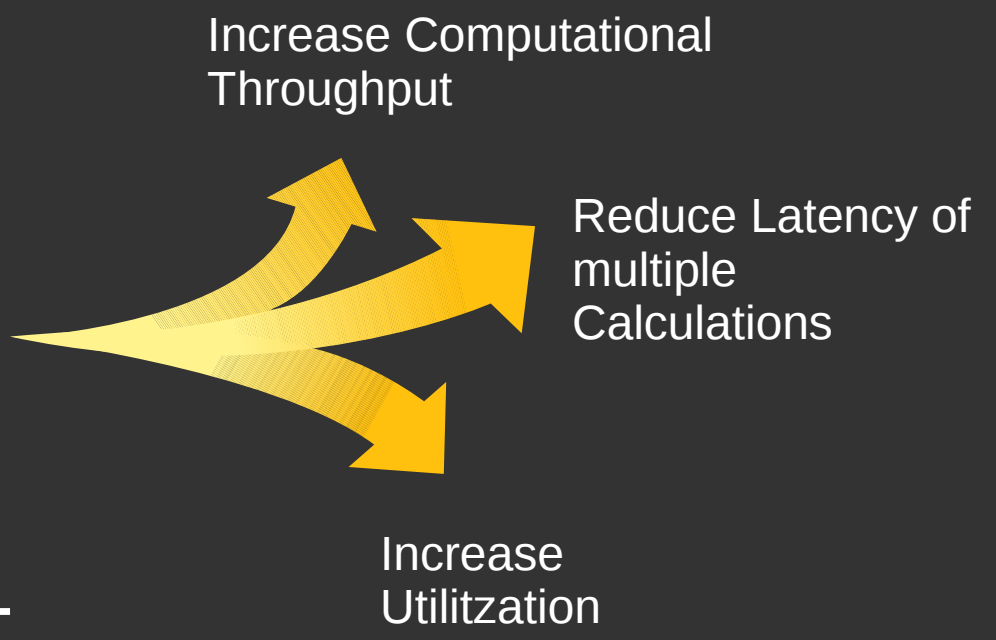
What You Can Mess Up When Writing Parallel Code

Ulrich Drepper

para//el 2016

When To Use Parallelism

- Generally not voluntarily
- As required to

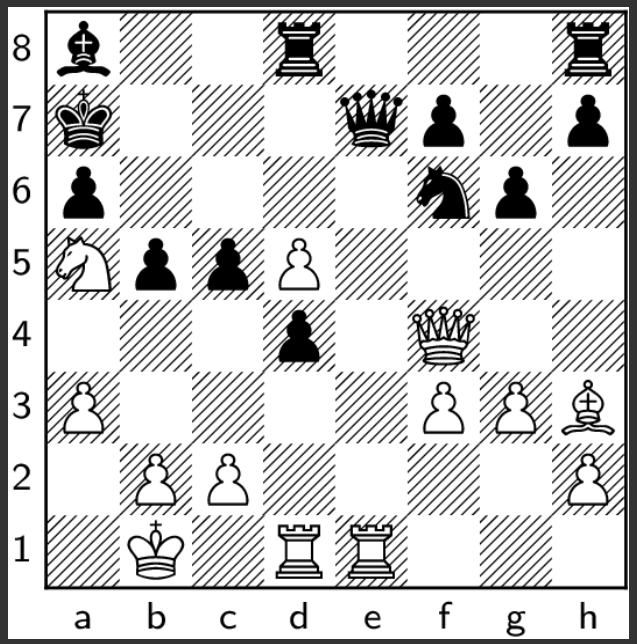


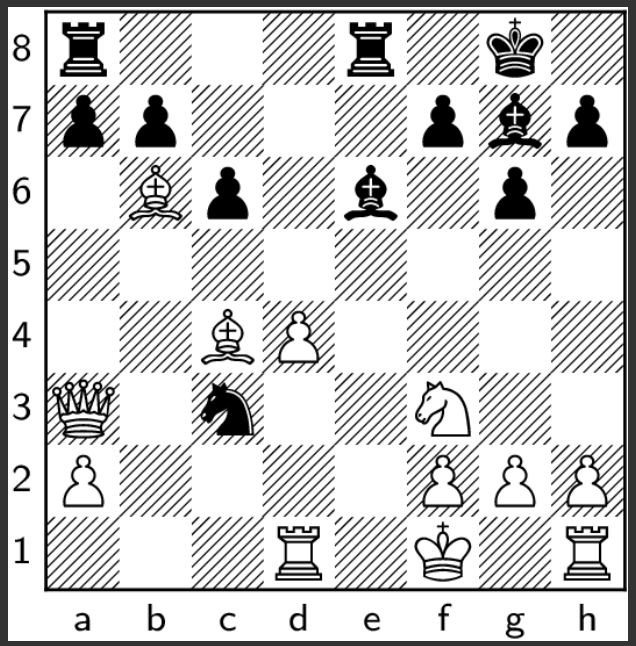
- Limitation: Cost
  - Hardware
  - Developer Time

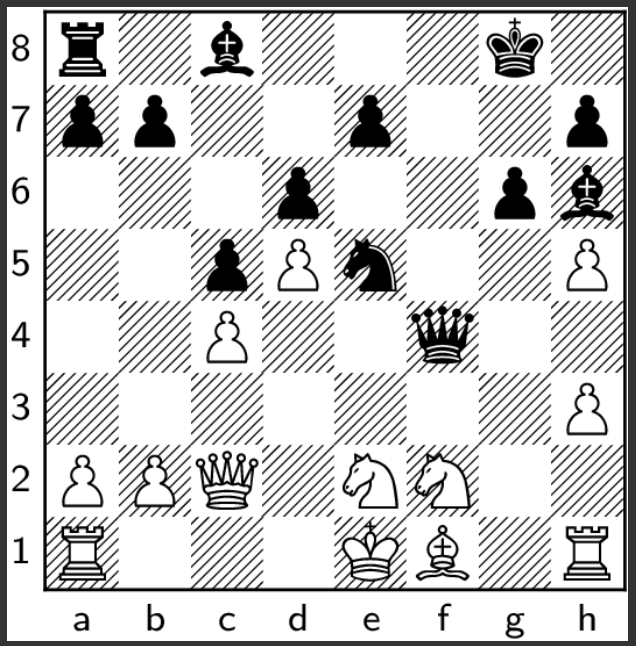
Handling Parallelism …

"Elisabeth Bykova" by http://www.chessgames.com/perl/chessplayer?pid=72242. Licensed under Fair use via Wikipedia - https://en.wikipedia.org/wiki/File:Elisabeth_Bykova.jpg#/media/File:Elisabeth_Bykova.jpg

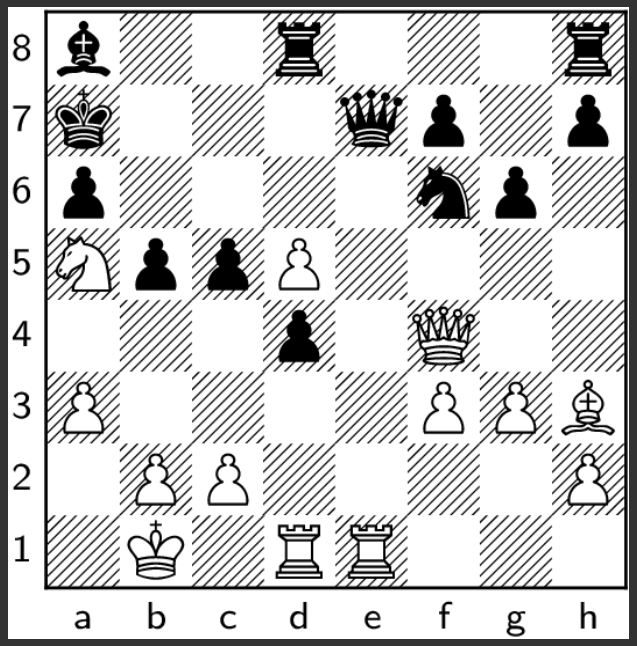
Three Different Sequences

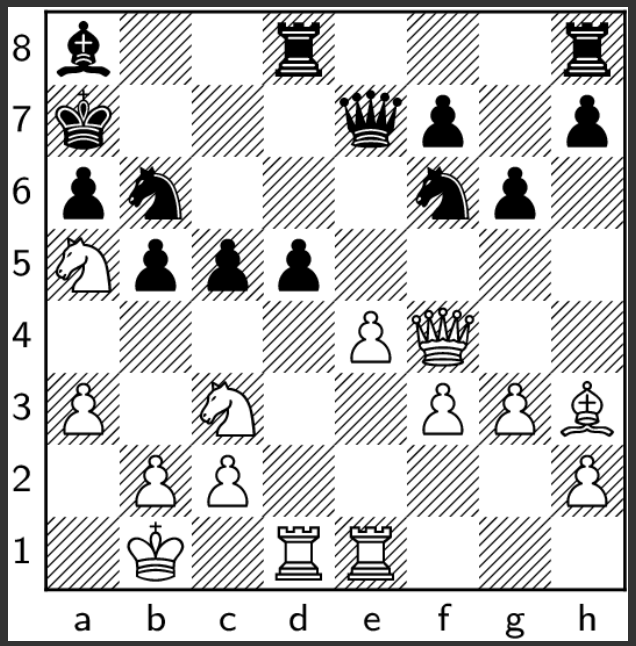


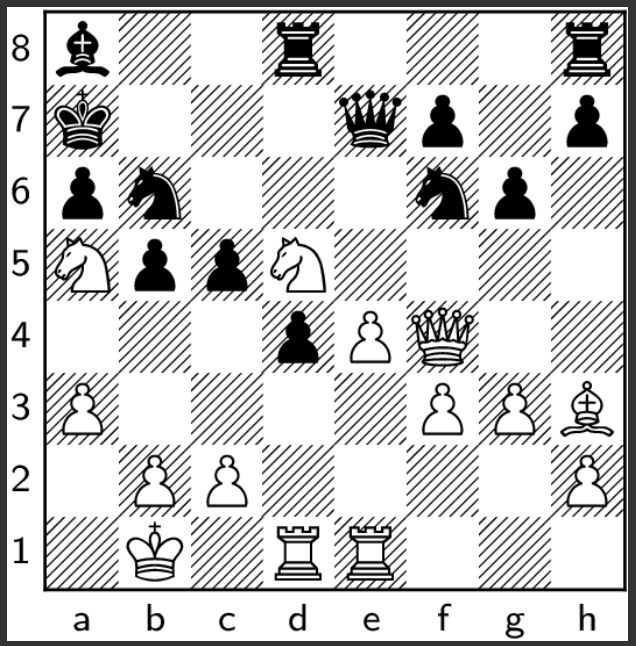




Similarity Is Useful







Drake Equation

$$N = R_* \cdot f_p \cdot n_e \cdot f_\ell \cdot f_i \cdot f_c \cdot L$$

Performance Equation

$$\text{Efficiency} = f_{\text{Cache}} \cdot f_{\text{NUMA}} \cdot f_{\text{link}} \cdot f_{\text{sync}} \cdots$$

Worse: factors are not independent

f_{NUMA} might depend on f_{link}

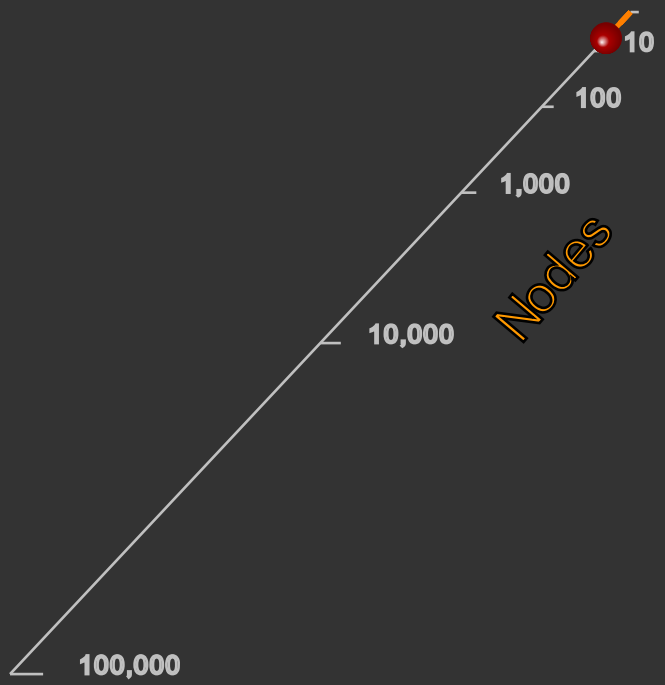
f_{Cache} might depend on f_{NUMA}

etc

What You Need to Discover

- What are the factors?
- Benefits of changes
  - Do not spend efforts with little benefits
- What are the interactions?
  - Also influences selection of changes

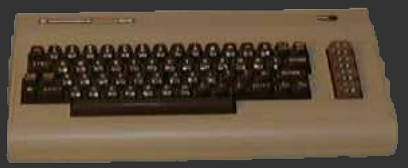
Always Needed More

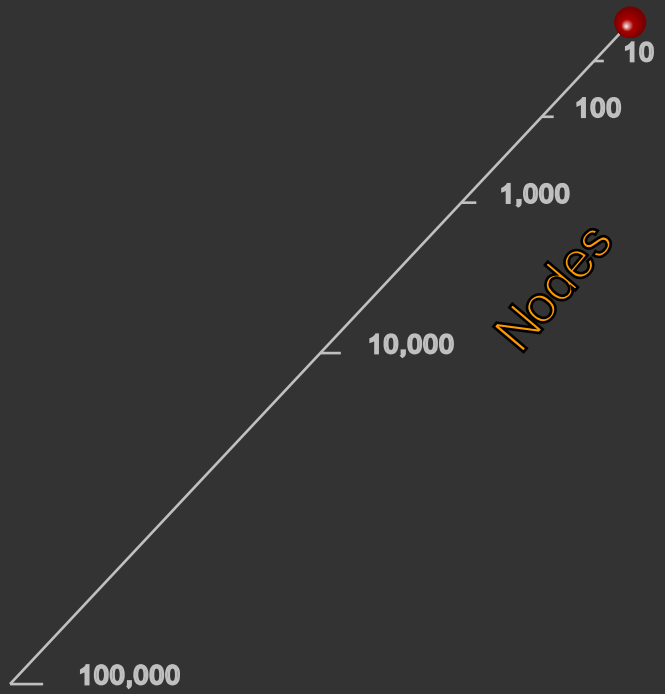


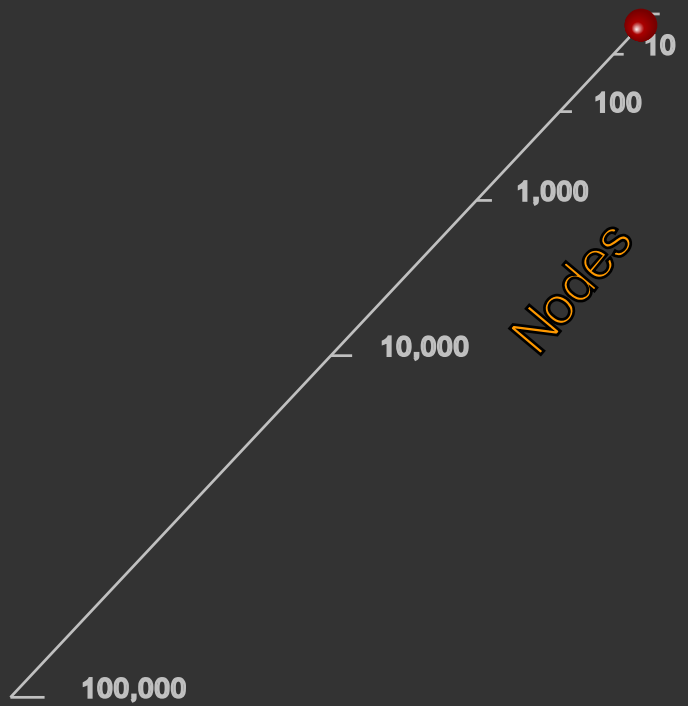
I think there is a world market for maybe five computers.

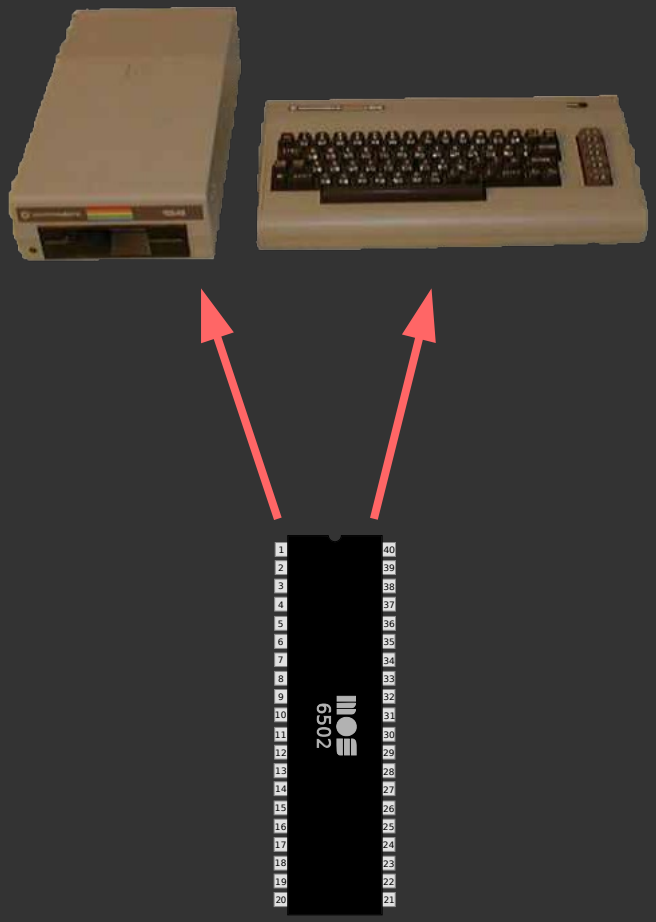
Thomas Watson, chairman of IBM, 1943.

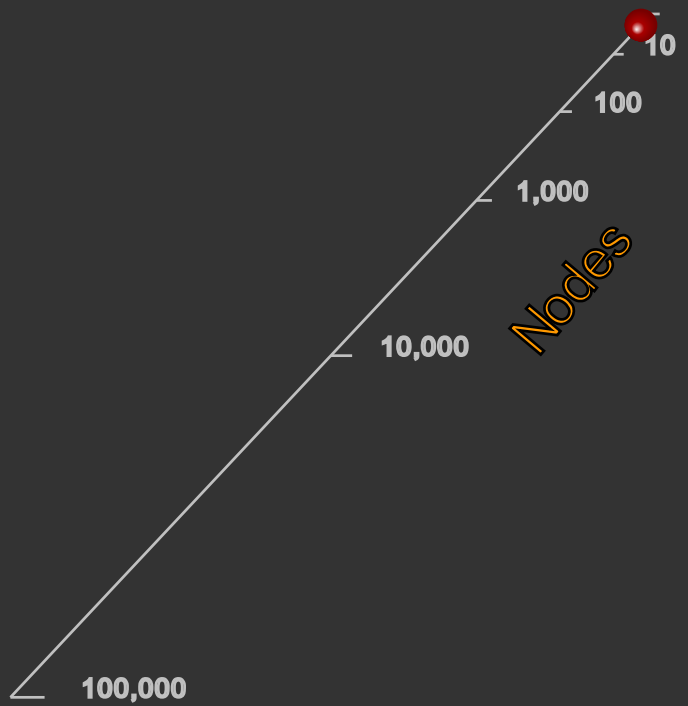
Remember?



