Generic Deterministic Random Number Generation in Dynamic-Multithreaded Platforms

Stefano Mor[1,2,3,4,⋆], Jean-Louis Roch[1,3,4], and Nicolas Maillard[2]

[1] Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
[2] Instituto de Informática, Univ. Federal do Rio Grande do Sul, Porto Alegre, Brazil
[3] CNRS, LIG, F-38000 Grenoble, France
[4] Inria
{Stefano.Mor,Jean-Louis.Roch}@imag.fr, nicolas@inf.ufrgs.br

Abstract. On dynamic multithreaded platforms with on-line scheduling such as work-stealing, randomized computations raise the issue of reproducibility. Compliant with *de facto* standard sequential Deterministic Random Number Generators (DRNGs) noted `R`, we propose a parallel DRNG implementation for finite computations that provides deterministic parallel execution. It uses the stateless sub-stream approach, enabling the use of efficient DRNG such as Mersenne Twister or Linear Congruential. We demonstrate that if `R` provides fast jump ahead in the random sequence, the re-seeding overhead is small, polylog in expectation, independently from the parallel computation's depth. Experiments benchmark the performance of randomized algorithms employing our solution against the stateful DRNG DotMix, tailored to the Cilk Plus dynamic multithreading runtime. The overhead of our implementation `ParDRNG<R>` compares favorably to the linear overhead of DotMix re-seedings.

Keywords: Random Numbers, Dynamic-Multithreading, Generic, DotMix, Cilk.

1 Introduction

Deterministic Random Number Generators (DRNGs), stateful abstractions that generate a random number stream from a given initial seed, provide reproducibility to random experiments and are useful in the debug of randomized algorithms.

Dynamic multithreading, defined by Leiserson *et al.* [1] as a synonym of task parallelism, is a processor-oblivious parallel programming model where keywords enable parallelism on the serial code without reference to the number of available processors. A scheduler, such as non-blocking randomized work stealing, manages the execution. These platforms guarantee deterministic results, despite the intrinsic non-determinism introduced by the scheduler, except if the result relies on stateful components. Such is the case of DRNGs. State-of-the-art DRNGs for dynamic multithreaded environments overcomes this by fixing a tailored generation algorithm, trading-off abstraction of implementation properties (*e.g.*, randomness, cryptography, regularity, *etc.*) for performance.

Contribution. As an alternative to fixed implementations for parallel DRNGs, we propose a generic parallel API called `ParDRNG<R>` that ensures deterministic parallel executions on dynamic multithreading platforms. `ParDRNG<R>`

⋆ Scholarship holder CNPq - Brazil, Eiffel Laureate - French Ministry of Foreign Affairs.

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 427–438, 2014.

uses as underlying engine a sequential DRNG `R` and inherits its qualities without compromising parallel efficiency. Its main insight is the use of `R`'s capability of "jumping-ahead" in the generated stream to ensure determinism; the application partitions the random sequence on-the-fly among the parallel tasks, and each task re-seeds its DRNG through a jump-ahead to generate only random numbers belonging to its subsequence. To ensure efficiency, these re-seeds occur only when triggered by a steal operation performed by the work stealing scheduler. We prove this method to introduce an overhead upper-bounded by the parallel work (*work-efficiency*) even when efficient jump-ahead is absent, and that the theoretical re-seeding overhead is polylog (*work-optimality*) whenever `R` provides at least polylog jump operations on the random sequence.

Related Works. Coddington [2] enumerates a useful array of techniques to parallelize conventional DRNGs, like "leapfrog" (cyclic partition among processors) and "sequence splitting" (block partition among processors) but these are not processor-oblivious. On the other hand, counter-based DRNGs [3] have excellent statistical properties and can be used in deterministic parallel executions. However, considering performance, each random generation from the counter requires an operation equivalent to re-seeding, and thus a linear overhead. The polylog overhead of `ParDRNG<R>` compares favourably. Moreover, `R` can itself use counter-based generators (*e.g.*, AES). The "re-seed through jump" strategy is also discussed by Haramoto *et al.* [4], which argued in favor of parallel programs to build a fast jump-ahead algorithm over Mersenne Twister, what resulted in the implementation of SIMD-oriented Fast Mersenne Twister (SFMT). This is also the case of L'Ecuyer's RNGStreams library (on the top of its MRG32k3a generator [5]). Both approaches deliver a jump with high constant cost, compensated by the large range skipped — which, contrary to `ParDRNG<R>`, is defined at compile time. Languages like Haskell also follow this sub-stream approach, offering their own splittable generators to the programmer. All these implementations offer a static set of properties, since the generation algorithm is fixed.

