Symposium on High-Level Parallel Programming & Applications
HLPP 2021

A Deterministic Portable Parallel Pseudo-Random Number Generator for Pattern-Based Programming of Heterogeneous Parallel Systems

August Ernstsson · Nicolas Vandenbergen · Jörg Keller · Christoph Kessler

Abstract SkePU is a pattern-based high-level programming model for transparent program execution on heterogeneous parallel computing systems. A key feature of SkePU is that, in general, the selection of the execution platform for a skeleton-based function call need not be determined statically. On single-node systems, SkePU can select among CPU, multithreaded CPU, single or multi-GPU execution. Many scientific applications use pseudo-random number generators (PRNGs) as part of the computation. In the interest of correctness and debugging, deterministic parallel execution is a desirable property, which however requires a deterministic parallel pseudo-random number generator. We present the API and implementation of a deterministic, portable parallel PRNG extension to SkePU that is scalable by design and exhibits the same behavior regardless where and with how many resources it is executed. We evaluate it with three probabilistic applications and show that the PRNG enables scalability on both multi-core CPU and GPU resources, and hence supports the universal portability of SkePU code even in the presence of PRNG calls, while source code complexity is reduced.

Keywords skeleton programming · parallelizable algorithmic pattern · heterogeneous system · GPGPU · deterministic parallel pseudo-random number generator

August Ernstsson · Christoph Kessler
PELAB, Dept. of Computer and Information Science
Linköping University, Linköping, Sweden
E-mail: <firstname>.<lastname>@liu.se

Nicolas Vandenbergen
Inst. f. Adv. Simulation
Jülich Supercomputing Center, FZ Jülich, Germany
E-mail: n.vandenbergen@fz-juelich.de

Jörg Keller
Faculty of Mathematics and Computer Science
FernUniversität in Hagen, Hagen, Germany
E-mail: joerg.keller@fernuni-hagen.de

1 Introduction

For the foreseeable future, computer systems for performance-demanding application domains such as HPC, machine learning and image processing, will continue to be characterized by multi-/many-core parallelism and heterogeneity. Faced with the increasing slowdown of Moore's Law, a "Cambrian explosion" of computer architectures is foreseen [16] that will continue to introduce new CPU and GPU architectures and entirely new accelerator types at a fast pace to sustain future hardware performance growth, while at the same time an increasing share of performance growth needs to come from both application and system software improvements. This imposes a challenge on the software side: How can we support the creation of truly portable, future-proof software that is high-level yet can efficiently leverage the hardware resources of today's and tomorrow's heterogeneous parallel architectures without permanent rewriting and re-optimization?

The skeleton programming approach [3, 4] is a powerful and programmer-friendly way to portable high-level parallel and heterogeneous programming, which has been demonstrated by a number of programming frameworks during the last decade [5,7,8,21,25,27]. *Skeletons* are generic programming constructs based on higher-order functions such as *map*, *reduce*, *stencil* etc. that express certain parallelism patterns, that can be parameterized in problem-specific code (the so-called *user functions*) and that come with parallel or accelerator-specific implementations (the so-called *backends*), which are hidden behind a portable high-level API, today usually based on C++. In short, skeletons expose *possible* application-level parallelism but not its implementation details to the programming framework and its runtime system, which might then be free to decide which skeleton instances in a program to use (and how), and which ones should better remain sequential.

SkePU [9] is a pattern-based high-level programming model for transparent program execution on heterogeneous parallel computing systems. A key feature of SkePU is that, in general, the selection of the backend, and thus, the execution platform for a skeleton-based function call need not be determined statically, i.e. prior to execution. On single-node systems, SkePU can select among CPU, multithreaded CPU, single or multi-GPU execution. For example, run-time selection of the expected fastest [6] backend (depending on operand size and location) can be tuned automatically based on training executions or manually set by a flag outside the program's source code. By careful API design, each SkePU program is a valid C++ program with sequential execution semantics if compiled with a standard C++(11 or later) compiler, and SkePU's design for portability aims at executions over multiple cores or one or several GPUs to show the same input-output behavior as this sequential view.

Many scientific applications, such as Monte-Carlo simulations, use pseudo-random number generators (PRNGs) as part of the computation. In the interest of correctness and debugging, deterministic parallel or heterogeneous execution of such a program that remains consistent with sequential execution also in terms of generated random numbers is a desirable property, which however requires a *deterministic* parallel pseudo-random number generator. This becomes a challenge with SkePU's design of late decision about sequential, parallel or accelerator execution.

In this paper, we present the principle, API and implementation of a deterministic, portable parallel PRNG extension to SkePU that exhibits the same behavior regardless where and with how many resources a SkePU program is executed. Our deterministic PRNG parallelization also relaxes the implicit dependence structure of applications using the PRNG. We show that the implementation is scalable on both multi-core CPU and GPU resources, and hence supports the universal portability of SkePU code even in the presence of PRNG calls. It also leads to more compact source code. Core contributions are the determinism and the high-level language integration of our approach. While our solution is prototyped and evaluated in SkePU, where it is important due to the execution unit of a skeleton call being statically unknown, the approach could be adapted and integrated into other frameworks for high-level portable pattern-based parallel programming.

The remainder of this paper is organized as follows: Section 2 introduces background about SkePU and parallel random number generators, shows two motivating examples of previous workarounds used with SkePU to achieve deterministic parallel PRNG behavior, and discusses their drawbacks. Section 2.3 discusses related work. Section 3 explains three fundamental parallelization methods for PRNGs and presents the new API and implementation of the new built-in deterministic parallel PRNG in SkePU. Section 4 presents experimental results, and Section 5 concludes.

2 Background and Related Work

2.1 SkePU

In its current version [9], SkePU (`https://skepu.github.io`) provides 7 data-parallel skeletons: `Map` (elementwise transformation), `MapOverlap` (stencil updates in 1D...4D), `MapPairs` (generic outer product of vectors), `Reduce` (generic reduction), `Scan` (generic prefix sums), and the combinations `MapReduce` and `MapPairsReduce`. In general, the skeletons allow both element-wise accessed, random-access and scalar operands and are fully variadic within each of these categories. Most skeletons also allow multiple return operands. Array-based operands can have 1 to 4 dimensions.

