HAL
archives-ouvertes.fr

TaskLocalRandom: A Statistically Sound Substitute to Pseudorandom Number Generation in Parallel Java Tasks Frameworks

Jonathan Passerat-Palmbach, Claude Mazel, David Hill

▶ **To cite this version:**

Jonathan Passerat-Palmbach, Claude Mazel, David Hill. TaskLocalRandom: A Statistically Sound Substitute to Pseudorandom Number Generation in Parallel Java Tasks Frameworks. Concurrency and Computation: Practice and Experience, Wiley, 2014, pp.N/A. <10.1002/cpe.3214>. <hal-01098598>

HAL Id: hal-01098598
https://hal.inria.fr/hal-01098598

Submitted on 27 Dec 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution| 4.0 International License

UMR 6158 CNRS

TaskLocalRandom: A Statistically Sound Substitute to Pseudorandom Number Generation in Parallel Java Tasks Frameworks

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

Originally published in: Concurrency and Computation: Practice and Experience — February 2014 — pp n/a–n/a
doi:/10.1002/cpe.3214
©2014 John Wiley & Sons, Ltd.

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 AUBIÈRE
† Clermont Université, Université Blaise Pascal, LIMOS, BP 10448, F-63000 CLERMONT-FERRAND
‡ CNRS, UMR 6158, LIMOS, F-63173 AUBIÈRE

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

Abstract: Several software efforts have been produced over the past few years in various programming languages to help developers handle pseudorandom streams partitioning. Parallel and Distributed Stochastic Simulations (PDSS) can obviously benefit from this kind of high-level tools. The latest release of the Java Development Kit (JDK 7) tries to tackle this problem by providing facilities to partition a pseudorandom stream across various threads thanks to the new class *ThreadLocalRandom*. Meanwhile, Java 7 offers a framework to split a problem in a divide and conquer way through the new class called *ForkJoinPool*. As any other Java Thread Pool, ForkJoin exploits threads as workers and manipulates the tasks that will be run on the workers. In *ThreadLocalRandom*, pseudorandom number generation is handled at a thread level. As a consequence, a scientific application taking advantage of a Java Thread Pool to parallelize its computation may suffer from a bad pseudorandom stream partitioning due to the behaviour of *ThreadLocalRandom*. The present work introduces TaskLocalRandom, a task-level alternative to *ThreadLocalRandom* that solves this partitioning problem and assigns an independent pseudorandom stream to each task run in the thread pool. TaskLocalRandom is compatible with existing Java thread pools such as Executors or ForkJoin.

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

1 Introduction

At the manycore era, simulation practitioners must take advantage of such powerful architectures. The new release of the Java Development Kit 7 (JDK 7) addresses this need by focusing on the concurrency features of Java. Java 7 offers a couple of new tools to enhance the already existing *concurrent* package. Namely, a new framework called Fork/Join appears [Lea, 2000]. It provides an easy-to-use implementation of the divide and conquer paradigm through lightweight tasks. The divide and conquer paradigm suggests to split an important workload among several Processing Elements (PEs). First, each PE will compute a subset of the whole workload. Then the results will be gathered when all the subtasks have returned their results.

This new task framework, 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 partition a random source in a parallel Java environment.

In fact, correct partitioning of random streams is the main concern of several studies [Hellekalek, 1998a, Hill et al., 2013], and neglecting this part of a simulation could lead to biased results [Reuillon et al., 2011]. To avoid such issues, one needs to ensure that the four main guidelines exposed in [Coddington, 1996, Passerat-Palmbach et al., 2012b] are followed:

1. Each computing element should dispose of its own random sequence;
2. The parallelization technique must be usable for any number of computing elements;
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 computing element must be the same no matter the number of computing elements and regardless of the way computing elements are scheduled.

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 for scientific applications is as follows: can *ThreadLocalRandom* serve as a random source with the statistical quality required by scientific applications?

Although this development is a good initiative that is worth being integrated in Java, we will see that the current implementation has still some major drawbacks for scientific purposes. In fact, the underlying PRNG used by *ThreadLocalRandom* can hardly be considered for any scientific application as explained in Section 2.2 and in [Hellekalek, 1998a, Ferrenberg et al., 1992,

Hellekalek, 1998b]. Moreover, as its name suggests, *ThreadLocalRandom* is designed to perform at a thread level. However, threads are now mostly used as worker threads in Java thread pools, following the introduction of tasks frameworks since JDK 5. Worker threads were designed to get rid of the overhead bound to threads creation. An application creating lots of threads will indeed be slowed down by frequent thread spawns. To overcome this issue, threads are created once and for all, and are then assigned tasks to achieve. Such permanent threads are gathered in thread pools over the lifetime of the application.

Due to architecture considerations, we usually use as many worker threads as there are available Processing Elements in the system. Processing Elements can take the shape of physical or logical cores depending on the underlying architecture that exploits the Java Virtual Machine. For instance, modern Intel CPUs integrate a feature called HyperThreading that enables a physical core to refine its parallelism capabilities by running two different threads in parallel, provided they perform operations involving different hardware resources (*e.g.*: one thread can compute a floating point instruction, while the other one treats an operation on integers).