Comparison. `ParDRNG<R>` is compared performance-wise with the stateful, counter-based DRNG DotMix [1], written in C++ for the Cilk Plus dynamic multithreading platform. DotMix supports infinite simulations, but requires any execution to match the same directed acyclic task graph (DAG). `ParDRNG<R>` supports non-deterministic DAGs, but only finite computations. The polylog overhead of `ParDRNG<R>` compares favorably with the linear overhead of DotMix re-seedings. Also contrary to DotMix, the programmer may choose different underlying engines providing different sets of properties. *E.g.*, our approach can be made secure by using underlying cryptographic generators.

Outline. Definitions for DRNGs, with general interface and required complexity are at Sec. 2. The main reasoning over work-efficient and work-optimal generic algorithms and its applicability to random number generators are on Sec. 3. Experiments and performance comparison with DotMix are reported on Sec. 4. Concluding remarks are on Sec. 5.

All the relevant data structures and algorithms are written in C++11 with template facilities, aiming reproducibility and pragmatic analysis, although knowledge on the language is not mandatory.

2 Sequential DRNGs and Generic Interface

A DRNG acts as a deterministic stream that provides new random numbers based on its current *state*. The initial state is given by a *seed* value. Random streams have a finite orbit, called its *period*, which corresponds to a sequence of numbers that will eventually be repeated over successive generations.

Two DRNG classes are distinguished to generate the stream $\langle r_n \rangle$ from a function r with finite output set and good statistical properties [3]. *Conventional* DRNGs iterate $r_n = r(r_{n-1})$ (*e.g.*, Mersenne Twister [6], Linear Congruential, Tausworth [7], BBS [8]); while *counter-based* DRNGs independently compute $r_n = r(n)$ (*e.g.*, Philox [3], DotMix). Thus, counter-based are parallel, but conventional DRNGs appear serial: implementations benefit from the previous value r_{n-1} to efficiently generate r_n with less overhead than counter-based ones. In addition, some conventional DRNGs provide efficient jump-ahead over multiple output values in less time than it takes to repeatedly invoke r [3].

A generic interface for DRNGs is now defined in order to set a common notation and complexity requirements for our parallel algorithms. It is assumed that the DRNGs work around integer types, for compatibility.

Function `NEXT` . Input: a reference to a DRNG. Return: the next random number produced by the DRNG– sets its internal state. Complexity is $\Theta(1)$.

Function `SEED`. Input: a reference to a DRNG and optionally an unsigned integer serving as the seed for the generator. Return: generator's seed after the call. Each call with the second parameter re-seeds the generator and resets the internal state. Complexity is $\Theta(1)$.

Function `CLONE` . Input: two DRNG references, source and destination. Return: nothing. Copies the state from source to destination. Complexity is $\Theta(1)$.

Function `JUMP`. Input: a reference to a DRNG and a natural number n. Return: nothing. Performs a *jump-ahead* operation, advancing the generator's state as if `NEXT` was called n times. Different constraints on the DRNGs usually allow faster implementations. Thus, the cost of jump is modelled as three variations of a function $\delta : \mathbb{N} \to \mathbb{N}$.

- *Linear* : $\delta(n) = O(n)$. Direct implementation. It requires no extra memory in order to operate, what may be prohibitive for other versions. Trade-offs between memory and space are considered by Haramoto *et al.* [4].
- *Log* : $\delta(n) = O(\log_2 n)$. Could be implemented, *e.g.*, by exponentiation over current state, like the BBS generator [8].
- *Const* : $\delta(n) = O(1)$. Could be implemented, *e.g.*, by extending the Log version through pre-computation of the required powers in its finite period.

Function `GENERATE` . The kernel of this paper. Input: a (seeded) DRNG of type `R` and non-negative memory range of size n. Output: a sequence of n numbers generated by the DRNG filling the range. Its reference sequential implementation is

```
template <OutputIterator I, DRNG R>
void GENERATE (I first, I last, R& r) {
    while (first != last) *(first ++) = ValueType<I> (NEXT (r)) ;
}
```

The generic parameter `R` denotes an arbitrary sequential DRNG. For this reason the interface is named `ParDRNG<R>`. Parallel implementations of `Generate` are detailed that do not presuppose thread-safeness for the functions provided by `R`.

3 Parallel DRNGs and Analysis

Dynamic-multithreading is examined through task-based computations. A *task* is an indivisible set of machine instructions. Two tasks can be executed in parallel unless related by a sequential dependency. Tasks are executed by *workers* (threads in this paper). A worker is *inactive* when it is idle and *active* otherwise. A *top* is a totally ordered integer time stamp regarding the execution of a parallel program. Current top is denoted s, previous top $s-$ and next top $s+$. The top before the execution is 0 and first top is 1. When a synchronization between worker i and j occurs at s, it is noted by $s(i, j)$. The platform provides a *scheduler*, an algorithm that decides which worker executes which task at each top.

As recurrent notation, a parallel algorithm operates over P workers and input size n and has work $T(n) = W(n)+V(n)$, where $W(n)$ is the sequential work and $V(n)$ is the parallelism overhead. The total work with an unbounded number of processors is the parallel algorithm's depth.

