# Fast Splittable Pseudorandom Number Generators

Guy L. Steele Jr.
Oracle Labs
guy.steele@oracle.com

Doug Lea
SUNY Oswego
dl@cs.oswego.edu

Christine H. Flood
Red Hat Inc
chf@redhat.com

## Abstract

We describe a new algorithm SPLITMIX for an *object-oriented* and *splittable* pseudorandom number generator (PRNG) that is quite fast: 9 64-bit arithmetic/logical operations per 64 bits generated. A conventional linear PRNG object provides a *generate* method that returns one pseudorandom value and updates the state of the PRNG, but a splittable PRNG object also has a second operation, *split*, that replaces the original PRNG object with two (seemingly) independent PRNG objects, by creating and returning a new such object and updating the state of the original object. Splittable PRNG objects make it easy to organize the use of pseudorandom numbers in multithreaded programs structured using fork-join parallelism. No locking or synchronization is required (other than the usual memory fence immediately after object creation). Because the *generate* method has no loops or conditionals, it is suitable for SIMD or GPU implementation.

We derive SPLITMIX from the DOTMIX algorithm of Leiserson, Schardl, and Sukha by making a series of program transformations and engineering improvements. The end result is an object-oriented version of the purely functional API used in the Haskell library for over a decade, but SPLITMIX is faster and produces pseudorandom sequences of higher quality; it is also far superior in quality and speed to `java.util.Random`, and has been included in Java JDK8 as the class `java.util.SplittableRandom`.

We have tested the pseudorandom sequences produced by SPLITMIX using two standard statistical test suites (DieHarder and TestU01) and they appear to be adequate for "everyday" use, such as in Monte Carlo algorithms and randomized data structures where speed is important.

***Categories and Subject Descriptors*** G.3 [*Mathematics of Computing*]: Random number generation; D.1.3 [*Software*]: Programming techniques—Concurrent programming

***General Terms*** Algorithms, Performance

***Keywords*** collections, determinism, Java, multithreading, nondeterminism, object-oriented, parallel computing, pedigree, pseudorandom, random number generator, recursive splitting, Scala, spliterator, splittable data structures, streams



*OOPSLA '14*, October 20–24, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2660193.2660195

## 1. Introduction and Background

Many programming language environments provide a *pseudorandom number generator* (PRNG), a deterministic algorithm to generate a sequence of values that is likely to pass certain statistical tests for "randomness." Such an algorithm is typically described as a finite-state machine defined by a *transition function* $\tau$, an *output function* $\mu$, and an initial state (or *seed*) $s_0$; the generated sequence is $z_j$ where $s_j = \tau(s_{j-1})$ and $z_j = \mu(s_j)$ for all $j \geq 1$. (An alternate formulation is $s_j = \tau(s_{j-1})$ and $z_j = \mu(s_{j-1})$ for all $j \geq 1$, which may be more efficient when implemented on a superscalar processor because it allows $\tau(s_{j-1})$ and $\mu(s_{j-1})$ to be computed in parallel.) Because the state is finite, the sequence of generated states, and therefore also the sequence of generated values, will necessarily repeat, falling into a cycle (possibly after some initial subsequence that is not repeated, but in practice PRNG algorithms are designed so as not to "waste state," so that in fact any given initial state $s_0$ will recur). The length $L$ of the cycle is called the *period* of the PRNG, and $s_i = s_j$ iff $i \equiv j \bmod L$.

Characteristics of PRNG algorithms and their implementations that are of practical interest include period, speed, quality (ability to pass statistical tests), size of code, size of data (some algorithms require large tables or arrays), ease of jumping forward (skipping over intermediate states), partionability or splittability (for use by multiple threads operating in parallel), reproducibility (the ability to run a program twice from a specified starting state and get exactly the same results, even if parallel computation is involved), and unpredictability (how difficult it is to predict the next output given preceding outputs). These characteristics trade off, and so different applications typically require different algorithms.

A *linear congruential generator* (LCG) [13, §3.2.1] has as its state a nonnegative integer less than some modulus $M$ (which is typically either a prime number or a power of 2); the transition function is $\tau(s) = (a \cdot s + c) \bmod M$ and the output function is just the identity function $\mu(s) = s$. The Unix library function `rand48` uses this algorithm with $a = 25214903917 =$ `0x5DEECE66Dull`, $c = 11 =$ `0xB`, and $M = 2^{48}$. Ever since the original Java™ Language Specification [10] in 1995, the class `java.util.Random` has been specified to use this same algorithm (see Figure 1). This code is simple, concise, and adequate for its originally intended purpose: to supply pseudorandomly generated values to be used relatively infrequently by a smallish number of concurrent threads running within a set-top box or web

```
public class Random {
 protected long seed;
 public Random() {
   this(System.currentTimeMillis()); }
 public Random(long seed) { setSeed(seed); }
 synchronized public void setSeed(long seed) {
   this.seed = (seed ^ 0x5DEECE66DL)
               & ((1L << 48) - 1); }
 synchronized protected int next(int bits) {
   seed = (seed * 0x5DEECE66DL + 0xBL)
          & ((1L << 48) - 1);
   return (int)(seed >>> (48 - bits)); }
 public int nextInt() { return next(32); }
 public int nextLong() {
   return ((long)next(32) << 32) + next(32); }
 public double nextDouble() {
   return (((long)next(26) << 27) + next(27))
               / (double)(1L << 53); }
}
```

**Figure 1.** The essence of `java.util.Random`

browser. It has a number of drawbacks, however, that make it inappropriate for use in "serious" applications: (1) Its short period (consider that $2^{48}$ nanoseconds is less than 80 hours). (2) While the `rand48` algorithm was considered to be of relatively high quality when introduced in 1982 [28], more recent tests [17] have uncovered significant flaws in its statistical behavior. (3) While it is thread-*safe*, thanks to its use of synchronization locks, it is not thread-*efficient*: if many threads share a single instance of `Random`, then contention for the lock may become a performance bottleneck. (4) On the other hand, if many instances of class `Random` are created, one for each thread, there is no guarantee that the values collectively generated by those many instances will be as statistically pseudorandom as if they had been generated by a single instance of class `Random`. (If 256 threads each have a `Random` object with its initial seed chosen at random, and then each thread generates $2^{32}$ values, it is more likely than not, thanks to the Birthday Paradox, that two of the generated sequences will have some overlap.)

Another kind of finite-state machine used for this purpose is the *linear feedback shift register* (LFSR), in which the state is a vector of bits contained in a shift register; the transition function shifts the register by one position, shifting in a new bit value that is computed as a fixed linear function of the current vector (typically the exclusive OR of specific bits of the vector), and the output function provides the bit that was shifted out. This arrangement provides a stream of bits, which may then be chunked into groups of, say, 32 or 64 consecutive bits to produce a sequence of integers. In software, it may be more convenient for code to execute multiple steps of the LFSR algorithm in one shot to produce a multibit output all at once. Marsaglia's XORWOW algorithm [22] is a hybrid that uses both an Xorshift method (which is provably equivalent to an LFSR [4]) and an LCG with $a = 1$:

```
unsigned long xorwow() {
  static unsigned long
    x=123456789, y=362436069, z=521288629,
    w=88675123, v=5783321, d=6615241;
  unsigned long t=(x^(x>>2));
  x=y; y=z; z=w; w=v; v=(v^(v<<4))^(t^(t<<1));
  return (d+=362437)+v; }
```

The period of XORWOW is pretty good ($2^{32}(2^{160} - 1) = 2^{192} - 2^{32}$), and generation of one 32-bit value requires just 9 arithmetic operations: 3 shifts, 4 XORs, and 2 adds. (There are also a number of assignment operations, which could be optimized away in the context of a suitably unrolled loop.) Unfortunately, while XORWOW is also fairly fast, very recent testing [29] has also revealed statistical flaws.

There are other PRNG algorithms with much longer period that produce sequences of much higher quality, but they are slower. These include the Mersenne Twister [23], which is related to LFSR algorithms in relying on bit-shifting and bitwise operations, and MRG32k3a [9], which is related to LCG algorithms in relying on computation of linear formulae modulo prime numbers. There are also many PRNG algorithms, such as those in `java.security.secureRandom`, that produce pseudorandom sequences of superb quality, suitable for use in cryptographic and security applications, but their algorithms are substantially slower.

The algorithms we have described so far are sequential (single-threaded), but there is a need for algorithms that are effective in a parallel (SIMD or multi-threaded) computing environment. One approach is to *partition* the cycle of an otherwise sequential PRNG algorithm. This is simple if there is a cheap way to "jump forward" $n$ steps from any given state $s$ by using some computational shortcut to compute $\tau^n(s)$. (For example, if $\tau(s) = (a \cdot s) \bmod M$ then $\tau^n(s) = (a^n \bmod M) \cdot s \bmod M$, and it may be possible to compute (or precompute) $a^n \bmod M$ cheaply.) This approach works well when the number of threads $N$ to be used is known at the start of a computation: if a PRNG has period $L$, then a global initial state $s_0$ is chosen and then the PRNG for thread $i$ is given initial state $\tau^{i\lfloor \frac{L}{N} \rfloor}(s_0)$. The MRG32k3a algorithm does have a good shortcut for jumping ahead (it requires precomputation of two $3 \times 3$ matrices), and there is a software package `RngStream` that uses such jumping ahead for partitioning its cycle into streams and substreams [19].

On the other hand, some applications are organized in such a way that new threads may be created dynamically, where any thread may choose to spawn one or more new threads at any time. To avoid synchronization overhead, it is desirable for an existing thread to be able to initialize the PRNG state for a new thread without having to communicate with any other thread and without having to update any shared global data structure; ideally it should need to update only the state of its own PRNG instance. In effect, any thread can turn one PRNG instance into two; a PRNG algorithm that can accommodate this requirement is said to be *splittable*.

One can imagine constructing a splittable PRNG from a partitionable PNRG as follows: when a thread is created, it is given a tuple $(s, l)$ to remember—this defines a portion of the PRNG cycle of length $l$ starting from state $s$—and then uses $s$ as the initial state for its own PRNG. (For the initial thread, $l = L$, the period of the PRNG.) When an existing thread $t$ having tuple $(s, l)$ spawns a new thread $u$, then it cuts its allotted portion in half and gives the other half to $u$: thread $t$ computes $h = \lfloor \frac{l}{2} \rfloor$, replaces its own tuple $(s, l)$ with $(s, h)$, and then gives new thread $u$ the tuple $(\tau^h(s), l - h)$. This idea could be applied to the `RngStream` package: while its API allows only sequential generation of its multiple streams, it could easily be extended to allow a set of multiple streams to be split in half instead (this would require the precomputation of a table of 192 pairs of $3 \times 3$ matrices, one for each power of 2 smaller than its period).

This works well if the tree of tasks induced by task spawning is balanced (more precisely, if it does not get too deep). But some applications may, depending on the data being processed, produce task trees that have long left spines or are otherwise very deep. A strategy that repeatedly splits a discrete resource of size $L$ in half will run out of steam after $\log_2 L$ splits; for MRG32k3a, that's only 192 splits.

Another idea is to take a small risk: instead of *guaranteeing* that different threads use different parts of the cycle, one can settle for making it *highly likely* that different threads use different parts of the cycle, by choosing the initial state for the PRNG of a new thread "randomly" [6]; the pitfall is that if cycle portions used by two threads do accidentally overlap, then their sequences become highly correlated.

The Haskell standard library `System.Random` provides an API and implementation for a splittable, pure PRNG:

```
data StdGen = StdGen Int32 Int32
stdNext :: StdGen -> (Int, StdGen)
stdSplit :: StdGen -> (StdGen, StdGen)
```

The generator state `StdGen` is simply a pair of integers; the function `stdNext` takes a state and returns a pair of a generated value and a new state—in other words, given state $s$ it returns $(\mu(s), \tau(s))$—while the function `stdSplit` takes a state and returns a pair of states that may thenceforward be used independently. The API is beautiful; the implementation turns out to have a severe flaw (see Section 7).

We are interested in splittable random number generators that are easy to use in multithreaded settings where the shape of the task tree is unpredictable, are quite fast, and have aggregate statistical properties of sufficiently good quality for "everyday use," by which we mean not only randomized data structures but also Monte Carlo algorithms for machine learning and scientific simulation. We want to integrate these algorithms into the *streams* framework of Java JDK8, which supports the processing of collections of data in a functional map/reduce style. For example, we would like the expression

```
prng.doubles(n).parallel().map(x->x*x).sum()
```

to compute the sum of the squares of `n` pseudorandomly chosen `double` values in the range $[0, 1)$, and to do so efficiently, using parallel computational resources if possible. We care about reproducibility but not unpredictability.

We present two parallel PRNG algorithms. The first is a highly optimized version of the pedigree-based DOTMIX algorithm of Leiserson, Schardl, and Sukha [20] that we believe is interesting and useful in its own right and may serve as an alternate jumping-off point for further research. We present an early version of this algorithm in Scala, and further refined versions of it in Java. The second algorithm, SPLITMIX, was inspired by the first, but makes no use of pedigrees, dot products, or tables of coefficients.