By instantiating a skeleton with one or several matching problem-specific user functions (detailed further below), a callable entity (a *skeleton instance*) is generated, see Listing 1 for an example. The `MapReduce` instance `dotprod` can be used like any manually written function, but comes with multiple backends (implementations) for the different target platforms, such as sequential execution, OpenMP multithreaded execution, CUDA and OpenCL for GPUs. There exists also a cluster backend for SkePU that targets the MPI interface of the StarPU runtime system [9].

Listing 1: A simple SkePU example: Dot product.

```
#include <skepu>
...
double mult (double x, double y) { return x * y; }  // user function
double add (double x, double y) { return x + y; }  // user function
...
void main( void )
{   ...
    skepu::Vector<double> u(size), v(size);   // two 1D data-containers
    ...
    auto dotprod = skepu::MapReduce<2>(mult, add); // instantiate
    std::cout << dotprod(u, v);  // call skeleton instance on 2 vectors
}
```

For passing array-type data into or out of skeleton instance calls, so-called *data containers* must be used, which transparently perform memory management, software caching and data transfers for contained array elements. SkePU 3 supports data containers for arrays in 1D (`Vector`), 2D (`Matrix`), 3D and 4D (`Tensor`X`D`). All data containers are generic in the element type.

User functions must be side-effect free and be written in a restricted subset of C++ (e.g., no dynamic memory allocation, no explicit parallelism, no skeleton instantiations or -calls, no global variable access) as they are translated into the various platform-specific programming models (e.g., OpenMP, CUDA, OpenCL) and may execute on an accelerator with a possibly separate address space. For array-based operands passed as arguments to user functions, the foreseen access pattern is specified by access proxy parameter objects such as `Vec<>` for random-accessed vector, `Region`X`D<>` for stencil halo regions in `MapOverlap` ($X \in \{1, ..., 4\}$) or `Index`X`D` for the index of the element operated on in `Map`-based skeletons; element-wise access is the default (no proxy parameter type required). Access to the proxy elements depends on where the user function will be executed and is thus entirely managed by SkePU's data containers. User functions can also be defined as C++ *lambda* (anonymous) functions, allowing for in-line skeleton instantiation.

2.2 Parallel pseudo-random number generation

A pseudo-random number generator is a finite state automaton. Each time it is invoked, its output function computes and outputs a pseudo-random number in a pre-defined range from the current inner state, and transitions the inner state via the state transition function (also called update function) into the follow-up state. The generator only receives input upon the time of seeding, when the seed is processed by the initialization function to produce an initial inner state. Thus, the generator only has a very limited amount of randomness, which is stretched over many outputs, i.e. pseudo-random. Still, current generators pass statistical tests such as Diehard. The complexity to achieve this may lie in the output function and/or the update function. For a complex output function, the update function can be as simple as a counter [17].

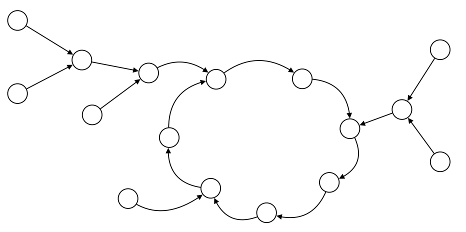


Fig. 1: State-space of a pseudo-random number generator.

If an output of m bits is produced, the inner state comprises more than m bits. The state transition function mostly is non-bijective[1]. Thus, the state graph of the PRNG comprises one node for each state x, and a directed edge $(x, f(x))$ for the transition from x to its follow-up state $f(x)$, assuming f as the state transition function. Thus each node has an outdegree of exactly 1, but the indegree can vary. An example state graph is shown in Fig. 1.

Flajolet and Odlyzko [11] investigated the expected structure of state graphs if all possible transition functions are equally likely. The graph falls into a small number of weakly connected components, of which one comprises the majority of the nodes (about 75%). Each component comprises a cycle with a number of trees directed towards the cycle, where the largest tree is expected to comprise 50% of all nodes. The expected length of the longest cycle is less than $2\sqrt{N}$, where N is the number of nodes, i.e. quite short. Trees are ragged with depth about $\sqrt{N}$.

The sequence of generated pseudo-random numbers is only dependent on the seed. In a sequential program with a deterministic program flow, the calls to the pseudo-random number generator will produce exactly the same numbers at the same program place if the seed is fixed. If the program is parallelized, then the PRNG state becomes a shared resource. Moreover, the order of calls to the PRNG changes: consider e.g. a nested loop with one call per iteration of the inner loop, where the outer loop is parallelized, so that now the first iterations of all instances of the inner loop call the PRNG first. Still, a deterministic parallel execution with results similar to the sequential version (and independent of the number of threads used to parallelize the outer loop) demands that the sequence of PRNG outputs for each inner loop execution remains unchanged, e.g. to do debugging in the sequential version when the parallel version has an error. This calls for a deterministic PRNG implementation as part of the parallelization.

2.3 Related Work

Kneusel [17] has a chapter on parallel PRNGs, but with respect to deterministic execution only reports a manual construction of duplicating the state

[1] A notable exception is the linear congruential generator with transition function $f(x) = ax + b \bmod N$ for a, b chosen such that the period is maximum [18], e.g. a prime and $b = 0$.

variable for each thread, plus skipping a number of states in order to achieve the same state as in a sequential execution. He also explains counter-based PRNGs and their suitability for parallelization because they allow skipping any given number of states with constant effort. Fog [12] discusses requirements on PRNGs in parallel computations, but focuses on avoiding overlapping sequences in different threads by combining generators, while L'Ecuyer et al. [19] focus on providing independent streams and substreams. Salmon et al. [26] focus on output functions for counter-based PRNGs to provide fast skipping of states but still provide good statistical quality. All do not focus on deterministic execution independent of parallelization, and have static mapping of tasks to threads in mind.

Leiserson et al. [20] argue that SPRNG [22], which provides a deterministic parallel PRNG, shows poor performance on Cilk programs and thus is not suitable for massive parallelism. They propose pedigrees, a mechanism to achieve a kind of linearization (i.e. a kind of equivalence to a sequential execution) in a Cilk program independent of the Cilk scheduler. However, they do not address pattern-based parallelization.