The discussion is contextualized over Cilk Plus' dynamic multithreading platform [9], the most recent incarnation of Cilk [10]. It provides a fork-join abstraction where user threads are spawned as parallel procedures (keyword `cilk_spawn`) and joined in a blocking way (keyword `cilk_sync`). This implies a processor-oblivious model of computation.

Cilk Plus assigns continuations (ready tasks) to workers through a randomized work stealing scheduler [10]. It is implemented as a collection of worker threads with a double-ended queue (deque) with two extremes, a front and a back. Parallel continuations produced by the worker are placed in its deque's front. Idle workers with an empty deque keep randomly selecting victim workers until choosing one with a non-empty deque. In this case, it steals the continuation at the deque's back. Idle workers with a non-empty deque remove and execute continuations from its deque's front. The runtime stops when all workers are idle. The main invariants are the fact that a stolen task is executed without entering the deque (prevents deadlocks) and the spawned task is immediately executed, while the spawner goes to the deque's front (depth-first execution). This model is considered in the implementations that follow.

3.1 Work-Efficiency

A parallel algorithm is defined to be work-efficient *iff* $T(n) = W(n) + V(n) = O(W(n))$, *i.e.*, its overhead is not asymptotically larger than the work parallelized. Consider a naive implementation of parallel generate:

```
1  template <ForwardIterator I, DRNG R>
2  void PARALLEL_GENERATE (I first, I last, R& r0) {
3      DistanceType<I> n = distance (first, last) ;
4      if (n < parallel_grain ()) return GENERATE (first, last, r0) ;
5      halve (n) ;
6      R r1 = r0 ; // CLONE
7      JUMP (r1, n) ;
8      I middle = successor_n (first, n) ;
9      cilk_spawn PARALLEL_GENERATE (first, middle, r0) ;
10     cilk_spawn PARALLEL_GENERATE (middle, last, r1) ;
11 }
```

Let n' be the parallel threshold returned by `parallel_grain()`. Also, let $\alpha = \Theta(1)$ be the work performed by `NEXT` and $\beta = \Theta(1)$ be the same for the assignment of DRNGs (function `CLONE`, Sec. 2). Thus, regarding DRNG operations,

naive parallel generate has total work $T(n) = \alpha n'$ when $n < n'$. Otherwise, $T(n) = \beta + \delta(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$. In closed form (only for powers of two, which maintain asymptotic behavior by the Akra-Bazzi Method): $T(n) = \alpha n + \beta(n-1) + \sum_{i=0}^{\log_2(n)-n'-1} 2^i \delta\left(n/2^{i+1}\right)$. Subtracting from both sides $W(n) = \alpha n + \beta$, delivers the overhead: $V(n) = \beta(n-2) + \sum_{i=0}^{\log_2(n)-n'-1} 2^i \delta\left(n/2^{i+1}\right)$ (1) determined by δ. Both Const and Log versions of δ are work-efficient because of its overhead $O(n)$ when applied in Equation (1); while Linear version has overhead $O(n \log_2 n)$, and thus is not.

It is possible to reduce the number of jump-ahead operations when the spawned routines run sequentially. Jumps are only performed to guarantee that determinism is preserved when the recursive calls operate in parallel. Since parallelism only unfolds in the presence of steals, execution can jump exclusively when a continuation is stolen; otherwise the original DRNG is used. This tactic effectively moves the determinism overhead to computation's depth, in a fashion inspired by the *work-first* principle of Cilk's scheduler [10].

Some meta-programming is applied over the work stealing scheduler, still at application level: an extra parameter is appended to the recursive function call with the id of the worker that invoked the method originally. Current worker's id (obtained by calling `__cilkrts_get_worker_number()` through wrapper (generic) function `me ()`) is compared to caller to determine whether actual parallel execution is in course. The code is written using tail recursion optimization, replacing the final recursive call by a loop. Also, in order to use the same DRNG in absence of steals, the DRNGs are passed by reference and cloned only whenever needed. This implies an occasional cancellation of the tail recursion optimization, but only when a successful steal takes place:

```
1  template <ForwardIterator I, DRNG R, Natural N>
2  void PARALLEL_GENERATE (I first, I last, R& r0, N worker = me ()) {
3      DistanceType<I> n = distance (first, last) ;
4      while (n > parallel_grain ()) {
5          halve (n) ;
6          I middle = successor_n (first, n) ;
7          R r1 = r0 ; // CLONE
8          cilk_spawn PARALLEL_GENERATE (first, middle, r0, worker) ;
9          if (worker != me ()) { // steal
10             JUMP (r1, n) ;
11             return PARALLEL_GENERATE (middle, last, r1, me ()) ;
12         }
13         first = middle ;
14     }
15     return generate (first, last, r0) ;
16 }
```

3.2 Analysis

As demonstrated by Blumofe *et al.* [11], the expected number of total steal attempts for a parallel execution with depth T_∞ and scheduled by randomized work stealing is $O(PT_\infty)$. Nevertheless, the performance of our method is bounded by the number of successful steals, *i.e.*, the steal attempts over non-empty deques. Next we employ a counting technique that estimates the size of an specific subset of the performed steal attempts (*e.g.*, successful ones) and does not depend on execution's depth. This generalizes the bound to non-deterministic DAGs.