The JVM considers these two execution paths as two logical cores. In such a case, the number of Processing Elements will denote the number of logical cores. In order not to limit the parallelism granularity to this logical cores boundary, the notion of tasks has been introduced. Tasks are purely equivalent to Threads in terms of development, since they implement the same Java Runnable interface. They only differ from threads in that they are queued within worker threads, and thus are scheduled when their number is greater than the number of workers. As a consequence, a single task will run in a worker thread at a given point, but the worker thread can preempt it in case it is stalled, waiting for data for instance.

These hardware considerations are not taken into account by the JVM, but task frameworks integrate a task stealing mechanism that enables an idle worker thread to steal tasks from the queue of a busy worker. As a result, Java task frameworks will benefit from the scheduling mechanism that spreads the workload among the available hardware resources. This behaviour is depicted in Figure 1. Tasks scheduling allows designing a finely grained parallel algorithm that will scale up smoothly on platforms with the number of worker threads.

Tasks make pseudorandom stream distribution from *ThreadLocalRandom* inefficient, since tasks are not taken into account by the class. As a consequence, a novelty brought by the latest release of the JDK 7 cannot handle one of the most common way to leverage threads in Java concurrent applications! Although *ThreadLocalRandom* is stated as "particularly appropriate when multiple tasks (for example, each a *ForkJoinTask*) use random numbers in parallel in thread pools"[1], it cannot be considered as safe in terms of pseudorandom stream distribution since all the tasks run by the same worker thread will share the same pseudorandom stream.

[1] `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadLocalRandom.html` (last access 11/1/13)

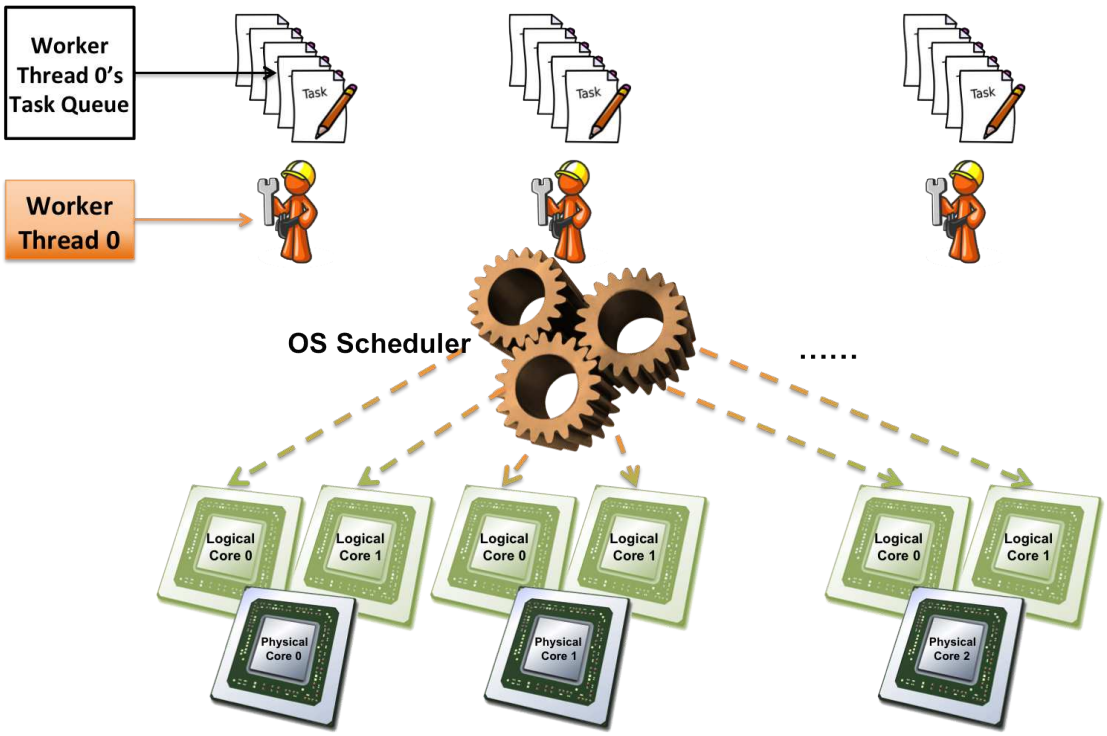

Figure 1: One worker thread per Processing Element is created. It is assigned a queue of tasks to process.

The present work will consequently tackle a correct way to distribute pseudorandom streams in parallel Java applications harnessing the power of tasks frameworks. To do so, we will:

- Study *ThreadLocalRandom*'s intrinsics to figure out whether its output is satisfying regarding stochastic simulations needs;
- Discuss *ThreadLocalRandom*'s capabilities when used in a Task framework context
- Present already existing libraries that could serve as alternatives to *ThreadLocalRandom*;
- Introduce TaskLocalRandom, our proposal based upon the MRG32k3a Pseudorandom Number Generator (PRNG) algorithm from Pierre L'Ecuyer [L'Ecuyer, 1999];
- Compare TaskLocalRandom 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 regarding use of random sources correctly in parallel applications. Each thread owns a *ThreadLocalRandom* instance, allowing each thread 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 et al., 2013] 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., 2010], the latter being randomly chosen by another algorithm. By doing so, each thread owns a pseudorandom sequence considered as highly independent (no mathematical proof allows asserting that two sequences are truly independent from a probabilistic point of view), 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[2]. Calling *setSeed* in every thread could easily solve this problem. Unfortunately, the *setSeed* public method, which would normally allow setting the seed of the PRNG of a thread 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,

[2] `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/VERSIONS/JAVA/PATCH/MTRandom.patch` (last access 11/1/13)

this also prevents us from adapting the class behaviour, and forcing a call to *setSeed* directly in the constructor of the subclass. 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 constructor of *Random* in order to take into account a potential use of *setSeed* by subclasses. This change is 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. It is indeed not recommended 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.