Parallel PRNG specifically for GPU include the cuRAND library for CUDA, SYCL-PRNG for SYCL, and work by Ciglarič et al. [2] for OpenCL. The Thrust skeleton library for CUDA also includes a PRNG library. Passerat et al. [23] discuss general aspects of PRNG on GPGPUs. GASPRNG [14] is an early attempt at realizing the full SPRNG generator set on CUDA GPUs, including clusters of CUDA GPU nodes.

2.4 Previous manual parallelization of PRNG in SkePU programs

With previous versions of SkePU, a deterministic parallel random number generator behavior had been achieved by the two workarounds described in the following. However, we will show that both have drawbacks.

2.4.1 Monte-Carlo Pi Calculation - index-based scrambling

As a first example, we consider a simple Monte-Carlo simulation, namely probabilistic Pi approximation. This computation can be easily expressed as a `MapReduce` instance, see Listing 2, where the user function needs to generate two pseudo-random numbers, one per dimension. Here, a deterministic parallel PRNG was simulated by an *index-scrambling technique*, i.e., generation of pseudo-random numbers does not follow the automaton-based best-practice technique described above; instead, they are calculated independently of each other based on a transformation of the index in the parallelized main loop. In the code example in Listing 2, the `scramble` function itself has been extracted from a SPH (Smoothed Particle Hydrodynamics) simulation code. The drawback of the index scrambling method is that it may not really produce random numbers of high quality but can expose more regular patterns.

Listing 2: Ad-hoc deterministic pseudo-random number generation by index scrambling in a Monte-Carlo method for Pi calculation in SkePU.

```
#include <iostream>
#include <skepu>

// Define c, s, s2, s3, MY_RAND_MAX as preprocessor constants

float scramble(int in)
{
    return ((((int)(10*s*s2*in + 4*c*s3 + 5*in + 10*s*in)) % MY_RAND_MAX)
            / ((float)MY_RAND_MAX));
}

float monte_carlo_sample(skepu::Index2D index)
{
    float x = scramble(index.row);
    float y = scramble(index.col);
    // check if (x,y) is inside region:
    return ((x*x + y*y) < 1) ? 1.f : 0.f;
}

float add( float lhs, float rhs ) { return lhs + rhs; }

int main(int argc, char *argv[])
{
    auto montecarlo = skepu::MapReduce<0>(monte_carlo_sample, add);

    const size_t samples = atoi(argv[1]);
    montecarlo.setDefaultSize(samples, samples);

    float pi = montecarlo() / (samples * samples) * 4;
    std::cout << pi << "\n";
}
```

2.4.2 Markov Chain Monte Carlo methods in LQCD – PRNG with explicit state

The code excerpt in Listing 3 is extracted from a Lattice QCD mini-application which computes the Yang-Mills theory of the $SU(3)$ group. This computation is typically done by applying the Metropolis algorithm, a common Markov Chain Monte Carlo (MCMC) based method. The Metropolis calculations are performed on a 4D tensor whose elements are structures of complex number arrays, with a 81-point ($3 \times 3 \times 3 \times 3$) stencil computation required to evaluate the Metropolis acceptance function. For an in-depth introduction to MCMC methods in LQCD, see [15].

Unlike the conventional Monte Carlo method showcased in Listing 2, MCMC methods are inherently sequential. Thus, a PRNG for MCMC methods has to be stateful, i.e. a finite state automaton as outlined in Section 2.2. This conflicts with the requirement that SkePU user functions must be side-effect free. The chosen solution for the user functions of Listing 3 is a sequential PRNG which is algorithmically equivalent to POSIX `drand48` but has an explicit state argument instead of `drand48`'s internal state variable.

For such an approach, the PRNG state has to be explicitly managed. As a dedicated PRNG state container is not a viable solution due to syntactical constraints of the `MapOverlap` skeleton, the state is embedded directly in the data set. This has the drawback of having an unusually large memory footprint for a PRNG. Specifically, the memory requirement for storing the PRNG states grows by $O(L^4)$ where L is the side length of the 4D tensor, i.e. linearly with the problem size. Usage of the proposed new library PRNG inside SkePU is expected to lower the memory footprint of PRNG state storage to $O(p)$ where p is the number of computational units used in the selected backend.

While it would be possible to adapt the index-based scrambling technique of Listing 2 to perform the initial seeding of the resulting parallel PRNG, Listing 3 contents itself with using the `Scan` skeleton to force a non-repeating state set into existence. While this is viable as a quick and dirty solution to deterministic parallel PRNG seeding, it is likely to produce random numbers of suboptimal quality; in that respect, a mathematically robust library solution is preferable.

3 Designing a Deterministic PRNG for SkePU

We will now introduce a more systematic approach that provides deterministic parallel random number generation for use in SkePU, together with an API extension of SkePU 3 that makes PRNG streams a fundamental part of the API. We will start by discussing inherent challenges to pseudo-random number generation in parallel programming and proceed step by step towards a deterministic PRNG implementation at the framework level.

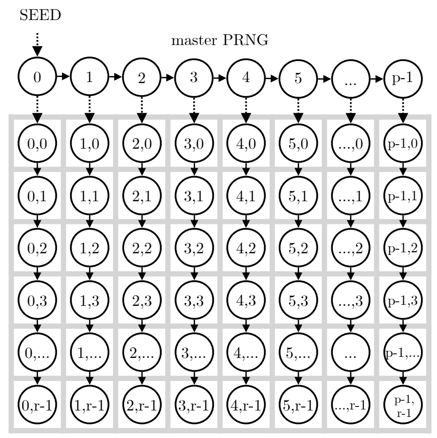
3.1 Global synchronization

A straightforward approach to random number generation in parallel applications is to consider the PRNG as a shared resource. As such, the PRNG needs to be protected by the appropriate synchronization operations during access, to avoid race conditions such as multiple threads reading the same random value, which would decrease the quality of the random number stream, or even the PRNG state itself being corrupted due to simultaneous writes.

