ThreadLocalMRG32k3a: A Statistically Sound Substitute to Pseudorandom Number Generation in Parallel Java Applications

Jonathan Passerat-Palmbach, Claude Mazel, David R.C. Hill

UMR 6158 CNRS

ThreadLocalMRG32k3a: A Statistically Sound Substitute to Pseudorandom Number Generation in Parallel Java Applications

Jonathan Passerat-Palmbach [1] [*] [†] [‡] [§],
Claude Mazel [*] [†] [‡] [§],
David R.C. Hill [*] [†] [‡] [§]

Originally published in: Proceedings of the IEEE High Performance Computing and Simulation conference 2012 — July 2012 — pp 543-550
http://dx.doi.org/10.1109/HPCSim.2012.6266971
(nominated for the outstanding paper award)

[1]This author's PhD is partially funded by the Auvergne Regional Council and the FEDER

[*] ISIMA, Institut Supérieur d'Informatique, de Modélisation et de ses Appplications, BP 10125, F-63173 AUBIERE
[†] Clermont Université, Université Blaise Pascal, LIMOS, BP 10448, F-63000 CLERMONT-FERRAND
[‡] Clermont Université, Université Blaise Pascal, BP 10448, F-63000 CLERMONT-FERRAND
[§] CNRS, UMR 6158, LIMOS, F-63173 AUBIERE

RESEARCH CENTRE
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France

Abstract: *Parallel And Distributed Simulations (PADS) become more and more spread since scientists always want more accurate results in the shortest time. PADS are often sensitive to several parameters, and when they own a stochastic component, one has to ensure he knows how to correctly deal with randomness in a parallel application. This point can appear to be very tricky for non-experts, who might be tempted to move this part of the simulation aside. Several software efforts have been produced in various programming languages to help developers handle pseudorandom streams partitioning in PADS. However, these tools remain third-party libraries that need to be integrated in already existing applications, and that might be hardly swappable. The latest release of the Java Development Kit (JDK 7) tries to tackle this problem with a new class called ThreadLocalRandom. The latter is in charge of safe pseudorandom number generation across Java threads. The present work studies the pros and cons of this approach, and introduces ThreadLocalMRG32k3a, an alternative to ThreadLocalRandom that shows better results in terms of generation speed and statistical quality. ThreadLocalMRG32k3a respects the same Application Programming Interface (API) as ThreadLocalRandom, thus enabling clients to use it in place of its JDK counterpart at no cost.*

Keywords: Java; Threads; Pseudorandom Number Generation; Parallelization of Simulation; Software Development Tools and Support; Object Oriented Programming & Design

1 Introduction

At the manycore era, simulation practitioners need to take advantage of these new architectures. However, designing Parallel And Distributed Simulations (PADS) is not straightforward, and is often seen as a potential long-term perspective rather than a real implementation consideration. Indeed, simulation developers are not offered many tools to harness manycore architectures.

The new release of the Java Development Kit 7 (JDK 7) comes to grips with this lack by focusing on Java's concurrency features. Java 7 offers a couple of new tools to enhance the already existing concurrent package. Mainly, a new framework called Fork/Join appears [Lea, 2000]. It provides an easy-to-use MapReduce implementation running in parallel thanks to a pool of worker threads. Such inputs, added to the already present tools allowing the distribution of the computing load across several threads, should attract more and more simulation practitioners to Java development. These users will also bring their own concerns bound to parallelization in their domain of expertise. Thus, simulationists working on stochastic simulations will ask for a tool to help them to partition a random source in a parallel Java environment.

As a matter of fact, correct partitioning of random streams is the main concern of several studies [Hellekalek, 1998a, Hill, 2010], and neglecting this part of a simulation could lead to biased results [Reuillon et al., 2011]. To avoid such issues, one needs to ensure of four main points that are exposed in [Coddington, 1996]:

1. Each thread should dispose of its own random sequence;

2. The parallelization technique must be usable for any number of threads;

3. The parallel random streams produced should be uncorrelated;

4. When the status of the PRNG is not modified, the sequence of random numbers generated for a given thread must be the same no matter the number of threads and no matter of threads scheduling.

Java 7 introduces the ThreadLocalRandom class, a tool that intends to enable developers to deal with pseudorandom numbers in parallel on a single shared memory computer, without having to figure out how to distribute numbers among the available processing elements. The question we have for scientific applications is the following: can ThreadLocalRandom serve as a random source with the statistical quality required by stochastic simulations? Although this development is a good initiative that is worth being integrated in Java, we will see that the current implementation still has some major drawbacks for scientific purposes.

