Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms

Charles E. Leiserson Tao B. Schardl Jim Sukha
MIT Computer Science and Artificial Intelligence Laboratory
{cel, neboat, sukhaj}@mit.edu

Abstract

Existing concurrency platforms for dynamic multithreading do not provide repeatable parallel random-number generators. This paper proposes that a mechanism called ***pedigrees*** be built into the runtime system to enable efficient deterministic parallel random-number generation. Experiments with the open-source MIT Cilk runtime system show that the overhead for maintaining pedigrees is negligible. Specifically, on a suite of 10 benchmarks, the relative overhead of Cilk with pedigrees to the original Cilk has a geometric mean of less than 1%.

We persuaded Intel to modify its commercial C/C++ compiler, which provides the Cilk Plus concurrency platform, to include pedigrees, and we built a library implementation of a deterministic parallel random-number generator called DOTMIX that compresses the pedigree and then "RC6-mixes" the result. The statistical quality of DOTMIX is comparable to that of the popular Mersenne twister, but somewhat slower than a nondeterministic parallel version of this efficient and high-quality serial random-number generator. The cost of calling DOTMIX depends on the "spawn depth" of the invocation. For a naive Fibonacci calculation with $n = 40$ that calls DOTMIX in every node of the computation, this "price of determinism" is about a factor of 2.3 in running time over the nondeterministic Mersenne twister, but for more realistic applications with less intense use of random numbers — such as a maximal-independent-set algorithm, a practical samplesort program, and a Monte Carlo discrete-hedging application from QuantLib — the observed "price" was at most 21%, and sometimes much less. Moreover, even if overheads were several times greater, applications using DOTMIX should be amply fast for debugging purposes, which is a major reason for desiring repeatability.

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

General Terms Algorithms, Performance, Theory

Keywords Cilk, determinism, dynamic multithreading, nondeterminism, parallel computing, pedigree, random-number generator

This research was supported in part by the National Science Foundation under Grant CNS-1017058. Tao Benjamin Schardl was supported in part by an NSF Graduate Fellowship. Jim Sukha's current affiliation is Intel Corporation.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.

1. Introduction

Dynamic multithreading,[1] or ***dthreading***, which integrates a runtime scheduler into a concurrency platform, provides a threading model which allows developers to write many deterministic programs without resorting to nondeterministic means. Dthreading concurrency platforms — including MIT Cilk [20], Cilk++ [34], Cilk Plus [28], Fortress [1], Habenero [2, 12], Hood [6], Java Fork/Join Framework [30], OpenMP 3.0 [40], Task Parallel Library (TPL) [33], Threading Building Blocks (TBB) [42], and X10 [13] — offer a ***processor-oblivious*** model of computation, where linguistic extensions to the serial base language expose the logical parallelism within an application without reference to the number of processors on which the application runs. The platform's runtime system schedules and executes the computation on whatever set of ***worker*** threads is available at runtime, typically employing a "work-stealing" scheduler [5, 10, 23], where procedure frames are migrated from worker to worker. Although a dthreading concurrency platform is itself nondeterministic in the way that it schedules a computation, it encapsulates the nondeterminism, providing the developer with a programming abstraction in which deterministic applications can be programmed without concern for the nondeterminism of the underlying scheduler.

A major reason parallel programming is hard is because nondeterminism precludes the repeatability programmers rely on to debug their codes. For example, the popular ***pthreading*** model — as exemplified by POSIX threads [27], Windows API threads [24], and the threading model of the Java programming language [22] — is well known to produce programs replete with nondeterminism and which are thus difficult to debug. Lee [31] cites the nondeterminism of multithreaded programs as a key reason that programming large-scale parallel applications remains error prone and difficult. Bocchino *et al.* [7] argue persuasively that multithreaded programs should be deterministic by default. The growing popularity of dthreading concurrency platforms seems due in part to their ability to encapsulate nondeterminism and provide a more deterministic environment for parallel programming and debugging.

Nevertheless, dthreading concurrency platforms fail to encapsulate an important source of nondeterminism for applications that employ (pseudo)random number generators (RNG's). RNG's are useful for randomized algorithms [37], which provide efficient solutions to a host of combinatorial problems, and are essential for Monte Carlo simulations, which consume a large fraction of computing cycles [35] for applications such as option pricing, molecular modeling, quantitative risk analysis, and computer games.

Unfortunately, typical implementations of RNG's are either nondeterministic or exhibit high overheads when used in dthreaded code. To understand why, we first review conventional serial RNG's and consider how they are traditionally adapted for use in parallel

[1] Sometimes called ***task parallelism***.

programs. Then we examine the ramifications of this adaptation on dthreaded programs.

A serial RNG operates as a stream. The RNG begins in some initial state S_0. The ith request for a random number updates the state S_{i-1} to a new state S_i, and then it returns some function of S_i as the ith random number. One can construct a parallel RNG using a serial RNG, but at the cost of introducing nondeterminism. One way that a serial RNG can be used directly in a dthreaded application is as a global RNG where the stream's update function is protected by a lock. This strategy introduces nondeterminism, however, as well as contention on the lock that can adversely affect performance.

A more practical alternative that avoids lock contention is to use ***worker-local RNG's***, i.e., construct a parallel RNG by having each worker thread maintain its own serial RNG for generating random numbers. Unfortunately, this solution fails to eliminate nondeterminism because the underlying nondeterministic scheduler may execute a given call to the RNG on different workers during different runs of the program, even if the sequence of random numbers produced by each worker is deterministic.

Deterministic parallel random-number generators (DPRNG's) exist for pthreading platforms, but they are ineffective for dthreading platforms. For example, SPRNG [35] is an excellent DPRNG which creates independent RNG's via a parameterization process. For a few pthreads that are spawned at the start of a computation and which operate independently, SPRNG can produce the needed RNG for each pthread. For a dthreaded program, however, which may contain millions of ***strands*** — serial sequences of executed instructions containing no parallel control — each strand may need its own RNG, and SPRNG cannot cope.

Consider, for example, a program that uses SPRNG to generate a random number at each leaf of the computation of a parallel, exponential-time, recursive Fibonacci calculation `fib`. Every time `fib` spawns a recursive subcomputation, a new strand is created, and the program calls SPRNG to produce a new serial RNG stream from the existing serial RNG. The `fib` program is deterministic, since each strand receives the same sequence of random numbers in every execution. In an implementation of this program, however, we observed two significant problems:

- When computing `fib(21)`, the program using SPRNG was almost 50,000 times slower than a nondeterministic version that maintains worker-local Mersenne twister [36] RNG's from the GNU Scientific Library [21].
- SPRNG's default RNG only guarantees the independence of 2^{19} streams, and computing `fib(n)` for `n` > 21 forfeits this guarantee.

Of course, SPRNG was never intended for this kind of use case where many streams are created with only a few random numbers generated from each stream. This example does show, however, the inadequacy of a naive solution to the problem of deterministic parallel random-number generation for dthreading platforms.

Contributions

In this paper, we investigate the problem of deterministic parallel random-number generation for dthreading platforms. In particular, this paper makes the following contributions:

- A runtime mechanism, called "pedigrees," for tracking the "lineage" of each strand in a dthreaded program, which introduces a negligible overhead across a suite of 10 MIT Cilk [20] benchmark applications.
- A general strategy for efficiently generating quality deterministic parallel random numbers based on compressing the strand's pedigree and "mixing" the result.
- A high-quality DPRNG library for Intel Cilk Plus, called DOTMIX, which is based on compressing the pedigree via a dot-product [17] and "RC6-mixing" [15, 43] the result, and whose statistical quality appears to rival that of the popular Mersenne twister [36].

Outline

The remainder of this paper is organized as follows. Section 2 defines pedigrees and describes how they can be incorporated into a dthreading platform. Section 3 presents the DOTMIX DPRNG, showing how pedigrees can be leveraged to implement DPRNG's. Section 4 describes other pedigree-based DPRNG schemes, focusing on one based on linear congruential generators [29]. Section 5 presents a programming interface for a DPRNG library. Section 6 presents performance results measuring the overhead of runtime support for pedigrees in MIT Cilk, as well as the overheads of DOTMIX in Cilk Plus on synthetic and realistic applications. Section 7 describes related work, and Section 8 offers some concluding remarks.

2. Pedigrees

A pedigree scheme uniquely identifies each strand of a dthreaded program in a scheduler-independent manner. This section introduces "spawn pedigrees," a simple pedigree scheme that can be easily maintained by a dthreading runtime system. We describe the changes that Intel implemented in their Cilk Plus concurrency platform to implement spawn pedigrees. Their runtime support provides an application programming interface (API) that allows user programmers to access the spawn pedigree of a strand, which can be used to implement a pedigree-based DPRNG scheme. We finish by describing an important optimization for parallel loops, called "flattening."

We shall focus on dialects of Cilk [20, 28, 34] to contextualize our discussion, since we used Cilk platforms to implement the spawn-pedigree scheme and study its empirical behavior. The runtime support for pedigrees that we describe can be adapted to other dthreading platforms, however, which we discuss in Section 8.

Background on dynamic multithreading

Let us first review the Cilk programming model, which provides the basic dthreading abstraction of ***fork-join parallelism*** in which dthreads are spawned off as parallel subroutines. Cilk extends C with two main keywords: `spawn` and `sync`.[2] A program's logical parallelism is exposed using the keyword `spawn`. In a function F, when a function invocation G is preceded by the keyword `spawn`, the function G is ***spawned***, and the scheduler may continue to execute the ***continuation*** of F — the statement after the `spawn` of G — in parallel with G, without waiting for G to return. The complement of `spawn` is the keyword `sync`, which acts as a local barrier and joins together the parallelism specified by `spawn`. The Cilk runtime system ensures that statements after `sync` are not executed until all functions spawned before the `sync` statement have completed and returned. Cilk's linguistic constructs allow a programmer to express the logical parallelism in a program in a processor-oblivious fashion.

