, 2012.

[5] M. E. O’Neill, “PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation,” Harvey Mudd College, Claremont, CA, Tech. Rep. HMC-CS-2014-0905, Sep. 2014.

[6] T. A. Collaboration, “The ATLAS Experiment at the CERN Large Hadron Collider,” *Journal of Instrumentation*, 2008.

[7] F. Rademakers, P. Canal, B. Bellenot, O. Couet, A. Naumann, G. Ganis, L. Moneta, V. Vasilev, A. Gheata, P. Russo, and R. Brun, “ROOT.” [Online]. Available: http://root.cern.ch/drupal/

[8] G. Marsaglia, “The Marsaglia Random Number CDROM Including the Diehard Battery of Tests of Randomness,” *http://www.stat.fsu.edu/pub/diehard/*, 2008.

[9] P. L’Ecuyer and R. Simard, “TestU01: AC Library for Empirical Testing of Random Number Generators,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 4, p. 22, 2007.

[10] B. Gough, *GNU Scientific Library Reference Manual*. Network Theory Ltd., 2009.

[11] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, “Intel Math Kernel Library,” in *High-Performance Computing on the Intel® Xeon Phi^{TM}*. Springer, 2014, pp. 167–188.

[12] N. A. Group and N. A. G. L. (Oxford), *Fortran Library Manual*. Numerical Algorithms Group, 1988, vol. 3.

[13] G. Marsaglia, W. W. Tsang *et al.*, “The Ziggurat Method For Generating Random Variables,” *Journal of Statistical Software*, vol. 5, no. 8, pp. 1–7, 2000.

[14] C. Nvidia, “Curand library,” 2010.

[15] D. R. Hill, C. Mazel, J. Passerat-Palmbach, and M. K. Traore, “Distribution of Random Streams for Simulation Practitioners,” *Concurrency and Computation: Practice and Experience*, vol. 25, no. 10, pp. 1427–1442, 2013.

[16] M. Saito and M. Matsumoto, “A Deviation of CURAND: Standard Pseudorandom Number Generator in CUDA for GPGPU,” in *Proceedings of 10th International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, 2012.

[17] T. Bradley, J. du Toit, R. Tong, M. Giles, and P. Woodhams, “Parallelization Techniques for Random Number Generators,” in *GPU Computing Gems Emerald Edition*. Elsevier, 2011, pp. 231–246.

[18] K. Claessen and M. H. Pałka, “Splittable Pseudorandom Number Generators Using Cryptographic Hashing,” in *ACM SIGPLAN Notices*, vol. 48, no. 12. ACM, 2013, pp. 47–58.

[19] Intel, “Intel Xeon Processor E5 v2 Family: Datasheet,” Intel Corporation, Tech. Rep., 2013.

[20] M. Saito and M. Matsumoto, “SIMD-Oriented Fast Mersenne Twister: A 128-bit Pseudorandom Number Generator,” in *Monte Carlo and Quasi-Monte Carlo Methods 2006*. Springer, 2008, pp. 607–622.