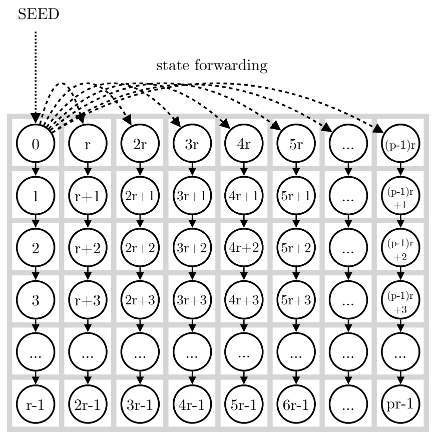
This approach ensures a high-quality random number stream as each value is generated in the same manner as in a sequential program. Any random number generator can be used in this approach, including external entropy sources, since synchronization guarantees protected sequentialized access. This synchronization does however add significant overhead and is unfeasible in massively parallel accelerators such as GPUs. Only if the synchronization method guarantees a deterministic order of accesses to the critical section containing the PRNG state (which is usually not the case for ordinary lock-based synchronization), the random number stream generated from this method will be itself deterministic. We cannot predict in which order the threads will generate a value from the PRNG and update the state space.

Listing 3: Simplified SkePU code for an explicit-state parallel PRNG for Markov-chain-based LQCD applications (Sect. 2.4.2).

```
typedef uint64_t PRNGState;

// Seeding:
skepu::Tensor4<PRNGState> ones( L, L, L, L, 1), prngs( L, L, L, L);
auto seedPRNGs = skepu::Scan( []( PRNGState x, PRNGState y){ return x+y;});
seedPRNGs( prngs, ones);

// Extracting:
inline PRNGState statelessDrand48( PRNGState prng)
{
    return (0x5deece66d * prng + 11) % (1LL<<48);
}
inline double normalize( PRNGState prng)
{
    return (double)prng / (double)(1LL<<48);
}

// Parallel state management:
struct localGauge; // 36 double-precision complex numbers
struct localGaugeAndPRNG
{
    localGauge data;
    PRNGState prng;
};
skepu::Tensor4<localGaugeAndPRNG> gaugeField( L, L, L, L);

// Gauge randomization:
localGaugeAndPRNG randomizeGauge( PRNGState prng)
{
    localGaugeAndPRNG gaugeNew;
    for (int idx = 0; idx < 36; idx++) {
        prng = statelessDrand48( prng);
        gaugeNew.data.at(idx).re = normalize( prng);
        prng = statelessDrand48( prng);
        gaugeNew.data.at(idx).im = normalize( prng);
    }
    gaugeNew.prng = prng;
    return gaugeNew;
}

// Metropolis step:
localGaugeAndPRNG localUpdate( skepu::Region4D<localGaugeAndPRNG> stencil)
{
    localGaugeAndPRNG proposal = randomizeGauge( stencil( 0,0,0,0).prng);
    double limen = someDeterministicStencilArithmetic( stencil, proposal);
    stencil( 0,0,0,0).prng = statelessDrand48( proposal.prng);
    if (normalize( stencil( 0,0,0,0).prng) >= limen) {
        stencil( 0,0,0,0).data = proposal.data;
    }
    return stencil( 0,0,0,0);
}

auto metropolisUpdate = skepu::MapOverlap( localUpdate);
for (int iter = 0; iter < Niter; iter++) {
    metropolisUpdate( gaugeField, gaugeField);
}
```


SEED
master PRNG
0
1
2
3
4
5
...
p-1
0,0 1,0 2,0 3,0 4,0 5,0 ...,0 p-1,0
0,1 1,1 2,1 3,1 4,1 5,1 ...,1 p-1,1
0,2 1,2 2,2 3,2 4,2 5,2 ...,2 p-1,2
0,3 1,3 2,3 3,3 4,3 5,3 ...,3 p-1,3
0,... 1,... 2,... 3,... 4,... 5,... ... p-1,...
0,r-1 1,r-1 2,r-1 3,r-1 4,r-1 5,r-1 ...,r-1 p-1, r-1

(a) Stream-splitting approach to parallel pseudo-random number generation.


SEED
state forwarding
0 r 2r 3r 4r 5r ... (p-1)r
1 r+1 2r+1 3r+1 4r+1 5r+1 ... (p-1)r +1
2 r+2 2r+2 3r+2 4r+2 5r+2 ... (p-1)r +2
3 r+3 2r+3 3r+3 4r+3 5r+3 ... (p-1)r +3
... ...
r-1 2r-1 3r-1 4r-1 5r-1 6r-1 ... pr-1

(b) State-forwarding approach to parallel pseudo-random number generation.

Fig. 2: Approaches for parallelizing a PRNG sequence.

3.2 Stream splitting

With the goal of avoiding or minimizing global synchronization of the PRNG state, we consider a different approach [13]. As a PRNG state has to be considered a shared resource for proper operation, we can get around the synchronization requirement by assigning each individual thread its own PRNG state. A thread-private PRNG stream does not need protected access and will yield a perfect sequential series of random values by itself. However — aside from a large increase in memory space consumed by the replicated states — with several or many parallel threads in the system, the aggregate random number stream over all task invocations will differ greatly from a sequential program.

Whether data-parallel tasks are assigned in blocks or interleaved, we effectively have *split* the single PRNG stream into many shorter sequences distributed over the working set in the same pattern as the data-parallel tasks. The resulting pattern can be seen in Figure 2a. This degrades the quality of the random values in aggregate, which is undesirable for sensitive applications.

There is another unfortunate consequence of this approach: ensuring determinism in the random value stream is possible, but with significant restrictions. Due to the aforementioned parallelization of the computation using the PRNG, the observed PRNG stream across the data-set is a mangled mixture of (a potentially large number of) individual streams. This mangling has to be replicated in the sequential execution of the program to preserve determinism; and worse, all parallel backends have to observe the same such mapping. This can prove tricky when the parallel backends vary significantly in properties such as the available parallelism degree. A consequence of this behavior is also that in any execution of the program for which deterministic random values are desired, the maximal number of threads has to be known *a priori*, before even executing a sequential backend variant. If the degree of parallelism ever is increased, e.g. by moving to a larger processor, GPU, or cluster, the previous runs are invalidated with respect to the determinism criterion.

