

Large-scale collision and birthday-spacings tests for pseudorandom number generators by space/time tradeoffs

Sebastiano Vigna 
Dipartimento di Informatica
Università degli Studi di Milano

Abstract

Collision and birthday-spacings tests for pseudorandom number generators become more powerful as the number of points grows, but the points must be stored, and at the scale required to stress modern generators they no longer fit in memory. We describe `coll-birth`, a Rust tool that runs these tests within a given memory budget using *space/time tradeoffs*. For the collision test, the tradeoff is simple: the cell space is partitioned by the top bits of the cell index into intervals processed in separate passes, each using a fraction of the memory, whose counts add up to the exact collision count. For the birthday-spacings test, instead, some spacings straddle interval boundaries, and equal spacings may come from different intervals: we thus introduce a two-level tradeoff, in which an inner level walks the intervals in order, carrying across their boundaries what is needed to compute exactly the spacings of the globally sorted sequence, and an outer level replays the inner one once per class of spacings, defined by their lower bits, counting coincidences class by class. Within the same memory budget, a single large test is much more powerful than many small ones. Points are stored, sorted in place, and scanned, rather than inserted into a hash table, and execution can be parallelized. On a single workstation with 64 GiB of RAM, `coll-birth` finds defects in the popular `wyrand` generator, in a linear congruential generator with an excellent spectral figure of merit, and in a multiply-with-carry generator, the latter after examining 12 PB of output in about five days.

Keywords: pseudorandom number generators, statistical testing, collision test, birthday spacings, space/time tradeoff, Rust.

1 Introduction

Empirical testing complements the theoretical analysis of a pseudorandom number generator (PRNG) by subjecting its output to a battery of statistical tests, each designed to expose a particular kind of regularity. The reference implementation in this area is TestU01,¹ whose `SmallCrush`, `Crush`, and `BigCrush` batteries have become a de facto standard; earlier batteries include DIEHARD² and the difficult tests of Marsaglia and Tsang.³

More recent batteries, such as PractRand,⁴ can process streams of many terabytes, but they are built around tests with a small, fixed memory footprint. Collision-type tests, on the other hand, need memory proportional to the number of points they examine, if they are to be run at maximum power.

Among the individual tests, the *collision test* and the *birthday-spacings test* are particularly informative. Both partition a t -dimensional hypercube into a large number of equal cells, generate points using the generator under test, and compare an observed count with its distribution under the hypothesis of uniformity (see Knuth,⁵ §3.3.2). The collision test counts how many points fall into an already occupied cell, whereas the birthday-spacings test sorts the points and counts coincidences among the *gaps* between consecutive occupied cells;^{6,7} in the sparse regime, both counts are well approximated by a Poisson distribution.⁸

The main property of these tests is that their power grows with the number of points: a generator that passes at 10^8 points may fail at 10^{10} . Running the tests at a larger scale is thus the most direct way to make them stronger, but there are two obstacles: the points must be stored and sorted, and tens of billions of multi-byte cell indices do not fit in the memory of a typical workstation; moreover, generating and sorting that many points is expensive enough that single-threaded execution is impractical.

In this paper we describe `coll-birth`, a tool written in Rust⁹ that addresses both obstacles using *space/time tradeoffs*. For the collision test, the tradeoff is simple: we partition the cell space into disjoint intervals using the upper bits of the cell index, and we process the intervals in separate passes. Since equal points fall in the same interval, the per-pass counts add up to the *exact* collision count, and we can run a test on a given number of points in a fraction of the memory, at a proportionally higher computational cost.

The birthday-spacings test is a different story, and its tradeoff is the main contribution of this paper. Spacings are differences between consecutive points in sorted order, so the spacing of the first point of each interval straddles a boundary, and equal spacings may come from different intervals: neither the spacings nor their coincidences can be computed one interval at a time. We thus design a *two-level* tradeoff, whose passes must be orchestrated precisely.

The inner, *distance* level walks the intervals in increasing order, carrying across their boundaries the maximum of the last nonempty interval, which is the predecessor of the first point of the next one, and the global minimum, which, together with the global maximum known at the end of the walk, yields the circular wrap-around spacing. In this way, every spacing we compute is exactly a spacing of the globally sorted sequence.

The outer, *counting* level replays the whole inner level once for each class of spacings, determined by their lower bits, and counts the coincidences within the class; since equal spacings belong to the same class, the per-class counts add up to the exact total. Only one interval of points and one class of spacings are ever resident in memory, at the price of a number of passes that is the square of that of the collision tradeoff.

The two tradeoffs make it possible to run, on a single workstation with 64 GiB of RAM, collision tests on hundreds of billions of points, and birthday-spacings tests examining petabytes of generator output; at this scale, the tests find defects that smaller tests do not reveal. Getting there, however, also required some care in the engineering.

Points are not inserted into a hash table, as in classical implementations, but stored sequentially, sorted in place by a multithreaded radix sort, and counted by a linear scan; the generator under test is selected at compile time, so that the generation loop is fully specialized and inlined. Generation is split across threads so that a parallel run is bit-for-bit identical to the sequential one.

Finally, we show that the Poisson approximation of the number of collisions, which TestU01 uses up to density one, overestimates its variance by a factor of up to about four, and that a direct switch to a normal approximation underestimates the tails when the expected number of collisions is small. Depending on the density, we use instead a Poisson, a binomial, or a normal approximation with the exact mean and variance, which keeps all tails out to six standard deviations within 25% of the exact ones.

Section 2 recalls the statistical model, and Section 3 develops the space/time tradeoff for the collision test and the two-level tradeoff for the birthday-spacings test. Section 4 discusses the implementation, including the parallelization and the designs that did not work, and Section 5 illustrates the tool in use. Section 6 reports results for a few 64-bit generators, together with an analysis of where the time goes.

`coll-birth` is free software, distributed under the Apache License 2.0 or the MIT License, at the user’s choice, through its GitHub repository.¹

¹<https://github.com/vigna/coll-birth-rs/>

2 The collision and birthday-spacings tests

2.1 The cell grid

A test is parameterized by a number of upper bits u , a dimension t , and an optional left shift s (zero by default). Each point is a t -tuple of cell coordinates; each coordinate is obtained from one 64-bit generator output by a left shift of s bits followed by the extraction of the upper u bits, which gives a value in $[0..2^u)$.

The t coordinates are concatenated into a single cell index in $[0..2^{ut})$, so the grid has

$$k = (2^u)^t = 2^{ut} \quad (1)$$

cells. The shift s selects which bits of the output are tested: $s = 0$ tests the upper bits, whereas a positive s moves the window toward the lower bits.

2.2 Collisions

When n points are thrown independently and uniformly into k cells, the number C of *collisions*, that is, of points that land in an already occupied cell, is n minus the number of distinct occupied cells. Its expectation is

$$\lambda = \mathbb{E}[C] = n - k + k \left(1 - \frac{1}{k}\right)^n, \quad (2)$$

and in the sparse regime $n \ll k$ the leading term is $\lambda \approx n^2/2k$.

