

Performance Engineering of Multicore Software

Developing a Science of Fast Code for the Post-Moore Era

Tao B. Schardl
August 25, 2016



CSAIL

MIT COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE LABORATORY

The Problem with Fast Code

Writing fast code is notoriously hard.

Writing Fast Code in the 1970's–80's



Knuth

“Premature optimization is the root of all evil.” [K79]

“More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason — including blind stupidity.” [W79]



Wulf



Jackson

“The First Rule of Program Optimization:
Don't do it.

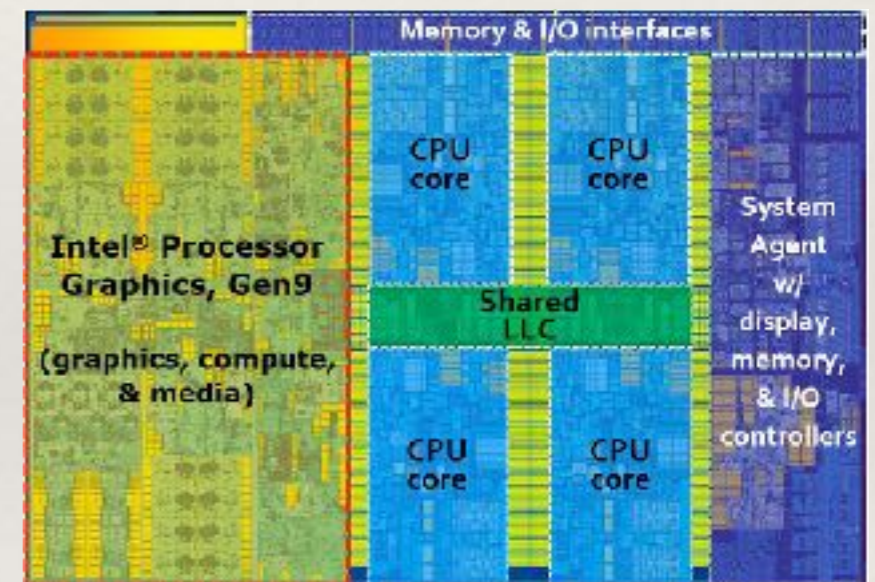
The Second Rule of Program
Optimization — For experts only:
Don't do it yet.” [J88]

Writing Fast Code Today

Today's complicated hardware and software systems make software performance engineering difficult.

Systems on a multicore machine:
parallel processor cores, vector units, caches, memory bandwidth, prefetchers, paging, discs, network bandwidth, GPU's, power...

Intel Skylake processor



Writing code that uses these systems efficiently in concert requires substantial expertise and *ad hoc* knowledge.

Hard Problem: Parallel Programming

“[W]hen we start talking about parallelism and ease of use of truly parallel computers, we’re talking about a problem as hard as any that computer science has faced.” [H06]



Hennessy

- ❖ Parallel programs are hard to reason about and debug because they behave *nondeterministically* due to their concurrent execution.
- ❖ Parallel program performance is hard to reason about because it’s measured in terms of *scalability* as well execution time.

Just Ignore the Problem?

Programmers today prefer to ignore performance concerns and just focus on writing simple, correct code.

Will writing fast code forever be too hard for average programmers to bother doing?

No! Moore's Law is ending!

Thesis Statement

- ❖ I contend that a **science of fast code** can be developed to alleviate the *ad hoc* and unprincipled aspects of software performance engineering.
- ❖ This thesis presents an array of artifacts that enable **principled approaches** to dealing with nondeterminism and scalability concerns in efficient multicore software.
- ❖ These artifacts develop three core technologies that support **scientific inquiry** into the behavior and performance of fast code: **simple programming models, theories of performance** that are borne out in practice, and **efficient diagnostic tools**.

Contributions to a Science of Fast Code

<i>Artifact</i>	<i>Simple programming models</i>	<i>Theories of performance</i>	<i>Efficient diagnostic tools</i>
PBFS		●	
DPRNG	●		
Cilk-P	●	●	
Prism		●	
Color		●	
Cilkprof			●
Rader			●
Tapir		●	
CSI	●		●

● The artifact primarily supports the technology enabling a science of fast code. 8

Content of My Thesis

- ❖ PBFS
- ❖ DPRNG
- ❖ Cilk-P
- ❖ Prism
- ❖ Color
- ❖ Cilkprof
- ❖ Rader
- ❖ Tapir
- ❖ CSI
- ❖ Life after Moore's Law

Content of This Talk

- ❖ PBFS
- ❖ DPRNG
- ❖ Cilk-P
- ❖ Prism
- ❖ Color
- ❖ Cilkprof
- ❖ Rader
- ❖ Tapir
- ❖ CSI
- ❖ Life after Moore's Law

Outline

- ❖ Deterministic parallel random-number generation (DPRNG)
- ❖ Life after Moore's Law



Leiserson



Schardl



Sukha

Randomized Applications

Random numbers are used in a variety of applications, including:

- ❖ Monte Carlo methods
- ❖ Black Scholes
- ❖ Machine learning (e.g., stochastic gradient descent)
- ❖ Computations on social networks
- ❖ Simulated annealing

How do we parallelize these applications?

Review of Serial RNG's

- ❖ Object with state.
- ❖ Each call to `rand` updates the state and returns a pseudorandom number.
- ❖ State is initialized with a seed.
- ❖ The RNG behaves *deterministically* for a fixed seed.

We want this property for debugging parallel codes.

Example serial RNG

```
class LCG {
public:
    LCG()
        : _s(0), _a(1103515245),
          _c(12345), _m(2147483648)
    {}

    int rand() {
        _s = (_a * _s + _c) % _m;
        return _s;
    }

    void seed(unsigned int seed) {
        _s = seed;
    }

protected:
    int _a, _c;
    unsigned int _m, _s;
};
```

Deterministic Parallelism

To address the difficulty of contending with nondeterminism, many researchers have called for some form of **deterministic parallelism**.

“Parallel programming must be deterministic by default.” [BAAS09]

“We should build from essentially deterministic, composable components.” [L06]

“Internally deterministic parallel algorithms can be fast.” [BFGS12]



Bocchino



Adve



Lee



Adve



Snir



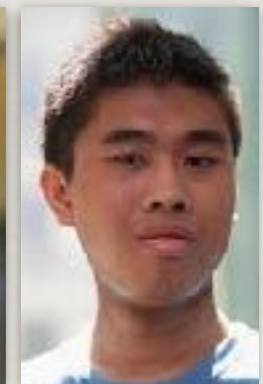
Blelloch



Fineman



Gibbons



Shun

Dynamic Multithreading

Dynamic multithreading concurrency platforms (e.g., Cilk [FLR98], DPJ [BAAS09], Habanero-Java [CZH11], OpenMP [ACD09], Java Fork/Join Framework [L00], TBB [R07], TPL [LH07]) enable programmers to write **deterministic parallel programs**.

Example Cilk code

```
void pqsort(int64_t array[], size_t n,
           size_t l, size_t h) {
    // ... base case ...
    size_t part;
    part = partition(array, n, l, h);
    cilk_spawn pqsort(array, n, l, part);
    pqsort(array, n, part, h);
    cilk_sync;
}
```

More on this shortly.

DPRNG Contributions [LSS12]

The *DotMix* DPRNG library produces pseudorandom numbers for dynamic multithreaded codes.

- ❖ DotMix is **deterministic** by making use of the *pedigree* mechanism.
- ❖ DotMix produces **high-quality numbers** — numbers of comparable statistical quality to the state-of-the-art Mersenne twister [MN98] RNG.
- ❖ The DotMix implementation is **fast**, incurring relatively little overhead compared to a nondeterministic use of Mersenne twister.

Outline

- ❖ DPRNG
 - ❖ Dynamic multithreading
 - ❖ The DPRNG problem
 - ❖ Pedigrees
 - ❖ DotMix
 - ❖ Evaluation
- ❖ Life after Moore's Law

A Simple Parallel Quicksort

Dynamic multithreading language constructs expose logical parallelism within a program.

Example Cilk code

```
void pqsort(int64_t array[], size_t n,
            size_t l, size_t h) {
    // ... base case ...
    size_t part;
    part = partition(array, n, l, h);
    cilk_spawn pqsort(array, n, l, part);
    pqsort(array, n, part, h);
    cilk_sync;
}
```

This call to pqsort is *allowed* (but not required) to execute in parallel with its continuation.

Both recursive pqsort calls must return before control passes this point.