Dthreading platforms enable a wide range of applications to execute deterministically by removing a major source of nondeterminism: load-balancing. Cilk's nondeterministic scheduler, for example, is implemented as a collection of ***worker*** pthreads that cooperate to load-balance the work of the computation. The Cilk run-

[2] The Cilk++ [34] and Cilk Plus [28] platforms use the keywords `cilk_spawn` and `cilk_sync`. They also include a `cilk_for` keyword for defining a parallel `for` loop, which can be implemented in terms of `spawn` and `sync`.

time employs ***randomized work-stealing*** [5, 20], where a worker posts parallel work locally, rather than attempting to share it when the parallel work is spawned, and idle workers become ***thieves*** who look randomly among their peers for ***victims*** with excess work. When a thief finds a victim, it ***steals*** a function ***frame*** from the victim, and resumes execution of the frame by executing the continuation after a `spawn` statement. Cilk-style dynamic multithreading encapsulates the nondeterminacy of the scheduler, enabling application codes without determinacy races[3] [18] to produce deterministic results regardless of how they are scheduled.

Pedigree schemes

Pedigrees are deterministic labels for the executed instructions in a dthreaded program execution that partition the instructions into valid strands. For the remainder of this section, assume that the dthreaded program in question would be deterministic if each RNG call in the program always returned the same random number on every execution. For such computations, a pedigree scheme maintains two useful properties:

1. ***Schedule independence***: For any instruction x, the value of the pedigree for x, denoted $J(x)$, does not depend on how the program is scheduled on multiple processors.
2. ***Strand uniqueness***: All instructions with the same pedigree form a strand.

Together, Properties 1 and 2 guarantee that pedigrees identify strands of a dthreaded program in a deterministic fashion, regardless of scheduling. Therefore, one can generate a random number for each strand by simply hashing its pedigree.

The basic idea of a pedigree scheme is to name a given strand by the path from the root of the ***invocation tree*** — the tree of function (instances) where F is a ***parent*** of G, denoted $F = \texttt{parent}(G)$, if F spawns or calls G. Label each instruction of a function with a ***rank***, which is the number of calls, spawns, or syncs that precede it in the function. Then the pedigree of an instruction x can be encoded by giving its rank and a list of ancestor ranks, e.g., the instruction x might have rank 3 and be the 5th child of the 1st child of the 3rd child of the 2nd child of the root, and thus its pedigree would be $J(x) = \langle 2, 3, 1, 5, 3\rangle$. Such a scheme satisfies Property 1, because the invocation tree is the same no matter how the computation is scheduled. It satisfies Property 2, because two instructions with the same pedigree cannot have a spawn or sync between them.

Spawn pedigrees improve on this simple scheme by defining ranks using only spawns and syncs, omitting calls and treating called functions as being "inlined" in their parents. We can define spawn pedigrees operationally in terms of a serial execution of a dthreaded program. The runtime system conceptually maintains a stack of ***rank counters***, where each rank counter corresponds to an instance of a spawned function. Program execution begins with a single rank counter with value 0 on the stack for the root (`main`) function F_0. Three events cause the rank-counter stack to change:

1. On a spawn of a function G, push a new rank counter with value 0 for G onto the bottom of the stack.
2. On a return from the spawn of G, pop the rank counter (for G) from the bottom of the stack, and then increment the rank counter at the bottom of the stack.
3. On a `sync` statement inside a function F, increment the rank counter at the bottom of the stack.

For any instruction x, the pedigree $J(x)$ is simply the sequence of ranks on the stack when x executes. Figure 1 shows the Cilk code for a recursive Fibonacci calculation and the corresponding invocation tree for an execution of `fib(4)` with spawn pedigrees labeled on instructions. Intuitively, the counter at the bottom of the rank-counter stack tracks the rank of the currently executing instruction x with respect to the spawned ancestor function closest to x. Thus, the increment at the bottom of the stack occurs whenever resuming the continuation of a `spawn` or a `sync` statement. This operational definition of spawn pedigrees satisfies Property 2, because an increment occurs whenever any parallel control is reached and the values of the pedigrees are strictly increasing according to a lexicographic order. Because a spawn pedigree is dependent only on the invocation tree, spawn pedigrees satisfy Property 1.

Runtime support for spawn pedigrees

Supporting spawn pedigrees in parallel in a dthreaded program is simple but subtle. Let us first acquire some terminology. We extend the definition of "parent" to instructions, where for any instruction x, the ***parent*** of x, denoted $\texttt{parent}(x)$, is the function that executes x. For any nonroot function F, define the ***spawn parent*** of F, denoted $\texttt{spParent}(F)$, as $\texttt{parent}(F)$ if F was spawned, or $\texttt{spParent}(\texttt{parent}(F))$ if F was called. Intuitively, $\texttt{spParent}(F)$ is the closest proper ancestor of F that is a spawned function. Define the ***spawn parent*** of an instruction x similarly: $\texttt{spParent}(x) = \texttt{spParent}(\texttt{parent}(x))$. The ***rank*** of an instruction x, denoted $R(x)$, corresponds to the value in the bottom-most rank counter at the time x is executed in a serial execution, and each more-distant spawn parent in the ancestry of x directly maps to a rank counter higher in the stack.

The primary complication for maintaining spawn pedigrees during a parallel execution is that while one worker p is executing an instruction x in $F = \texttt{spParent}(x)$, another worker p' may steal a continuation in F and continue executing, conceptually modifying the rank counter for F. To eliminate this complication, when p spawns a function G from F, it saves $R(G)$ — the rank-counter value of F when G was spawned — into the frame of G, thereby guaranteeing that any query of the pedigree for x has access to the correct rank, even if p' has resumed execution of F and incremented its rank counter.

Figure 2 shows an API that allows a currently executing strand s to query its spawn pedigree. For any instruction x belonging to a strand s, this API allows s to walk up the chain of spawned functions along the x-to-root path in the invocation tree and access the appropriate rank value for x and each ancestor spawned function. The sequence of ranks discovered along this walk is precisely the reverse of the pedigree $J(x)$.

We persuaded Intel to modify its Cilk Plus [28] concurrency platform to include pedigrees. The Intel C/C++ compiler with Cilk Plus compiles the spawning of a function G as a call to a ***spawn-wrapper*** function $\widehat{G}$, which performs the necessary runtime manipulations to effect the spawn, one step of which is calling the function G. Thus, for any function G, we have $\texttt{spParent}(G) = \widehat{G}$, and for any instruction x, the pedigree $J(x)$ has a rank counter for each spawn-wrapper ancestor of x.

Implementing this API in Cilk Plus requires additional storage in spawn-wrapper frames and in the state of each worker thread. For every spawned function F, the spawn wrapper $\widehat{F}$ stores the following rank information in F's frame:

- $\widehat{F} \rightarrow \mathit{brank}$: a 64-bit[4] value that stores $R(F)$.
- $\widehat{F} \rightarrow \mathit{parent}$: the pointer to $\texttt{spParent}(\widehat{F})$.

In addition, every worker p maintains two values in worker-local storage for its currently executing instruction x:

- $p \rightarrow \mathit{current\text{-}frame}$: the pointer to $\texttt{spParent}(x)$.
- $p \rightarrow \mathit{rank}$: a 64-bit value storing $R(x)$.

As Figure 2 shows, to implement the API, the runtime system reads these fields to report a spawn pedigree. In terms of the operational

[3] Also called ***general races*** [39].

[4] A 64-bit counter never overflows in practice, since 2^{64} is a *big* number.

```
1  int main (void) {
2    int x = fib(4);
3    printf("x=%d\n", x);
4    return (x);
5  }

6  int fib(int n) {
7    if (n < 2) return n;
8    else {
9      int x, y;
10     x = spawn fib(n-1);
11     y = fib(n-2);
12     sync;
13     return (x+y);
14   }
15 }
```

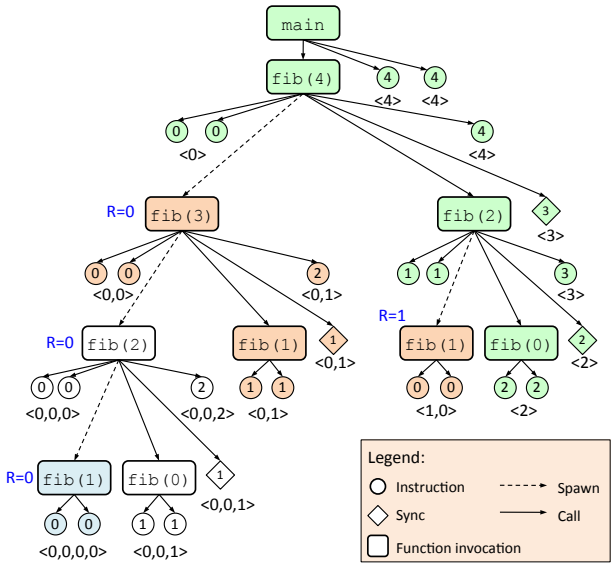

Figure 1: Cilk code for a recursive Fibonacci calculation and the invocation tree for an execution of `fib(4)`. Pedigrees are labeled for each instruction. For example, `sync` instruction in `fib(4)` has pedigree $\langle 3 \rangle$. A left-to-right preorder traversal of the tree represents the serial execution order. For example, for the children of the node for `fib(4)`, the first two instructions with rank 0 correspond to lines 7 and 9 from Figure 1, the subtree rooted at node `fib(4)` between `fib(3)` and the sync node corresponds to the execution of the sync block (lines 10–12), and the last instruction with rank 4 corresponds to the return in line 13. Instructions and functions are labeled with their ranks. For example, `fib(3)` has a rank of 0.