Another weakness in the implementation of *ThreadLocalRandom* lies in the impossibility to reproduce the same pseudorandom sequences throughout several runs of the application by default. Scientific applications such as stochastic simulations need to ensure reproducibility between executions for their results to be checked or for debug purposes. 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 behaviour is to use the current system time as seed. This could be interesting for games but not for a scientific software. This problem can be fixed by basing the initial seed on a unique identifier for each thread, so that for a given identifier, a thread will always be assigned the same stochastic stream. Still, although Java provides a thread identifier at runtime through the *Thread.currentThread().getId()* call, this identifier is not reliable since it is global for the whole JVM and thus depends on how many threads were created before the one considered. Therefore, *ThreadLocalRandom* must be extended with its own thread identifier to make it safe in terms of pseudorandom stream distribution and reproducibility. We have proposed such an extension in [Passerat-Palmbach et al., 2012a].

2.2 Statistical Quality Discussion

Nowadays, several renowned tools exist to check the statistical quality of pseudorandom streams. The sole Knuth's tests [Knuth, 1969] cannot characterize the statistical quality of a pseudorandom sequence on their own. They have been integrated in wider test batteries that give a more thorough judgment on the statistical quality of the pseudorandom sequence. A testing suite, named DieHard, highly regarded for many years, was proposed by Marsaglia [Marsaglia, 1996], and was improved by Brown [Brown et al., 2009] who proposed the DieHarder testing suite. The SPRNG [Mascagni and Srinivasan, 2000] parallel random number library is also providing a thorough set of statistical tests. For six 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 focusing on cryptographic applications. In this category the leading tool is instead the NIST STS proposal [Rukhin et al., 2001], although TestU01 also owns tests targeting cryptographic generators.

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 significantly when faced with this software. The underlying PRNG of *ThreadLocalRandom*, is a well-known and widely studied LCG from Knuth [Knuth, 1969] (it also rules the output of the POSIX *drand48* C function for example), is among the algorithms at fault. LCG generators should be discarded from scientific applications since their structure is not adapted to many modern applications [L'Ecuyer, 2010]. 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].

In regards to the parallel utilization of *ThreadLocalRandom*, we can barely imagine that such a bad generator [Hellekalek, 1998a, Ferrenberg et al., 1992, Hellekalek, 1998b] 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. Consequently, testing campaigns often focus on samples that are particularly representative of the distribution technique used.

2.3 ThreadLocalRandom Plunged into Tasks Frameworks

Java tasks frameworks are now widely spread across Java applications exploiting concurrency. Introduced in JDK 5 through the *ExecutorService* class, these tools are thread pools that create threads for the whole lifetime of an application. These threads are then used as workers that will pick up tasks from queues created by the task framework and execute their content, instead of creating new threads. By doing so, the application no longer suffers from the overhead induced by frequent thread creations. The power of this approach is that it relieves developers from low-level thread management without impacting the application or requesting new knowledge.

The tasks queued to be executed by worker threads are nothing more than instances implementing the Runnable interface. This latter interface is already used to implement handcrafted concurrent Java applications: they contain the

workload to be performed by threads when they are used without any task framework. This simplicity explains the wide adoption of tasks frameworks amongst the Java community.

However, *ThreadLocalRandom*'s internal mechanisms make it unable to handle tasks. Most of the features provided by *ThreadLocalRandom* to distribute pseudorandom streams amongst threads lie behind the *current()* method. The latter is a static method that every thread must call in order to retrieve its own *ThreadLocalRandom* instance. The method basically acts like a singleton that builds a *ThreadLocal* instance parameterized with the PRNG class, *ThreadLocalRandom* in our case. *ThreadLocal* is a generic Java class that appeared in JDK 2, and provides easy copy-on-access facilities to concurrent threads. When a thread first accesses a *ThreadLocal* object, it gets its own copy of the object that does not require synchronized accesses with other threads anymore. Typical applications of this mechanism are thread-based counters such as thread identifiers.

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 *ThreadLocalRandom* is used by tasks from these frameworks.

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 that provide 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 the random number generation features of the *java.util.Random* class. This approach is not satisfying in the world of High Performance Computing (HPC): in addition to not ensuring reproducibility of simulations because of thread scheduling and of scaling problems, 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 et al., 2013].

JAPARA [Coddington and Newell, 2004] was 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. JAPARA proposes that every Processing Element (Java threads in that case) handles its own pseudorandom stream. In doing so, only the initialization phase is synchronized, and a referenced partitioning technique is then used to distribute the underlying pseudorandom streams. JAPARA comes with three PRNGs implemented, each coupled with a distribution technique that matches its intrinsic characteristics. 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.