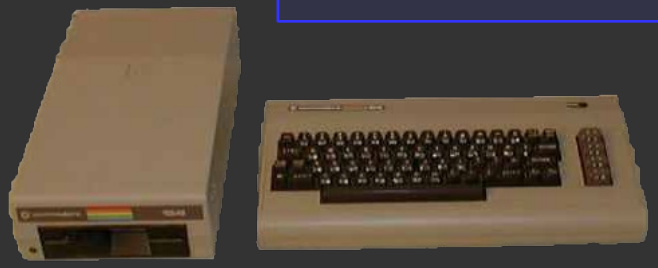


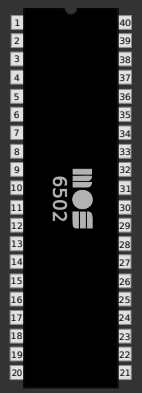

10
100
1,000
10,000
100,000
Nodes


MOS
6502

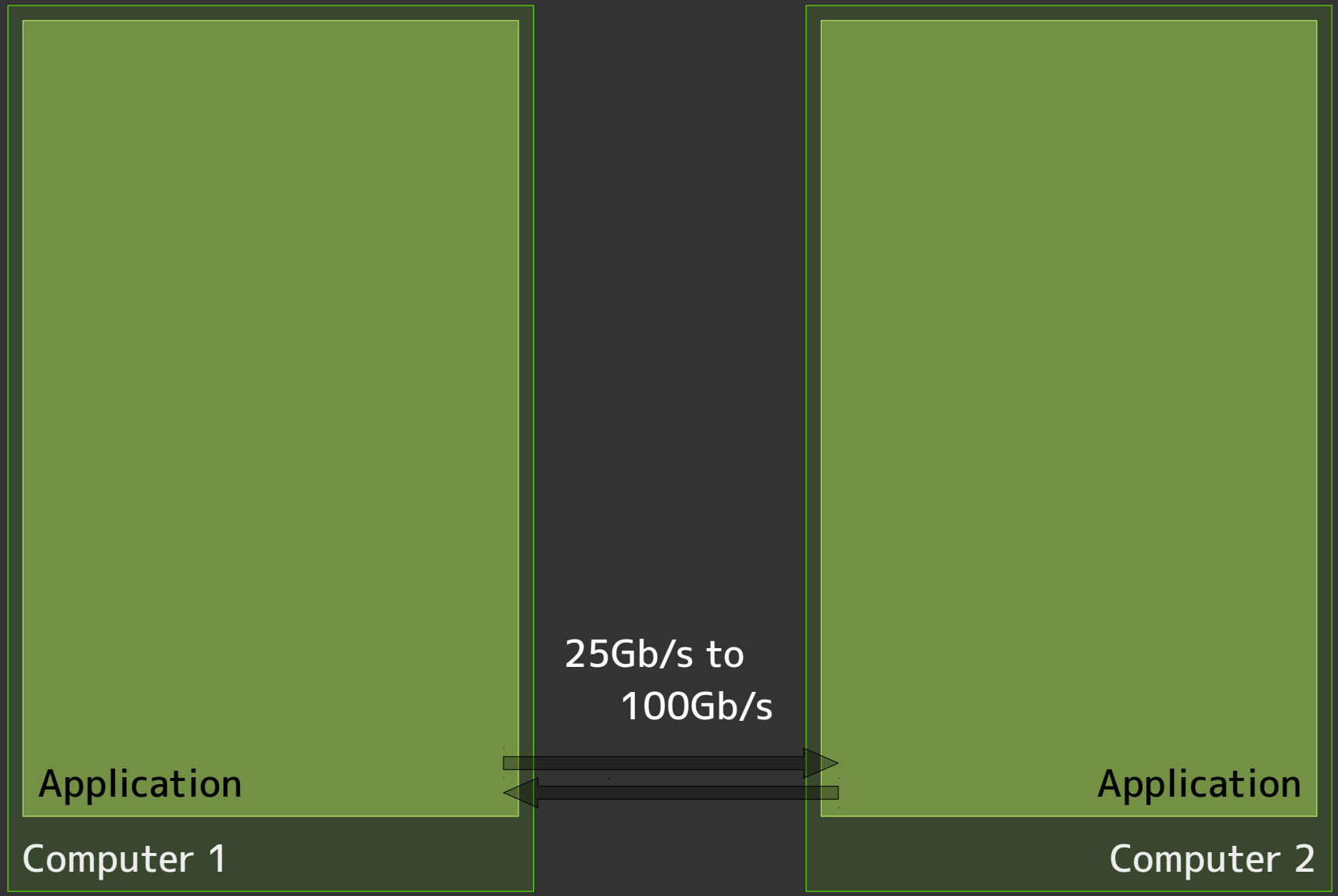


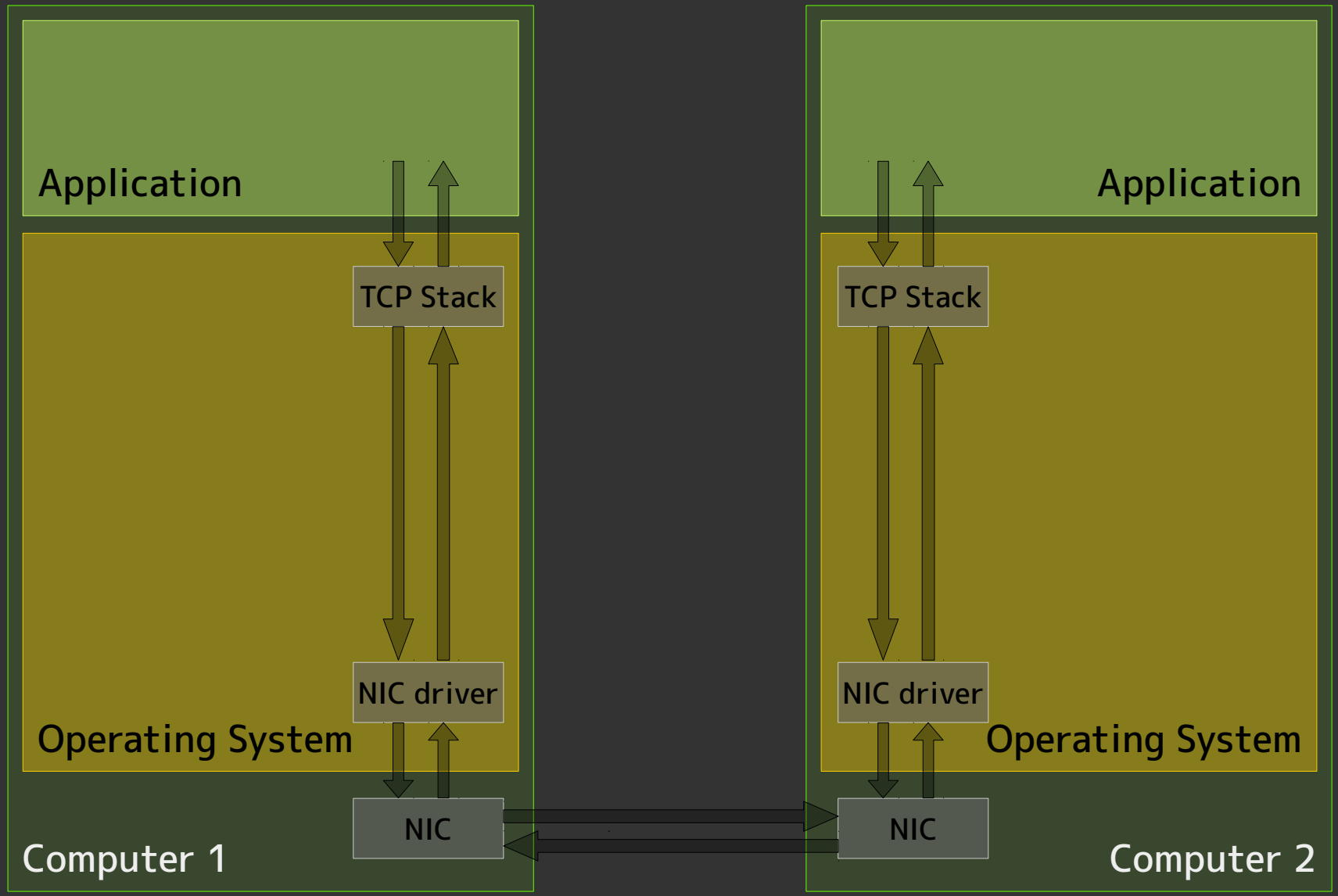
2400 b/s – 32kb/s

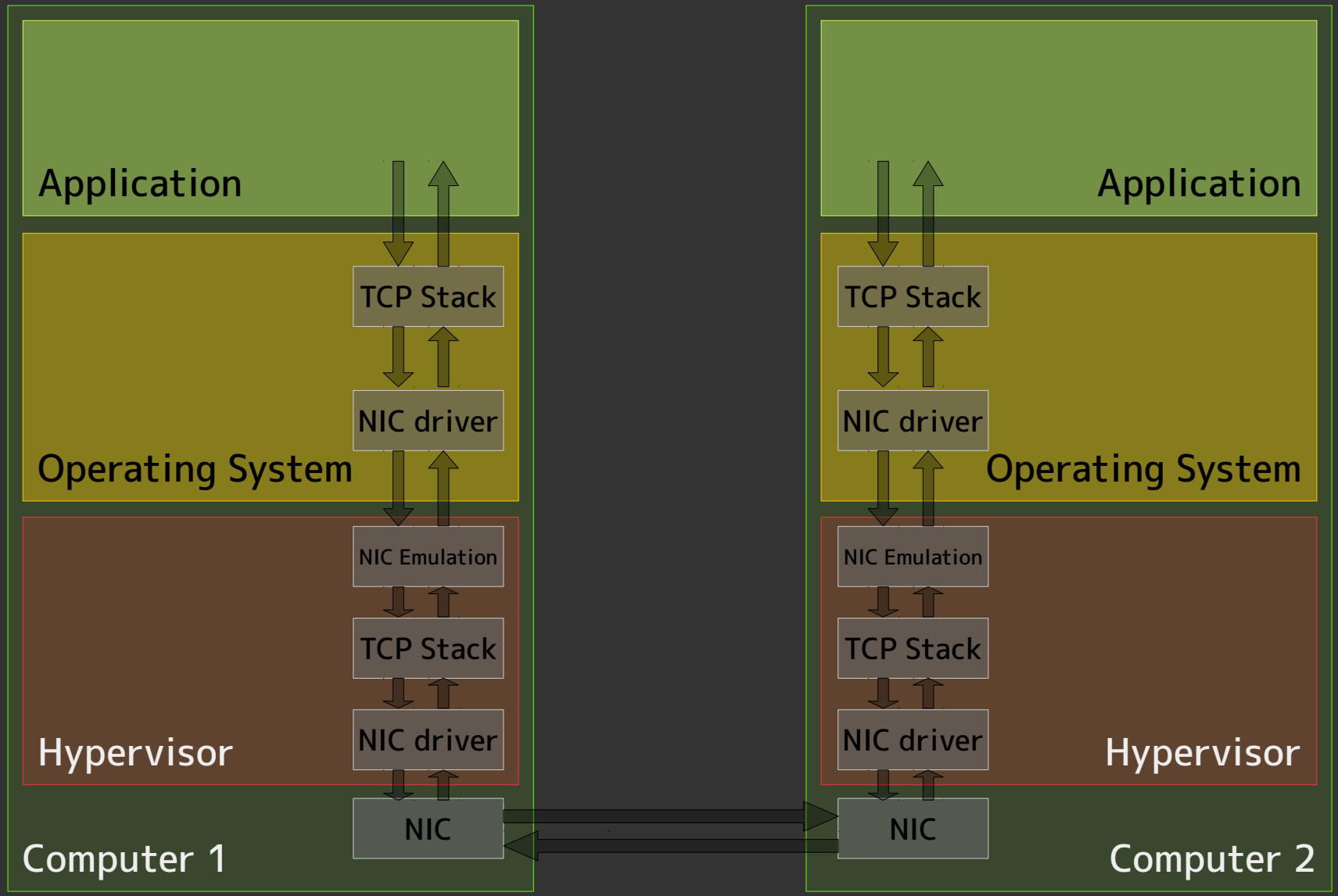


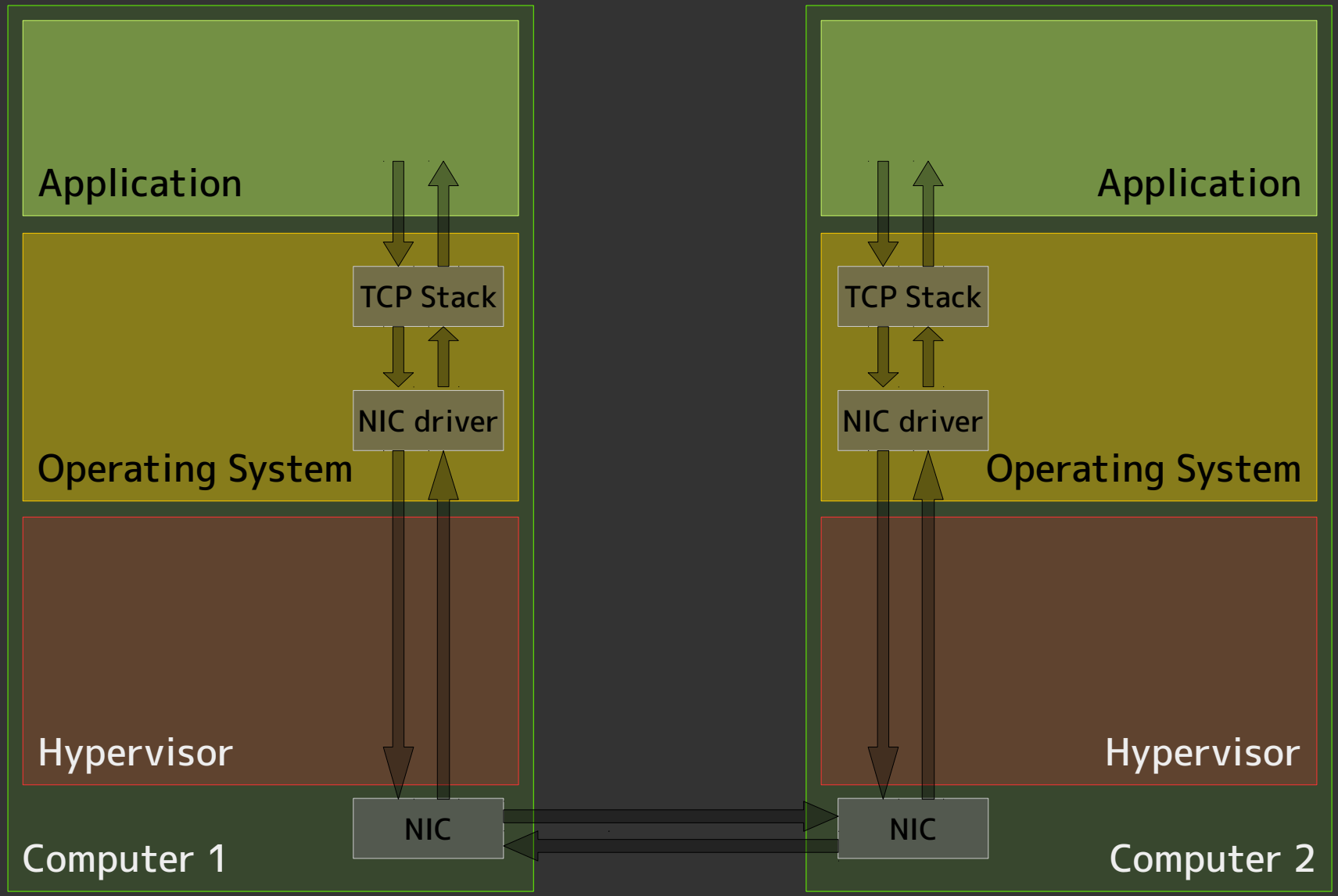


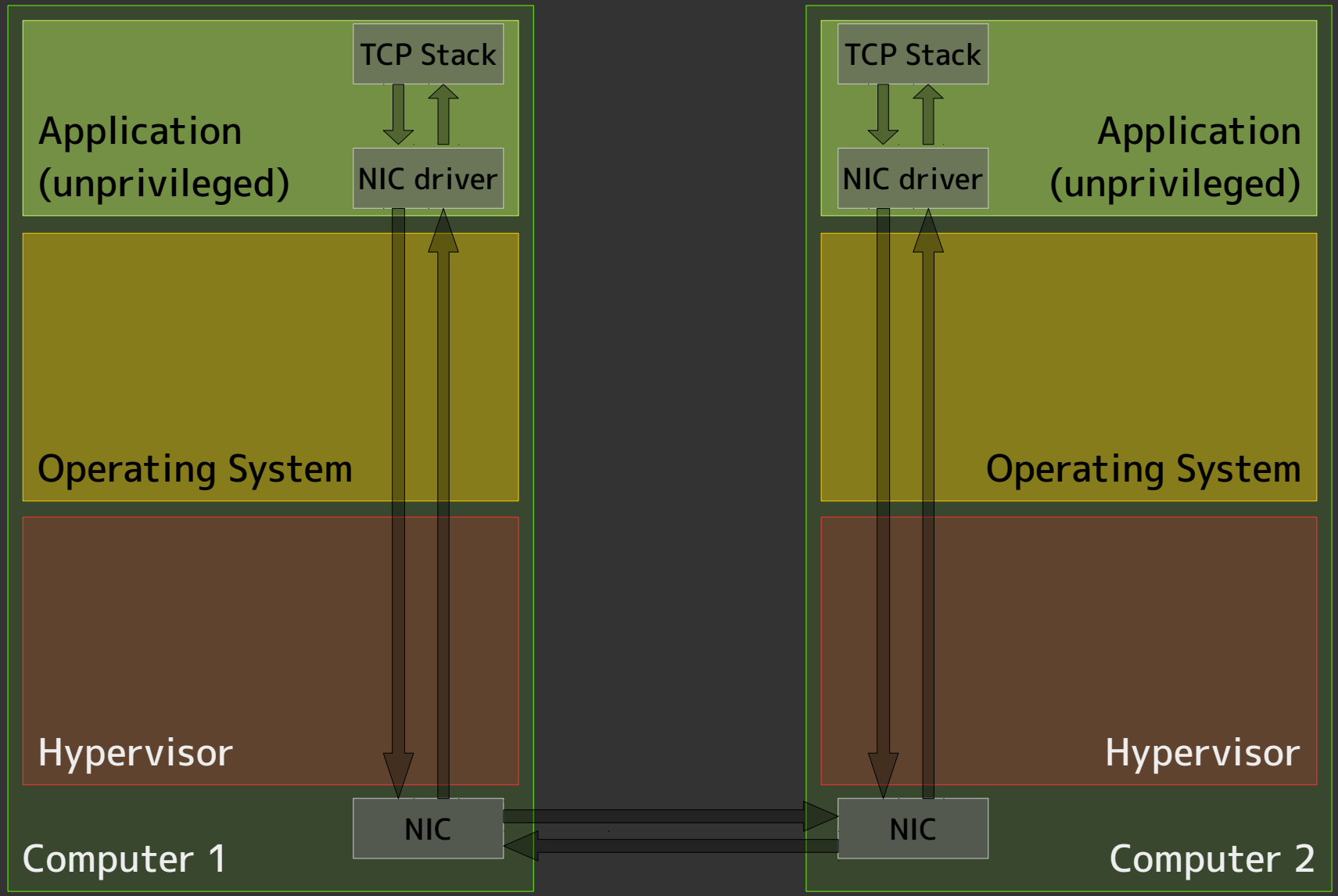
Better Today

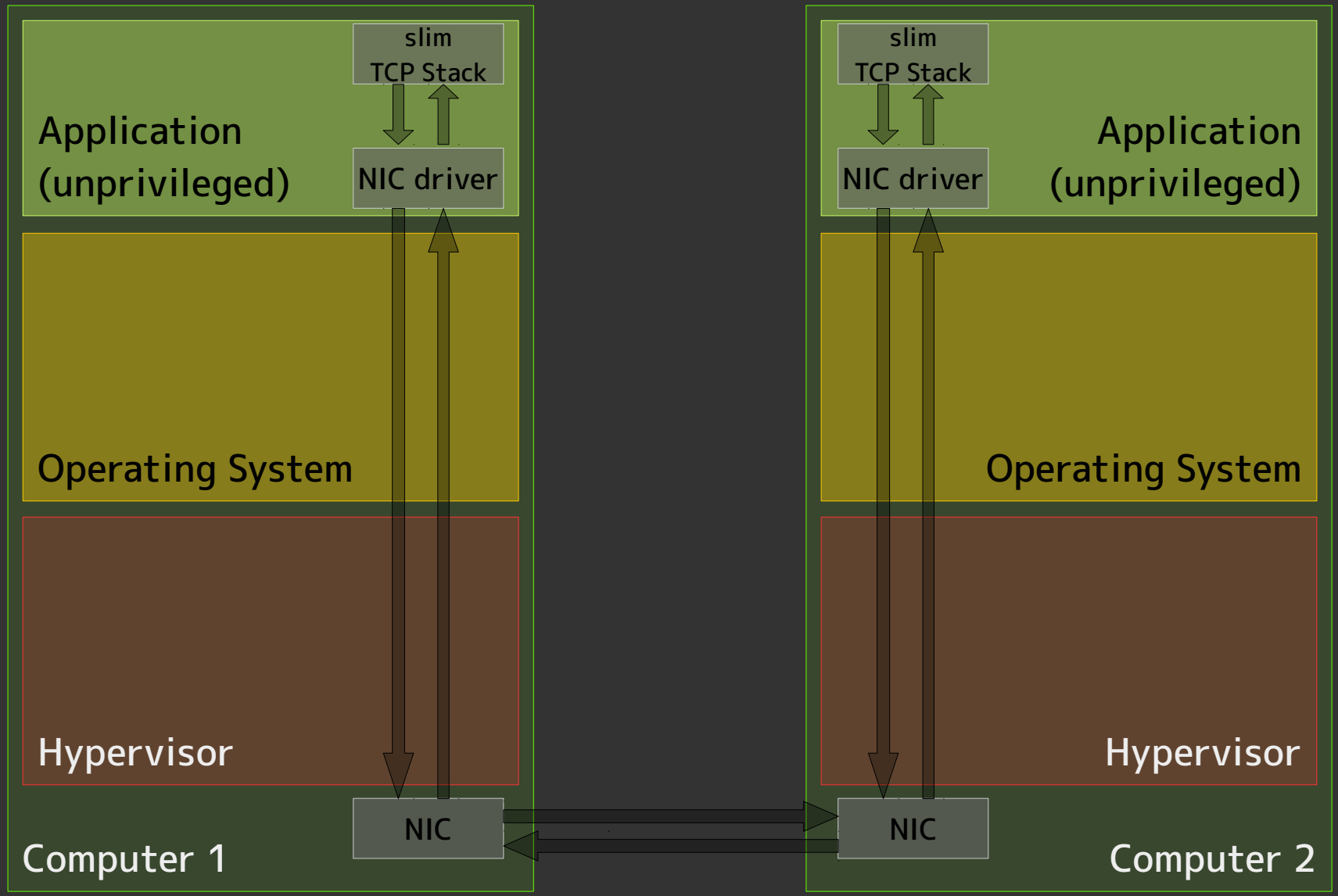


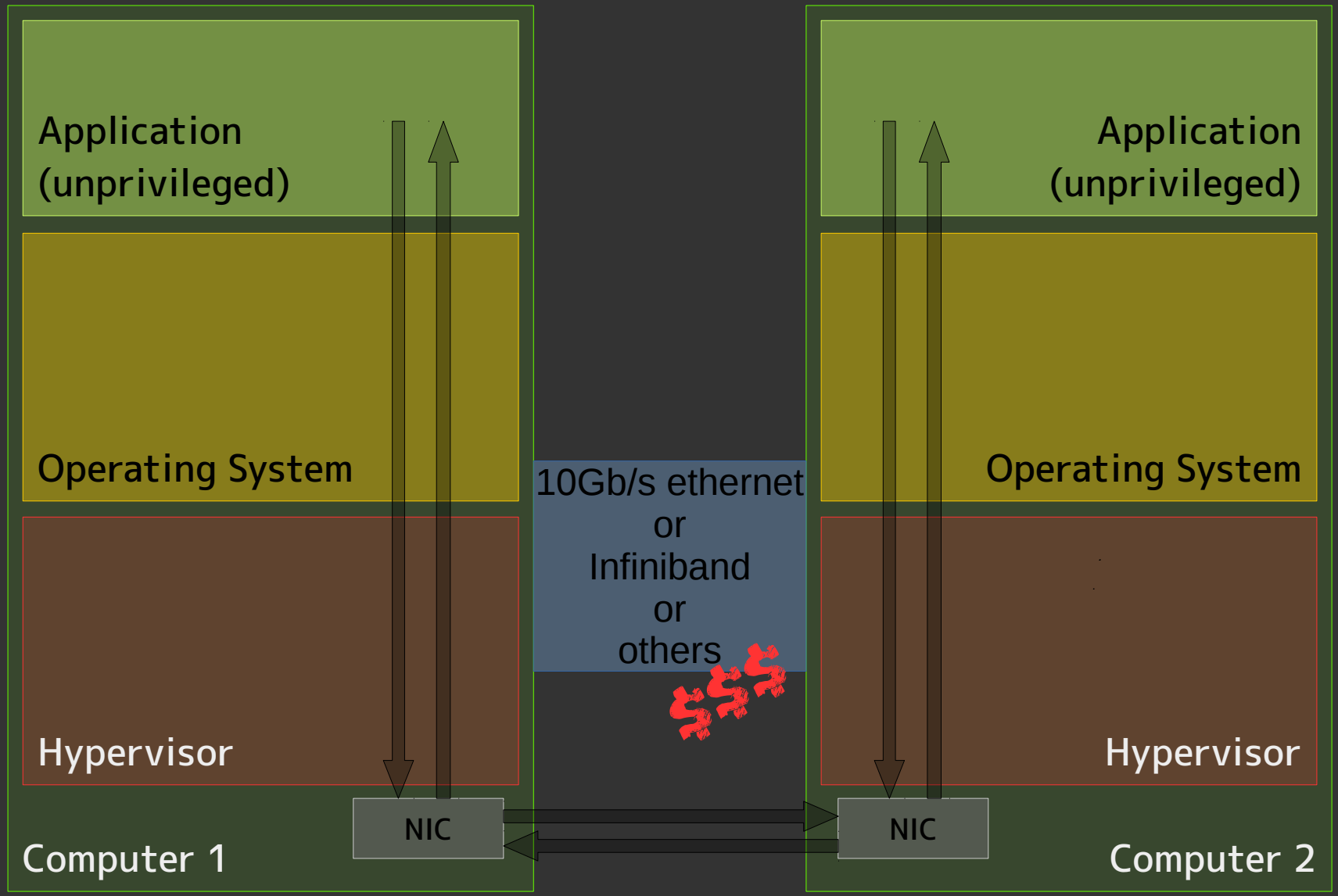

Application
TCP Stack
NIC driver
Operating System
NIC
Computer 1
Application
TCP Stack
NIC driver
Operating System
NIC
Computer 2


Application
TCP Stack
NIC driver
Operating System
NIC Emulation
TCP Stack
NIC driver
Hypervisor
NIC
Computer 1
Application
TCP Stack
NIC driver
Operating System
NIC Emulation
TCP Stack
NIC driver
Hypervisor
NIC
Computer 2

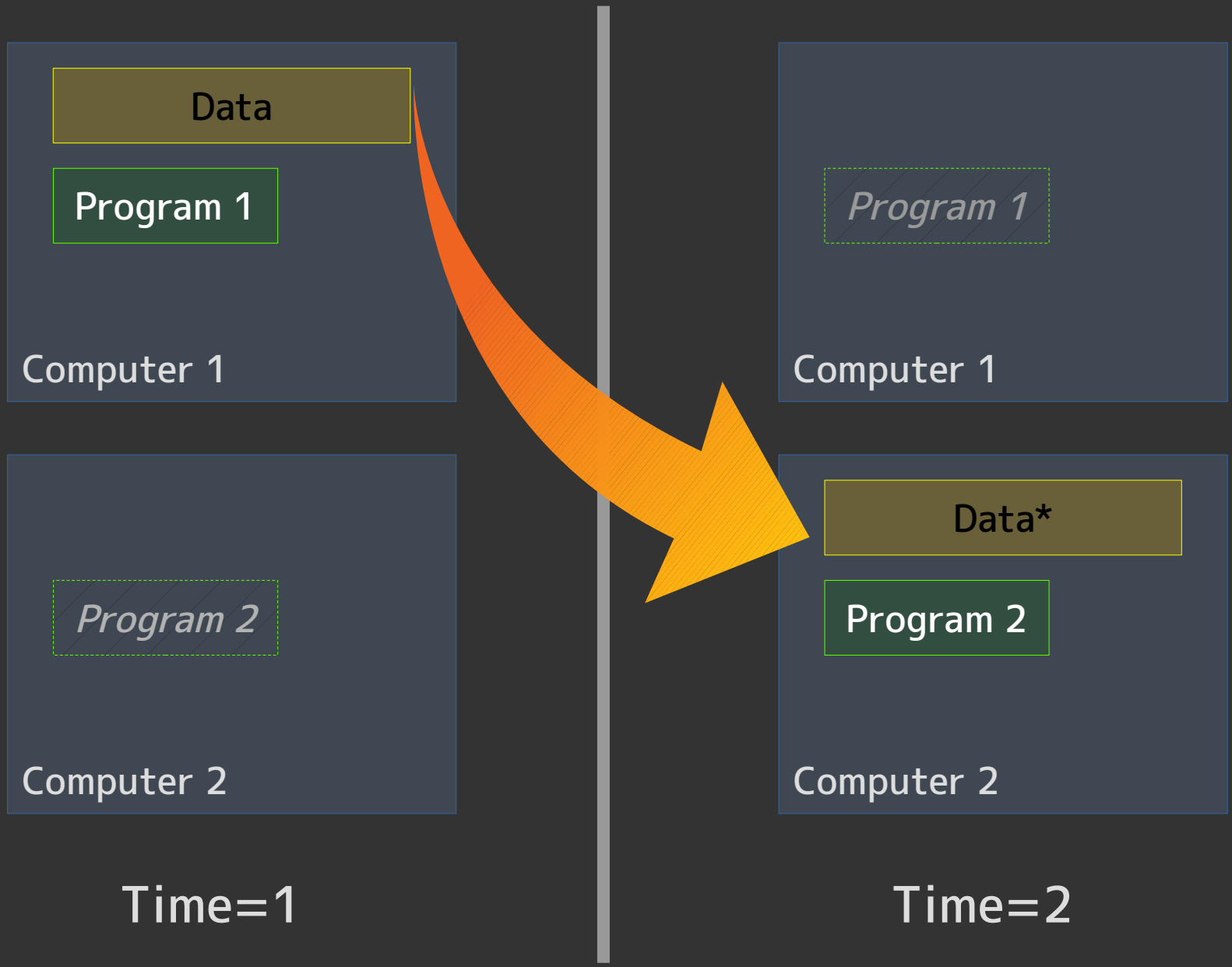

Application
TCP Stack
NIC driver