Processor-Oblivious Execution Model

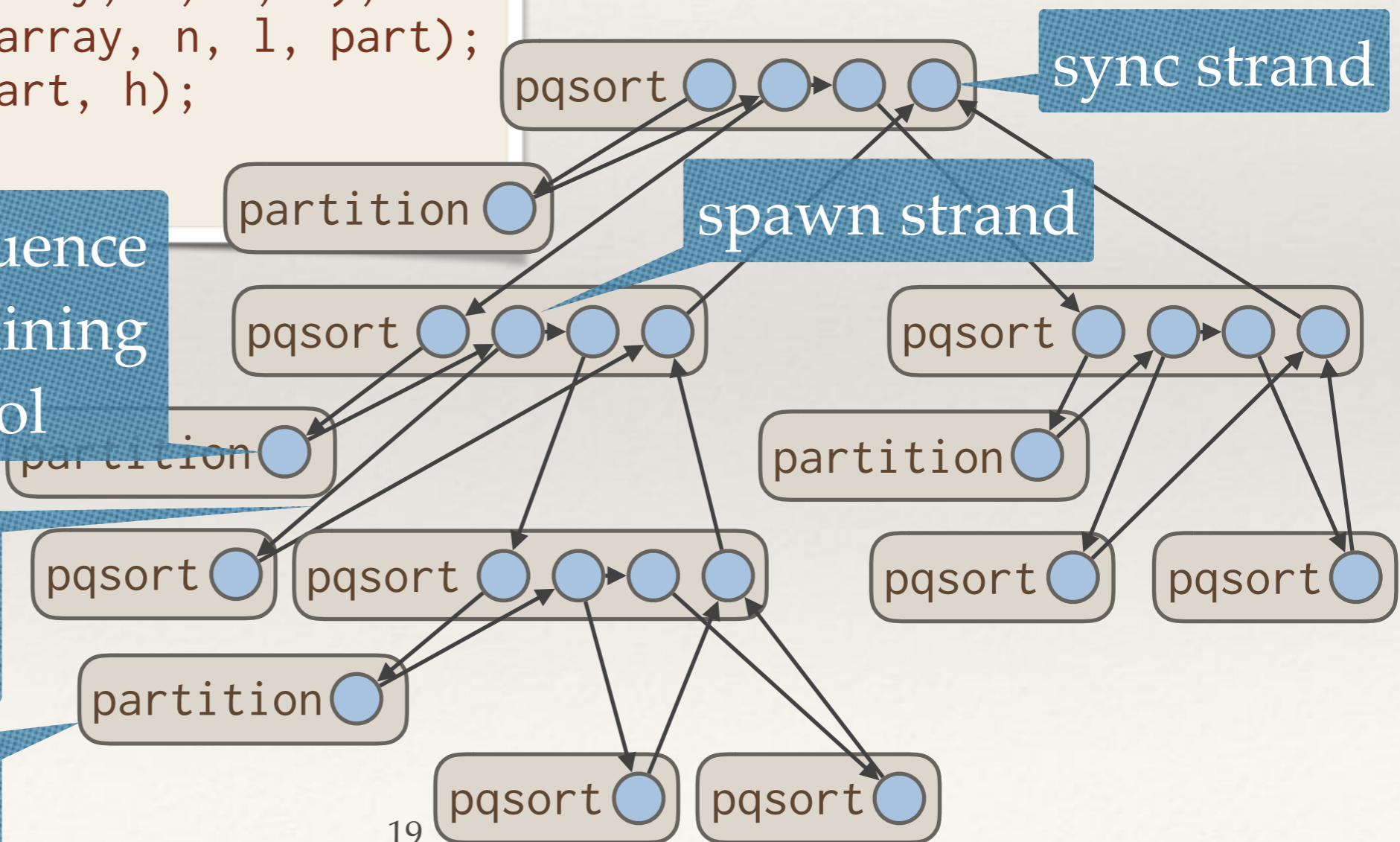
```
void pqsort(int64_t array[], size_t n,  
           size_t l, size_t h) {  
    // ... base case ...  
    size_t part;  
    part = partition(array, n, l, h);  
    cilk_spawn pqsort(array, n, l, part);  
    pqsort(array, n, part, h);  
    cilk_sync;  
}
```

The executed **computation** forms a dag embedded in the function invocation tree.

Strand — serial sequence of instructions containing no parallel control

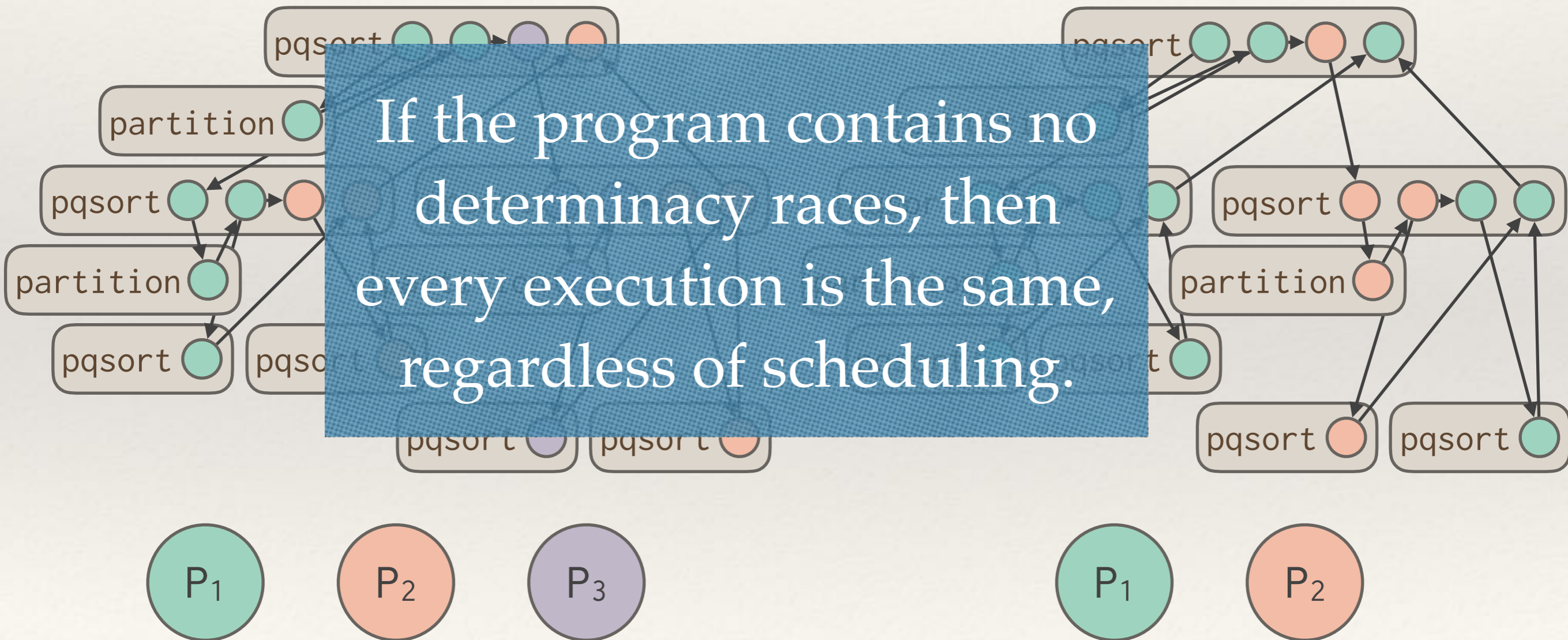
parallel control dependency

function frame



Scheduling on Parallel Processors

The runtime system automatically load-balances the program efficiently on available processors.



Outline

- ❖ DPRNG
 - ❖ Dynamic multithreading
 - ❖ The DPRNG problem
 - ❖ Pedigrees
 - ❖ DotMix
 - ❖ Evaluation
- ❖ Life after Moore's Law

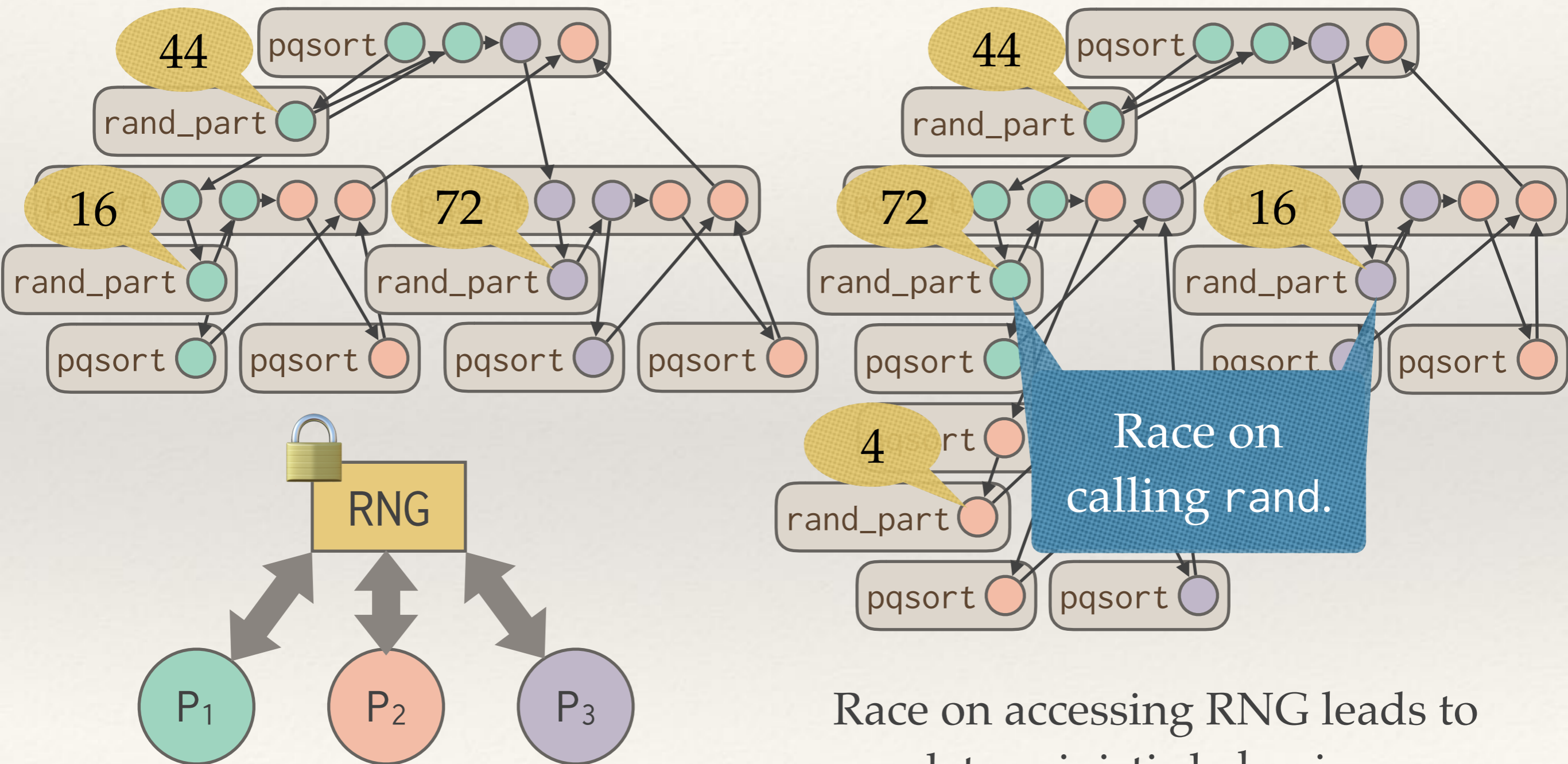
Randomized Parallel Quicksort

```
void pqsort(int64_t array[], size_t n,  
            size_t l, size_t h) {  
    // ... base case ...  
    size_t part;  
    part = rand_partition(array, n, l, h);  
    cilk_spawn pqsort(array, n, l, part);  
    pqsort(array, n, part, h);  
    cilk_sync;  
}
```

Partition the array randomly to guarantee $O(n \log n)$ running time with high probability.