After having first proposed a random number package with splitting facilities [L'Ecuyer and Côté, 1991], L'Ecuyer's team proposed an object-oriented pseudorandom generation package in 2002 with [L'Ecuyer et al., 2002b]. This was achieved in [L'Ecuyer et al., 2002b] that proposes a C++ implementation of the MRG32k3a PRNG, which independent streams are partitioned from an original stream thanks to the Sequence Splitting technique [Hill et al., 2013]. A Java declination comes with the SSJ (Stochastic Simulation in Java) [L'Ecuyer et al., 2002a] framework and its *RandomStream* interface, the pseudorandom streams parallelization utility of the library. It provides a greater set of PRNGs (including the famous Mersenne Twister [Matsumoto and Nishimura, 1998] for instance), and a compliant set of distribution techniques.

The latest Java random number generation framework that has retained our attention is DistRNG [Reuillon et al., 2011, Reuillon, 2008]. 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.

Now considering other languages, SPRNG [Mascagni and Srinivasan, 2000] is one of the most widely used libraries in C++ to automatically distribute pseudorandom streams across MPI [MPI Forum, 1993] processes. While this framework achieves the same result than *ThreadLocalRandom* for MPI processes, it takes advantage of the MPI rank (MPI term for process identifier) assigned to processes by MPI, whereas we cannot rely on such a mechanism in Java.

Random123 [Salmon et al., 2011] looks like an interesting development, especially for memory constrained environments such as GPUs. This set of PRNGs have shown a good statistical quality when faced with empirical testing battery such as TestU01 [L'Ecuyer, 1999]. Still, beyond the fact that the period length of these PRNG is long, we cannot assess uniformity theoretically via a spectral test, but only empirically by typical tests. Consequently, we are still waiting for more thorough studies regarding these algorithms from domain experts before using them in our own developments.

In 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* features but require modifications on the application source code to meet their functioning requirements. Still, none of these frameworks integrates the task notion, thus calling for further developments if they are to be used within a task execution framework.

4 TaskLocalRandom Implementation

In this section, we present the Java solutions enabling pseudorandom stream distribution across Java tasks. Apart from our software engineering inputs, the PRNG algorithm we use is a wrapper of the *RNGStream* class from Pierre L'Ecuyer [L'Ecuyer, 2001]. It implements the MRG32k3a PRNG algorithm described in [L'Ecuyer, 1999]. Recall that our software engineering inputs aim at providing each task of a Java application a different pseudorandom sequence. Several features of MRG32k3a retained our attention, from its internal data structure to the results it displays when faced with 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 and tasks, because its lightweight data structure only stores 6 integers to handle its state. The algorithm itself is relatively 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 approximately 2^{191} numbers [L'Ecuyer, 1999]. This period is long enough in regards to what Pierre L'Ecuyer suggests: periods between 2^{100} and 2^{200} are highly sufficient [L'Ecuyer, 2010]. Even with our modern large-scale simulations with computing power going to Exascale, we will not reach 2^{200}. 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 streams as independent as possible is very important when tackling the safe distribution of random numbers across parallel computational elements. The Sequence Splitting approach of MRG32k3a 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 tasks based, the parallel grain is limited to how many tasks a single many-core machine can handle. This figure depends on the underlying architecture hosting the Java platform, but we really do not expect to deal with more than 2^{64} parallel tasks, the total number of independent streams bearing 2^{127} random numbers each that MRG32k3a can provide.

In addition, the most important point 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 that comes 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 of which few PRNGs can be proud; even the cur-

rent Mersenne Twister family of generators fails some (very limited) tests. PRNGs stated as bad according to TestU01 criteria have led to incorrect simulation results in the past [Ferrenberg et al., 1992, De Matteis and Pagnutti, 1988, Maigne et al., 2004], and even good PRNGs can miss some tests [Salmon et al., 2011, Reuillon, 2008]. As we did not want to take any risks with our PRNG choice as a replacement to the LCG of ThreadLocalRandom, we focused on Crush-resistant PRNGs such as MRG32k3a.

4.2 Assigning Independent Pseudorandom Streams to Different Tasks

Provided that we are able to uniquely identify tasks (this aspect will be tackled in Section 4.3), an independent pseudorandom sequence can be assigned to each of them. This section determines how these sequences 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 substreams. We have chosen to give an independent stream to each task, 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, [L'Ecuyer, 1999] details a Jump Ahead algorithm that enables us to advance the state of the original sequence at almost no extra cost, no matter how many elements we skip. Thus, if a task has been assigned an identifier k, the seed-status of its TaskLocalRandom instance is initialized by the constructor to X_n with $n = 2^{127} * k$. The latter situation is summed up in Figure 2.

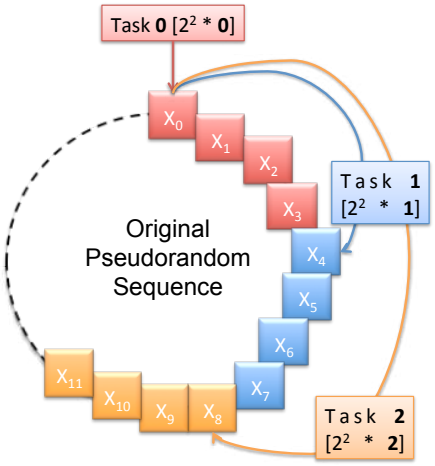

Figure 2: 3 Tasks performing respective Jump Ahead on an original pseudorandom sequence, according to their unique identifier. In this pedagogic example streams are limited to 2^2 elements each.