Operating System
Hypervisor
NIC
Computer 1
Application
TCP Stack
NIC driver
Operating System
Hypervisor
NIC
Computer 2


TCP Stack
Application
(unprivileged)
NIC driver
Operating System
Hypervisor
NIC
Computer 1
TCP Stack
NIC driver
Application
(unprivileged)
Operating System
Hypervisor
NIC
Computer 2

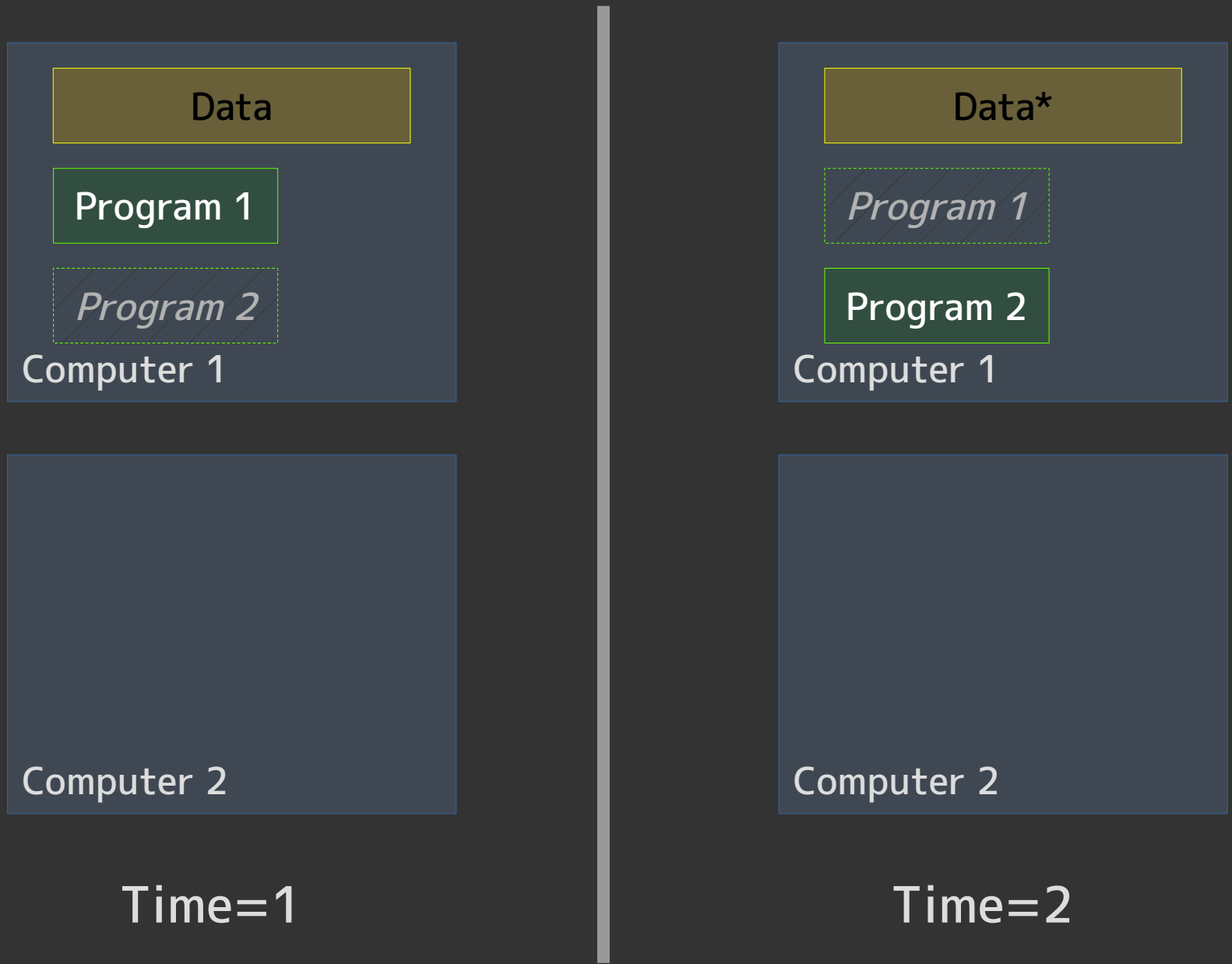

slim
TCP Stack
Application
(unprivileged)
NIC driver
Operating System
Hypervisor
NIC
Computer 1
slim
TCP Stack
NIC driver
Application
(unprivileged)
Operating System
Hypervisor
NIC
Computer 2


Application
(unprivileged)
Operating System
Hypervisor
NIC
Computer 1
10Gb/s ethernet
or
Infiniband
or
others
Application
(unprivileged)
Operating System
Hypervisor
NIC
Computer 2

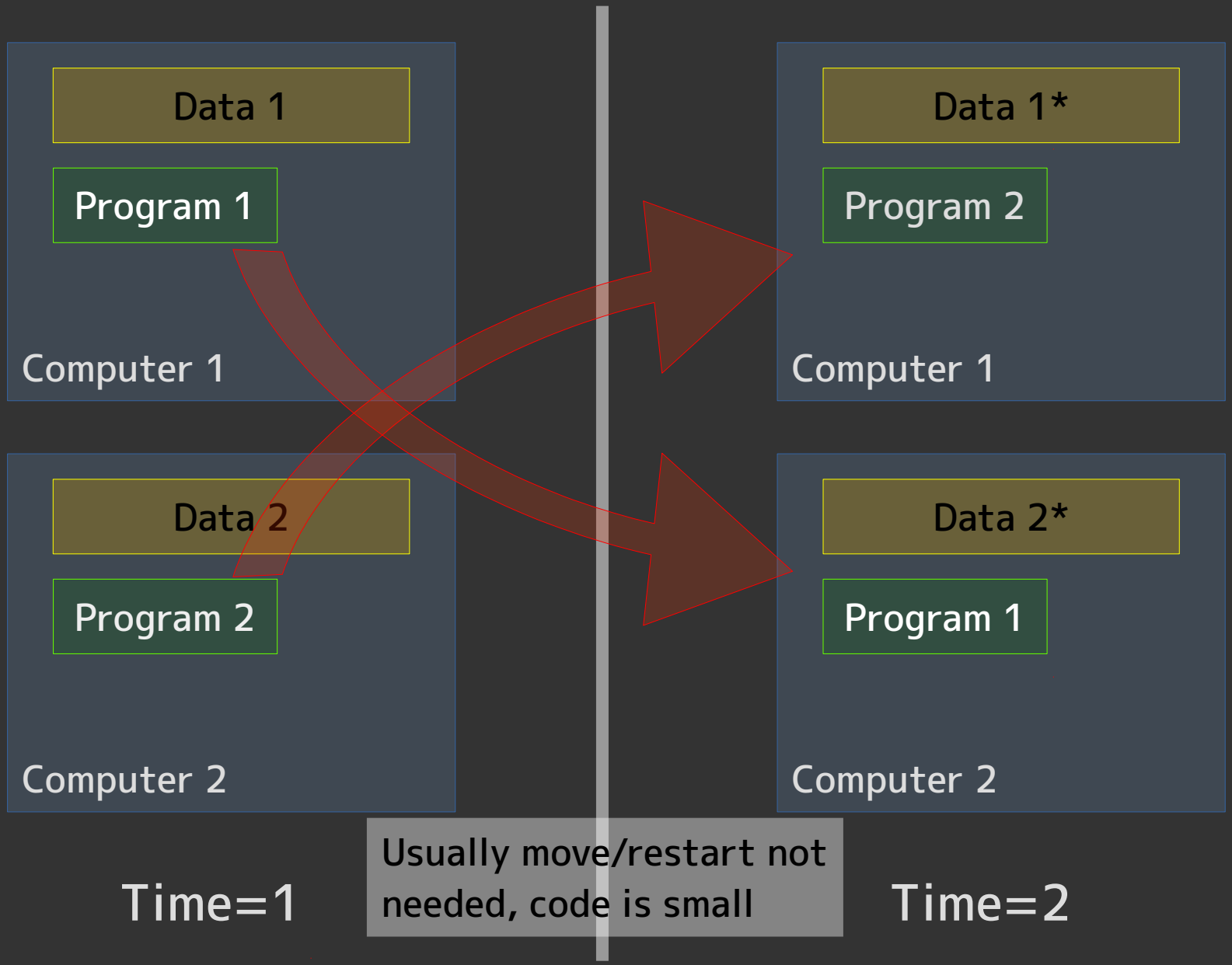
Moving Data


Data
Program 1
Computer 1
Program 2
Computer 2
Program 1
Computer 1
Data*
Program 2
Computer 2
Time=1
Time=2

