PRNG-Broker: A High-Performance Broker to Supply Parallel Streams of Pseudo-Random Numbers for Large Scale Simulations*

Andre Pereira[1] and Alberto Proenca[1]

Algoritmi Centre, Department of Computer Science
Universidade do Minho, Gualtar, 4710-059, Portugal,
ampereira,aproenca@di.uminho.pt

Abstract. The generation of large streams of Pseudo-Random Numbers may lead to performance degradation in simulation applications. Both the PRN Generator and how it is used impact the efficiency of generating multiple PRNs.
A PRNG-Broker was developed for parallel servers with/without accelerators, which transparently manages the efficient execution of PRNG implementations from CPU/GPU libraries, with an intuitive API that replaces the user PRNGs requests. PRNG-Broker allows the development of efficient PRN intensive applications without the need of explicit parallelization and optimization of PRNG tasks.
It was validated with scientific analyses of proton beam collisions from CERN, which require 30 Ki PRNs per collision. Outcomes show a performance boost over the original code: 48x speedup on a 2*12-core server and over 70x speedup when using a GPU.

Keywords: Pseudo-Random Number Generation, Efficient Parallel PRNG, Scientific Computing, Software Library, High Performance Computing

1 Introduction

Scientific data analyses are developed to test, validate, and simulate hypothesis, theories, and phenomena. These applications often rely on tasks that operate on large sets of measured data with an associated measurement uncertainty and that may be computationally intensive. A common strategy to limit this uncertainty is to sample the data within its margin of error, often resorting to Monte Carlo algorithms, which may account for a significant portion of the overall execution time. This need for randomness in a deterministic environment 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 [24]. There are several

* This work has been supported by FCT – Fundacao para a Ciencia e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

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 key issues should be considered when selecting a PRNG for a scientific application that requires very large sets of PRNs: the statistical quality, which is out of the scope in this work, the computational performance of the algorithm and its implementation, and how an implementation is used in the code. These issues are particularly critical in parallel environments, such as in multicore and manycore compute servers, where algorithmic and computational inefficiencies may lead to significant performance degradation and poor scalability.

This paper presents PRNG-Broker, a *middleware* that manages the interaction of the user code with efficient implementations of popular PRNGs provided by widely adopted libraries in the scientific community. It transparently implements a dual-buffer approach for efficient management of large sets of PRNs, which can be natively generated in multicore and manycore servers, as well as offloaded to hardware accelerators and other servers. This paper also evaluates different approaches to use PRNGs in these libraries and their performance on different devices, detailing their PRN throughput and possible memory transfer costs associated to distributed memory environments. The analysis used as case studies three real parallel applications related to the search of the Higgs boson [2], which required different amounts of PRNs.

The PRNG-Broker currently supports different implementations of the popular Mersenne Twister PRNG [16], provided by the ROOT [19], MKL [25], STL [23], and cuRAND [17] libraries, but this list can be easily extended in future releases. A PRNG from the permuted congruential generator (PCG) family [18] was also evaluated, as the authors claim that it performs better than any other algorithm; however, it is not yet fully accepted by the scientific community.

This paper is structured as follows: section 2 contextualises the generation of random numbers, presenting the most popular PRNGs, the distribution transformations and the different approaches to use them in parallel multicore/manycore servers with accelerator devices; section 3 provides an in-depth analysis of the PRNG-Broker library; section 4 presents the three case studies used to evaluate the different PRNGs and their implementations; section 5 evaluates the different PRNG implementations in the case studies; section 6 makes a critical analysis of the developed work with suggestions for further improvements.

2 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 random numbers, for which different random number generators may be used. In this context, 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. 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. 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. However, the generated values are not truly random as they are determined by an initial value (seed, which ensures result reproducibility). The main benefit of this type of random generator is their performance, which, depending on the algorithm, may scale with the increase of available cores. This type of generators are mostly used in scientific applications due to its higher performance and adequate mathematical properties. A short introduction to the most popular PRNGs, distribution transformations, and libraries follows through the next subsections.

2.1 Popular PRNG Algorithms

A wide range of algorithms to generate PRNs is 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 [14] and TestU01 [11] suites.