We evaluate (2) accurately in both regimes. For $n/k \leq 0.1$ we sum the alternating series $\lambda = \sum_{i \geq 2} (-1)^i \binom{n}{i} k^{1-i}$, which is an exact rewriting of (2) whose terms decrease geometrically in that range, so the truncation error is bounded by the first omitted term; otherwise, we use the closed form, expanding $\log(1 - 1/k)$ as a series for numerical stability when k is large.

Under the uniformity hypothesis, in the sparse regime C is approximately Poisson with mean λ ,⁸ so a two-sided p -value can be read directly from the Poisson tails. Since the distribution is discrete, following L'Ecuyer and Simard¹ (Section 3) we distinguish the left and right tail values $p_L = \Pr[X \leq C]$ and $p_R = \Pr[X \geq C]$ for $X \sim \text{Pois}(\lambda)$.

We then fold the two tails into a single value using the convention of the `gofw_pDisc` function of TestU01: the reported value is p_R when it is the smaller tail, $1 - p_L$ when $p_L < 1/2$, and $1/2$ otherwise. The tails are computed by the `cdflib` crate, a pure-Rust port of CDFLIB.¹⁰

As the density n/k grows, however, the Poisson approximation deteriorates: C is $n - k$ plus the number of empty cells, so its variance is that of the number of empty cells,

$$\sigma^2 = k(k-1) \left(1 - \frac{2}{k}\right)^n + k \left(1 - \frac{1}{k}\right)^n - k^2 \left(1 - \frac{1}{k}\right)^{2n}, \quad (3)$$

which for large k is approximately $ke^{-\alpha}(1 - (1+\alpha)e^{-\alpha})$, where $\alpha = n/k$. The ratio σ^2/λ is about $1 - 4\alpha/3$ when α is small, and about 0.26 at $\alpha = 1$: a Poisson approximation thus overestimates the variance, and with it the tails.

How large is the error? For moderate n , the exact distribution of C can be computed by a recursion on the number of points: compared with it, at six standard deviations the Poisson tails are about 20% too large when $\sigma^2 = 0.99\lambda$, ten times too large at $\alpha = 0.1$, and a million times too large at $\alpha = 1$.

The `smultin_CreateCollisions` function of TestU01 uses the Poisson approximation up to $n/k = 1.0001$, so a test at density close to one is judged against a distribution whose variance is almost four times too large, and its p -values cluster around $1/2$. L'Ecuyer, Simard, and Wegenkittl⁸ switch instead to a normal approximation at density $1/32$. However, when λ is small the normal approximation ignores the skewness of C : for example, at $n = 10^4$ and $\alpha = 0.0075$ it underestimates the right tail at six standard deviations by a factor of about 30.

`coll-birth` uses three approximations, chosen by the ratio σ^2/λ : the Poisson distribution when the ratio is at least 0.99 ($\alpha \lesssim 0.0075$); a binomial distribution with mean λ and variance

σ^2 , that is, with success probability $q = 1 - \sigma^2/\lambda$ and $N = \lambda/q$ trials, when it is between 0.54 and 0.99 ($\alpha \lesssim 0.46$); and a normal distribution with mean λ and variance σ^2 below 0.54.

The binomial tails are computed using the regularized incomplete beta function, which extends them to non-integer N , and the normal tails are, as in TestU01, $p_L = \Phi((C - \lambda)/\sigma)$ and $p_R = 1 - \Phi((C - 1 - \lambda)/\sigma)$, which turn out to be more accurate than continuity-corrected tails. As the ratio decreases, the binomial approximation deteriorates and the normal one improves: 0.54 is the threshold at which the two have the same error, independently of n .

With this choice, all tails out to six standard deviations are within 25% of the exact ones already at $n = 10^4$, the minimum number of points accepted by the tool, and the error decreases as n grows; the tails are then folded as above. Note that p -values very far in the tails (such as 5.2×10^{-292} in Table 2) are nominal, as no approximation has relative accuracy that deep.

We remark that (3) suffers from catastrophic cancellation when k is large. We thus evaluate it as $k(a - b) + k^2(b - a^2)$, with $a = (1 - 1/k)^n$, $b = (1 - 2/k)^n$, $a - b = -a((1 - 1/(k - 1))^n - 1)$, and $b - a^2 = a^2((1 - 1/(k - 1)^2)^n - 1)$, where the powers are computed through $\log(1 + x)$ and $e^x - 1$ primitives that are accurate for small x .

As a function of the density, the variance is asymptotically $kf(\alpha)$, where $f(\alpha) = e^{-\alpha} - (1 + \alpha)e^{-2\alpha}$. Since $f'(\alpha) = (1 + 2\alpha)e^{-2\alpha} - e^{-\alpha}$, the variance is maximized when $e^\alpha = 1 + 2\alpha$, that is, at $\alpha \approx 1.256431$. Beyond this point a denser test costs more, but the spread of the number of collisions shrinks, as more and more cells become occupied: thus, by default `coll-birth` uses $\lfloor 1.256431 k \rfloor$ points, and it accepts at most $1.26 k$ points.

2.3 Birthday spacings

The birthday-spacings test, introduced by Marsaglia,⁶ sorts the cell indices, computes the spacings between consecutive values (wrapping around from the last to the first), sorts the spacings, and counts coincidences among them (see also Knuth,⁵ §3.3.2). Under uniformity, the number of such coincidences is approximately Poisson with mean

$$\lambda = \frac{n^3}{4k}. \quad (4)$$

TestU01 restricts the birthday-spacings test with r repetitions to $8r\lambda \leq k^{1/4}$, that is, $n \leq k^{5/12}/(2r)^{1/3}$, a region in which the Poisson approximation is accurate.⁷ Since power grows with the number of points, by default `coll-birth` uses the largest n in this region, for which the overall expected number of coincidences is $k^{1/4}/8$, and it rejects larger values.

3 The space/time tradeoff

Storing n points is the main cost of the collision (or birthday-spacings) test. Counting collisions is, however, equivalent to counting distinct elements, a widely studied problem in streaming algorithms and in database systems. Beame¹¹ proves an inherent quadratic time/space tradeoff for the related problem of listing the elements that appear exactly once in a sequence; for our purposes, the relevant fact is that space can always be traded for time.

There are general adaptive techniques to obtain this result, but in the case of PRNG testing the situation is much simpler, as unless the generator is pathological the points will be approximately uniformly distributed. Thus, we can trade space for time just by running the test only on the points whose upper b bits (the *top bits*) are equal to a fixed value $j \in [0..2^b)$, for all possible values of j . Collisions are possible only among points with the same top bits, so by adding up the collisions across all values of j we obtain the overall number of collisions; running the test for a fixed j , however, requires only about $n/2^b$ memory locations.²

²The approximation is due to the binomial distribution of the number of points with the same top bits, but the probability of a large deviation can be controlled using balls-into-bins bounds (Section 3.4).