3.3 State forwarding

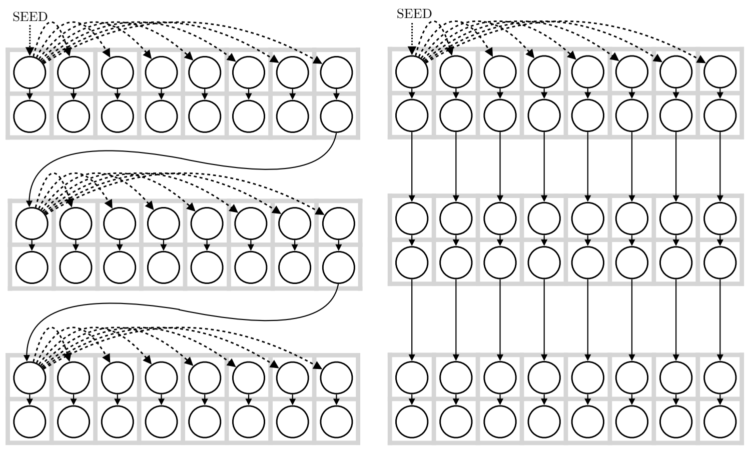
The approach taken in this work is *state forwarding.* We attempt to side-step the issues of both the global synchronization as well as the stream splitting approaches. This is done by utilizing properties of the PRNG state spaces. A true sequential single-stream variant of the program is taken as the gold standard output, and the goal is to replicate the same output on any parallel backend, without the need of global synchronization or advance knowledge of parallelism degree. As in the stream splitting approach, data-parallel work items are deterministically mapped across available computational units (threads). This means that the number of tasks assigned to each thread is known ahead of time, and for simplicity without the loss of generalization we assume the work can be split evenly among threads.

Furthermore, we assume that the number of times a PRNG state is updated (i.e., the number of times a random value is generated) is known ahead of time for each work unit. Combining the knowledge of *work unit count* and *random calls per work unit*, we know exactly how many state-forwards each thread will generate in the respective data-parallel construct (i.e., skeleton invocation).

We can therefore, for each thread, *pre-forward* the state of the PRNG and store a copy of the forwarded state. These per-thread forwarded clones of the original PRNG can now act as the thread-private PRNG streams in the stream-splitting approach, with the additional property that when interleaved during the data-parallel execution, the aggregate observed stream now is equivalent to the sequential stream, which was the primary hurdle in the stream-splitting approach. Figure 2b illustrates the resulting pattern.

Still, the extra memory footprint of the thread-private PRNG states persists and will lead to additional overhead. The state-forwarding adds an additional computation step before the execution of the tasks, which can in the worst case be equally costly as the PRNG value extraction process itself (though it can also be parallelized). Properties of the PRNG state space have to be exploited to speed up the forwarding process and reduce the induced overhead.

The leapfrog resp. sequence splitting method for state forwarding, introduced by Celmaster and Moriarty [1] for use with vector computers, considers a special case that allows to parallelize the forwarding phase of the PRNG. A linearly congruential PRNG with factor a is partitioned into p linearly congruential PRNGs each to be used r times, which are defined based on the same linear factor a, by $seed(i) = (a^r \cdot seed(i-1)) \bmod m$ for $i = 1, ..., p$, $rand(i, 0) = seed(i)$ and $rand(i, j) = a \cdot rand(i, j-1) \bmod m$. Hence, the p PRNGs equally partition the period of the seed PRNG in contiguous subsequences of length r. First, the a^{ir} for $i = 0, ..., p-1$ and the seed sequence can be calculated in parallel by a `Scan` in $O(\log p)$ steps, using the property $a^{2k} \bmod m = ((a^k \bmod m)^2) \bmod m$. Then the *rand* calls are independent for each i. (For reasonably low numbers of p such as for a current multicore CPU, sequential computation of the seeds should be faster; this is done in the current implementation.) The leapfrog / sequence splitting method scales well



(a) No pre-forwarding between iterations.

(b) Pre-forwarding the PRNGs once before the iterative loop.

Fig. 3: Container indexing and memory layout.

but is known to have problems for lcg with power-of-2 values for modulus and p. Skipping can also be applied for counter-based PRNGs [26] with output functions based on block ciphers for better statistics at a higher cost.

3.4 Optimizing long or iterated skeleton chains by pre-forwarding

While some applications may consist of a single parallelized step (such as a parallel for loop or skeleton call; we will use the latter here), others, in particular larger applications, will have multiple phases which are individually parallelized. A common example is iterative applications where each iteration in turn consists of one or more skeleton calls. To achieve good efficiency, we need to ask the question: *when* is the PRNG state split and forwarded for the purposes of parallelization in a skeleton invocation scenario?

In a naive implementation of the state-splitting approach, the state splitting and forwarding step (see Fig. 3a) is done right before each skeleton call. On the other hand, if we have a known number of skeleton calls (determinable by static analysis, lineage building [10], or program instrumentation), we only need to perform the splitting and forwarding of the PRNG states once per application. This is referred to as *pre-forwarding* and is illustrated in Figure 3b. In practice, restrictions such as data-dependent control flow (e.g., branches or iteration bounds) may limit the degree to which pre-forwarding can be applied, and application programmers may benefit from awareness of the cost-reduction opportunities from pre-forwarding already during program design.

3.5 API Extension Design

We have implemented the state-forwarding approach in the skeleton programming framework SkePU 3. SkePU did not previously have a random number generation component, and as shown in Section 2.4, previous manual implementations of PRNG-like functionality in SkePU applications have been adhoc and substantially different from each other. A baseline contribution of a framework-level PRNG library in SkePU is the programmability gains from reducing the effort of designing probabilistic applications on top of SkePU, as well as readability benefits from having a unified system for random number generation across all SkePU programs.

3.5.1 Random number extraction in user functions

As explained in Section 2.1, a SkePU skeleton is defined entirely by its type (e.g., `Map`), the signature of its instantiating user function, and state properties set on the resulting skeleton instance (such as `.setOverlap(...)` for `MapOverlap` instances). PRNG extraction is made available in all skeletons with a fully data-parallel mapping stage, which is the entire skeleton set except for `Reduce` and `Scan`.[2]

As such, the user function signature ("header") itself should encode the use of random number extraction. This is analogous to the preexisting option for mapping user functions to request the index of the currently processed element (see Listing 2). Therefore, we encode PRNG reliance in the same way. At the start of the parameter list (after the index parameter, if any), a parameter of type `skepu::Random<N>&` is added. `N` is a compile-time constant used in SkePU's *template metaprogramming*-based implementation to deduce the number of random values extracted by the user function in the *dynamic extent* of its evaluation. N is required to be known ahead-of-time for the state forwarding to work and determinism to be preserved.[3] A compilation option allows for run-time verification that the extraction count is obeyed.