The present work will:

- Study ThreadLocalRandom's intrinsics to figure out whether its output is satisfying regarding stochastic simulations needs;

- Present already existing libraries that could serve as alternatives to ThreadLocalRandom;

- Introduce ThreadLocalMRG32k3a, our proposal based upon the MRG32k3a Pseudorandom Number Generator (PRNG) algorithm from Pierre L'Ecuyer [L'Ecuyer, 1999];

- Compare ThreadLocalMRG32k3a to ThreadLocalRandom, and consider its potential evolutions.

2 ThreadLocalRandom

2.1 Implementation concerns

Officially released with JDK 7, the ThreadLocalRandom facility was developed within the *jsr166y* initiative by Doug Lea. ThreadLocalRandom tries to solve the complexity to use random sources correctly in parallel applications. Each thread owns a ThreadLocalRandom instance, allowing each to be independent from the others to pick up random numbers. Still, the most important point behind this technique is that it is supposed to distribute pseudorandom streams safely among threads.

ThreadLocalRandom inherits from *java.util.Random*, thus sharing its interface. Every thread must call a method named *current* before calling the classical *nextXXX* methods to pick up a random number which type is indicated by the *XXX* suffix.

ThreadLocalRandom makes use of the Random Spacing technique [Hill, 2010] to distribute pseudorandom streams across threads. This technique consists in initializing an identical PRNG instance in each thread with a different seed-status [Passerat-Palmbach et al., 2011], the latter being randomly chosen by another algorithm. By doing so, each thread owns an independent pseudorandom sequence, provided the PRNG algorithm has a long enough period, and is not subject to long-range correlations [De Matteis and Pagnutti, 1988].

Random Spacing is implemented in ThreadLocalRandom through its constructor. Indeed, this method calls the Random constructor before setting a Boolean to true, thus depicting that initialization has been done and cannot be performed again. ThreadLocalRandom must then rely on the constructor of the Random class to set its initial seed. Until JDK6, the Random constructor used to automatically perform a call to *setSeed*, the method in charge of the Random Spacing initialization of the seed. However, this is not true anymore with JDK7, where the constructor of Random does not summon *setSeed* anymore. Consequently, any PRNG class that extends Random and relies on it to call *setSeed*, will see its seed-status remain uninitialized. According to [Gosling et al., 2005], the seed of each thread is thus set to zero as any class member of the long type would be. Hence, every thread will pick up the same pseudorandom sequence in such a case!

We have already spotted a similar problem in a Java Mersenne Twister implementation and proposed a corrective patch that solves it. Calling *setSeed* in every thread could easily solve this problem. Unfortunately, the *setSeed* public method, which would normally allow setting a thread's PRNG seed to a new value is locked by the previously mentioned boolean. Such a feature is important to prevent any user to harm pseudorandom streams independence between threads by setting several seeds to the same value. However, this also prevents us to adapt the class behaviour, and force a call to *setSeed* directly in the subclass's constructor. Moreover, this solution relies on user-awareness of the problem, which goes against the initial purpose of ThreadLocalRandom to hide random streams distribution to the user.

The problem was finally solved in the second update of the JDK7 by changing the Random constructor in order to take into account a potential use of *setSeed* by subclasses. This change is quite confusing in two ways. Not only does it break encapsulation, one of the elementary concepts of the object-oriented paradigm, but it also appears as a lack of good software engineering that discourages to adapt an implementation according to an already existing source code. Instead, it is safer to rely on the specification only. In our case, the Random class documentation issued by Oracle makes no mention of a potential call to the setSeed method by the Random constructor. As a consequence, we cannot blame Oracle for this bug, but rather advise developers to focus on the official documentation only, especially when they are working on such sensitive aspects of the implementation.

2.2 Statistical quality discussion

Nowadays, several renowned tools exist to check the statistical quality of pseudorandom streams. When Knuth's tests [Knuth, 1969] cannot be considered as a proof of quality on their own, they have been integrated in wider test batteries. A testing suite, named DieHard, highly regarded for many years, was proposed by Marsaglia [Marsaglia, 1996], and was improved by Brown [Brown et al., 2010] who proposed the DieHarder testing suite. The SPRNG [Mascagni et al., 1999] parallel random number library is also providing a thorough set of statistical tests. For five years now, the scientific community has widely agreed that the current reference test battery is TestU01 from [L'Ecuyer and Simard, 2007]. TestU01 currently offers the most complete collection of utilities for the empirical statistical testing of uniform random number generators. Please note that this enumeration does not take into account testing suites targeting cryptographic applications, which leading tool is rather the NIST STS proposal [Rukhin et al., 2001].