Function	Description	Implementation
RANK()	Returns $R(x)$	Returns $p \rightarrow rank$.
SPPARENT()	Returns spParent(x)	Returns $p \rightarrow current\text{-}frame$.
RANK($\widehat{F}$)	Returns $R(\widehat{F})$	Returns $\widehat{F} \rightarrow brank$.
SPPARENT($\widehat{F}$)	Returns spParent$(\widehat{F})$	Returns $\widehat{F} \rightarrow parent$.
STRANDBREAK()	Ends the currently-executing strand	$p \rightarrow rank{+}{+}$.

Figure 2: An API for spawn pedigrees in Intel Cilk Plus. In these operations, x is the currently executing instruction for a worker, and $\widehat{F}$ is a spawn-wrapper function which is an ancestor of x in the computation tree. These operations allow the worker to walk up the computation tree to compute $J(x)$. A worker can also call STRANDBREAK() to end its currently executing strand.

definition of spawn pedigrees, the $rank$ field in p holds the bottom-most rank counter on the stack for the instruction x that p is currently executing.

To maintain these fields, the runtime system requires additional storage to save and restore the current spawn-parent pointer and rank counter for each worker whenever it enters or leaves a nested spawned function. In particular, we allocate space for a rank and parent pointer in the stack frame of every ***Cilk function*** — function that can contain `spawn` and `sync` statements:

- $G \rightarrow rank$: a 64-bit value that stores $R(x)$ for some instruction x with spParent$(x) = G$.
- $G \rightarrow sp\text{-}rep$: the pointer to spParent(G).

These fields are only used to save and restore the corresponding fields for a worker p. Whenever p is executing a Cilk function G which spawns a function F, it saves its fields into G before beginning execution of F. When a worker p' (which may or may not be p) resumes the continuation after the `spawn` statement, p' restores its values from G. Similarly, saving and restoring also occurs when a worker stalls at a `sync` statement. Figure 3 summarizes the runtime operations needed to maintain spawn pedigrees.

Although the implementation of spawn pedigrees in Cilk Plus required changes to the Intel compiler, ordinary C/C++ functions need not be recompiled for the pedigree scheme to work. The reason is that the code in Figure 3 does not perform any operations on entry or exit to called functions. Consequently, the scheme

(a) On a spawn of F from G:
1 $G \rightarrow rank = p \rightarrow rank$
2 $G \rightarrow sp\text{-}rep = p \rightarrow current\text{-}frame$
3 $\widehat{F} \rightarrow brank = G \rightarrow rank$
4 $\widehat{F} \rightarrow parent = G \rightarrow sp\text{-}rep$
5 $p \rightarrow rank = 0$
6 $p \rightarrow current\text{-}frame = \widehat{F}$

(b) On stalling at a sync in G:
1 $G \rightarrow rank = p \rightarrow rank$

(c) On resuming the continuation of a spawn or sync in G:
1 $p \rightarrow rank = G \rightarrow rank{+}{+}$
2 $p \rightarrow current\text{-}frame = G \rightarrow sp\text{-}rep$

Figure 3: How a worker p maintains spawn pedigrees. **(a)** $\widehat{F}$ is a pointer to the spawn wrapper of the function F being spawned, and G is a pointer to the frame of the Cilk function that is spawning F. **(b)** G is the Cilk function that is attempting to execute a `sync`. The value $p \rightarrow current\text{-}frame$ need not be saved into $G \rightarrow sp\text{-}rep$, because the first `spawn` in G will have saved this value already, and this value is fixed for G. **(c)** G is the Cilk function containing the continuation being resumed.

works even for programs that incorporate legacy and third-party C/C++ binaries.

To implement DPRNG's, it is useful to extend the API in Figure 2 to include a STRANDBREAK function that allows the DPRNG to end a currently executing strand explicitly. In particular, if a user requests multiple random numbers from a DPRNG in a serial sequence of instructions, the DPRNG can let each call to get a random number terminate a strand in that sequence using this function, meaning that the DPRNG produces at most one random

number per strand. Like a spawn or sync, when a worker p encounters a STRANDBREAK call, the next instruction after the STRANDBREAK that p executes is guaranteed to be part of a different strand, and thus have a different pedigree. The STRANDBREAK function is implemented by incrementing $p \rightarrow rank$.

Pedigree flattening for parallel loops

As an optimization, we can simplify spawn pedigrees for parallel loops. Intel Cilk Plus provides a parallel looping construct called `cilk_for`, which allows all the iterations of the loop to execute in parallel. The runtime system implements `cilk_for` using a balanced binary recursion tree implemented with spawn's and sync's, where each leaf performs a chunk of iterations serially. Rather than tracking ranks at every level of this recursion tree, the Cilk Plus pedigree scheme conceptually "cuts out the middle man" and ***flattens*** all the iterations of the `cilk_for` loop so that they share a single level of pedigree. The idea is simply to let the rank of an iteration be the loop index. Consequently, iterations in a `cilk_for` can be referenced within the `cilk_for` by a single value, rather than a path through the binary recursion tree. To ensure that spawn and sync statements within a loop iteration do not affect the pedigrees of other loop iterations, the body of each loop iteration is treated as a spawned function with respect to its pedigree. This change simplifies the pedigrees generated for `cilk_for` loops by reducing the effective spawn depth of strands within the `cilk_for` and as Section 6 shows, the cost of reading the pedigree as well.

3. A pedigree-based DPRNG

This section presents DOTMIX, a high-quality statistically random pedigree-based DPRNG. DOTMIX operates by hashing the pedigree and then "mixing" the result. We investigate theoretical principles behind the design of DOTMIX, which offer evidence that pedigree-based DPRNG's can generate pseudorandom numbers of high quality for real applications. We also examine empirical test results using Dieharder [9], which suggest that DOTMIX generates high-quality random numbers in practice.

The DOTMIX *DPRNG*

At a high level, DOTMIX generates random numbers in two stages. First, DOTMIX compresses the pedigree into a single machine word while attempting to maintain uniqueness of compressed pedigrees. Second, DOTMIX "mixes" the bits in the compressed pedigree to produce a pseudorandom value.

To describe DOTMIX formally, let us first establish some notation. We assume that our computer has a word width of w bits. We choose a prime $p < m = 2^w$ and assume that each rank j_i in the pedigree falls in the range $1 \leq j_i < p$. Our library implementation of DOTMIX simply increments each rank in the spawn-pedigree scheme from Section 2 to ensure that ranks are nonzero. Let $\mathbb{Z}_m$ denote the universe of (unsigned) w-bit integers over which calculations are performed, and let $\mathbb{Z}_p$ denote the finite field of integers modulo p. Consequently, we have $\mathbb{Z}_p \subseteq \mathbb{Z}_m$. We assume that the spawn depth $d(x)$ for any instruction x in a dthreaded program is bounded by $d(x) \leq D$.[5] A pedigree $J(x)$ for an instruction x at spawn depth $d(x)$ can then be represented by a length-D vector $J(x) = \langle j_1, j_2, \ldots, j_D \rangle \in \mathbb{Z}_p^D$, where $j_i = 0$ for $D - d(x)$ entries. Which entries are padded out with 0 does not matter for the theorems that follow, as long as distinct pedigrees remain distinct. In our implementation, we actually pad out the first entries.

For a given pedigree $J \in \mathbb{Z}_p^D$, the random number produced by DOTMIX is the hash $h(J)$, where $h(J)$ is the composition of

1. a ***compression function*** $c : \mathbb{Z}_p^D \rightarrow \mathbb{Z}_p$ that hashes each pedigree J into a single integer $c(J)$ less than p, and
2. a ***mixing function*** $\mu : \mathbb{Z}_m \rightarrow \mathbb{Z}_m$ that "mixes" the compressed pedigree value $c(J)$.

Let us consider each of these functions individually.

The goal of a compression function c is to hash each pedigree J into an integer in $\mathbb{Z}_p$ such that the probability of a ***collision*** — two distinct pedigrees hashing to the same integer — is small. To compress pedigrees, DOTMIX computes a dot product of the pedigree with a vector of random values [17]. More formally, DOTMIX uses a compression function c chosen uniformly at random from the following hash family.

DEFINITION 1. *Let $\Gamma = \langle \gamma_1, \gamma_2, \ldots, \gamma_D \rangle$ be a vector of integers chosen uniformly at random from $\mathbb{Z}_p^D$. Define the compression function $c_\Gamma : \mathbb{Z}_p^D \rightarrow \mathbb{Z}_p$ by*

$$c_\Gamma(J) = \left(\sum_{k=1}^{D} \gamma_k j_k \right) \bmod p \, ,$$

where $J = \langle j_1, j_2, \ldots, j_D \rangle \in \mathbb{Z}_p^D$. The **DOTMIX** ***compression-function family*** *is the set*

$$C_{\text{DOTMIX}} = \left\{ c_\Gamma : \Gamma \in \mathbb{Z}_p^D \right\} \, .$$

The next theorem proves that the probability is small that a randomly chosen compression function $c_\Gamma \in C_{\text{DOTMIX}}$ causes two distinct pedigrees to collide.