A specific 64-bit instantiation of this novel SPLITMIX design (a principal contribution of this paper) is presented in Section 3, but it is far from obvious that such a simple approach to generating values and splitting objects is a good one. Therefore we first explain in Section 2 how we derived this design by a stepwise refinement of the DOTMIX algorithm that included one or two leaps of faith requiring after-the-fact analysis and testing; the details of this process are the other principal contribution of this paper. This includes an improved use of avalanche statistics to select good mixing functions, and an understanding of why the avalanche statistics cannot (or should not) be improved past a particular point; we discuss this in Section 4. We present quality measurements of SPLITMIX in Section 5 and speed measurements in Section 6, discuss related work in Section 7, and offer conclusions and future work in Section 8.

## 2. Derivation of the SPLITMIX Algorithm

We were inspired by the 2012 PPoPP paper of Leiserson, Schardl, and Sukha [20] (hereafter "LSS"). We summarize their idea here and discuss other related work in Section 7.

Assume a fork-join framework for parallel tasks, in which each task performs a sequence of actions. There are three kinds of actions: *generate* a pseudorandom value, *spawn* (that is, *fork*) a child task, and *sync* with (that is, *join*) a previously spawned child task. Except for the causality constraints implied by forking and joining, tasks execute freely in parallel. Initially there is a single root task.

Each task has some local state, namely an integer counter and a pointer to its parent task (the task that spawned it); the root task has a null parent pointer. The algorithm described by LSS takes certain steps to maintain this state at each *generate*, *spawn*, and *sync* action; this three-way division of labor was chosen so as to minimize overhead for tasks that do not generate random numbers. We present a simplified reformulation of their idea here that requires state maintenance only during *generate* and *spawn* actions; therefore in the remainder of this paper we need not discuss the detailed behavior of *sync* actions any further, and we speak generally of only two kinds of action, *generate* and *spawn*.

If we identify each task other than the root with the action that spawned it, we have a tree whose internal nodes

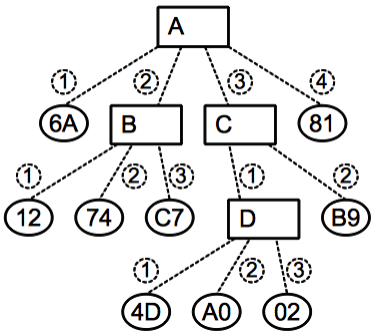


**Figure 2.** Tree of tasks and generated numbers

are tasks; all *generate* actions are leaves of this tree. (If a spawned task happens to perform no actions, it will also be a leaf.) Figure 2 shows such a tree. Task A is the root task; it performs four actions *generate*, *spawn*, *spawn*, *generate* (in that order). Task B is the first task spawned by task A, and it performs three *generate* actions. Task C is the second task spawned by task A, and it performs a *spawn* and a *generate*. Task D is the task spawned by task C, and it performs three *generate* actions. In the figure, the edges connecting each task to its actions are numbered sequentially starting from 1.

LSS observe that every action in such a tree has a unique *pedigree*, the sequence of edge labels on the path within the tree from the root task down to the action. As examples, the second value "74" generated by task B has pedigree $\langle 2, 2\rangle$, the first value "4D" generated by task D has pedigree $\langle 3, 1, 1\rangle$, and the pedigree of the last value "81" generated by task A has pedigree $\langle 4\rangle$. Their idea is that every *generate* action can produce a value by hashing its pedigree. They describe more than one way of doing this, but we focus on their "DOTMIX" technique, which hashes a pedigree in two steps: (1) compute the dot product of the pedigree (regarded as a vector) with a vector of random coefficients, then (2) apply a bijective "mixing function" $\mu$ to the result. We also adopt their suggestion of adding a seed value $\sigma$ to the dot product before applying the mixing function.

The calculation of the dot product and the addition of $\sigma$ to the result must be done modulo a prime $p$. For a computer with a word width of $w$ bits, LSS choose $p$ to be the largest prime less than $m = 2^w$. Assume that every element $j_k$ of a pedigree satisfies $1 \le j_k < p$ and every random coefficient $\gamma_i$ satisfies $1 \le \gamma_k < p$. The sum of $\sigma$ and the dot product (modulo $p$) lies in the range $[0, p)$ and so can be treated as a $w$-bit binary value (because $p < 2^w$), from which the mixing function $\mu$ can produce a $w$-bit result. A *generate* action having pedigree $\langle j_1, j_2, \ldots, j_D\rangle$ therefore computes $\mu\big(\big(\sigma + \sum_{k=1}^{D} \gamma_k j_k\big) \bmod p\big)$. LSS present a proof that if the $\gamma_i$ are chosen uniformly at random from the range $[0, p)$ then the probability that two distinct pedigrees will produce the same dot-product value is $1/p$. A specific version presented by LSS uses a fixed table of 64-bit coefficients $\gamma_i$ (chosen by some random or pseudorandom process) and uses $p = 2^{64} - 59$ (we shall call this prime number "Fred").

LSS then note that while the dot-product formula effectively hashes pedigrees into the range $[0, p)$ with a small probability of collision, nevertheless two similar pedigrees may produce "similar" hash values, whereas we would prefer them to be statistically "dissimilar"; the mixing function is intended to address this point. LSS use a mixing function based on the RC6 block cipher [27], namely $\mu(z) = f(f(f(f(z))))$ where $f(z) = \phi((2z^2 + z) \bmod m)$ and $\phi(x)$ is the function that swaps the two halves of a $w$-bit word $x$ (for even $w$); if $w = 64$, then a standard "rotate left 32" instruction does the job. Thus $f$ is easily calculated on most computers using four operations (one of them an integer multiply), and so calculating $\mu$ requires 16 instructions.

LSS describe an implementation of the DOTMIX algorithm as part of MIT Cilk 5.4.6, a programming language and runtime that support fork-join parallelism, and present and measure a sample Cilk program that uses pseudorandom sequences generated by this algorithm. They also describe a technique for using "scoped" pedigrees so as to allow a subcomputation that uses pseudorandom values to be executed more than once in such a way that the multiple runs generate exactly the same random values, even though corresponding actions in the subcomputations would of course have distinct pedigrees globally. Such a facility allows for controlled and repeatable testing of the code for such subcomputations.

Computing a dot product involves following the chain of parent pointers from the current task to the root task (or to the limit of the pedigree scope); the counter values found along the way are the pedigree. LSS remark that dot products can be computed incrementally and report that they tried "memoizing" portions of the dot-product calculations to allow *generate* operations to be performed in time $O(1)$ rathe than $O(d)$ (where $d$ is the length of the pedigree), but did not see any speed benefit in practice [20, §8].

We derive a series of algorithms from DOTMIX by stepwise refinement. Our final SPLITMIX algorithm is discussed in detail in Section 2.4; before that, Sections 2.1, 2.2, and 2.3 present three intermediate points in the refinement process (in part because these algorithms are plausible alternate points of departure for future work).

### 2.1 An Implementation in Scala

We set out to implement the DOTMIX algorithm within the framework of the Scala programming language [24], both to obtain the stylistic benefits of the DOTMIX approach and to see whether we could identify any useful improvements. (Our earliest prototype was actually done within the runtime system for Fortress [1], for which language the LSS design seemed ideal.) Our idea was to extend the Scala `App` class (which conveniently provides access to certain programming facilities in a way that Java does not), so that a main program need only extend our new library class `ParallelApp` in order to have available (1) a `parallel` construct that would evaluate two expressions in parallel, and (2) operations such as `nextLong()` and `nextDouble()` for obtaining

```
import ParallelRNG.ParallelApp
object ScalaPiCalc extends ParallelApp {
  def tries(n: Int): Int = {
    var count: Int = 0
    for (k <- 0 until n) {
      val x = nextDouble()
      val y = nextDouble()
      if (x*x + y*y < 1.0) count += 1
    }
    count
  }
  val iters = 1000000000
  var xtime: Long = 0
  for (j <- 0 to 9) {
    val basetime = System.nanoTime()
    val result = (tries(iters) * 4.0) / iters
    xtime += (System.nanoTime() - basetime)
    println("Result: " + result) }
  println("Total time: " + xtime)
}
```

**Figure 3.** Monte Carlo approximation of $\pi$ in Scala

```
  def tries(n: Int): Int = {
    if (n < 1000000) {
      var count: Int = 0
      for (k <- 0 until n) {
        val x = nextDouble()
        val y = nextDouble()
        if (x*x + y*y < 1.0) count += 1
      }
      count
    } else {
      val h = n >>> 1
      val (p, q) = parallel(tries(h),
                            tries(n-h))
      p+q
    }
  }
```

**Figure 4.** Parallel-sequential approximation of $\pi$ in Scala

pseudorandomly generated values. Figure 3 shows a typical simple application: Monte Carlo calculation of approximate values for $\pi$ by sequentially choosing random points uniformly within a unit square and counting how many fall within a quarter-circle of unit radius. The code also contains timing instrumentation and reporting. Figure 4 shows an alternate definition for the `tries` method of Figure 3 that recursively splits each request into two parallel tasks until the number of requested iterations is less than `1000000`. Calling `nextDouble()` generates a pseudorandom `Double` value, chosen uniformly from the range $[0.0, 1.0)$, using implicitly maintained local task state; all that is needed to execute two expressions *p* and *q* in parallel is to say `parallel(`*p*`, `*q*`)`.

The code for our library is shown in Figures 5–10. We made ten changes to the DOTMIX algorithm:

```
object Rand {
  val signbit = 0x8000000000000000L
  def unsignedGE(x: Long, y: Long) =
    ((x ^ signbit) >= (y ^ signbit))
  def update(dotp: Long, gamma: Long) = {
    val result = dotp + gamma
    if (unsignedGE(result, dotp)) result
    else (if (unsignedGE(result, 13L)) result
          else result + gamma) - 13L }
  def mix64(x: Long): Long = {
    var z = x
    z = (x ^ (x >>> 33)) * 0xff51afd7ed558ccdL
    z = (z ^ (z >>> 33)) * 0xc4ceb9fe1a85ec53L
    z ^ (z >>> 33) }
  val longs = Array(    / Random 64-bit values
    0x14CBB762934FA47FL, 0xA7224BC18A26FD10L,
    0xF2281E2DBA6606F3L, 0xBD24B73A95FB84D9L,
    ...                  // 1024 entries in all
   0x91673D71CFCAA9A1L, 0xBBD6771B9BA86215L )
}
```

**Figure 5.** Utility routines in object `Rand`

```
trait AnyTask {
  val depth: Int
  val gamma: Long
  var dotProd: Long
  def newChild[T](x: =>T): Task[T]
  def next64Bits(): Long }
```

**Figure 6.** Scala task trait containing PRNG state

**[1]** Every task maintains a count of its actions (LSS call this the *rank*); we increment this counter at the *start* of every action (LSS do this incrementation at other points in the computation, which causes *sync* operations to get involved).
**[2]** Instead of adding the seed $\sigma$ after calculating the dot product, we use $\sigma$ to initialize the accumulator for the dot product calculation; this enables inclusion of $\sigma$ in common subexpressions. From now on, when we speak of the "dot product" we will mean a value that already incorporates $\sigma$.
**[3]** Rather than memoizing dot products in a cache, we apply common-subexpression elimination and strength reduction. Each task retains in its local state the most recent dot product $\sigma+\sum_{k=1}^{D}\gamma_k j_k$ it computed; to compute the next dot product after incrementing $j_D$, the task need only add $\gamma_D$ to the previous dot product value. This improves speed greatly.
**[4]** When a task spawns a child task, it first computes a new dot product exactly as if it were going to perform a *generate* action. It then sets the child's counter to 0 and its dot product equal to its own new dot product value. A routine inductive proof shows that when the child then proceeds to calculate a new dot product of its own, the correct value is produced.
**[5]** Each task uses exactly one of the random coefficicents, namely $\gamma_D$ where $D$ is the depth of the task in the tree. Instead of repeatedly fetching $\gamma_D$ from the table, we made each task cache $\gamma_D$ as part of its local state when it is created.

**[6]** The consequence of previous changes is that no action actually uses the local counters! Therefore the counters can be eliminated. Rather then maintaining the pedigree explicitly, the algorithm maintains dot products directly.

**[7]** At this point, the only use of the parent pointer is to calculate the depth $D$ of the task. We eliminate the parent pointer and introduce a local variable in each task to track its depth. When a task spawns a child task, it sets the child's depth to 1 more than its own depth.

**[8]** After all these changes, measurements showed that the mixing function was now taking a substantial fraction of the total time for a *generate* action. We searched for cheaper mixing functions. LSS reported that $\mu(z) = f(f(f(f(z))))$ is an adequate mixing function but $\mu(z) = f(f(z))$ is not (as determined by using the DieHarder [5] test suite to test the resulting pseudorandom sequences). Curiously, LSS did not report also trying $\mu(z) = f(f(f(z)))$, so we tested it and found it adequate, using 12 instructions instead of 16. Then we searched the literature for other bit-mixing functions and found the MurmurHash3 64-bit finalizer [2], identified by Austin Appleby using a simulated-annealing algorithm designed to find mixing functions with good avalanche statistics. (We discuss this idea in detail in Section 4.) The MurmurHash3 finalizer needs only 8 arithmetic/logical operations and seems to be adequate as a mixing function in the DOTMIX algorithm. The 14 variants identified by David Stafford [33] also seem to work well for this purpose.