4.3 The Challenge of Uniquely Identifying Tasks

The main struggle at the heart of TaskLocalRandom is to provide a unique task identifier, which the Java language does not support at the time of writing. As explained previously, *ThreadLocalRandom* benefits of the *ThreadLocal* mechanism from the Java SDK. *ThreadLocal* relies on JNI (Java Native Interface) calls, which means its implementation is directly tied to the underlying Operating System (OS) that supports the JVM. Each OS deals with threads on its own way, using native APIs. However, these native APIs do not provide a common concept of threads. It seems consequently difficult to implement a mechanism equivalent to *ThreadLocal* at a task level.

Two approaches can be considered to avoid this lack: either each task can autonomously distinguish itself from the others using a particular algorithm, or a central element in the system needs to uniquely identify each task.

In the first case, the most spread algorithm used to provide unique identifiers without the help of a central element is called UUID [Leach et al., 2005]. It was designed for the purpose of online Internet services and is now frequently exploited in programming techniques to distinguish elements. For instance, Java uses it to allot a unique version number to classes that support serialization. Several algorithms are referenced by the RFC (Request for Comments) standard to produce UUIDs. UUIDs issued by the *UUID* class from the Java SDK are from class 4. It means that the underlying algorithm used here is powered by a PseudoRandom Numbers Generator (PRNG). The actual PRNG algorithm is not explicitly mentioned, but it is stated as cryptographically secure by the documentation. More pragmatically, a 128-bit identifier issued by *UUID* would have 50% of chance to overlap with another one if 1 billion of UUIDs had been picked up every second for 100 years. Consequently, this approach is reliable enough when it comes to generate unique random identifiers.

Still, UUIDs would directly represent the identifier of the task in our case. Let us recall that the latter identifier is also at the heart of the Jump Ahead algorithm of the MRG32k3a PRNG, which allows it to assign independent random streams to each task. Unfortunately, this Jump Ahead algorithm only accepts 32-bit integers in input to determine the amount of streams to jump over. In order to preserve the uniqueness characteristics of UUIDs, we cannot imagine to shrink them from 128-bit to 32-bit without introducing a risk of collision between two UUIDs. As a result, UUIDs are not a satisfying approach to uniquely identify tasks in our case.

The other option to achieve unique task identification is to request the identifiers atomically to a central element. This way to get identifiers has the drawback to create a bottleneck at the task creation, when each new task will claim its own identifier. Although this assertion is technically true, it is important to consider its impact in a more pragmatical way. To do so, let us figure out the typical number of tasks that might be created at a at some point in an application.

Tasks are typically created prior to any execution launched in workers. Still, we can imagine that tasks are spawned in parallel in order to fasten this ini-

tialization stage. Then, the maximum number of tasks created at a given time cannot exceed the number of worker threads leveraged by the application. We know that the number of workers created is bound to the number of logical cores available in the machine. Any greater number of worker threads would quickly make the performance of the application drop. Thus, the number of worker threads, and consequently the maximum number of tasks potentially created at a given time will remain in the range of the number of logical cores. At the time of writing, this number can grow up to hundreds of logical cores in the cutting-edge HPC hosts. In Java, the number of logical cores available is obtained through a call to *Runtime.getRuntime().availableProcessors()*.

That being said, we have studied the execution time of several numbers of sequential calls to the *getTaskId()* method of our own *Runnable* implementation. The number of calls were chosen to match the typical number of logical cores contained in nowadays systems, but also to extrapolate any potential leap ahead this figure could see in the future. Table 1 sums up the results of this small experiment, executed on an old Intel Core 2 Duo running at 2.8GHz.

Number of calls to *getTaskId()*	32	64	128	1,024	1,024,000
Execution time (ms)	0.00397	0.00773	0.01601	0.05129	0.10813

Table 1: Computation time of several sequential calls to the *getTaskId()* method

As results in Table 1 show, even a great number of calls to *getTaskId()* will not introduce an overhead in applications making use of TaskLocalRandom. Eventually, the potential synchronization that could appear when the first tasks are started will quickly vanish due to scheduling considerations. All the worker threads will thus scarcely request for a new task identifier at the very same time. In conclusion, the central-element approach is satisfying since it fulfils our needs without impacting the computation time of the application.

4.4 Implementation Details

We have designed TaskLocalRandom for it to be used as an alternative to *ThreadLocalRandom*. It displays the very same interface as its counterpart. The methods contained in our class can produce two kinds of random outputs: double precision floating point values and integers. These are the two kinds of data types that are handled by the original MRG32k3a implementation described in [L'Ecuyer et al., 2002b]. Double precision numbers are natively produced by the algorithm, which manipulates 64-bit floating point values at its heart in order to take advantage of hardware-implemented operations on modern CPUs.

In contrast with *ThreadLocalRandom*, TaskLocalRandom does not inherit from *java.util.Random*, which contains superfluous methods directly bound to the underlying LCG of this class. Still, methods in TaskLocalRandom respect the same interface than *java.util.Random* so that a minimal compatibility is

maintained, without harming our design.

Although TaskLocalRandom might sound similar to the implementation from [L'Ecuyer et al., 2002b], only the rather classical interface is mimicked. Let us recall that TaskLocalRandom is task-aware. It can then be employed safely by users in order to produce highly independent pseudorandom sequences within the tasks of their Java parallel application.