Partitioning by key value is a classical technique in database systems, where it is the basis of hash-based partitioning for joins and duplicate elimination.¹² In that setting, however, each pass rereads the input from external storage. In PRNG testing the input need not even be stored, as the generator can simply be restarted: the partition trades calls to the generator, that is, computation, for memory. To the best of our knowledge, no implementation of collision tests has exploited this possibility.

For the collision test, the partition is all we need. The birthday-spacings test, instead, requires a more elaborate, two-level structure, which we describe in Section 3.5, after developing the collision case in full.

3.1 Samples and passes

We begin by changing the way collision tests are traditionally parameterized: rather than using the number of points n as the main parameter, we fix a *memory budget* m and derive n from the desired tradeoff (and from further parameters, as we will see in the next sections). The memory budget is respected up to a few percent (Section 3.4).

We call a *sample* a position in the orbit of the generator at which a candidate t -tuple is drawn. In the standard test there is a bijection between samples and points.

Fix b tradeoff bits. The combined cell index is partitioned into 2^b contiguous value intervals by its top b bits, and pass $j \in [0 \dots 2^b)$ retains exactly the samples whose top b bits are equal to j . Each pass scans $n = m \cdot 2^b$ samples and keeps the $\approx m$ points that fall in its interval, computing the associated collisions; at the end of the passes, the overall number of collisions is the sum of the per-pass counts.

A *call* is a call to the next-value function of the generator. In the standard test, the numbers of samples and of points are both n , and the number of calls is tn (a sample requires t calls). In a tradeoff run, every sample is scanned 2^b times, once per pass, so the number of calls grows by the same factor.

Note that from the viewpoint of the memory budget m , amplifying the number of points of the test linearly by a factor of 2^b (i.e., $n = 2^b \cdot m$) increases the number of calls by a factor of $(2^b)^2$ (i.e., quadratically): the product of time and space thus remains quadratic in the number of points, consistently with the lower-bound curve of Beame.¹¹

3.2 A pass is a collision test

A single pass is a test in itself: retaining only the samples whose top b bits are equal to a fixed value restricts the points to a contiguous interval of the cell space, so pass j is a collision test on an effective cell space of 2^{ut-b} cells, and the whole tradeoff recombines the 2^b such sub-tests into the exact, full-resolution count.

O'Neill¹³ already noted that running a collision test on a reduced cell space might strengthen it (in the regime where fewer cells yield more collisions per point), and reduced the space by *decimation*, fixing instead the *lower* bits of each coordinate. Decimation is orthogonal to our top-bit partition, and we add it in Section 3.3.

Note that each pass scans a fixed number of *samples*, not a fixed number of points: the number of points that fall in a given interval depends on the output of the generator, and it is therefore a random variable. As we will see in Section 4.4, this fixed-sample formulation makes it possible to parallelize the test.

3.3 Adding decimation

Decimation is interesting also by itself: given the same budget for memory and number of calls, the space/time tradeoff provides a larger collision test, whereas decimation explores a larger fraction of the orbit of the generator. More precisely, with memory m and the same number

of calls, which happens when $td = 2b$, the two techniques yield the same expected number of collisions up to lower-order terms, as both are approximately $m^2 2^{2b}/2k$: the tradeoff examines $m 2^b$ points at full resolution, whereas decimation examines $\approx m$ points at a resolution reduced by d bits per coordinate, drawn from a stretch of the orbit that is 2^b times longer.

Which of the two detects a defect more easily? This depends on the generator, but since the two techniques are orthogonal, they can be easily combined. We thus give the following general definition, whose sampling modes are summarized in Table 1.

A *collision test* is given by the following parameters:

- w : the width in bits of the generator output;
- $s < w$: the left shift applied to the output before extracting the upper bits;
- t : the dimension (each point is given by t coordinates);
- $u \leq w - s$: the upper bits extracted from the shifted output, per coordinate;
- $d < u$: the decimation bits, per coordinate;
- $v \in [0 \dots 2^{td}]$: a decimation value (only for $d \neq 0$);
- $b \leq t(u - d)$: the tradeoff bits (top bits of the combined index);
- m : the approximate memory budget;
- r : the number of repetitions.

A *sample* is generated by taking t times the upper u bits of the w -bit generator output shifted to the left by s , forming a t -tuple of cell coordinates.

The test draws $m \cdot 2^b \cdot 2^{td}$ samples 2^b times, starting each time from the same state of the generator. Only the tuples in which the concatenated lower d bits of the coordinates match the decimation value v are kept (`coll-birth` fixes $v = 0$), so the cell space has size $(2^{u-d})^t$. Decimation thus increases the expected number of collisions by a factor of 2^{td} ; in some circumstances (e.g., a generator producing no collisions at all) this is enough to reject the generator using little memory.

The combined cell index is then partitioned into 2^b contiguous value intervals by its top b bits, and each interval is processed in a pass that keeps the samples that both survive decimation and fall in the interval, and counts their collisions. The overall count is the sum of the per-pass counts.

The number of calls to the generator is thus $m \cdot (2^b)^2 \cdot 2^{td} \cdot t$, whereas the number of points of the test is approximately $m \cdot 2^b$. The approximation is due to the fact that decimation does not reduce the number of samples exactly by a factor of 2^{td} (the number of points kept by each tradeoff pass is random, too, but without decimation the passes keep exactly $m \cdot 2^b$ points in total).

For this reason, all reported p -values are computed using the null distribution conditioned on the number of points actually kept. Given this number, the collision count has the distribution of Section 2.2: indeed, under the null hypothesis whether a sample is kept depends only on the discarded lower bits, which are independent of the retained ones, so the kept points are independent and uniformly distributed on the reduced grid.

Without conditioning, the randomness of the number of kept points would add to the variance a term of order $4\lambda^2/n$, which is not negligible precisely when decimation has made λ/n large. Without decimation, the number of kept points is deterministic, and conditioning has no effect.

Finally, the test is repeated r times: since the collision counts of different repetitions are independent, the mean and the variance of their sum are r times those of a single repetition, and we approximate its distribution as in Section 2.2. Small deviations in each repetition thus

Table 1: The four sampling modes, as a function of the upper bits u , dimension t , memory locations m , decimation bits d , and the (total, top) tradeoff bits b . “Points” is the number examined by the test; “peak memory” is the working set; “generator outputs” is the approximate number of calls to the underlying generator. Counts involving decimation are expectations, as the kept count is a random variable. For the birthday-spacings test the tradeoff has a second level (Section 3.5), which multiplies the generator outputs by a further factor of 2^b .