The scientific community has been using several PRNG algorithms , but one stands above all other in popularity: the Mersenne Twister [16]. This algorithm was developed in 1997 and features a period of $2^{19937} - 1$, passes most statistical tests, and is extremely fast to generate both 32 and 64-bit numbers. It has some limitations, such as low throughput, but these are often overcome by alternative implementations of this algorithm, which take advantage of vector/SIMD instructions, GPU architectures and multithreaded environments.

Recently, the PCG family of PRNGs was proposed [18], claiming better statistical quality and computational 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 also be included in our performance evaluation in section 5, since the authors claim it is one of the best performing PRNGs currently available.

2.2 Transforming Uniformly Distributed PRNs

Most PRNGs only generate uniformly distributed PRNs, but other distributions may be required. A PRNG that supports post processing of uniformly distributed PRNs to generate different distributions is an essential feature that should be included: for instance, Gaussian distributed PRNs are often used.

Box-Muller [6] is a common algorithm to generate a pair of independent Gaussian distributed PRNs, based on a set of uniformly distributed numbers.

However, its computationally efficiency is flawed 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 PRN number can be afterwards adjusted to a specific mean and standard deviation, as required by a Gaussian distribution [1,5,8,10]. 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 this and Box-Muller methods will be assessed and evaluated on real scientific case studies.

2.3 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 4, the most popular scientific libraries are provided in the ROOT framework [19]. This framework only offers the Mersenne Twister PRNG with the Box-Muller transformation and is used by default in the three case study variants.

MKL [25] is one of the most popular scientific computing libraries that offers a wide range of mathematical functionalities, also featuring the Mersenne Twister. 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, whose performance is also evaluated in section 5.

The Standard Template Library (STL) [23] implements a small selection of PRNGs, from which the Mersenne Twister will be considered. However, it is not clear which algorithm STL uses to transform uniform numbers to follow a Gaussian distribution, as chosen by the developers of each C++ compiler.

PCG [18]only supports generation of uniformly distributed PRNs, so it has to be coupled with external implementations of distribution transformations.

PRNG offload to GPU devices is supported through the use of specific libraries, such as cuRAND [17], available in the the NVidia CUDA toolkit. It provides an efficient parallel implementation of the Mersenne Twister algorithm and the Box-Muller transformation.

The efficient use of PRNGs on parallel environments, such as multicore or GPU devices, requires multiple independent PRN streams. Several less popular libraries support PRN generation with multiple streams, such as clRNG [13] in OpenCL and RngStreams [12] in C. Adaptations of the latter library to work directly with OpenMP, MPI, and R are also available [9].

A new PRN generation in a parallel environment can follow these approaches:

- a single PRNG to feed requests from all concurrent consumer threads, where each PRNG is atomic; results are reproducible as PRNs consumed by each thread varies between runs [7]; no support for PRNG concurrent execution;

[1] Explained in https://en.wikipedia.org/wiki/Inverse_transform_sampling.

- a single PRNG to feed each consumer stream request, using a transition function to guarantee no correlations among streams, known as leapfrog; used in some cuRAND PRNGs [21]; supports concurrent execution;
- a single PRNG to feed requests from all concurrent consumer threads, with a different pre-computed seed for each stream, known as block splitting, causing the generated PRNs to be equally spaced in the overall PRN sequence, which may be slow as shown in [3]; it supports concurrent execution;
- an independent PRNG per consumer thread, initialised with different sets of parameters, known as parameterization; if these parameters are not adequate, streams may not be truly independent, as referenced in [4]; it is the most common and portable approach used in scientific code, and is implemented by most libraries, such as MKL and cuRAND, and also in libraries specialized for multicore and GPU devices [15, 22].

3 The PRNG-Broker

An implementation of a PRNG on a library is as important to the overall application performance as the approach used to interact with the PRNG itself. 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, namely in parallel code where multiple threads may access shared PRNs and/or PRNGs.

The PRNG-Broker acts as a middle layer between the application code, e.g., a Monte Carlo simulation, and specialised PRNG libraries. It efficiently manages parallel PRN requests to external PRNG libraries, adequately using the computational resources available in multicore, manycore, and GPU devices. This efficient management of PRN generation focus on improving the performance of parallel compute-bound applications, but also provides a significant benefit for both sequential and memory-bound codes.