The Java implementation of the central element described in Section 4.3 is achieved through a new abstract class called *RandomSafeRunnable*. This class implements the *Runnable* interface that is traditionally used to describe the behaviour of tasks and threads in Java concurrent applications. *RandomSafeRunnable* stores the identifier of the new task using a single instance of the class *AtomicInteger*, available in the *java.util.concurrent.atomic* package. After being initialized to 0 prior to any task creation, the constructor of *RandomSafeRunnable* performs a call to the thread-safe *getAndIncrement* method from the *AtomicInteger* object. The result of this call acts as the unique task identifier for the lifetime of the task represented by an instance of *RandomSafeRunnable*.

Please note that the way TaskLocalRandom is implemented also ensures that a new task will not keep the identifier of a formerly completed task. In such a case, tasks making use of *ThreadLocalRandom* would have been assigned the same identifier by the JVM. This would have led different tasks to exploit the same pseudorandom stream.

In order to concretely assign a unique pseudorandom sequence to each task, the Jump Ahead algorithm evoked in Section 4.2 is called with the identifier of the task as a parameter. As a result, each task now uniquely identified is assigned the stream corresponding to its identifier. Streams are labelled from the starting point of the original MRG32k3a pseudorandom stream. Please note that this is a design choice we made to assign a different pseudorandom stream to each task. There are situations where each task may require several distinct streams. In that case, TaskLocalRandom could be extended to enable the use of the intrinsic substreams of MRG32k3a, that split each stream in 2^{76} element-long substreams.

4.5 Presentation of the API

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

```
1 public class Foo extends RandomSafeRunnable {
2
3   @Override
4   public void run() {
5
6     TaskLocalRandom rng = new TaskLocalRandom(this);
7
8     for (int i = 0; i < 5; ++i) {
```

```
9         System.out.println("Task[" + this.getTaskId() +
10         "] from Thread[" + Thread.currentThread().getId() +
11         "] {" + i + "} = " + rng.next());
12       }
13     }
14
15     public static void main(String[] args) {
16
17       ExecutorService executor = Executors.newFixedThreadPool(
18             Runtime.getRuntime().availableProcessors() );
19
20       for (int i = 0; i < 100; ++i) {
21         Runnable task = new Foo();
22
23         executor.execute(task);
24       }
25
26       executor.shutdown();
27       while (!executor.isTerminated()) {}
28     }
29 }
```

Listing 1: Presentation of the API of TaskLocalRandom

The implementation detailed in this section makes TaskLocalRandom the equivalent of *ThreadLocalRandom* in regards to its API and features. However, our proposal is more suited to parallelize scientific applications where statistically sound random sources are necessary, and it also fulfils the requirements needed by Java tasks frameworks. That being said, let us compare the performances of *TheadLocalRandom* and TaskLocalRandom.

5 Results

In this part, we compare three aspects of the initial *ThreadLocalRandom* with our proposal TaskLocalRandom: their memory footprint, their numbers throughput and their statistical quality. Then, TaskLocalRandom is faced with the other software tools of the literature introduced in Section 3.

5.1 Memory Footprint and Speed

ThreadLocalRandom wraps a LCG that uses only one integer to store its internal state, whereas MRG32k3a needs at least 6 integers. TaskLocalRandom also relies on an extra task identifier to provide reproducibility as required by stochastic simulations. Thus, *ThreadLocalRandom* is more efficient in terms of memory footprint.

Considering speed, it is hard to isolate accurately the methods involved in random number generation across several threads. That is why we based our comparison on the data produced by the VisualVM[3] profiler to figure out which algorithm was the most efficient. These results shows that TaskLocalRandom is about twice as fast as *ThreadLocalRandom*, requiring about 0.5 ms to pick up a random number whereas *ThreadLocalRandom* requires about 0.8 ms. Therefore, our Java wrapper does not impact the original fastness of the MRG32k3a algorithm. MRG32k3a is actually announced faster than the LCG used by *ThreadLocalRandom* in [L'Ecuyer, 1999].

5.2 Statistical Quality

We have already discussed the statistical quality of LCGs, 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: the whole range of input data will be regularly sliced to feed each computational element. This configuration is also encountered for pseudorandom numbers: each thread or task receives its own pseudorandom stream and uses it to process its part of the data. The data of the corresponding sequential process would be equivalent to a concatenation of all the data chunks, but also of the pseudorandom streams used. As a result, knowing the parallelization techniques used for both random numbers and input data, we could recreate the 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. Although two or more pseudorandom sequences considered independently can produce bad statistical results, their combination can behave differently when faced with the same statistical tests [L'Ecuyer, 1988].

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 task. This situation is sketched in Figure 3. Please note that for the purpose of this test, we fall back to standard Java threads so that *ThreadLocalRandom* can compete fairly with TaskLocalRandom. TaskLocalRandom can actually handle pseudorandom streams distribution across both threads and tasks, the latter being impossible for *ThreadLocalRandom*. Still, this parameter does not impact the results of our experience.

To simulate this situation, we have faced the two PRNGs to TestU01. The random stream studied by the testing battery was provided by concatenating the substreams of a given number of threads. In Table 2, each PRNG is tested using combined streams resulting from what would be the concatenated random sequence of 16 to 64 threads.