Mode	Cells k	Points n	Peak memory	Generator outputs
Plain	2^{ut}	m	$\Theta(n)$	$m t$
Tradeoff (b)	2^{ut}	$m 2^b$	$\Theta(n/2^b)$	$m 2^{2b} t$
Decimation (d)	$2^{(u-d)t}$	$\approx m$	$\Theta(n)$	$m 2^{td} t$
Both (d, b)	$2^{(u-d)t}$	$\approx m 2^b$	$\Theta(n/2^b)$	$m 2^{2b} 2^{td} t$

accumulate, and can provide a stronger result.³ The number of repetitions is also the only open-ended parameter: r can simply be set to a very large value, and the repetitions can be stopped manually.

3.4 Sizing the working set

Since a pass keeps a binomially distributed number of points, its buffer is sized using a balls-into-bins bound. Each pass is one bin of an experiment that throws the candidate points into $N = 2^{td+b}$ bins using their decimation and tradeoff bits. The load of a single bin is a sum of independent indicator variables with mean m , so by Bernstein’s inequality (Boucheron, Lugosi, and Massart,¹⁴ Equation 2.10) it exceeds $m + h$ with probability at most $\exp(-h^2/2(m + h/3))$.

A union bound over the N bins then shows that if we allocate $m + h$ slots, where

$$h = \frac{L}{3} + \sqrt{\frac{L^2}{9} + 2mL}, \quad L = \ln N + 1000 \ln 10$$

is obtained by inverting the exponent exactly, the probability of overflow in the uniform case is below 10^{-1000} for every m and N .

This headroom has the optimal form: by Theorem 1 of Raab and Steger,¹⁵ in the heavily loaded regime $m \gg (\ln N)^3$ the maximum bin load does reach $m + \sqrt{2m \ln N} (1 - o(1))$. The larger constant in L is the price we pay for the overflow target: a matching lower bound on the upper tail of the load shows that about $\sqrt{2m \ln(N/\varepsilon)}$ additional slots are in fact necessary for overflow probability ε . The same bound is used to size the auxiliary buffer of the progressive checkpoints of a decimated run.

3.5 Birthday spacings: a two-level tradeoff

The birthday-spacings test sorts the points and replaces each point by its spacing to the predecessor, and then runs a collision test on the spacings. A tradeoff for it must keep the points of a pass in a *contiguous* interval of the cell space, as the spacings within a pass are correct only if the pass is an interval of the global sorted order. This is exactly what a partition by the *top* b bits provides, and it is an independent reason for keying the collision tradeoff on the top bits, too.

The test thus runs on two nested levels, each with 2^b passes. The inner, *distance* level walks the 2^b top-bit value intervals in increasing order; within each interval, the points are generated, sorted, and replaced in place by their spacings.

³This is how, traditionally, the collision test is strengthened without increasing memory usage.

The first point of an interval has no predecessor within the interval: its spacing is computed with respect to the maximum of the previous nonempty interval, which is carried along the walk (an empty interval simply forwards the carried maximum). The spacing of the overall minimum point is the circular wrap-around $k - \max + \min$, which is added once, at the end of the walk, when the global maximum is known. Since the intervals are contiguous and visited in increasing order, the spacings obtained in this way are exactly the spacings of the globally sorted sequence.

Equal spacings, however, may come from different intervals, so they cannot be counted interval by interval, and since there are as many spacings as points, they cannot be buffered within the memory budget either. The outer, *counting* level thus replays the distance level 2^b times, each time keeping only the spacings in one *class*, which are then sorted and counted.

Classes are given by the *lower* b bits of the spacing, not by its top bits: spacings cluster near zero (their distribution is approximately exponential with mean $k/n \gg 2^b$), so a split by top bits would put almost all spacings in the same class, whereas the lower bits are balanced. Since equal spacings share all bits, they belong to the same class, and the per-class counts sum to the exact total.

Note that the two levels have a different structure. The classes of the outer level are independent, and they can even be run on different machines (Section 5.1); the intervals of the inner level, instead, must be visited in increasing order, as each interval needs the maximum carried from the previous ones, and every replay must start from the same state of the generator, so that each class sees exactly the same spacings. Within an interval, however, the points are generated and sorted in parallel, as in the collision test.

The two nested levels give $(2^b)^2$ passes overall, and thus a further factor of 2^b in generator calls with respect to the collision tradeoff. However, only the $\approx m$ points of one interval and the $\approx m$ spacings of one class are ever resident, so the peak memory remains $\Theta(n/2^b)$.

4 Implementation

4.1 Generator selection at compile time

The generator under test is chosen at build time using a Rust *feature*; exactly one must be enabled. Each feature compiles a single `Prng` type exposing a `new(seed)` constructor and a `next_u64()` step function, so the generation loop is monomorphized and inlined around one concrete generator, with no indirect calls.

The distribution includes several generators, among which all the generators used in Section 6, and new generators can be added by writing one feature and one `Prng` implementation. Generators narrower than 64 bits return their output in the upper bits, so that the standard upper-bit extraction tests their good bits.

4.2 Store, sort, scan

Classical implementations of collision tests use a dictionary (typically, a hash table) to store the points seen so far. This is the case, for instance, of TestU01,¹ whose hash-table implementation, moreover, cannot handle more than about 2×10^9 points. We suggest instead a store/sort/scan approach that is typical of deduplication: once again, we treat the problem of counting collisions as a problem of counting distinct elements.

Hash tables are notoriously expensive in terms of cache misses, as each access hits a random-looking memory address. The store/sort/scan approach, on the other hand, requires a sorting pass that might look expensive; moreover, we have to store *all* points, even duplicates, which in principle requires more space, and thus, with the same budget, a test using fewer samples.

However, points can be stored sequentially at memory bandwidth (locations are prefetched), and an in-place unstable (possibly parallel) radix sort is very effective on random data. In our experience, on a variety of hardware configurations, store/sort/scan is an order of magnitude faster

than the hash-table approach: each insertion hits a random memory location, and insertions are inherently sequential, whereas all three phases of our approach access memory mostly sequentially and run in parallel.

The objection based on space is also easily dismissed: the densest collision test accepted by `coll-birth` has $n = 1.26k$ points (Section 2.2), and in that case the expected number of collisions is

$$k \left(1 - \frac{1}{k}\right)^n - k + n \approx k \cdot e^{-1.26} - k + 1.26k \approx 0.5437k.$$

Thus, in this worst case for the store/sort/scan approach we store $n = 1.26k$ entries where a hash table would store only the $\approx 0.7163k$ distinct ones, that is, about 76% more. Sequential storage, however, has no overhead, whereas a hash table, due to load factors and to the sizing of the backing table, has a similar overhead in the worst case.

Moreover, this evaluation is unrealistic for any collision test on a significant number of bits (e.g., collisions on the 64-bit output of a 64-bit generator): in most practical cases, the number of collisions is a tiny fraction of the number of samples.

