Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms

Charles E. Leiserson, **Tao B. Schardl**, and Jim Sukha

MIT Computer Science and Artificial Intelligence Laboratory

PPoPP 2012

Serial random-number generation

A ***serial (pseudo)random-number generator (RNG)*** operates as a stream.

- It starts in some initial state S_1.
- The ith call generates the random number $f(S_i)$.
- The ith call also updates the state $S_{i+1} = g(S_i)$.

State-of-the-art: Mersenne twister [MN98].

Property: From a fixed initial state, an RNG generates a deterministic sequence of pseudorandom numbers.

Why determinism?

- Determinism allows a program's execution to be repeatable.
- Repeatable execution supports efficient debugging, allowing programmers to repeatedly observe bugs, and thereby hone in on those bugs.

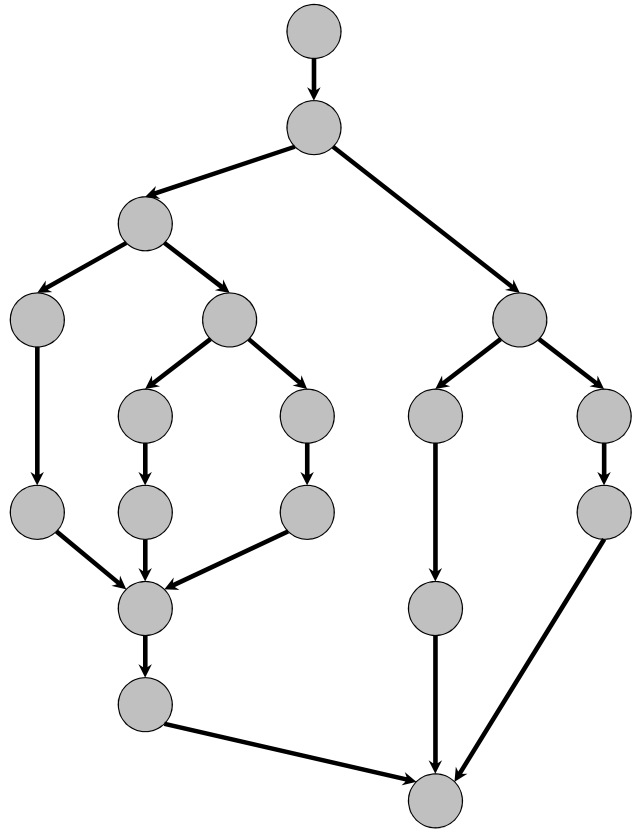
Determinism is particularly appealing in parallel programming.

- Lee [Lee06] cites the nondeterminism of multithreaded programs as a key reason that programming large-scale parallel applications remains error prone and difficult.
- Bocchino *et al.* [BAAS09] argue persuasively that multithreaded programs should be deterministic by default.

Dynamic multithreading

Dynamic multithreading (dthreading) concurrency platforms offer a ***processor-oblivious*** model of computation.

- The language exposes logical parallelism within an application.
- The runtime system schedules and executes the computation on whatever ***worker*** threads are available.
- The platform encapsulates the nondeterminism of scheduling, allowing programmers to write deterministic processor-oblivious codes.



Examples: Cilk, Fortress, Habenero, OpenMP, TBB, TPL, X10, etc.

Dthreading example: Cilk

Cilk extends C/C++ with two keywords, `cilk_spawn` and `cilk_sync`, which express *potential* parallelism in code.

```
1 int fib (int n) {
2   if (n < 2) return n;
3   else {
4     int x, y;
5     x = cilk_spawn fib(n-1);
6     y = fib(n-2);
7     cilk_sync;
8     return x+y;
9 } }
```

- The named ***child*** function of a `cilk_spawn` may execute in parallel with the ***parent*** caller.
- Control cannot pass a `cilk_sync` until all spawned children have returned.

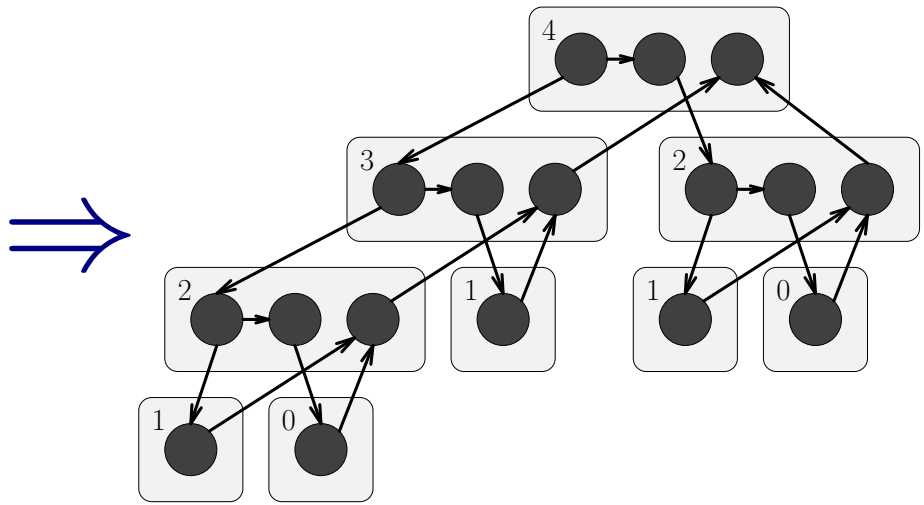
Cilk keywords *grant permission* for parallel execution. They do not *command* parallel execution.

Dthreading example: Cilk execution model

A Cilk program's execution can be modeled with a ***computation dag***.

Example: `fib(4)`

```
1 int fib (int n) {
2   if (n < 2) return n;
3   else {
4     int x, y;
5     x = cilk_spawn fib(n-1);
6     y = fib(n-2);
7     cilk_sync;
8     return x+y;
9 } }
```



- Nodes represents ***strands*** — serial sequences of instructions containing no parallel control.
- Edges depict parallel control dependencies between strands.

Dthreading example: Cilk execution model

The computation dag for a Cilk program's execution unfolds *dynamically* and is embedded in the ***invocation tree***.

Example: `fib(4)`
Program text corresponding to various strands is color-coded.

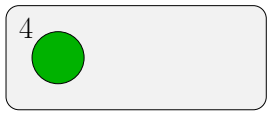
```
1 int fib (int n) {
2   if (n < 2) return n;
3   else {
4     int x, y;
5     x = cilk_spawn fib(n-1);
6     y = fib(n-2);
7     cilk_sync;
8     return x+y;
9 } }
```

Dthreading example: Cilk execution model

The computation dag for a Cilk program's execution unfolds *dynamically* and is embedded in the ***invocation tree***.

Example: `fib(4)`
Program text corresponding to various strands is color-coded.

```
1 int fib (int n) {
2   if (n < 2) return n;
3   else {
4     int x, y;
5     x = cilk_spawn fib(n-1);
6     y = fib(n-2);
7     cilk_sync;
8     return x+y;
9 } }
```

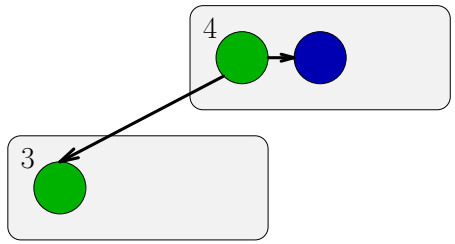


Dthreading example: Cilk execution model

The computation dag for a Cilk program's execution unfolds *dynamically* and is embedded in the ***invocation tree***.

Example: `fib(4)`
Program text corresponding to various strands is color-coded.

```
1 int fib (int n) {
2   if (n < 2) return n;
3   else {
4     int x, y;
5     x = cilk_spawn fib(n-1);
6     y = fib(n-2);
7     cilk_sync;
8     return x+y;
9 } }
```

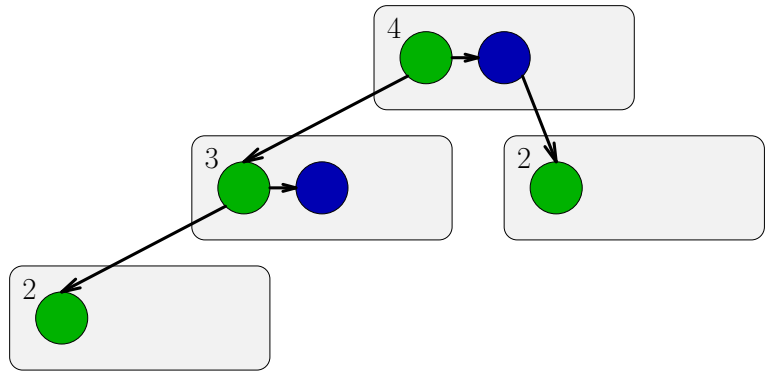


Dthreading example: Cilk execution model

The computation dag for a Cilk program's execution unfolds *dynamically* and is embedded in the ***invocation tree***.

Example: `fib(4)`
Program text corresponding to various strands is color-coded.

```
1 int fib (int n) {
2   if (n < 2) return n;
3   else {
4     int x, y;
5     x = cilk_spawn fib(n-1);
6     y = fib(n-2);
7     cilk_sync;
8     return x+y;
9 } }
```

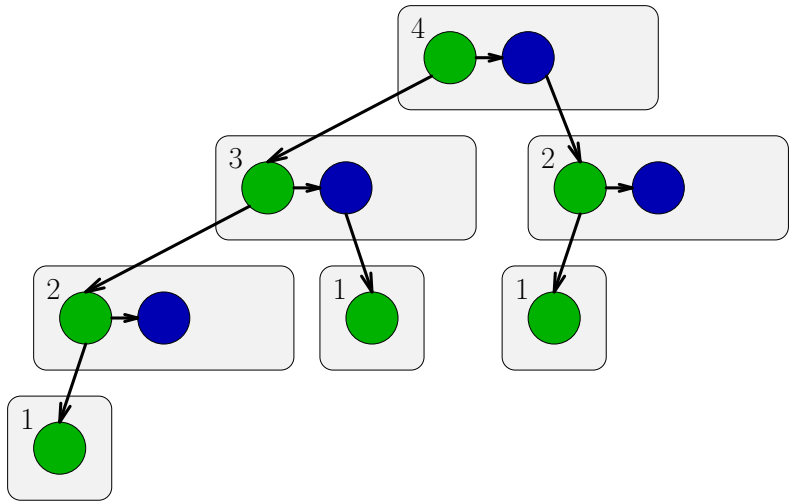


Dthreading example: Cilk execution model

The computation dag for a Cilk program's execution unfolds *dynamically* and is embedded in the ***invocation tree***.

Example: `fib(4)`
Program text corresponding to various strands is color-coded.

```
1 int fib (int n) {
2   if (n < 2) return n;
3   else {
4     int x, y;
5     x = cilk_spawn fib(n-1);
6     y = fib(n-2);
7     cilk_sync;
8     return x+y;
9 } }
```

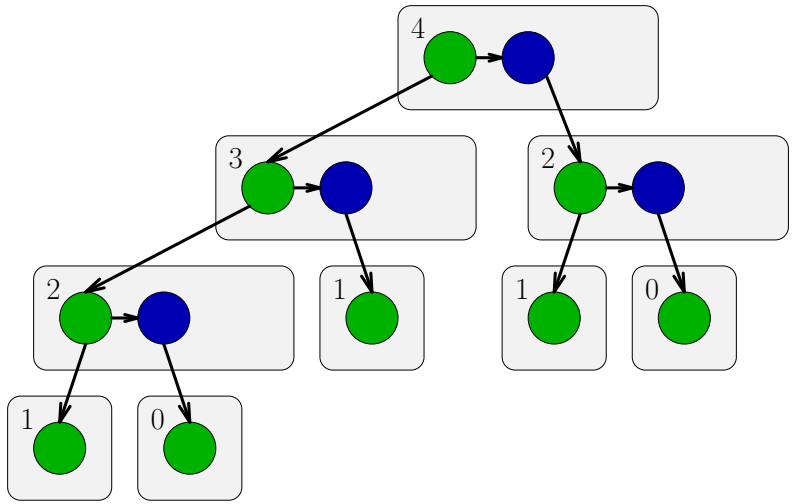


Dthreading example: Cilk execution model

The computation dag for a Cilk program's execution unfolds *dynamically* and is embedded in the ***invocation tree***.

Example: `fib(4)`
Program text corresponding to various strands is color-coded.

```
1 int fib (int n) {
2   if (n < 2) return n;
3   else {
4     int x, y;
5     x = cilk_spawn fib(n-1);
6     y = fib(n-2);
7     cilk_sync;
8     return x+y;
9 } }
```

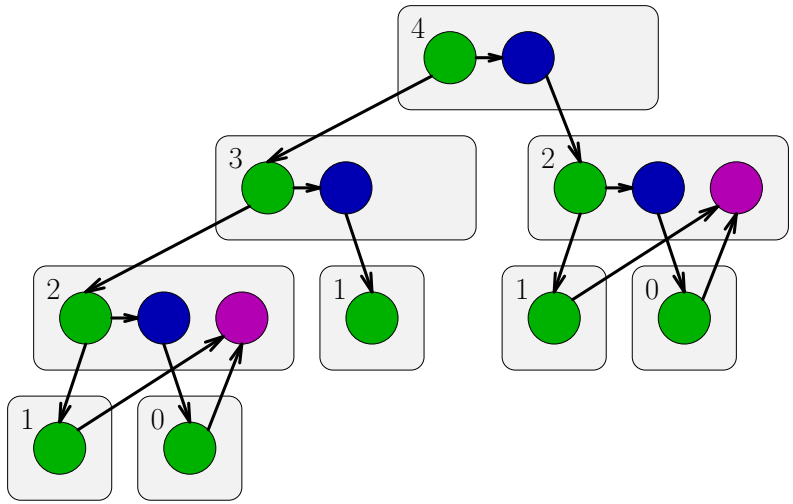


Dthreading example: Cilk execution model

The computation dag for a Cilk program's execution unfolds *dynamically* and is embedded in the ***invocation tree***.