First, let each worker $1 \leq i \leq P$ to have associated a *local counter* φ_i, and their union to be the *global counter*:

Definition 1 (Local Counter). *Let $\mathcal{S}$ be the poset of all events during a parallel execution (identified by the respective tops). A local counter is any function $\varphi_i : \mathcal{S} \to \mathbb{R}^+$ where: (1) If i is inactive at $s \in \mathcal{S}$, then $\varphi_i(s) = 0$. (2) If i is active at $s \in \mathcal{S}$, then $\varphi_i(s) > \varphi_i(s-)$.*

Definition 2 (Global Counter). *Let Σ be a (possibly non maximal) subset of $\mathcal{S}$ containing only synchronization operations. A global counter is any function $\varphi : \mathcal{S} \to \mathbb{N}^P$ with $s \mapsto (\varphi_1(s), \cdots, \varphi_P(s))$ where: (1) Function φ_i is a local counter for worker i. (2) If $s(i,j) \in \Sigma$, then $\min(\varphi_i(s+), \varphi_j(s+)) \geq \min(\varphi_i(s-), \varphi_j(s-)) + 1$*

Henceforward all successful steals are considered to be the interesting synchronizations, *i.e.*, the ones in Σ. The local counter $\varphi_i(s)$ is the size of worker i's deque at s. The global counter is the total number of successful steals. Limit M is defined as the maximum size of any deque during computation.

An upper-bound for all local counters also bounds the global counter:

Lemma 1. *During a randomized work stealing execution over P workers, let Σ be subset of steal operations, φ be a global counter over Σ. Also let u be a random variable whose value is the number of occurrences of the steals in Σ and $\mathbb{E}(u)$ be its expected value. If there is a constant M such that $\varphi_i(s) \leq M$ for all $1 \leq i \leq P$ active at s, then $\mathbb{E}(u) \leq M(P-1)H_{(P-1)}$, where $H_{(P-1)} = \sum_{k=1}^{P-1} 1/k$ is the harmonic number, and $\pi^2(P-1)^2/6$ is the expected variance.*

Proof (Sketch). First, any synchronization operation is named "local step". Let $\phi_{\min}(\varphi, s)$ be a function that returns the value of the minimal non-zero local counter at top s. A local counter is increasing while i is active. A round of consecutive local step where each processor has been victim of at least a steal request in Σ is named a "global step". Yet a global step is a coupon collector's problem, thus the expected number of consecutive successful steals in Σ is $(P-1)H_{(P-1)}$, with variance $\pi^2(P-1)^2/6$. By Def. 2 the number of such steps is less than M which states $\mathbb{E}(u) \leq M(P-1)H_{(P-1)}$.

Fig. 1 shows a snapshot at top s of a global counter from Def. 2 (bounded by M) and function $\phi_{\min}$ used in the proof sketch of Lemma 1.

The cost of a jump-ahead operation was modelled to be a function of current sub-range's size. Thanks to Lemma 1, the next corollary bounds in expectation the overhead introduced by each successful steal of a given range size:

Corollary 1. *Let u_m be a random variable whose value is the number of occurrences of the steals of size m in Σ. Then, $\mathbb{E}(u_m) \leq (P-1)H_{(P-1)}$.*

Proof (Sketch). Once a steal of size m is suffered, it cannot be suffered again until processor becomes idle (size is strictly decreasing). Thus, for any size m, the maximum M is 1. The remaining follows directly from Lemma 1.

`JUMP`'s overhead is bounded by summing the costs of different u_m. Since half of the range is put at deque's front at each spawn, there are $\log_2 n$ different steal sizes to appear, minus size n. Therefore, the expected overhead is: $V(n) = (n-1)\beta + H_{(P-1)}(P-1)\sum_{i=0}^{\log_2(n)-1} \delta(2^i)$. For a fixed P the expected overhead is $O(n)$ when using the Linear version of δ. Thus, in expectation, work-efficiency is always assured.