Table 2 shows that using MRG32k3a instead of the LCG implemented in

[3] `http://visualvm.java.net/` (last access 11/1/13)

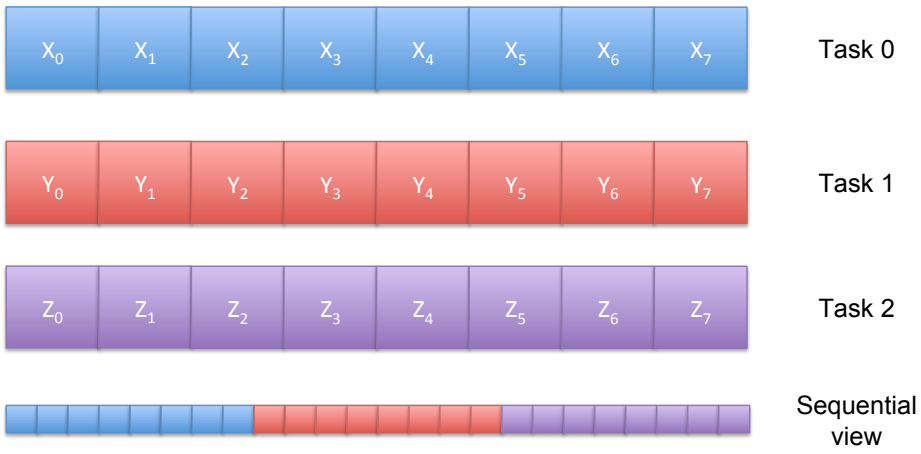

Figure 3: Substreams allotted to 3 different tasks and the corresponding pseudorandom sequence from the point of view of a sequential process

	16 threads	32 threads	64 threads
ThreadLocalRandom	all	all	all
TaskLocalRandom	none	none	none

Table 2: BigCrush failed results for *ThreadLocalRandom* and TaskLocalRandom used by 16, 32 and 64 threads. Each test configuration was initialized with 60 different seed-statuses

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, whereas TaskLocalRandom does not generate any failed output. This figure backs our PRNG choice for the underlying algorithm of TaskLocalRandom. A more thorough examination of the multiple streams of MRG32k3a is to be found in [L'Ecuyer et al., 2002b]. The authors have tested their algorithm both theoretically via the spectral test and empirically by statistical tests.

5.3 Comparison of Java PRNG Libraries

This section recalls the major characteristics of all the Java PRNG facilities that we described in Section 3. Table 3 aims at comparing these characteristics with those of TaskLocalRandom. We particularly focus on the ability of each library to automatically deal with pseudorandom streams distribution. The different criteria involved in the comparison are as follows:

- How many PRNG algorithms are embedded in the library?

- Does the library automatically handle pseudorandom streams distribution across threads?

- Does the library automatically handle pseudorandom streams distribution across tasks?

	Number of embedded PRNGs	Automatic distribution across threads	Automatic distribution across tasks
JAPARA	2	No	No
SSJ	11	No	No
DistRNG	9	No	No
ThreadLocalRandom	1	**Yes**	No
TaskLocalRandom	1	**Yes**	**Yes**

Table 3: Ability of Java PRNG facilities to deal with threads and tasks

As we can see in Table 3, TaskLocalRandom is the only library that automatically takes into account pseudorandom streams distribution at a task level, while the others would force the developer to determine a distribution scheme through the tasks of his concurrent application.

6 Discussion

In this paper, we propose a Java implementation of L'Ecuyer's MRG32k3a that behaves correctly when used with Java tasks frameworks. However, simulation practitioners often expect to challenge their stochastic models with different random sources. In this way, providing a wider set of PRNGs is relevant for the simulation community. This complete framework would obviously display an API identical to TaskLocalRandom. 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 another highly reliable distribution technique: parameterization [Hill et al., 2013]. While Sequence Splitting intends to slice an original random sequence in several independent random streams, parameterization tackles the problem differently. 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 tasks.

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 TaskLocalRandom: 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 shipped with an adapted version of the Dynamic Creator (DC) software tool [Matsumoto and Nishimura, 2000] that can create more than $2^{32} \times 2^{16}$ highly independent statuses. As always with Mersenne-Twister-like PRNGs, this algorithm is based upon linear recurrences. Then, Matsumoto and Nishimura assume that "a set of PRNGs based on linear recurrences is 'mutually' independent if the characteristic polynomials are relatively prime to each other" [Matsumoto and Nishimura, 2000].

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, this PRNG might show less flexible than MRG32k3a since its parameterized statuses need to be precomputed by the Dynamic Creator algorithm. DC relies on several C++ libraries, and would thus be difficult to reimplement in Java in a portable way. Thus, to provide a full Java concurrent implementation, not only we need to implement the algorithm, but also to ship precomputed statuses with it. The point is to find a tradeoff between a sufficient amount of parameterized statuses and a reasonable memory footprint, so that the sole PRNG does not bloats the whole application. Each task will then receive an instance of TaskLocalTinyMT 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 partitionning 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 number throughput. They appear to be better suited than TinyMT (or any other member of the Mersenne Twister family) to target a smooth integration in Java tasks frameworks since their parameters are formed by a single key that can be set at runtime according to each task'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 task'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. Furthermore, *ThreadLocalRandom* is intrinsically unable to deal with tasks executed within Java Thread Pools: it assigns the same pseudorandom stream to all the tasks handled by the same worker thread. We have detailed why this behaviour was obviously unsuitable when considering scientific applications.