THEOREM 1. *Let $c_\Gamma \in C_{\text{DOTMIX}}$ be a randomly chosen compression function. Then for any two distinct pedigrees $J, J' \in \mathbb{Z}_p^D$, we have $\Pr\{c_\Gamma(J) = c_\Gamma(J')\} = 1/p$.*

PROOF. Let $J = \langle j_1, j_2, \ldots, j_D \rangle$, and let $J' = \langle j'_1, j'_2, \ldots, j'_D \rangle$. Because $J \neq J'$, there must exist some index k in the range $1 \leq k \leq D$ such that $j_k \neq j'_k$. Without loss of generality, assume that $k = 1$. We therefore have (modulo p) that

$$c_\Gamma(J) - c_\Gamma(J') = \gamma_1 j_1 - \gamma_1 j'_1 + \sum_{k=2}^{D} \gamma_k j_k - \sum_{k=2}^{D} \gamma_k j'_k \, ,$$

and thus $c_\Gamma(J) - c_\Gamma(J') = 0$ implies that

$$(j_1 - j'_1)\gamma_1 = \sum_{k=2}^{D} \gamma_k (j'_k - j_k) \, .$$

Consider fixed values for J, J', and $\gamma_2, \ldots, \gamma_k$. Let $a = j_1 - j'_1 \neq 0$, let $x = \gamma_1$, and let $y = \sum_{k=2}^{D} \gamma_k (j'_k - j_k)$.

We now show that for any fixed choice of $y \in \mathbb{Z}_p$ and nonzero $a \in \mathbb{Z}_p$, there is exactly one choice of $x \in \mathbb{Z}_p$ such that $y = ax$, namely, $x = a^{-1}y$. For the sake of contradiction, suppose that there are two distinct values x_1 and x_2 such that $y = ax_1 = ax_2$. This supposition implies that $0 = ax_1 - ax_2 = a(x_1 - x_2)$ modulo p, which is satisfied if and only if either $a = 0$ or $x_1 - x_2 = 0$, since p is prime. Because $a \neq 0$, we must have $x_1 - x_2 = 0$, contradicting the supposition that x_1 and x_2 are distinct. Therefore, there is one value of x satisfying $y = ax$. Because $x = \gamma_1$ is a randomly chosen value from $\mathbb{Z}_p$, the probability that x satisfies $y = ax$ is $1/p$. □

This low probability of collision allows DOTMIX to generate many random numbers with a low probability that any pair collide. By Boole's Inequality [16, p. 1195], given n distinct pedigrees and using a random function from C_{DOTMIX}, the probability of a collision among any of their compressed pedigrees is at most $\binom{n}{2}(1/p) = n(n-1)/2p$. For the choice of the prime $p = 2^{64} - 59$,

[5] A reasonable upper bound is $D = 100$.

for example, the probability that hashing 5 million pedigrees results in a collision in their compressed values is less than 1 in a million.

Although the compression function effectively hashes a pedigree into an integer less than p with a small probability of collision, two similar pedigrees may yet have "similar" hash values, whereas we would like them to be statistically "dissimilar." In particular, for a given compression function c_Γ, two pedigrees that differ only in their kth coordinate differ in their compressions by a predictable multiple of γ_k. To reduce the statistical correlation in generated random values, DOTMIX "mixes" the bits of a compressed pedigree using a mixing function based on the RC6 block cipher [15, 43]. For any w-bit input z, where we assume that w is even, let $\phi(z)$ denote the function that swaps the high- and low-order $w/2$ bits of z, that is,

$$\phi(z) = \left\lfloor \frac{z}{\sqrt{m}} \right\rfloor + \sqrt{m}\,(z \bmod \sqrt{m}) \ ,$$

and let

$$f(z) = \phi(2z^2 + z) \bmod m \ .$$

DOTMIX uses the mixing function $\mu(z) = f^{(r)}(z)$, which applies r rounds of $f(z)$ to the compressed pedigree value z. Contini *et al.* [15] prove that $f(z)$ is a one-to-one function, and hence, for two distinct pedigrees J and J', the probability that $\mu(c(J)) = \mu(c(J'))$ is $1/p$, unchanged from Theorem 1.

DOTMIX allows a seed to be incorporated into the hash of a pedigree. The random number generated for a pedigree J is actually the value of a hash function $h(J, \sigma)$, where σ is a seed. Such a seed may be incorporated into the computation of $\mu(c_\Gamma(J))$ in several ways. For instance, we might XOR or otherwise combine the seed with the result of $\mu(c_\Gamma(J))$, computing, for example, $h(J, \sigma) = \sigma \oplus \mu(c_\Gamma(J))$. This scheme does not appear to be particularly good, because it lacks much statistical variation between the numbers generated by one seed versus another. A better scheme, which DOTMIX adopts, is to combine the seed with the compressed pedigree *before* mixing. DOTMIX is formally defined as follows.

DEFINITION 2. *For a given number r of mixing rounds, the* DOTMIX *DPRNG generates a random number by hashing the current pedigree $J = \langle j_1, j_2, \ldots, j_D \rangle$ with a seed σ according to the function*

$$\begin{aligned} h_\Gamma(J, \sigma) &= \mu(c_\Gamma(J, \sigma)) \\ &= f^{(r)}(c_\Gamma(J, \sigma)) \ , \end{aligned}$$

where

$$c_\Gamma(J, \sigma) = (\sigma + c_\Gamma(J)) \bmod m$$

and $c_\Gamma(J)$ is a hash function chosen uniformly at random from the DOTMIX *compression-function family C_{DOTMIX}.*

The statistical quality of DOTMIX

Although DOTMIX is not a cryptographically secure RNG, it appears to generate high-quality random numbers as evinced by Dieharder [9], a collection of statistical tests designed to empirically test the quality of serial RNG's. Figure 4 summarizes the Dieharder test results for DOTMIX and compares them to those of the Mersenne twister [36], whose implementation is provided in the GNU Scientific Library [21]. As Figure 4 shows, with 2 or more iterations of the mixing function, DOTMIX generates random numbers of comparable quality to Mersenne twister. In particular, Mersenne twister and DOTMIX with 2 or more mixing iterations generally fail the same set of Dieharder tests. Because the Dieharder tests are based on P-values [26], it is not surprising to see statistical variation in the number of "Weak" and "Poor" results even from high-quality RNG's. We report the median of 5 runs using 5 different seeds to reduce this variation.

Test	r	*Passed*	*Weak*	*Poor*	*Failed*
Mersenne twister	–	79	7	7	14
DOTMIX (tree)	16	83	6	4	14
	8	84	6	4	13
	4	81	5	7	14
	2	81	5	5	16
	1	3	2	3	99
	0	0	0	0	107
DOTMIX (loop)	16	82	2	8	15
	8	79	6	8	14
	4	79	5	8	15
	2	79	4	8	16
	1	55	2	8	42
	0	2	0	1	104
LCGMIX (tree)	4	84	4	6	13
	0	24	6	21	56

Figure 4: A summary of the quality of DOTMIX on the Dieharder tests compared to the Mersenne twister. For the entries labeled "tree," DOTMIX generates 3^{20} random numbers in a parallel divide-and-conquer ternary tree fashion using spawns. For the entries labeled "loop," a `cilk_for` loop generates 3^{20} random numbers. The column labeled r indicates the number of mixing iterations. Each successive column counts the number of tests that produced the given status, where the status of each test was computed from the median of 5 runs of the generator using 5 different seeds. The table also summarizes the Dieharder test results for LCGMIX.

When using Dieharder to measure the quality of a parallel RNG, we confronted the issue that Dieharder is really designed to measure the quality of serial RNG's. Since all numbers are generated by a serial RNG in a linear order, this order provides a natural measure of "distance" between adjacent random numbers, which Dieharder can use to look for correlations. When using an RNG for a parallel program, however, this notion of "distance" is more complicated, because calls to the RNG can execute in parallel. The results in Figure 4 use numbers generated in a serial execution of the (parallel) test program, which should maximize the correlation between adjacent random numbers due to similarities in the corresponding pedigrees. In principle, another execution order of the same program could generate random numbers in a different order and lead to different Dieharder test results.

As a practical matter, DOTMIX uses $r = 4$ mixing iterations to generate empirically high-quality random numbers. The difference in performance per call to DOTMIX with $r = 0$ and with $r = 4$ is less than 2%, and thus DOTMIX can generate high-quality random numbers without sacrificing performance.

4. Other pedigree-based DPRNG's

This section investigates several other pedigree-based schemes for DPRNG's. Principal among these schemes is LCGMIX, which uses a compression function based on linear congruential generators and the same mixing function as DOTMIX. We prove that the probability that LCGMIX's compression function generates a collision is small, although not quite as small as for DOTMIX. We examine Dieharder results which indicate that LCGMIX is statistically good. We also discuss alternative DPRNG schemes and their utility. These DPRNG's demonstrate that pedigrees can enable not only DOTMIX, but a host of other DPRNG implementations. We close by observing a theoretical weakness with DOTMIX which would be remedied by a 4-independent compression scheme.

The LCGMIX *DPRNG*

LCGMIX is related to the "Lehmer tree" DPRNG scheme of [19]. LCGMIX uses a family of compression functions for pedigrees that generalize linear congruential generators (LCG's) [29, 32].

LCGMIX then "RC6-mixes" the compressed pedigree to generate a pseudorandom value using the same mixing function as DOTMIX.