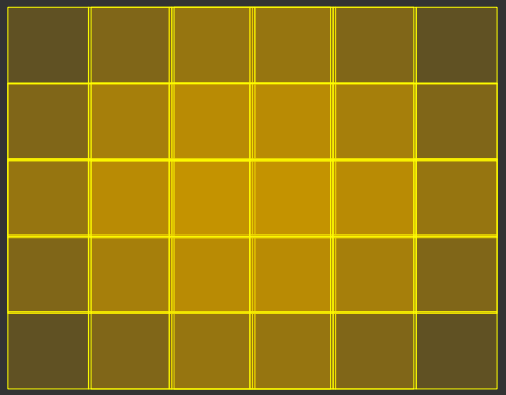
Avoiding to Move



Moving Code

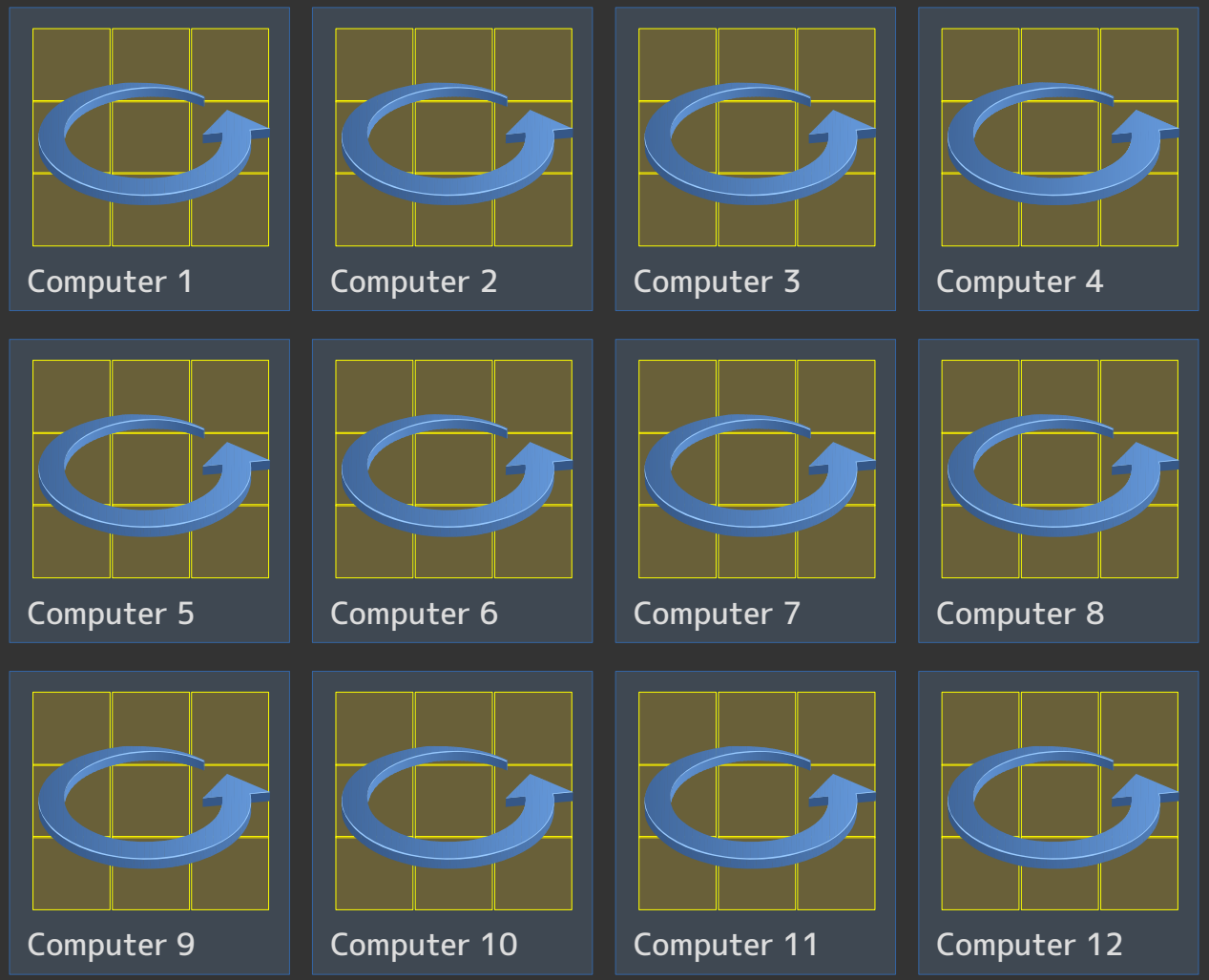

Data 1
Program 1
Computer 1
Data 2
Program 2
Computer 2
Data 1*
Program 2
Computer 1
Data 2*
Program 1
Computer 2
Usually move/restart not needed, code is small
Time=1
Time=2

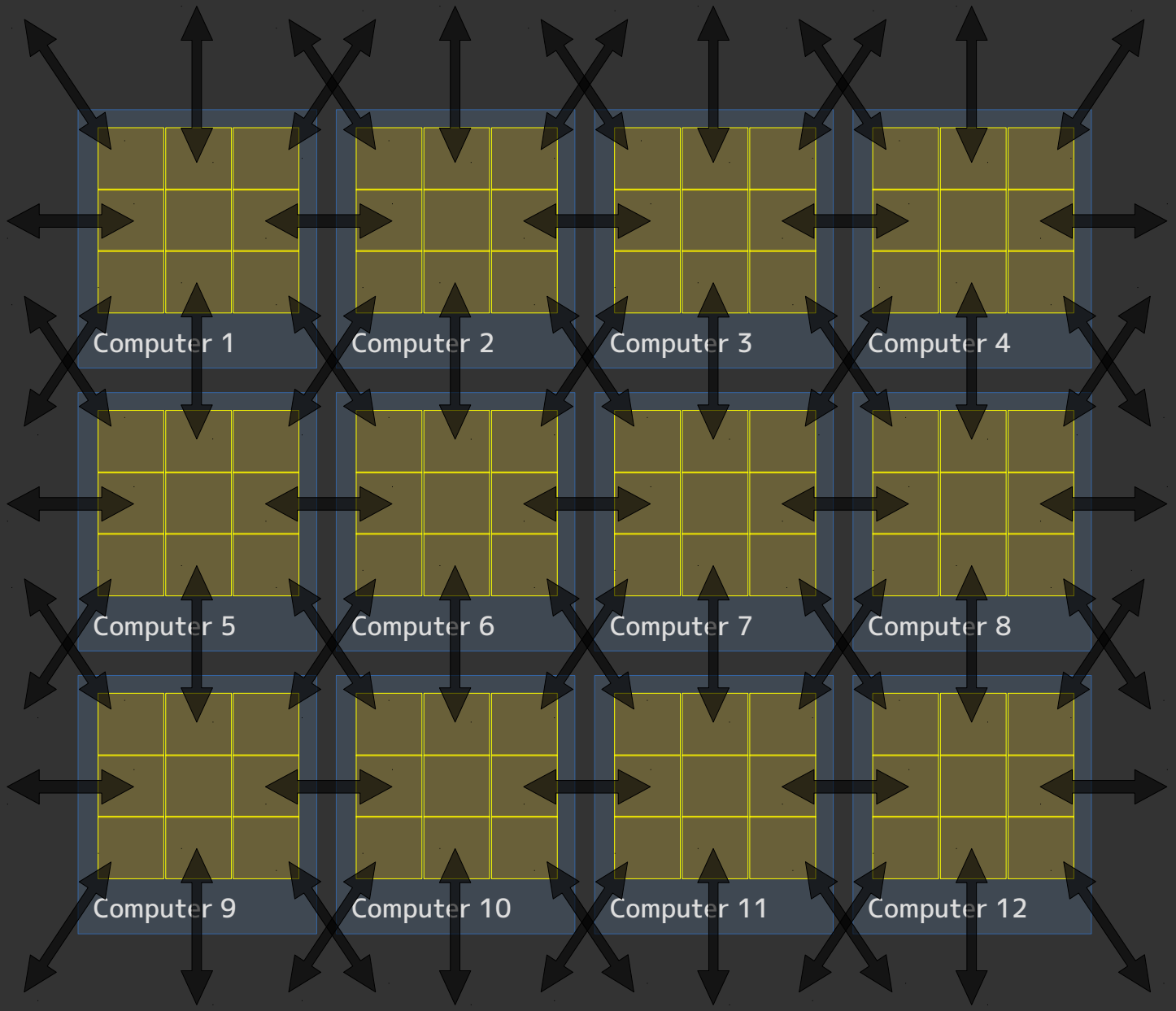
HPC World

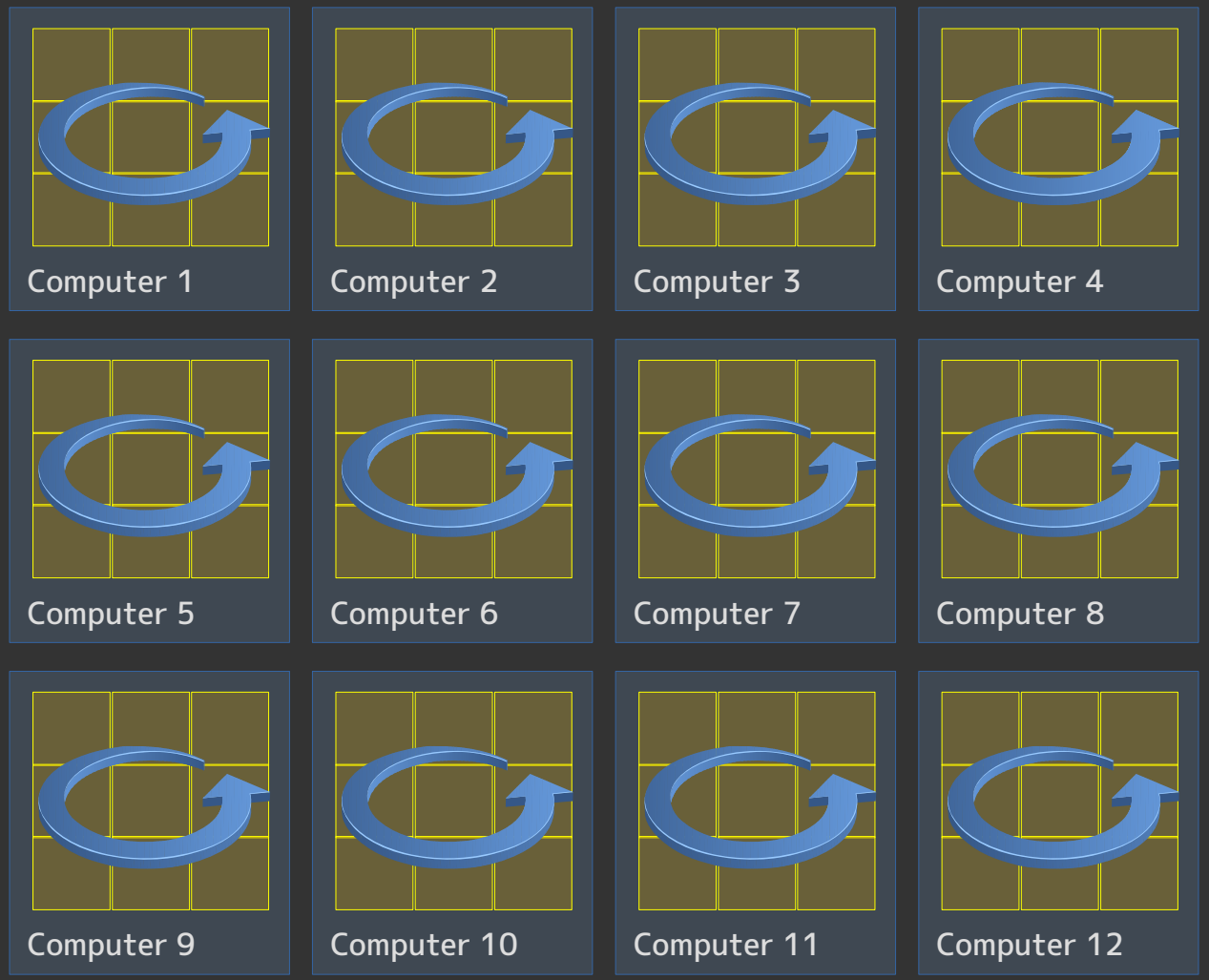


2D or 3D
Data Grid


Computer 1
Computer 2
Computer 3
Computer 4
Computer 5
Computer 6
Computer 7
Computer 8
Computer 9
Computer 10
Computer 11
Computer 12

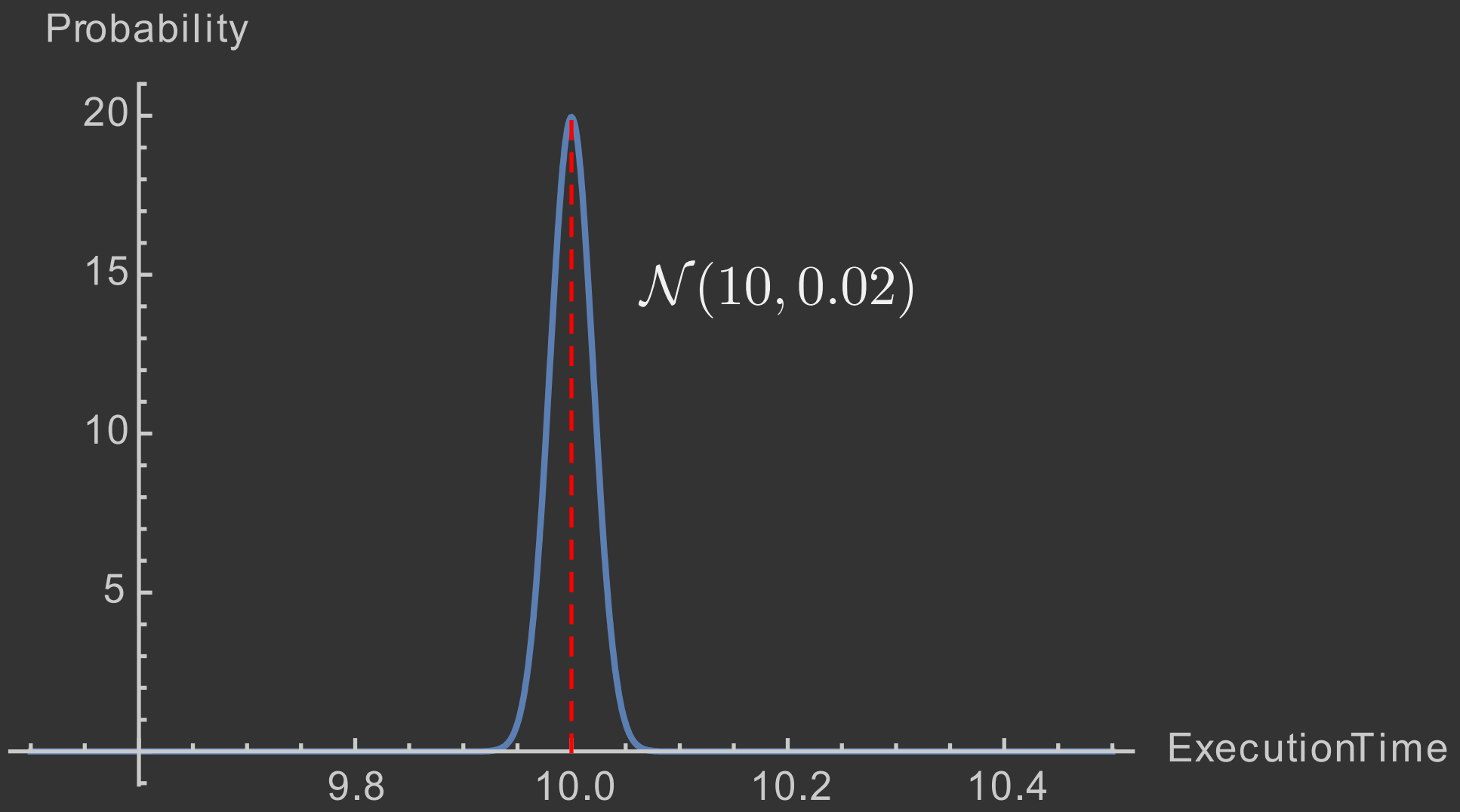

Computer 1
Computer 2
Computer 3
Computer 4
Computer 5
Computer 6
Computer 7
Computer 8
Computer 9
Computer 10
Computer 11
Computer 12