Value extraction is carried out by a call to one out of two member functions of the `skepu::Random<N>` object. `random.get()` produces integers in $[0, \text{SKEPU\_RAND\_MAX})$ while `random.getNormalized()` returns real numbers in $[0, 1)$. Each call corresponds to one extraction and state update of the PRNG stream. A basic example of a user function with 5 random number extractions is shown in Listing 4.

SkePU user functions are allowed to call other functions, subject to some but not all restrictions of skeleton-instantiating user functions. As the extraction count N is only required for instantiation, passing a PRNG stream object to indirect user functions is instead done with a `skepu::Random<>*` parameter with no positional requirement.

[2] `Reduce` and `Scan` are parallelized through tree reductions reliant on the associativity property of their user functions.

[3] If determinism is not required by the application, `N` can be treated as an upper bound, which instead guarantees that no sub-sequences of random numbers are overlapping.

Listing 4: User function with calls to the SkePU random number generator.

```
float uf( skepu::Random<5> &prng, skepu::Region1D<float> region )
{
    float res = 0;
    for (int i = -2; i <= 2; ++i)  // 1D stencil with random weights
        res += region(i) * prng.getNormalized();
    return res;
}
```

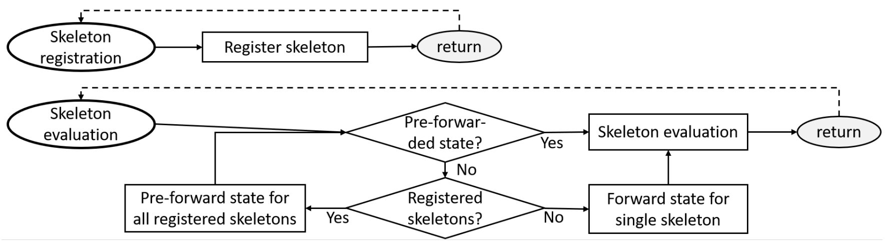


Fig. 4: Flow-chart of the deterministic PRNG implementation. Here ellipses are events and boxes correspond to processes.

3.5.2 PRNG streams and skeleton invocations

Once a `skepu::Random<N>&`-enabled user function is present, a skeleton can be instantiated as usual. In addition to the skeleton instance, a PRNG stream object needs to be defined in the program: an object of type `skepu::PRNG`. Initialization of the PRNG stream takes an optional *seed* integer argument. The seed changes the deterministic sequence generated in the stream and can be assigned from an external entropy source (e.g., a timestamp) if non-determinism across program runs is preferred.

The stream object is a state machine which registers skeletons ahead of invocation time. Also in this way PRNG streams work like SkePU's index parameters: the stream is not part of a skeleton call's argument list. Instead they are registered as `skeleton.setPRNG(prng)`. The full flow chart of the registration and evaluation process is shown in Figure 4. In short, several skeleton instances may be *registered* before reaching an *evaluation* event. Only at this point is the PRNG sequence split across computational units and forwarded to the appropriate state. The input size (i.e., the maximum degree of parallelism) has direct impact on the forwarding leaps and is only known at the evaluation point from the input arguments to the skeleton call.[4] In subsequent skeleton invocations, the PRNG object checks for existing forwarded state and skips directly to evaluation (refer to Fig. 3b).

Listing 5 shows a variant of the Monte-Carlo Pi calculation algorithm using the new SkePU API. Implementation with a `MapReduce<0>` skeleton enables a data-parallel computation without explicit container allocation, as the al-

[4] The input size is assumed to be uniform over a sequence of skeleton calls.

Listing 5: Pi approximation using the new SkePU PRNG API.

```
#include <iostream>
#include <skepu>

int monte_carlo_sample(skepu::Random<2> &random)
{
    float x = random.getNormalized();
    float y = random.getNormalized();
    // check if (x,y) is inside region:
    return ((x*x + y*y) < 1) ? 1 : 0;
}

int add(int lhs, int rhs) { return lhs + rhs; }

int main(int argc, char *argv[])
{
    auto montecarlo = skepu::MapReduce<0>(monte_carlo_sample, add);

    skepu::PRNG prng;
    montecarlo.setPRNG(prng);

    const size_t samples = atoi(argv[1]);
    montecarlo.setDefaultSize(samples);

    double pi = (double)montecarlo() / samples * 4;
    std::cout << pi << "\n";
}
```

gorithm needs no element-wise input data to the user function; all input is derived from the PRNG stream. Internally, SkePU will use two containers: one input data set for the split PRNG sub-sequence states, and one output data set for the results of the user function invocations. Note, however, that SkePU will optimize the size of these intermediate data sets; they grow by $O(p)$, the number of computational units, and not $O(n)$, problem size (here the sample count).

Our prototype implementation handles multiple PRNG stream objects across different skeleton calls, but a single skeleton call (and thus its user function) can only receive values from one PRNG stream per invocation. `skepu::Random` usage can be combined with most other SkePU features, with a notable exception being dynamic scheduling for multi-core execution introduced [9] in SkePU 3.

4 Experimental Evaluation

For the performance evaluations in this section, we use a server with two six-core Xeon E5-2630L CPUs with two-way hardware multi-threading, a Nvidia K20c GPU, and 64 GiB of main memory. The system runs Ubuntu 18.04.5 LTS and GCC 10.3.0 is used as backend compiler with -O3 optimization level.

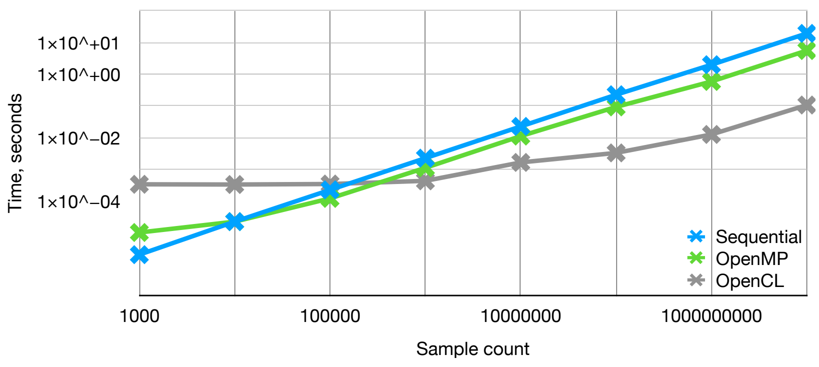


Fig. 5: Monte-Carlo Pi calculation with varying sample count on different backends.