In addition to the classical statistical tests for PRNGs, and the other tests previously cited and proposed in the literature, TestU01 proposes new original tests as well as predefined tests suites (SmallCrush, Crush and BigCrush with more than a hundred tests). Many of the most spread PRNGs fail quite significantly when faced to this software. ThreadLocalRandom's underlying PRNG, which is a well-known and widely studied LCG from Knuth [Knuth, 1969], also ruling the output of the POSIX *drand48* C function for example, is among the algorithms at fault. LCG generators should be banned from scientific applications since their structure is not adapted to many modern applications, and the problem is even bigger when parallel and distributed computing is considered. In addition, the period proposed by ThreadLocalRandom is relatively small for modern scientific applications: it is 2^{48} numbers long, when Pierre L'Ecuyer suggests that for modern applications periods should be at least 2^{100} numbers long [L'Ecuyer, 2010].

Now thinking of a parallel utilization of ThreadLocalRandom, we can barely imagine that such a bad generator [Ferrenberg et al., 1992, Hellekalek, 1998b, Hellekalek, 1998a] when considered in sequential environments could behave better in a parallel environment. Thanks to TestU01 parallel filters [L'Ecuyer and Simard, 2007], we can easily create a random sequence formed by the combination of any number of input sequences from different ThreadLocalRandom initializations. However, as stated in [Salmon et al., 2011], it is impossible to perform a complete coverage of all possible logical sequences because many strategies can be set up to distribute both tasks and random streams across parallel computational units. Nonetheless, some samples are particularly representative of how most users will use random sequences, and we will study them in section 5.3.

3 Related Works

Several attempts to provide a user-friendly interface to generate random numbers in parallel environments can be found in the literature. Here we recall the major proposals that can compete and replace ThreadLocalRandom in scientific applications. We only consider frameworks providing ways to automatically distribute pseudorandom streams through threads without the user's help.

As we have seen previously, the standard Java library only ensures thread safety through synchronized methods when accessing to the random number generation features of the *java.util.Random* class. This approach is not satisfying at all in the world of High Performance Computing: in addition to not ensuring reproducibility of simulations because of thread scheduling, it impacts performance of parallel stochastic applications because of the sequential bottleneck implied by the synchronization guarding random facilities. This method to partition pseudorandom sequences is known as Central Server in the literature [Hill, 2010].

JAPARA [Coddington and Newell, 2004] has been proposed by Coddington and Newell in 2004 to tackle this lack in Java libraries. They bring up a Java API to support parallel generation of random streams. On the other hand, JAPARA proposes that every processing element (be it Java threads in our case) handles its own pseudorandom stream. By doing so, only the initialization phase is synchronized, and a referenced partitioning technique is then used to distribute the underlying pseudorandom streams. As a matter of fact, JAPARA comes with three PRNGs implemented, each coupled to a distribution technique that matches its intrinsic characteristics. The point is that the user only has to select the PRNG he wants to employ, and then rely on the framework to ensure independence between the different streams assigned to the threads. Furthermore, JAPARA allows the user to save and restore the current state of a PRNG, thus permitting to checkpoint a simulation.

L'Ecuyer's team first minded on an object oriented pseudorandom generation package in 2002 with [L'Ecuyer et al., 2002]. This has been concretized in the rStream library [L'Ecuyer and Leydold, 2005] that implements a single MRG32k3a PRNG, whose independent streams are partitioned from an original stream thanks to the Sequence Splitting technique [Hill, 2010]. A declination of rStream comes with the SSJ (Stochastic Simulation in Java) [L'Ecuyer et al., 2002] framework as the pseudorandom streams parallelization utility of the library. It provides a greater set of PRNGs, with the famous Mersenne Twister [Matsumoto and Nishimura, 1998] for instance, and also a compliant set of distribution techniques.

The latest Java random number generation framework that has retained our attention is DistRNG [Reuillon, 2008, Reuillon et al., 2011]. While its API does not diverge from the two other proposals described in this section, DistRNG focuses on correct partition of random streams. To do so, this framework handles XML generic statuses that model any PRNG state. Every computational element is initialized with a different XML status that needs to be built upstream. DistRNG displays a fine choice of statistically sound PRNGs according to the TestU01 reference testing library.