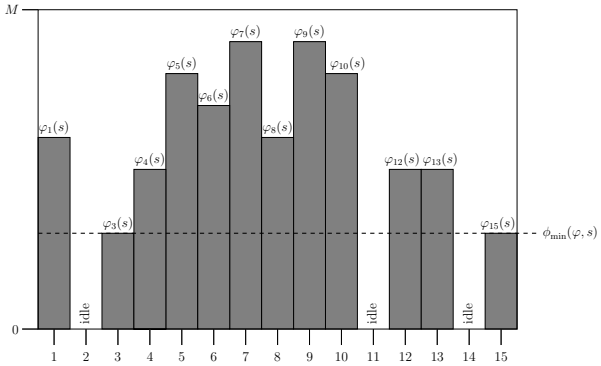

Fig. 1. Example of a Global Counter at top s. Here, $P = 15$ and each active worker i has an value $\varphi_i(s)$. In this example, $\phi_{\min}(\varphi, s) = \varphi_3(s) = \varphi_{15}(s)$. Workers 2, 11, an 14 are inactive (idle); they are not accounted in the calculus of $\phi_{\min}(s)$, and have $\varphi(s) = 0$.

The proof considers a loose bound of one idle worker per top. Nevertheless, the local steps are generally performed in parallel, mitigating the $P-1$ factor. Also, as $P \to \infty$, the harmonic number $H_{(P-1)}$ approximates $\log(P-1)+\gamma+1/2(P-1)+o(1)$, where $\gamma \approx 0.58$ is the Euler-Mascheroni constant. Indeed, asymptotically for large value of P, Theorem 3 in Tchiboukdjian *et al.* [12] states that the expected total number of steals is asymptotically less than $3.65 \cdot M \cdot (P-1)$.

3.3 Work-Optimality

A parallel algorithm is defined to be work-optimal *iff* $V(n) = O(\log_2^{\mathrm{O}(1)} n)$.

Our technique can be refined in order to obtain work-optimal parallel generation. The problem may be reduced to eliminate the fixed overhead introduced by `CLONE`. In order to track the quantity of random numbers generated until a given execution point, a counter is used, which is passed (by copy) as parameter to recursive calls. This eliminates unnecessary DRNGs copies, in exchange for paying the price of longer jumps. Nevertheless, the jumps are mitigated by parallelism and "cheap" when the DRNG provides polylog time jump-ahead. The algorithm also relies on the polymorphic behavior of function *seed* described at Sec. 2 and a seed constructor:

```
1  template <ForwardIterator I, DRNG R, Natural N0, Natural N1>
2  void PARALLEL_GENERATE (I first, I last, R& r0, N0 worker = me (), N1 hist = 0) {
3      DistanceType<I> n = distance (first, last) ;
4      while (n > parallel_grain ()) {
5          halve (n) ;
6          I middle = successor_n (first, n) ;
7          cilk_spawn PARALLEL_GENERATE (first, middle, r0, worker, hist) ;
8          hist += n ;
9          if (worker != me ()) { // steal
10             R r1 (SEED (r0)) ; // seed constructor
11             JUMP (r1, hist) ;
12             return PARALLEL_GENERATE (middle, last, r1, me (), hist) ;
13         }
14         first = middle ;
15     }
16     return generate_seq (first, last, r0) ;
17 }
```

Now we are able to cut off the $\beta(n-2)$ term from on Equation (1). Even the more expensive `JUMP` calls are yet upper-bounded by the most expensive

possible jump: $H_{(P-1)}(P-1)\sum_{i=0}^{\log_2(n)-1}\delta(n-n/2^{i+1})$. The cost of call to seed constructor per successful steal is added, but it is assumed to be a small constant. Now work-optimality for Const and Log versions can be guaranteed, because its overhead results in $O(\log_2^{O(1)} n)$, although work-efficiency for Linear version is lost, since it results in an overhead of $O(n \log_2 n)$.

4 Performance Results

This section provides experimental evidence that the asymptotic limits shown previously do not excessively penalize the execution with its hidden constants and whether they are competitive against Cilk Plus' parallel DRNG, DotMix [1].

DotMix relies on *pedigrees*, thread-unique numerical labels, a feature its authors persuaded Intel to include in its Cilk Plus implementation. A given reference to a global DotMix generator compresses the pedigree and then "RC6-mixes" (hashes) the result with a small collision probability. To maintain pedigrees on the runtime overcharges it with less than 1% overhead. DotMix show statistical quality rivaling (with high variance) the one of Mersenne Twister upon the Dieharder random number test suite.

All experiments were performed using computer "Turing" from the Group of Parallel and Distributed Computing of the Universidade Federal do Rio Grande do Sul (Brazil): Linux 3.2.0-40-generic #64-Ubuntu SMP x86_64. CPUs Intel Xeon X7550 2GHz ×32 (2 thread per core), Caches d32K + i32K/256K/18432K. Mem. Total: 132,018,988 kB. Intel's ICPC 2013 compiler with O2 optimizations is used because it is currently the only compiler that supports Cilk Plus' pedigrees. Other relevant software are Cilkpub 1.03 (for DotMix) and Boost C++ Libraries 1.55. Sources are available in `http://www.inf.ufrgs.br/~sdkmor/Europar2014/`.