The basic idea behind LCGMIX is to compress a pedigree by combining each successive rank using an LCG that operates modulo a prime p, where p is close to but less than $m = 2^w$, where w is the computer word width. LCGMIX uses only three random nonzero values $\alpha, \beta, \gamma \in \mathbb{Z}_p$, rather than a table, as DOTMIX does. Specifically, for an instruction x at depth $d = d(x)$ with pedigree $J(x) = \langle j_1, j_2, \ldots, j_d \rangle$, the LCGMIX compression function performs the following recursive calculation modulo p:

$$X_d = \begin{cases} \gamma & \text{if } d = 0, \\ \alpha X_{d-1} + \beta j_d & \text{if } d > 0. \end{cases}$$

The value X_d is the compressed pedigree. Thus, the LCGMIX compression function need only perform two multiplications modulo p and one addition modulo p per rank in the pedigree.

Assume, as for DOTMIX, that the spawn depth $d(x)$ for any instruction x in a dthreaded program is bounded by $d(x) \leq D$. The family of compression functions used by LCGMIX can be defined as follows.

DEFINITION 3. *Let α, β, γ be nonzero integers chosen uniformly at random from $\mathbb{Z}_p$. Define $c_{\alpha,\beta,\gamma} : \bigcup_{d=1}^{\infty} \mathbb{Z}_p^d \to \mathbb{Z}_p$ by*

$$c_{\alpha,\beta,\gamma}(J) = \left(\alpha^d \gamma + \beta \sum_{k=1}^{d} \alpha^{d-k} j_k \right) \bmod p \,,$$

where $J = \langle j_1, j_2, \ldots, j_d \rangle \in \mathbb{Z}_p^d$. The **LCGMIX *compression-function family*** *is the set of functions*

$$C_{\text{LCGMIX}} = \{ c_{\alpha,\beta,\gamma} : \alpha, \beta, \gamma \in \mathbb{Z}_p - \{0\} \} \,.$$

The next theorem shows that the probability a randomly chosen compression function $c_{\alpha,\beta,\gamma} \in C_{\text{LCGMIX}}$ hashes two distinct pedigrees to the same value is small, although not quite as small as for DOTMIX.

THEOREM 2. *Let $c_{\alpha,\beta,\gamma} \in C_{\text{LCGMIX}}$ be a randomly chosen compression function. Then for any two distinct pedigrees $J \in \mathbb{Z}_p^d$ and $J' \in \mathbb{Z}_p^{d'}$ we have $\Pr\{c_{\alpha,\beta,\gamma}(J) = c_{\alpha,\beta,\gamma}(J')\} \leq D/(p-1)$, where $D = \max\{d, d'\}$.*

PROOF. Let $J = \langle j_1, j_2, \ldots, j_d \rangle$ and $J' = \langle j'_1, j'_2, \ldots, j'_{d'} \rangle$. The important observation is that the difference $c_{\alpha,\beta,\gamma}(J) - c_{\alpha,\beta,\gamma}(J')$ is a nonzero polynomial in α of degree at most D with coefficients in $\mathbb{Z}_p$. Thus, there are at most D roots to the equation $c_{\alpha,\beta,\gamma}(J) - c_{\alpha,\beta,\gamma}(J') = 0$, which are values for α that cause the two compressed pedigrees to collide. Since there are $p-1$ possible values for α, the probability of collision is at most $D/(p-1)$. □

This pairwise-collision probability implies a theoretical bound on how many random numbers LCGMIX can generate before one would expect a collision between any pair of numbers in the set. By Boole's Inequality [16, p. 1195], compressing n pedigrees with a random function from C_{LCGMIX} gives a collision probability between any pair of those n compressed pedigrees of at most $\binom{n}{2} D/(p-1) = n(n-1)D/2(p-1)$. With $p = 2^{64} - 59$ and making the reasonable assumption that $D \leq 100$, the probability that compressing 500,000 pedigrees results in a collision is less than 1 in a million. As can be seen, 500,000 pedigrees is a factor of 10 less than the 5 million for DOTMIX for the same probability. Since our implementation of LCGMIX was no faster than our implementation of DOTMIX per function call, we favored the stronger theoretical guarantee of DOTMIX for the Cilk Plus library.

We tested the quality of random numbers produced by LCGMIX using Dieharder, producing the results in Figure 4. The data suggest that, as with DOTMIX, $r = 4$ mixing iterations in LCGMIX are sufficient to provide random numbers whose statistical quality is comparable to those produced by the Mersenne twister.

Further ideas for DPRNG's

We can define DPRNG's using families of compression functions that exhibit stronger theoretical properties or provide faster performance than either DOTMIX or LCGMIX.

One alternative is to use ***tabulation hashing*** [11] to compress pedigrees, giving compressed pedigree values that are 3-independent and have other strong theoretical properties [41]. This DPRNG is potentially useful for applications that require stronger properties from their random numbers. To implement this scheme, the compression function treats the pedigree as a bit vector whose 1 bits select entries in a table of random values to XOR together.

As another example which favors theoretical quality over performance, a DPRNG could be based on compressing the pedigree with a SHA-1 [38] hash, providing a cryptographically secure compression function that would not require any mixing to generate high-quality random numbers. Other cryptographically secure hash functions could be used as well. While cryptographically secure hash functions are typically slow, they would allow the DPRNG to provide pseudorandom numbers with very strong theoretical properties, which may be important for some applications.

On the other side of the performance-quality spectrum, a DPRNG could be based on compressing a pedigree using a faster hash function. One such function is the hash function used in UMAC [3], which performs half the multiplications of DOTMIX's compression function. The performance of the UMAC compression scheme and the quality of the DPRNG it engenders offers an interesting topic for future research.

4-independent compression of pedigrees

Although Theorem 1 shows that the probability is small that DOTMIX's compression function causes two pedigrees to collide, DOTMIX contains a theoretical weakness. Consider two distinct pedigrees J_1 and J_2 of length D, and suppose that DOTMIX maps J_1 and J_2 to the same value, or more formally, that DOTMIX chooses a compression function c_Γ such that $c_\Gamma(J_1) = c_\Gamma(J_2)$. Let $J + \langle j \rangle$ denote the pedigree that results from appending the rank j to the pedigree J. Because J_1 and J_2 both have length D, it follows that

$$\begin{aligned} c_\Gamma(J_1 + \langle j \rangle) &= c_\Gamma(J_1) + \gamma_{D+1} j \\ &= c_\Gamma(J_2) + \gamma_{D+1} j \\ &= c_\Gamma(J_2 + \langle j \rangle) \,. \end{aligned}$$

Thus, DOTMIX hashes the pedigrees $J_1 + \langle j \rangle$ and $J_2 + \langle j \rangle$ to the same value, regardless of the value of j. In other words, one collision in the compression of the pedigrees for two strands, however rare, may result in many ancillary collisions.

To address this theoretical weakness, a DPRNG scheme might provide the guarantee that if two pedigrees for two strands collide, then the probability remains small that any the pedigrees collide for any other pair of strands. A 4-independent hash function [46] would achieve this goal by guaranteeing that the probability is small that any sequence of 4 distinct pedigrees hash to any particular sequence of 4 values. Tabulation-based 4-independent hash functions for single words are known [45], but how to extend these techniques to hash pedigrees efficiently is an intriguing open problem.

5. A scoped DPRNG library interface

This section presents the programming interface for a DPRNG library that we implemented for Cilk Plus. This interface demon-

```
1 template <typename T>
2 class DPRNG {
3   DPRNG();                      // Constructor
4   ~DPRNG();                     // Destructor
5   DPRNG_scope current_scope(); // Get current scope
6   void set(uint64_t seed, DPRNG_scope scope); // Init
7   uint64_t get();               // Get random #
8 };
```

Figure 5: A C++ interface for a pedigree-based DPRNG suitable for use with Cilk Plus. The type `T` of the DPRNG object specifies a particular DPRNG library, such as DOTMIX, that implements this interface. In addition to accepting an argument for an initial seed, the initialization method for the DPRNG in line 6 also requires an lexical scope, restricting the scope where the DPRNG object can be used.

strates how programmers can use a pedigree-based DPRNG library in applications. The interface uses the notion of "scoped" pedigrees, which allow DPRNG's to compose easily.

Scoped pedigrees solve the following problem. Suppose that a dthreaded program contains a parallel subcomputation that uses a DPRNG, and suppose that the program would like to run this subcomputation the same way twice. Using scoped pedigrees, the program can guarantee that both runs generate the exact same random numbers, even though corresponding RNG calls in the subcomputations have different pedigrees globally.

Programming interface

Figure 5 shows a C++ interface for a DPRNG suitable for use with Cilk Plus. It resembles the interface for an ordinary serial RNG, but it constrains when the DPRNG can be used to generate random numbers by defining a "scope" for each DPRNG instance. The `set` method in line 6 initializes the DPRNG object based on two quantities: an initial seed and a scope. The seed is the same as for an ordinary serial RNG. The ***scope*** represented by a pedigree J is the set of instructions whose pedigrees have J as a common prefix. Specifying a scope (represented by) J to the DPRNG object `rand` restricts the DPRNG to generate numbers only within that scope and to ignore the common prefix J when generating random numbers. By default, the programmer can pass in the global scope $\langle 0 \rangle$ to let the DPRNG object be usable anywhere in the program. The interface allows programmers to limit the scope of a DPRNG object by getting an explicit scope (line 5) and setting the scope of a DPRNG object (line 6).