As a conclusion, this section has shown that several satisfying proposals of APIs for parallel pseudorandom number generation can be found in the literature. Consequently, users have many reliable solutions at their disposal if they want to take advantage of statistically sound pseudorandom sequences in their Java applications. Moreover, most of these solutions can replace ThreadLocalRandom's features, but require modifications on the application source code to meet their functioning requirements.

4 MRG32k3a Implementation

In this section, we present the Java implementation we made of MRG32k3a PRNG, described by Pierre L'Ecuyer in [L'Ecuyer, 1999]. Several features of this algorithm retained our attention, from its internal data structure to the results it displays when faced to today's most stringent testing batteries.

4.1 The choice of MRG32k3a

Talking about its internal properties, MRG32k3a is really suited to parallelization among small computational elements such as threads, because its lightweight data structure only stores 6 integers to handle its state. It means that introducing this PRNG in already existing Java applications will have roughly no impact on their memory footprint. The algorithm itself is quite short, relying on simple operations only to issue new random numbers. The parameters chosen for MRG32k3a are such that it has a full period of 2^{191} numbers. This period is fairly enough since L'Ecuyer suggests periods between 2^{100} and 2^{200} are highly sufficient even for

large-scale simulations. MRG32k3a has been designed to produce independent streams and sub-streams from its original random sequence thanks to its parameters that enable safe Sequence Splitting. Thus, the internal parameters split the initial sequence into 2^{64} adjacent streams of 2^{127} random numbers, themselves divided into sub-streams containing 2^{76} elements.

The ability to issue independent streams is very important when tackling the safe distribution of random numbers across parallel computational elements. MRG32k3a's Sequence Splitting approach suggests an obvious partition of the original sequence by assigning each computational element a stream or a sub-stream, depending on the application eagerness for random numbers. As long as we are focusing on parallel applications that are Java threads based, the parallel grain is limited to how many threads a single manycore machine can handle. This figure depends on the underlying architecture hosting the Java platform, but we do not expect having to deal with more than 2^{64} parallel threads, which is the total number of independent streams bearing 2^{127} random numbers each that MRG32k3a can provide.

Still, the most important point in our opinion is that this generator displays a great statistical quality, according to its TestU01 results related in [L'Ecuyer and Simard, 2007]. MRG32k3a passes all the tests of BigCrush, the most stringent and complete testing battery coming with TestU01, and is so referred to as a "Crush-resistant" PRNG in [Salmon et al., 2011]. While being Crush-resistant cannot ensure a perfect randomness of the considered pseudorandom stream, it is a satisfying property that few PRNGs can be proud of. Furthermore, PRNGs stated as bad according to TestU01 criteria have led to incorrect simulation results in the past [De Matteis and Pagnutti, 1988, Ferrenberg et al., 1992, Maigne et al., 2004], and even good PRNGs can miss some tests [Reuillon, 2008, Salmon et al., 2011]. Thus, as we did not want to take any risks with our PRNG choice as a replacement of ThreadLocalRandom's LCG, we focused on Crush-resistant PRNGs such as MRG32k3a.

4.2 Implementation Details

We have designed ThreadLocalMRG32k3a so that it can be used as an alternative to ThreadLocalRandom. Thus it displays the very same interface as its counterpart. The methods contained in our class are all dedicated to produce various kinds of random outputs: from integers to double precision floating point values, so as ThreadLocalRandom performs. In the same way, we also reused the *current()* method introduced in ThreadLocalRandom: it actually aims to provide its independent instance of ThreadLocalMRG32k3a to each thread calling it. The *current()* method is the core of ThreadLocalMRG32k3a in a sense that the call hierarchy it implies highly differs from the original behaviour of ThreadLocalRandom.

Current() is a static method that every thread must call in order to retrieve its own ThreadLocalMRG32k3a instance. The method basically acts like a singleton that builds a ThreadLocal instance parameterized with the PRNG class, ThreadLocalMRG32k3a in our case. ThreadLocal is a generic Java class appeared in JDK 2 that provides easy copy-on-access facilities to concurrent threads. When a thread first accesses a ThreadLocal object, the latter gets an instance especially built for it that does not require synchronized accesses with other threads. Typical applications of this mechanism are thread-based counters such as thread identifiers for example.

Our implementation first takes advantage of this technique to ensure reproducibility between executions. Indeed, stochastic simulations need to be reproduced for debug purposes or their results to be checked. When ThreadLocalRandom is used by default, it does not satisfy this need because it relies on the Random constructor to set its internal seed, which behavior is to use the current system time as seed. We fix this problem by basing the seed initialization on the thread unique identifier, so that for a given identifier, a thread will always be assigned the same stochastic stream. Although Java enables us to figure out a thread identifier at runtime

through the *Thread.currentThread().getId()* call, we cannot rely on this identifier since it is global for the whole JVM and thus depends on how many threads were created before ours are. Therefore, we have stored a handcrafted unique identifier within ThreadLocalMRG32k3a, thanks to a synchronized atomic counter handled through the ThreadLocal mechanism.