The key features of the current PRNG-Broker are summarised as follows:

- select and use a PRNG and associated number distribution per consumer thread, to fill in advance a thread-private dual-buffer with a batch of PRNs, for each consumer thread;
- a simple and intuitive API in C++ to hide from the programmer the complexities of the parallel broker [2];
- easy replacement of requests to a PRNG library in the user code by the PRNG-Broker: simply change the instantiation of the PRNG library class;
- supported generators: statistically tested sequential and parallel implementations of a LCG subset and Mersenne Twister, available in the ROOT, MKL, PCG, STL, and cuRAND libraries; more PRNGs can be easily added;
- supported distributions of generated PRNs: uniform and Gaussian;
- PRNG parallelisation strategies: parameterization and block splitting;

[2] An in-depth description of the PRNG-Broker API and its installation process is available at https://github.com/prng-broker/prng-broker/wiki/PRNG-Broker.

– hide the overhead of the PRN generation and the memory transfer on PRNG accelerators connected through PCI-Express and Ethernet: a thread-private dual-buffer PRN storage to reduce the applications waiting time for PRNs;
– adequate use of the available parallel computing resources in multicore and manycore servers (as well as using additional servers for PRN generation), and manycore and GPU accelerator devices.

3.1 Core of the Broker

The request for PRNs in a multithreaded application can be performed using one of the following approaches:

– to call the PRNG whenever a single PRN is needed; or
– to generate a batch of PRNs and store the result in a shared buffer; when a PRN is needed, the consumer thread removes it from the buffer; when the buffer is empty, a new batch is requested;
– similar to the previous one, but storing the result in a thread private buffer;
– 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 consumer threads degrades performance, due to thread contention when accessing shared resources. The PRNG with leapfrogging is not supported by the current PRNGs in the PRNG-Broker. Sharing a single sequential PRNG among multiple concurrent threads may affect the statistical quality of the generators; this approach should also be avoided, although it is still a common practice in some simulations.

The key benefit of the dual-buffer PRNG approach is to remove the need to synchronise a contention-free access to the buffer between the broker and a consumer. The PRNG-Broker always stores the generated PRNs in thread private dual-buffers, whose size is user configurable. It offers two alternative approaches to generate batches of PRNs, which can be selected by the user:

– a shared PRNG, where PRNs are placed in the different sets of dual-buffers using block splitting;
– a PRNG per thread, where each PRNG instance is initialised with different internal-state parameters (not only a different seed; see Figure 1).

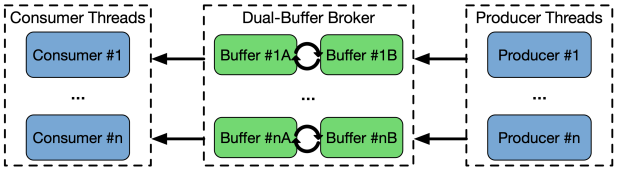

Fig. 1. Dual-buffer broker with multiple PRNG instances using parameterization.

Figure 1 illustrates the dual-buffer approach with multiple PRNG instances, each initialised with a different set of parameters. Each PRNG instance will be

executed by a concurrent thread, as preliminary results showed that using a single PRNG degraded performance, and is responsible for the dual-buffer of a single consumer thread, which allows simultaneous loads of different consumer thread's buffers. A management thread is responsible for each dual-buffer, swapping buffers once one is consumed and signalling the producer thread to generate a new batch of PRNs for the depleted buffer. The threads on the host application may never wait for PRNs, as the generation latency is hidden by the dual-buffer.

3.2 Hardware Aware PRN Generation

The PRNG-Broker supports *native* and *offload* execution of most PRNGs that it interfaces with, as illustrated in Figure 2. *Native* execution processes the PRNGs on the computing device running the host application, such as a multicore or manycore device, sharing computational resources with other computational tasks. *Offload* execution processes PRNGs on an alternative computing device, such as a GPU or a manycore accelerator on the same server, or even a separate compute server, freeing computing resources on the host device that can be used to process the user application.

The PRNG-Broker supports PRNG offload to accelerators over PCI-Express, using NVidia GPUs or the Intel KNC, or to additional compute servers over a network interconnection. However, using these devices connected introduces the overhead of transferring data from these devices to the host memory space. Offloading PRNGs to other devices/servers may be less efficient, depending on the interconnection, but the dual-buffer approach has the potential to hide the latency of memory transfers.