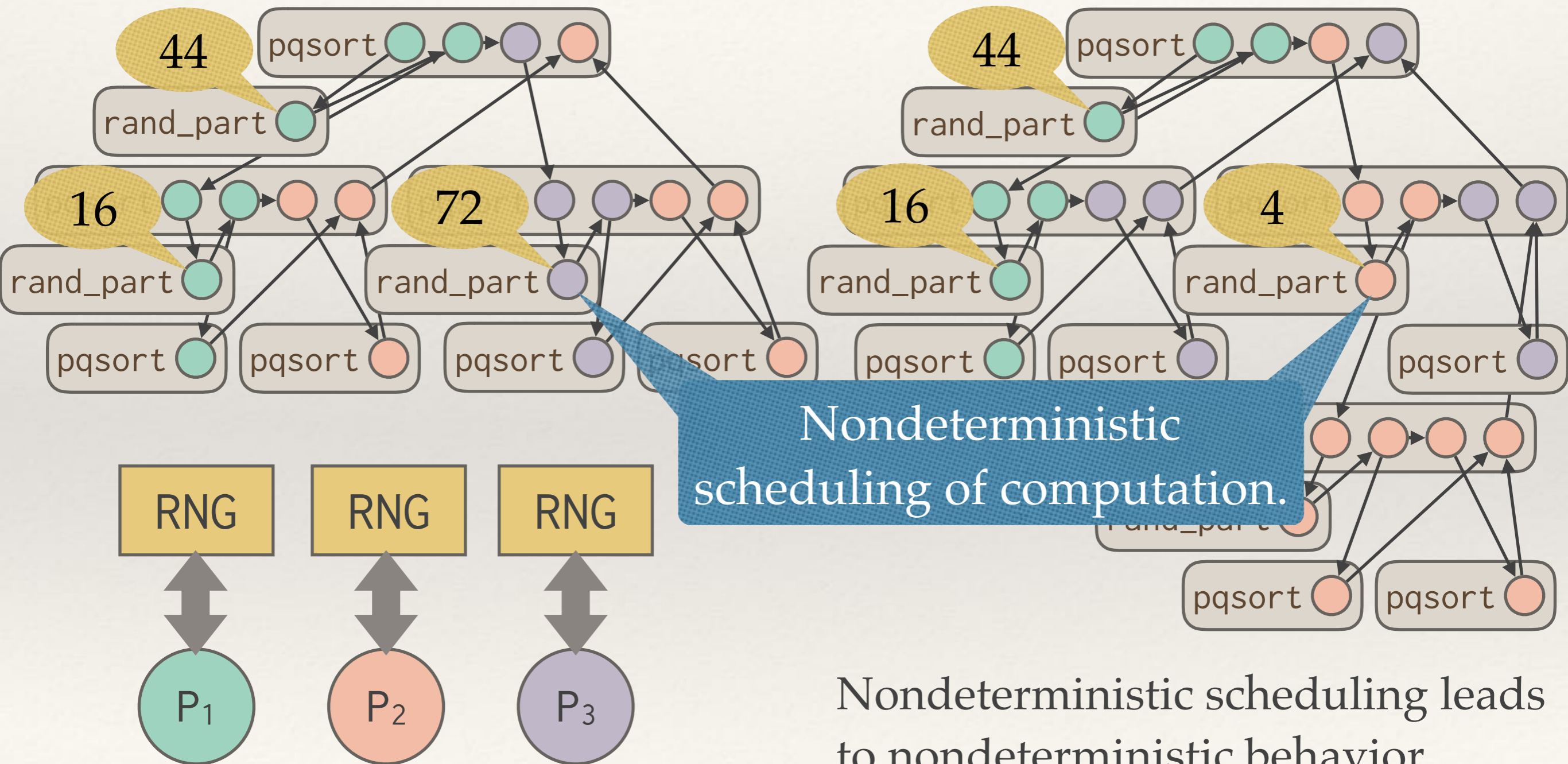
How do we generate the pseudorandom numbers for `rand_partition`?

Idea 1: Global RNG



Race on accessing RNG leads to nondeterministic behavior.

Idea 2: Processor-Local RNG's



Nondeterministic scheduling leads to nondeterministic behavior.

Idea 3: Spawning RNG's

Assign a different RNG for each spawned subroutine.

```
void pqsort(int64_t array[], size_t n,  
            size_t l, size_t h, RNG r) {  
    // ... base case ...  
    size_t part;  
    part = rand_partition(array, n, l, h, r);  
    RNG r2 = // ???  
    cilk_spawn pqsort(array, n, l, part, r2);  
    pqsort(array, n, part, h, r);  
    cilk_sync;  
}
```

Create a new RNG object for each spawned subroutine.

Pass different RNG's to appropriate methods.

Issues:

1. Not obvious how to make many new RNG's and still produce quality pseudorandom numbers.
2. Can't use a global RNG.
3. Might require extensive code changes.

Previous Research

Previous research has developed DPRNG's for Pthreaded programs:

- ❖ SPRNG [MS01] is a popular DPRNG that creates independent RNG's for different Pthreads via a parameterization process.
- ❖ Coddington [C97] surveys alternative RNG-creation schemes, such as “leapfrogging” and “splitting.”
- ❖ Salmon *et al.* [SMDS11] explore the idea of generating parallel RNG's via independent transformations of counter values.

Idea 3: Spawning RNG's

Idea: Create a new, independent RNG for each spawned subcomputation.

SPRNG [MS01] provides a spawn routine for creating new RNG's.

```
void pqsort(int64_t array[], size_t n,
            size_t l, size_t h, SPRNG r) {
    // ... base case ...
    size_t part;
    part = rand_partition(array, n, l, h, r);
    SPRNG r2 = r.spawn();
    cilk_spawn pqsort(array, n, l, part, r2);
    pqsort(array, n, part, h, r);
    cilk_sync;
}
```

Problem: When we tried using SPRNG on a simple recursive Cilk code, we found that:

- ❖ SPRNG runs 50,000 times slower than using Mersenne twister nondeterministically.
- ❖ SPRNG could not guarantee the independence of the numbers generated for computations that perform many spawns.

SPRNG is not designed to handle dynamic multithreaded computations.

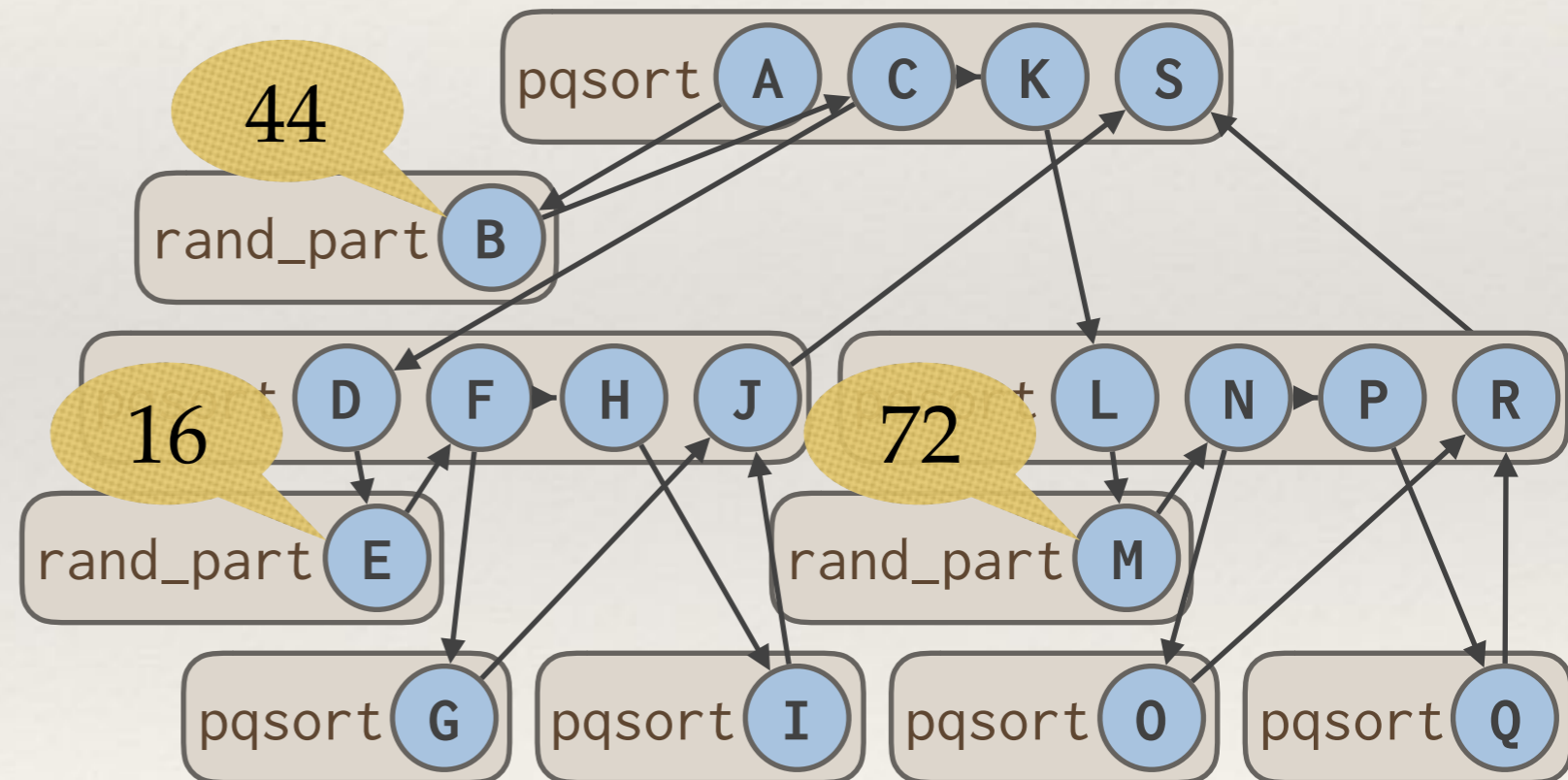
Outline

- ❖ DPRNG
 - ❖ Dynamic multithreading
 - ❖ The DPRNG problem
 - ❖ Pedigrees
 - ❖ DotMix
 - ❖ Evaluation
- ❖ Life after Moore's Law