Figure 6 demonstrates how restricting the scope of a DPRNG can be used to generate repeatable streams of random numbers within a single program. Inside `f`, the code in lines 3–4 limits the scope of `rand` so that it generates random numbers only within `f`. Because of this limited scope, the assertion in line 23 holds true. If the programmer sets `rand` with a global scope, then each call to `f` would generate a different value for `sum`, and the assertion would fail.

Intuitively, one can think of the scope as extension of the seed for a serial RNG. To generate exactly the same stream of random numbers in a dthreaded program, one must (1) use the same seed, (2) use the same scope, and (3) have exactly the same structure of spawned functions and RNG calls within the scope. Even if `f` from Figure 6 were modified to generate random numbers in parallel, the use of scoped pedigrees still guarantees that each iteration of the parallel loop in line 16 behaves identically.

Implementation

To implement the `get` method of a DPRNG, we use the API in Figure 2 to extract the current pedigree during the call to `get`, and then we hash the pedigree. The principal remaining difficulty in the DPRNG's implementation is in handling scopes.

```
1 uint64_t f(DPRNG<DotMix>* rand, uint64_t seed, int i) {
2   uint64_t sum = 0;
3   DPRNG_scope scope = rand->current_scope();
4   rand->set(seed, scope);
5   for (int j = 0; j < 15; ++j) {
6     uint64_t val = rand->get();
7     sum += val;
8   }
9   return sum;
10 }
11 int main(void) {
12   const int NSTREAMS = 10;
13   uint64_t sum[NSTREAMS];
14   uint64_t s1 = 0x42;  uint64_t s2 = 31415;
15   // Generate NSTREAMS identical streams
16   cilk_for (int i = 0; i < NSTREAMS; ++i) {
17     DPRNG<DotMix>* rand = new DPRNG();
18     sum[i] = f(rand, s1, i);
19     sum[i] += f(rand, s2, i);
20     delete rand;
21   }
22   for (int i = 1; i < NSTREAMS; ++i)
23     assert(sum[i] == sum[0]);
24   return 0;
25 }
```

Figure 6: A program that generates `NSTREAMS` identical streams of random numbers. Inside function `f`, the code in lines 3–4 limits the scope of `rand` so that it can generate random numbers only within `f`.

Intuitively, a scope can be represented by a pedigree prefix that should be common to the pedigrees of all strands generating random numbers within the scope. Let y be the instruction corresponding to a call to `current_scope`(), and let x be a call to `get`() within the scope $J(y)$. Let $J(x) = \langle j_1, j_2, \ldots, j_{d(x)} \rangle$ and $J(y) = \langle j'_1, j'_2, \ldots, j'_{d(y)} \rangle$. Since x belongs to the scope $J(y)$, it follows that $d(x) \geq d(y)$, and we have $j_{d(y)} \geq j'_{d(y)}$ and $j_k = j'_k$ for all $k < d(y)$. We now define the ***scoped pedigree*** of x with respect to scope $J(y)$ as

$$J_{J(y)}(x) = \langle j_{d(y)} - j'_{d(y)}, j_{d(y)+1}, \ldots, j_{d(x)} \rangle .$$

To compute a random number for $J(x)$ excluding the scope $J(y)$, we simply perform a DPRNG scheme on the scoped pedigree $J_{J(y)}(x)$. For example, DOTMIX computes $\mu(c_\Gamma(J_{J(y)}(x)))$. Furthermore, one can check for scoping errors by verifying that $J(y)$ is indeed the prefix of $J(x)$ via a direct comparison of all the pedigree terms.

Scoped pedigrees allow DPRNG's to optimize the process of reading a pedigree. By hashing scoped pedigrees, a call to the DPRNG need only read a suffix of the pedigree, i.e. the scoped pedigree itself, rather than the entire pedigree. To implement this optimization, each scope may store a pointer to the spawn parent for the deepest rank of the scope, and then the code for reading the pedigree extracts ranks as usual until it observes the spawn parent the scope points to. One problem with this optimization is that a DPRNG may not detect if it is hashing a pedigree outside of its scope. To overcome this problem, DOTMIX supports a separate mode for debugging, in which calls checks their pedigrees term-by-term to verify they are within the scope.

6. Performance results

This section reports on our experimental results investigating the overhead of maintaining pedigrees and the cost of the DOTMIX DPRNG. To study the overhead of tracking pedigrees, we modified the open-source MIT Cilk [20], whose compiler and runtime system were both accessible. We discovered that the overhead of tracking pedigrees is small, having a geometric mean of only 1% on all tested benchmarks. To measure the costs of DOTMIX, we implemented it as a library for a version of Cilk Plus that Intel en-

Application	*Default*	*Pedigree*	*Overhead*
`fib`	11.03	12.13	1.10
`cholesky`	2.75	2.92	1.06
`fft`	1.51	1.53	1.01
`matmul`	2.84	2.87	1.01
`rectmul`	6.20	6.21	1.00
`strassen`	5.23	5.24	1.00
`queens`	4.61	4.60	1.00
`plu`	7.32	7.35	1.00
`heat`	2.51	2.46	0.98
`lu`	7.88	7.25	0.92

Figure 7: Overhead of maintaining 64-bit rank pedigree values for the Cilk benchmarks as compared to the default of MIT Cilk 5.4.6. The experiments were run on an AMD Opteron 6168 system with a single 12-core CPU clocked at 1.9 GHz. All times are the minimum of 15 runs measured in seconds.

gineers had augmented with pedigree support, and we compared its performance to a nondeterministic DPRNG implemented using worker-local Mersenne twister RNG's. Although the price of determinism from using DOTMIX was approximately a factor of 2.3 greater per function call than Mersenne twister on a synthetic benchmark, this price was much smaller on more realistic codes such as a sample sort and Monte Carlo simulations. These empirical results suggest that pedigree-based DPRNG's are amply fast for debugging purposes and that their overheads may be low enough for some production codes.

Pedigree overheads

To estimate the overhead of maintaining pedigrees, we ran a set of microbenchmarks for MIT Cilk with and without support for pedigrees. We modified MIT Cilk 5.4.6 to store the necessary 64-bit rank values and pointers in each frame for spawn pedigrees and to maintain spawn pedigrees at runtime. We then ran 10 MIT Cilk benchmark programs using both our modified version of MIT Cilk and the original MIT Cilk. In particular, we ran the following benchmarks:

- `fib`: Recursive exponential-time calculation of the 40th Fibonacci number.
- `cholesky`: A divide-and-conquer Cholesky factorization of a sparse 2000×2000 matrix with 10,000 nonzeros.
- `fft`: Fast Fourier transform on 2^{22} elements.
- `matmul`: Recursive matrix multiplication of 1000×1000 square matrices.
- `rectmul`: Rectangular matrix multiplication of 2048×2048 square matrices.
- `strassen`: Strassen's algorithm for matrix multiplication on 2048×2048 square matrices.
- `queens`: Backtracking search to count the number of solutions to the 24-queens puzzle.
- `plu`: LU-decomposition with partial pivoting on a 2048×2048 matrix.
- `heat`: Jacobi-type stencil computation on a 4096×1024 grid for 100 timesteps.
- `lu`: LU-decomposition on a 2048×2048 matrix.

The results from these benchmarks, as summarized in Figure 7, show that the slowdown due to spawn pedigrees is generally negligible, having a geometric mean of less than 1%. Although the overheads run as high as 10% for `fib`, they appear to be within measurement noise caused by the intricacies of modern-day processors. For example, two benchmarks actually run measurably faster despite the additional overhead. This benchmark suite gives us confidence that the overhead for maintaining spawn pedigrees should be close to negligible for most real applications.

DPRNG overheads

To estimate the cost of using DPRNG's, we persuaded Intel to modify its Cilk Plus concurrency platform to maintain pedigrees, and then we implemented the DOTMIX DPRNG.[6] We compared DOTMIX's performance, using $r = 4$ mixing iterations, to a nondeterministic parallel implementation of the Mersenne twister on synthetic benchmarks, as well as on more realistic applications. From these results, we estimate that the "price of determinism" for DOTMIX is about a factor of 2.3 in practice on synthetic benchmarks that generate large pedigrees, but it can be significantly less for more practical applications. For these experiments, we coded by hand an optimization that the compiler could but does not implement. To avoid incurring the overhead of multiple worker lookups on every call to generate a random number, within a strand, DOTMIX looks up the worker once and uses it for all calls to the API made by the strand.

We used Intel Cilk Plus to perform three different experiments. First, we used a synthetic benchmark to quantify how DOTMIX performance is affected by pedigree length. Next, we used the same benchmark to measure the performance benefits of flattening pedigrees for `cilk_for` loops. Finally, we benchmarked the performance of DOTMIX on realistic applications that require random numbers. All experiments described in the remainder of this section were run on an Intel Xeon X5650 system with two 6-core CPU's, each clocked at 2.67 GHz. The code was compiled using the Intel C++ compiler v13.0 Beta with the `-O3` optimization flag and uses the Intel Cilk Plus runtime, which together provide support for pedigrees.

First, to understand how the performance of DOTMIX varies with pedigree length, we constructed a synthetic benchmark called CBT. This benchmark successively creates n/k complete binary trees, each with k leaves, which it walks in parallel by spawning two children recursively. The pedigree of each leaf has uniform length $L = 2 + \lg k$, and within each leaf, we call the RNG.