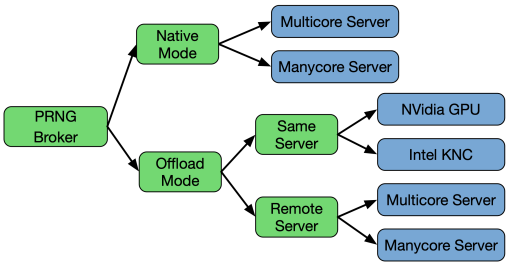

Fig. 2. Hardware configurations supported by the PRNG-Broker library.

The management thread allocates the buffers in memory, each with a default capacity for 50K PRNs (configurable by the user), initialises the PRNGs on the devices, and handles data transfers. The overhead of the management threads is minimal as they are only used to swap the buffers and are asleep the remaining of the application runtime. In *native* mode a thread or a set of threads execute a single PRNG (using block splitting) or multiple PRNG instances (using parameterization) to fill the empty buffers of multiple management threads.

The allocation of the PRN buffers may have a significant impact on performance on NUMA multi-socket servers: when a computing thread has to access

a buffer allocated in a memory bank of the companion multicore device, it will suffer a larger latency penalty than accessing a memory bank of the multicore device where the thread is being executed. To mitigate this penalty management threads are assigned and bound to a core on the same device as the associated computing/consumer thread, ensuring that the buffers are allocated in the memory bank closer to the computing threads.

When offloading PRNGs to accelerator devices, each management thread in the host uses a private stream to launch the PRNGs and to perform the memory transfers, ensuring simultaneous execution. This approach is only used for PRNGs that support parameterization initialisation, to guarantee the statistical quality of the generated PRNs.

3.3 Libraries and the PRNG-Broker API

The PRNG-Broker interfaces with a set of libraries to use efficient and statistically tested implementations of PRNGs and distribution transformations for various computing devices. These libraries are linked to PRNG-Broker during installation to use multicore, manycore, and GPU devices. Currently, PRNG-Broker supports a set of PRNG implementations, based on the Mersenne Twister algorithm, from ROOT, Intel MKL, GNU STL, PCG and NVidia cuRAND.

The MKL library is used to generate PRNs for Intel multicore servers, Xeon Phi Knights Landing manycore server (native or offloaded over Ethernet) and the Knights Corner accelerator (offload over PCI-Express), as it provides optimised implementations for these devices. cuRAND is used to generate PRNs for NVidia GPUs (offloaded over PCI-Express), as it provides optimised implementations for most generations of GPUs. GNU STL, ROOT, and PCG are used to generate PRNs for multicore and manycore servers (both native or offloaded over Ethernet), but are not the most computationally efficient due to their lack of vectorized and multithreaded code. They are included for compatibility reasons.

The PRNG-Broker API replaces calls to traditional PRNGs to an access to the PRN buffers, hiding the buffers implementation and PRNG parallelization, as shown below.

```
PseudoRandomGenerator rnd;

void some_function () {
    val1 = rnd.gauss(this_thread_id);
    val2 = rnd.uniform(this_thread_id);
}

int main (void) {
    // ... PRNG-Broker initialisation ...
    rnd.init(nthreads); // Amount of computing threads in the user code
    rnd.setGenerator(CURAND); // Any supported PRNG (optional)
    rnd.setSeed(0); // Sets the seed / library pre-computed parameters
    rnd.gaussianConfig(0.0, 0.02); // Avg and stddev if Gaussian is required
    // ... PRNG-Broker cleanup ...
    rnd.shutdown();
}
```

The `gauss` and `uniform` methods return a PRN following either a Gaussian or an uniform distribution, according to the parameters passed to `gaussianConfig`.

The `init` method initialises the internal state, such as the number of buffers, the parameters to use by the PRNGs and the broker itself. The different API options are described at https://github.com/prng-broker/prng-broker/wiki/PRNG-Broker.

4 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. These applications usually deal with large amounts of experimental data, which is read from one or more files in variable sized chunks or datasets, and placed into adequate data structures. Parallel implementations of these analyses, where concurrent threads process independent dataset elements, are often used in scientific data analyses. The use of PRNGs often plays a significant role in performance degradation of these applications.