Example: `fib(4)`
Program text corresponding to various strands is color-coded.

```
1 int fib (int n) {
2   if (n < 2) return n;
3   else {
4     int x, y;
5     x = cilk_spawn fib(n-1);
6     y = fib(n-2);
7     cilk_sync;
8     return x+y;
9 } }
```

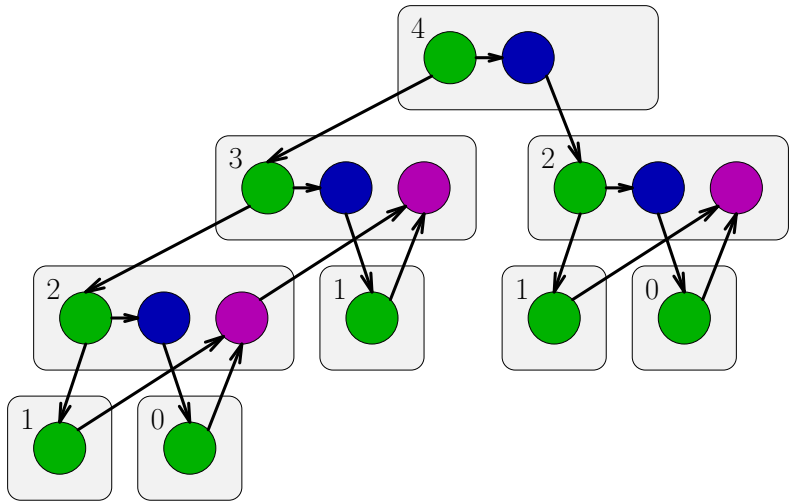


Dthreading example: Cilk execution model

The computation dag for a Cilk program's execution unfolds *dynamically* and is embedded in the ***invocation tree***.

Example: `fib(4)`
Program text corresponding to various strands is color-coded.

```
1 int fib (int n) {
2   if (n < 2) return n;
3   else {
4     int x, y;
5     x = cilk_spawn fib(n-1);
6     y = fib(n-2);
7     cilk_sync;
8     return x+y;
9 } }
```

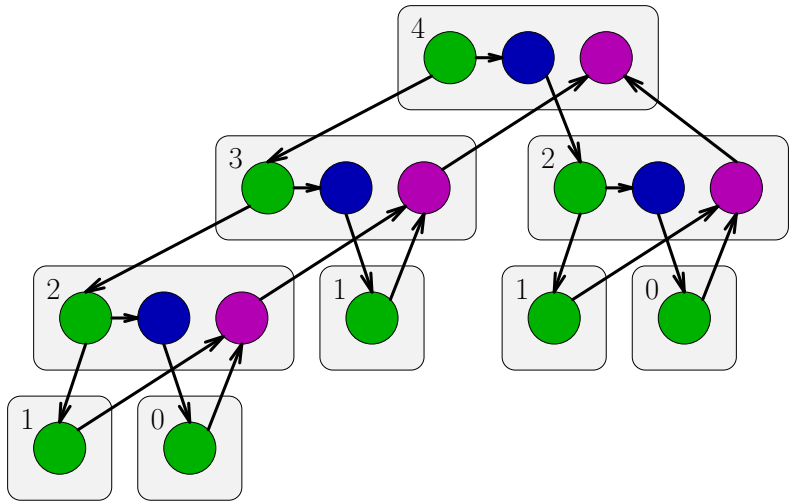


Dthreading example: Cilk execution model

The computation dag for a Cilk program's execution unfolds *dynamically* and is embedded in the ***invocation tree***.

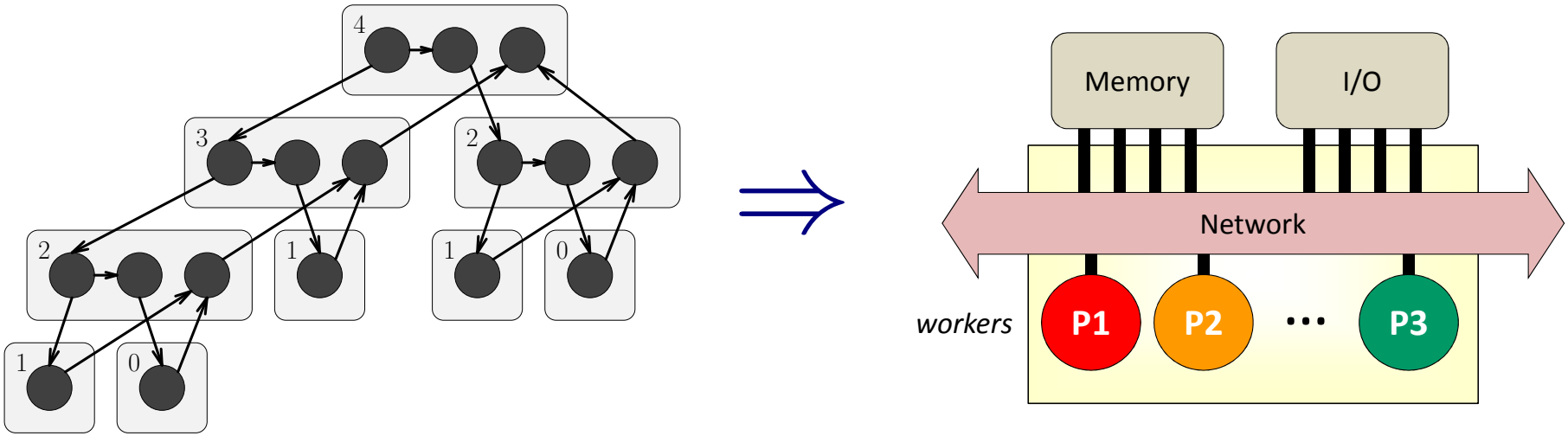
Example: `fib(4)`
Program text corresponding to various strands is color-coded.

```
1 int fib (int n) {
2   if (n < 2) return n;
3   else {
4     int x, y;
5     x = cilk_spawn fib(n-1);
6     y = fib(n-2);
7     cilk_sync;
8     return x+y;
9 } }
```



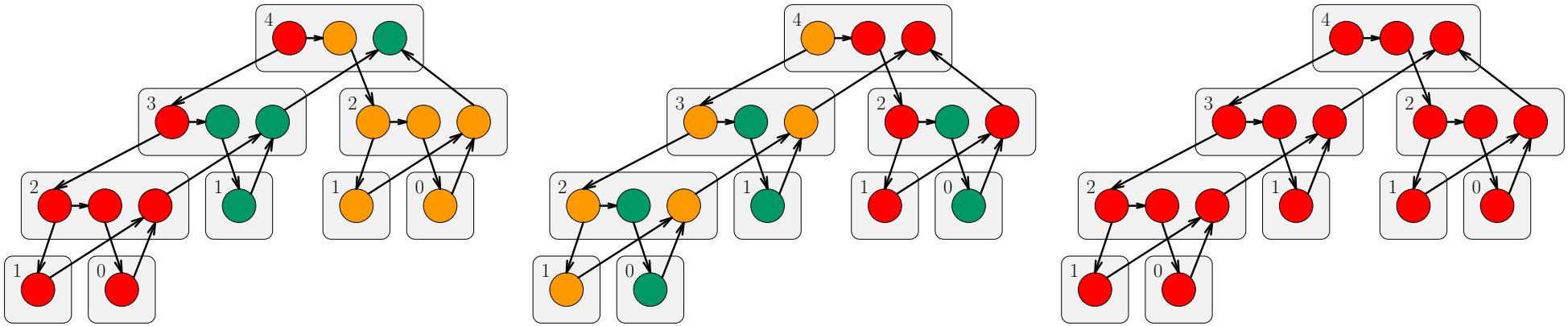
Dthreading example: Cilk scheduler

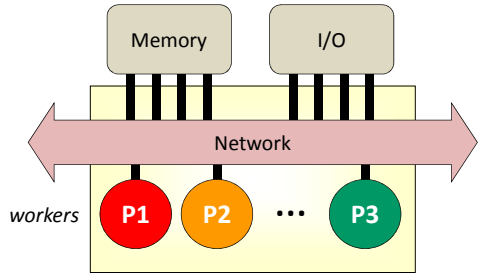
Cilk's runtime system incorporates a ***scheduler***, which maps the executing program onto ***worker*** threads dynamically at runtime.



Dthreading example: Cilk scheduler

Cilk's ***randomized work-stealing*** scheduler achieves provably good performance.



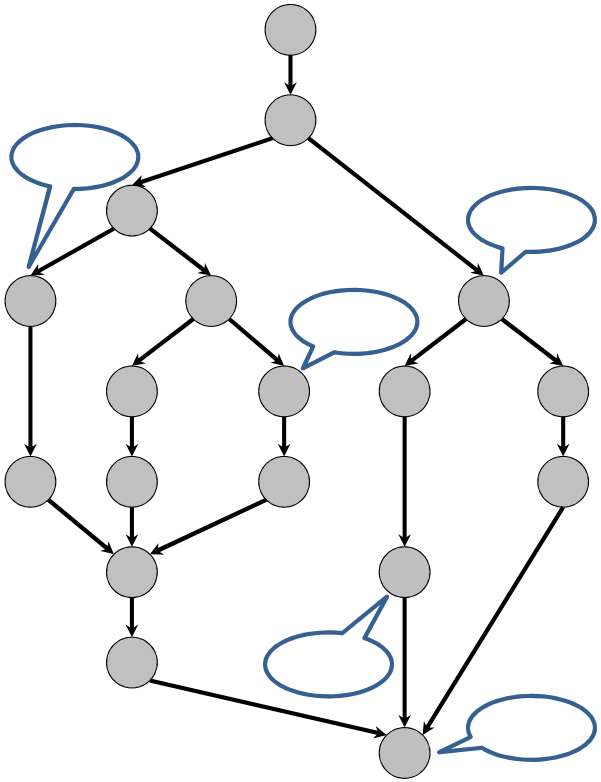


Cilk *encapsulates* the nondeterminism of scheduling, allowing programmers to write deterministic, processor-oblivious codes.

Outline

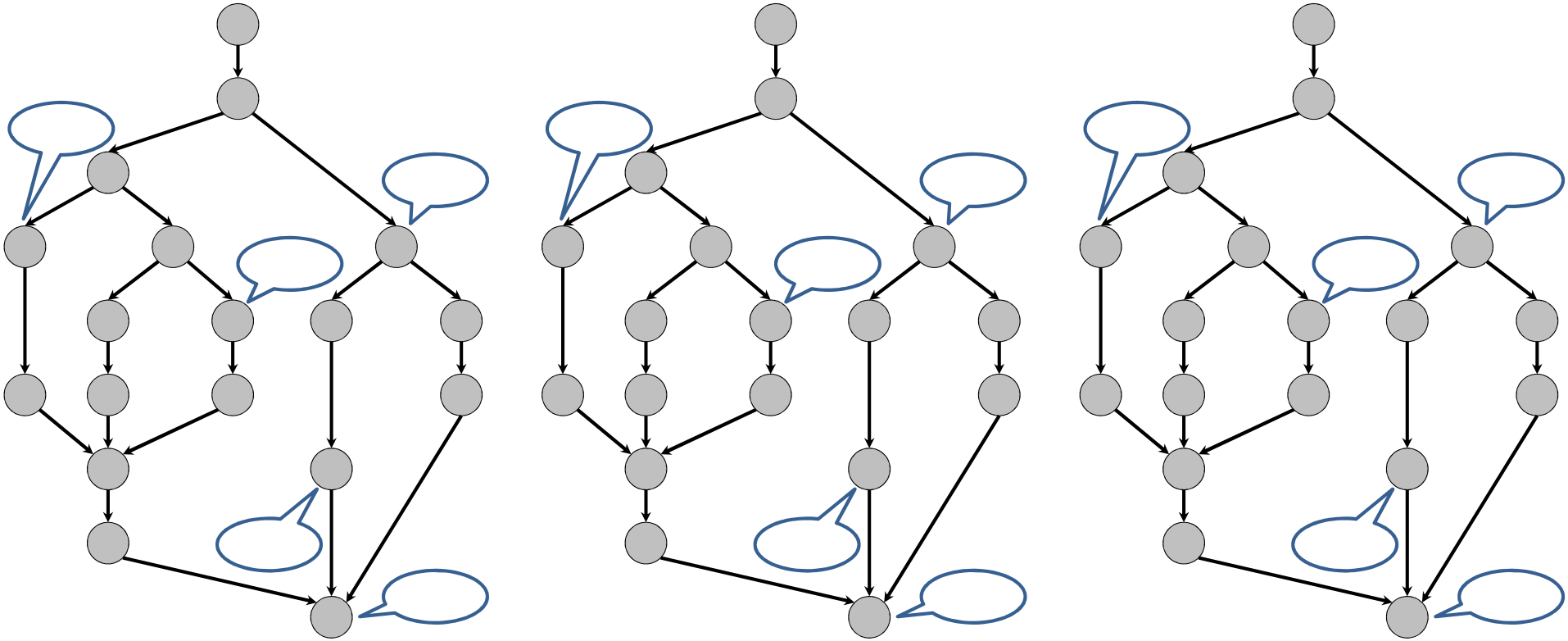
DPRNG's

For dthreaded programs, we want a ***deterministic parallel RNG (DPRNG)*** — for a fixed initial state, each call to a DPRNG generates the same pseudorandom number on every program execution.



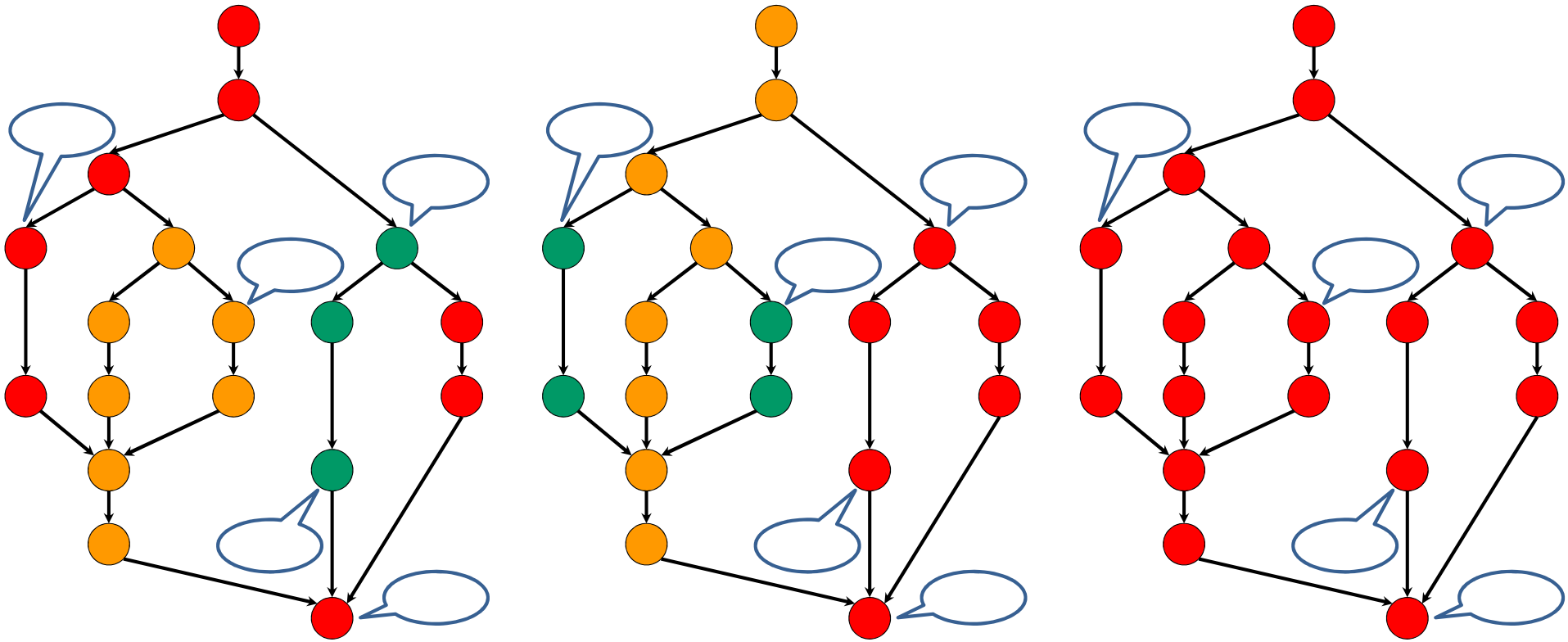
DPRNG's

For dthreaded programs, we want a ***deterministic parallel RNG (DPRNG)*** — for a fixed initial state, each call to a DPRNG generates the same pseudorandom number on every program execution.