**[9]** We noticed an aesthetic flaw in DOTMIX: Because arithmetic for the dot product is performed modulo $p$ and $p < m$, there are some 64-bit values that are never generated, namely $\mu(z)$ for $p \le z < m$. For $m = 2^{64}$ and $p =$ Fred, 59 of the possible $2^{64}$ 64-bit values cannot occur. This is a relatively tiny nonuniformity, unlikely to be picked up by statistical tests, yet it bothered us. Then we realized that there were two principal reasons for choosing $p < m$: (i) if $p > m$ then arithmetic calculations mod $p$, especially multiplication, are much more difficult to implement using $w$-bit arithmetic; and (ii) if $p > m$ then not all arithmetic results can be represented in $w$ bits. But we can overcome both of these reasons: (i) Thanks to the strength reduction optimization, we no longer perform any multiplications mod $p$, and addition mod $p$ is not so difficult even if $p > m$; and (ii) we don't really care about generating all $p$ possible values—we just want dot product values likely to be different—so if we ever calculate a dot product that is not representable, we simply discard it and try again. If we calculate $p$ dot products successively, necessarily generating all $p$ values in the range $[0, p)$, then $p-m$ values (those not less than $m$) will be discarded and $m$ values will be accepted, and so we will have generated every value in the range $[0, m)$ exactly once. Therefore our Scala implementation performs its dot-product arithmetic modulo George $= 2^{64} + 13$.[1] Moreover, if we ensure that no $\gamma_i$ is smaller than $p - m$ (which is almost surely true anyway if the $\gamma$ values are chosen randomly) then it is never necessary to "try again" more than once, so we need only a conditional, not a loop, in the code that computes new dot products. We believe that this use of an unrepresentable prime modulus in a PRNG is a theoretical novelty.

[1] At least one of us likes to call two primes "magical twins" if there is no other prime number between them but there is a power of 2 between them.

**[10]** We found a simpler way to provide the "scoped pedigree" functionality described by LSS: a construct to spawn a new task such that the user, rather than the spawning task, provides the initial value for the dot product. The Scala construct `withNewRNG(`*seed*`) { `*body*` }` executes the *body* in a new task whose dot product has been initialized to *seed* and then immediately does a *join* with that new task. Unlike the LSS scoped pedigree approach, `withNewRNG` does not allow any given task to access more than one generator at a time, but an advantage is that there is no need for user code to explicitly manipulate data structures describing the scope. We believe that this technique is also novel.

In Figure 5, the method `unsignedGE` compares `Long` values as if they were bit representations of unsigned 64-bit values (an extremely clever compiler would compile this to a single unsigned-compare machine instruction). The method `update` uses arithmetic modulo George to add `gamma` to `dotp` once, or twice if necessary, to produce a new value representable in 64 bits. The method `mix64` implements the MurmurHash3 64-bit finalizer function [2]. The array `longs` is a table (shown here only in part) of 1024 "truly random" values obtained from HotBits [35].

Figure 6 declares a Scala trait `AnyTask`, which abstractly defines the local state of a task (fields `depth`, `gamma`, and `dotProd`) and methods for calculating the next dot product, spawning a child task, and generating 64 pseudorandom bits.

Figures 7 and 8 contain necessary plumbing to extend Scala's existing task-pool framework so that every worker thread will keep track of which task it is currently running. The last two methods of the `Par` object in Figure 8 define the `parallel` and `withNewRNG` constructs. Each is defined using parametric types, so that the result type of `parallel` is a tuple of the types of the two argument expressions, and the result type of `withNewRNG` is the type of its body.

Figure 9 declares a type-parametric Scala class `Task[T]` (a task that computes a result of type `T`) that extends the existing Scala class `RecursiveTask[T]` as well as the trait `AnyTask` declared in Figure 8. The method `compute` is invoked by the Scala task-pool infrastructure when the task is executed by a worker thread; it computes the `body` of the task after using `Par.setCurrentTask` to save the current task in the local state of the thread. The method `runUsingForkJoinThread` is called by the `Par.parallel` method declared in Figure 8 to cause this task to be executed by some worker thread for the task pool. The method `next64Bits` is used to compute 64 new pseudorandom bits; it simply computes the next dot product and gives the result to method `Rand.mix64` of Figure 5. The method `nextDotProd` computes a new dot product from its local `dotProd` and `gamma` state variables by using the method

```
import scala.concurrent.forkjoin._
import scala.concurrent.forkjoin.ForkJoinPool._
class TaskForkJoinWorkerThread(group: ForkJoinPool)
          extends ForkJoinWorkerThread(group: ForkJoinPool) { var task: AnyTask = null }
class TaskForkJoinPool(parallelism: Int, factory: TaskForkJoinWorkerThreadFactory)
          extends ForkJoinPool(parallelism: Int, factory: ForkJoinWorkerThreadFactory) {}
class TaskForkJoinWorkerThreadFactory
          extends ForkJoinWorkerThreadFactory {
  def newThread(group: ForkJoinPool) = new TaskForkJoinWorkerThread(group) }
```

**Figure 7.** Scala threads and thread pools extended to hold tasks that contain PRNG state

```
object Par {
 def nprocs =
   Runtime.getRuntime().availableProcessors()
 val taskPool: TaskForkJoinPool =
   new TaskForkJoinPool(nprocs,
       new TaskForkJoinWorkerThreadFactory())
 def getCurrentTask(): AnyTask = {
   Thread.currentThread() match {
     case cur: TaskForkJoinWorkerThread =>
       cur.task
     case _ => throw new ClassCastException } }
 def setCurrentTask(t: AnyTask) = {
   Thread.currentThread() match {
     case cur: TaskForkJoinWorkerThread =>
       cur.task = t
     case _ => } }
 def parallel[T,U](f: =>T, g: =>U): (T,U) = {
   val t: Task[T] =
     Thread.currentThread() match {
       case cur: TaskForkJoinWorkerThread =>
         cur.task.newChild(f)
       case _ => new Task(f, 0, 0L, 0L) }
   t.runUsingForkJoinThread()
   val v: U = g
   (t.join(), v) }
 def withNewRNG[T](seed: Long)(f: =>T): T = {
   val newThread =
     new Task[T](f, 1, Rand.longs(1), seed)
   newThread.runUsingForkJoinThread()
   newThread.join() }
}
```

**Figure 8.** Scala PRNG task utility routines

`Rand.update` of Figure 5; it stores the result back in `dotProd` and then returns it. The method `newChild` makes a new child task, using a depth one greater than its own, the gamma value from the `Rand.longs` table corresponding to that new depth, and a newly computed dot product. Problem: what to do if the depth exceeds the length of the table? In this implementation, we just let it wrap around modulo the length of the table, which in principle disrupts the proof that collisions will be unlikely, but in practice may be acceptable, especially if the table is fairly large (1024 should be plenty for applications that do relatively balanced task splitting).

```
import scala.concurrent.forkjoin.RecursiveTask
class Task[T](body: =>T,
              val depth: Int,
              val gamma: Long,
              var dotProd: Long)
      extends RecursiveTask[T] with AnyTask {
 def compute(): T = {
   Par.setCurrentTask(this)
   body }
 def runUsingForkJoinThread() = {
   if (Thread.currentThread
         .isInstanceOf[ForkJoinWorkerThread])
     this.fork()
   else Par.taskPool.execute(this) }
 override def next64Bits(): Long =
   Rand.mix64(nextDotProd())
 override def nextDotProd(): Long = {
   dotProd = Rand.update(dotProd, gamma)
   dotProd }
 override def newChild[T](x: =>T): Task[T] = {
   val d = (depth + 1) % Rand.longs.length
   new Task(x, d, Rand.longs(d), nextDotProd())
 }
}
```

**Figure 9.** New task type for maintaining PRNG state

Figure 10 declares a Scala class `ParallelApp` that extends the existing Scala class `App`. Instances of `App` simply execute their bodies; `ParallelApp` arranges to make a few extra facilities available during such execution. The methods `parallel` and `withNewRNG` simply forward their arguments to the corresponding methods of object `Par` in Figure 8. The method `nextLong()` uses `Par.getCurrentTask()` to retrieve the current task from the current worker thread and calls its `next64Bits` method; the methods `nextInt()`and `nextDouble()` make use of `nextLong()`. Method `main` is the interface that causes the body of an instance of `App` to be run; the override in class `ParallelApp` causes a new random number generator task to be established first. Method `initialSeed` chooses the initial seed for this root task; it attempts to use environmental information so as to make it likely that distinct processes will compute distinct values for this initialization value. (The details of this rather difficult process do not concern us here.)

```
class ParallelApp extends App {
 def parallel[T,U](x: => T, y: => U): (T,U) =
   Par.parallel(x, y)
 def withNewRNG[T](seed: Long)(f: => T) =
   Par.withNewRNG(seed)(f)
 def nextLong(): Long =
   Par.getCurrentTask()
      .asInstanceOf[AnyTask].next64Bits()
 def nextInt(): Int = nextLong().toInt
 val DOUBLE_ULP: Double = (1.0 / (1L << 53))
 def nextDouble(): Double =
   (nextLong() >>> 11) * DOUBLE_ULP
 override def main(args: Array[String]) =
   withNewRNG(initialSeed())(super.main(args))
 def initialSeed(): Long = ...
}
```

**Figure 10.** Class for declaring parallel Scala applications

### 2.2 A 64-bit Implementation in Java

Next we constructed a version of the algorithm suitable for the Java™ programming language environment [11]. The code for this class, `Splittable64`, is shown in Figures 11 and 12, We made six changes relative to the Scala version:

**[1]** A drawback of tying the PRNG state to tasks is that programs that use parallel tasks must pay the overhead of maintaining the PRNG state even in parts of the program that are not generating pseudorandom values. Also, we hoped to develop an algorithm that might supersede the existing `java.util.Random` facility. With the elimination of parent pointers and explicit pedigrees, there is no real need to keep the task structure involved if we are willing to pass around explicit references to PRNG objects, as is already done in Java with `java.util.Random`. Therefore our Java implementation is object-oriented rather than task-oriented.

**[2]** We then realized that if the *spawn* operation became a method called `split`, PRNG objects could act as collections in the JDK8 library framework, and a PRNG object could easily produce a *spliterator* [25] capable of delivering a stream (of pseudorandom values) that could provide methods such as `map` and `reduce`, and interact with streams produced by other sorts of collections such as lists and hashmaps.

**[3]** We eliminated the use of unsigned compare operations by adopting a change of representation: we regard the dot product (but not the random coefficient) as being offset in its nominal value by `0x8000000000000000`. This allows us to replace calls to the unsigned comparison `unsignedGE` with uses of the Java signed comparison operator `>=`. (Java JDK8 includes a library of unsigned comparison operations, but we do not yet know whether, or how soon, compilers will inline these as single unsigned-compare machine instructions.)

**[4]** A drawback of our Scala implementation is that the table of random coefficients has fixed size. We decided to use pseudorandom coefficients, and to generate them on the fly using a copy of the DOTMIX algorithm that always uses pedigrees of length 1, and therefore needs only one "random coefficient" $\gamma_\gamma$. This gets rid of the big table. (We must still take care to ensure that such generated coefficients are not smaller than 13, so that the `update` method that performs arithmetic mod George will not require a loop.)

**[5]** At this point, speed measurements showed that the `update` method was a large fraction of the time for a *generate* action. We further restricted the range of pseudorandomly generated coefficients so that they are substantially smaller than $2^{64}$: instead of using arithmetic mod George, this calculation is performed modulo Percy $= 2^{56} - 5$ to produce a 56-bit value that is then mixed by a separate mixing method `mix56`. Adding 13 to this result produces a coefficient that is still much smaller than $2^{64}$ but not smaller than 13. The point of using a small coefficient is that the addition in the first line of method `update` will overflow only rarely and therefore the conditional expression nearly always takes the first choice, avoiding further arithmetic. Moreover, it makes that conditional test highly predictable, further increasing speed on processors that do branch prediction.

**[6]** Like `java.util.Random`, `Splittable64` provides two public constructors. One takes a `seed` argument, and the other takes no argument and provides a pseudorandomly generated seed value. For the purpose of generating such default seeds, a third copy of the algorithm is used, again using arithmetic mod George.