High energy physics scientists at CERN developed a 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 a 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 3.

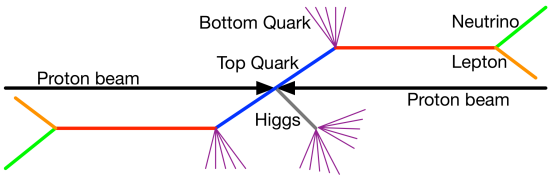

Fig. 3. Schematic representation of the $t\bar{t}$ system and Higgs boson decay.

$t\bar{t}H$ analytically computes the characteristics of the neutrinos with the measured data of other particles, to reconstruct at the end of the data processing both top quarks and the Higgs boson.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. Three variants of the $t\bar{t}H$ analysis were considered as representative case studies:

- the `ttH_as` (*accurate sensors*) assumes that the data measured by the detector is 100% accurate; requires only 30 PRNs per event;
- the `ttH_sci` (*sensors with a confidence interval*) performs an extensive sampling within the 99% confidence interval in the kinematic reconstruction; this version works with 1024 samples, requiring a total of 30Ki PRNs per event;
- the `ttH_scinp` (`sci` *with a new pipeline*) performs different operations, maintains the same inter-stage dependencies, and uses 10Ki PRNs per event.

The PRNG used by default by these data analyses is the Mersenne Twister implementation provided by ROOT using the Box-Muller transformation.

5 Results and Discussion

Once the PRNG-Broker was validated, several aspects were considered for a performance evaluation, such as a characterisation of the testbeds and associated case studies, and a set of different experimental measurements:

- the throughput of different PRNG algorithms and implementations;
- the execution times of the 3 variants of the case study managed by the PRNG-Broker, varying algorithms and configurations;
- a comparative performance evaluation of the fastest configuration on different server architectures, with and without accelerators.

5.1 Testbeds and associated cases studies

Three testbeds were used for the quantitative evaluation of PRNG-Broker:

- a dual socket server with 12-core Intel Xeon E5-2695v2 Ivy Bridge devices (addressed as IB) at 2.4 GHz, with one NVidia Tesla K20 with 2496 CUDA cores and one Intel Xeon Phi 7120p Knights Corner (addressed as KNC);
- a dual socket server with 16-core Intel Xeon E5-2683v4 Broadwell devices (addressed as BW) at 2.1 GHz;
- a manycore server with 64-core Intel Xeon Phi 7210 Knights Landing device (addressed as KNL) at 1.30 GHz, using the quadrant core clustering and the on-package embedded RAM configured as cache.

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 1 computing thread per core, as preliminary tests showed that using core multithreading provided no performance improvements. The testbeds used in the multi-server tests are interconnected by a 10 Gibit Ethernet network. The compiler used was Intel Compiler version 18.

The `ttH_sci` application, which is the most PRNG intensive, spent around 90% of the execution time calling the ROOT PRNG, while the application that required slightly less PRNs (1/3), `ttH_scinp`, spent around 50% of the execution time. `ttH_as` spent less than 10% of its execution time on PRN generation, but it is used to evaluate the overhead of the PRNG-Broker.

5.2 Performance of the case studies with the PRNG-Broker

Figure 4 shows the speedup of the three 24-threaded versions of the $t\bar{t}H$ analyses on the dual 12-core server, with the selected PRNG algorithms and different approaches for PRNG execution, *vs* the original code.The batch MKL generator

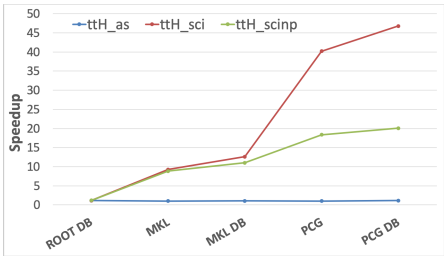

Fig. 4. Speedup of the parallel $t\bar{t}H$ analyses with different PRNG-Broker approaches *vs* the original ROOT single number PRNG on the 2*12-core IB server.

with the Inverse Transform Sampling method was used as the PRNG representative of the MKL library, due to its performance in preleminary tests.