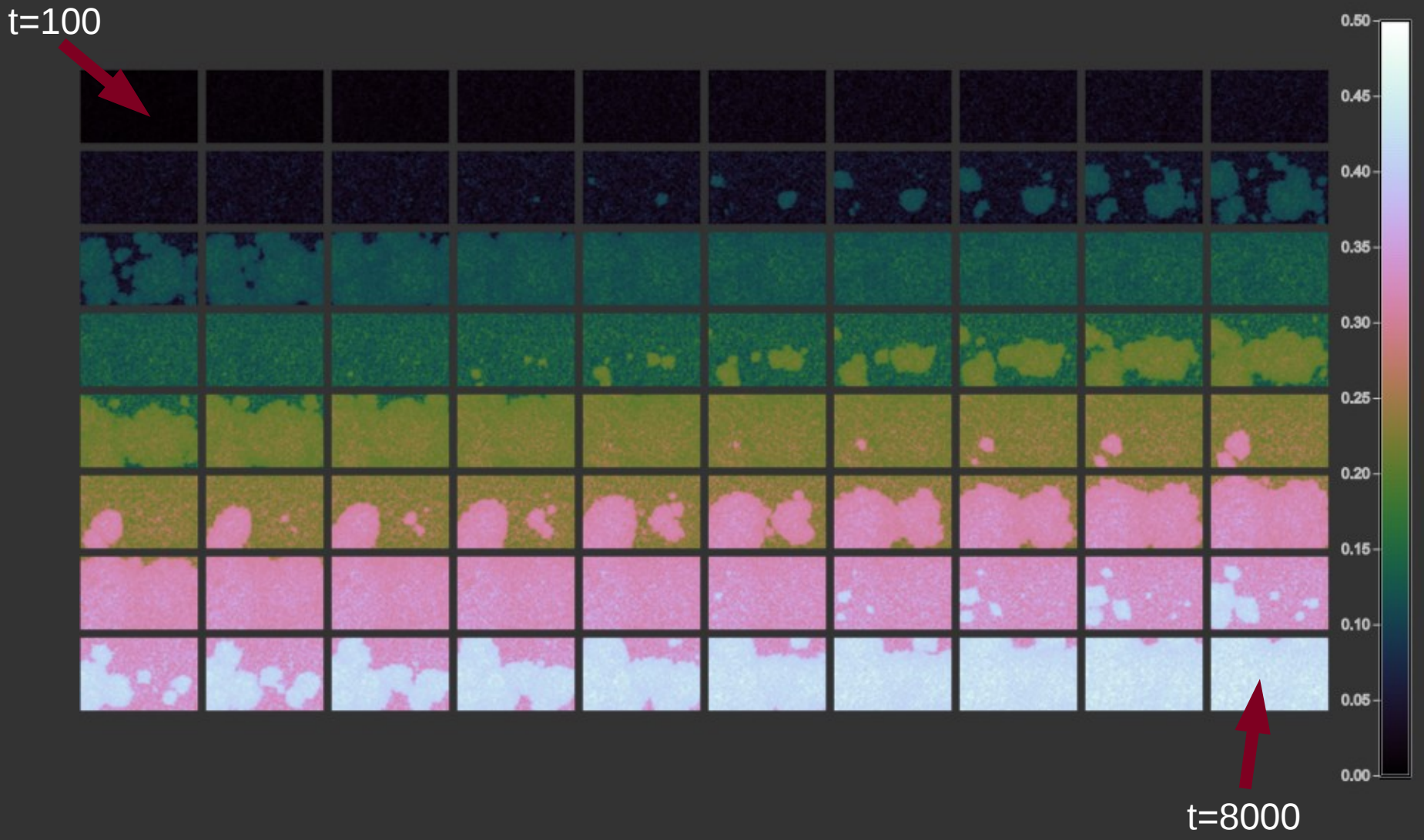

Computer 1
Computer 2
Computer 3
Computer 4
Computer 5
Computer 6
Computer 7
Computer 8
Computer 9
Computer 10
Computer 11
Computer 12

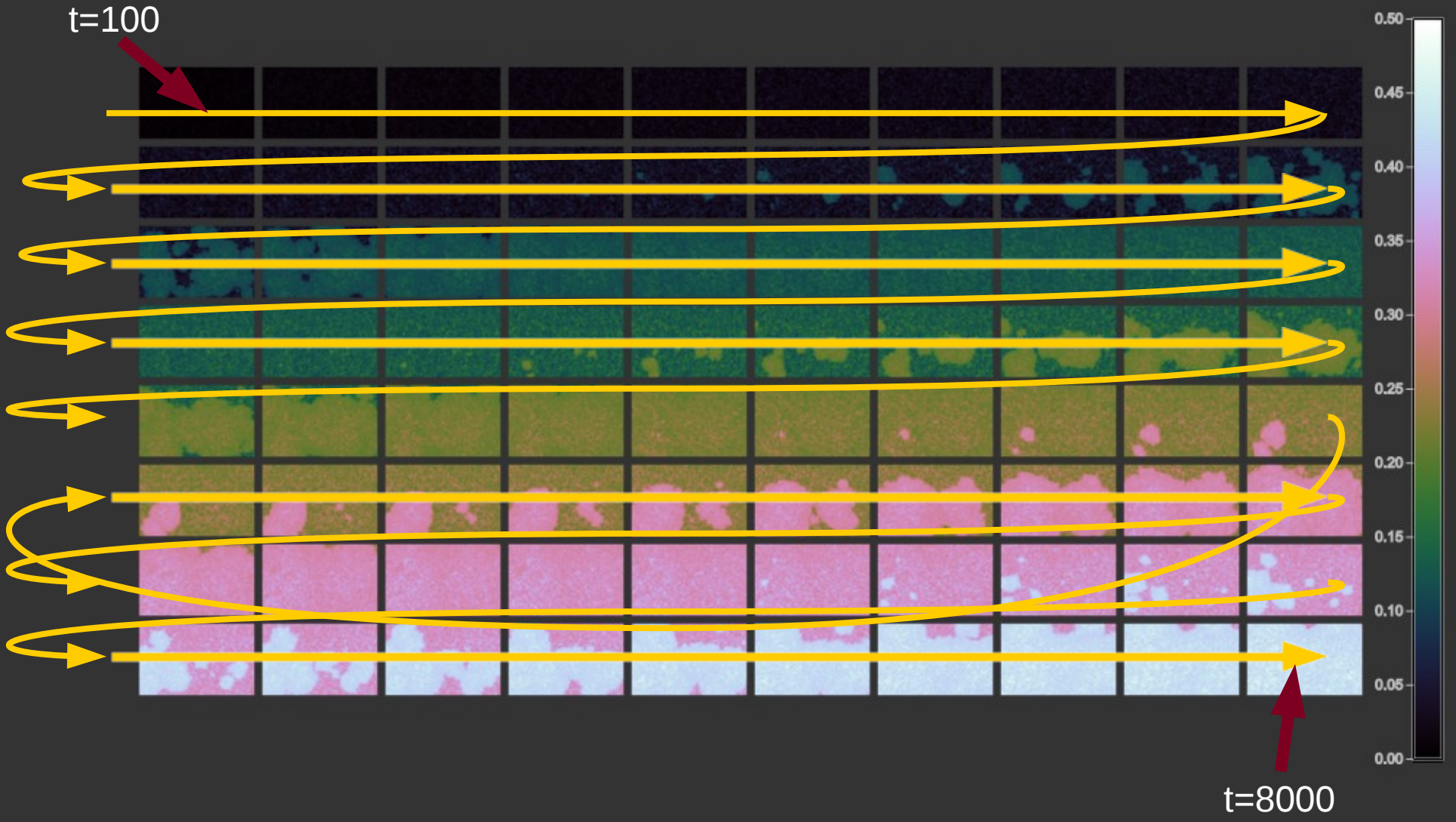


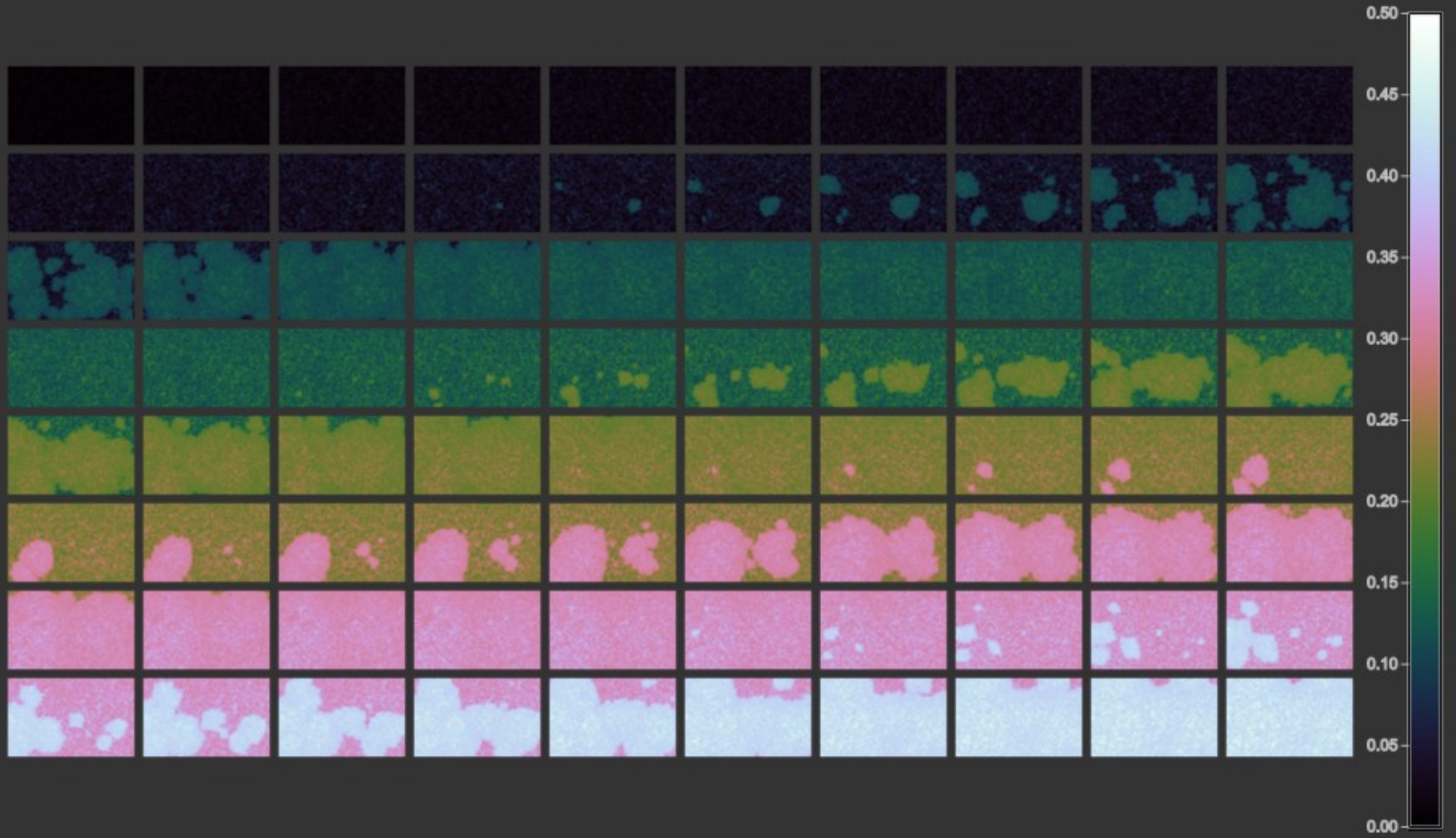
Compute Restart Requires Synchronization

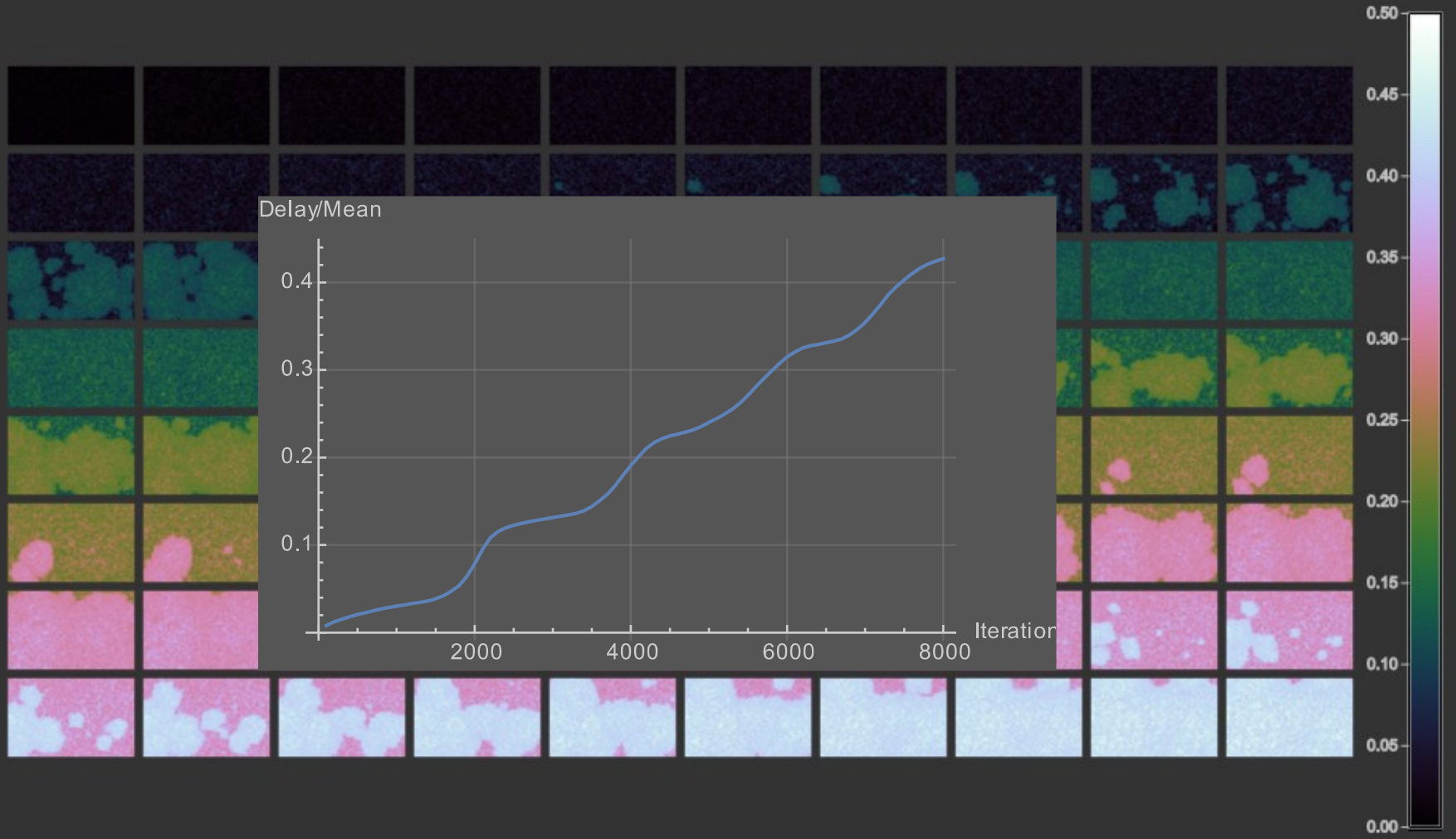
Complete Time Distribution for One Time Step




t=100
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
t=8000


t=100
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
t=8000


0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00


Delay/Mean
0.4
0.3
0.2
0.1
2000
4000
6000
8000
Iteration
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00

Reduction of Variance

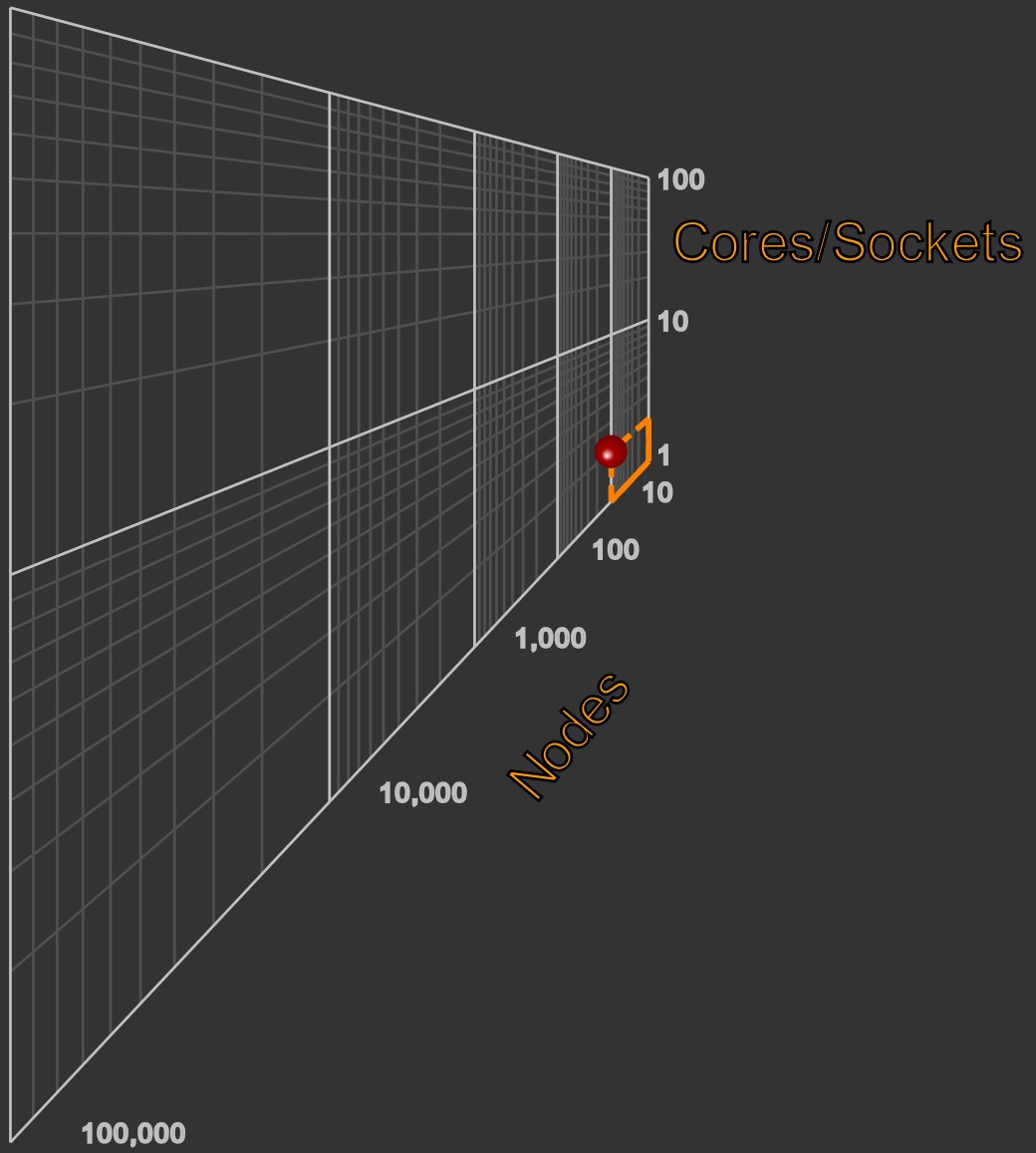
1. Isolate CPU cores
   - Instruct kernel to avoid CPU cores
   - Reroute interrupts to other CPU cores
   - Explicitly run threads/processes on now isolated CPU cores
2. Run single application
   - Easy-er with containers
   - Combine with 1.