Figure 11 declares a Java class `Splittable64`. It uses an `AtomicLong` value called `defaultGen` for generating default seeds for the parameterless constructor. The `update` and `mix64` methods are identical in behavior to the Scala versions in Figure 5, except that the Java version of `update` assumes the offset representation for argument `s` and therefore uses signed comparisons. We have given `nextDotProd` the new name `nextRaw64`, but its purpose is the same: to produce 64 "raw" bits for input to the mixing function. The (private) two-argument constructor takes the usual `seed` argument and also a seed for generating $\gamma$ coefficients; the constructor adds $\gamma_\gamma$ to this second seed (modulo Percy), applies `mix56` to the result, then adds 13 to produce the $\gamma$ coefficient for the new PRNG object. It also saves the updated $\gamma$ seed in its `nextSplit` field for use by its `split` method. Method `nextDefaultSeed` similarly updates the `defaultGen` seed (modulo George), using an atomic `compareAndSet` operation, then applies `mix64` and returns that result.

Figure 12 shows the public methods of `Splittable64`. The constructors are straightforward; each calls the private constructor with appropriate arguments. The `split` method calls `nextRaw64` to produce a new dot product, then creates a new `SplittableRandom` object with that value and the saved value in `nextSplit`. The methods `nextLong`, `nextInt`, and `nextDouble` are similar to their Scala counterparts in Figure 10. Finally, the method `longs` returns a stream of pseudorandom values of the specified length; it does this by calling the Java JDK8 library method `StreamSupport.longStream` and giving it a spliterator (an instance of class `RandomLongs`) as its first argument.

```
public class Splittable64 {
 private static final long
   GAMMA_PRIME = (1L << 56) - 5,    // "Percy"
   GAMMA_GAMMA = 0x00281E2DBA6606F3L,
   DEFAULT_SEED_GAMMA = 0xBD24B73A95FB84D9L;
 private static final double DOUBLE_ULP =
   1.0 / (1L << 53);
 private static final AtomicLong defaultGen =
     new AtomicLong(initialSeed());
 private long seed;
 private final long gamma, nextSplit;
 private static long update(long s, long g) {
   // Add g to s modulo George.
   long p = s + g;
   return (p >= s) ? p
        : (p >= 0x8000000000000000DL) ? p - 13L
        : (p - 13L) + g; }
 private static long mix64(long z) {
   z = (z ^ (z >>> 33)) * 0xff51afd7ed558ccdL;
   z = (z ^ (z >>> 33)) * 0xc4ceb9fe1a85ec53L;
   return z ^ (z >>> 33); }
 private static long mix56(long z) {
   z = ((z ^ (z >>> 33)) * 0xff51afd7ed558ccdL)
     & 0x00FFFFFFFFFFFFFFL;
   z = ((z ^ (z >>> 33)) * 0xc4ceb9fe1a85ec53L)
     & 0x00FFFFFFFFFFFFFFL;
   return z ^ (z >>> 33); }
 private long nextRaw64() {
   return (seed = update(seed, gamma)); }
 private Splittable64(long seed, long s) {
   // We require 0 <= s < Percy
   this.seed = seed;
   s += GAMMA_GAMMA;
   if (s >= GAMMA_PRIME) s -= GAMMA_PRIME;
   this.gamma = mix56(s) + 13;
   this.nextSplit = s; }
 private static long nextDefaultSeed() {
   long p, q;
   do { p = defaultGen.get();
        q = update(p, DEFAULT_SEED_GAMMA);
   } while (!defaultGen.compareAndSet(p, q));
   return mix64(q); }
 // Public methods go here
}
```

**Figure 11.** Class `Splittable64` and its private methods

The method `longs` consists of fairly standard "boilerplate" code for creating a new spliterator-based stream [25] of pseudorandomly chosen `long` values that can potentially be processed in parallel; similarly the method `ints` produces a stream of pseudorandomly chosen `int` values, and `doubles` produces a stream of pseudorandomly chosen `double` values. As a simple example, if `prng` refers to an instance of `SplittableRandom`, then the expression

```
prng.doubles(n).map(x->x*x).sum()
```

```
public Splittable64(long seed) {
  this(seed, 0); }
public Splittable64() {
  this(nextDefaultSeed(), GAMMA_GAMMA); }
public Splittable64 split() {
  return new Splittable64(nextRaw64(),
                          nextSplit); }
public long nextLong() {
  return mix64(nextRaw64()); }
public int nextInt() {
  return (int)nextLong(); }
public double nextDouble() {
  return (nextLong() >>> 11) * DOUBLE_ULP; }
public LongStream longs(long size) {
  if (size < 0L)
    throw new IllegalArgumentException();
  return StreamSupport.longStream
    (new RandomLongs(this, 0L, size),
     false); }
// The next two methods are just like "longs"
public IntStream ints(long size) { ... }
public DoubleStream doubles(...) { ... }
```

**Figure 12.** Public methods of class `Splittable64`

will compute the sum of the squares of `n` pseudorandomly chosen `double` values in the range $[0, 1)$, and the expression

```
prng.doubles(n).parallel().map(x->x*x).sum()
```

will do so using parallel threads if available. The instance of `RandomLongs` created by the `longs` method can supply (to a given *consumer* of `long` values) a sequence of pseudorandomly chosen `long` values.

For completeness, we exhibit in Figure 13 a slightly simplified version of the code for the spliterator `RandomLongs`; its job is to apply a given *consumer* of `long` values to a sequence of pseudorandomly chosen `long` values (methods `tryAdvance` and `forEachRemaining`). The `RandomLongs` spliterator, like any other spliterator, must also be prepared to (try to) split itself into two such streams (method `trySplit`) that can then be processed in parallel; when it does so, it also splits the underlying instance of `SplittableRandom` so that each substream will have its own PRNG rather than trying to share the same one. As a result, if the two streams are then processed in parallel, they do not contend for the use of a single PRNG object; each has its own, and they may be used independently. This eliminates the need for any locking or other synchronization in the parallel execution of stream expressions that involve `SplittableRandom` objects. This is a hallmark of the spliterator paradigm: for effective parallel processing, objects are designed not to be *shared among* tasks, but *split across* tasks. (The actual details of parallel execution—thread management and so on—are handled by the Java library code that implements streams and need not concern us here.)

```
package java.util;
import java.util.function.LongConsumer;
import java.util.stream.StreamSupport;
import java.util.stream.LongStream;
static final class RandomLongs
       implements Spliterator.OfLong {
 final SplittableRandom prng;
 long index;  final long fence;
 RandomLongs(SplittableRandom prng,
             long index, long fence) {
   this.prng = prng; this.index = index;
   this.fence = fence; }
 public RandomLongs trySplit() {
   long i = index, m = (i + fence) >>> 1;
   return (m <= i) ? null :
          new RandomLongs(prng.split(), i,
                          index = m); }
 public long estimateSize() {
   return (fence - index); }
 public int characteristics() {
   return (Spliterator.SIZED |
           Spliterator.SUBSIZED |
           Spliterator.NONNULL |
           Spliterator.IMMUTABLE); }
 public boolean tryAdvance(LongConsumer c) {
   if (c == null)
     throw new NullPointerException();
   long i = index, f = fence;
   if (i < f) { c.accept(prng.nextLong());
                index = i + 1; return true; }
   return false; }
 public void
        forEachRemaining(LongConsumer c) {
   if (c == null)
     throw new NullPointerException();
   long i = index, f = fence;
   if (i < f) { index = f;
                do { c.accept(prng.nextLong());
                } while (++i < f); } }
}
```

**Figure 13.** Spliterator for generating pseudorandom 64-bit `long` values (slightly simplified from actual JDK8 code)

We also remark on a subtle aspect of the semantic contract for the `split` method: We do not guarantee that the set of pseudorandom values produced by a pair of PRNG objects, one split from the other, will be exactly the same set of values that would have been produced by a single unsplit PRNG object; we promise only that the two sets will have similar desirable statistical properties (thanks to the LSS proof about the probability of dot-product collisions, backed up by empirical testing). On the other hand, we have this guarantee of reproducibility: if two PRNG objects are created with the same initial state, and then the same series of method calls is performed on those two PRNG objects, and an identical series of method calls is performed on every pair of corresponding PRNG objects recursively produced by calls to the `split` method, then exactly the same pseudorandom values will be produced.

### 2.3 A 128-bit Implementation in Java

The `Splittable64` algorithm has at least two remaining drawbacks. First, the period for any single PRNG object is equal to the number of distinct values it can produce ($2^{64}$), and therefore adjacent values are never, ever the same—indeed, if you generate up to $2^{64}$ values successively, there will be no duplicates—so the behavior is not quite the same as "truly random." (On the other hand, many applications use only a portion of each generated 64-bit value—in particular, the generation of a 64-bit floating-point value uses only 53 bits—and so this drawback is often not serious in practice.) Second, the fact that we reduced the range of random coefficients from $[13, 2^{64})$ to $[13, 2^{56} + 13)$ (in order to gain some speed) undesirably increases the probability of dot-product collisions. One way out is to increase the dot-product size from 64 bits to 128 bits; then we can afford to decrease the $\gamma$ coefficient size from 128 bits to, say, 120 bits and still have plenty of headroom on the probability of dot-product collisions. The idea is straightforward, but the details are a bit messy because Java does not have a primitive 128-bit integer type. This, of course, provides ample opportunity for additional engineering cleverness.

The 128-bit dot-product arithmetic is performed modulo Arthur $= 2^{128} + 51$. The main new idea here is to choose 128-bit $\gamma$ values that have 114 random bits represented in a very specific way: as two 64-bit words, with the high-order word having a value in the range $[0, 2^{54})$ and the low-order word having a value in the range $[51, 2^{60} + 51)$.

To generate 114 random bits, we use a generator of 57-bit results by using arithmetic modulo Ginny $= 2^{57} - 13$ and a mixing function `mix57`, then use two successive such values to construct one $\gamma$ value.

Figure 14 declares a Java class `Splittable128`, similar in spirit and overall organization to `Splittable64`. The value `GAMMA_PRIME` is now Ginny rather than Percy. While we could have chosen to use 128-bit arithmetic for generating default seeds, there really is not much point since their generation is serialized by the atomic synchronization and therefore it is highly unlikely that more than $2^{64}$ or even $2^{57}$ will be needed, so in order to share code we just use arithmetic modulo Ginny for that as well. Therefore arithmetic modulo George is not used at all. The field `seed` is replaced by fields `seedHi` and `seedLo`; similarly the field `gamma` is replaced by fields `gammaHi` and `gammaLo`. Method `mix64` is as in Figure 11. The method `nextRaw64` performs arithmetic modulo Arthur. The point of our choice of $\gamma$ values is that the sum exceeds Arthur only very rarely (about one time in a thousand), and therefore method `seedFixup` (which handles this overflow case) is very rarely called. Furthermore, the fact that at least 3, and very probably 4, of

```
import java.util.concurrent.atomic.AtomicLong;
public class Splittable128 {
 private static final long
   GAMMA_PRIME = (1L << 57) - 13,   // "Ginny"
   GAMMA_GAMMA = 0x00281E2DBA6606F3L,
   DEFAULT_SEED_GAMMA = 0x0124B73A95FB84D9L;
 private static final double DOUBLE_ULP =
     1.0 / (1L << 53);
 private static final AtomicLong defaultGen =
     new AtomicLong(initialSeed());
 private long seedHi, seedLo;
 private final long gammaHi, gammaLo;
 private final long nextSplit;
 private static long mix64(long z) { ... }
 private static long mix57(long z) {
   z = (z ^ (z >>> 33)) * 0xff51afd7ed558ccdL;
   z &= 0x01FFFFFFFFFFFFFFL;
   z = (z ^ (z >>> 33)) * 0xc4ceb9fe1a85ec53L;
   z &= 0x01FFFFFFFFFFFFFFL;
   return z ^ (z >>> 33); }
 private long nextRaw64() {
   // Add gamma to seed modulo Arthur.
   long s = seedLo, h = seedHi, gh = gammaHi;
   seedLo = s + gammaLo;
   if (seedLo < s) ++gh;          // Rare
   seedHi = h + gh;
   if (seedHi < h) seedFixup();   // Very rare
   return (seedHi ^ seedLo); }
 private void seedFixup() {
   if (seedLo >= (0x8000000000000000L + 51)) {
     seedLo -= 51;
   } else if (seedHi != 0x8000000000000000L) {
     --seedHi; seedLo -= 51;
   } else {
     long s = seedLo - 51L;
     seedLo = s + gammaLo;
     if (seedLo < s) ++seedHi;
     seedHi += (gammaHi - 1); } }
 private Splittable128(
       long seedHi, long seedLo, long s) {
   // We require 0 <= s < Ginny
   this.seedHi = seedHi; this.seedLo = seedLo;
   s += GAMMA_GAMMA;
   if (s >= GAMMA_PRIME) s -= GAMMA_PRIME;
   long b = mix57(s);
   this.gammaHi = b >>> 3;
   long extraBits = (b << 61) >>> 4;
   s += GAMMA_GAMMA;
   if (s >= GAMMA_PRIME) s -= GAMMA_PRIME;
   this.gammaLo = (extraBits | mix57(s)) + 51;
   nextSplit = s; }
 // Other methods go here
}
```

**Figure 14.** Class `Splittable128` and its private methods

```
private static long nextDefaultSeed() {
  long p, q;
  do { p = defaultGen.get();
       q = p + GAMMA_GAMMA;
       if (q >= GAMMA_PRIME) q -= GAMMA_PRIME;
  } while (!defaultGen.compareAndSet(p, q));
  return mix57(q); }
public Splittable128(long seedLo) {
  this(0, seedLo, 0); }
public Splittable128(long hi, long lo) {
  this(hi, lo, 0); }
public Splittable128() {
  this(nextDefaultSeed(), 0, GAMMA_GAMMA); }
public Splittable128 split() {
  nextRaw64();
  return new Splittable128(
               seedHi, seedLo, nextSplit); }
// Methods nextLong, nextInt, nextDouble,
// longs, ints, and doubles are the same
// as in the 64-bit Java implementation.
```

**Figure 15.** Other methods of class `Splittable128`

the high-order bits of `gammaLo` are 0-bits means that the incrementation of `gh` in method `nextRaw64` (reflecting a carry from the low half to the high half) is rare (about one time in 16), which makes the incrementation rare and the conditional governing it highly predictable. Note that the method `nextRaw64` updates a 128-bit seed, but then returns a 64-bit value by returning the bitwise XOR of the two halves of the seed. The rationale for this is that because `mix64` has good first-order avalanche statistics, `mix64` of this XOR will also have good first-order avalanche statistics (though not second-order). The two-argument constructor `Splittable128` is analogous to the two-argument constructor `Splittable64` in Figure 11 but performs calculations modulo Ginny to generate two 57-bit values and then uses them to construct one $\gamma$ value as described above.