Please note that the ThreadLocal mechanism only operates at thread level and is not aware of any task concept introduced by top-level frameworks such as Fork/Join. Thus, reproducibility cannot be expected when either ThreadLocalRandom or ThreadLocalMRG32k3a is used by tasks from these frameworks.

Now that we are able to assign a stream to each thread, we need to determine how these streams are actually handled within our MRG32k3a implementation. We have seen previously that this PRNG had been designed to partition its original sequence into streams and sub-streams. We have chosen to give an independent stream to each thread, so that they can all benefit of their own independent 2^{127} numbers long pseudorandom sequence. As long as streams are contiguous in the original sequence, the beginning state of each independent stream is located every 2^{127} elements in the original sequence. Hopefully, a Jump Ahead algorithm is detailed in [L'Ecuyer, 1999] that enables us to advance the state of the original sequence at almost no extra cost, no matter how much elements we skip. Thus, if a thread has been assigned an identifier k, the seed-status of its ThreadLocalMRG32k3a instance is initialized by the constructor to X_n with $n = 2^{127} * k$. The latter situation is summed up in Figure 1:

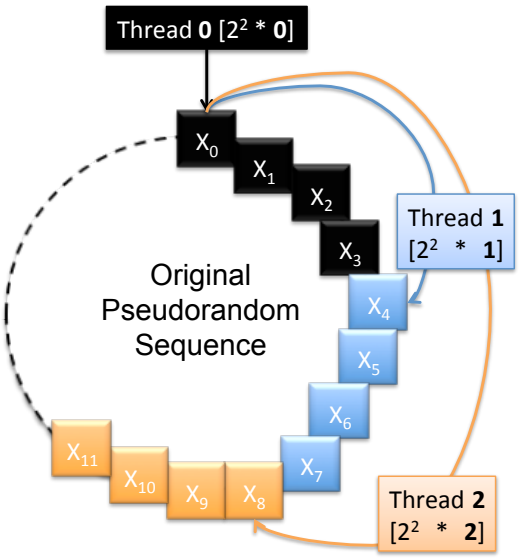

Figure 1: 3 Threads performing respective Jump Ahead on an original pseudorandom sequence, according to their unique identifier. Streams are here limited to 2^2 elements each

4.3 Example of use

From a Java developer point of view, picking up random numbers from ThreadLocalMRG32k3a is as simple as using the original Java Random API as exposed in Listing 1:

Listing 1: Example of use of ThreadLocalMRG32k3a

```
ThreadLocalMRG32k3a myRNG = ThreadLocalMRG32k3a.current();
```

```
Thread tid = new Runnable {
    public void run() {
        for (int i = 0; i < numbersCount_; ++i) {
            myRNG.next();
        }
    }
}
```

The implementation detailed in this section makes ThreadLocalMRG32k3a the equivalent of ThreadLocalRandom concerning API and features. However, our proposal is more suited to parallelize scientific applications where statistically sound random sources are necessary. That being said, let us see how ThreadLocalMRG32k3a outcomes its counterpart with detailed analysis of both PRNGs' performances.

5 Results

In this part, we compare three aspects of the original ThreadLocalRandom and of our proposal ThreadLocalMRG32k3a: their memory footprint, their numbers throughput and their statistical quality.

5.1 Memory footprint

Here we want to evaluate the memory footprint added to each thread by both PRNGs. ThreadLocalRandom wraps a LCG that uses only one integer to store its internal state, whereas MRG32k3a needs at least 6 integers. Our ThreadLocalMRG32k3a also relies on an extra thread identifier to provide reproducibility as required by stochastic simulations. As a conclusion, ThreadLocalRandom is taking the lead on the memory footprint criterion, consuming only one integer when our proposal implies 6 more.

5.2 Speed

It is quite hard to isolate accurately the methods involved in random number generation across several threads when trying to evaluate the speed of two PRNGs. That is why we based our comparison on the Netbeans' profiler data to figure out which algorithm was the most efficient. Figure 2 presents the average profiling results of the two PRNGs obtained by probing an 8-thread application, which each thread issues the same amount of random numbers:

Figure 2 shows that ThreadLocalMRG32k3a is about 10 times faster than ThreadLocalRandom. Therefore, our Java wrapper has no impact on MRG32k3a's performance since its genuine version was announced 10 times faster than the LCG used by ThreadLocalRandom [L'Ecuyer, 1999].