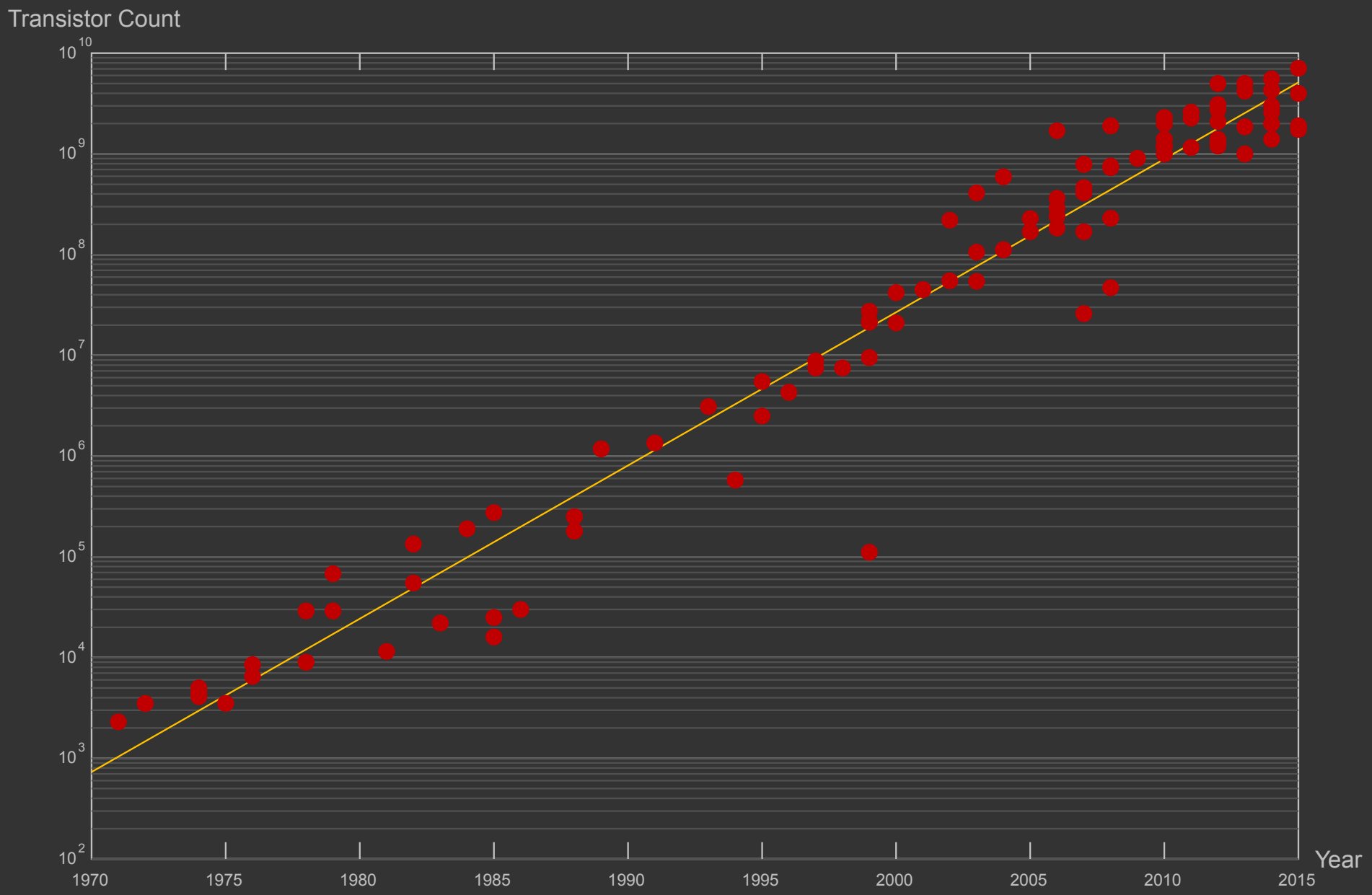
3. Use unikernel

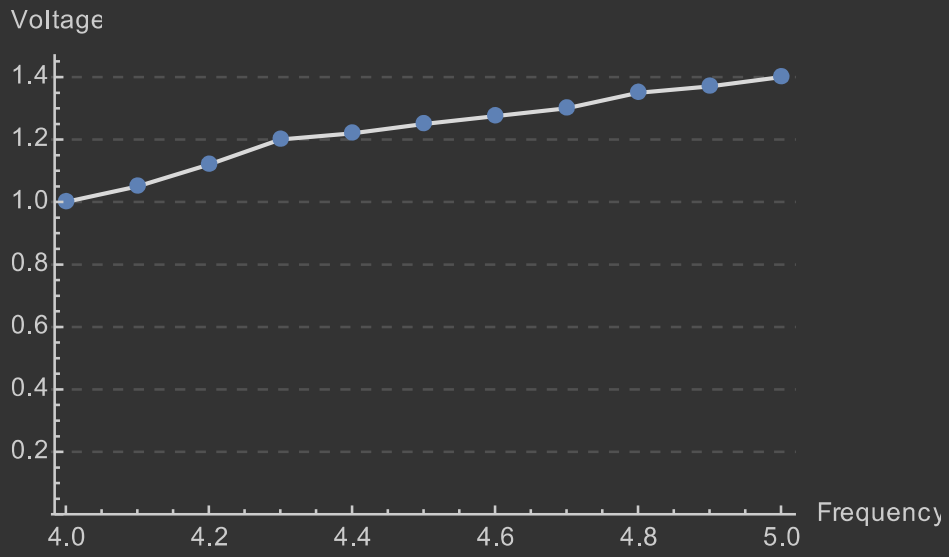
- Develop application as before and then
  - Link with kernel as library
  - Include all necessary device drivers
  - Link with runtime which calls into kernel instead of system calls
- Complete control over CPU cores

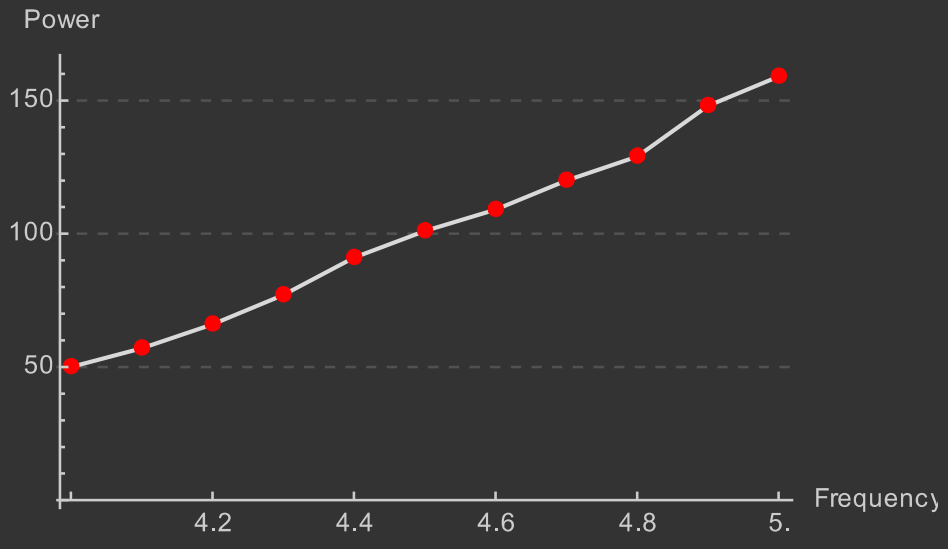
Two approaches to code distribution

- Native code
  - Homogenic environment required
  - Recompilation after every change
- Byte code
  - Needs JIT compilation for performance
  - Cannot use CPU features fully


100
Cores/Sockets
10
1
10
100
1,000
Nodes
10,000
100,000


Transistor Count
10^{10}
10^{9}
10^{8}
10^{7}
10^{6}
10^{5}
10^{4}
10^{3}
10^{2}
1970
1975
1980
1985
1990
1995
2000
2005
2010
2015
Year

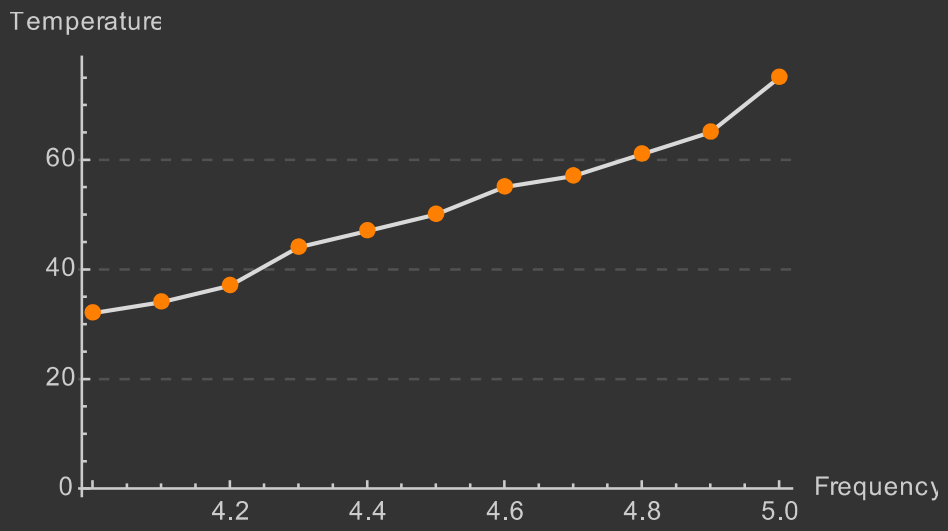




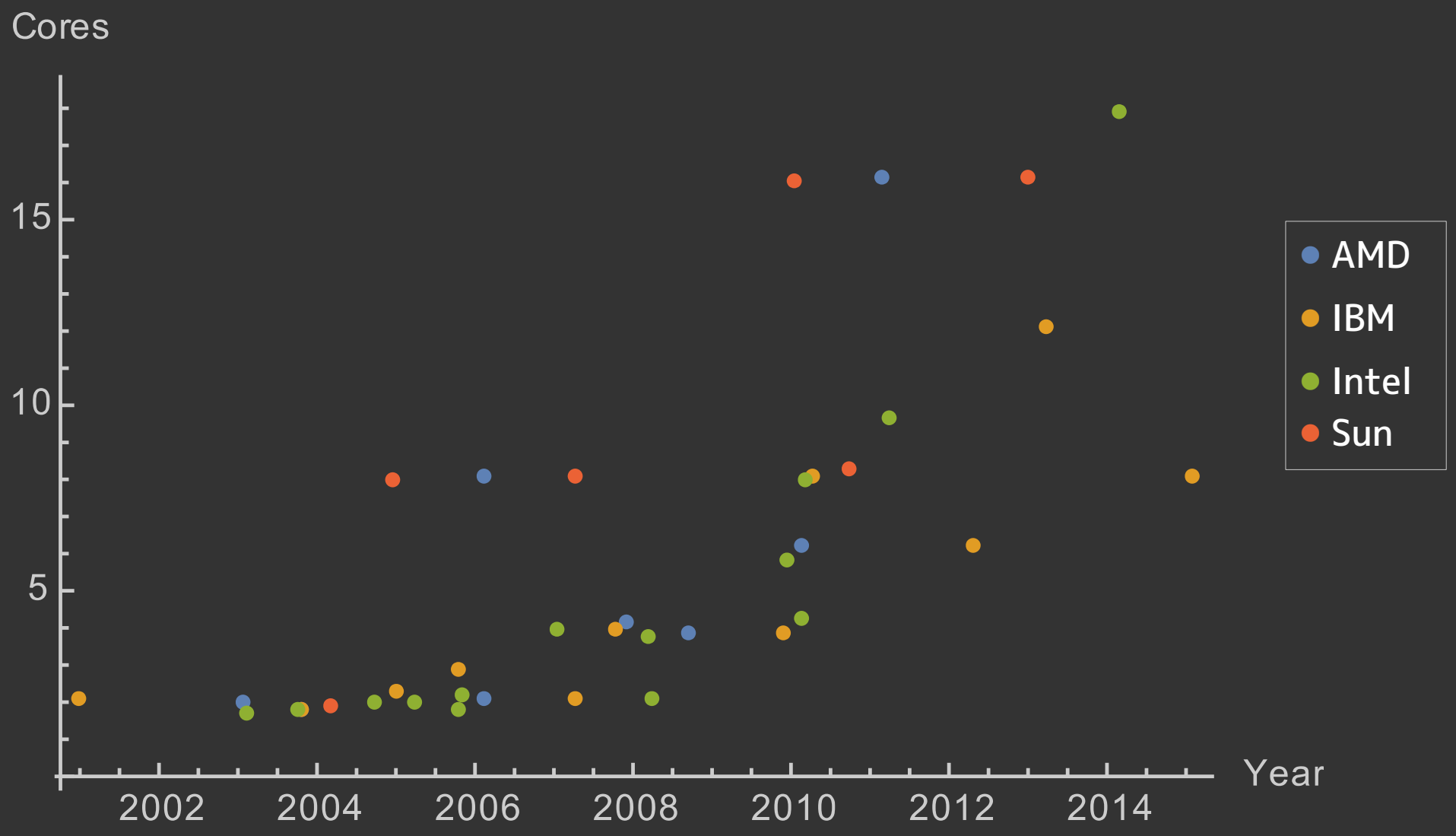
What it takes to increase frequency:

- Dynamic power:

$$P = C \cdot V^2 \cdot f$$

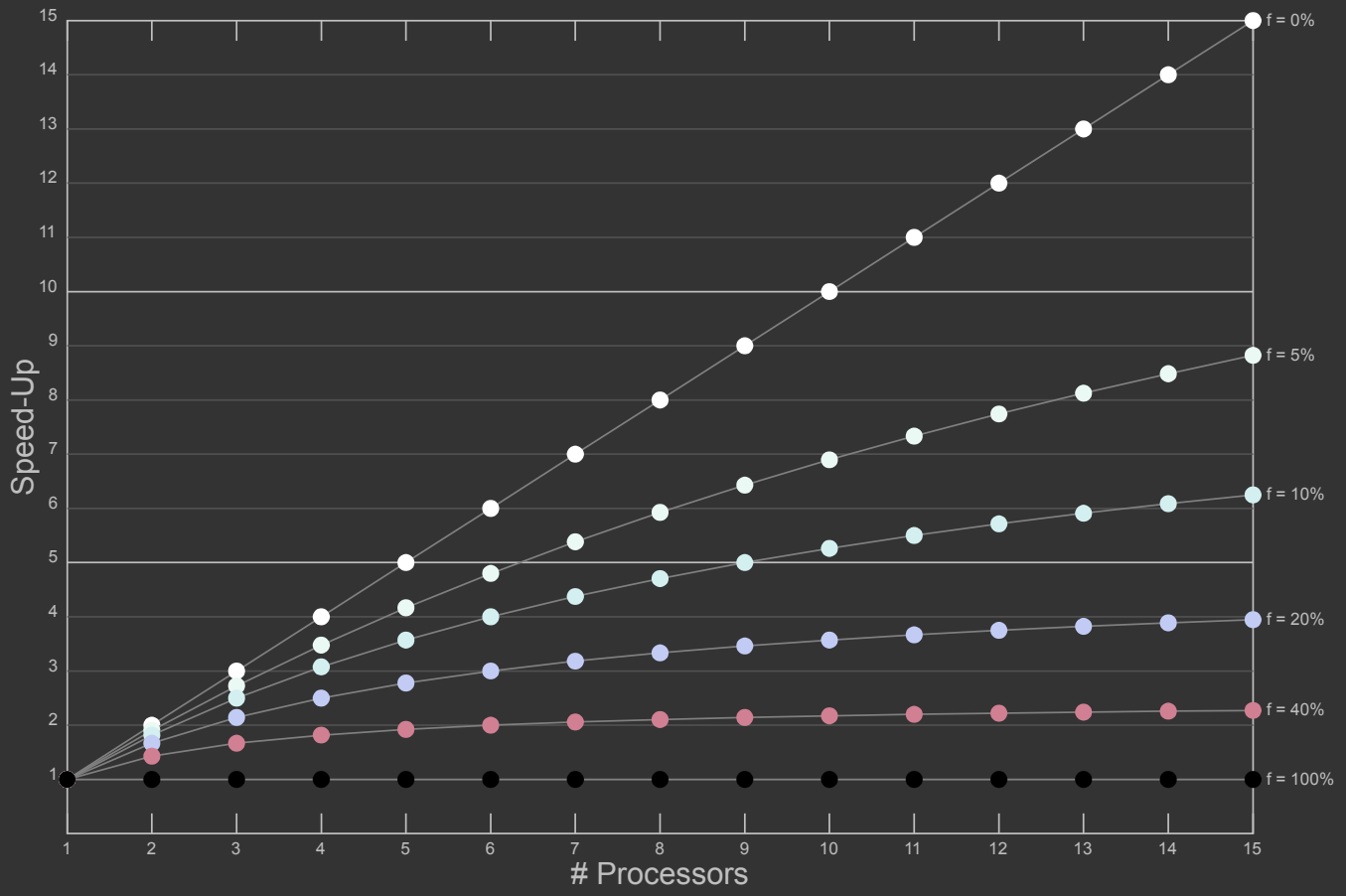


Core Count (WS and Server Chips)



Values are jittered horizontally to make all points visible

Amdahl's Law



$$S(n) = \frac{n}{nf + (1 - f)} \qquad \lim_{n \to \infty} S(n) = f^{-1}$$

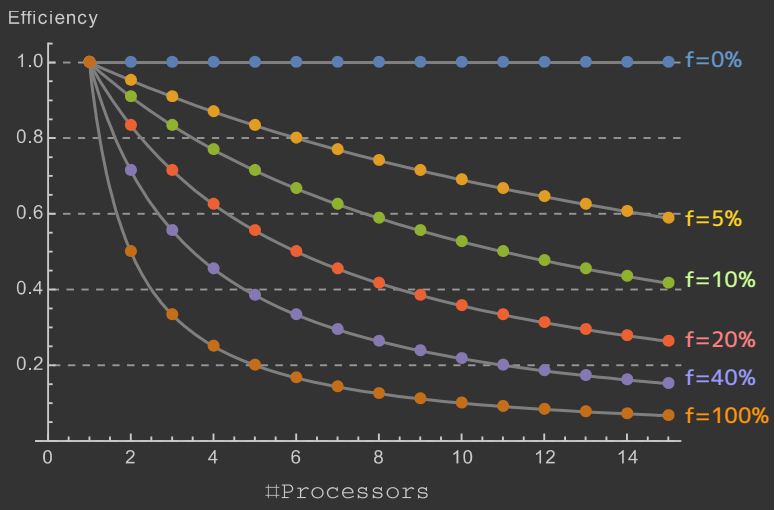
$$f = \text{fraction of serial code}$$

Serializing

- Components of f
  - Waiting on shared resources
    - Implicitly controlled by hardware
    - Enforced by software through synchronization
  - Explicit synchronization
    - Needed for correctness