Figure 15 shows method `nextDefaultSeed` and the public methods of `Splittable128`. The constructors are straightforward; each calls the private constructor with appropriate arguments. Note that a two-argument public constructor is provided to allow the user to specify a 128-bit seed if desired. The `split` method calls `nextRaw64` to produce a new dot product (discarding the 64 bits it returns), then creates a new `SplittableRandom` object with the newly updated 128-bit seed value in `seedHi` and `seedLo` and the saved `nextSplit` value (we think this implementation semi-elegant in its semi-simplicity).

We explored the alternative of using 127-bit arithmetic (modulo Molly $= 2^{127} + 29$), with a dot-product representation that does not use the high-order bit of the low-order word, and $\gamma$ values with 117 random bits (54 in the high-order word and 63 in the low-order word). Carry propagation from the low-order half to the high-order half can be performed without branch instructions by adding the two low-

order 63-bit halves, then shifting their 64-bit sum rightward 63 positions to produce the necessary carry bit. In our experiments this failed to produce a speed improvement.

Of course, if this algorithm were to be coded directly in assembly language, one might code a 128-bit integer addition as a 64-bit integer add instruction followed by a 64-bit integer add-with-carry instruction, avoiding the need for these tricks. Nevertheless, method `nextRaw64` is an interesting example of a situation where code can be made substantially and usefully faster not by rewriting the code but by adjusting the data that is supplied.

### 2.4 The SPLITMIX Algorithm

One doubt remained about `Splittable128`: while the probability of a dot-product collision is small, the consequences are extremely undesirable: if two PRNG objects happen to produce the same dot-product value and also have the same $\gamma$ coefficient, then from that point they will produce duplicate streams of values—and if new PRNG objects are then produced by balanced recursive splitting to some uniform depth, all the PRNG objects will have the same $\gamma$ coefficient.

But what if every PRNG object could have a different $\gamma$ coefficient? It would be as if the length of every pedigree were equal to the total number of PRNG objects, and each object corresponds to one element of the pedigree (and increments only that element). Even if two PRNG objects happened to produce the same dot-product value (and therefore emit the same pseudorandom value), the next values they generated would be different, and so the problem of duplicate subsequences would be avoided. This situation can be accomplished with the algorithms we have already presented by doing a long, linear chain of splits rather than a binary (or $n$-ary) tree of splits. Unfortunately, that can take a long time, and may not match the structure of the user code. In a parallel computation, it's difficult to ensure that all $\gamma$ values are different without using a shared data structure that may require synchronized access—just what we hope to avoid.

But then we realized that perhaps it's good enough for different PRNG objects to have distinct $\gamma$ values *with high probability*, and this can be achieved if they are chosen pseudorandomly. Furthermore, our intuition was that it's okay in practice for there to be *occasional* (that is, *rare*) collisions of $\gamma$ values, because two PRNG object with the same $\gamma$ value are likely to have distinct seed values, and in fact seed values that are distant from each other within the periodic cycle of seed values generated by that $\gamma$ value. (We must, of course, take the Birthday Paradox into account when calculating how rare is rare enough.)

Now, the principal reason for doing arithmetic modulo a prime number was to support the string LSS proof that dot-product collisions are rare. If we are no longer worried about dot-product collisions (thanks to the high probability that $\gamma$ values are different), then maybe arithmetic modulo a power of 2 suffices (a weaker version of the LSS proof still applies, but with a bound on the probability of dot-product collision that is about two orders of magnitude larger than for the prime-modulus case). Using arithmetic modulo a power of 2 makes it unnecessary to restrict $\gamma$ values to a range such as $[13, 2^{64})$, because the speed tricks for doing arithmetic modulo a prime number are no longer relevant.

All these considerations led us to the SPLITMIX strategy: to create a new PRNG, simply use an existing PRNG instance to generate a new seed and a new $\gamma$ value pseudorandomly. In practice, we tweak this idea slightly. In the next section we present the specific 64-bit implementation of this strategy that has been incorporated into Java JDK8.

## 3. The Class `SplittableRandom`

Figure 16 declares a Java class `SplittableRandom` that uses 64-bit arithmetic to generate a pseudorandom sequence of 64-bit values (from which 32-bit `int` or 64-bit `double` values may then be derived). It has two 64-bit fields `seed` and `gamma`; for any given instance of `SplittableRandom`, the `seed` value is mutable and the `gamma` value is unchanging. To ensure that every instance has period $2^{64}$, it is necessary to require that the `gamma` value always be odd; thus each instance actually has 127 bits of state, not 128. The private constructor is trivial, taking two arguments and using them to initialize the `seed` and `gamma` fields.

The method `nextSeed` simply adds `gamma` into `seed` and returns the result. (How it could be any simpler?)

The methods `mix64`, `mix32`, `mix64variant13`, and `mixGamma` "mix" (that is, "scramble and blend") the bits of a 64-bit argument to produce a result. Each of the first three computes a bijective function on 64-bit values; `mix32` furthermore discards 32 bits to produce a 32-bit result. The method `mix32` is a simply a version of `mix64` optimized for the case where only the high-order 32 bits of the result will be used (there is no need to use an XOR to update the low-order bits, only to discard them). The method `mix64variant13` is David Stafford's Mix13 variant of the MurmurHash3 finalizer [33].

The value `DOUBLE_ULP` is the positive difference between `1.0` and the smallest `double` value larger than `1.0`; it is used for deriving a `double` value from a 64-bit `long` value.

A predefined `gamma` value is needed for initializing "root" instances of `SplittableRandom` (that is, instances not produced by splitting an already existing instance). We chose the odd integer closest to $2^{64}/\phi$, where $\phi = (1 + \sqrt{5})/2$ is the golden ratio, and call it `GOLDEN_GAMMA`.

A globally shared, atomically accessed `seed` value is also needed for creating root instances; we call it `defaultGen`. It is initialized by procedure `initialSeed`.

Figure 17 contains declarations of public methods for class `SplittableRandom`. The two constructors are simple: if a seed is provided, it is used along with `GOLDEN_GAMMA`, but if a seed is not provided, then the shared `defaultGen` variable is updated once but used to provide two distinct values that are mixed by methods `mix64` and `mixGamma` to produce `seed` and `gamma` values for the new instance.

```
package java.util;
import java.util.concurrent.atomic.AtomicLong;
import java.util.stream.LongStream;
import java.util.stream.IntStream;
import java.util.stream.DoubleStream;
public final class SplittableRandom {
 private long seed;
 private final long gamma;   // An odd integer
 private SplittableRandom(long seed,
                          long gamma) {
   // Note that "gamma" should always be odd.
   this.seed = seed; this.gamma = gamma; }
 private long nextSeed() {
   return (seed += gamma); }
 private static long mix64(long z) {
   z = (z ^ (z >>> 33)) * 0xff51afd7ed558ccdL;
   z = (z ^ (z >>> 33)) * 0xc4ceb9fe1a85ec53L;
   return z ^ (z >>> 33); }
 private static int mix32(long z) {
   z = (z ^ (z >>> 33)) * 0xff51afd7ed558ccdL;
   z = (z ^ (z >>> 33)) * 0xc4ceb9fe1a85ec53L;
   return (int)(z >>> 32); }
 private static long mix64variant13(long z) {
   z = (z ^ (z >>> 30)) * 0xbf58476d1ce4e5b9L;
   z = (z ^ (z >>> 27)) * 0x94d049bb133111ebL;
   return z ^ (z >>> 31); }
 private static long mixGamma(long z) {
   z = mix64variant13(z) | 1L;
   int n = Long.bitCount(z ^ (z >>> 1));
   if (n >= 24) z ^= 0xaaaaaaaaaaaaaaaaL;
   return z; }   // This result is always odd.
 private static final double
   DOUBLE_ULP = 1.0 / (1L << 53);
 private static final long
   GOLDEN_GAMMA = 0x9e3779b97f4a7c15L;  // odd
 private static final AtomicLong defaultGen =
   new AtomicLong(initialSeed());
 // Public methods go here
}
```

**Figure 16.** Private methods of class `SplittableRandom`

The pseudorandom generation of values is then straightforward. To generate a new 64-bit `long` value, simply compute the next seed and feed it to `mix64`; to generate a new 32-bit `int` value, feed the next seed to `mix32`. To generate a new `double` value, generate a 64-bit `long` value, discard 11 bits, and multiply the remaining 53 bits by `DOUBLE_ULP`, thus producing a value chosen uniformly from the range $[0, 1)$.

The `split` method is equally straightforward; in effect, it creates a new instance of `SplittableRandom` by simply choosing its `seed` and `gamma` values "randomly"; to be specific, it generates two new seed values and mixes them using methods `mix64` and `mixGamma`.

Not shown in Figure 17 but implemented in Java JDK8 are versions of methods `longs`, `ints`, and `doubles` that

```
public SplittableRandom(long seed) { // Okay
  this(seed, GOLDEN_GAMMA); }
public SplittableRandom() {     // Preferred
  long s = defaultGen.getAndAdd(
                          2 * GOLDEN_GAMMA);
  this.seed = mix64(s);
  this.gamma = mixGamma(s + GOLDEN_GAMMA); }
public long nextLong() {
  return mix64(nextSeed()); }
public int nextInt() {
  return mix32(nextSeed()); }
public double nextDouble() {
  return (nextLong() >>> 11) * DOUBLE_ULP; }
public SplittableRandom split() {
  return new SplittableRandom(
    mix64(nextSeed()),
    mixGamma(nextSeed())); }
public LongStream longs(long size) { ... }
public IntStream ints(long size) { ... }
public DoubleStream doubles(...) { ... }
// Other convenience methods as well
```

**Figure 17.** Public methods of class `SplittableRandom`

take no argument and produce an indefinitely long stream rather than a stream of a prespecified length, as well as a variety of convenience methods that allow the user to specify ranges from which pseudorandom values should be uniformly chosen. For example, `prng.nextInt(0, 6)` returns a value chosen uniformly from the set $\{0, 1, 2, 3, 4, 5\}$, and `prng.ints(10000, 0, 6)` returns a stream of 10000 values chosen uniformly and independently from that same set.

While we have chosen not to provide a method for "jumping ahead" [16] in a sequence, to do so would be easy:

```
public void jump(long n) {
  seed += (gamma * n); }
```

and for negative $n$ this also jumps backward.

The mixing function `mix64` is easily inverted:

```
private static long unmix64(long z) {
  z = (z ^ (z >>> 33)) * 0x9cb4b2f8129337dbL;
  z = (z ^ (z >>> 33)) * 0x4f74430c22a54005L;
  return z ^ (z >>> 33); }
```

because

```
0xff51afd7ed558ccdL * 0x4f74430c22a54005L == 1
```

and

```
0xc4ceb9fe1a85ec53L * 0x9cb4b2f8129337dbL == 1
```

(using `long` arithmetic, that is, modulo $2^{64}$); such modular inverses are easy to compute. Note also that the transformation `z = z ^ (z >>> 33)` is self-inverse. Therefore, for any 64-bit value `q`, `unmix64(mix64(q)) == q` and `mix64(unmix64(q)) == q`. If `p` and `q` are values produced by consecutive calls to the `nextLong` method of a specific instance of `SplittableRandom` (with no intervening calls to other methods such as `split`), then the current value of

seed for that instance must be unmix64(q), and the gamma value for that instance must be unmix64(q)-unmix64(p). From these values, all future behavior of the instance can be predicted. Therefore SplittableRandom and other members of the SPLITMIX family of PRNG algorithms should not be used for applications (such as security and cryptography) where unpredictability is an important characteristic.