5.3 Statistical quality

We have already discussed LCGs statistical quality, but in our case, the LCG at the heart of ThreadLocalRandom is used in parallel thanks to the Random Spacing distribution technique. When parallelizing an application, data processing is spread among the available computational elements following a particular pattern, so as random numbers. As a result, knowing the parallelization techniques used for both random numbers and input data, we could recreate the

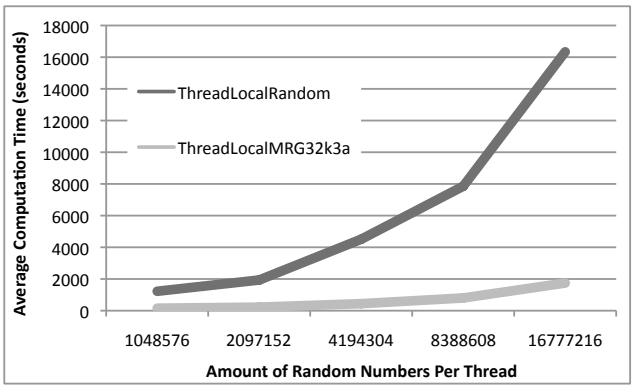

Figure 2: Computation time of the two PRNGs according to the profiler's output

computation scenario that would have taken place in a sequential environment. This allows us to check the corresponding random sequence resulting from the concatenation of the subsequences. We know that it is nearly impossible to examine every possible combination, thus we decided to focus on the most obvious technique to process input data: assign an equally sized subset from the original data to each thread.

To simulate this situation, we have faced the two PRNGs to TestU01. The random stream studied by the testing battery was provided by combining the sub-streams of a given number of threads. In the following chart, each PRNG is tested using combined streams resulting from what would be the concatenated random sequence of 16 to 64 threads:

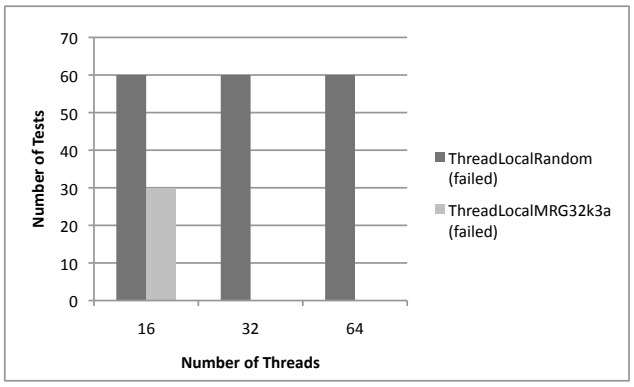

Figure 3: BigCrush results for ThreadLocalRandom and ThreadLocalMRG32k3a used by 16, 32 and 64 threads. Each test configuration was initialized with 60 different seed-statuses

Figure 3 shows that using MRG32k3a instead of the LCG implemented in ThreadLocalRandom is particularly relevant when considering the statistical output of both generators. Here, we see that none of the 180 configurations of ThreadLocalRandom tested can pass the TestU01 Bigcrush testing battery, when ThreadLocalMRG32k3a only misses 30. This figure backs our PRNG choice for the underlying algorithm ruling our ThreadLocalRandom alternative.

6 Discussion

In this paper, we propose a Java implementation of L'Ecuyer's MRG32k3a that can be used in the very same conditions as ThreadLocalRandom. However, simulation practitioners often expect to challenge their stochastic models with different random sources. In this way, providing a full framework offering implementations of other PRNGs, wrapped in an API identical to ThreadLocalRandom, could help simulationists who champion Java threads based developments for their parallel simulations to harness parallel architectures. In this section, we review the algorithms that we plan to include in future versions of this work.

Having already considered a Sequence Splitting partitioning technique with MRG32k3a, we chose to focus our further investigations on what seems the most reliable parallelization technique: parameterization [Hill, 2010]. While Sequence Splitting intends to slice an original random sequence in several independent random streams, parameterization tackles the problem differently. Indeed, PRNGs employing parameterization own a parameter that can distinguish one instance of a given PRNG from one another. This unique parameter then contributes to issue highly independent random streams that can be assigned to different processing elements, such as threads.