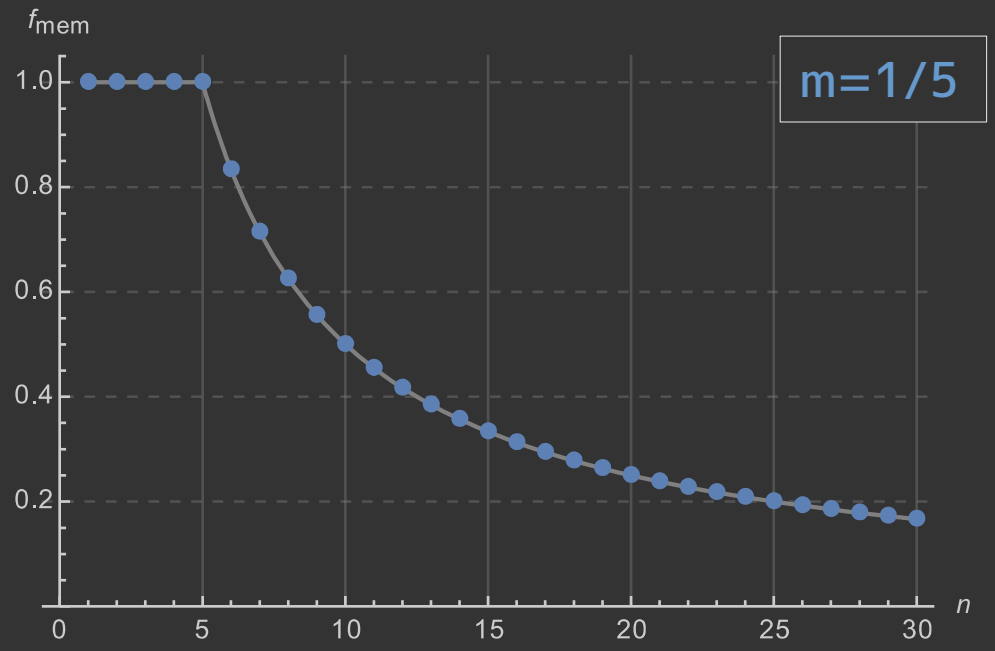
$$\text{Efficiency} \approx \frac{1}{n} S(n) = \frac{1}{nf + (1 - f)}$$

$$\lim_{n \to \infty} \text{Efficiency} = \begin{cases} 1 & f = 0 \\ 0 & f \neq 0 \end{cases}$$



Memory Bandwidth/Latency Factor

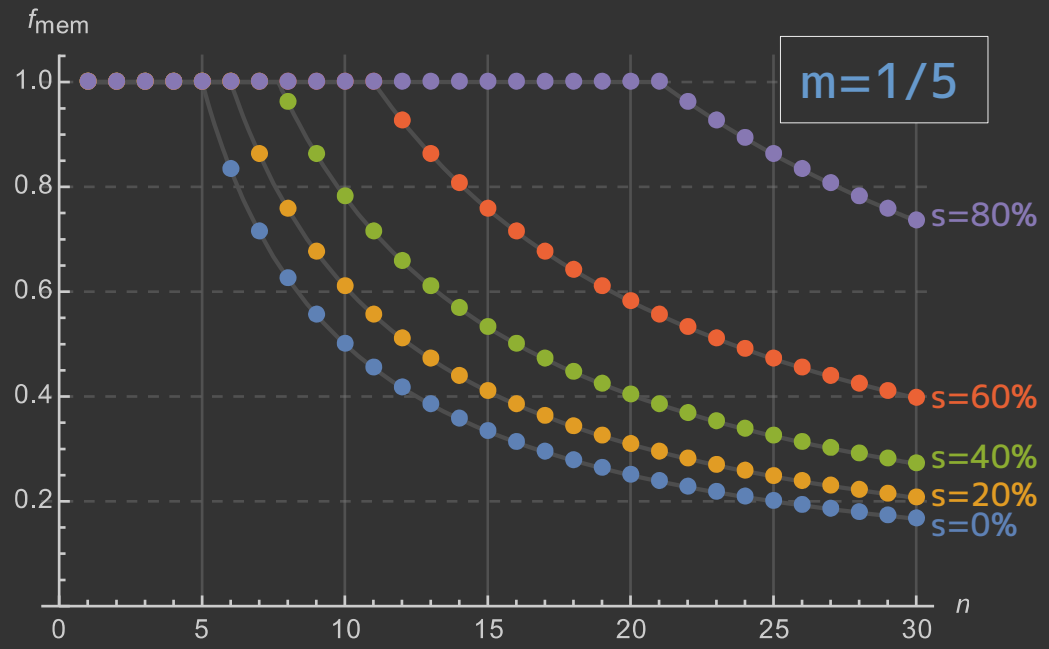
	CPU Clock	Memory Access	Effective Clock
Early 80s	1MHz	1 cycle	1 MHz
today	4GHz	250 cycles	16 MHz



$$f_{\text{mem}} = \frac{1}{mn} \min(1, mn)$$

Memory Bandwidth Factor with Sharing

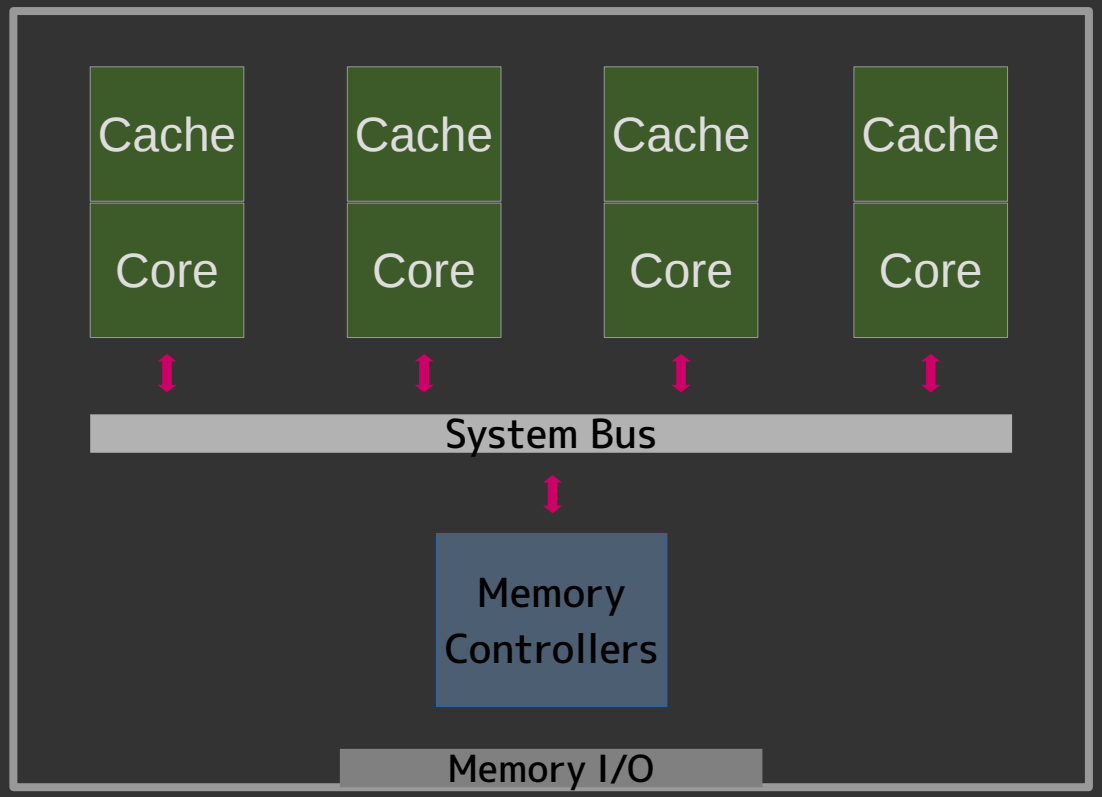
Working sets overlap



s = fraction shared

$$f_{\text{mem}} = \frac{1}{mnf_s} \min\left(1, mnf_s\right) \text{ with } f_s = 1 - s\frac{n-1}{n}$$

Bus Collisions

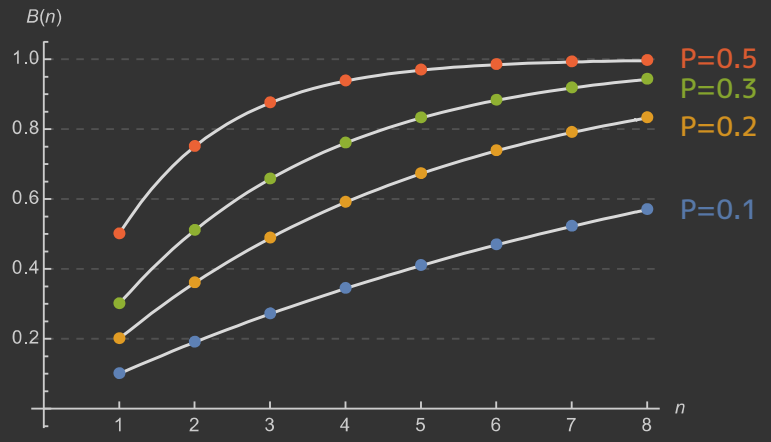


Collision Probability:

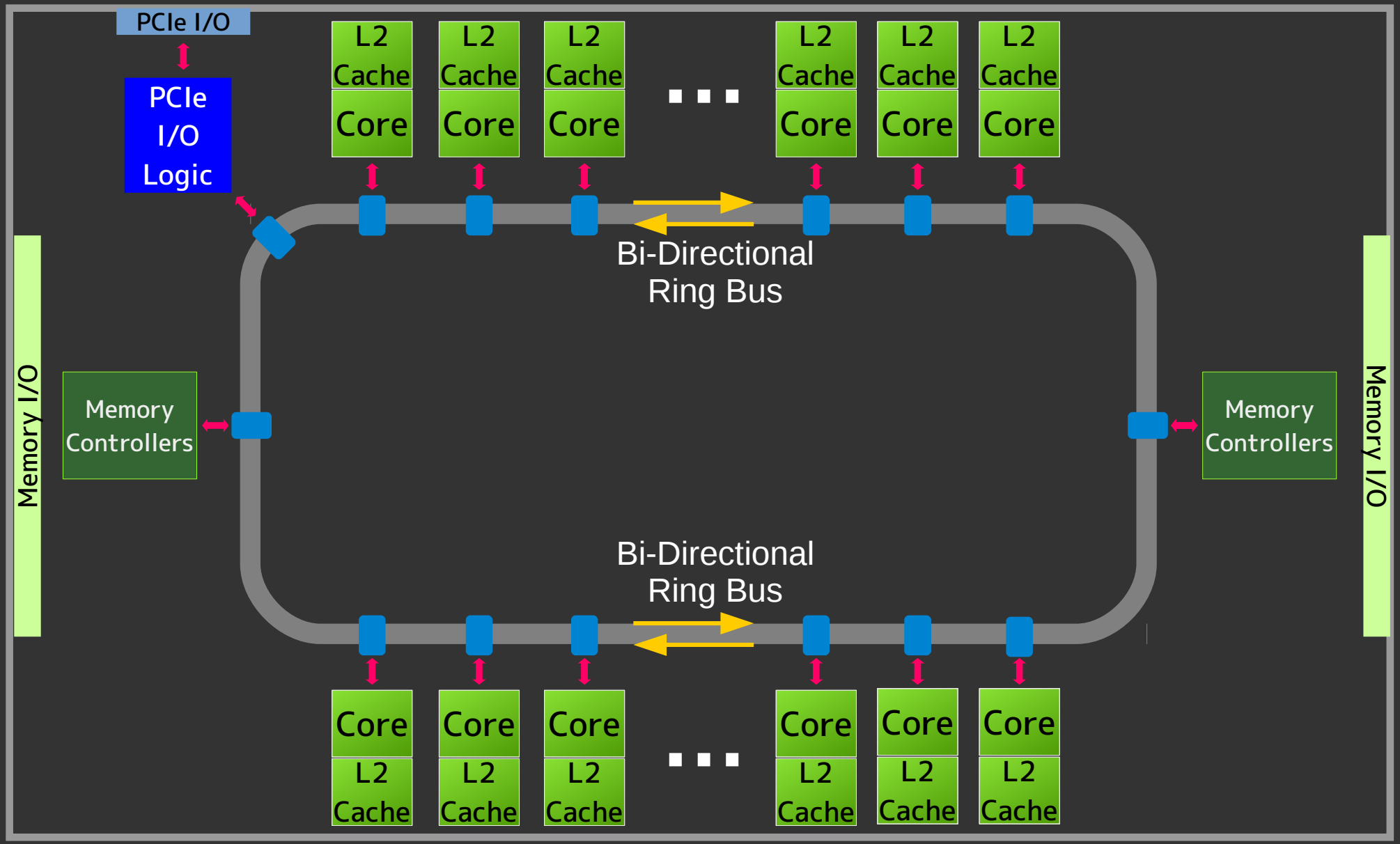
- P_i: Communication Probability for core *i*