The method mixGamma takes an input, mixes its bits using mix64variant13, and then forces the low bit to be 1. At first we thought that would suffice, but then we tested PRNG objects with "sparse" $\gamma$ values whose representations have either very few 1-bits or very few 0-bits, and found that such cases produce pseudorandom sequences that DieHarder regards as "weak" just a little more often than usual. A little thought shows that $\gamma$ values of the form $000\ldots0011\ldots111$ and $111\ldots1100\ldots000$ could also perform poorly; our informal explanation is as follows. Any $\gamma$ value would be fine if the mixing function were perfect; but for a less-than-perfect mixing function, the more bits change from one seed to the next, the more effective is avalanching in producing apparently random sequences—so we can "help" the mixing function by ensuring that the nextSeed method changes many bits in the seed. Long runs of 0-bits or of 1-bits in the $\gamma$ value do not cause bits of the seed to flip; an approximate proxy for how many bits of the seed will flip might be the number of bit pairs of the form 01 or 10 in the candidate $\gamma$ value z. Therefore we require that the number of such pairs, as computed by Long.bitCount(z ^ (z >>> 1)), exceed 24; if it does not, then the candidate z is replaced by the XOR of z and 0xaaaaaaaaaaaaaaaaL, a constant chosen so that (a) the low bit of z remains 1, and (b) every bit pair of the form 00 or 11 becomes either 01 or 10, and likewise every bit pair of the form 01 or 10 becomes either 00 or 11, so the new value necessarily has more than 24 bit pairs whose bits differ. Testing shows that this trick appears to be effective. The threshold value 24 was chosen somewhat arbitrarily as being large enough to ensure good bit-flipping in successive seeds but small enough that the correction is required relatively rarely, for $\left(\sum_{k=0}^{23}\binom{63}{k}\right)/2^{63} \approx 2.15\%$ of all candidates. (Our first implementation simply rejected such candidates using a do–while loop; we changed it to the XOR method purely to avoid using a loop. While the loop would perform a second iteration only rarely, we prefer loopless algorithms for parallel implementation on SIMD or GPU archtectures. For the same reason we prefer to avoid conditionals, but this case can be handled well using a conditional move instruction.) At most four distinct 64-bit inputs to the method mixGamma will be mapped to the same final $\gamma$ value.

If the calls to methods nextSeed and mix64 are inlined, then the body of the method nextLong is straight-line code consisting of a read of gamma, a read and a write of seed, and 9 64-bit arithmetic/logical operations (2 multiplies, 1 add, 3 shifts, and 3 XOR operations). Likewise, aside from the allocation of a new SplittableRandom object, the work performed by method split is straight-line code if the if statement in method mixGamma is compiled using a conditional move instruction. Suppose, then, that one has a SIMD architecture with $n$-way parallelism ($n$ might be 4 for Intel SSE, 32 for a modern GPU, or 65536 for an old-fashioned Connection Machine CM-2 [34]). A special SIMD version of the split method can start with one PRNG object, put its seed and gamma values into the first elements of two arrays of length $n$, and then use $\lceil \log_2 n \rceil$ doubling steps to perform the split computation on each seed/gamma pair already in the arrays, thus doubling the number of such pairs in the arrays. This initialization can be done just once and the resulting seed/gamma pairs used many times. After that, a SIMD version of method nextLong can process all $n$ seed/gamma pairs in parallel using width-$n$ SIMD instructions on the two arrays. (Alternatively, one can give each thread a copy of the *same* PRNG state, then have thread $k$ jump forward $2k$ states and then perform a split operation to obtain its own PRNG; this works because the jump operation is also amenable to SIMD implementation.)

## 4. Some Remarks about Avalanche Statistics

The idea behind the avalanche effect is that a good bit-mixing function will have the property that changing a single bit of the input will likely cause many (about half) of the output bits to change. Feistel [8] first used the term "avalanche," and Webster and Tavares [36] formulated the Strict Avalanche Criterion: whenever a single input bit is flipped, each of the output bits should flip with probability (approximately) $\frac{1}{2}$, averaged over all possible inputs. Measurement of this criterion can be approximated by testing only a fraction of all possible inputs.

Appleby used this criterion as the "energy function" in a simulated annealing process to explore a specific space of potential 8-operation bit-mixing functions (both 32-bit and 64-bit versions). The results are known as the MurmurHash3 finalizers [2]. Stafford remarked that a non-random set of inputs might make a better training set, did his own extensive simulated-annealing experiments, and reported a set of fourteen 64-bit variants that have better avalanche statistics than the MurmurHash3 64-bit finalizer. Neither Appleby nor Stafford described the precise energy function used to drive the simulated annealing process. However, Stafford reported both maximum error and mean error from the avalanche matrix for each of his variants, where "error" is presumably deviation from the "perfect" probability of $\frac{1}{2}$.

We have conducted our own searches for good 64-bit bit-mixing functions. We tested each candidate $\mu$ using a set of 64-bit input values chosen at random (and using several different PRNG algorithms for this purpose, including SPLITMIX and java.util.Random). From this set of $N$ input values, the $64 \times 64$ avalanche matrix $A$ was constructed as a histogram: for each input value $v$ and for every $0 \le i < 64$ and $0 \le j < 64$, $a_{ij}$ is incremented if and only if $(\mu(v) \oplus \mu(v \oplus 2^i)) \wedge 2^j$ is nonzero, where $\oplus$ is bitwise XOR and $\wedge$ is bitwise AND on binary integers. The absolute value

```
long[][] A = new long[64][64];
for (long n = 0; n < N; n++) {
  long v = rng.nextLong(), w = mix64(v);
  for (int i = 0; i < 64; i++) {
    long x = w ^ m.mix(v ^ (1 << i));
    for (int j = 0; j < 64; j++) {
      if (((x >>> j) & 1) != 0) A[i][j] += 1;
    } } }
double sumsq = 0.0;
for (int i = 0; i < 64; i++) {
  for (int j = 0; j < 64; j++) {
    double v = a[j][k] - N / 2.0;
    sumsq += v*v; } }
double result = sumsq / ((N / 4.0) * 4096.0);
```

**Figure 18.** Calculating sum-of-squares avalanche statistic

of the difference of each histogram entry and $\frac{N}{2}$, normalized by dividing by $N$, represents avalanche "error" (deviation from the desired 50% probability of flipping).

A compromise for minimizing both maximum and mean is to minimize the root-mean-square (equivalently, the sum-of-squares). For our simulated-annealing searches, we let a state be a tuple of multipliers and shift distances; we let the neighbors of a state be a copy with either (a) just one bit of one multiplier flipped, or (b) just one shift distance increased or decreased by 1; and we used the sum-of-squares of the avalanche matrix entries as the energy function. This proved effective up to a certain point, beyond which additional computational effort resulted in no further improvement—we were unable to drive the energy any closer to zero.

A little thought reveals why. The avalanche histogram is a set of $64 \times 64 = 4096$ variables, each of which is, in effect, the sum of $N$ values selected from the set $\{0, 1\}$. If the mixing function is "good" (that is, behaving as if each output bit were "perfectly random") then each variable would be the sum of $n$ "coin flips"—in other words, a random variable chosen from a binomial distribution. For large $N$, this binomial distribution can be approximated as a normal (Gaussian) distribution with mean $\frac{N}{2}$ and variance $\frac{N}{4}$. The sum-of-squares of those 4096 variables is therefore (approximately) a chi-squared distribution with 4096 degrees of freedom. The Java code in Figure 18 performs this calculation, normalizing with respect to both $N$ and the number of matrix entries 4096 so that the result has expected mean 1. The energies of the mixing functions that we bottomed out on were right around that expected value for the mean every time; we also measured this avalanche statistic for MurmurHash3 (with $N = 10^6$) and got 1.0502 on one run, and had similar results for the Stafford variants. So there is every reason to believe that if we had found a mixing function with a much lower energy than that, it would be *worse*, in that its behavior would differ in a statistically noticeable way from truly random behavior. We conclude that the MurmurHash3 and Stafford mixing functions are quite good by this measure, and one can do perhaps only a tiny bit better.

## 5. Measurements of Quality

We used the DieHarder test suite version 3.31.1 [5] to test the algorithm used in class `SplittableRandom`. For this purpose, we recoded the algorithms into C so that they could be linked directly with DieHarder, rather than using the file I/O interface (which is considerably slower). DieHarder uses 32-bit inputs, so the 64-bit outputs from our algorithms are broken into two halves, supplying first the low-order half, then the high-order half, to the single input stream of 32-bit values. We used only the switches `-a` (selecting all tests) and `-g` $n$ (to select the generator algorithm). DieHarder reports any $p$-value outside the range $[10^{-6}, 1 - 10^{-6}]$ as indicating that the tested generator has " failed." DieHarder also flags all $p$-values outside the range $[5 \times 10^{-3}, 1 - 5 \times 10^{-3}]$; such values, if they are not clear failures, are reported as "weak." For each generator variant we ran DieHarder 7 times and we report the total number of weak and failed values over all 7 runs, *omitting* results for tests `opso`, `oqso`, `dna`, and `sums`, each of which the DieHarder software itself describes as "Suspect" or "Do Not Use"; thus each run of DieHarder performs 110 distinct tests, so 7 runs collectively perform 770 tests). (LSS used Dieharder version 2.28.1 [31], so the number—and quality—of tests shown here is slightly different from what they report.)

We also used the BigCrush test of TestU01 [18, 32], again recoding into C for direct linking. TestU01 expects to test 64-bit `double` values, so we use 53 out of every 64 bits generated by our algorithm to generate a `double` value as for our method `nextDouble`. TestU01 regards any $p$-value outside the range $[10^{-10}, 1 - 10^{-10}]$ as indicating "clear failure" [18] of the tested generator. The TestU01 software flags all $p$-values outside the range $[10^{-3}, 1 - 10^{-3}]$; such values, if they are not clear failures, are regarded as "suspect." For each generator variant we ran BigCrush 3 times and we report the total number of suspect $p$-values and clear failures over all 3 runs; each run of BigCrush performs 160 distinct tests, so 3 runs collectively perform 480 tests. With 480 tests at two-sided $p = 0.001$, we expect even a perfect generator to exhibit 0.96 "suspect" values on average.

We tested sequential use of the `generate` operation for a single PRNG object using 16 different $\gamma$ values, 8 of which have few `01` or `10` bit pairs and 8 of which have many. For comparison, we also used DieHarder to test 9 of the "standard" algorithms supplied with DieHarder, including the venerable Mersenne Twister [23]. Summary results are shown in Table 1. These results show "poor" $\gamma$ values having slightly more "weak" results than others, but not consistently so; the average is only slightly higher. We believe that more investigation is required. The average number of "weak" scores over all 16 gamma values tested is 15.3, which is close to the number of "weak" scores for such excellent algorithms as `R_mersenne_twister` and `AES_OFB`.

To test the behavior of split PRNG objects, we used three strategies: (a) Form a balanced binary tree of splits to create

| | $\gamma$ value | DieHarder weak (of 770) | DieHarder failed (of 770) | TestU01 suspect (of 480) | TestU01 failure (of 480) |
|---|---|---|---|---|---|
| "poor" | 0x0000000000000001L | 19 | 0 | 0 | 0 |
| | 0x0000000000000003L | 18 | 0 | 2 | 0 |
| | 0x0000000000000005L | 12 | 0 | 1 | 0 |
| | 0x0000000000000009L | 19 | 0 | 0 | 0 |
| | 0x0000010000000001L | 14 | 0 | 0 | 0 |
| | 0xffffffffffffffffL | 12 | 1 | 0 | 0 |
| | 0x000000000ffffffL | 15 | 0 | 2 | 0 |
| | 0xffffff0000000001L | 15 | 0 | 2 | 0 |
| "okay" | 0x0000000000555555L | 11 | 0 | 1 | 0 |
| | 0x1111111111110001L | 17 | 0 | 1 | 0 |
| | 0x7777777777770001L | 15 | 1 | 1 | 0 |
| | 0x7f7f7f7f33333333L | 12 | 0 | 3 | 0 |
| | 0x5555550000000001L | 16 | 0 | 2 | 0 |
| random | 0xc45a11730cc8ffe3L | 16 | 0 | 2 | 0 |
| | 0x2b13b77d0b289bbdL | 15 | 0 | 1 | 0 |
| | 0x40ead42ca1cd0131L | 19 | 0 | 0 | 0 |
| | AES_OFB | 18 | 0 | — | |
| | R_knuth_taocp | 13 | 0 | — | |
| | R_marsaglia_multic. | 13 | 28 | — | |
| | R_mersenne_twister | 15 | 0 | — | |
| | R_super_duper | 16 | 19 | — | |
| | R_wichmann_hill | 19 | 0 | — | |
| | Threefish_OFB | 24 | 0 | — | |
| | kiss | 17 | 0 | — | |
| | superkiss | 13 | 14 | — | |

**Table 1.** Test results for $\gamma$ values used sequentially

$2^k$ PRNG objects, then round-robin interleave their streams; (b) start with one object, use it to generate one number, then replace it with a split-off object; (c) start with one object, split off a second object, then use the first object to generate one number, then replace the first object with the second. The results are shown in Table 2.