Additionally, this study surveys the most spread libraries targeting 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 TaskLocalRandom as another alternative to *ThreadLocalRandom*. Our proposal respects the same API as *ThreadLocalRandom*, but it relies on MRG32k3a, a well-known Crush-resistant PRNG. TaskLocalRandom displays not only a far better statistical quality than its JDK counterpart but it is also much more suited for scientific applications, given that it issues a reproducible output by default. TaskLocalRandom is a bit greedier than *ThreadLocalRandom* in terms of memory consumption, but it completely outperforms its counterpart in both speed and statistical quality. According to our measures, TaskLocalRandom is about twice as fast as *ThreadLocalRandom* and passes all the tests of BigCrush: the most stringent testing battery from TestU01.

The major input brought by TaskLocalRandom lies in its cooperation with the *RandomSafeRunnable* abstract class that we also introduced in this study. This pair of classes enables a correct distribution of pseudorandom streams among tasks. It is, to our knowledge, the sole PRNG facility that can be used safely within a Java task framework such as *Executors* or *Fork/Join*.

Among the simulation community, it is a safe practice to check the results of a stochastic simulation using several PRNGs which rely on different internal mechanisms. This is why 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. This way, simulationists could change the PRNG they use in an instant, and still benefit of correct pseudorandom streams distribution across their Java tasks.

Acknowledgments

The authors would like to thank Florian Guillochon for his careful reading and useful suggestions on the draft. We also thank Sarah Dean for having meticulously proofread this paper.

References

[Brown et al., 2009] Brown, R. G., Eddelbuettel, D., and Bauer, D. (2009). DieHarder: a random number test suite.

[Coddington and Newell, 2004] Coddington, P. and Newell, A. (2004). JAPARA-A java parallel random number generator library for high-performance computing. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)-Workshop*, volume 5, page 156–166.

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

[De Matteis and Pagnutti, 1988] De Matteis, A. and Pagnutti, S. (1988). Parallelization of random number generators and long-range correlations. *Numerische Mathematik*, 53(5):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):3382–3384.

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

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

[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 et al., 2013] Hill, D. R., Mazel, C., Passerat-Palmbach, J., and Traoré, M. K. (2013). Distribution of random streams for simulation practitioners. *Concurrency and Computation: Practice and Experience*, 25:1427–1442. doi:/10.1002/cpe.2942.

[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*, page 36–43.

[Leach et al., 2005] Leach, P. J., Mealling, M., and Salz, R. (2005). A universally unique identifier (UUID) URN namespace - RFC 4122. Technical report, Internet Engineering Task Force (IETF).

[L'Ecuyer, 1988] L'Ecuyer, P. (1988). Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–751.

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

[L'Ecuyer, 2001] L'Ecuyer, P. (2001). Software for uniform random number generation: Distinguishing the good and the bad. In *Simulation Conference, 2001. Proceedings of the Winter*, volume 1, pages 95–105. IEEE.

[L'Ecuyer, 2010] L'Ecuyer, P. (2010). Pseudorandom number generators. In *Encyclopedia of Quantitative Finance*, volume Simulation Methods in Financial Engineering, pages 1431–1437. John Wiley & Sons, Ltd, Chichester, UK, e. platen and p. jaeckel edition.

[L'Ecuyer and Côté, 1991] L'Ecuyer, P. and Côté, S. (1991). Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software (TOMS)*, 17(1):98–111.

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

[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., 2002b] L'Ecuyer, P., Simard, R., Chen, E. J., and Kelton, W. D. (2002b). 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. R. C., 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). DIEHARD: a battery of tests of randomness.

[Mascagni and Srinivasan, 2000] Mascagni, M. and Srinivasan, A. (2000). Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software (TOMS)*, 26(3):436–461.

[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.

[Matsumoto and Nishimura, 2000] Matsumoto, M. and Nishimura, T. (2000). Dynamic creation of pseudorandom number generators. In Niederreiter, H. and Spanier, J., editors, *Monte Carlo and Quasi-Monte Carlo Methods 1998*, pages 56–69. Springer.

[MPI Forum, 1993] MPI Forum (1993). Document for a standard message-passing interface. Technical report, University of Tennessee.

[Passerat-Palmbach et al., 2012a] Passerat-Palmbach, J., Mazel, C., and Hill, D. R. C. (2012a). Pseudo-random streams for distributed and parallel stochastic simulations on GP-GPU. *Journal of Simulation*, 6(3):141–151. doi:10.1057/jos.2012.8.

[Passerat-Palmbach et al., 2012b] Passerat-Palmbach, J., Mazel, C., and Hill, D. R. C. (2012b). ThreadLocalMRG32k3a: a statistically sound substitute to pseudorandom number generation in parallel java applications. In *Proceedings of the IEEE High Performance Computing and Simulation conference*, pages 543–550. *(nominated for the outstanding paper award).*

[Passerat-Palmbach et al., 2010] Passerat-Palmbach, J., Mazel, C., Mahul, A., and Hill, D. R. C. (2010). Reliable initialization of GPU-enabled parallel stochastic simulations using mersenne twister for graphics processors. In *Europeans Simulation and Modeling Conference 2010*, pages 187–195. ISBN: 978-90-77381-57-1.

[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*, 24(7):723–738.

[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). Tiny mersenne twister (TinyMT): a small-sized variant of mersenne twister.

[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, page 16:1–16:12, Seattle, Washington. 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/