The PRNGs with the *DB* indication use the dual-buffer approach with the PRNG-Broker, while without *DB* indicates that the PRNG is used in single-PRN generation without PRNG-Broker. The efficient use of PRNGs in PRNG-Broker leads to a better performance at several levels:

- a significant amount in the high demanding PRNs `ttH_sci`, when compared to the original code, up to 48x;
- the `ttH_scinp` also benefited from PRNG-Broker, with a speedup improvement up to 20x using PCG;
- the performance of `ttH_as` was not degraded by the use of PRNG-Broker;
- the use of a dual-buffer approach outperformed a single-buffer by 15%, 23%, and 18% for the ROOT, MKL, and PCG PRNGs, respectively, and may improve further with parallel code.

However, these improvements are still limited by the hardware resources, since the analyses computing threads are competing for the same computational resources as the PRNG-Broker. Vectorization also had a significant impact on the performance of PRNGs: the generation of a large amount of PRNs adequately explore this feature, which is not present in the single PRN generation.

5.3 Comparative performance on different servers

Offloading PRNG to Kepler or KNC accelerators frees computational resources on the multicore devices to be used by other computations of the $t\bar{t}H$ analyses. Additional KNL and IB servers were also used to offload the PRNG, to ensure that the performance improvements obtained using the accelerators were not only due to the increase in computational resources allocated to the $t\bar{t}H$ analyses. Figure 5 shows the execution times of the parallel case studies on different server configurations, using the dual-buffer PRNG-Broker: (i) on single servers with no accelerators, (ii) on the IB server with an accelerator dedicated to PRNGs and (iii) on the IB server accessing an external server dedicated to PRNGs.

The performance of the `ttH_sci` benefited from offloading the PRN generation from the host server to an additional server or accelerator devices, with

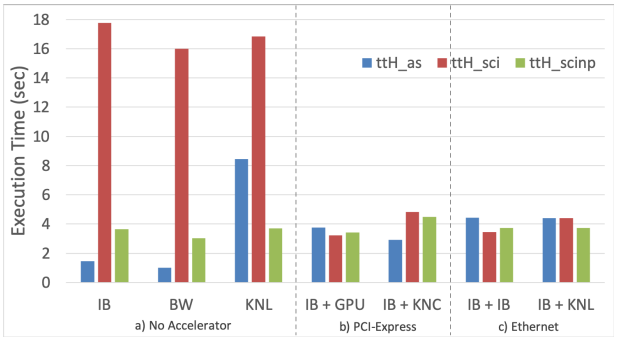

Fig. 5. Execution times of the PRNG-Broker on different server configurations.

speedups of 5x using the IB server with the Kepler GPU over the BW server.This performance gap would be greater if the Kepler GPU was coupled to a BW server, instead of the IB server, which has 8 less computing cores and AVX, instead of AVX2. A similar behaviour was observed when offloading the PRN generation to a KNC accelerator, or an additinal IB and KNL servers. The performance of `ttH_as` and `ttH_scinp` did not improve as much, since the PRN generation accounts for a small portion of their execution time.

The performance of the `ttH_as` application was decreased by 2.9x and 4.4x when using the KNC accelerator or an external KNL server, respectively. Since the code lacks heavy computations and does not need large sets of PRNs, the PRNG-Broker can not hide the memory transfer costs over PCI-Express and Ethernet. The GPU and the external IB server over Ethernet spent $> 90\%$ of the PRNG time to transfer the PRNs to the host memory, while the KNC accelerator and the external KNL server spent around 80%.

The overall performance of the `ttH_sci` and `ttH_scinp` applications *vs* the original code that requests successive single PRNs from the inefficient ROOT PRNG, using the Kepler GPU accelerator, was improved: up to 70x and 12x, respectively. An increase of 11x and 12x were due to a faster PRNG and the remainder due to the PRNG-Broker. A similar behaviour is observed for the `ttH_sci` and `ttH_scinp` applications with either the KNC accelerator or an external KNL server: speedups of 47x and 51x, respectively. Using for these applications an external IB server to run the PRNGs, the performance improvements were up to 65x and 11x.