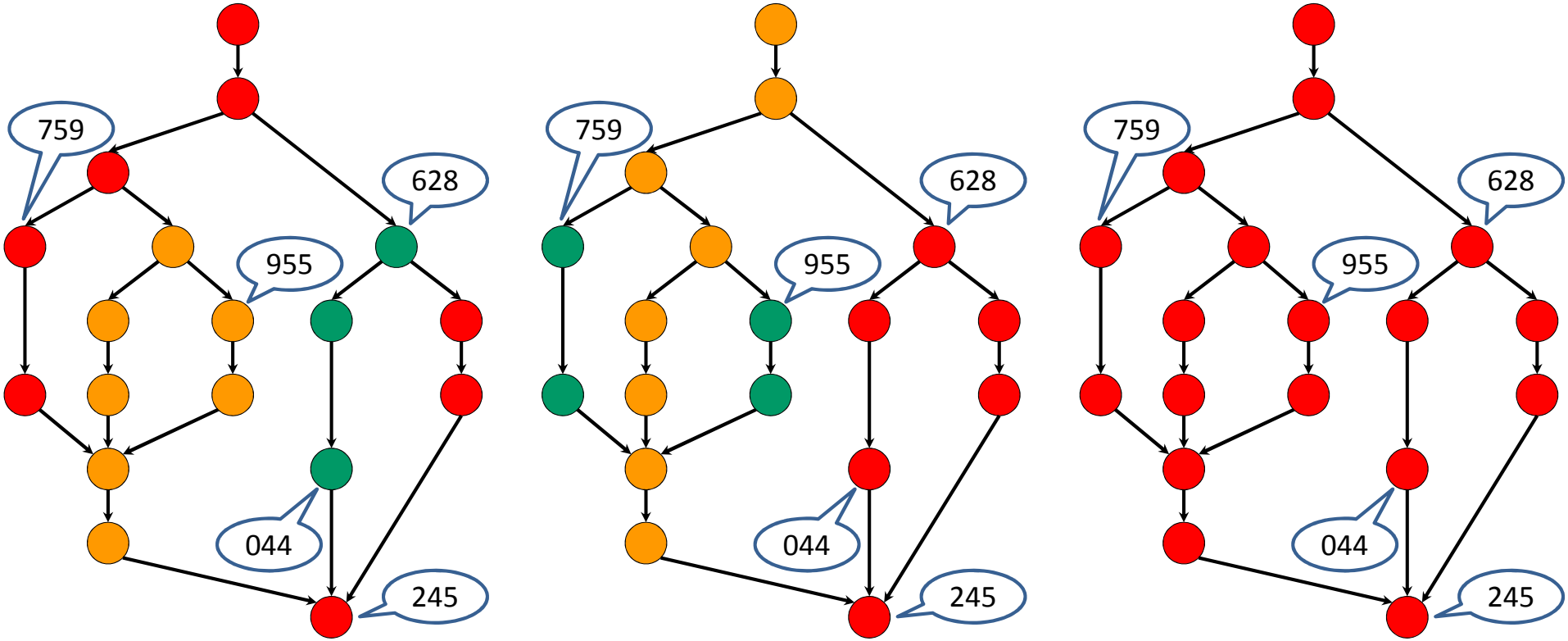
DPRNG's

For dthreaded programs, we want a ***deterministic parallel RNG (DPRNG)*** — for a fixed initial state, each call to a DPRNG generates the same pseudorandom number on every program execution.



DPRNG's

For dthreaded programs, we want a ***deterministic parallel RNG (DPRNG)*** — for a fixed initial state, each call to a DPRNG generates the same pseudorandom number on every program execution.

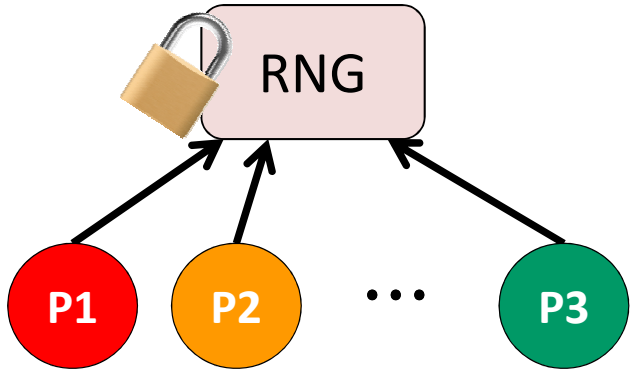


Parallelizing serial RNG's

There are some conventional techniques for parallelizing serial RNG's.

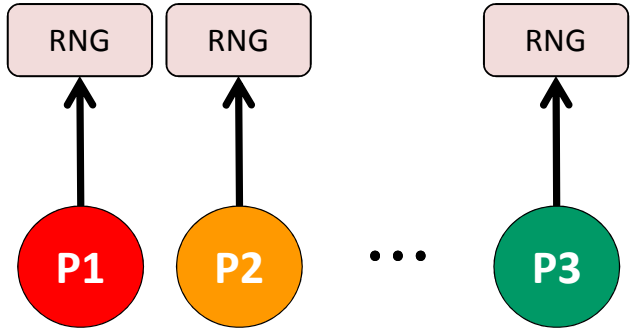
Idea: Make a serial RNG thread-safe by locking the state.

- Results in contention.



Idea: Give each worker thread its own RNG.

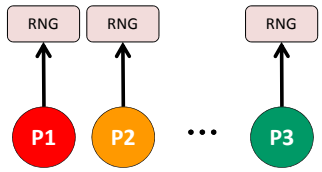
- No contention.

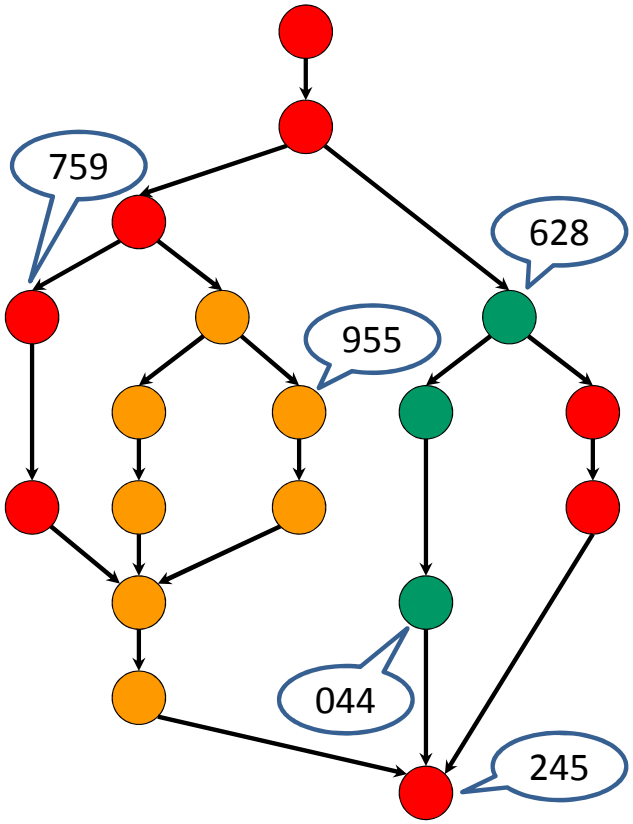


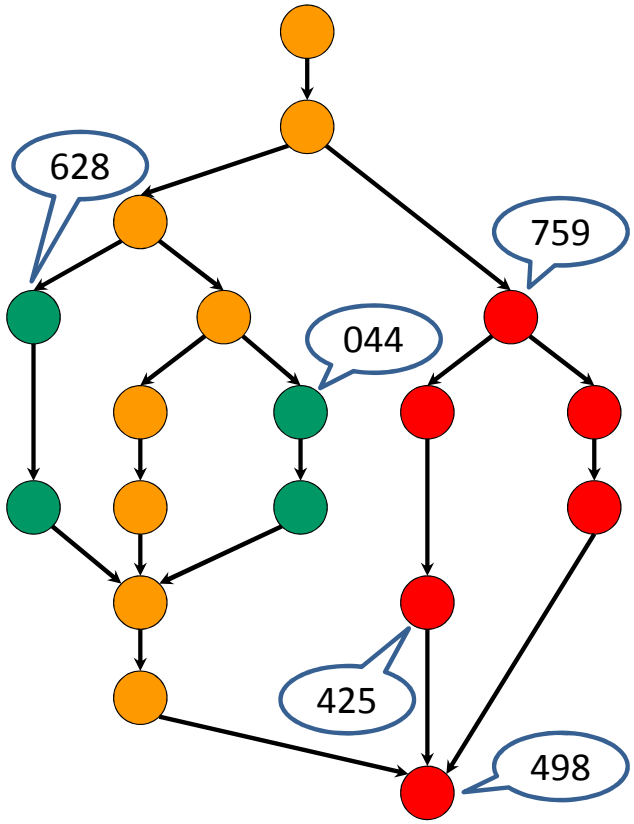
Neither of these techniques produce a deterministic parallel RNG.

Example: thread-local parallel RNG

Problem: Nondeterministic scheduling results in nondeterministic behavior.







Note: This does produce an effective *nondeterministic* parallel RNG.

SPRNG [MS00]

SPRNG is a DPRNG for pthreaded programs that creates an independent RNG for each pthread via a deterministic parameterization process.

- ***Pthreaded programs*** are programs where parallel work is explicitly mapped onto parallel threads, or ***pthreads***.

Idea: Use SPRNG to create an independent RNG for each potentially parallel subcomputation in a dthreaded program.

SPRNG [MS00]

Problem: SPRNG is not designed to handle a computation with a large number of dynamic threads.

```
 1 int rfib (int n, PRNG g) {
 2   if (n < 2) {
 3     g.get();
 4     return n;
 5   } else {
 6     int x, y;
 7     PRNG h = g.spawn();
 8     x = cilk_spawn rfib(n-1, h);
 9     y = rfib(n-2, g);
10     cilk_sync;
11     return x+y;
12 } }
```

Experiment: We ran `rfib()` using SPRNG and using Mersenne twister parallelized using thread-local RNG's.

- For Mersenne twister, `g.spawn()` returns `g`.
- For SPRNG, `g.spawn()` deterministically forks a new RNG stream from `g`.

Results:

- SPRNG runs $50,000\times$ slower than Mersenne twister on `rfib(21)`.
- SPRNG's default RNG only guarantees the independence of 2^{19} streams, and computing `rfib(n)` for $n > 21$ forfeits this guarantee.

Our contributions

1. A runtime mechanism, called "pedigrees", for tracking the "lineage" of each strand in a dthreaded program.

Application	*Default (s)*	*Pedigree (s)*	*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

Our contributions

1. A runtime mechanism, called “pedigrees”, for tracking the “lineage” of each strand in a dthreaded program.
2. A general strategy for efficiently generating high statistical quality pseudorandom numbers deterministically in parallel, based on hashing a strand’s pedigree.

Our contributions

1. A runtime mechanism, called "pedigrees", for tracking the "lineage" of each strand in a dthreaded program.
2. A general strategy for efficiently generating high statistical quality pseudorandom numbers deterministically in parallel, based on hashing a strand's pedigree.
3. A high-quality DPRNG library for Intel Cilk Plus, called DotMix, which is 2–3 times as costly per call as a worker-local Mersenne twister solution (`mt`) and passes most of the Dieharder [Bro11] RNG statistical tests.

Application	T_1(DotMix)/T_1(mt)	T_{12}(DotMix)/T_{12}(mt)
`fib`	2.65	2.42
`pi`	0.74	0.72
`maxIndSet`	0.99	0.98
`sampleSort`	0.99	0.99
`DiscreteHedging`	1.03	1.03

Outline

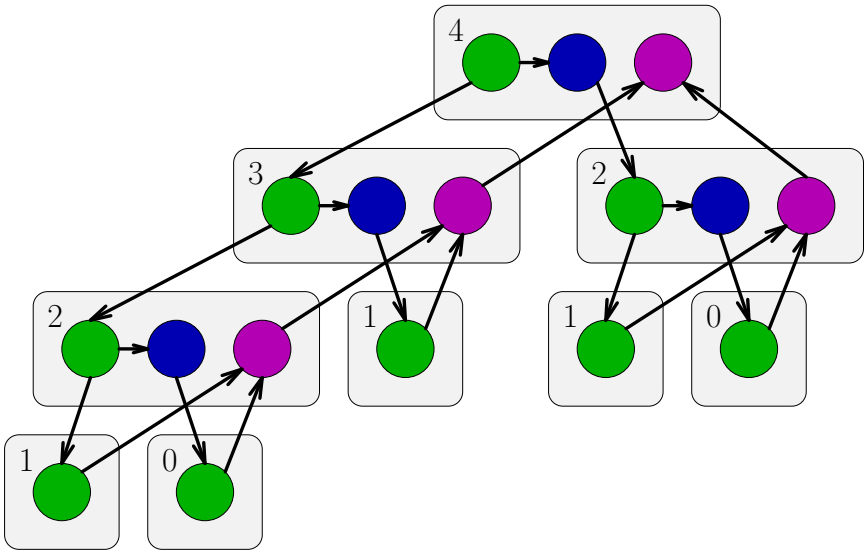
Outline

Pedigrees

A ***pedigree*** is a unique, processor-oblivious identifier for a strand.

Simple Idea: We can uniquely identify strands by their location in the invocation tree.