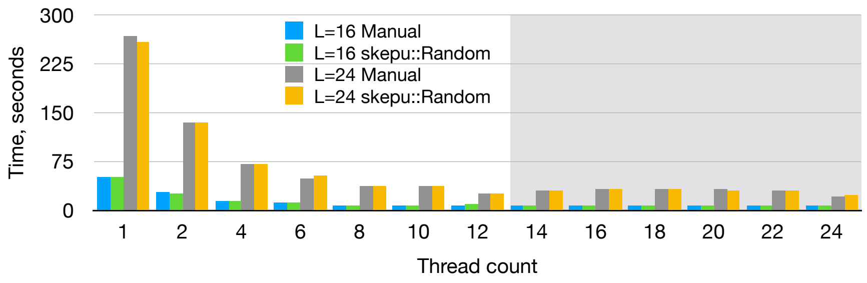


Fig. 6: Time (seconds) for 10 iterations of LQCD with lattice sizes $L = 16$ and $L = 24$ for varying number of hardware threads in the OpenMP backend.

4.1 Monte-Carlo Pi approximation

We begin with the probabilistic Pi calculation from Section 2.4.1. SkePU code using the new `skepu::Random` API is shown in Listing 5.

Figure 5 contains the performance results from executing the SkePUized program on various backends. The Monte-Carlo Pi calculation algorithm is an interesting stress test due to the random number generation dominating the total work. The application scales well on the GPU using the OpenCL backend (up to 180x speedup compared to sequential in the presented results), even though the work done in the user function is very lightweight.

4.2 LQCD Mini-Application

For the LQCD mini-application introduced in Section 2.4.2, SkePU code using the new PRNG API is shown in Listing 6.

Figure 6 shows the times of 10 iterations of LQCD with the OpenMP backend, comparing the manual workaround of Listing 3 to the new version using `skepu::Random` of Listing 6 (note that the optimization for pre-forwarding is

Listing 6: Markov-chain-based LQCD application with new SkePU PRNG API

```
// Data management:
struct localGauge; // 36 double-precision complex numbers
skepu::Tensor4<localGauge> gaugeField( L, L, L, L);

// Gauge randomization:
localGauge randomizeLocalGauge( skepu::Random<> *prng)
{
    localGauge gaugeNew;
    for (int idx = 0; idx < 36; idx++) {
        gaugeNew.at(idx).re = prng->getNormalized();
        gaugeNew.at(idx).im = prng->getNormalized();
    }
    return gaugeNew;
}

// Metropolis step:
localGauge localUpdate( skepu::Random<73>& prng,
                        skepu::Region4D<localGauge> stencil )
{
    localGaugeAndPRNG update = randomizeLocalGauge( prng);
    double limen = someMatrixArithmetic( stencil, update);
    if (prng.getNormalized() >= limen) {
        stencil( 0,0,0,0) = update;
    }
    return stencil( 0,0,0,0);
}

...
skepu::PRNG prng;
auto metropolisUpdate = skepu::MapOverlap( localUpdate);
metropolisUpdate.setPRNG( prng);
for (int iter = 0; iter < Niter; iter++) {
    metropolisUpdate( gaugeField, gaugeField);
}
```

Listing 7: Pseudocode of Miller-Rabin probabilistic primality testing

```
int test( int n )
{
   result = true;
   for (int i = 1; i <= t; i++) {
      int a = rand();
      result = result & millerrabin( n, a );
      // if (result == false) break;
   }
   return result;
}
```

not activated yet and will be contained in the final paper; the times are expected to improve). We can see that no new overheads are introduced while code complexity decreases (see Sect. 4.4).

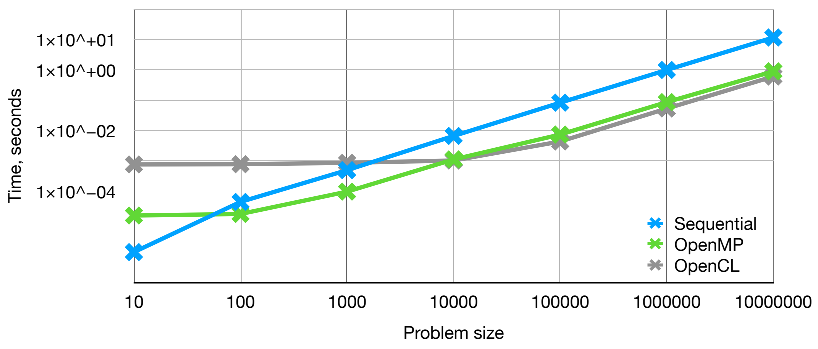


Fig. 7: Miller-Rabin primality test with varying sample count on different backends.

4.3 Miller-Rabin primality testing

The Miller-Rabin primality test [24] is a probabilistic algorithm to determine for a given number if it is likely prime or not. The actual test gets two inputs: n, the number to be tested for primality, and a value a in the range $\{2...n-2\}$. It performs a computation on a and n, and depending on the result it outputs "n is prime" or "n is composite". While the latter answer is always true, there is a certain probability that the former answer is wrong, and this probability can be reduced by doing the computation repeatedly with randomly chosen a, see Listing 7. This can be easily parallelized, as the t iterations are independent (except for calls to the PRNG), but for comparability it is helpful that the random choices are similar to the sequential version.

Our SkePU implementation of the Miller-Rabin algorithm is largely based on an open-source implementation in C++ by Larsen[5] where the main Monte-Carlo parallelism is expressed by a `MapReduce<0>` skeleton instance.

Parallel performance of the SkePUized Miller-Rabin application can be seen in Figure 7. Instruction flow is highly divergent throughout the algorithm due to data-dependent control flow, which is challenging for the GPU backend: it is just barely faster than multi-core CPU computation. This property distinguishes the program from the Monte-Carlo sampling algorithm wherein the PRNG values have no effect on control flow. For the multi-core OpenMP backend we observe speedup up to 13x.