6.1 TinyMT

TinyMT is the latest offspring from the Mersenne Twister family. TinyMT is not described in any scientific article yet, but information about it can be found on its dedicated webpage [Saito, 2011]. This PRNG matches the requirements we have formulated for a PRNG to be integrated in our ThreadLocalRandom alternative: it is stated as producing a good quality output, according to TestU01 statistical tests, and displays a long-enough period of 2^{127} numbers. As explained in the introduction of this section, this PRNG champions parameterization to provide highly independent streams. It is consequently shipped with a software tool that can create over than $2^{32} \times 2^{16}$ independent statuses.

We are now considering the implementation of this PRNG as another alternative to ThreadLocalRandom. Part of this development work will be close to what has already been achieved with the implementation of MRG32k3a. However, because of parameterization, we might be faced to technical difficulties to provide a user-friendly implementation. Actually, TinyMT parameterized statuses need to be precomputed by a third-party application called Dynamic Creator, which is delivered with the PRNG. Thus, to provide a full Java concurrent implementation, not only we need to implement the algorithm, but also to ship a large amount of precomputed statuses with it, provided that Dynamic Creator relies on several C++ libraries, and would thus be difficult to reimplement in Java in a portable way. Each thread will then receive an instance of ThreadLocalTinyMT initialized by a different status. Since the data structure representing a status weights no more than a hundred of bytes, delivering lots of ready to be used parameterized statuses should be possible.

6.2 Threefry/Philox

Threefry and Philox are counter-based PRNGs [Salmon et al., 2011] also relying on parameterization to solve random streams partition concerns. Like any other PRNGs considered in this study, they are Crush-resistant and display good performance in regards to their low memory footprint and high numbers throughput. They appear to be better suited than TinyMT (or any other member of the Mersenne Twister family) to target a smooth integration in the Java threads API since their parameters are formed by a single key that can be set at runtime according to

each thread's unique identifier. Indeed, Mersenne Twister-like PRNGs might not fit some applications that cannot afford wasting any memory space to store the state and the initialization parameters of each thread's PRNG.

7 Conclusion

This work has studied the recent ThreadLocalRandom proposal shipped with JDK 7 that intends to provide independent random streams for parallel Java applications. Having stressed the importance of using statistically sound PRNGs and partitioning techniques, we have asserted that Crush-resistant generators were in our opinion the only category of generators that should be trusted for scientific applications development. Considering this criterion, we have evaluated ThreadLocalRandom, as having a satisfying design but a poor implementation. On the other hand, this study surveys the most spread libraries aiming the same goal as ThreadLocalRandom, but displaying improved quality. We strongly recommend some of them, like SSJ or DistRNG, to replace ThreadLocalRandom as much as possible.

In addition, we propose in this work ThreadLocalMRG32k3a as another alternative to ThreadLocalRandom. Our proposal respects the same API as ThreadLocalRandom, but it relies on MRG32k3a, a well-known Crush-resistant PRNG. Not only does ThreadLocalMRG32k3a display a far better statistical quality than its JDK counterpart, it is also much more suited for stochastic simulations, given that it issues a reproducible output by default. ThreadLocalMRG32k3a is a bit greedier in memory but it completely outperforms its counterpart in both speed and statistical quality. Indeed, ThreadLocalMRG32k3a is more than 10 times faster than ThreadLocalRandom and passes TestU01's most stringent testing battery: BigCrush.

We now plan to implement other Crush-resistant PRNG algorithms such as TinyMT, Threefry or Philox that display statistical properties equivalent to MRG32k3a. This effort would allow simulation practitioners to compare the results of their simulations when fed with different random sources that would integrate smoothly in their developments, in the same way as ThreadLocalRandom. We want to enable them to switch the PRNG they use with the slightest impact on the source code of their simulations.

Acknowledgements

The authors would like to thank Florian Guillochon for his careful reading and useful suggestions on the draft. This work received financial support from the Auvergne Regional Council.

References

[Brown et al., 2010] Brown, R., Eddelbuettel, D., and Bauer, D. (2010). Dieharder: A random number test suite.

[Coddington, 1996] Coddington, P. (1996). Random number generators for parallel computers. Technical Report 2, NHSE.

[Coddington and Newell, 2004] Coddington, P. and Newell, A. (2004). Japara-a java parallel random number generator library for high-performance computing. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. IEEE.

[De Matteis and Pagnutti, 1988] De Matteis, A. and Pagnutti, S. (1988). Parallelization of random number gemerators and long-range correlations. *Numerische Mathematik*, 53:595–608.

[Ferrenberg et al., 1992] Ferrenberg, A., Landau, D., and Wong, Y. (1992). Monte carlo simulations: Hidden errors from "good" random number generators. *Physical Review Letters*, 69(23):3382.