Idea 4: The Platform Helps

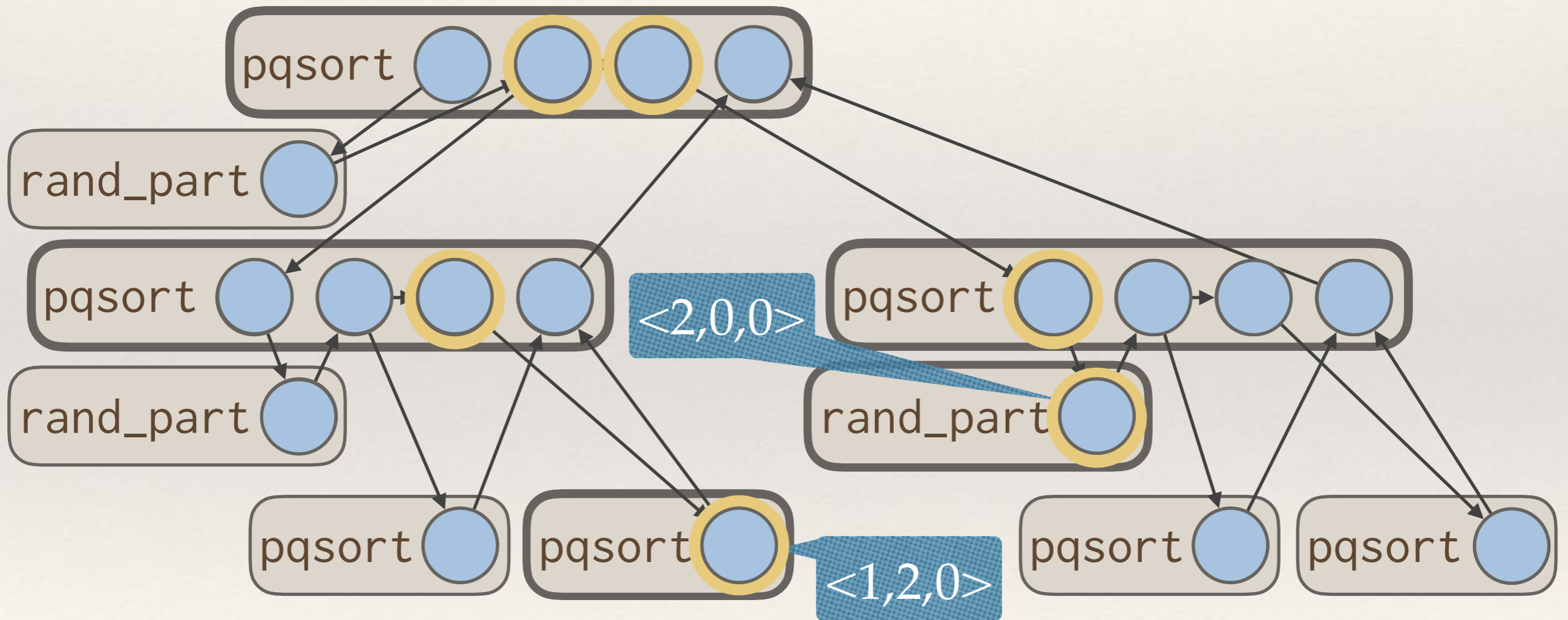
Idea: If each strand is assigned schedule-independent coordinates, then the RNG just needs to hash those coordinates.

- ❖ Coordinates are **deterministic** for a given computation.
- ❖ Call to `rand` can **encapsulate** the extraction of coordinates.
- ❖ **Number quality** and **efficiency** depend on hash function and compiler/runtime system.



Pedigrees

The pedigree mechanism provides schedule-independent coordinates for all strands as the computation unfolds dynamically.

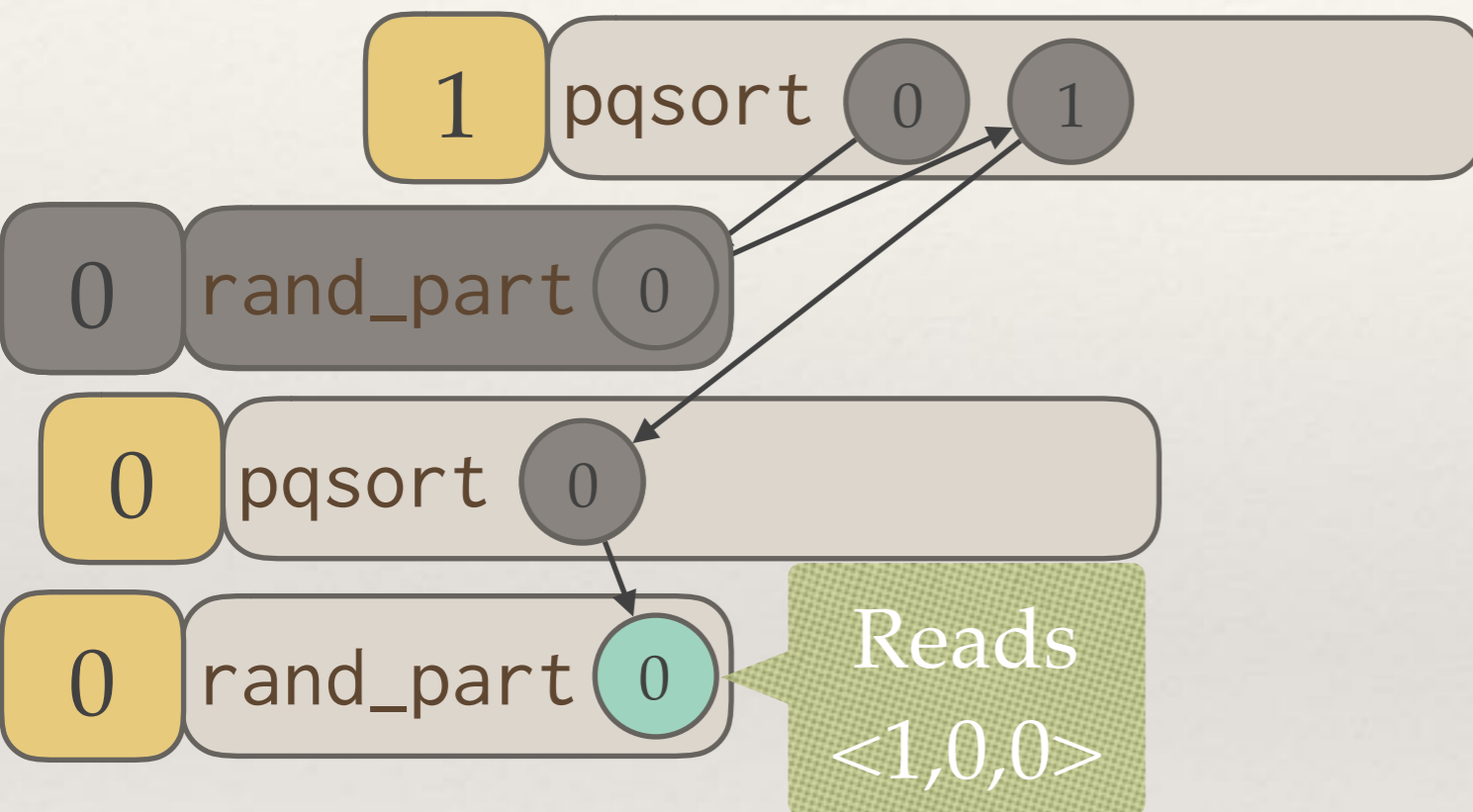


❖ Assign each strand a *rank* in its function frame.

❖ A *pedigree* is a vector of ranks of *ancestor strands*.

Idea for Maintaining Pedigrees

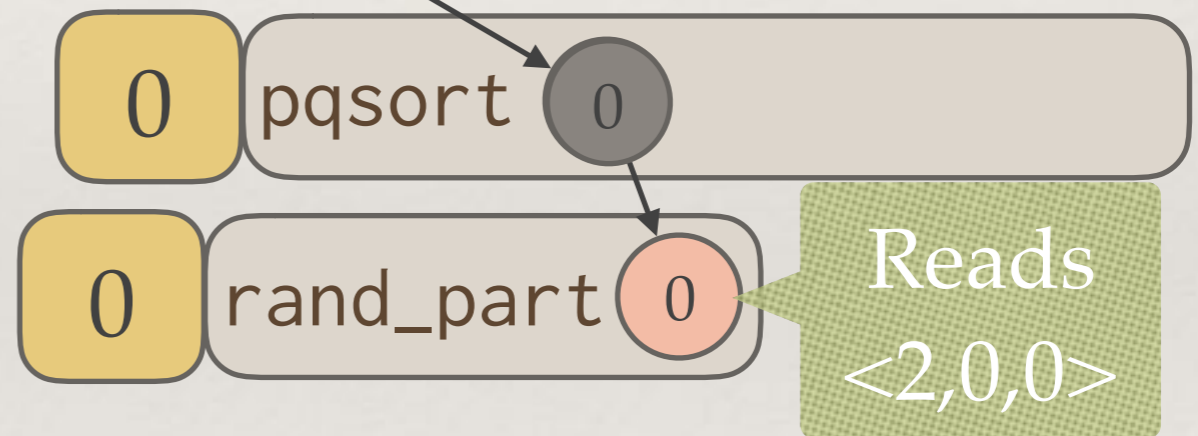
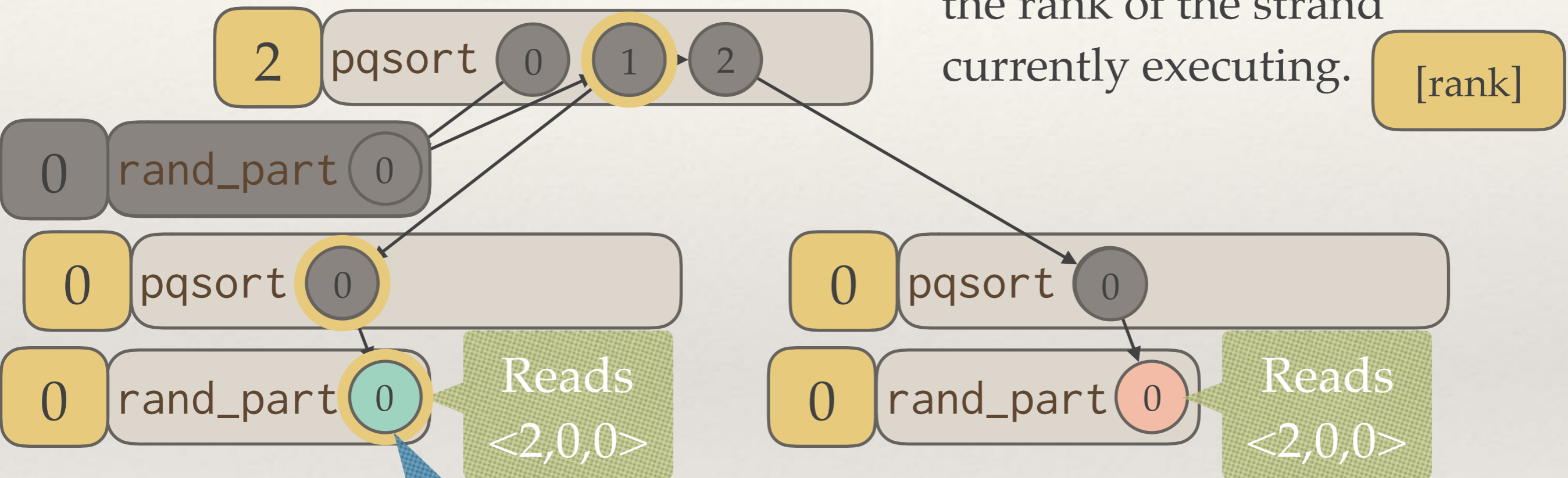
Idea: Each frame maintains the rank of the strand currently executing. [rank]