Example: `fib(4)`



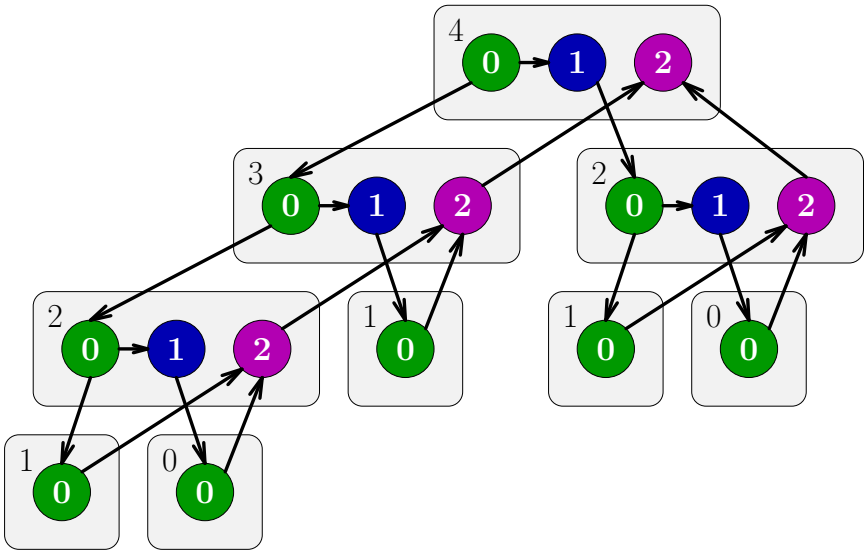
- The invocation tree of a deterministic, processor-oblivious program is deterministic and processor-oblivious.
- The pedigree $J(s)$ of a strand s can be viewed as the path in the invocation tree from the root to s.

Pedigrees

A ***pedigree*** is a unique, processor-oblivious identifier for a strand.

Simple Idea: We can uniquely identify strands by their location in the invocation tree.

Example: `fib(4)`



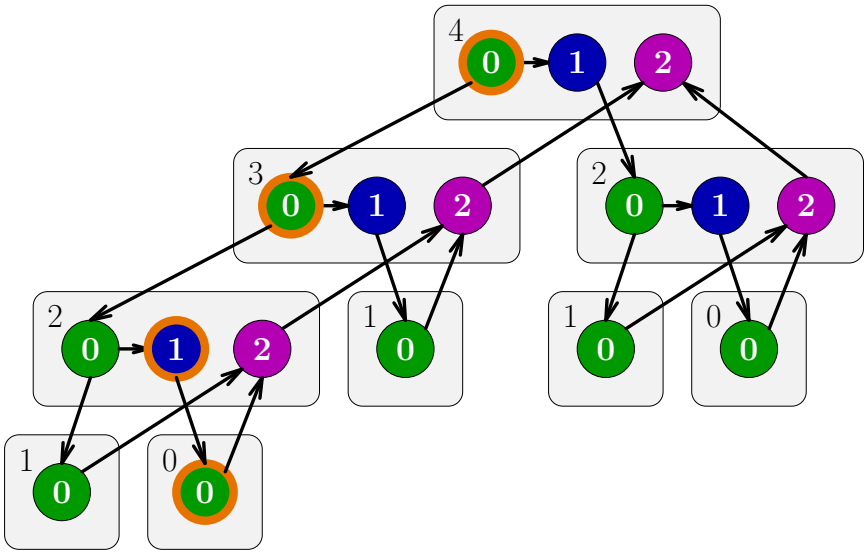
- The invocation tree of a deterministic, processor-oblivious program is deterministic and processor-oblivious.
- The pedigree $J(s)$ of a strand s can be viewed as the path in the invocation tree from the root to s.

Pedigrees

A ***pedigree*** is a unique, processor-oblivious identifier for a strand.

Simple Idea: We can uniquely identify strands by their location in the invocation tree.

Example: `fib(4)`



$J = \langle 0, 0, 1, 0 \rangle$

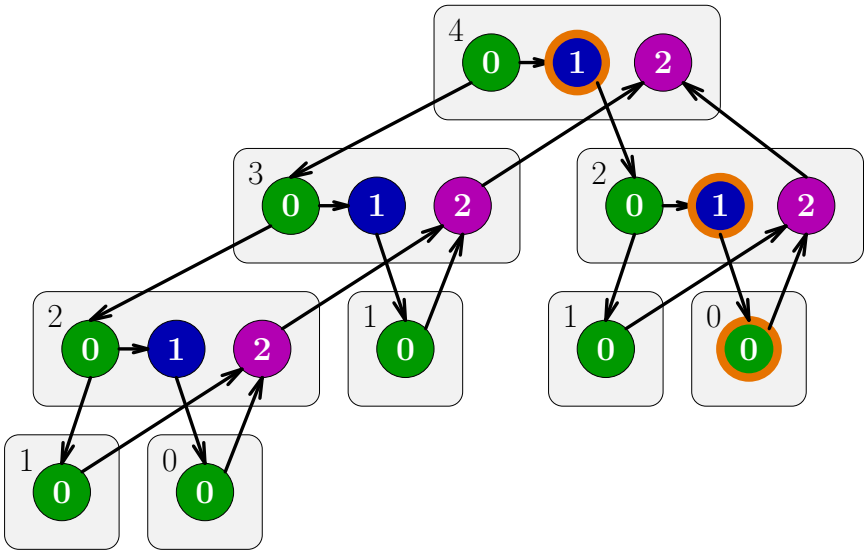
- The invocation tree of a deterministic, processor-oblivious program is deterministic and processor-oblivious.
- The pedigree $J(s)$ of a strand s can be viewed as the path in the invocation tree from the root to s.

Pedigrees

A ***pedigree*** is a unique, processor-oblivious identifier for a strand.

Simple Idea: We can uniquely identify strands by their location in the invocation tree.

Example: `fib(4)`



$J = \langle 1, 1, 0 \rangle$

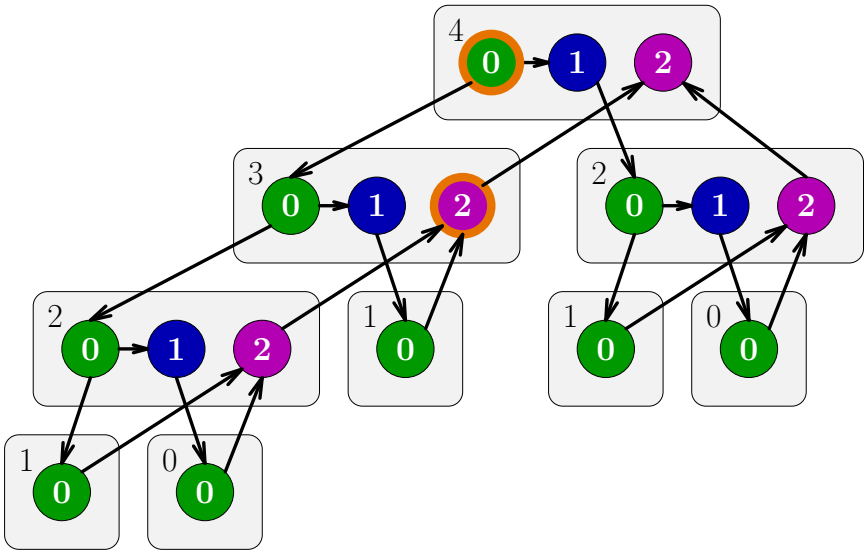
- The invocation tree of a deterministic, processor-oblivious program is deterministic and processor-oblivious.
- The pedigree $J(s)$ of a strand s can be viewed as the path in the invocation tree from the root to s.

Pedigrees

A ***pedigree*** is a unique, processor-oblivious identifier for a strand.

Simple Idea: We can uniquely identify strands by their location in the invocation tree.

Example: `fib(4)`



$J = \langle 0, 2 \rangle$

- The invocation tree of a deterministic, processor-oblivious program is deterministic and processor-oblivious.
- The pedigree $J(s)$ of a strand s can be viewed as the path in the invocation tree from the root to s.

Pedigree-based DPRNG's

Simple Idea: We can generate a deterministic pseudorandom number in a strand s by hashing the pedigree of s.

Complicating Issues: Tracking pedigrees involves changing the Cilk compiler and runtime.

- Since even code that does not need DPRNG's must pay for the overhead of pedigrees, the pedigree mechanism should be as lightweight as possible.
- Cilk applications may contain legacy or third-party C/C++ functions, some of which cannot be recompiled.

Spawn pedigrees

*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. $p \rightarrow current\text{-}frame = \widehat{F}$

*On stalling at a **sync** in G:*

1. $G \rightarrow rank = p \rightarrow rank$

*On resuming the continuation of a **spawn** or **sync** in G:*

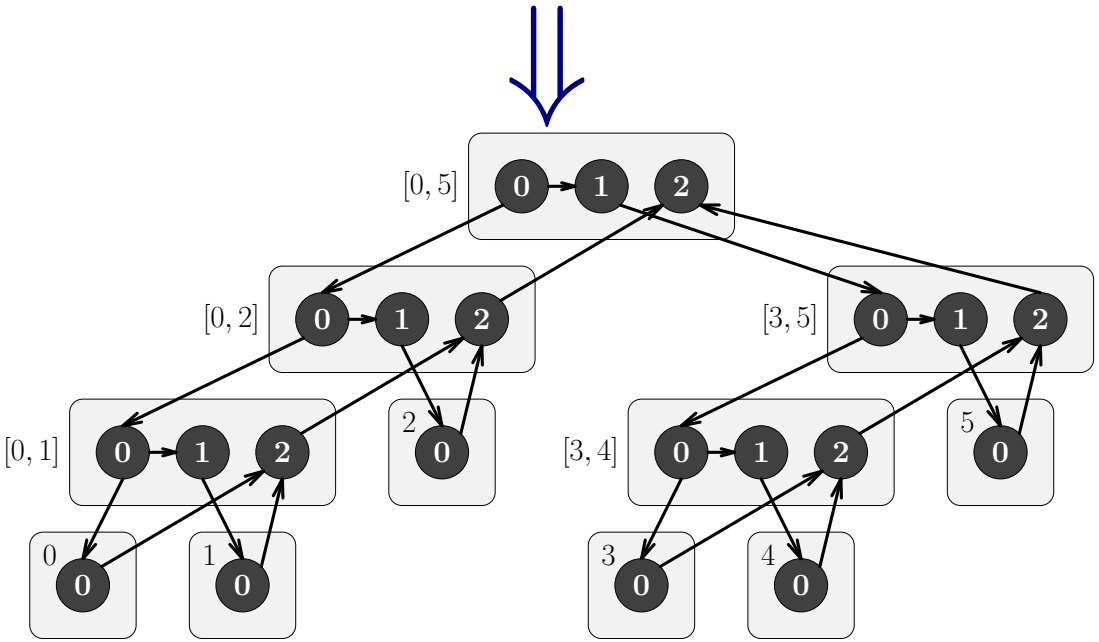
1. $p \rightarrow rank = G \rightarrow rank + 1$
2. $p \rightarrow current\text{-}frame = G \rightarrow sp\text{-}rep$

- Functions that are not spawned are viewed as being "inlined."
- Intuitively, strands are identified by the number of preceding `cilk_spawn` and `cilk_sync` statements in their parent spawned functions.

Loop pedigrees

Cilk provides a `cilk_for` keyword for writing parallel loops, which is implemented using `cilk_spawn` and `cilk_sync`.

```
int n=6;
cilk_for(int i=0; i<n; ++i)
{ ... }
```



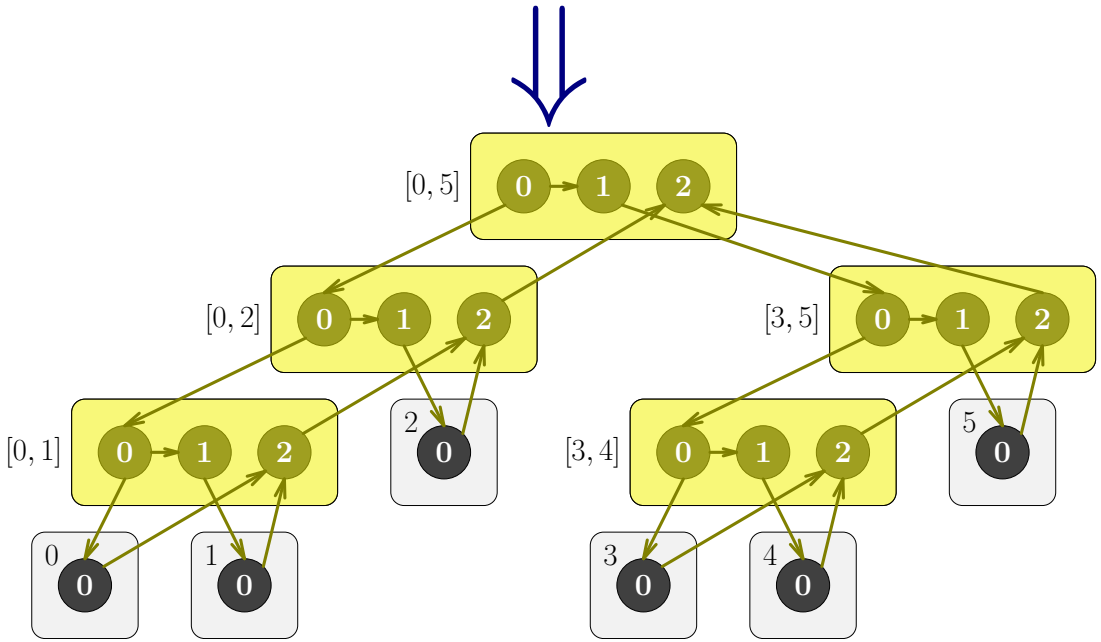
We optimize the pedigrees for `cilk_for` loops.

- The pedigree of a `cilk_for` loop iteration is set to the loop iteration index.
- The cost of reading the pedigree for a loop iteration is reduced by $\Theta(\lg n)$.

Loop pedigrees

Cilk provides a `cilk_for` keyword for writing parallel loops, which is implemented using `cilk_spawn` and `cilk_sync`.

```
int n=6;
cilk_for(int i=0; i<n; ++i)
{ ... }
```



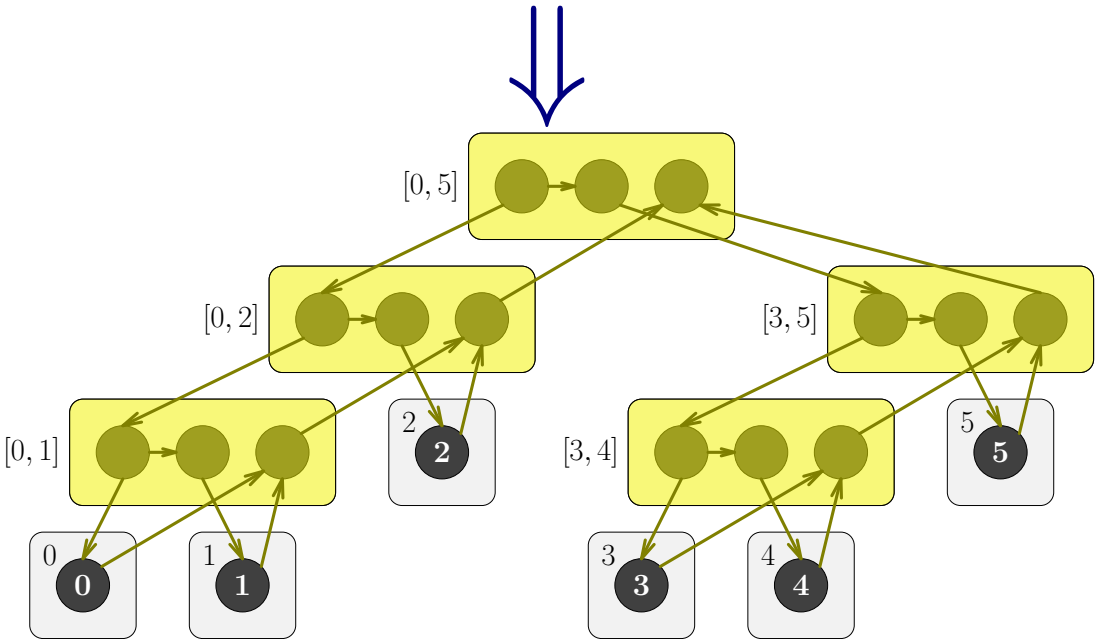
We optimize the pedigrees for `cilk_for` loops.