In practice, we store cell indices in a region allocated with `mmap`, requesting transparent huge pages to reduce TLB pressure, and we fault the pages eagerly so that the generation loop never stalls on a page fault. Sorting uses a multithreaded in-place radix sort;¹⁶ after sorting, collisions are pairs of adjacent equal elements, which are counted by a parallel scan.

Being in place matters as much as being fast: a sort that allocates a scratch buffer as large as its input doubles the peak memory, and thus halves the size of the largest test that fits in a given budget. This is not a hypothetical concern, as the convenience entry point of the radix-sort library we use sorts 64-bit keys out of place: `coll-birth` thus explicitly calls its in-place variants.

4.3 Choosing the cell-index width

The cell index is stored in the narrowest unsigned integer that can hold every value: `u32` when $k \leq 2^{32}$, `u64` when $k \leq 2^{64}$, and `u128` otherwise. Halving the element width halves the memory and roughly doubles sorting throughput, so the narrowest viable type is always chosen.

The same widths serve the birthday-spacings test, even though its wrap-around spacing $k - \max + \min$ lies in $[1..k]$, and can thus exceed the largest representable value when $k = 2^{32}$ in `u32` storage or $k = 2^{64}$ in `u64` storage. The wrap-around is thus evaluated through the always-representable $k - 1$ as $(k - 1 - \max) + \min + 1$, in which every intermediate result fits whenever $\min < \max$. In the only remaining case, $\min = \max$, all points fall into the same cell: the wrap-around is k , whereas every other spacing is zero, so it collides with nothing and it is ignored, leaving the collision count unchanged.

4.4 Parallelization

A naive parallel run gives each thread its own decorrelated sub-stream, which is fast, but tests a different point set than the sequential run, so the two results differ. We instead partition the *single* sequential orbit into contiguous segments, one per thread, and each thread fills a disjoint sub-region of one shared buffer. The gaps left by underfilled sub-regions (decimation and tradeoff passes keep a random number of points) are then closed by an in-place compaction, and the buffer is sorted and scanned as in a sequential run.

Since the collision count depends only on the multiset of cells drawn, which is exactly the sequential multiset, the parallel result is identical to the sequential one for any number of threads.

Why compaction? Each sub-region must be sized for the worst case of its own random number of kept points (Section 3.4), so in general the kept points do not tile the buffer. Writing them directly at their final positions would require knowing in advance how many points each thread keeps, which happens only in plain mode; padding the gaps with a sentinel value, which

the sort would move to the end, requires a value that is not a cell index, and no such value exists when the cell space saturates the storage width (e.g., 2^{64} cells stored in a `u64`).

Compaction moves the blocks to the left in order, so no block overwrites data that has not been moved yet; in plain mode, every block is already in place, and compaction does nothing.

Reaching the start of the segment of a thread means advancing the generator by a known number of steps without drawing them one at a time. `coll-birth` asks each generator to do so through a `try_skip(j)` method, which advances the state as if `next_u64` had been called j times, and returns an error for generators that have no closed-form jump. Since the method must either succeed for every argument or fail for every argument, the parallel runner probes the capability once (with `try_skip(0)`) and, if it succeeds, has every thread jump to the start of its segment.

When `try_skip` reports no jump, as is the case for some of the nonlinear generators distributed with the tool, `coll-birth` falls back to a single sequential *pre-scan* that walks the orbit once and saves the state of the generator at each segment boundary. The pre-scan is sequential, but it runs once, and it is amortized over all passes of a tradeoff run: since it costs about as much as one pass run on a single thread, on p threads it increases the running time by a fraction of about $p/2^b$ (or $p/2^{2b}$ for birthday spacings), which is negligible when the number of passes is large with respect to the number of threads.

In a single-pass run, instead, the pre-scan is a full sequential generation, and parallelizing a generator without a jump-ahead is counterproductive. Note that since every sample consumes exactly t outputs, a decimated run is contiguous in the orbit (Section 3.3), and it can be split faithfully in the same way.

4.5 Two designs that did not work

The single-buffer design is the third we tried, and the first two failed in instructive ways. In the first design, each thread stored its points in a private buffer, and sorted it with a single-threaded in-place radix sort. This is the obvious way of keeping all cores busy, and indeed it kept them busy, but not sorting.

Maybe surprisingly, the culprit was memory allocation: the sort allocates a small bucket array at each node of its recursion, and with all threads sorting at the same time the allocations serialized on a lock inside the glibc memory allocator. The cores spent essentially all their time in the kernel, and the sorts made essentially no progress.

In the second design, the private buffers were sorted one at a time, each with the multithreaded sort, and the collisions spanning different buffers were found by a p -way merge of the sorted buffers driven by a heap. The merge has the right asymptotic cost, $O(n \log p)$ on p threads, but on large inputs it turned out to be slower than the sort itself: every element pays a heap operation whose next memory access depends on the result of the previous one, so the merge is bound by memory latency, whereas the radix sort and the linear scan stream through memory, and are bound by its bandwidth.

A single buffer avoids the merge altogether: collisions spanning the boundaries of the former thread buffers are just pairs of adjacent equal elements, which are found by the same linear scan. As a result, counting, which used to dominate the running time, now takes a few percent of the time of the sort (Section 6.5).

4.6 Testing

Faithfulness is not only a matter of reproducibility: it makes the parallel code testable, as every parallel run has an exact sequential oracle. The crate contains integration tests checking that the per-pass counts of a tradeoff run sum to the count of the plain run, that parallel runs return the same results as sequential runs, and that single passes run separately (Section 5.1) recombine into the result of the full run.

Moreover, unit tests compare the approximating distributions of Section 2.2 with the exact distribution of the number of collisions, computed by a recursion on the number of points, out to six standard deviations.

5 Using the software

The tool is built and run using `cargo`, the Rust package manager: the generator is selected using a feature at build time, and the grid parameters are passed at run time. For example, a basic collision test of `SplitMix` on a grid of $u = 16$ upper bits in $t = 2$ dimensions, with 10^7 points and a fixed seed, is invoked as follows (in the excerpts below, [...] elides timing information, which varies from run to run, and other repetitive fragments):⁴

```
$ cargo run -r -F splitmix -- 16 2 10000000 -S 1
Generator: SplitMix
Seed: 0x0000000000000001
Running a 2-dimensional collision test sequentially on the upper
16 bits of the full 64-bit output (10000000 points, 32-bit cells,
10000000 memory locations, 0.037 GiB RAM)
u: 16 t: 2 cells: 4294967296 expected collisions: 11632.501259768274
Generating points...[...] sorting...[...] counting collisions...[...]
done.
11754 p=0.1310416600017843
Test completed in [...] seconds
11754 p=0.1310416600017843
```

The grid has $k = 2^{32}$ cells, and the expected number of collisions is $\lambda \approx 11632.5$. We observe 11754 collisions, which yield a p -value of 0.13, as expected for a generator of this quality. All diagnostics are printed on standard error, but the final line (the collision count and its p -value) is repeated on standard output, so that results can be collected by scripts.