Reading the pedigree involves reading the ranks in the current frame and all ancestor frames.

Problem with Idea

Idea: Each frame maintains the rank of the strand currently executing. [rank]



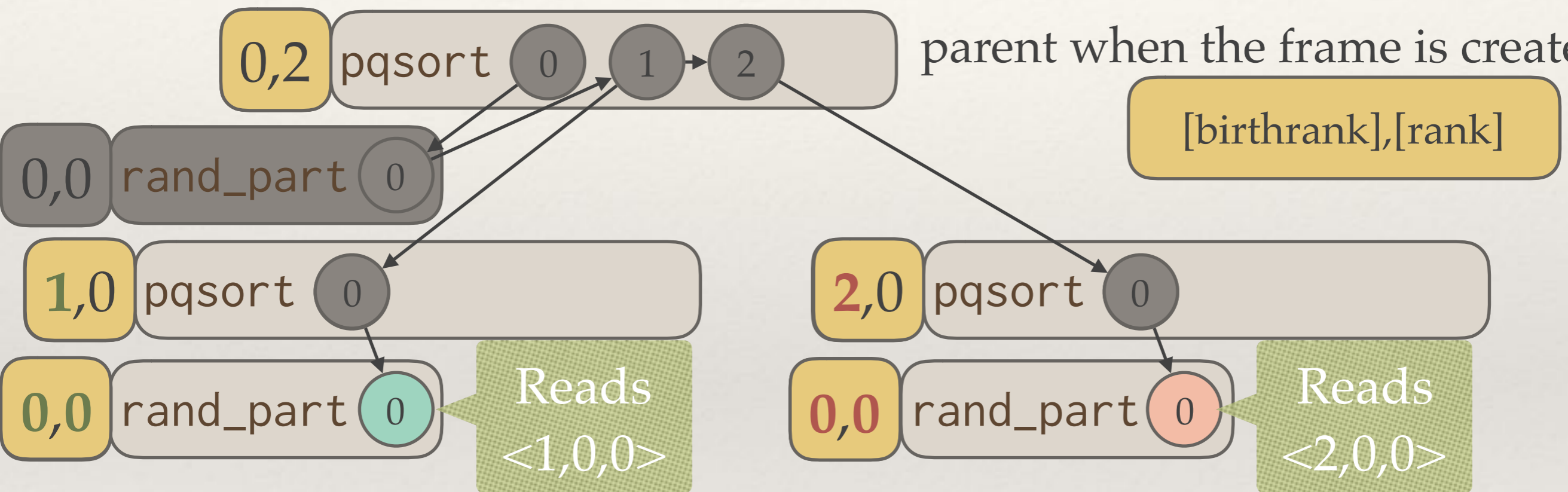
Reading the pedigree
reading the ranks in the current
frame and all ancestor frames.

Actual pedigree
is <1,0,0>!

Ranks can change when parallel
processors execute different
strands in an ancestor frame.

How To Maintain Pedigrees

Each frame also stores a *birthrank* — the rank in the parent when the frame is created.



Reading the pedigree involves reading the rank of the current frame and the birthrank of the current frame and all ancestor frames.

A frame's birthrank is constant for the lifetime of the frame.

Pseudocode to Maintain Pedigrees

The Cilk compiler and runtime system maintain pedigrees with 2 integers per Cilk frame, $\Theta(1)$ time, and no additional synchronization.

On a *spawn* of F from G:

```
1   $G \rightarrow rank = p \rightarrow rank$ 
2   $G \rightarrow sp-rep = p \rightarrow current-frame$ 
3   $\hat{F} \rightarrow brank = G \rightarrow rank$ 
4   $\hat{F} \rightarrow parent = G \rightarrow sp-rep$ 
5   $p \rightarrow rank = 0$ 
6   $p \rightarrow current-frame = \hat{F}$ 
```

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   $p \rightarrow rank = G \rightarrow rank++$ 
2   $p \rightarrow current-frame = G \rightarrow sp-rep$ 
```

Reading a Cilk pedigree takes $\Theta(d)$ time, where d is the depth of nested spawns.

Outline

- ❖ DPRNG
 - ❖ Dynamic multithreading
 - ❖ The DPRNG problem
 - ❖ Pedigrees
 - ❖ DotMix
 - ❖ Evaluation
- ❖ Life after Moore's Law

A Pedigree-Based DPRNG: DotMix

DotMix hashes a pedigree into a pseudorandom number in two steps:

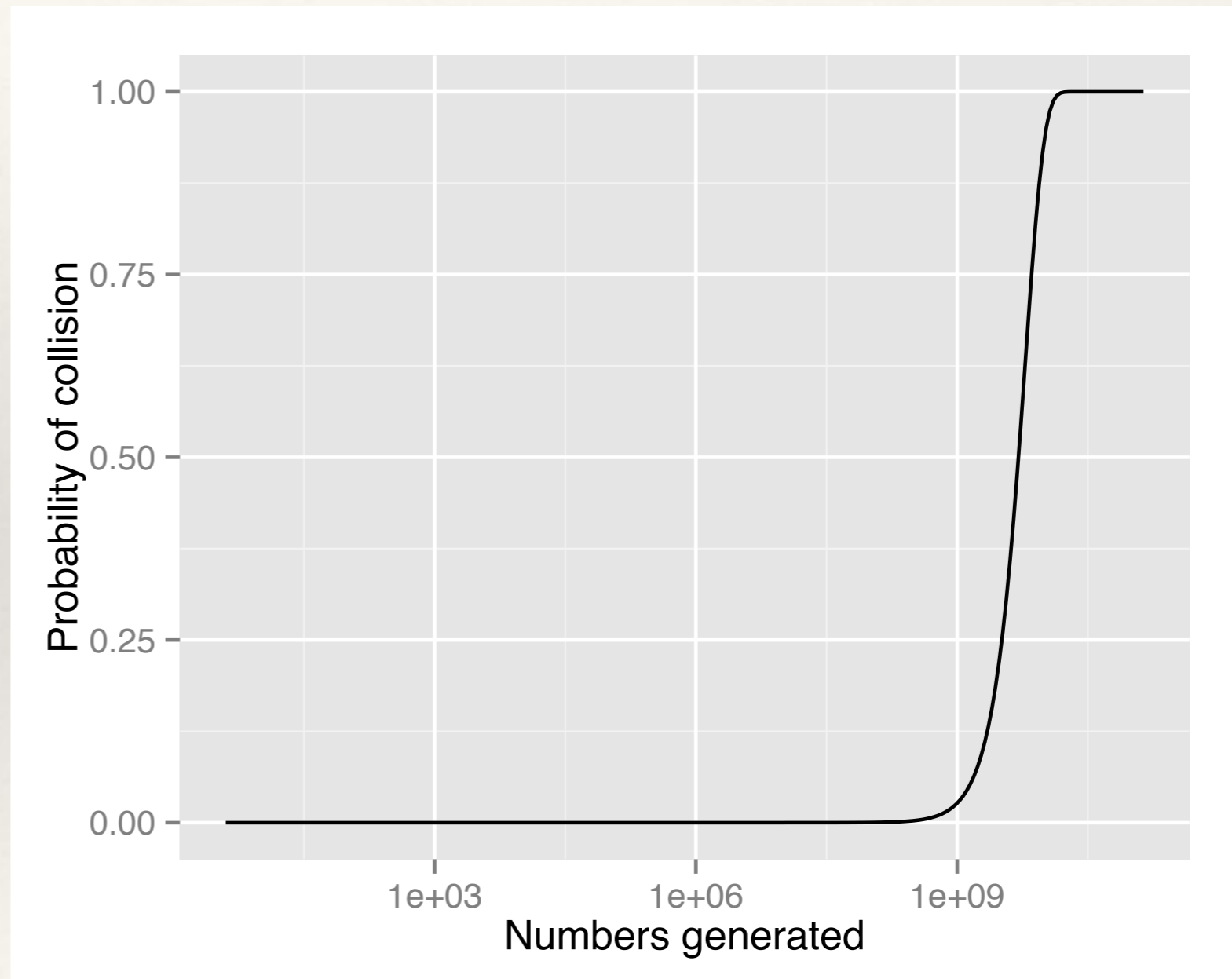
1. *Compression*: Convert the pedigree into a single machine word while preserving uniqueness.
2. *Mixing*: Remove correlations between compressed pedigrees.

Step 1. Compressing a Pedigree

- ❖ *Dot-product compression:* Compute the dot product of the pedigree J and a random vector Γ of integers mod p , where p is a prime.
- ❖ **Theorem:** This hash is 2-independent: for any randomly chosen vector Γ , any two distinct pedigrees J and J' , and two arbitrary values h and h' , the probability that $\Gamma \cdot J = h$ and $\Gamma \cdot J' = h'$ is at most $1/p^2$.
- ❖ **Corollary:** The probability of $\Gamma \cdot J = \Gamma \cdot J'$ is at most $1/p$.

Efficacy of Compression

For the DotMix implementation ($p = 2^{64}-59$), a program can compress half a billion pedigrees and, with 99% probability, no compressions will collide.



Step 2. Mixing the Result

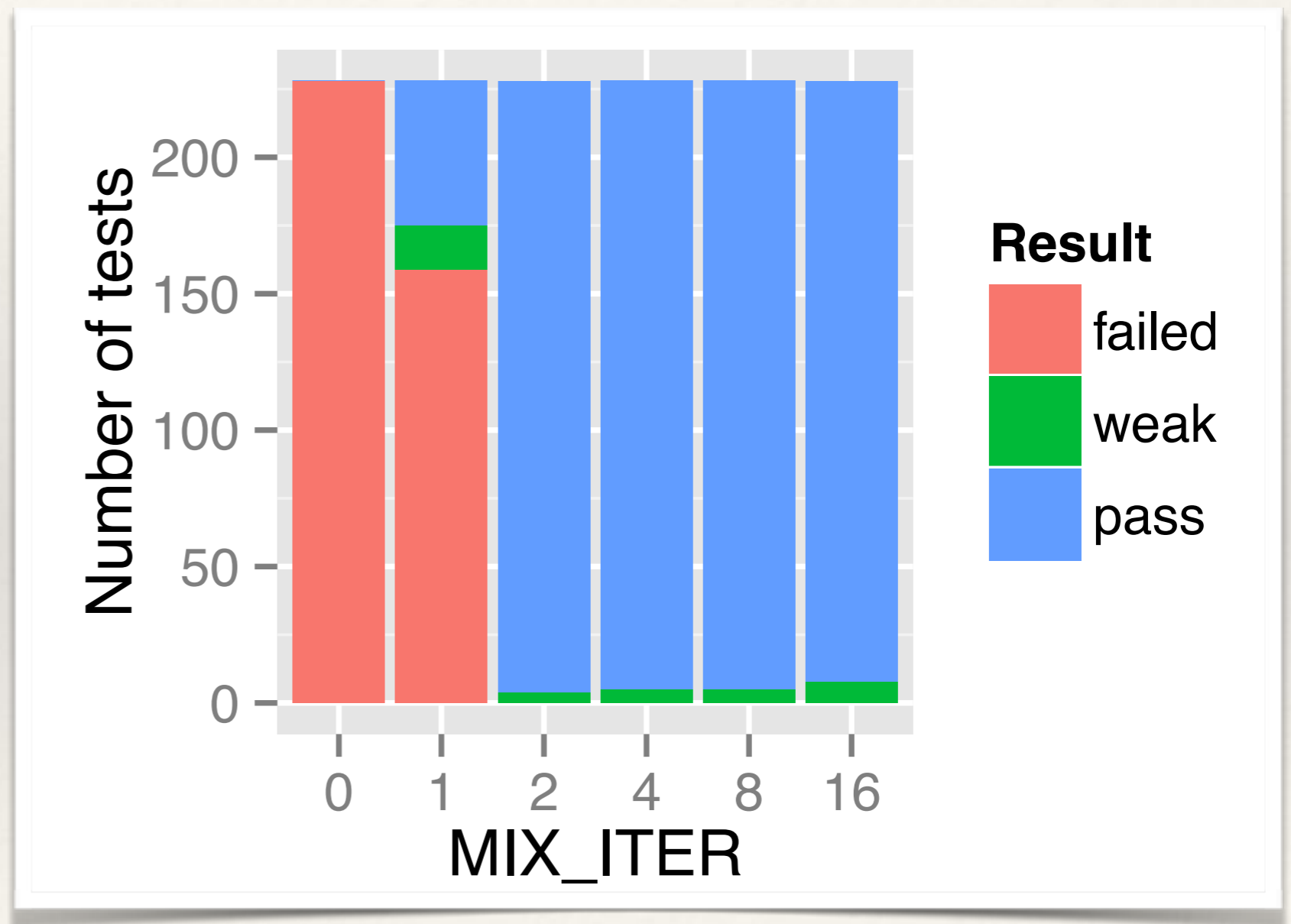
Mixing: Permute the compressed pedigree using `MIX_ITER` iterations of this mixing routine from RC6:

```
uint64_t x; // Compressed pedigree
for (int i = 0; i < MIX_ITER; ++i) {
    x = x * (2 * x + 1); // mod 2^64
    x = (x << 32) | (x >> 32);
}
```

Because this function is a bijective mapping [CRRY98], mixing does not generate additional collisions.

Dieharder Statistical Test Results

DotMix with
 $MIX_ITER \geq 2$
performs as well as
Mersenne twister
on the Dieharder
statistical tests.



Outline

- ❖ DPRNG
 - ❖ Dynamic multithreading
 - ❖ The DPRNG problem
 - ❖ Pedigrees
 - ❖ DotMix
 - ❖ Evaluation
- ❖ Life after Moore's Law

Pedigree Overhead

Across 10 benchmarks, the overhead to maintain pedigrees in the MIT Cilk runtime is less than 1% on average (geometric mean).

<i>Application</i>	<i>Default (s)</i>	<i>Pedigree (s)</i>	<i>Overhead</i>
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

DotMix Performance

Comparing DotMix to the nondeterministic processor-local Mersenne twister solution:

- ❖ DotMix is 2.3 times as costly as Mersenne twister in a pathological case.
- ❖ On realistic randomized applications, DotMix is at most 21% more costly than Mersenne twister.

<i>Application</i>	$T_1(\text{DotMix}) / T_1(\text{mt})$	$T_{12}(\text{DotMix}) / T_{12}(\text{mt})$
rfib	2.33	2.25
pi	1.21	1.13
maxIndSet	1.14	1.08
sampleSort	1.00	1.00
DiscreteHedging	1.03	1.03

Summary of DPRNG

- ❖ DotMix provides an efficient library for generating pseudorandom numbers deterministically in parallel.
- ❖ DotMix exposes the same API as a serial RNG, thereby requiring minimal code modifications to use.
- ❖ DotMix enables randomized dynamic multithreaded programs to exhibit deterministic, repeatable execution, which programmers can use to investigate program behavior in a principled manner that is familiar from serial programming.

Impact of Pedigrees and DotMix

- ❖ Pedigrees have been incorporated into the Intel Cilk Plus runtime and both the Intel and GNU C/C++ compilers.
- ❖ Intel adopted the DotMix algorithm to replace their deterministic parallel random-number generators. Spend time here.
- ❖ DotMix directly inspired the design of `java.util.SplittableRandom` in JDK8, written by Guy Steele, Doug Lea, and Christine Flood [SLF14].



Steele



Lea



Flood

Outline

- ❖ DPRNG
- ❖ Life after Moore's Law



Emer



Kuszmaul



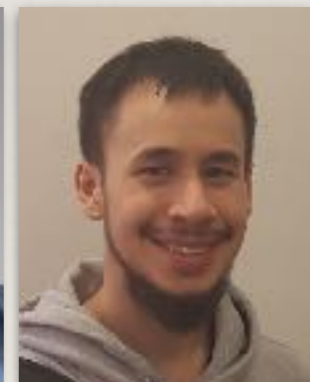
Lampson



Leiserson



Sanchez

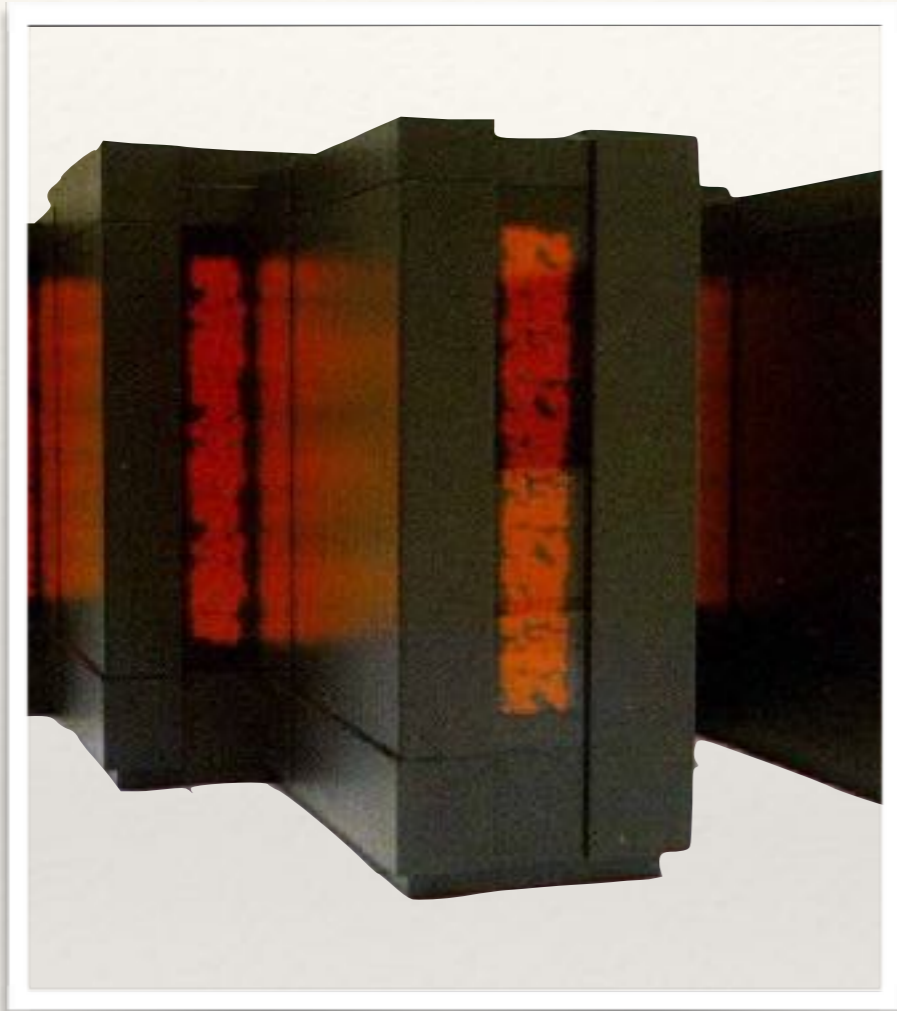


Schardl



Thompson

The Effect of Moore's Law



- Connection Machine CM-5
- 60 GFLOPS on LINPACK
 - \$47 million in 1993



- Apple 13" MacBook Pro
- 70 GFLOPS on LINPACK
 - \$1500 in 2015

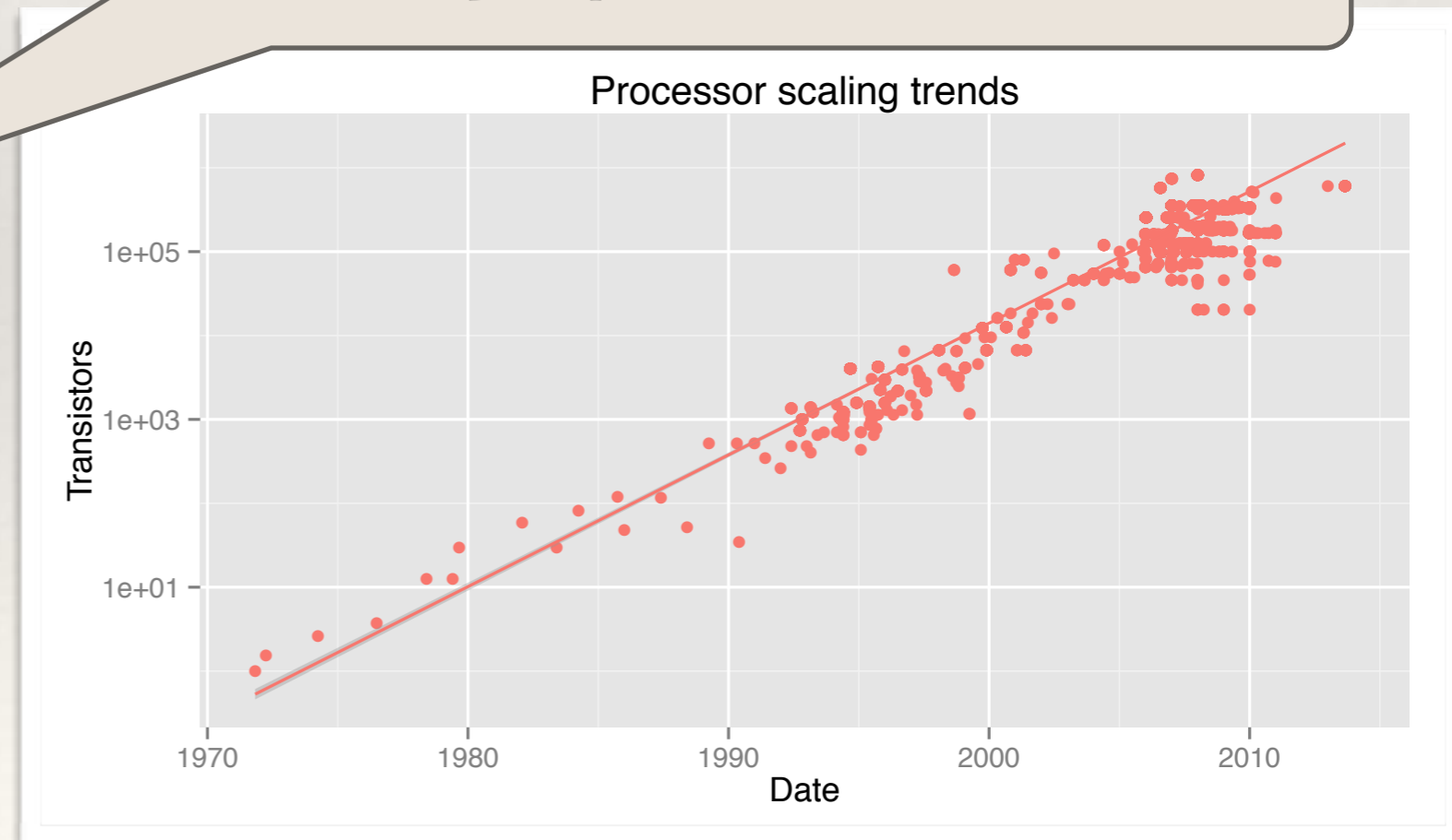
Moore's Law [M65, M75]