Figure 8 compares the performance of various RNG's on the CBT benchmark, fixing $n = 2^{20}$ random numbers but varying the pedigree length L. These results show that the overhead of DOTMIX increases roughly linearly with pedigree length, but that DOTMIX is still within about a factor of 2 compared to using a Mersenne Twister RNG. From a linear regression on the data from Figure 8, we observed that the cost per additional term in the pedigree for both DOTMIX and LCGMIX was about 15 cycles, regardless of whether $r = 4$ or $r = 16$.[7]

Figure 9 breaks down the overheads of DOTMIX in the CBT benchmark further. To generate a random number, DOTMIX requires looking up the currently executing worker in Cilk (from thread-local storage),[8] reading the pedigree, and then generating a random number. The figure compares the overhead of DOTMIX with the overhead of simply spawning a binary tree with n leaves while performing no computation within each leaf. From this data, we can attribute at least 35% of the execution time of DOTMIX calls to the overhead of simply spawning the tree itself. Also, by measuring the cost of reading a pedigree, we observe that for the longest pedigrees, roughly half of the cost of an RNG call can be attributed to looking up the pedigree itself.

[6] The Intel C++ compiler v12.1 provides compiler and runtime support for maintaining pedigrees in Cilk.

[7] This linear model overestimates the runtime of this benchmark for $L < 4$ (not shown). For small trees, it is difficult to accurately measure and isolate the RNG performance from the cost of the recursion itself.

[8] Our implementation of a parallel RNG based on Mersenne Twister also requires a similar lookup from thread-local storage to find the worker-thread's local RNG.

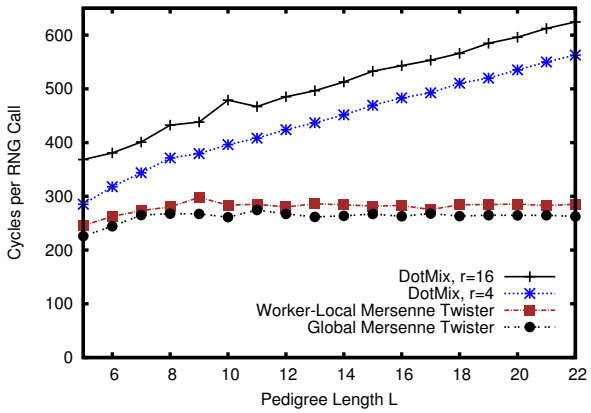

Figure 8: Overhead of various RNG's on the CBT benchmark when generating $n = 2^{20}$ random numbers. Each data point represents the minimum of 20 runs. The global Mersenne twister RNG from the GSL library [21] only works for serial code, while the worker-local Mersenne twister is a nondeterministic parallel implementation.

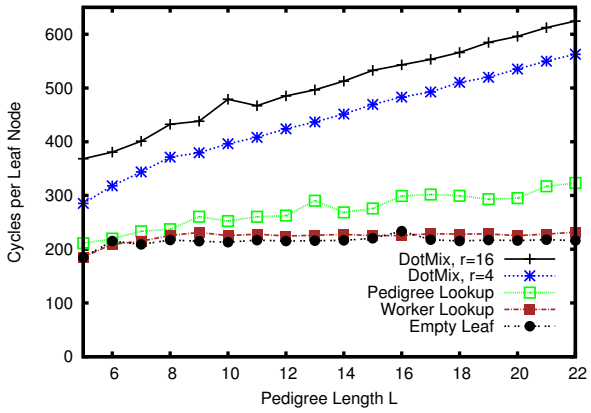

Figure 9: Breakdown of overheads of DOTMIX in the CBT benchmark, with $n = 2^{20}$. This experiment uses the same methodology as for Figure 8.

Pedigree flattening

To estimate the performance improvement of the pedigree-flattening optimization for `cilk_for` loops described in Section 2, we compared the cost performing n pedigree lookups for the CBT benchmark (Figure 9) to the cost of pedigree lookups in a `cilk_for` loop performing n pedigree lookups in parallel. Figure 10 shows that the `cilk_for` pedigree optimization substantially reduces the cost of pedigree lookups. This result is not surprising, since the pedigree lookup for recursive spawning in the CBT benchmark cost increases roughly linearly with $\lg n$, whereas the lookup cost remains nearly constant for using a `cilk_for` as $\lg n$ increases.

Application benchmarks

Figure 11 summarizes the performance results for the various RNG's on four application benchmarks:

- `pi`: A simple Monte-Carlo simulation that calculates the value of the transcendental number π using 256M samples.
- `maxIndSet`: A randomized algorithm for finding a maximum independent set in graphs with approximately 16M vertices, where nodes have an average degree of between 4 and 20.
- `sampleSort`: A randomized recursive samplesort algorithm on 64M elements, with the base case on 10,000 samples.

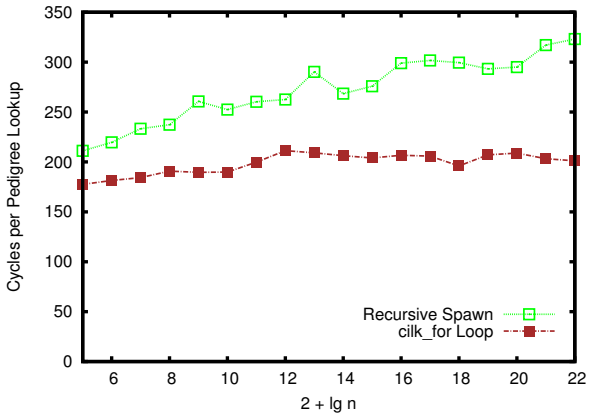

Figure 10: Comparison of pedigree lookups in a `cilk_for` loop with recursive spawns in a binary tree. The recursive spawning generates n leaves as in the CBT benchmark, with each pedigree lookup having $L = 2 + \lg n$ terms. The `cilk_for` loop uses a grain size of 1, and generates pedigrees of length 4.

Application	T_1(`DotMix`)/T_1(`mt`)	T_{12}(`DotMix`)/T_{12}(`mt`)
`fib`	2.33	2.25
`pi`	1.21	1.13
`maxIndSet`	1.14	1.08
`sampleSort`	1.00	1.00
`DiscreteHedging`	1.03	1.03

Figure 11: Overhead of DOTMIX as compared to a parallel version of the Mersenne twister (denoted by `mt` in the table) on four programs. All benchmarks use the same worker-local Mersenne twister RNG's as in Figure 8 except for `DiscreteHedging`, which uses QuantLib's existing Mersenne twister implementation.

- `DiscreteHedging`: A financial-model simulation using Monte Carlo methods.

We implemented the `pi` benchmark ourselves. The `maxIndSet` and `sampleSort` benchmarks were derived from the code described in [4]. The `DiscreteHedging` benchmark is derived from the QuantLib library for computation finance. More specifically, we modified QuantLib version 1.1 to parallelize this example as described in [25], and then supplemented QuantLib's existing RNG implementation of Mersenne Twister with DOTMIX.

To estimate the per-function-call cost of DOTMIX, we also ran the same `fib` benchmark that was used for the experiment described in Figure 7, but modified so that the RNG is called once at every node of the computation. The results for `fib` in Figure 11 indicate that DOTMIX is about a factor of 2.3 slower than using Mersenne twister, suggesting that the price of determinism for parallel random-number generation in dthreaded programs is at most 2–3 per function call.

The remaining applications pay a relatively lesser price for determinism for two reasons. First, many of these applications perform more computation per random number obtained, thereby reducing the relative cost of each call to DOTMIX. Second, many of these applications call DOTMIX within a `cilk_for` loop, and thus benefit from the pedigree-flattening optimization to reduce the cost per call of reading the pedigree.

7. Related work

The problem of generating random numbers deterministically in multithreaded programs has received significant attention. SPRNG [35] is a popular DPRNG for pthreading platforms that works by creating independent RNG's via a parameterization process. Other

approaches to parallelizing RNG's exist, such as leapfrogging and splitting. Coddington [14] surveys these alternative schemes and their respective advantages and drawbacks. It may be possible to adapt some of these pthreading RNG schemes to create similar DPRNG's for dthreaded programs.

The concept of deterministically hashing interesting locations in a program execution is not new. The `maxIndSet` and `sampleSort` benchmarks we borrowed from [4] used an *ad hoc* hashing scheme to afford repeatability, a technique we have used technique ourselves in the past and which must have been reinvented numerous times before us. More interesting is the pedigree-like scheme due to Bond and McKinley [8] where they use an LCG strategy similar to LCGMIX to assign deterministic identifiers to calling contexts for the purposes of residual testing, anomaly-based bug detection, and security intrusion detection.

Recently, Salmon *et al.* [44] independently explored the idea of "counter-based" parallel RNG's, which generate random numbers via independent transformations of counter values. Counter-based RNG's use similar ideas to pedigree-based DPRNG's. Intuitively, the compressed pedigree values generated by DOTMIX and LCGMIX can be thought of as counter values, and the mixing function corresponds to a particular kind of transformation. Salmon *et al.* focus on generating high-quality random numbers, exploring several transformations based on both existing cryptographic standards and some new techniques, and show that these transformations lead to RNG's with good statistical properties. Counter-based RNG's do not directly lead to DPRNG's for a dthreaded programs, however, because it can be difficult to generate deterministic counter values. One can, however, apply these transformations to compressed pedigree values and automatically derive additional pedigree-based DPRNG's.

8. Concluding remarks

We conclude by discussing two enhancements for pedigrees and DPRNG's. We also consider how the notion of pedigrees might be extended to work on other concurrency platforms.