5.1 The space/time tradeoff

The same test can be run in 2^b passes using the `-b (--tradeoff-bits)` option, where b is the number of top bits of the combined index used to partition it. With $b = 4$ there are $2^4 = 16$ passes, and the per-pass memory parameter $m = 625\,000$ yields the same $n = 16 \cdot 625\,000 = 10^7$ points as above, in one sixteenth of the memory:

```
$ cargo run -r -F splitmix -- 16 2 625000 -b 4 -S 1
Generator: SplitMix
Seed: 0x0000000000000001
Running a 2-dimensional collision test sequentially on the upper
16 bits of the full 64-bit output (10000000 points, 32-bit cells,
625000 memory locations, 0.003 GiB RAM (+8.71%), tradeoff on 4 top
bits over 16 passes)
u: 16 t: 2 cells: 4294967296 expected collisions: 11632.501259768274
Pass 1/16: [...], 624572 points, 740 collisions, p=0.307[...]
Pass 2/16: [...], 623942 points, 707 collisions, p=0.735[...]
[...]
Pass 16/16: [...], 625096 points, 709 collisions, p=0.743[...];
combined: 10000000 points, 11754 collisions, p=0.1310416600017843
11754 p=0.1310416600017843
Test completed in [...] seconds
11754 p=0.1310416600017843
```

⁴On Linux, an additional header line reports the status of transparent huge pages.

The combined count after the sixteenth pass is still 11754, but the working set is almost sixteen times smaller. A p -value is printed after each pass, both for the pass alone and cumulatively; the (+8.71%) in the header is the balls-into-bins headroom of Section 3.4 added to the per-pass buffer.

Since the passes are independent, they can also be distributed across machines: the `--pass` option runs a single pass and prints its raw collision count together with its share of the null distribution, so that the 2^b results can be recombined afterwards. This is not possible with decimation, as in that case the null distribution depends on the total number of points kept.

Note that only the combined p -value is the result of the test: the per-pass values are diagnostics, and looking for occasional small values among them would require a multiple-testing correction.

5.2 Decimation

A decimated run is requested using `-d` (`--decimation-bits`) and a target number m of kept points: the tool draws a fixed number of samples, and keeps the $\approx m$ samples that survive. Decimating $d = 8$ bits per coordinate on a $u = 20$, $t = 2$ grid shrinks the effective cell space to $2^{(20-8) \cdot 2} = 2^{24}$, so the same two million kept points produce many more collisions than they would on the full grid:

```
$ cargo run -r -F splitmix -- 20 2 2000000 -d 8 -S 1
Generator: SplitMix
Seed: 0x0000000000000001
Running a 2-dimensional collision test sequentially on the upper
20 bits of the full 64-bit output (2000000 points, 32-bit cells,
2000000 memory locations, 0.008 GiB RAM (+4.85%), decimating 8
bits per dimension (~216 candidate samples per kept sample))
u: 20 t: 2 cells: 16777216 expected collisions: 114610.15717447922
(binomial approximation, variance: 97767.77139757853) (effective
cells after decimation: 224)
Scanning 131072000000 samples (decimating low 8 bits per
dimension)...[...] sort...[...] count...[...] 1998618 points done.
114466 p=0.4863214713122816
Test completed in [...] seconds
114466 p=0.4863214713122816
```

The effective density is $2 \times 10^6 / 2^{24} \approx 0.12$: thus, the variance of the number of collisions is noticeably smaller than its mean, and the p -value is computed using the binomial approximation of Section 2.2.

The number of kept points is close to, but not exactly, two million, as each candidate sample is kept with probability 2^{-td} . The null distribution, and thus the p -value, is computed using the actual number of kept points (Section 3.3), whereas the header reports the a priori mean and variance for the nominal number. The price we pay for the smaller cell space is about 2^{16} candidate samples (that is, $t \cdot 2^{16} = 2^{17}$ generator outputs) per kept point.

The option `-c` prints a cumulative p -value at $\lfloor \sqrt{2^{td}} \rfloor$ equally spaced checkpoints, so that one can follow the evolution of the statistic as the sample grows. The auxiliary buffer holding the kept points of each checkpoint is sized using the balls-into-bins bound of Section 3.4.

5.3 Parallel execution

Parallel generation is requested using `-P` (`--parallel`). The number of threads is that of the underlying `rayon` thread pool, which uses all available cores unless the environment variable `RAYON_NUM_THREADS` specifies otherwise. On the plain run above, four threads produce

Table 2: Medium-scale to large-scale runs, all with the default (zero) seed. “Mode” reports repetitions (r), tradeoff bits (b), or decimation bits (d); “Points” is the total number of points (for decimation, the target number of kept points); λ is the expected number of collisions (of coincident spacings for the birthday-spacings test), combined over repetitions, and C is the observed one. The memory parameter is $m = 8 \times 10^9$ for `SplitMix`, `LCG64`, and the $b = 3$ `wyrand` run, and $m = 2 \times 10^9$ for the remaining runs. p -values close to one report *too few* collisions.

Generator	Test	u	t	Mode	Points	λ	C	p -value
<code>SplitMix</code>	collision	64	1	plain	8×10^9	1.73	0	0.82
<code>SplitMix</code>	collision	64	1	$r = 16$	1.28×10^{11}	27.8	0	$1 - 8.8 \times 10^{-13}$
<code>SplitMix</code>	collision	64	1	$b = 2$	3.2×10^{10}	27.8	0	$1 - 8.8 \times 10^{-13}$
<code>SplitMix</code>	collision	64	1	$d = 4$	8×10^9	27.8	0	$1 - 8.8 \times 10^{-13}$
<code>wyrand</code>	collision	64	1	$b = 3$	6.4×10^{10}	111.0	222	1.3×10^{-20}
<code>wyrand</code>	collision	64	1	$r = 16384$	3.28×10^{13}	1776.4	1789	0.39
<code>wyrand</code>	collision	64	1	$b = 7$	2.56×10^{11}	1776.4	3524	5.2×10^{-292}
<code>LCG64</code>	collision	32	2	$b = 3$	6.4×10^{10}	111.0	16	$1 - 1.8 \times 10^{-29}$
<code>MWC128</code>	birthday	36	3	$b = 6$	1.28×10^{11}	1.6	65	8.6×10^{-79}
<code>MWC128</code>	birthday	36	3	$d = 6$	2×10^9	1.6	51	5.6×10^{-57}

```

$ RAYON_NUM_THREADS=4 cargo run -r -F splitmix -- 16 2 10000000 -P -S 1
Generator: SplitMix
Seed: 0x0000000000000001
Running a 2-dimensional collision test using 4 parallel generators
(jump-ahead) on the upper 16 bits of the full 64-bit output
(10000000 points, 32-bit cells, 10000000 memory locations, 0.037
GiB RAM)
u: 16 t: 2 cells: 4294967296 expected collisions: 11632.501259768274
Pass 1/1: [...], 10000000 points, 11754 collisions,
p=0.1310416600017843
11754 p=0.1310416600017843
Test completed in [...] seconds
11754 p=0.1310416600017843

```

We find the same 11754 collisions as the sequential and tradeoff runs. The header reports how the points were generated, that is, how many generators ran in parallel, and how each of them reached the start of its segment: `jump-ahead` for generators with a closed-form jump, and `pre-scan` for the others (Section 4.4).