4.4 Programmability Evaluation

In all of Pi calculation, Lattice QCD and Miller-Rabin primality test, the implementations see a reduction in lines-of-code count after applying the SkePU

[5] C.S. Larsen: The Miller-Rabin primality test in C++. `https://github.com/cslarsen/miller-rabin`

PRNG API. This effect primarily comes from abstracting the implementation details of the PRNG engine itself. The entire code base of the SkePUized LQCD application is reported by `sloccount` to be 1,212 lines of code before applying the new `skepu::Random` API, and 1,137 afterwards. This amounts to a reduction by 6.2%. In addition, the change simplifies the data structure hierarchy, and fewer skeleton calls and user function declarations are necessary.

5 Conclusion and Future Work

We have proposed a method for realizing a deterministic parallel PRNG for use with skeleton-based high-level programming of heterogeneous parallel systems where the type and number of parallel execution resources of skeleton calls can be selected dynamically at run-time. We provided an extension of the SkePU API and its prototype implementation based on state forwarding. We evaluated it with three probabilistic applications on multi-core CPU and GPU backends, and found that the proposed API and parallelization approach performs at least equally well as manual workarounds while code complexity is reduced. The PRNG scales on both CPU and GPU backends.

Ongoing work includes the extension of the prototype implementation in SkePU to support `skepu::Random` for other skeletons and for other GPU backends (e.g., CUDA); GPU implementations will also include state forwarding using a parallel prefix sum where applicable. Updated results will be reported in the final version of the paper, likewise GPU timings for LQCD. Moreover we plan to include one more test application.

Future work will extend the current implementation of the new PRNG functionality to also work with cluster and hybrid backends of SkePU. The PRNG API will be integrated in the public open source SkePU repository.

Acknowledgment

This work was partly funded by EU H2020 project 801015 (EXA2PRO), `exa2pro.eu` and by CUGS, Linköping University.

References

1. William Celmaster and Kevin J.M. Moriarty. A method for vectorized random number generators. *J. of Comput. Physics*, 64:271–275, 1986.
2. Tadej Ciglarič, Erik Štrumbelj, et al. An OpenCL library for parallel random number generators. *The Journal of Supercomputing*, 75(7):3866–3881, 2019.
3. Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004.
4. Murray I. Cole. *Algorithmic skeletons: Structured management of parallel computation.* Pitman and MIT Press, Cambridge, Mass., 1989.
5. Marco Danelutto and Massimo Torquati. Structured parallel programming with "core" FastFlow. In *Central European Functional Programming School*, volume 8606 of *LNCS*, pages 29–75. Springer, 2015.

6. Usman Dastgeer, Lu Li, and Christoph Kessler. Adaptive implementation selection in the SkePU skeleton programming library. In *Advanced Parallel Processing Technologies*, pages 170–183. Springer, 2013.
7. David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, 29(24):e4175, 2017.
8. Johan Enmyren and Christoph W Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14. ACM, 2010.
9. August Ernstsson, Johan Ahlqvist, Stavroula Zouzoula, and Christoph Kessler. SkePU 3: Portable high-level programming of heterogeneous systems and HPC clusters. *Int. J. of Parallel Programming*, May 2021. `https://doi.org/10.1007/s10766-021-00704-3`.
10. August Ernstsson and Christoph Kessler. Extending smart containers for data locality-aware skeleton programming. *Concurrency and Computation: Practice and Experience*, 31(5):e5003, 2019.
11. Philippe Flajolet and Andrew M. Odlyzko. Random mapping statistics. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology - Proc. EUROCRYPT '89 Workshop on the Theory and Application of Cryptographic Techniques, Houthalen, Belgium*, volume 434 of *LNCS*, pages 329–354. Springer, 1989.
12. Agner Fog. Pseudo-random number generators for vector processors and multicore processors. *J. Modern Appl. Stat. Meth.*, 14:308–334, December 2015.
13. Paul Frederickson, Robert Hiromoto, Thomas L. Jordan, Burton Smith, and Tony Warnock. Pseudo-random trees in monte carlo. *Parallel Comput.*, 1(2):175–180, December 1984.
14. Shuang Gao and Gregory D. Peterson. GASPRNG: GPU accelerated scalable parallel random number generator library. *Computer Physics Comm.*, 184(4):1241–1249, 2013.
15. Masanori Hanada. Markov chain monte carlo for dummies. arXiv 1808.08490, 2018.
16. John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.
17. Ronald T. Kneusel. *Random Numbers and Computers*. Springer, Cham (CH), 2018.
18. Donald E. Knuth. *The Art of Computer Programming Volume 2: Seminumerical Algorithms*. Addison-Wesley Longman, Boston, MA, 3rd edition, 1997.
19. Pierre L'Ecuyer, David Munger, Boris N. Oreshkin, and Richard J. Simard. Random numbers for parallel computers: Requirements and methods, with emphasis on GPUs. *Math. Comput. Simul.*, 135:3–17, 2017.
20. Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *Proc. 17th Symposium on Principles and Practice of Parallel Programming*, page 193–204. ACM, 2012.
21. Ricardo Marques, Hervé Paulino, Fernando Alexandre, and Pedro D. Medeiros. Algorithmic skeleton framework for the orchestration of GPU computations. In *Euro-Par 2013 Parallel Processing*, volume LNCS 8097, pages 874–885. Springer, 2013.
22. Michael Mascagni and Ashok Srinivasan. Algorithm 806: SPRNG: A scalable library for pseudorandom number generation. *ACM Trans. Math. Softw.*, 26(3):436–461, September 2000.
23. Jonathan Passerat-Palmbach, Claude Mazel, and David R.C. Hill. Pseudo-random number generation on GP-GPU. In *Proc. IEEE/ACM/SCS Workshop on Principles of Advanced and Distributed Simulation*, pages 146–153, June 2011.
24. Michael O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and complexity*, pages 21–39. Academic Press, 1976.
25. Christoph Rieger, Fabian Wrede, and Herbert Kuchen. Musket: a domain-specific language for high-level parallel programming with algorithmic skeletons. In *Proc. Symposium on Applied Computing (SAC'19)*, pages 1534–1543. ACM, 2019.
26. John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. Parallel random numbers: As easy as 1, 2, 3. In *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 16:1–16:12. ACM, 2011.
27. Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. SkelCL–a portable skeleton library for high-level GPU programming. In *16th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'11)*, pages 1176–1182, 2011.