The first enhancement addresses the problem of multiple calls to a DPRNG within a strand. The mechanism described in Section 2 involves calling the STRANDBREAK function, which increments the rank whenever a call to the DPRNG is made, thereby ensuring that two successive calls have different pedigrees. An alternative idea is to have the DPRNG store for each worker p an ***event counter*** e_p that the DPRNG updates manually and uses as an additional pedigree term so that multiple calls to the DPRNG per strand generate different random numbers.

The DPRNG can maintain an event counter for each worker as follows. Suppose that the DPRNG stores for each worker p the last pedigree p read. When worker p calls the DPRNG to generate another random number, causing the DPRNG to read the pedigree, the DPRNG can check whether the current pedigree matches the last pedigree p read. If it matches, then p has called the DPRNG again from the same strand, and so the DPRNG updates e_p. If it does not match, then p must be calling the DPRNG from a new strand. Because each strand is executed by exactly one worker, the DPRNG can safely reset e_p to a default value in order to generate the next random number.

This event counter scheme improves the composability of DPRNG's in a program, because calls to one DPRNG do not affect calls to another DPRNG in the same program, as they do for the scheme from Section 2. In practice, however, event counters may hurt the performance of a DPRNG. From experiments with the `fib` benchmark, we found that an event-counter scheme runs approximately 20%–40% slower per function call than the scheme from Section 2, and thus we favored the use of STRANDBREAK for our main results. Nevertheless, more efficient ways to implement an event-counter mechanism may exist, which would enhance composability.

Our second enhancement addresses the problem of "climbing the tree" to access all ranks in the pedigree for each call to the DPRNG, the cost of which is proportional to the spawn depth d. Some compression functions, including Definitions 1 and 3, can be computed ***incrementally***, and thus results can be "memoized" to avoid walking up the entire tree to compress the pedigree. In principle, one could memoize these results in a ***frame-data*** cache — a worker-local cache of intermediate results — and then, for some computations, generate random numbers in $O(1)$ time instead of $O(d)$ time. Preliminary experiments with using frame-data caches indicate, however, that in practice, the cost of memoizing the intermediate results in every stack frame outweighs the benefits from memoization, even in an example such as `fib`, where the spawn depth can be quite large. Hence, we opted not to use frame-data caches for DOTMIX. Nevertheless, it is an interesting open question whether another memoization technique, such as selective memoization specified by the programmer, might improve performance for some applications.

We now turn to the question of how to extend the pedigree ideas to "less structured" dthreading concurrency platforms. For some parallel-programming models with less structure than Cilk, it may not be important to worry about DPRNG's at all, because these models do not encapsulate the nondeterminism of the scheduler. Thus, a DPRNG would seem to offer little benefit over a nondeterministic parallel RNG. Nevertheless, some models that support more complex parallel control than the fork-join model of Cilk do admit the writing of deterministic programs, and for these models, the ideas of pedigrees can be adapted.

As an example, Intel Threading Building Blocks [42] supports software pipelining, in which each stage of the pipeline is a fork-join computation. For this control construct, one could maintain an outer-level pedigree to identify the stage in the pipeline and combine it with a pedigree for the interior fork-join computation within a stage.

Although Cilk programs produce instruction traces corresponding to fork-join graphs, the pedigree idea also seems to extend to general dags, at least in theory. One can define pedigrees on general dags as long as the children (successors) of a node are ordered. The rank of a node x indicates the birth order of x with respect to its siblings. Thus, a given pedigree (sequence of ranks) defines a unique path from the source of the task graph. The complication arises because in a general dag, multiple pedigrees (paths) may lead to the same node. Assuming there exists a deterministic procedure for choosing a particular path as the "canonical" pedigree, one can still base a DPRNG on canonical pedigrees. It remains an open question, however, as to how efficiently one can maintain canonical pedigrees in this more general case, which will depend on the particular parallel-programming model.

9. Acknowledgments

Thanks to Ron Rivest and Peter Shor of MIT for early discussions regarding strategies for implementing a DPRNG for dthreaded computations. Ron suggested using the RC6 mixing strategy. Thanks to Guy Blelloch of Carnegie Mellon for sharing his benchmark suite from which we borrowed the sample sort and maximal-independent-set benchmarks. Angelina Lee of MIT continually provided us with good feedback and generously shared her mastery of Cilk runtime systems. Loads of thanks to Kevin B. Smith and especially Barry Tannenbaum of Intel Corporation for implementing pedigrees in the Intel compiler and runtime system, respectively.

References

[1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification Version 1.0*. Sun Microsystems, Inc., Mar. 2008.

[2] R. Barik, Z. Budimlic, V. Cavè, S. Chatterjee, Y. Guo, D. Peixotto, R. Raman, J. Shirako, S. Taşirlar, Y. Yan, Y. Zhao, and V. Sarkar. The Habanero multicore software research project. In *OOPSLA*, pp. 735–736. ACM, 2009.

[3] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and secure message authentication. In *CRYPTO*, pp. 216–233. IACR, Springer-Verlag, 1999.

[4] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP*. ACM, 2012.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[6] R. D. Blumofe and D. Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical Report, University of Texas at Austin, 1999.

[7] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *HOTPAR*. USENIX, 2009.

[8] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA*, pp. 97–112. ACM, 2007.

[9] R. G. Brown. Dieharder: A random number test suite. Available from `http://www.phy.duke.edu/~rgb/General/dieharder.php`, Aug. 2011.

[10] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA*, pp. 187–194. ACM, Oct. 1981.

[11] J. L. Carter and M. N. Wegman. Universal classes of hash functions. In *STOC*, pp. 106–112. ACM, 1977.

[12] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habenero-Java: the new adventures of old X10. In *PPPJ*. ACM, 2011.

[13] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, pp. 519–538. ACM, 2005.

[14] P. D. Coddington. Random number generators for parallel computers. Technical report, Northeast Parallel Architectures Center, Syracuse University, Syracuse, New York, 1997.

[15] S. Contini, R. L. Rivest, M. J. B. Robshaw, and Y. L. Yin. The security of the RC6 block cipher. Available from `http://people.csail.mit.edu/rivest/publications.html`, 1998.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[17] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *ICALP*, pp. 235–246. Springer-Verlag, 1992.

[18] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. *Theory of Computing Systems*, 32(3):301–326, 1999.

[19] P. Frederickson, R. Hiromoto, T. L. Jordan, B. Smith, and T. Warnock. Pseudo-random trees in Monte Carlo. *Parallel Computing*, 1(2):175–180, 1984.

[20] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pp. 212–223. ACM, 1998.

[21] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, and F. Rossi. *GNU Scientific Library Reference Manual*, 1.14 edition, March 2010. Available from `http://www.gnu.org/software/gsl/`.

[22] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, second edition, 2000.

[23] R. H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *LFP*, pp. 9–17. ACM, 1984.

[24] J. M. Hart. *Windows System Programming*. Addison-Wesley, third edition, 2004.

[25] Y. He. Multicore-enabling discrete hedging in QuantLib. Available from `http://software.intel.com/en-us/articles/multicore-enabling-discrete-hedging-in-quantlib/`, Oct. 2009.

[26] W. Hines and D. Montgomery. *Probability and Statistics in Engineering and Management Science*. J. Wiley & Sons, third edition, 1990.

[27] Institute of Electrical and Electronic Engineers. Information technology — Portable Operating System Interface (POSIX) — Part 1: System application program interface (API) [C language]. IEEE Standard 1003.1, 1996 Edition.

[28] Intel Corporation. *Intel Cilk Plus Language Specification*, 2010. Document Number: 324396-001US. Available from `http://software.intel.com/sites/products/cilk-plus/cilk_plus_language_specification.pdf`.

[29] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1998.

[30] D. Lea. A Java fork/join framework. In *JAVA*, pp. 36–43. ACM, 2000.

[31] E. A. Lee. The problem with threads. *Computer*, 39:33–42, 2006.

[32] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Second Symposium on Large-Scale Digital Calculating Machinery*, volume XXVI of *Annals of the Computation Laboratory of Harvard University*, pp. 141–146, 1949.

[33] D. Leijen and J. Hall. Optimize managed code for multi-core machines. *MSDN Magazine*, 2007. Available from `http://msdn.microsoft.com/magazine/`.

[34] C. E. Leiserson. The Cilk++ concurrency platform. *J. Supercomputing*, 51(3):244–257, 2010.

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

[36] M. Matsumoto and T. Nishimura. Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM TOMACS*, 8:3–30, 1998.

[37] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, England, June 1995.

[38] National Institute of Standards and Technology, Washington. *Secure Hash Standard (SHS)*, 2008. Federal Information Standards Publication 180-3. Available from `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf`.

[39] R. H. B. Netzer and B. P. Miller. What are race conditions? *ACM LOPLAS*, 1(1):74–88, March 1992.

[40] OpenMP application program interface, version 3.0. Available from `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.

[41] M. Pătraşcu and M. Thorup. The power of simple tabulation hashing. In *STOC*, pp. 1–10. ACM, 2011.

[42] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., 2007.

[43] R. L. Rivest, M. Robshaw, R. Sidney, and Y. Yin. The RC6 block cipher. Available at `http://people.csail.mit.edu/rivest/publications.html`, 1998.

[44] J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw. Parallel random numbers: as easy as 1, 2, 3. In *SC*, pp. 16:1–16:12. ACM, 2011.

[45] M. Thorup and Y. Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In *SODA*, pp. 615–624. ACM/SIAM, 2004.

[46] M. N. Wegman and L. Carter. New hash functions and their use in authentication and set equality. *JCSS*, 22(3):265–279, 1981.