“The complexity for minimum component costs has increased at a rate of roughly a factor of two per year.” [M65]



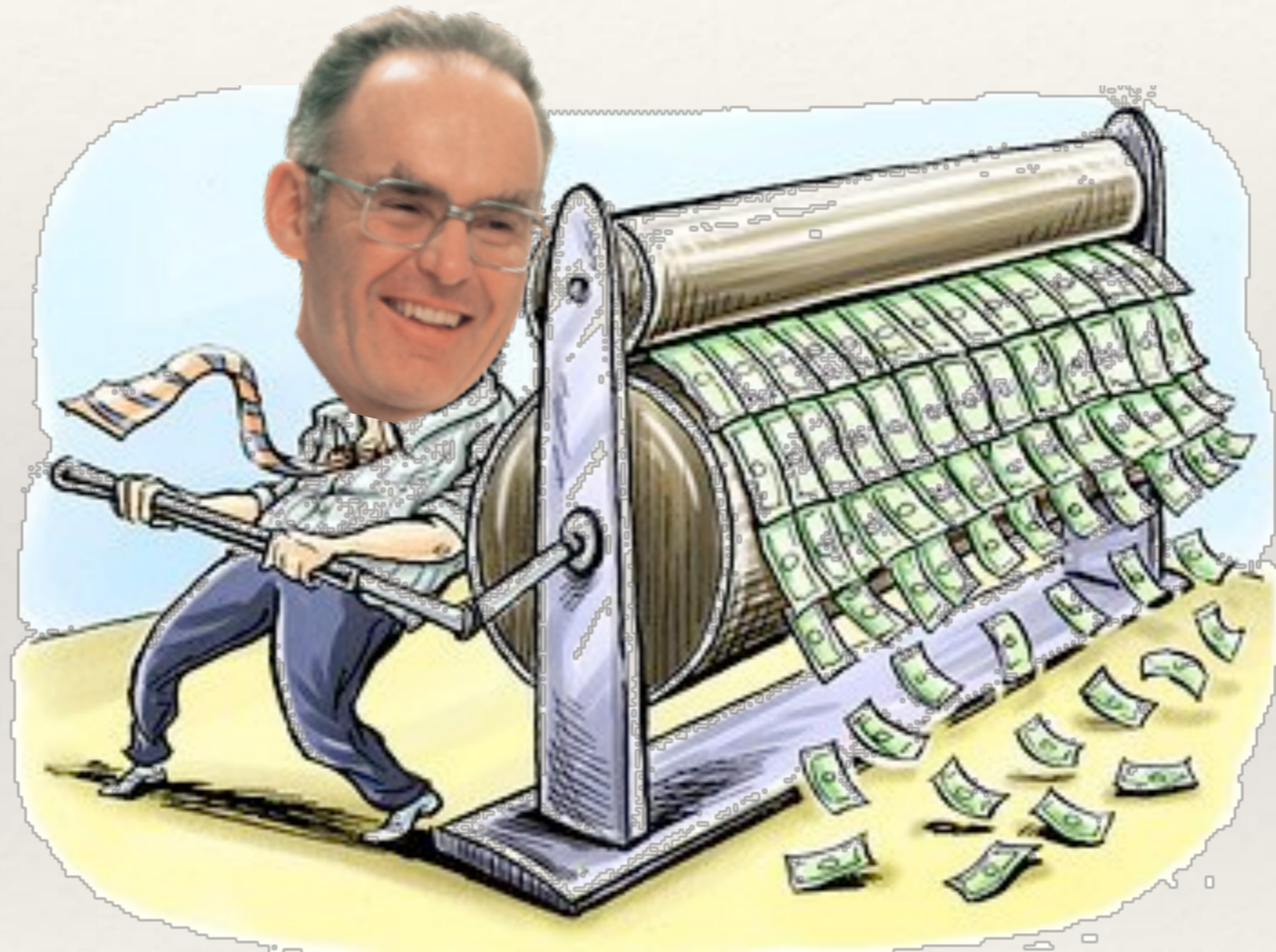
Moore

“The new slope might approximate a doubling every two years, rather than every year, by the end of the decade.” [M75]



50-Year Impact of Moore's Law

More transistors means cheaper computing.



Moore's Law is a printing press for processor cycles.

50-Year Impact of Moore's Law

More transistors mean cheaper computing.




Moore's Law is a printing press for processor cycles.

Outline

- ❖ DPRNG
- ❖ Life after Moore's Law
 - ❖ Why do we think it's ending?
 - ❖ What happens next?

Moore's Law Will End

Robert Colwell, chief architect for the Intel Pentium Pro, Pentium II, Pentium III, and Pentium 4 processors, and former director of the Microsystems Technology Office at DARPA, said in 2013:



“For planning horizons, I pick 2020 as the earliest date where I think we could call [Moore's Law] dead.”

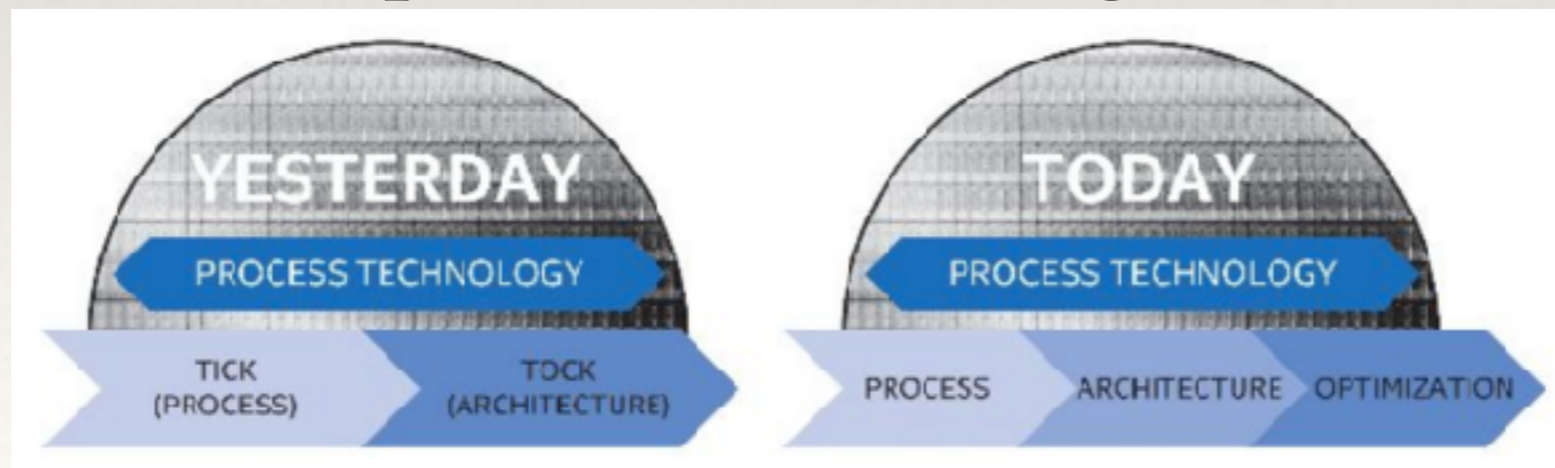
“You can talk me into 2022.”

Colwell

The End is Nigh

The semiconductor industry is giving up on Moore's Law.

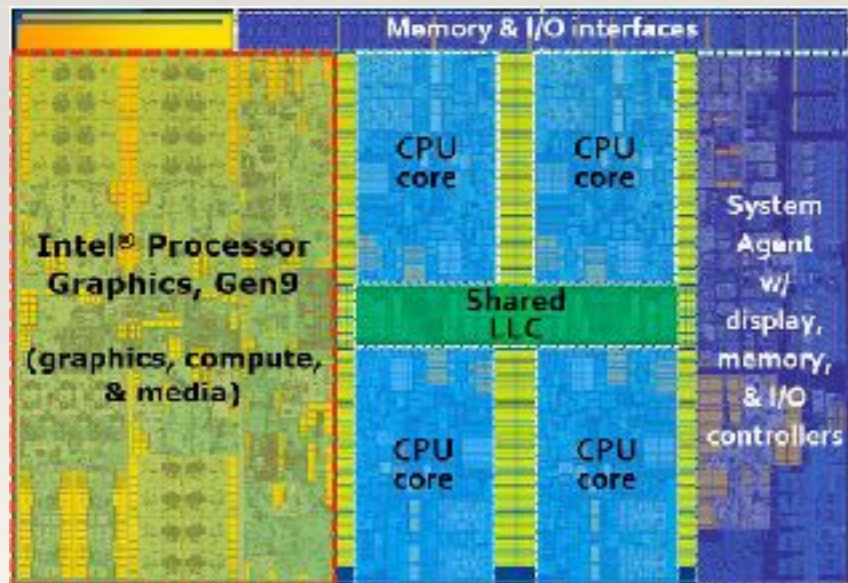
- ❖ ITRS 2.0, 2015: "By 2020-25...it will become practically impossible to reduce device dimensions any further."
- ❖ Intel's 10-K SEC filing, 2016: "We expect to lengthen the amount of time we will utilize our 14nm and our next-generation 10nm process technologies."



Why Must It End Now?

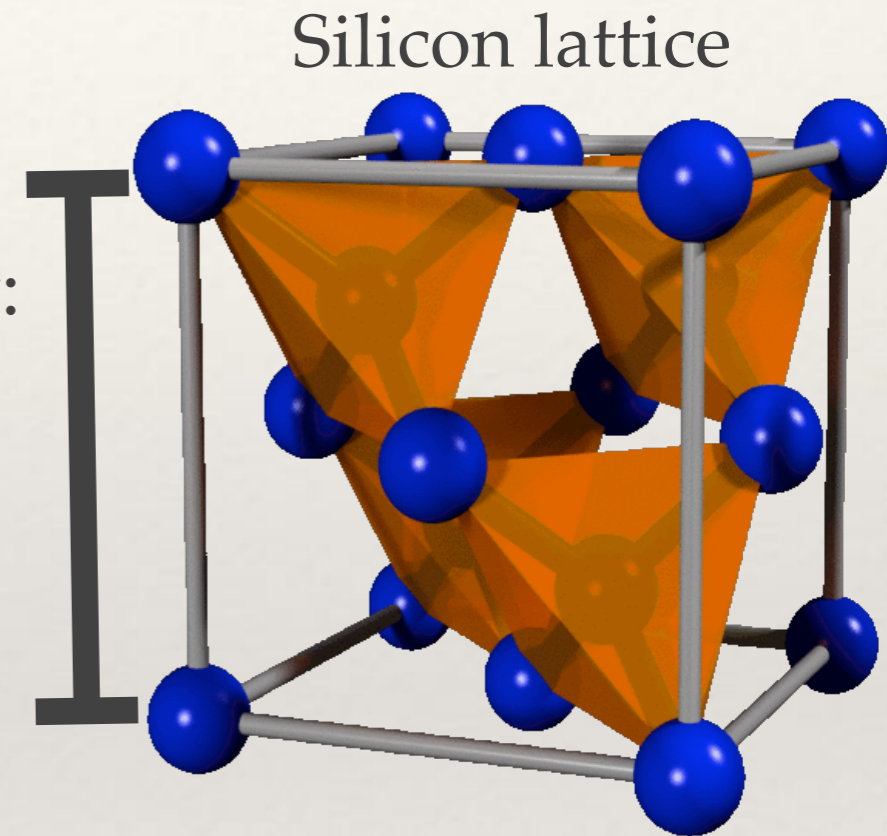
We're running out of atoms.

Intel Skylake processor, 2015



14 nanometer transistors

Silicon lattice constant:
0.543 nanometers
(5.43 angstroms)



Transistors are now
25 atoms wide.

Outline

- ❖ DPRNG
- ❖ Life after Moore's Law
 - ❖ Why do we think it's ending?
 - ❖ What happens next?

What Happens Next?

Can rapid growth in computer performance continue after Moore's Law ends?

Yes, with caveats.



All Is Not Lost



There are “replacement technologies” that can provide many applications with rapid growth in performance after the demise of Moore’s Law.

❖ But semiconductor physics and silicon-fabrication technologies **won’t help much.**

❖ **Computer science itself must provide the impetus with performance-engineering techniques to get from writing faster code?** architecture, programming languages, compilers, systems, algorithms, applications, and tools.

❖ Unlike the broad-based nature of Moore’s Law, these CS technologies will drive up performance **unevenly** in an **opportunistic** fashion.

4k-by-4k Matrix Multiplication

Quiz: How long does the following code take to execute?

Python code