- The pedigree of a `cilk_for` loop iteration is set to the loop iteration index.
- The cost of reading the pedigree for a loop iteration is reduced by $\Theta(\lg n)$.

Loop pedigrees

Cilk provides a `cilk_for` keyword for writing parallel loops, which is implemented using `cilk_spawn` and `cilk_sync`.

```
int n=6;
cilk_for(int i=0; i<n; ++i)
{ ... }
```



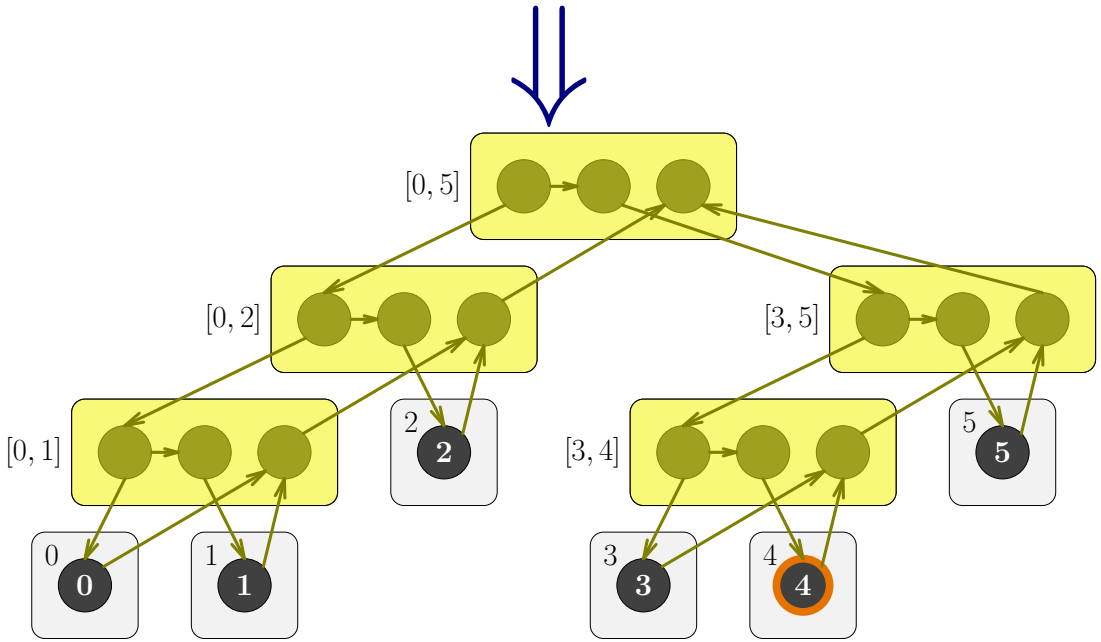
We optimize the pedigrees for `cilk_for` loops.

- The pedigree of a `cilk_for` loop iteration is set to the loop iteration index.
- The cost of reading the pedigree for a loop iteration is reduced by $\Theta(\lg n)$.

Loop pedigrees

Cilk provides a `cilk_for` keyword for writing parallel loops, which is implemented using `cilk_spawn` and `cilk_sync`.

```
int n=6;
cilk_for(int i=0; i<n; ++i)
{ ... }
```



$J = \langle 4 \rangle$

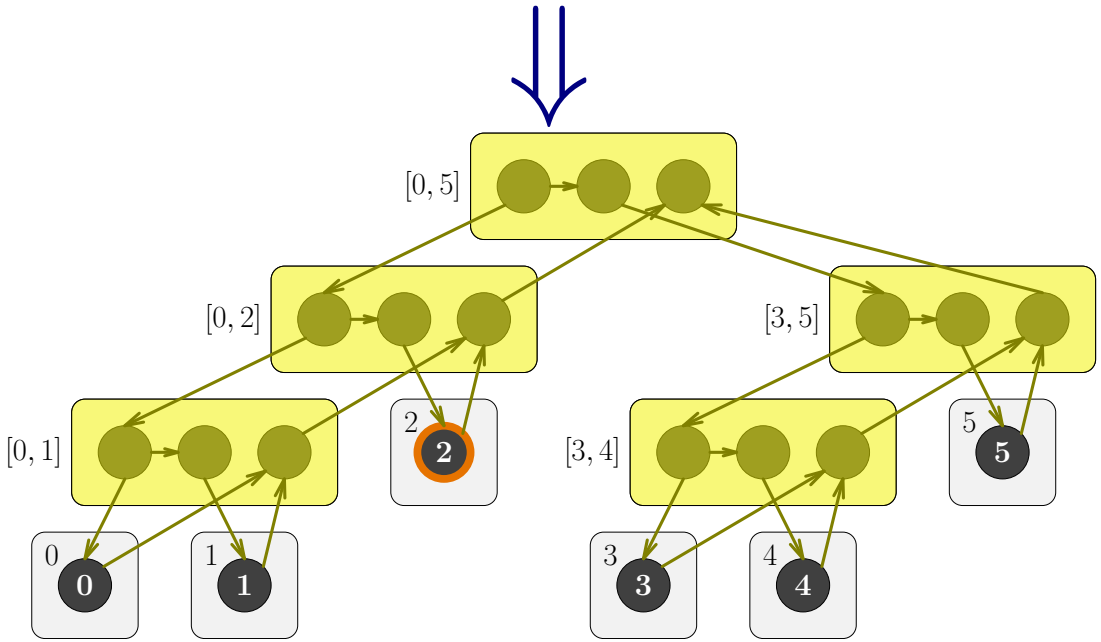
We optimize the pedigrees for `cilk_for` loops.

- The pedigree of a `cilk_for` loop iteration is set to the loop iteration index.
- The cost of reading the pedigree for a loop iteration is reduced by $\Theta(\lg n)$.

Loop pedigrees

Cilk provides a `cilk_for` keyword for writing parallel loops, which is implemented using `cilk_spawn` and `cilk_sync`.

```
int n=6;
cilk_for(int i=0; i<n; ++i)
{ ... }
```



$J = \langle 2 \rangle$

We optimize the pedigrees for `cilk_for` loops.

- The pedigree of a `cilk_for` loop iteration is set to the loop iteration index.
- The cost of reading the pedigree for a loop iteration is reduced by $\Theta(\lg n)$.

Outline

The DotMix DPRNG

DotMix hashes a pedigree in two stages.

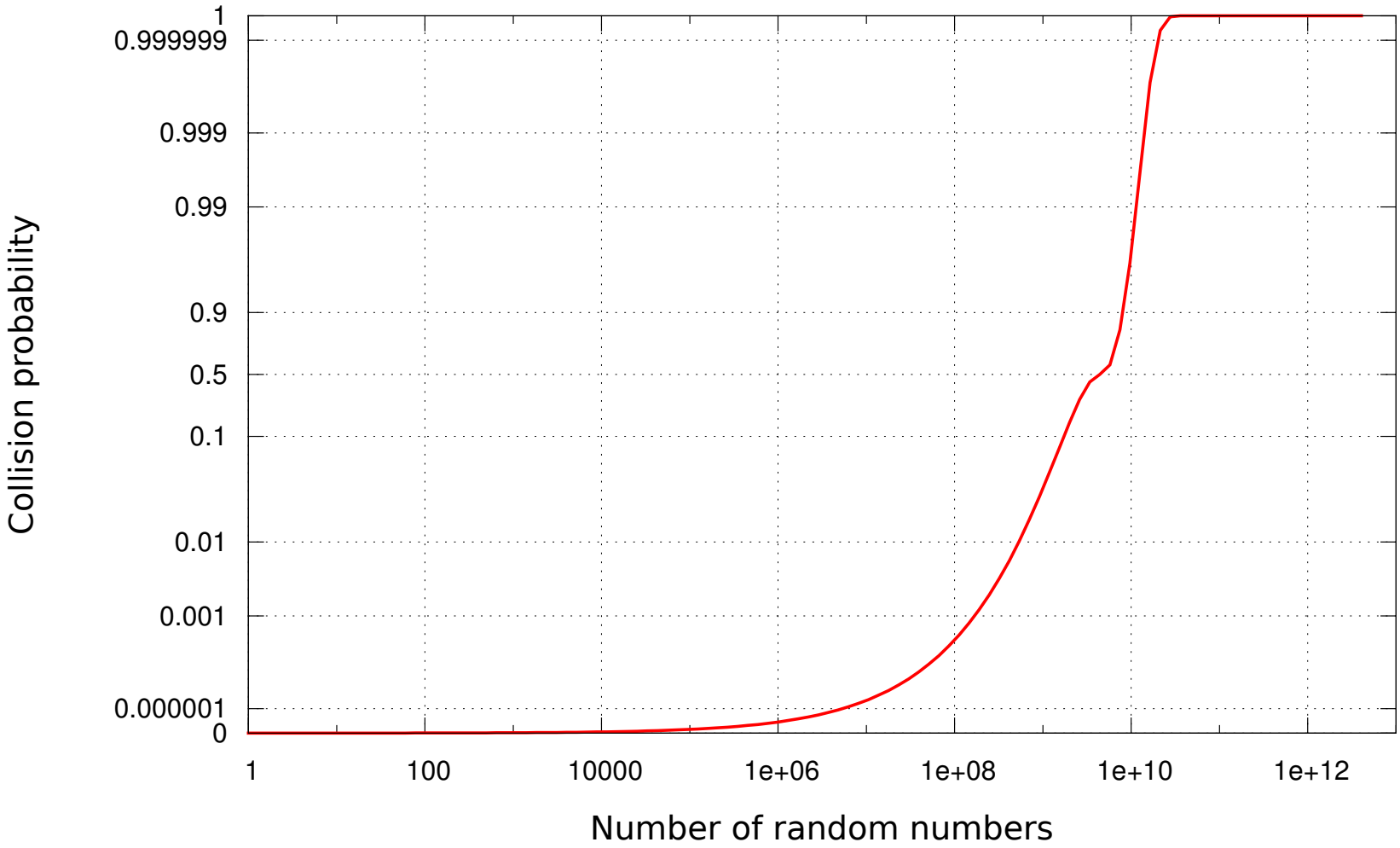
1. ***Compression***: Convert the pedigree into a single word while preserving uniqueness.
2. ***Mixing***: Remove correlation between the compressed pedigrees.

DOTMIX compression

Dot-product compression: Compute the dot product of the pedigree with a vector of random odd 64-bit integers.

Theorem: For any randomly chosen vector Γ of odd integers and any two distinct pedigrees J and J', the probability that $\Gamma \cdot J = \Gamma \cdot J'$ is at most $1/2^{63}$.

Efficacy of DOTMIX



DotMix mixing

DotMix(r) "randomly" permutes the result of the compression function using r iterations of the following "mixing" routine.

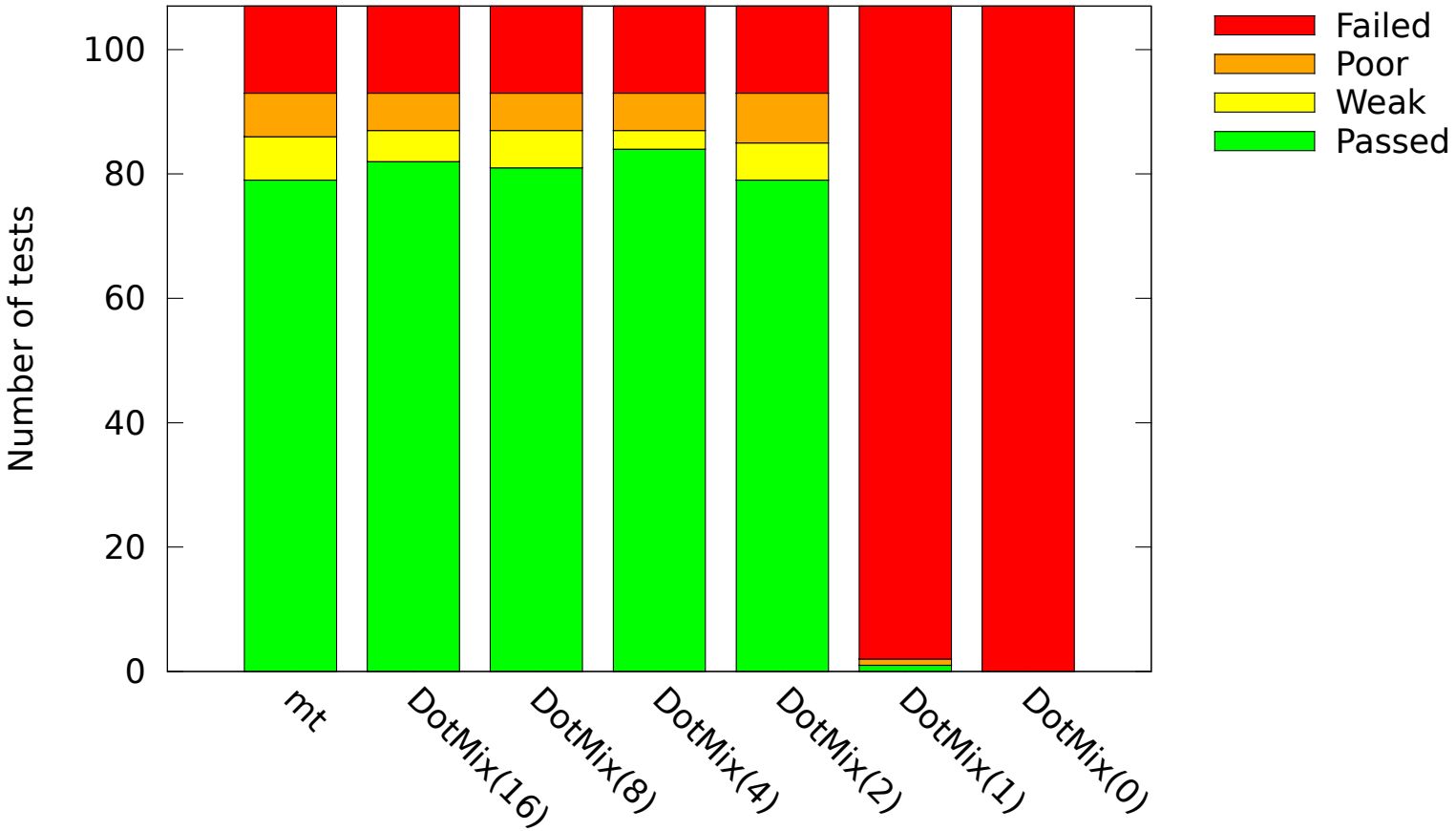
RC6 mixing: Let X_i designate the result of the ith round of mixing, where X_0 is the result of the compression function.

```
1  for (int i = 0; i < r; ++i) {
2        Y = X_i · (2X_i + 1) mod 2^64;
3        X_{i+1} = swap left and right halves of Y;
4  }
```

One can show that this function is bijective [CRRY98], so mixing does not generate further collisions.