For comparison, we did three runs of TestU01 BigCrush on `java.util.Random`; 19 tests produced clear failure on all three runs. These included 9 Birthday Spacings tests, 8 ClosePairs tests, a WeightDistrib test, and a CouponCollector test. This confirms L'Ecuyer's observation that `java.util.Random` tends to fail Birthday Spacings tests [17].

In early experiments, we tested algorithms `ParallelApp`, `Splittable64`, and `Splittable128` (Sections 2.1–2.3) using DieHarder, with results similar to those in Table 1. We have not yet tried TestU01 on these algorithms.

## 6. Measurements of Speed

To evaluate the performance consequences of introducing class `SplittableRandom` for Java JDK8, we compared it against two existing alternatives in the Java core library. The class `java.util.Random` uses the UNIX `rand48` algorithm and is thread-safe, serializing generation from a common shared generator when accessed by multiple threads. Class `java.util.concurrent.ThreadLocalRandom`, introduced in JDK7, also uses `rand48`, but lazily creates a new generator per thread using Java's `ThreadLocal` facilities.

| | number of PRNGs | DieHarder weak (of 770) | DieHarder failed (of 770) | TestU01 suspect (of 480) | TestU01 failure (of 480) |
|---|---|---|---|---|---|
| binary tree of splits | 2 | 17 | 0 | 0 | 0 |
| | 4 | 9 | 0 | 2 | 0 |
| | 8 | 20 | 0 | 0 | 0 |
| | 16 | 14 | 0 | 0 | 0 |
| | 32 | 13 | 0 | 1 | 0 |
| | 64 | 19 | 0 | 3 | 0 |
| | 128 | 18 | 0 | 1 | 0 |
| | 256 | 15 | 0 | 1 | 0 |
| chained splits | generate then split | 17 | 0 | 1 | 0 |
| | split then generate | 20 | 1 | 1 | 0 |

**Table 2.** Test results for split PRNG objects used in parallel

Comparisons are shown for two representative computer systems, with recent 64-bit processors, both with 32 effective cores spread over multiple sockets (four Intel i7 X7560 chips vs. two AMD 6278 chips; the Intel system is hyperthreaded to allow 64 hardware threads, but experiments used a maximum of 32 threads). Both run Linux 2.6.39 kernels. Measurements of `ThreadLocalRandom` ("TLR7") used JDK7; measurements of `java.util.Random` ("JUR8") and `SplittableRandom` ("SR8") used OpenJDK JDK8 early access build 103. All entries represent the median of five runs using loops with $2^{26}$ iterations, following a warmup period of 15 shorter runs.

Table 3 compares sequential performance for a simple loop that sums `nextLong`, `nextDouble`, or `nextInt` results. Throughput for `SplittableRandom` is faster (by factors ranging from 1.22 to 8.29) than either alternative, except for being 15%–20% slower than `ThreadLocalRandom` when generating 32-bit values (`nextInt`), the only measured case for which the use of cheaper (32-bit) arithmetic instructions in the `rand48` algorithm leads to a performance advantage.

Figure 19 compares parallel performance using the Java JDK8 `Stream` library to compute sums; for example, to compute the sum of `n` values of type `long` from a pseudo-random number generator `prng`, we used this expression:

```
prng.longs(n).parallel().sum()
```

The `Stream` library subdivides expressions like this into multithreaded subcomputations using the `ForkJoin` framework. In the case of `SplittableRandom`, each subdivision is associated with a split-off generator, while for `java.util.Random`, which has no `split` method, all threads use the same generator. Such a stream expression can also be computed sequentially by omitting the call to the `parallel` method: `prng.longs(n).sum()`. Using `SplittableRandom` produces performance that scales approximately linearly with the number of threads with a slope of roughly $\frac{5}{8}$ (approximately 17× on the AMD machine and 22× on the Intel machine when using 32 threads).

| | Millions of generated values per second | | | | | | Speedup of SR8 over the others | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | AMD 6278 | | | Intel X7560 | | | AMD 6378 | | Intel X7560 | |
| | JUR8 | TLR7 | SR8 | JUR8 | TLR7 | SR8 | JUR8 | TLR7 | JUR8 | TLR7 |
| Long | 29.8 | 143.8 | 247.1 | 33.2 | 175.9 | 274.1 | ×8.29 | ×1.71 | ×8.25 | ×1.55 |
| Double | 28.8 | 149.2 | 189.5 | 33.1 | 176.4 | 215.7 | ×6.57 | ×1.27 | ×6.51 | ×1.22 |
| Int | 57.5 | 302.7 | 248.4 | 65.3 | 288.2 | 250.2 | ×4.32 | ×0.82 | ×3.83 | ×0.86 |

**Table 3.** Sequential throughput (using a sequential loop that sums generated pseudorandom values)

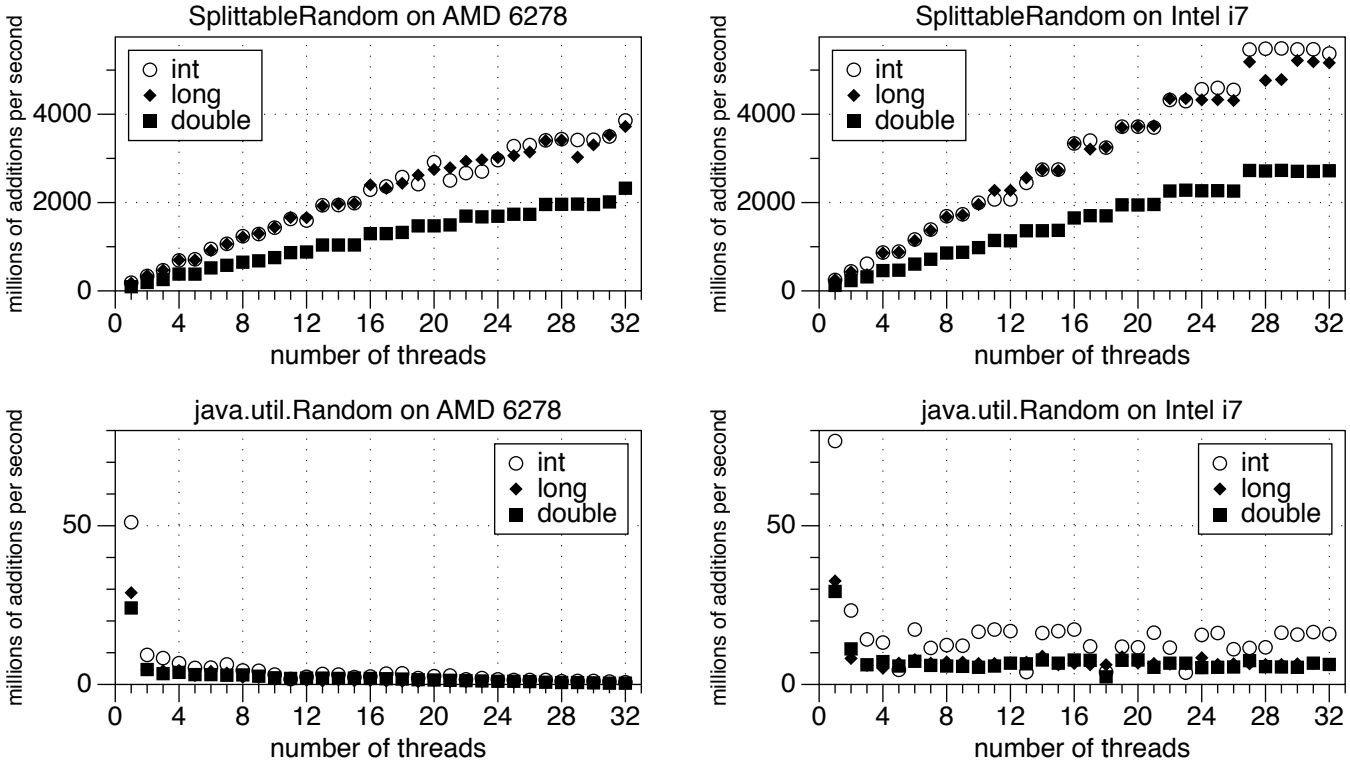


**Figure 19.** Parallel stream throughput (1 to 32 threads, millions of generated values per second, median of three runs)

On the other hand, when using `java.util.Random`, throughput *drops* precipitously (by a factor of 4 or more) when going from 1 thread to 2 threads, and adding more threads beyond that only makes things worse. Because a single synchronized generator is being shared among all threads, we would not expect any speedup on the generation of pseudorandom numbers; we might have hoped that other parts of the application might enjoy some parallel speedup. But the measurements tell a different story; while we have not done a detailed performance analysis, it appears that contention among threads for the shared generator (possibly compounded by poor memory system behavior as ownership of the generator state is transferred among threads) swamps any gains from parallel execution. The obvious way to mitigate synchronization overhead is for each thread to have its own PRNG state rather than trying to share state, while still maintaining good statistical quality—which `java.util.Random` was never designed to do.

Figure 20 shows measurements of parallel scaling of a Monte Carlo algorithm using `SplittableRandom`.

We also measured the performance difference between a purely sequential version of our benchmarks and the parallel version using just one thread. Performance is roughly the same: for the calculation of $\pi$, the sequential version was 1% slower on AMD 6278 and 1% faster on Intel X7560 when using `SplittableRandom`. We observed a reversed effect when using `java.util.Random`: the sequential version was 2% *faster* on AMD 6278 and 2% *slower* on Intel X7560. For stream summation, the two versions sometimes differed in execution time by as much as 16%, but we also observed similar (unexplained) run-to-run variation when comparing runs of each version separately.

LSS do not report much about absolute performance of DOTMIX, but they do remark that, for one benchmark, using DOTMIX was about 2.3 times as slow as using Mersenne Twister [20, §6]. Sean Luke reports that his optimized Java implementation of Mersenne Twister [21] is about $\frac{1}{3}$ faster than using `java.util.Random`, and our Table 3 indicates that using `SplittableRandom` is about 4 to 8 times as fast as using `java.util.Random`. These data imply that

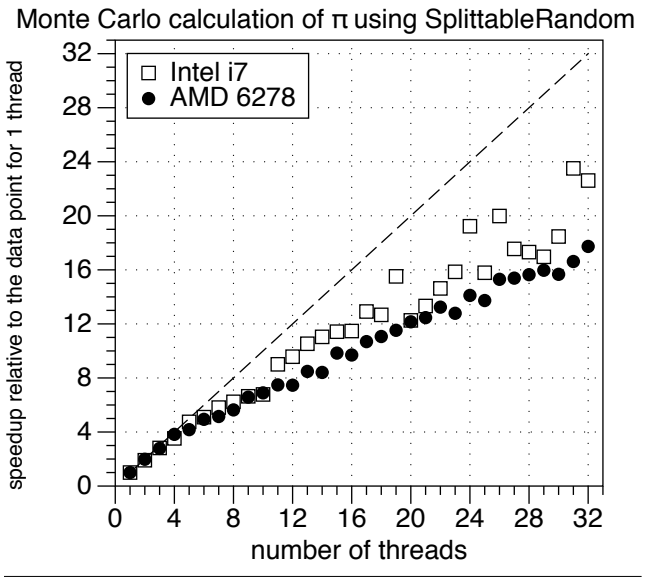


**Figure 20.** Parallel scaling for calculation of $\pi$

`SplittableRandom` is roughly 6 to 12 times as fast as DOTMIX. Given the need for DOTMIX to chase pointers among stack frames and to perform multiplications modulo a prime number, this estimate seems plausible.

## 7. Other Related Work

L'Ecuyer [15, Figure 3] describes a way to create a PRNG of good quality and very long period by combining two or more multiplicative linear congruential generators, and remarks that the period of the combined generator can be "split" (partitioned) into separate sections easily because each of the underlying generators can be so partitioned.

Park and Miller [26], surveying the situation in 1988, demonstrated that many PRNG algorithms then in use were of terrible quality, and recommended a specific prime-modulus multiplicative linear-congruential ("Lehmer") generator $z_{n+1} = 16807 z_n \bmod (2^{31} - 1)$, and challenged future PRNG designers to do at least as well. They noted that the multiplier 16807, while not necessarily the absolute best choice, was quite good and furthermore small enough to permit certain implementation tricks to improve speed.

Burton and Page [6] describe partitioning the period of a sequential PRNG in three ways: alternation (elsewhere called "leapfrogging" or "lagging"), halving ("jumping ahead" by a fixed amount, typically to cut the stream in half), and jumping to a random point in the cycle. In effect, our `split` method takes this third approach after randomly choosing among a large number of possible cycles.