Three sequential DRNGs from Boost C++ serve as the underlying engine of the generic scheme: Mersenne Twister 19937 (MT19937) [6], Linear Congruential (Rand48), both over 64-bit integers, and Tausworth Generator (Taus88) [7], over 32-bit integers. The only Boost generator that implements a jump operation in log time is Rand48, the others executing in linear time. A Blum Blum Shub (BBS) [8] crypto-secure generator over 512-bit integers with logarithmic time jump was also implemented. In all tests, work-optimal algorithms are used with Rand48 and BBS and the work-efficient versions with the others.

There are four test algorithms: Generate, Introsort, Maximal Independent Set (by Luby's Method), and Fibonacci, designed to evaluate performance in an increasing level of adversity against our methods. The algorithms run for a number of workers $1 \leq P \leq 32$ as well as a sequential version. In order to provide statistical confidence the pointed plots are the means of 50 executions for each P and sequential version, lying within a 95.45% confidence interval. The standard deviation is at worst case under 8% of the mean, a reasonable range for randomized algorithms. T_s (resp. T_P) denotes the sequential time (resp. parallel time on P processors) with DRNG `R` (resp. `ParDRNG<R>`). Yet T_1 is the time of `ParDRNG<R>` scheduled on one processor.

The comparison criteria is total execution time. Since the algorithms do not have a common sequential implementation (because of different implementations of the generator components), speedup and efficiency measurements are not

Table 1. Average time (in milliseconds, rounded up) of parallel algorithms' execution. Shown sequential time T_s and parallel times T_1 and T_2.

DRNG	Generate			Intro Sort			MIS			Fibonacci		
	T_s	T_1	T_2	T_s	T_1	T_2	T_s	T_1	T_2	T_s	T_1	T_2
Rand48	559	529	268	5649	5730	2994	39	67	43	17	194	97
Taus88	703	660	1033	6132	6412	3661	38	67	45	30	193	146
MT19937	877	901	611	6451	6577	3680	38	66	43	30	327	199
DotMix	4201	1713	863	6227	9798	5217	51	67	42	129	389	195
BBS	25954	25602	13006	6316	6424	3503	149	182	102	701	910	455

meaningful when compared against each other, since a slow sequential implementation may wrongly boost the results. This way, we take out the unfairness of comparing relative speedups, but are use it to show anomalies in sequential executions.

In fact, some DotMix benchmarks running in sequential showed unusual measurements for T_s and T_1, but are as expected for T_2 and above. Thus, for clearness of comparison, these execution times are displayed separately; measurements on T_s, T_1, and T_2 are in Tab. 1 and measurements for T_P with $P > 2$, are in Fig. 2. The unusual behavior of DotMix is contextualized within each benchmark. Highlights on the implementations and reviews over the results follow.

Generate. Implementation of `PARALLEL_GENERATE`. Generates 10^8 64-bit random numbers in parallel. The sequential version for all DRNGs is a for loop calling method `NEXT`. The parallel version of DotMix is a call to its own `fill_buffer` function, implemented with the same tail-recursion optimization of our codes, with parallel a threshold of 2,048. Target implementation has the same grain size for comparison fairness. Boost generators and BBS have a minor difference between T_s and T_1, with BBS being much slower because of its extensive use of integer modulus. DotMix, has a T_1 that is $2.45\times$ faster than its T_s. Since DotMix is projected with a parallel-first principle, `fill_buffer` is optimized regarding pedigree initialization (scope bounding), which is mandatory in order to generate deterministic results, introducing large sequential overhead, what does not affect `ParDRNG<R>`. A speedup comparison between T_1 and T_2 shows Rand48 (work-optimal), BBS, and DotMix with ≈ 1.97 of speedup while work-efficient MT19937 has ≈ 1.47 of speedup. Taus88, work-efficient and 32 bits, has speed-down of ≈ 0.63. DotMix scales until $P = 11$ processors, being better than MT19937 for $P > 4$ processors (it scales up to 6 processors). DotMix is never better than Rand48. Taus88 does not profit at all from `ParDRNG<R>`, probably due to 32 to 64-bit casting. Even BBS is faster for 26 or more processors. Overall we are competitive with DotMix for fast underlying generators.