A run without `-P` reports `sequentially` instead, so a transcript always records the number of generators that were actually in use.

6 Experimental results

In this section we present the results of a few medium-scale to large-scale runs (Table 2), which were performed using version 0.3.1 of `coll-birth`, compiled with Rust 1.98.1, on a Linux workstation with 20 cores and 64 GiB of RAM, using the default (zero) seed. All runs use parallel generation (Section 4.4), so their results do not depend on the number of threads. Times are wall-clock times.

Table 2 contains failures of two opposite kinds: some generators produce too many collisions, and others *too few*, that is, they spread their outputs more evenly than a random sequence would. Failures of the latter kind appear as p -values close to one.

6.1 A bijective generator: SplitMix

The generator `SplitMix`¹⁷ scrambles a 64-bit counter with an invertible function: its 64-bit output thus never repeats within its period, whereas a random sequence repeats values at the birthday rate. In a plain test on 8×10^9 points (the largest number fitting in 60 GiB) we expect only 1.73 collisions, so observing none is not significant.

With the same memory, 16 repetitions, a $b = 2$ tradeoff, or a $d = 4$ decimation all raise λ to 27.8: in all three cases we observe no collision, and the generator is rejected because of the *lack* of collisions ($p = 1 - 8.8 \times 10^{-13}$). The three approaches, however, have very different costs: less than a minute for decimation, about two and a half minutes for the tradeoff, and eight minutes for the repetitions.

6.2 One large test versus many small ones: wyrand

The generator `wyrand` of the `wyhash` family¹⁸ adds a constant to a 64-bit counter, multiplies the counter by its exclusive or with a constant, and folds the two halves of the 128-bit product with an exclusive or. This mixing function is not a bijection, and the collision test shows that it does not behave like a random function either.

At 6.4×10^{10} points ($b = 3$, five minutes) there are 222 collisions against 111 expected ($p \approx 1.3 \times 10^{-20}$), and at 2.56×10^{11} points ($b = 7$) there are 3524 against 1776.4 ($p \approx 5.2 \times 10^{-292}$). In other words, the output of `wyrand` contains about twice as many collisions as a random sequence.

The last two `wyrand` rows are the empirical counterpart of the discussion in Section 3.1: the two runs use the same memory and the same number of generator calls, and have the same mean $\lambda = 1776.4$. However, 16384 repetitions of a test on 2×10^9 points yield 1789 collisions and pass ($p = 0.39$) after 36 hours, whereas a single tradeoff test fails in less than 50 minutes: the excess collisions involve outputs that are far apart in the orbit, which a small test never compares.

6.3 A linear congruential generator that is too good

Multipliers of linear congruential generators are traditionally chosen using the *spectral test* (see Knuth,⁵ §3.3.4), which measures the distance between the hyperplanes covering the lattice of vectors of t consecutive outputs. The distance is normalized into a *figure of merit* $f_t \leq 1$, and larger values are better. The multiplier `0xa5b9ee81534fa94d` of our 64-bit generator (with increment one) comes from the search of Steele and Vigna,¹⁹ and it has an almost perfect $f_2 = 0.977689$.

Nonetheless, on the upper 32 bits of pairs of consecutive outputs, at 6.4×10^{10} points the generator yields just 16 collisions where 111 are expected ($p = 1 - 1.8 \times 10^{-29}$). The reason is that figures of merit measure uniformity, rather than randomness: the generator visits each of its 2^{64} states once per period, and the pair of upper halves of two consecutive outputs is a function of the state with values in 2^{64} cells.

The lattice structure of the pairs is very regular: two pairs can fall in the same cell only if their difference is one of the few short lattice vectors, so the function is closer to a bijection than to a random function, and there are too few collisions. For our multiplier, the short vectors predict about $0.154 \lambda \approx 17$ collisions, in agreement with the 16 observed.

This phenomenon was already observed by L’Ecuyer, Simard, and Wegenkittl⁸ (Section 5.3): in their experiments, multiplicative congruential generators with an excellent lattice structure fail the collision test in two dimensions because of too few collisions, whereas generators with a mediocre lattice structure fail because of too many collisions. Their generators, however, have periods of at most about 2^{40} , and the results they report for the collision test stop at 2^{30} : detecting analogous defects in excellent, real-world 64-bit generators requires a large-scale test such as those made possible by `coll-birth`.

6.4 A multiply-with-carry generator in three dimensions

A multiply-with-carry generator with base 2^{64} , lag one, and multiplier a is essentially equivalent to a multiplicative congruential generator with prime modulus $a \cdot 2^{64} - 1$ and multiplier a ,²⁰ whose figures of merit can thus be computed. For multipliers close to 2^{64} , as those used in practice, f_3 is inherently very poor: for the multiplier `0xffebb71d94fcdaf9` of MWC128, which has 128 bits of state, $f_2 \approx 0.93$, but $f_3 \approx 0.0006$.

The last two rows of Table 2 show that this defect is detectable. With the two-level tradeoff of Section 3.5 ($b = 6$), MWC128 fails the birthday-spacings test on triples of 36-bit coordinates at 1.28×10^{11} points, with 65 coincident spacings where 1.6 are expected ($p = 8.6 \times 10^{-79}$). The run keeps the working set within 60 GiB, where a plain run would need about 2 TB, and examines about 1.6×10^{15} outputs, that is, more than 12 PB, in about five days.

Decimating $d = 6$ bits per coordinate gives the same mean with the same number of calls, as the two techniques are matched when $td = 3b$ (the mean grows with the cube of the number of points). The decimated run yields 51 coincident spacings ($p = 5.6 \times 10^{-57}$) in about 30 hours, as it avoids the overhead of 4096 passes.

6.5 Where the time goes

The tool times separately the generation, sorting, and counting phases of each pass, and the runs of Table 2 show how the cost shifts from one phase to another as the tradeoff grows.

In the plain `SplitMix` run, generating 8×10^9 points (59.6 GiB of u64 cell indices) takes 1.7 s, that is, about 35 GiB/s; sorting them takes about 25 s, and counting the collisions about 1 s, so sorting accounts for about 85% of the 29 s of the run. With $b = 2$, each of the four passes scans 3.2×10^{10} samples to keep a quarter of them: generation grows to about 13 s per pass, whereas sorting the same number of points still takes 22–26 s.