Thanks to Ron Rivest for suggesting this mixing function.

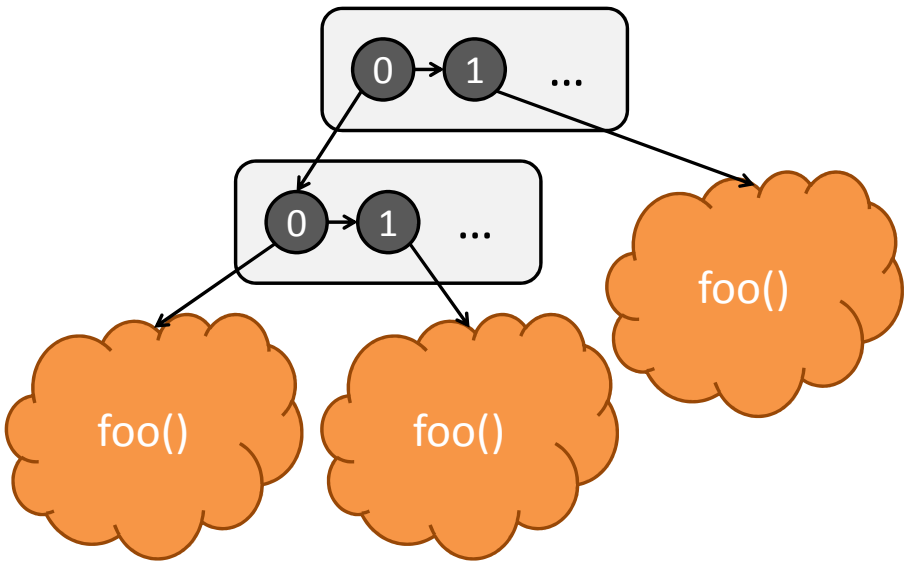
Dieharder statistical tests



Outline

Scoped pedigrees

Problem: How do we run a parallel subcomputation with the same random numbers multiple times?

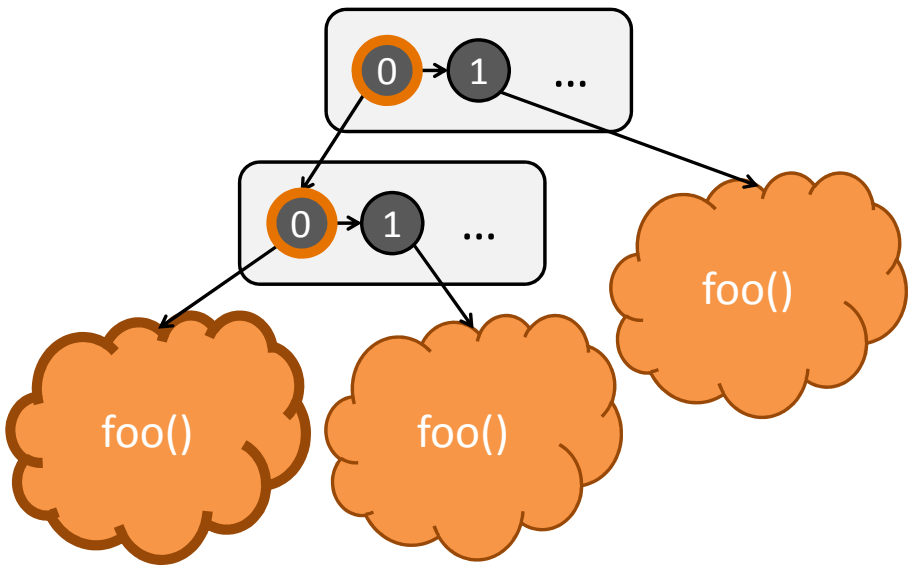


Solution: A ***scoped pedigree*** is a pedigree excluding some prefix, called a ***scope***.

- Initialize the pedigree-based DPRNG with a scope.
- The DPRNG ignores the scope of the pedigree when computing pseudorandom numbers.

Scoped pedigrees

Problem: How do we run a parallel subcomputation with the same random numbers multiple times?



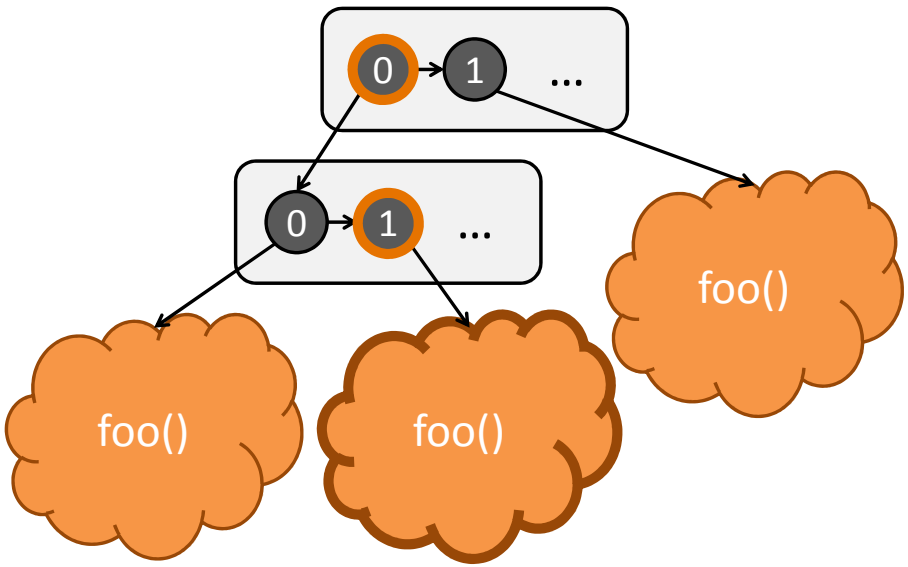
$J_{left} = \langle 0, 0 \rangle + J_{foo}$

Solution: A ***scoped pedigree*** is a pedigree excluding some prefix, called a ***scope***.

- Initialize the pedigree-based DPRNG with a scope.
- The DPRNG ignores the scope of the pedigree when computing pseudorandom numbers.

Scoped pedigrees

Problem: How do we run a parallel subcomputation with the same random numbers multiple times?



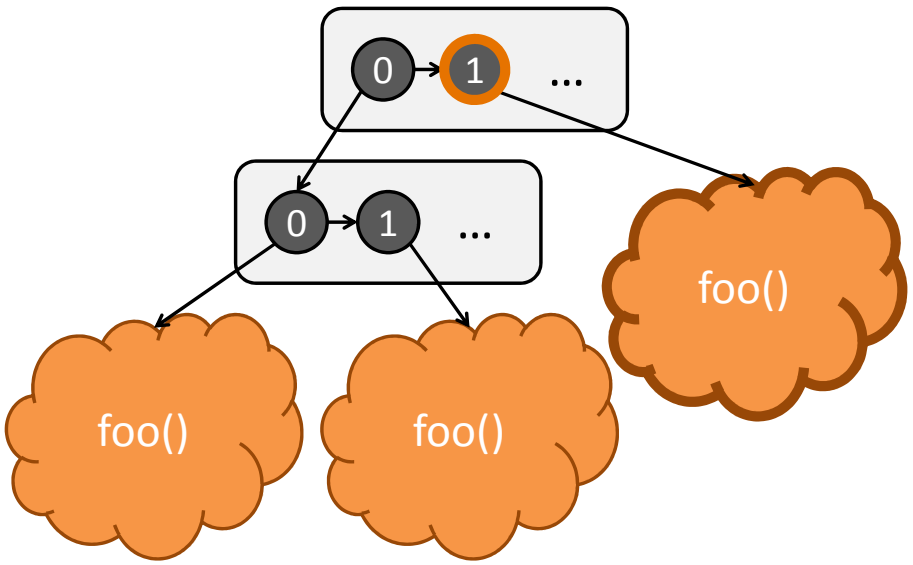
$J_{mid} = \langle 0, 1 \rangle + J_{\text{foo}}$

Solution: A ***scoped pedigree*** is a pedigree excluding some prefix, called a ***scope***.

- Initialize the pedigree-based DPRNG with a scope.
- The DPRNG ignores the scope of the pedigree when computing pseudorandom numbers.

Scoped pedigrees

Problem: How do we run a parallel subcomputation with the same random numbers multiple times?



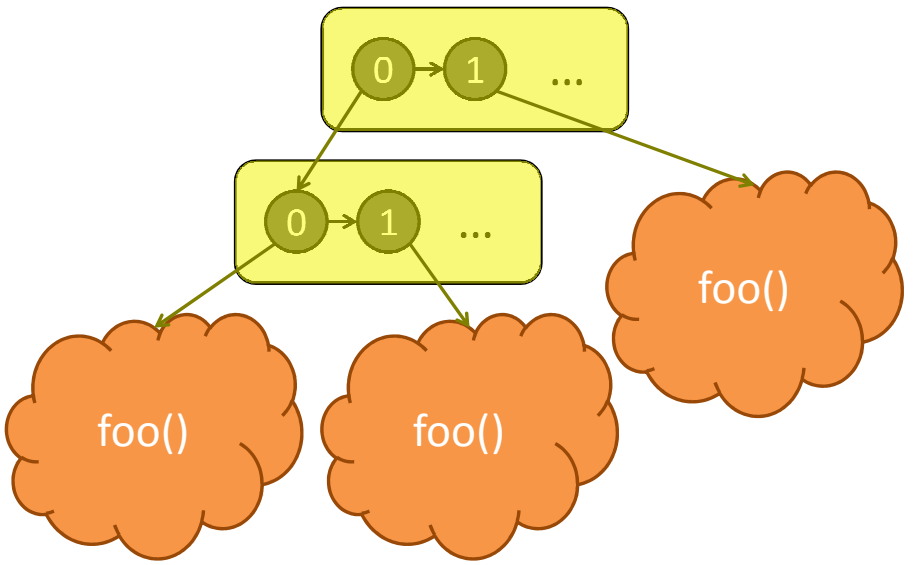
$J_{right} = \langle 1 \rangle + J_{foo}$

Solution: A ***scoped pedigree*** is a pedigree excluding some prefix, called a ***scope***.

- Initialize the pedigree-based DPRNG with a scope.
- The DPRNG ignores the scope of the pedigree when computing pseudorandom numbers.

Scoped pedigrees

Problem: How do we run a parallel subcomputation with the same random numbers multiple times?



$J_{left} = J_{mid} = J_{right} = J_{foo}$

Solution: A ***scoped pedigree*** is a pedigree excluding some prefix, called a ***scope***.

- Initialize the pedigree-based DPRNG with a scope.
- The DPRNG ignores the scope of the pedigree when computing pseudorandom numbers.

Write your own DPRNG!

Write your own DPRNG library! Intel has put the pedigree mechanism into its current compiler release for Cilk Plus, and will publish an interface.

DPRNG library interface

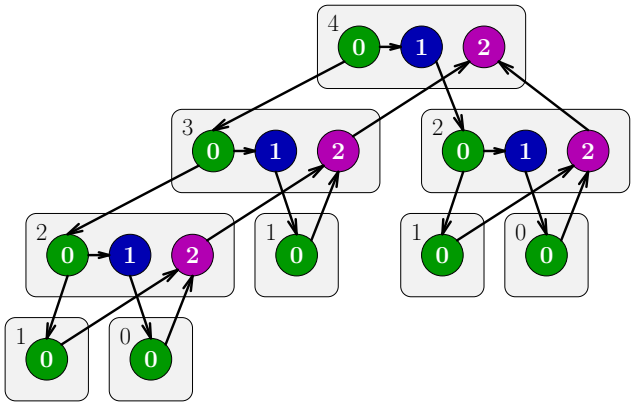
```
template <typename T>
class DPRNG
{
  DPRNG();
  ~DPRNG();
  void seed(uint64_t seed);
  void scope(DPRNG_scope scope);
  uint64_t get();
};
```

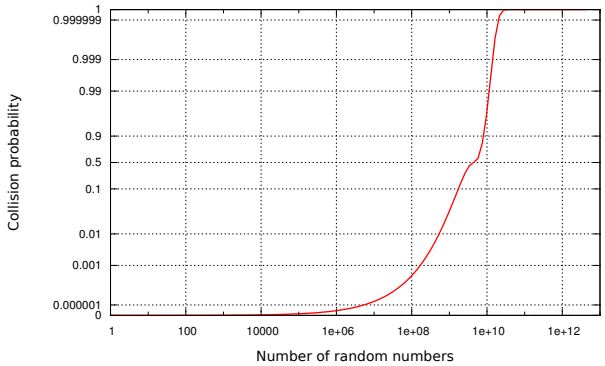
Pedigree interface

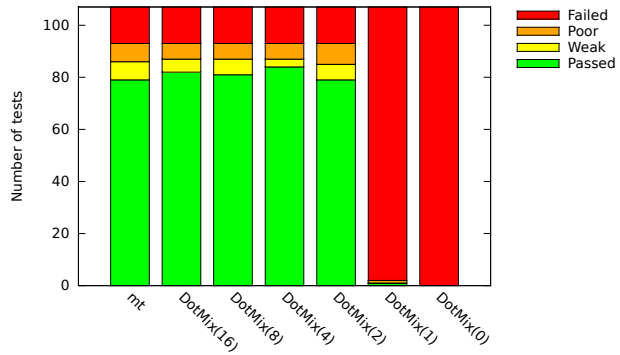
```
typedef struct pedigree
{
  uint64_t rank;
  pedigree* parent;
};
```

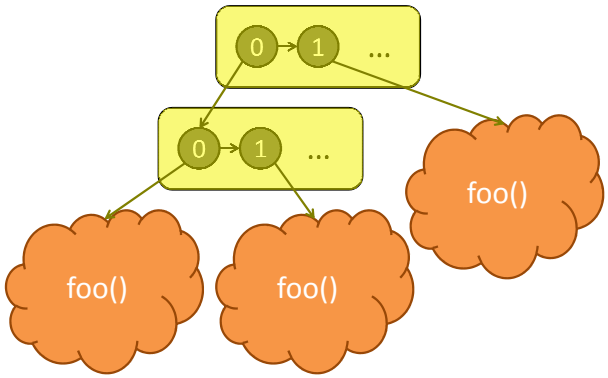
Thank you

Questions?