Randomized Introsort. STL's sort, it is a quicksort algorithm that is switched to heapsort whenever its tree depth goes beyond $2\log_2 n$. We use a modified partition procedure to always divide the interval by half for comparison fairness between DRNGs. The pivots are generated in an "online" fashion, as they are needed. To determine how many terms are to be jumped, it is supposed that each recursive call will advance the generator as much as the size of the subsequence it takes as input. This implies an "over-estimation overhead", because for under threshold instances the algorithm is switched to a non-randomized sort and yet, the DRNGs need to be advanced accordingly to the subsequence. DotMix, because of its use of pedigrees, is not implemented with this overhead. We

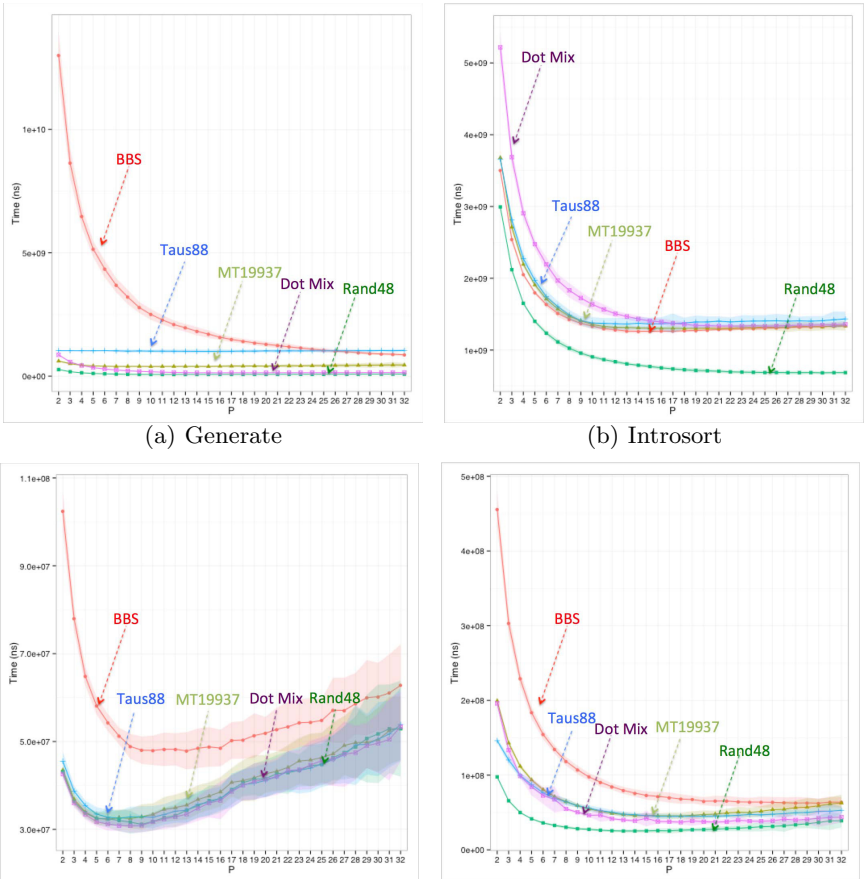

Fig. 2. Average time (in nanoseconds) of parallel algorithms' execution. Shown parallel times from T_2 to T_{32}. The respective colored areas around the points are the confidence interval of 95.45%.

sort 10^8 integers. All generators have $T_s \approx T_1$, except DotMix, that has large overhead $T_1 \approx 1.58T_s$ without optimized `fill_buffer`. Indeed, until $P = 13$ DotMix has the worst performance, even when comparing to BBS, whose slow performance seems to be mitigated by the work-optimal implementation, placing it at the same level and sometimes better than its work-efficient rivals. For $P > 13$ DotMix is at most statistically equal to the work-efficient implementations. Rand48, being fast and work-optimal, is the incontestable winner. Taus88 has a significant gain, since no type casting is necessary.

Maximal Independent Set. Implementation of Luby's method, it is divided in three steps, repeated until the input is marked as empty: (1) select nodes with probability $1/2^i$, where i is the node's degree; (2) unselect lowest degree node of two neighbor selected nodes; (3) move the remaining selected nodes to the MIS and removes its neighbors from input. Steps (1) and (2) are performed in parallel for each node, but step (2) only executes after (1). We use random numbers for the

probabilistic selection in step (1), but the parallel generate function is initialized by a step (0) to generate random numbers in an "off-line" fashion at each round – to the highest level of over-estimation. To provide comparison fairness, the same numbers are selected despite a given generators output. The input is a grid graph with 10^6 nodes. The implementation was written to have irregular scalability: at each step a worker may have assigned a node already marked as unselected, performing no useful work. For small P this behavior eliminates node removal operations, but the parallel performance degradetes fast for larger values. For the fixed values we generate and for the selected input graph, performance loss begins between 6 and 8 workers. When considering both this highly irregular scalability and the maximun level of over-estimation the non-secure DRNGs have the same statistical performance — with larger confidence interval due to the other non-DRNG operations the algorithm performs – while BBS penalizes execution because of its integer modulus operations not being mitigated by online generation.