In the $b = 7$ `wyrand` run, each of the 128 passes scans 2.56×10^{11} samples to keep 2×10^9 points: on average, generation takes 16.3 s, sorting 5.8 s, and counting 0.26 s, so generation accounts for 73% of the 48 minutes of the run. In the $b = 6$ birthday-spacings run of MWC128, each of the $64 \times 64 = 4096$ units (a value interval, replayed for a spacing class) generates for about 90 s, and sorts 128-bit cell indices for about 12.5 s: generation accounts for 87% of the five days of the run, and in the decimated run, which sorts just once, for essentially all of it.

In the end, the space/time tradeoff turns a problem of memory into a problem of computation, and computation is the part that parallelizes best: each thread generates its own segment of the orbit independently, with no communication until the points are sorted. Note also that counting, which dominated the running time in our second design (Section 4.5), is now a few percent of the time of the sort.

7 Conclusions

We have described `coll-birth`, a tool that runs collision and birthday-spacings tests on a very large scale using a space/time tradeoff: the cell space is partitioned by the top bits of the combined index, and the test is run in one pass per interval, in a fraction of the memory and without giving up exactness. For birthday spacings, a simple partition does not suffice, as some spacings straddle interval boundaries: we orchestrate instead a two-level tradeoff, in which an inner level computes the exact spacings interval by interval, and an outer level counts their coincidences one class of lower bits at a time.

We also provide decimation, in the sense of O’Neill,¹³ which strengthens the test by acting on the lower bits, and is thus orthogonal to the tradeoff. The fixed-sample formulation shared by decimation and by each tradeoff pass makes it possible to split a run across threads so that a multithreaded execution reproduces the sequential result bit for bit; together with the selection of

the generator at compile time, this makes it practical to test a generator at hundreds of billions of points on a single workstation.

The first takeaway lesson from our experience is that sorting beats hashing for counting duplicates at this scale, and that in-place algorithms are a necessity, rather than an optimization, when memory is the constraint. Moreover, a parallelization reproducing the sequential result exactly is worth its cost, as it turns the sequential code into a test oracle for the parallel one.

The balls-into-bins headroom assumes near-uniform acceptance, so a pathologically structured generator (e.g., a linear congruential generator whose lower bits are periodic) can in principle overflow a per-pass buffer; at present, `coll-birth` stops the run, and reports the overflow as evidence of gross nonuniformity. We leave to future work turning such an overflow into a proper test statistic, which would improve the behavior of the tool on the weak generators it is designed to detect.

Acknowledgments

The author would like to thank Alexey Voskov for a very interesting discussion that stimulated the investigation of space/time tradeoffs for statistical testing.

Conflict of interest

The author declares no potential conflict of interests.

Data availability statement

The source code of `coll-birth` is openly available at <https://github.com/vigna/coll-birth-rs/> and on <https://crates.io/crates/coll-birth> under the Apache License 2.0 or the MIT License. The results in this paper were obtained with version 0.3.1 and the default seed; since parallel runs are faithful (Section 4.4), they can be reproduced on any number of threads.

References

1. Pierre L'Ecuyer and Richard Simard. TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software*, 33(4):22:1–22:40, 2007. doi: 10.1145/1268776.1268777.
2. George Marsaglia. The Marsaglia random number CDROM including the DIEHARD battery of tests of randomness. Technical report, Florida State University, 1995. Available at <https://web.archive.org/web/20160119150146/http://stat.fsu.edu/pub/diehard/cdrom/>.
3. George Marsaglia and Wai Wan Tsang. Some difficult-to-pass tests of randomness. *Journal of Statistical Software*, 7(3):1–9, 2002. doi: 10.18637/jss.v007.i03.
4. Chris Doty-Humphrey. PractRand: Practically random, a C++ library of statistical tests for RNGs. <https://prcrand.sourceforge.net/>, 2018. Version 0.95.
5. Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Boston, 3rd edition, 1998.
6. George Marsaglia. A current view of random number generators. In *Computer Science and Statistics, Sixteenth Symposium on the Interface*, pages 3–10, Amsterdam, 1985. Elsevier.

7. Pierre L'Ecuyer and Richard Simard. On the performance of birthday spacings tests with certain families of random number generators. *Mathematics and Computers in Simulation*, 55(1–3):131–137, 2001. doi: 10.1016/S0378-4754(00)00253-6.
8. Pierre L'Ecuyer, Richard Simard, and Stefan Wegenkittl. Sparse serial tests of uniformity for random number generators. *SIAM Journal on Scientific Computing*, 24(2):652–668, 2002. doi: 10.1137/S1064827598349033.
9. Nicholas D. Matsakis and Felix S. Klock II. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT)*, pages 103–104. ACM, 2014. doi: 10.1145/2663171.2663188.
10. Barry W. Brown, James Lovato, and Kathy Russell. CDFLIB: Library of Fortran routines for cumulative distribution functions, inverses, and other parameters. Technical report, University of Texas, M.D. Anderson Cancer Center, 1994. Available at <https://www.netlib.org/random/>.
11. Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM Journal on Computing*, 20(2):270–277, 1991. doi: 10.1137/0220017.
12. Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–169, 1993. doi: 10.1145/152610.152611.
13. Melissa E. O'Neill. A birthday test: Quickly failing some popular PRNGs. <https://www.pcg-random.org/posts/birthday-test.html>, 2018. Blog post, accessed 2026-08-01.
14. Stéphane Boucheron, Gábor Lugosi, and Pascal Massart. *Concentration Inequalities: A Nonasymptotic Theory of Independence*. Oxford University Press, Oxford, 2013. doi: 10.1093/acprof:oso/9780199535255.001.0001.
15. Martin Raab and Angelika Steger. “Balls into Bins” — a simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science (RANDOM 1998)*, volume 1518 of *Lecture Notes in Computer Science*, pages 159–170. Springer-Verlag, 1998. doi: 10.1007/3-540-49543-6_13.
16. Axelle Piot. voracious_radix_sort: State of the art radix sort algorithms for Rust. https://crates.io/crates/voracious_radix_sort, 2023. Rust crate version 1.2.0.
17. Guy L. Steele Jr., Doug Lea, and Christine H. Flood. Fast splittable pseudorandom number generators. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 453–472. ACM, 2014. doi: 10.1145/2660193.2660195.
18. Yi Wang. wyhash and wyrand. <https://github.com/wangyi-fudan/wyhash>, 2021. Version 4.2, accessed 2026-08-01.
19. Guy L. Steele Jr. and Sebastiano Vigna. Computationally easy, spectrally good multipliers for congruential pseudorandom number generators. *Software: Practice and Experience*, 52(2):443–458, 2022. doi: 10.1002/spe.3030.
20. Raymond Couture and Pierre L'Ecuyer. Distribution properties of multiply-with-carry random number generators. *Mathematics of Computation*, 66(218):591–607, 1997. doi: 10.1090/S0025-5718-97-00827-2.