```
for i in xrange(n):  
    for j in xrange(n):  
        for k in xrange(n):  
            C[i][j] += A[i][k] * B[k][j]
```

- A. 7 milliseconds
- B. 7 seconds
- C. 7 minutes
- D. 7 hours
- E. 7 days

Machine: Amazon Web Services
c4.8xlarge spot instance.

- ❖ Dual socket Intel Xeon E5-2666 v3 (Haswell)
- ❖ 18 cores, 2.9 GHz, 60 GiB DRAM

Recall that this computation performs $2 \times 4096^3 = 128$ billion floating-point operations.

4k-by-4k Matrix Multiplication

Quiz: How long does the following code take to execute?

Python code

```
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
```

- A. 7 milliseconds
- B. 7 seconds
- C. 7 minutes
- D. 7 hours
- E. 7 days

Machine: Amazon Web Services
c4.8xlarge spot instance.

- ❖ Dual socket Intel Xeon E5-2666 v3 (Haswell)
- ❖ 18 cores, 2.9 GHz, 60 GiB DRAM

Recall that this computation performs $2 \times 4096^3 = 128$ billion floating-point operations.

4k-by-4k Matrix Multiplication

Version	Implementation	Running time (s)	GFLOPS	Absolute speedup	Relative speedup	Fraction of peak
1	Python	25,000	0.000	1	—	0.00%
2	Java	2,500	0.000	11	10.8	0.01%
3	C	542.67	0.253	47	4.4	0.03%
4	Parallel divide-and-conquer C	60.00	1.000	366	7.8	0.24%
5	Parallel divide-and-conquer C with AVX	6.00	10.000	3660	78	2.4%
6	Parallel divide-and-conquer C with AVX and Strassen	2.40	25.000	10500	233	7.2%
7	+ AVX intrinsics	0.41	337.812	62,806	2.7	40.45%
8	Strassen	0.38	361.177	67,150	1.1	43.24%

Spend time here.

The parallel divide-and-conquer C-implementation of Strassen that uses AVX intrinsics has 40 times more lines than the Python version.

Comparable to 32 years of Moore's Law improvements!

Problem: Fast Code vs. Simple Code

Simple code is slow.

Fast code is complicated.

Let's make a world with far less distance
between fast code and simple code.

How? Remedy the *ad hoc* nature of software
performance engineering by developing a
science of fast code.

Most important slide! Spend time here!

My Contributions to a Science of Fast Code

<i>Artifact</i>	<i>Simple programming models</i>	<i>Theories of performance</i>	<i>Efficient diagnostic tools</i>
PBFS		●	
DPRNG	●		
Cilk-P	●	●	
Prism		●	
Color		●	
Cilkprof			●
Rader			●
Tapir		●	
CSI	●		●

There's still a lot to do.

● The artifact primarily supports the technology enabling a science of fast code. 62

Developing a Science of Fast Code

Four action items stand out as key to developing a science of fast code:

- ❖ We need *simple programming models* one can reason about because they obey mathematical properties such as **determinism** and **composability**. (E.g., DPRNG, Commutative Building Blocks [Shun15])
- ❖ We need *theories of performance* that are borne out in practice. (E.g., Work-span analysis, weighted dag model [MA16])
- ❖ We need *efficient diagnostic tools* for correctness and performance whose efficacy is mathematically grounded. (E.g., Cilkprof, C-RACER [UAFL16])
- ❖ We need to *educate* programmers in these software performance engineering technologies and in thinking critically about software performance.

Questions?