Fibonacci. A randomized recursive calculus of 30th Fibonacci term that uses three random numbers (before, after and between the recursive calls) and adds them to the recursive sum. As in Introsort, the random numbers are also generated "online", as needed. We use it to illustrate the weakness of our design (it is also a weak point for DotMix because of its depth [1]); the distance of jump is not calculated in constant time, because although we are able to calculate how much previous calls will advance the main generator, this calculus involves computing how many nodes the tree will spawn. This is as much computational work compared to the computation being performed. We used the fast doubling Fibonacci algorithm to mitigate this cost. This decrease of arithmetical work prevents the randomized algorithm to be work-optimal. For this algorithm, DotMix is statistically paired with the work-efficient implementations, although slightly faster for $P > 5$. Taus88 is nearly always better than MT193937, which reinforces its previous improvements for online algorithms. Rand48 is the best until $P = 25$, when it becomes statistically equal to DotMix. For the same range BBS is the worst, being statistically equal to MT19937 afterwards.

5 Concluding Remarks

Despite the fact that we rely on Cilk Plus to implement our designs, our scheme is not dependent on it. Its coding is simple to be written in another dynamic multithreaded environment and the theoretical analisys does not rely on a fixed execution's depth. This implies correctness even on the presence of a non-deterministic DAG, such as those on adaptive algorithms [13].

We have significant performance gains as described in Sec. 4. We are competitive with DotMix for off-line generation algorithms and generally faster with online generation and fast underlying generators. With our generic scheme we are able to choose the desired point between quality and speed of several DRNGs. In addition, it is possible to be drastically more performatic than DotMix or other parallel DRNGs with fixed implementations by selecting underlying DRNGs whose generated sequence is especially effective for a particular application.

An hybrid solution of our approach and DotMix is compelling. However, because DotMix does not have an equivalent to jump-ahead operation, the linear version becomes mandatory. In our tests, this approach was more than 10×

slower than SFMT, a 128bit generator. As shown at Tab. 1, DotMix is faster when using its own internal generate function. However, maybe there is some optimization inside DotMix to allow it. We plan to verify it as future work. Also, we plan to extend the jump on steals technique to numerical algorithms.

In `ParDRNG<R>` the number of required random numbers must be known *a priori* to the computation. This is a strong limitation to our method. There is, however, a range of algorithms that are suited to it besides direct parallel generation, such as randomized sorts, randomized graph generation, randomized genetic algorithms (crossing over), *etc*. Additionally, one may overcome this limitation by guessing large non-overlapping ranges between the different workers, thus enabling algorithms to not know exactly how many numbers they will need in runtime, given an upper-bound. Combining over-estimation and polylog jumps mitigates largely the overheads. This is similar to what is done, for instance, by SFMT [4] and RNGStreams [14].

References

1. Leiserson, C.E., Schardl, T.B., Sukha, J.: Deterministic parallel random-number generation for dynamic-multithreading platforms. In: Proc. of PPoPP 2012, pp. 193–204. ACM, New York (2012)
2. Coddington, P.: Random number generators for parallel computers. The NHSE Review 2 (1997)
3. Salmon, J.K., Moraes, M.A., Dror, R.O., Shaw, D.E.: Parallel random numbers: As easy as 1, 2, 3. In: Proc. of SC 2011, pp. 16:1–16:12. ACM, New York (2011)
4. Haramoto, H., Matsumoto, M., L'Ecuyer, P.: A fast jump ahead algorithm for linear recurrences in a polynomial space. In: Golomb, S.W., Parker, M.G., Pott, A., Winterhof, A. (eds.) SETA 2008. LNCS, vol. 5203, pp. 290–298. Springer, Heidelberg (2008)
5. Fischer, G.W., Carmon, Z., Ariely, D., Zauberman, G., L'Ecuyer, P.: Good parameters and implementations for combined multiple recursive random number generators. Oper. Res. 47(1), 159–164 (1999)
6. Matsumoto, M., Nishimura, T.: Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. 8(1), 3–30 (1998)
7. L'Ecuyer, P.: Maximally equidistributed combined tausworthe generators. Mathematics of Computation 65(213), 203–213 (1996)
8. Blum, L., Blum, M., Shub, M.: A simple unpredictable pseudo random number generator. SIAM J. Comput. 15(2), 364–383 (1986)
9. Intel Corporation: Intel cilk plus language specification (2013)
10. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. In: Proc. of PLDI 1998, pp. 212–223. ACM, New York (1998)
11. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM 46(5), 720–748 (1999)
12. Tchiboukdjian, M., Gast, N., Trystram, D., Roch, J.-L., Bernard, J.: A tighter analysis of work stealing. In: Cheong, O., Chwa, K.-Y., Park, K. (eds.) ISAAC 2010, Part II. LNCS, vol. 6507, pp. 291–302. Springer, Heidelberg (2010)
13. Traoré, D., Roch, J.-L., Maillard, N., Gautier, T., Bernard, J.: Deque-free work-optimal parallel stl algorithms. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 887–897. Springer, Heidelberg (2008)
14. L'Ecuyer, P., Simard, R.J., Chen, E.J., Kelton, W.D.: An object-oriented random-number package with many long streams and substreams. Operations Research 50(6), 1073–1075 (2002)