These tests proved that applications that require a very large amount of PRNs can greatly benefit by efficiently using PRNGs, regardless of the server architecture and configuration in which they will execute. The overall higher speedups, compared to using only the host server, are also due to the higher availability of the computing cores to perform application specific computations, since they were freed from generating PRNs. The use of a dual-buffer *vs* a single PRN/buffer contributed to minimise the impact of the PRNs transfer from the accelerators on the same server to the host memory or from external servers: these costs were hidden from the application, since the memory transfers occurred while a buffer still had PRNs in the host memory space.

The PCG PRNG was the best performing when using only multicore devices to process both the application code and PRNG. However, the PCG suite uses a more computationally efficient algorithm than the Mersenne Twister, which may not be a fair comparison. It is the responsibility of the end user to assess if this PRNG should be used over other traditional PRNGs available in PRNG-Broker, which are well accepted and extensively tested by the mathematics's community.

6 Conclusions and Future Work

This paper presented the PRNG-Broker library and associated API that ensures efficient generation and management of large sets of PRNs and their statistical distribution, in a transparent way to the user. It supports PRNG native execution on multicore and manycore devices, or offloaded to manycore and GPU accelerator devices or external servers. A detailed analysis of the PRNG-Broker gave an insight into the best way to use PRNGs with real software codes, evaluating the performance of their implementation and the different approaches to use them. PRNG-Broker interfaces with existing libraries to use efficient implementations of popular PRNGs, and focus on an adequate management of the generated PRNs to provide significant performance improvements for applications that require large amounts of PRNs.

Three variations of a real scientific data analysis from CERN high-energy physicists were used as case studies: `ttH_sci` and `ttH_scinp`, both compute-bound codes, and `ttH_as`, an I/O bound code. `ttH_scinp` and `ttH_sci` require 10Ki and 30Ki PRNs per event (dataset element), while `ttH_as` only requires 30, with the tested dataset containing around 800K events. In both single and dual buffer approaches, the PRNG is managed by an additional thread per computing thread, so that data processing and the PRNG can be simultaneously executed.

The dual buffer approach provided speedups over single PRN and single buffer generation for every PRNG tested in the `ttH_sci` and `ttH_scinp` applications. The most significant improvements were obtained by using external multicore/manycore servers (over Ethernet) and computing accelerators (over PCI-Express) to generate the PRNs, as these approaches free the host multicore device that would be occupied on the PRNGs to process the applications. These approaches provided speedups up to 35x and 71x on single and dual-socket servers, using the Kepler GPU, as well as up to 51x improvement using an external manycore server for the `ttH_sci` application. The use of better vector operations may have a significant improvement on PRNG performance, as proven by the improvement up to 65% of the `ttH_as` performance on a server with AVX-2 *vs* a server with AVX instruction sets.

Four main conclusions can be extracted from this analysis:

- the choice of an efficient implementation of a given PRNG is imperative for the application performance: both ROOT and MKL implement the Mersenne Twister but MKL is, at least, 10x faster;

- 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;
- offloading PRN generation to computing accelerators and other servers using the dual buffer approach provides a significant performance improvement, as PRNs can be concurrently generated with the application execution.

When using the PRNG-Broker natively on multicore/manycore devices the execution of the PRNG algorithm shares computational resources with the application, i.e., threads executing the PRNG have to compete with application threads for multicore time. Alternatively, computing cores could be reserved for the PRNG-Broker, but with two caveats: (i) the user would have to ensure that the threads from the application would not use the cores assigned for the PRNG-Broker and (ii) the amount of reserved cores would be dependent on the PRN requirements of each application, which is usually not known without profiling. The former caveat goes against the purpose of PRNG-Broker, which is to hide the complexities of generating and managing PRNs from the user, and would unnecessarily increase the learning curve of the broker. Assuming that this would not be a problem, the latter would require the user to configure the application threads to not use some computing cores, which could vary according to the application and even during the execution, due to dynamic adjustments of the PRNG-Broker. The feasibility of this approach is being evaluated, but preliminary tests already showed no significant improvement for the case studies.

This work focused mainly on the popular Mersenne Twister algorithm, but others could be integrated into PRNG-Broker to extend its range of target applications, 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 multicore devices is provided by MKL, as proved by the initial benchmark results.