[Gosling et al., 2005] Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java language specification*. Prentice Hall, 3 edition.

[Hellekalek, 1998a] Hellekalek, P. (1998a). Don't trust parallel monte carlo! In *Proceedings of Parallel and Distributed Simulation PADS98*, pages 82–89. IEEE.

[Hellekalek, 1998b] Hellekalek, P. (1998b). Good random number generators are (not so) easy to find. *Mathematics and Computers in Simulation*, 46(5-6):485–505.

[Hill, 2010] Hill, D. (2010). Practical distribution of random streams for stochastic high performance computing. In *IEEE International Conference on High Performance Computing & Simulation (HPCS 2010)*, pages 1–8. invited paper.

[Knuth, 1969] Knuth, D. (1969). *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley.

[Lea, 2000] Lea, D. (2000). A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM.

[L'Ecuyer and Leydold, 2005] L'Ecuyer, P. and Leydold, J. (2005). rstream: Streams of random numbers for stochastic simulation. Technical report, Department of Statistics and Mathematics, Abt. f. Angewandte Statistik u. Datenverarbeitung, WU Vienna University of Economics and Business.

[L'Ecuyer et al., 2002] L'Ecuyer, P., Meliani, L., and Vaucher, J. (2002). Ssj: a framework for stochastic simulation in java. In *Proceedings of the 34th conference on Winter simulation: exploring new frontiers*, pages 234–242. Winter Simulation Conference.

[L'Ecuyer, 1999] L'Ecuyer, P. (1999). Good parameters and implementations for combined multiple recursive generators. *Operations Research*, 47(1):159–164.

[L'Ecuyer, 2010] L'Ecuyer, P. (2010). *Encyclopedia of Quantitative Finance*, chapter Pseudorandom Number Gemerators.

[L'Ecuyer and Simard, 2007] L'Ecuyer, P. and Simard, R. (2007). Testu01: A c library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):22:1–40.

[L'Ecuyer et al., 2002] L'Ecuyer, P., Simard, R., Chen, E., and Kelton, W. (2002). An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075.

[Maigne et al., 2004] Maigne, L., Hill, D., Calvat, P., Breton, V., Reuillon, R., Legre, Y., and Donnarieix, D. (2004). Parallelization of monte carlo simulations and submission to a grid environment. *Parallel processing letters*, 14(2):177–196.

[Marsaglia, 1996] Marsaglia, G. (1996). The marsaglia random number cdrom, with the diehard battery of tests of randomness. Produced under a grant from the National Science at Florida State University.

[Mascagni et al., 1999] Mascagni, M., Ceperley, D., and Srinivasan, A. (1999). Sprng: A scalable library for pseudorandom number generation. In *Proceedings of the Ninth SIAM conference on parallel processing for scientific computing*.

[Matsumoto and Nishimura, 1998] Matsumoto, M. and Nishimura, T. (1998). Mersenne twister: A 623-dimensionnally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulations: Special Issue on Uniform Random Number Generation*, 8(1):3–30.

[Passerat-Palmbach et al., 2011] Passerat-Palmbach, J., Mazel, C., Bachelet, B., and Hill, D. (2011). Shoverand: a model-driven framework to easily generate random numbers on gpgpu. In *IEEE International Conference on High Performance Computing & Simulation*, pages 41–48.

[Reuillon, 2008] Reuillon, R. (2008). *Simulations stochastiques en environnements distribués - Application aux grilles de calcul*. PhD thesis, Université Blaise Pascal - École Doctorale Sciences pour l'Ingénieur.

[Reuillon et al., 2011] Reuillon, R., Traore, M. K., Passerat-Palmbach, J., and Hill, D. R. (2011). Parallel stochastic simulations with rigorous distribution of pseudo-random numbers with distme: Application to life science simulations. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a.

[Rukhin et al., 2001] Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., and Vo, S. (2001). A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical report, NIST.

[Saito, 2011] Saito, M. (2011). Tinymt.

[Salmon et al., 2011] Salmon, J. K., Moraes, M. A., Dror, R. O., and Shaw, D. E. (2011). Parallel random numbers: as easy as 1, 2, 3. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 16:1–16:12, New York, NY, USA. ACM.

UMR 6158 CNRS

RESEARCH CENTRE
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France

Publisher
LIMOS - UMR CNRS 6158
Campus des Cézeaux
Bâtiment ISIMA
BP 10125 - 63173 Aubière Cedex
France
http://limos.isima.fr/