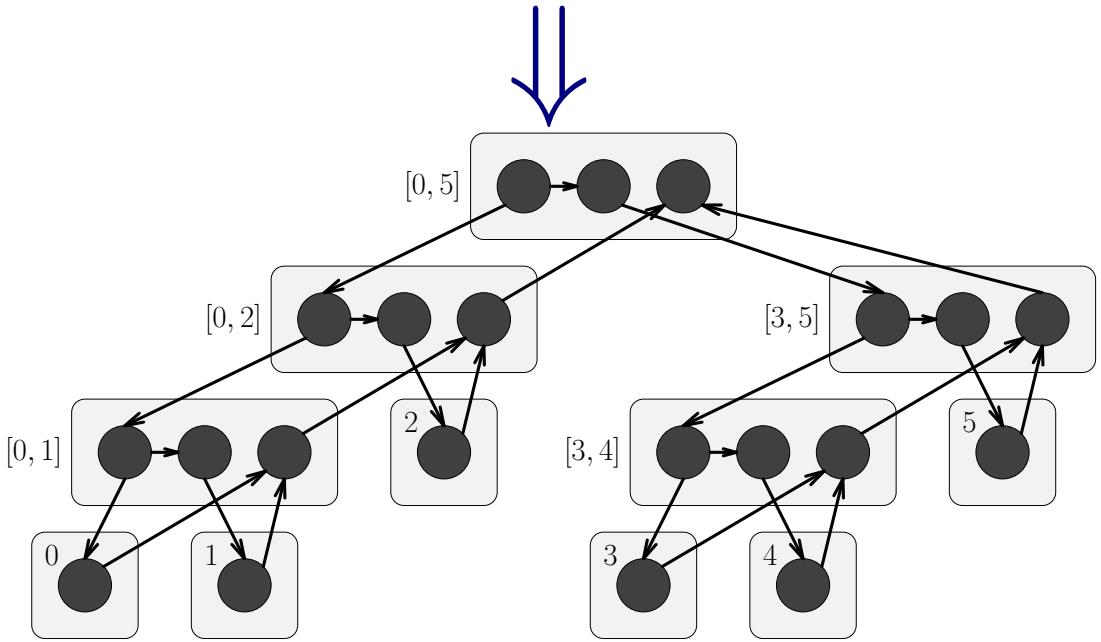


Thanks to Guy Blelloch (CMU), David Chang (MIT), Angelina Lee (MIT), Ron Rivest (MIT), Peter Shor (MIT), Kevin B. Smith (Intel), and Barry Tannenbaum (Intel) for helping us with this research.

Loop pedigrees

Cilk provides a `cilk_for` keyword for writing parallel loops, which is implemented using `cilk_spawn` and `cilk_sync`.

```
int n=6;
cilk_for(int i=0; i<n; ++i)
{ ... }
```



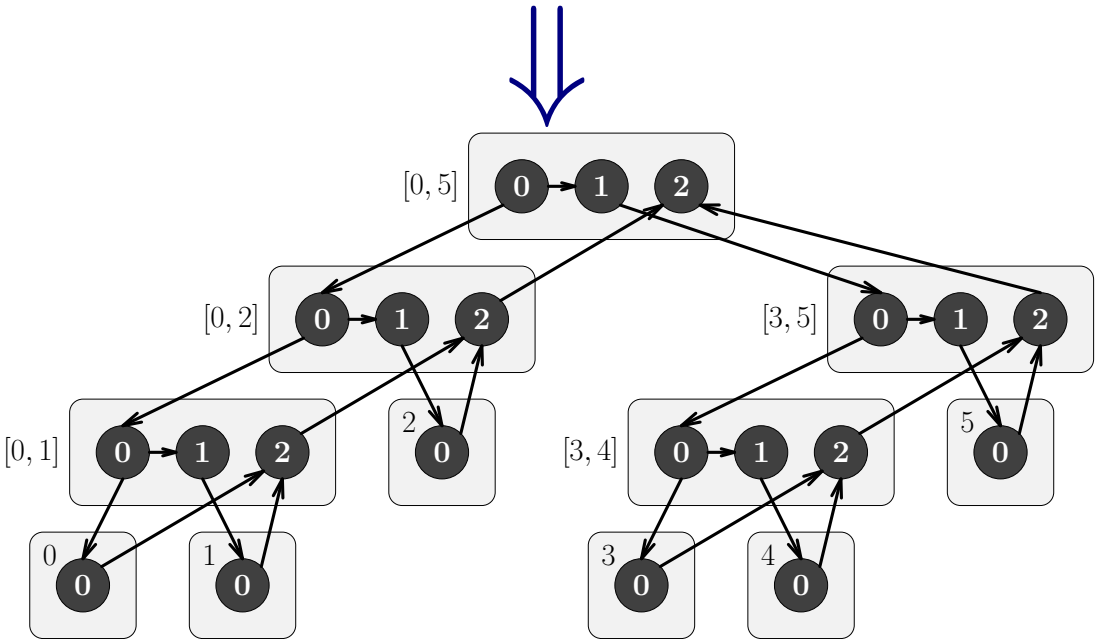
We optimize the pedigrees for `cilk_for` loops.

- The pedigree of a `cilk_for` loop iteration is set to the loop iteration index.
- The cost of reading the pedigree for a loop iteration is reduced by $\Theta(\lg n)$.

Loop pedigrees

Cilk provides a `cilk_for` keyword for writing parallel loops, which is implemented using `cilk_spawn` and `cilk_sync`.

```
int n=6;
cilk_for(int i=0; i<n; ++i)
{ ... }
```



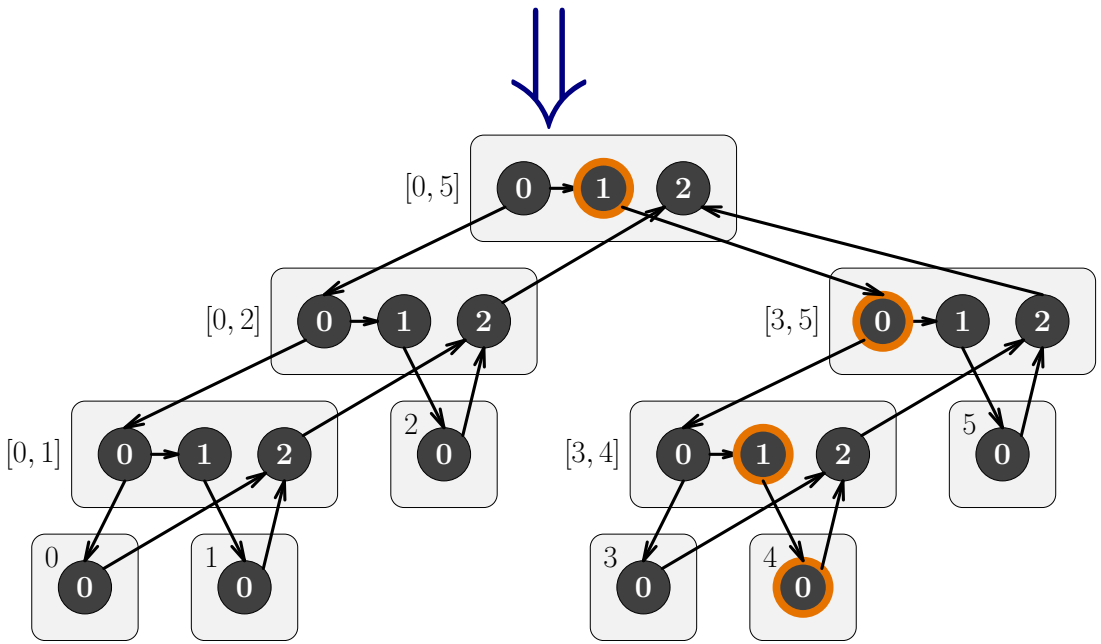
We optimize the pedigrees for `cilk_for` loops.

- The pedigree of a `cilk_for` loop iteration is set to the loop iteration index.
- The cost of reading the pedigree for a loop iteration is reduced by $\Theta(\lg n)$.

Loop pedigrees

Cilk provides a `cilk_for` keyword for writing parallel loops, which is implemented using `cilk_spawn` and `cilk_sync`.

```
int n=6;
cilk_for(int i=0; i<n; ++i)
{ ... }
```



$J = \langle 1, 0, 1, 0 \rangle$

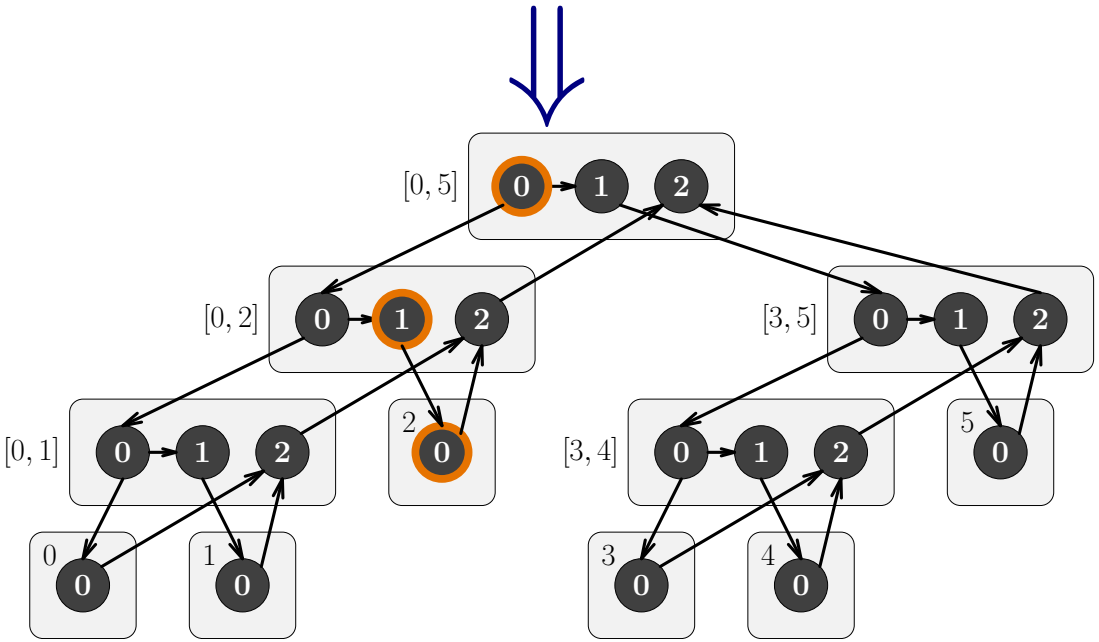
We optimize the pedigrees for `cilk_for` loops.

- The pedigree of a `cilk_for` loop iteration is set to the loop iteration index.
- The cost of reading the pedigree for a loop iteration is reduced by $\Theta(\lg n)$.

Loop pedigrees

Cilk provides a `cilk_for` keyword for writing parallel loops, which is implemented using `cilk_spawn` and `cilk_sync`.

```
int n=6;
cilk_for(int i=0; i<n; ++i)
{ ... }
```



$J = \langle 0, 1, 0 \rangle$

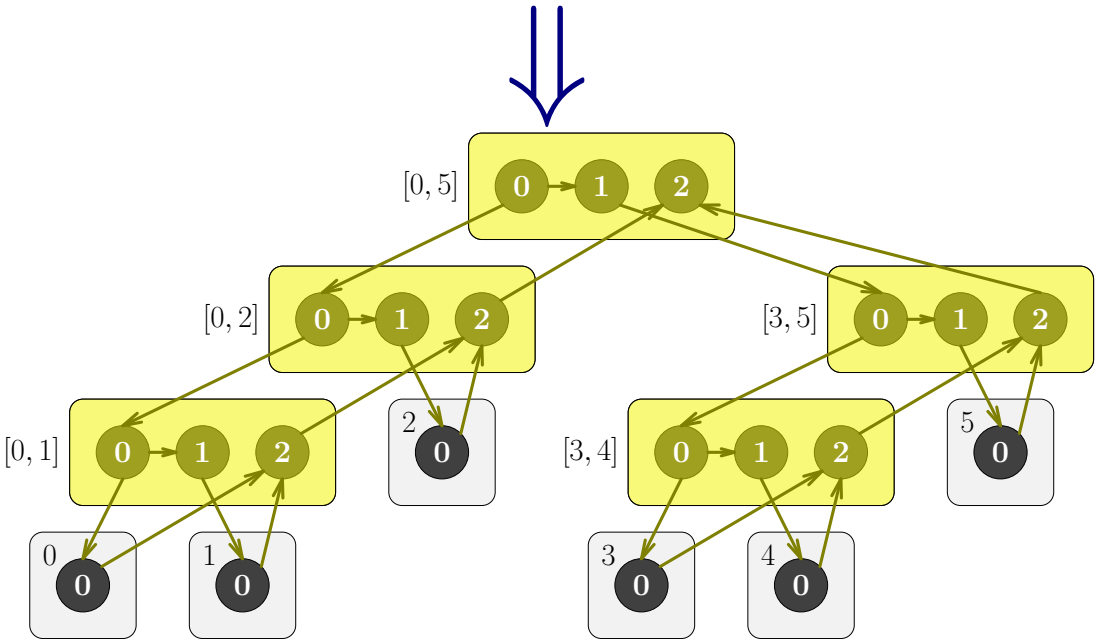
We optimize the pedigrees for `cilk_for` loops.

- The pedigree of a `cilk_for` loop iteration is set to the loop iteration index.
- The cost of reading the pedigree for a loop iteration is reduced by $\Theta(\lg n)$.