References

1. Asmussen, S., Glynn, P.W.: Stochastic Simulation: Algorithms and Analysis, vol. 57. Springer Science & Business Media (2007)
2. ATLAS Collaboration: Observation of a New Particle in the Search for the Standard Model Higgs Boson with the ATLAS Detector at the LHC. Physics Letters B **716**(1), 1–29 (2012)
3. Bradley, T., du Toit, J., Tong, R., Giles, M., Woodhams, P.: Parallelization Techniques for Random Number Generators. In: GPU Computing Gems Emerald Edition, pp. 231–246. Elsevier (2011)
4. Claessen, K., Pałka, M.H.: Splittable Pseudorandom Number Generators Using Cryptographic Hashing. In: ACM SIGPLAN Notices. vol. 48, pp. 47–58. ACM (2013)
5. Devroye, L.: Sample-based Non-uniform Random Variate Generation. In: Proceedings of the 18th Conference on Winter Simulation. pp. 260–265. ACM (1986)
6. Golder, E., Settle, J.: The Box-Muller Method for Generating Pseudo-Random Normal Deviates. Applied Statistics **25**, 12–20 (1976)

7. Hill, D.R., Mazel, C., Passerat-Palmbach, J., Traore, M.K.: Distribution of Random Streams for Simulation Practitioners. Concurrency and Computation: Practice and Experience **25**(10), 1427–1442 (2013)
8. Hörmann, W., Leydold, J., Derflinger, G.: Automatic Nonuniform Random Variate Generation. Springer Science & Business Media (2013)
9. Karl, A.T., Eubank, R., Milovanovic, J., Reiser, M., Young, D.: Using RngStreams for Parallel Random Number Generation in C++ and R. Computational Statistics **29**(5), 1301–1320 (2014)
10. Law, A.M., Kelton, W.D., Kelton, W.D.: Simulation Modeling and Analysis, vol. 3. McGraw-Hill New York (2000)
11. L'Ecuyer, P., Simard, R.: TestU01: AC Library for Empirical Testing of Random Number Generators. ACM Transactions on Mathematical Software **33**(4), 22:1–22:40 (2007)
12. L'Ecuyer, P., Simard, R.: RngStreams: An Object-Oriented Random-Number Package in C with Many Long Streams and Substreams. Operations Research **50**(6), 1073–1075 (2012)
13. L'Ecuyer, P., Munger, D., Kemerchou, N.: clRNG: a Random Number API with Multiple Streams for OpenCL. Tech. rep., University of Montreal (2015)
14. Marsaglia, G.: The Marsaglia Random Number CDROM Including the Diehard Battery of Tests of Randomness. Computer file, Florida State University, 1995 (1995)
15. Mascagni, M., Srinivasan, A.: Algorithm 806: SPRNG: A Scalable Library for Pseudorandom Number Generation. ACM Transactions in Mathematical Software **26**(3), 436–461 (2000)
16. Matsumoto, M., Nishimura, T.: Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. ACM Transactions on Modeling and Computer Simulation (TOMACS) **8**(1), 3–30 (1998)
17. Nvidia: CURAND library. NVDIA Corporation (2010)
18. O'Neill, M.E.: PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation. Tech. Rep. HMC-CS-2014-0905, Harvey Mudd College, Claremont, CA (2014)
19. Rademakers, F.: ROOT — A C++ Framework for Petabyte Data Storage, Statistical Analysis and Visualization. Computer Physics Communications **180**(12), 2499–2512 (2009)
20. Saito, M., Matsumoto, M.: SIMD-Oriented Fast Mersenne Twister: A 128-bit Pseudorandom Number Generator. In: Monte Carlo and Quasi-Monte Carlo Methods 2006, pp. 607–622. Springer (2008)
21. Saito, M., Matsumoto, M.: 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. pp. 92–100 (2012)
22. Saito, M., Matsumoto, M.: Variants of Mersenne Twister Suitable for Graphic Processors. ACM Transactions Mathematical Software **39**(2), 12:1–12:20 (2013)
23. for Standardization, I.O., the International Electrotechnical Commission: ISO/IEC 9899:2018. Tech. rep., International Organization for Standardization and the International Electrotechnical Commission, Geneva, Switzerland (2018)
24. Von Neumann, J.: Various Techniques Used in Connection With Random Digits. National Bureau of Standards Applied Mathematics Series **12**, 36–38 (1951)
25. Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., Wang, Y.: Intel Math Kernel Library. Springer (2014)