In the mid-1990s, Augustsson [3] implemented L'Ecuyer's algorithm in purely functional form as part of the Haskell standard library `System.Random`; the code now in that library, dated 2001, contains a kernel (Figure 21) with two functions `stdNext` and `stdSplit`. The implementation of `stdNext` is a faithful rendition of L'Ecuyer's algorithm [15, Figure 3], but the `stdSplit` method does not split the period in the manner suggested by L'Ecuyer; instead, it uses

```
stdNext :: StdGen -> (Int, StdGen)
stdNext (StdGen s1 s2) =
 (fromIntegral z', StdGen s1'' s2'') where
   z'   = if z < 1 then z + 2147483562 else z
   z    = s1'' - s2''
   k    = s1 `quot` 53668
   s1'  = 40014 * (s1 - k * 53668) - k * 12211
   s1'' = if s1' < 0 then s1' + 2147483563 else s1'
   k'   = s2 `quot` 52774
   s2'  = 40692 * (s2 - k' * 52774) - k' * 3791
   s2'' = if s2' < 0 then s2' + 2147483399 else s2'
stdSplit :: StdGen -> (StdGen, StdGen)
stdSplit std@(StdGen s1 s2) = (left, right) where
  -- no statistical foundation for this!
  left    = StdGen new_s1 t2
  right   = StdGen t1 new_s2
  new_s1 | s1 == 2147483562 = 1
         | otherwise        = s1 + 1
  new_s2 | s2 == 1          = 2147483398
         | otherwise        = s2 - 1
  StdGen t1 t2 = snd (next std)
```

**Figure 21.** Haskell library `System.Random` kernel (2001)

an *ad hoc* method that, by its own admission, has "no statistical foundation," but is not that different in structure from SPLITMIX except that it fails to try to compute "random" values for initializing the new `StdGen` objects.

Hellekalek [12] pointed out the difficulty of finding good-quality PRNG algorithms for parallel use, and that splitting via "leapfrogging" can produce unexpected surprises.

L'Ecuyer [17] describes severe failures of the algorithm used by `java.util.Random` (as well as the PRNG facilities of Visual Basic and Excel as of 2001 and the "Lehmer 16807" generator recommended by Park and Miller [26]) to pass a "birthday spacings" test. He also comments, "In the Java class `java.util.Random`, RNG streams can be declared and constructed dynamically, without limit on their number. However, no precaution seems to have been taken regarding the independence of these streams."

L'Ecuyer et al. [19] describe an object-oriented C++ PRNG package `RngStream` that supports repeatedly splitting its very long period (approximately $2^{191}$) into streams and substreams. (We discuss this package at length in Section 1.)

Salmon et al. [30] describe a parallel PRNG algorithm with period $2^{128}$ that furthermore relies on a key that allows selection of one of $2^{64}$ independent streams. They remark that their paper "revisits the underutilized design approach of using a trivial $f$ and building complexity into the output function"; their $f$ corresponds to our `nextSeed` method and their output function to our bit-mixing function, so in that respect we follow in their footsteps. They suggest that $f$ might be as simple as $f(x) = x + 1$, and so refer to their methods as "counter-based"; they rely on powerful (perhaps cryptographically strong) output functions. Our approach differs in that we are willing to choose $f$ more carefully in order to reduce the cost of the output (bit-mixing) function. Salmon et al. also make use of the golden ratio and Weyl sequences

to generate "round keys" for AES encryption (though, curiously, they use 64 bits of the golden ratio for the high half of the initial round key and 64 bits of $\sqrt{3} - 1$ for the low half).

Claessen and Pałka [7] remark on an application that exposes a severe defect of the `stdSplit` function in the Haskell standard library `System.Random`, then describe a superior implementation of the same purely functional API that is similar in spirit to LSS: it generates random values by encoding the path in the split tree as a sequence of numbers, and then applies a cryptographically strong hash function. Their path encoding and hash function are designed to allow successive pseudorandom values to be computed incrementally in constant time, independent of the path length. We have not yet made comparative measurements, but it appears that their approach produces pseudorandom streams of higher quality, while our approach is likely much faster.

## 8. Conclusions and Future Work

Inspired by Leiserson, Schardl, and Sukha, we have traced a winding development path that led us to the `SplitMix` family of algorithms for parallel generation of sets and sequences of pseudorandom values. They should not be used for cryptographic or security applications, because they are too predictable (the mixing functions are easily inverted, and two successive outputs suffice to reconstruct the internal state), but because they pass standard statistical test suites such as DieHarder and TestU01, they appear to be suitable for "everyday" use such as in Monte Carlo algorithms. One version seems especially suitable for use as a replacement for `java.util.Random`, because it produces sequences of higher quality, is faster in sequential use, is easily parallelized for use in JDK8 stream expressions, and is amenable to efficient implementation on SIMD and GPU architectures.

Each `SplittableRandom` object produces a stream of pseudorandom values of period $2^{64}$; this period is relatively short, but if values were generated sequentially at a rate of one per nanosecond, it would take over five centuries to cycle through the period once. In practice, one would generate such a large quantity of pseudorandom values in parallel, using splitting, which gives each thread (or core) a different $\gamma$ value and therefore a different permutation of the $2^{64}$ values. Since almost $2^{63}$ distinct $\gamma$ values may be used, the state space of a `SplittableRandom` object has almost $2^{127}$ states. (Salmon et al. [30] make this same point, and declare: "A PRNG with a period of $2^{19937}$ is not substantially superior to a family of $2^{64}$ PRNGs, each with a period of $2^{130}$.") If this is insufficient for certain applications, a version using arithmetic of higher precision (128 bits or more) may suffice. However, a specific attraction of `SplittableRandom` is that it has the relatively small "likelihood of accidental overlap" of a PRNG with $2^{127}$ states but the computational speed of a 64-bit algorithm. Code size and data size are quite small.

In the future we would like to subject these algorithms to even more rigorous testing. It may also be that even more effective mixing functions will be discovered. Finally, we hope that theoreticians will be able to illuminate the strengths and weaknesses of this family of algorithms, and to that end we offer the following observations. In an alternate version of the code for `SplittableRandom` (Figure 16), we used `mix64` rather than `mix64variant13` in method `mixGamma`. In the end we chose not to share code and use `mix64variant13`, not because of any hard evidence or solid theory, but simply because of an intuition (or engineering superstition?) that, all else being equal, it is best to avoid opportunities for accidental correlation. On the other hand, we chose to use `GOLDEN_GAMMA` in two places, and one would think the same intuition might lead us to use a different value (perhaps the odd integer closest to $2^{64}/\delta_S$, where $\delta_S = 1 + \sqrt{2}$ is the silver ratio?). And even our choice of the odd integer closest to $2^{64}/\phi$ was based only on the intuition that it might be a good idea to keep $\gamma$ values "well spread out" and the fact that prefixes of the Weyl sequence generated by $1/\phi$ are known to be "well spread out" [14, exercise 6.4-9]. Finally, guided by intuition and informal argument, we made a somewhat arbitrary decision to suppress certain $\gamma$ values in the method `mixGamma`—and yet we are haunted by the memory of the Enigma cipher machines, which were carefully designed so that "bad" encryptions could never occur by guaranteeing that no letter would ever map to itself, and so the word "LONDON" would never be encrypted as the word "LONDON", yet that avoidance of "bad" encryptions was the Achilles' heel that allowed the codebreaking machinery (the "cryptologic bombes" designed by Alan Turing and Gordon Welchman) to be so effective. By analogy, we worry that our effort to avoid "bad" random sequences that we believe would occur only rarely has in fact just introduced an unnecessary and undesirable bias. True randomness requires not that "miracles" or "disasters" be absent, but that they be present in due proportion—and human intuition is notoriously poor at judging this proportion. In the future, all of the choices outlined in this paragraph should be questioned and tested against better theory than we have so far mustered. It would be a delightful outcome if, in the end, the best way to split off a new PRNG is indeed simply to "pick one at random."

## Acknowledgments

We thank Claire Alvis for implementing our first prototype for Fortress, Martin Buchholz for suggesting that we test sparse $\gamma$ values. and Brian Goetz, Paul Sandoz, Peter Levart, Kasper Neilsen, Mike Duigou, and Aleksey Shipilev for discussion of suitability of these algorithms for Java JDK8.

## References

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification Version 1.0, March 2008.

[2] Austin Appleby. Murmurhash3, April 3, 2011. Project wiki entry. `http://code.google.com/p/smhasher/wiki/MurmurHash3` Accessed Sept. 10, 2013.

[3] Lennart Augustsson. Personal communication, Aug. 30, 2013.

[4] Richard P. Brent. Note on Marsaglia's Xorshift random number generators. *Journal of Statistical Software*, 11(4):1–5, Aug. 2004.

[5] Robert G. Brown, Dirk Eddelbuettel, and David Bauer. Dieharder: A random number test suite, version 3.31.1, 2003–2006. `http://www.phy.duke.edu/~rgb/General/dieharder.php` Accessed Sept. 10, 2013.

[6] F. Warren Burton and Rex L. Page. Distributed random number generation. *J. Functional Programming*, 2(2):203–212, April 1992.

[7] Koen Claessen and Michał Pałka. Splittable pseudorandom number generators using cryptographic hashing. In *Proc. ACM SIGPLAN Haskell Symp.*, pages 47–58, 2013.

[8] Horst Feistel. Cryptography and computer privacy. *Scientific American*, 228(5):15–23, May 1973.

[9] Gregory W. Fischer, Ziv Carmon, Dan Ariely, Gal Zauberman, and Pierre L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, Jan. 1999.

[10] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification (first edition)*. Addison-Wesley, 1996.

[11] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley, Upper Saddle River, New Jersey, 2013.

[12] P. Hellekalek. Don't trust parallel Monte Carlo! In *Proc. Twelfth Workshop on Parallel and Distributed Simulation (PADS 98)*, pages 82–89. IEEE, 1998.

[13] Donald E. Knuth. *Seminumerical Algorithms* (third edition), volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, third edition, 1998.

[14] Donald E. Knuth. *Sorting and Searching* (second edition), volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1998.

[15] Pierre L'Ecuyer. Efficient and portable combined random number generators. *Comm. ACM*, 31(6):742–751, June 1988.

[16] Pierre L'Ecuyer. Uniform random number generators. In *Proc. 30th Winter Simulation Conf.*, WSC '98, pages 97–104. IEEE Computer Society Press, 1998.

[17] Pierre L'Ecuyer. Software for uniform random number generation: Distinguishing the good and the bad. In *Proc. 33nd Winter Simulation Conf.*, WSC '01, pages 95–105, Washington, DC, USA, 2001. IEEE Computer Society.

[18] Pierre L'Ecuyer and Richard Simard. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4), August 2007.

[19] Pierre L'Ecuyer, Richard Simard, E. Jack Chen, and W. David Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, Nov. 2002.

[20] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proc. 17th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming*, pages 193–204, New York, 2012. ACM.

[21] Sean Luke. Documentation for the Mersenne Twister in Java, October 2004. `http://www.cs.gmu.edu/~sean/research/mersenne` Accessed March 11, 2014.

[22] George Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, Jul. 2003.

[23] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.

[24] Martin Odersky. *The Scala Language Specification, Version 2.7*. EPFL Lausanne, Switzerland, 2009.

[25] Oracle. Interface `spliterator<t>`. Documentation for Java™ Platform Standard Ed. 8 DRAFT ea-b106. `http://download.java.net/jdk8/docs/api/java/util/Spliterator.html` Accessed Sept. 13, 2013.

[26] Stephen K. Park and Keith W. Miller. Random number generators: Good ones are hard to find. *Comm. ACM*, 31(10):1192–1201, October 1988.

[27] Ronald L. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6 block cipher, version 1.1, August 20, 1998. Unpublished paper. `http://people.csail.mit.edu/rivest/pubs/RRSY98.pdf`.

[28] C. S. Roberts. Implementing and testing new versions of a good, 48-bit, pseudo-random number generator. *Bell System Technical Journal*, 61(8):2053–2063, Oct. 1982.

[29] Mutsuo Saito and Makoto Matsumoto. A deviation of CURAND: Standard pseudorandom number generator in CUDA for GPGPU, Feb. 2012. Slides presented at the Tenth Intl. Conf. Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing. `http://www.mcqmc2012.unsw.edu.au/slides/MCQMC2012_Matsumoto.pdf`.

[30] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: As easy as 1, 2, 3. In *Proc. 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 16:1–16:12. ACM, 2011.

[31] Tao B. Schardl. Personal communication, Aug. 13, 2012.

[32] Richard Simard. TestU01 version 1.2.3, August 2009. Website at `http://www.iro.umontreal.ca/~simardr/testu01/tu01.html`.

[33] David Stafford. Better bit mixing: Improving on MurmurHash3's 64-bit finalizer, September 28, 2011. Blog "Twiddling the Bits." `http://zimbry.blogspot.com/2011/09/better-bit-mixing-improving-on.html` Accessed Sept. 10, 2013.

[34] Thinking Machines Corporation. CM-2 Technical Summary. Technical Report HA87-4, Cambridge, Massachusetts, April 1987. `https://archive.org/details/06Kahle001885`.

[35] John Walker. HotBits: Genuine random numbers, generated by radioactive decay. Website at `http://www.fourmilab.ch/hotbits/`.

[36] A. F. Webster and Stafford E. Tavares. On the design of S-boxes. In *Advances in Cryptology (CRYPTO '85), August 18-22, 1985, Proceedings*, volume 218 of *Lecture Notes in Computer Science*, pages 523–534. Springer, 1985.