Loop pedigrees

Cilk provides a `cilk_for` keyword for writing parallel loops, which is implemented using `cilk_spawn` and `cilk_sync`.

```
int n=6;
cilk_for(int i=0; i<n; ++i)
{ ... }
```



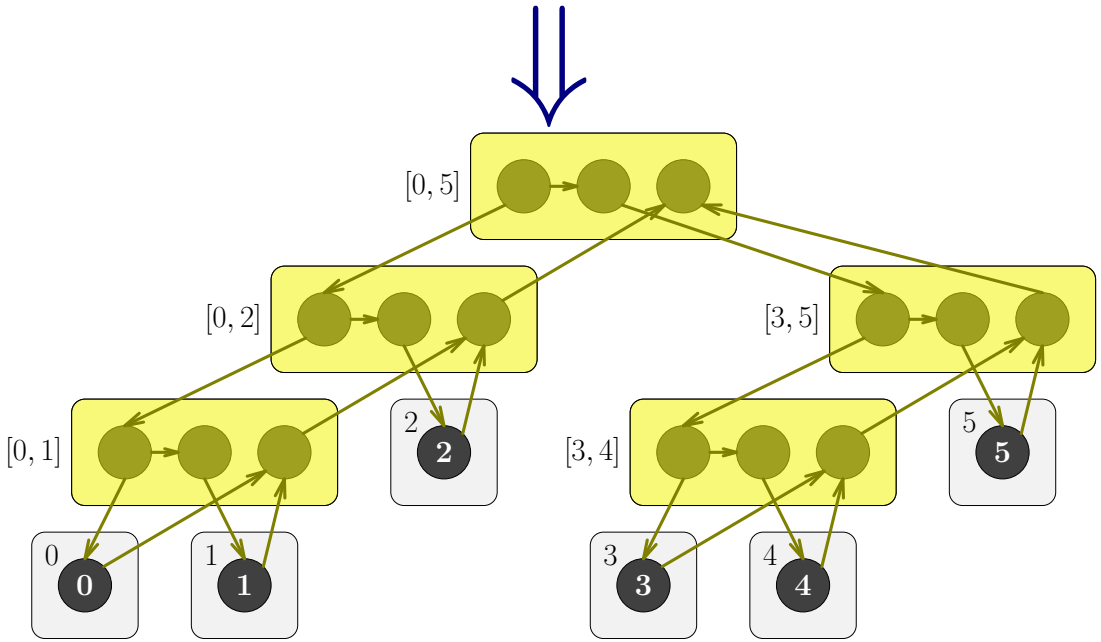
We optimize the pedigrees for `cilk_for` loops.

- The pedigree of a `cilk_for` loop iteration is set to the loop iteration index.
- The cost of reading the pedigree for a loop iteration is reduced by $\Theta(\lg n)$.

Loop pedigrees

Cilk provides a `cilk_for` keyword for writing parallel loops, which is implemented using `cilk_spawn` and `cilk_sync`.

```
int n=6;
cilk_for(int i=0; i<n; ++i)
{ ... }
```



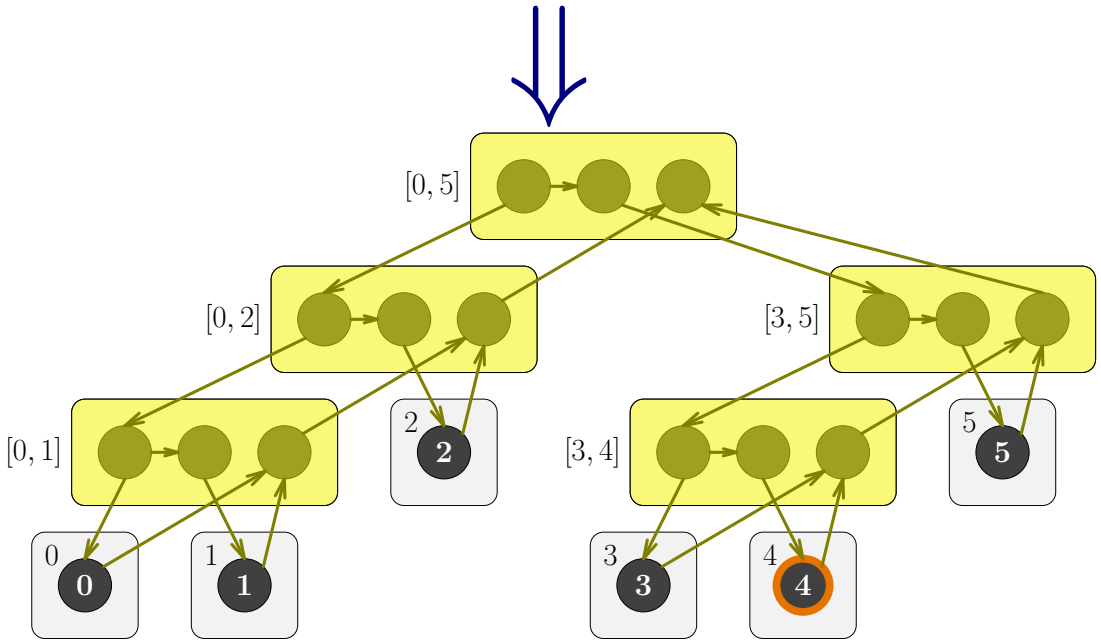
We optimize the pedigrees for `cilk_for` loops.

- The pedigree of a `cilk_for` loop iteration is set to the loop iteration index.
- The cost of reading the pedigree for a loop iteration is reduced by $\Theta(\lg n)$.

Loop pedigrees

Cilk provides a `cilk_for` keyword for writing parallel loops, which is implemented using `cilk_spawn` and `cilk_sync`.

```
int n=6;
cilk_for(int i=0; i<n; ++i)
{ ... }
```



$J = \langle 4 \rangle$

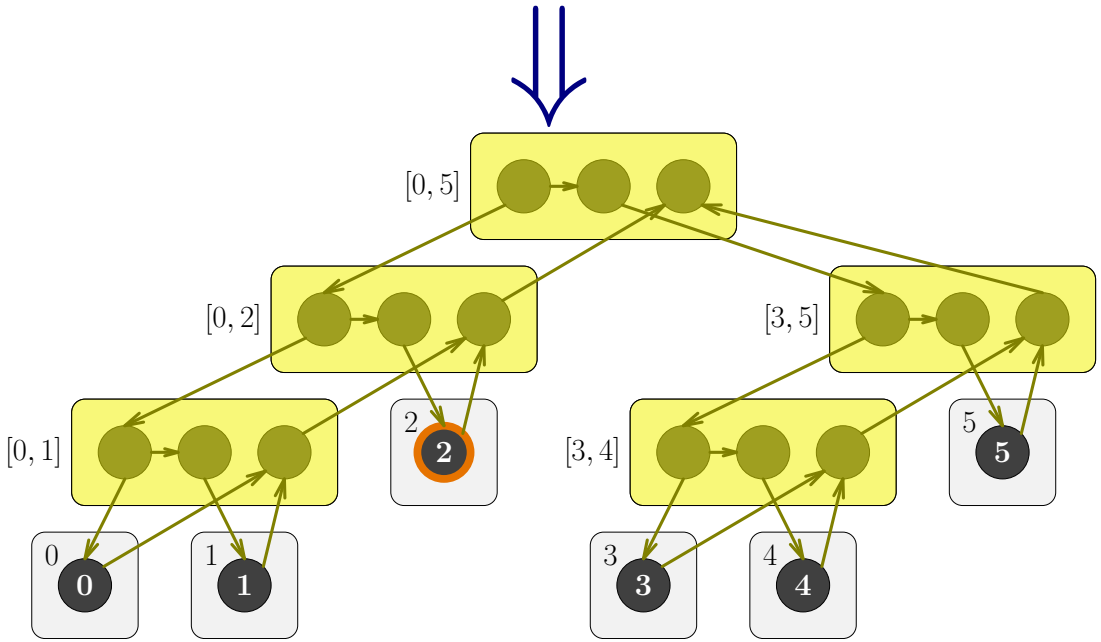
We optimize the pedigrees for `cilk_for` loops.

- The pedigree of a `cilk_for` loop iteration is set to the loop iteration index.
- The cost of reading the pedigree for a loop iteration is reduced by $\Theta(\lg n)$.

Loop pedigrees

Cilk provides a `cilk_for` keyword for writing parallel loops, which is implemented using `cilk_spawn` and `cilk_sync`.

```
int n=6;
cilk_for(int i=0; i<n; ++i)
{ ... }
```



$J = \langle 2 \rangle$

We optimize the pedigrees for `cilk_for` loops.

- The pedigree of a `cilk_for` loop iteration is set to the loop iteration index.
- The cost of reading the pedigree for a loop iteration is reduced by $\Theta(\lg n)$.

DotMix compression

Dot-product compression: Compute the dot product of the pedigree with a vector of random odd 64-bit integers.
Formally: Let $m = 2^{64}$, and let $\Gamma = \langle \gamma_1, \gamma_2, \ldots, \gamma_D \rangle$ be a vector of odd integers chosen uniformly at random from $(2\mathbb{Z}_{m/2} + 1)^D$. DotMix compresses a pedigree $J = \langle j_1, j_2, \ldots, j_D \rangle$ using the function:

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

Theorem: For a randomly chosen compression function c_Γ and any two distinct pedigrees $J, J' \in \mathbb{Z}_m^D$, we have $\Pr\{c_\Gamma(J) = c_\Gamma(J')\} \leq 2/m$.

- For $m = 2^{64}$, the probability of collision is at most $1/2^{63}$.

Efficacy of DotMix

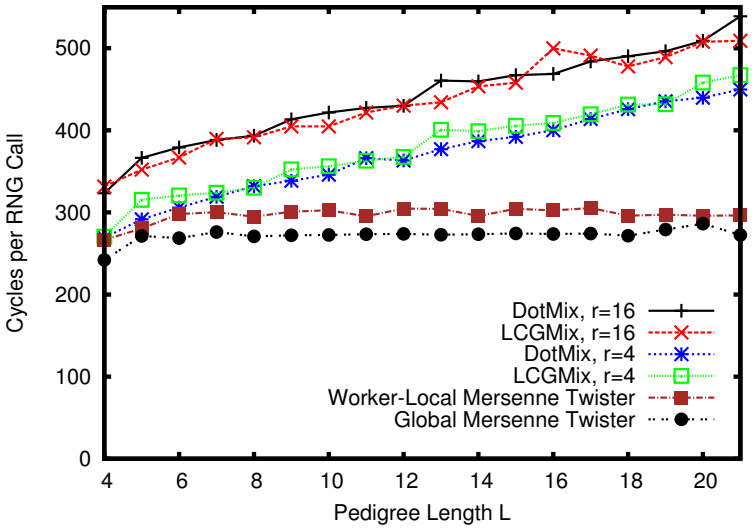
Theorem: For a randomly chosen compression function c_Γ and any two distinct pedigrees $J, J' \in \mathbb{Z}_m^D$, we have $\Pr\{c_\Gamma(J) = c_\Gamma(J')\} \leq 2/m$.
Proof: Suppose $c_\Gamma(J) = c_\Gamma(J')$. WLOG, assume J and J' differ in rank 1. We therefore have (modulo m)

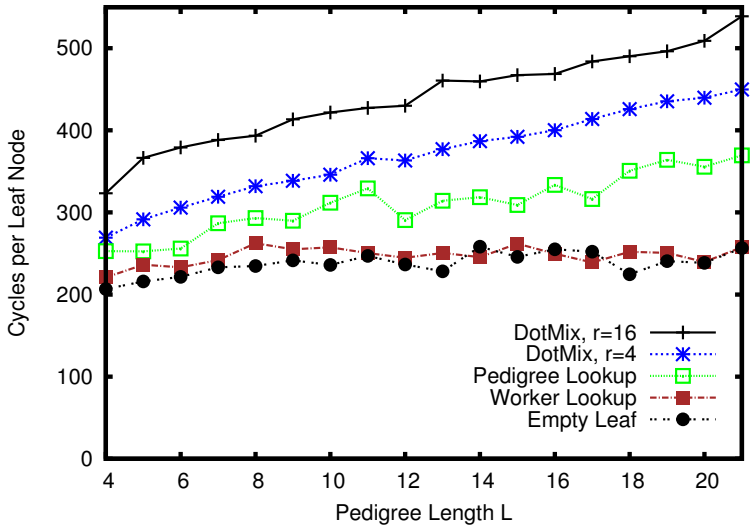
$$0 = c_\Gamma(J) - c_\Gamma(J') = \gamma_1 j_1 - \gamma_1 j_1' + \sum_{k=2}^{D} \gamma_k \cdot j_k - \sum_{k=2}^{D} \gamma_k \cdot j_k'$$

$$\Longrightarrow j_1 - j_1' = \left(\sum_{k=2}^{D} \gamma_k \cdot (j_k' - j_k) \right) \cdot \gamma_1^{-1} .$$

Let $y = j_1 - j_1'$, $a = (\sum_{k=2}^{D} \gamma_k \cdot (j_k' - j_k))$, and $x = \gamma_1^{-1}$. For fixed y and a, there is ≤ 1 choice of $x \in 2\mathbb{Z}_{m/2} + 1$ such that $y = ax$. By unique inverses, there is ≤ 1 choice of γ_1 such that $y = ax$. Hence, the probability that the randomly chosen γ_1 satisfies $y = ax$ is $\leq 2/m$.

DOTMIX performance breakdown





- The longer the pedigree of a strand, the longer DOTMIX takes to run.
- Most of the cost of DOTMIX is in looking up the pedigree.

Linear congruential DPRNG

We can write other DPRNG's using pedigrees, such as LCGMIX, which is based on linear congruential generators.

1. **Compression:** Let X be the state of the DPRNG in the rank-r strand s of function F, and let $a \in_R (2\mathbb{Z}_{m/2} + 1)$ and $b \in_R \mathbb{Z}_m$. Generate two pseudorandom numbers for s's successor strands.
   - The spawned child of s gets $L(X) = (aX) \bmod m$.
   - The rank-r strand of F gets $R(X) = (X + rb) \bmod m$.

   This is similar to the Lehmer-tree scheme of [FHJ+84].
2. **Mixing:** RC6 mixing.

This DPRNG achieves similar performance and statistical quality as DOTMIX, but provides weaker theoretical guarantees.

References

Robert L. Bocchino, Jr., Vikram S. Adve, Sarita V. Adve, and Marc Snir.

Parallel programming must be deterministic by default.
In *HOTPAR*. USENIX, 2009.

Robert G. Brown.

Dieharder: A random number test suite.
Available from `http://www.phy.duke.edu/~rgb/General/dieharder.php`, August 2011.

Scott Contini, Ronald L. Rivest, M. J. B. Robshaw, and Yiqun Lisa Yin.

The security of the RC6 block cipher.
Available from `http://people.csail.mit.edu/rivest/publications.html`, 1998.

Paul Frederickson, Robert Hiromoto, Thomas L. Jordan, Burton Smith, and Tony Warnock.

Pseudo-random trees in Monte Carlo.
Parallel Computing, 1(2):175–180, 1984.

Edward A. Lee.

The problem with threads.
Computer, 39:33–42, 2006.

Makoto Matsumoto and Takuji Nishimura.

Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator.
ACM TOMACS, 8:3–30, 1998.

Michael Mascagni and Ashok Srinivasan.

Algorithm 806: SPRNG: A scalable library for pseudorandom number generation.
ACM TOMS, 26(3):436–461, 2000.