$$B(n) = 1 - \prod_{i=1}^{n} 1 - P_i$$

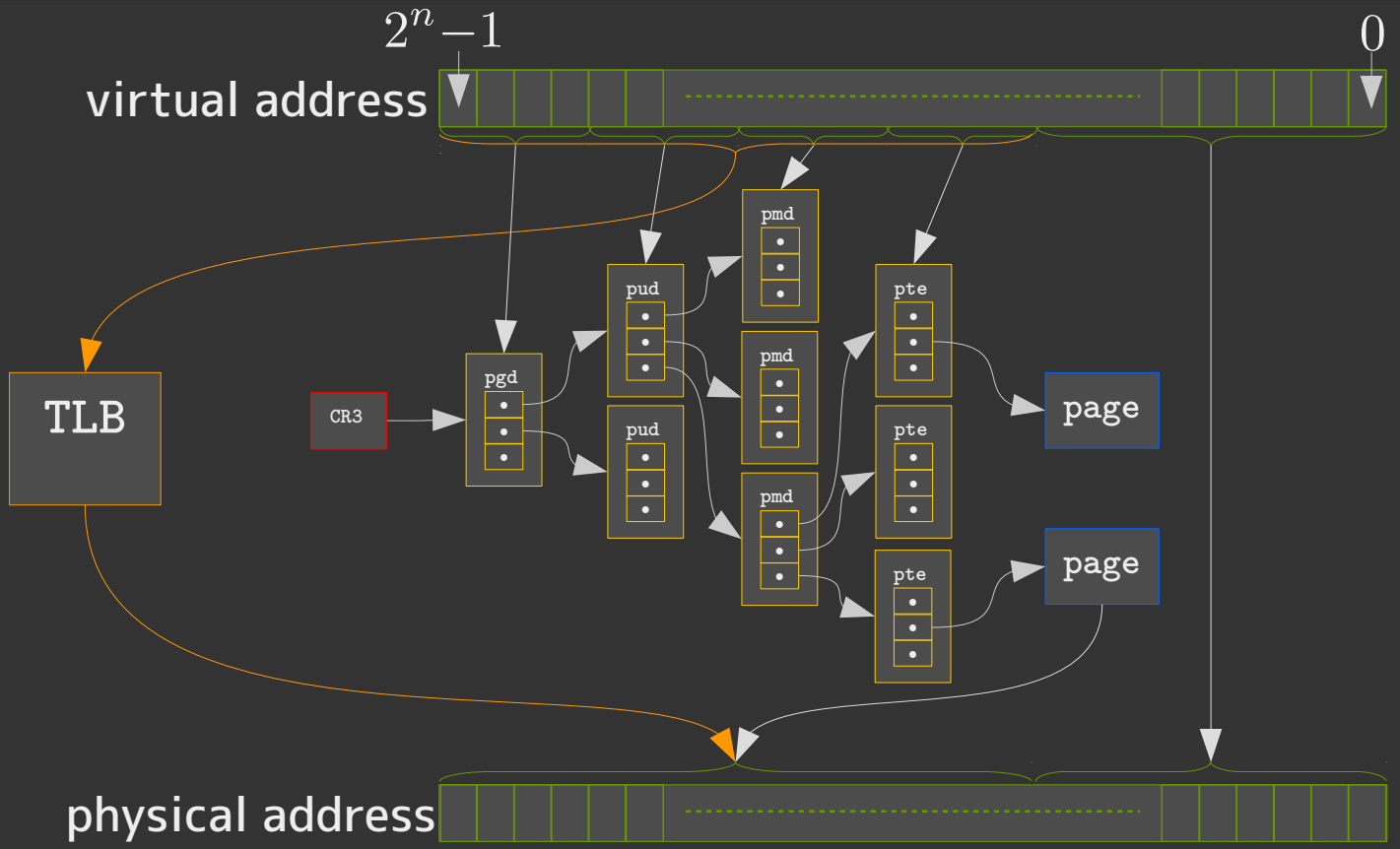
$$\approx 1 - (1 - P)^n$$



Busses Even in Modern Designs

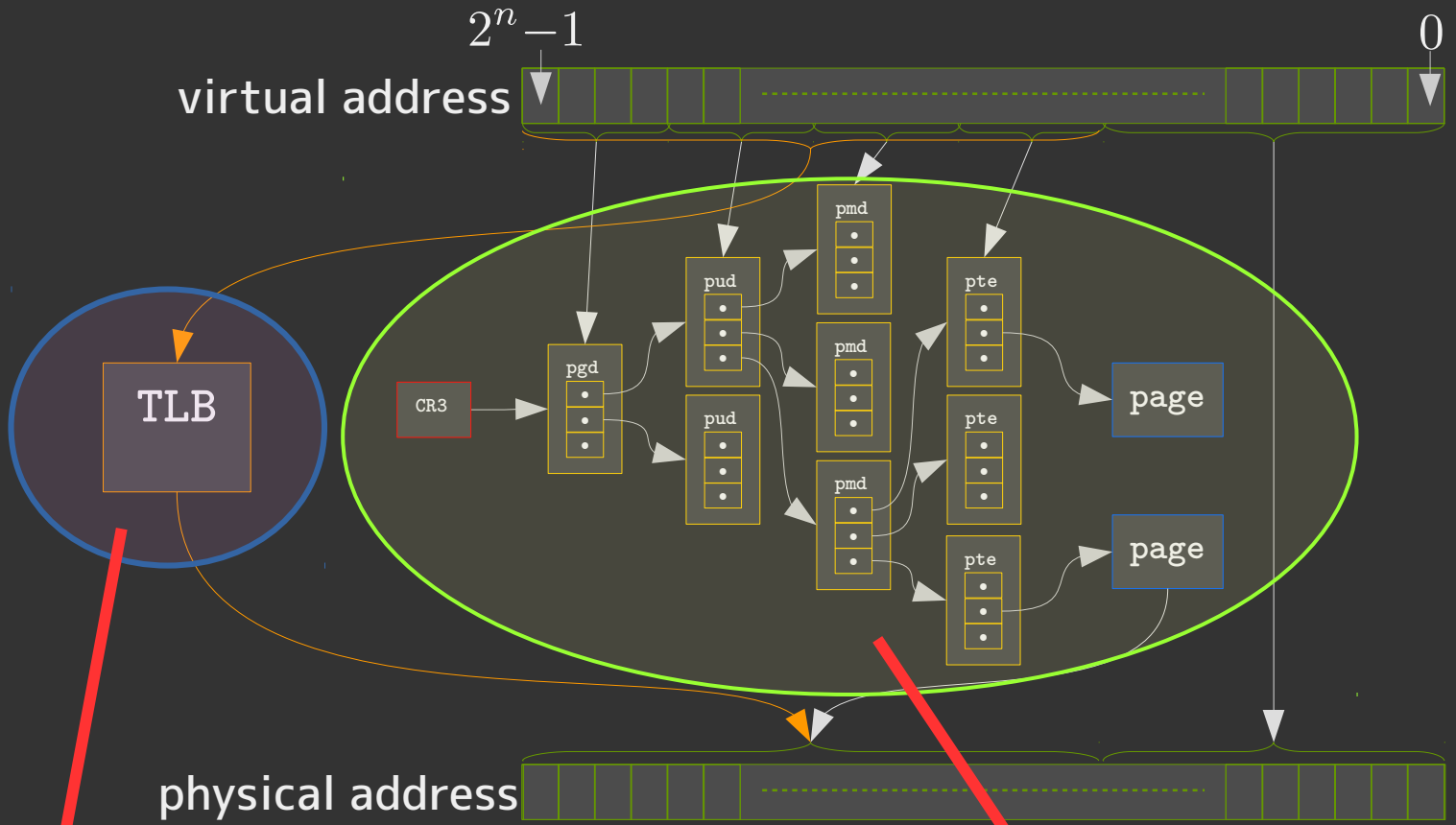


Address Resolution



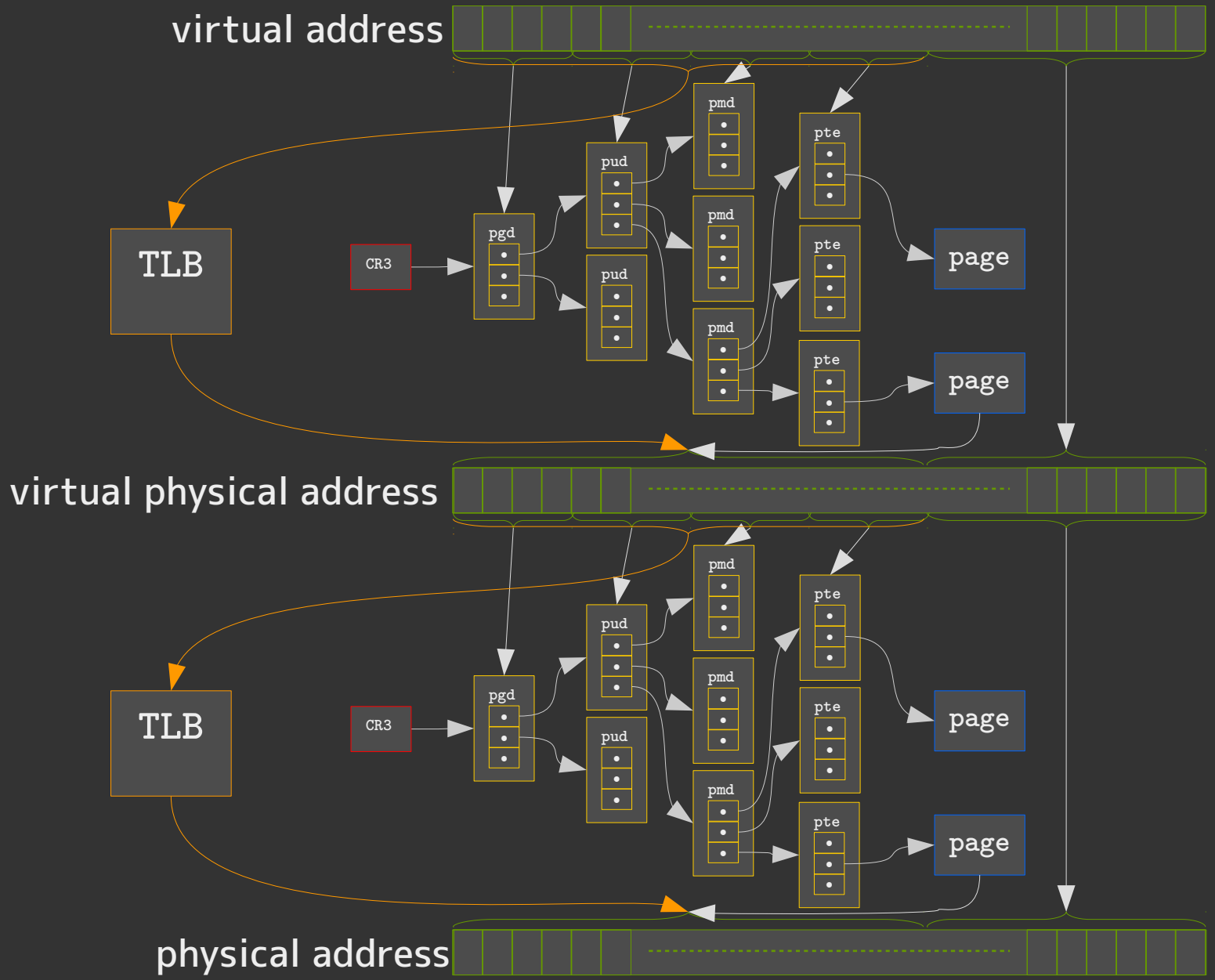
Shared resources:

- Cache for page table tree
- TLB

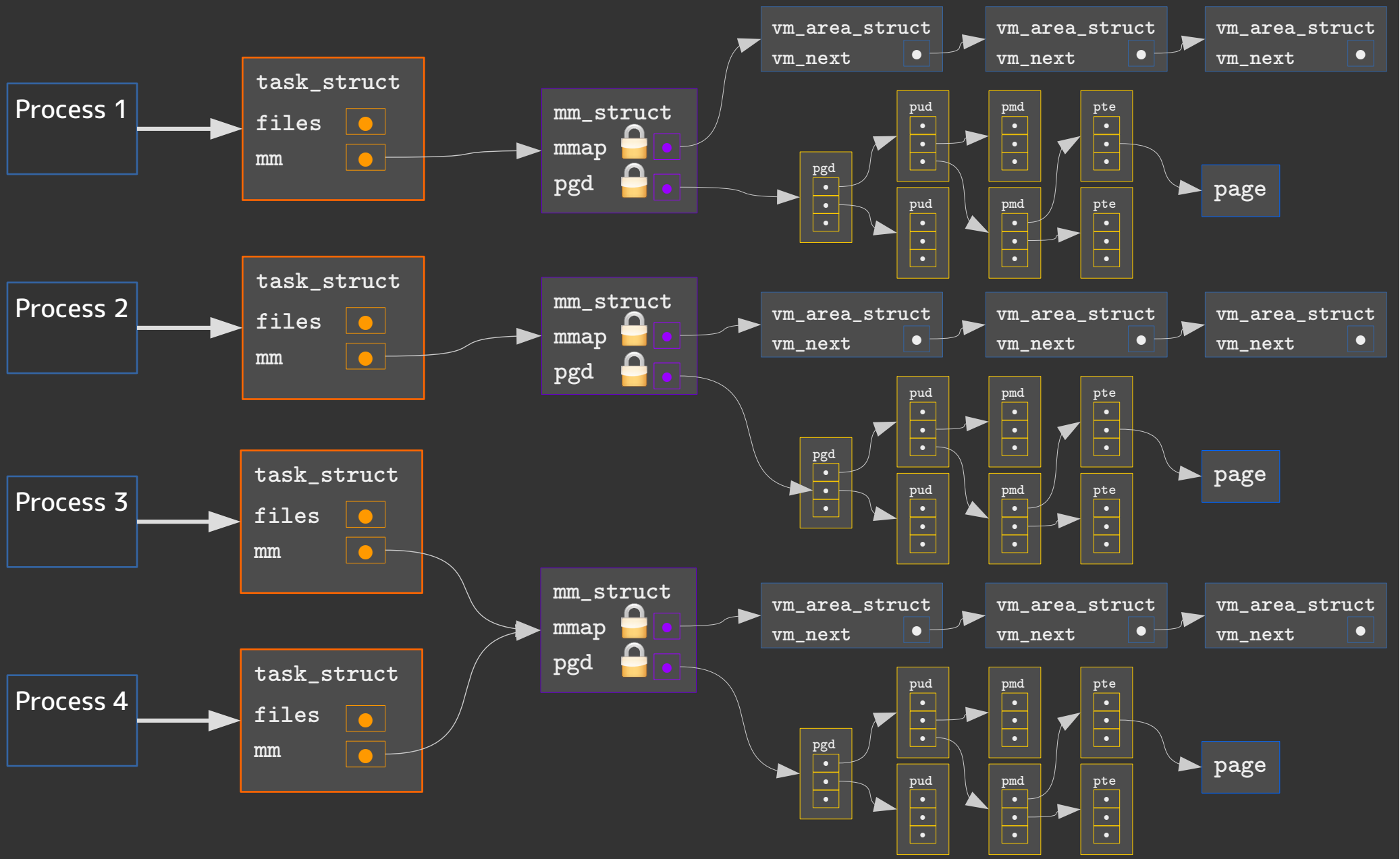


$\leq 10 - 30$ cycles

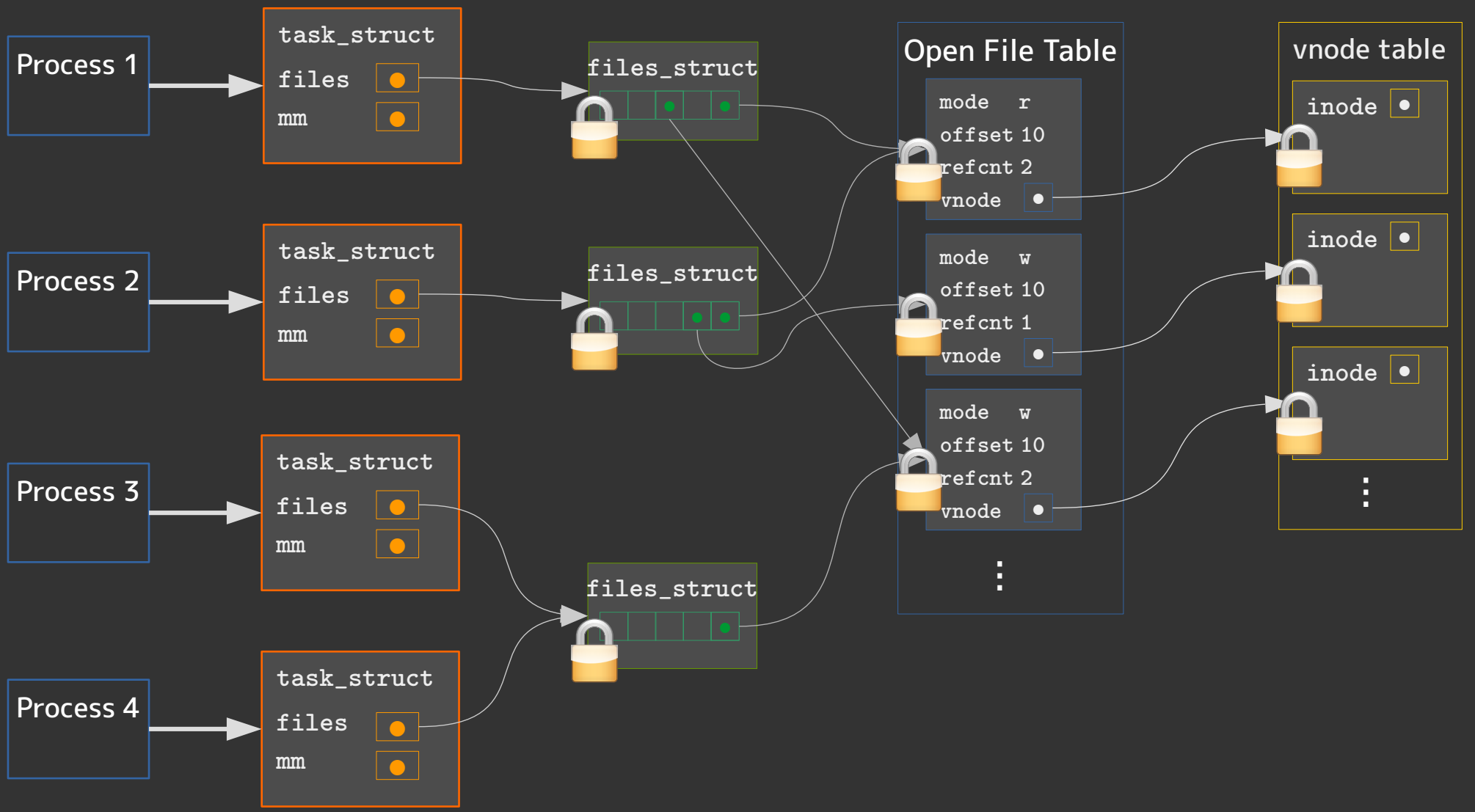
up to ≈ 1500 cycles


virtual address
TLB
CR3
pgd
pud
pud
pmd
pmd
pmd
pte
pte
pte
page
page
virtual physical address
TLB
CR3
pgd
pud
pud
pmd
pmd
pmd
pte
pte
pte
page
page
physical address

Shared Data Structures: VM

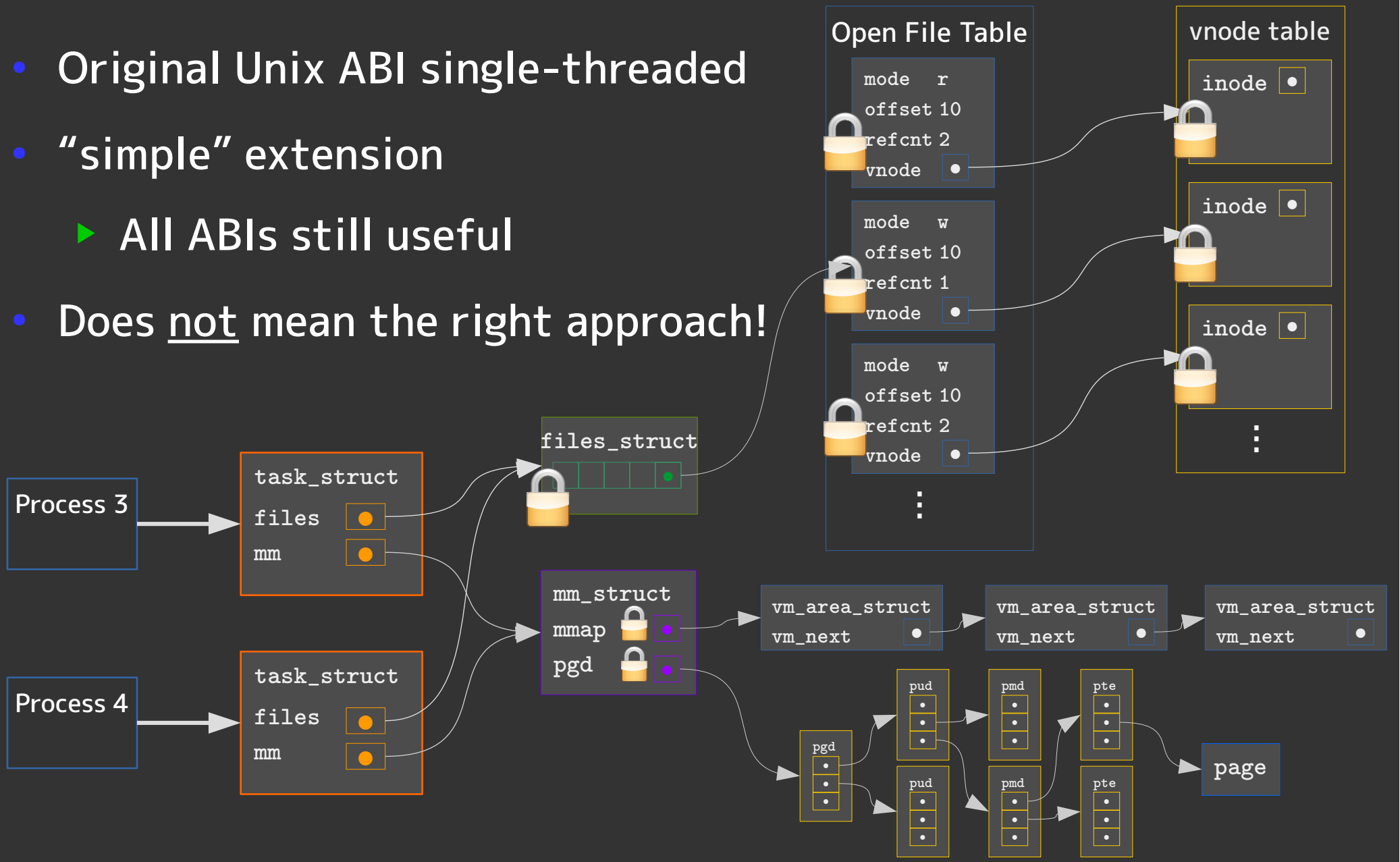


Shared Data Structures: File System



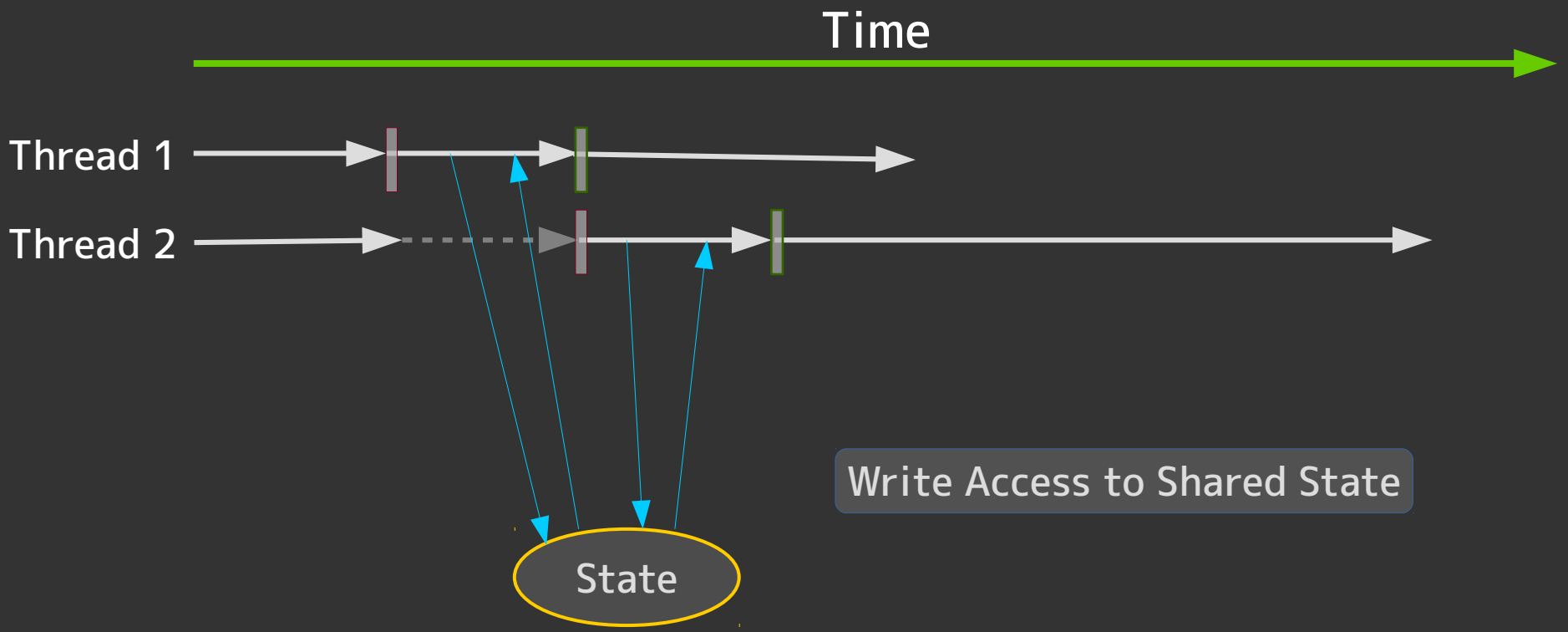
Threads Are More Of An Accident

- Original Unix ABI single-threaded
- “simple” extension
  - All ABIs still useful
- Does not mean the right approach!



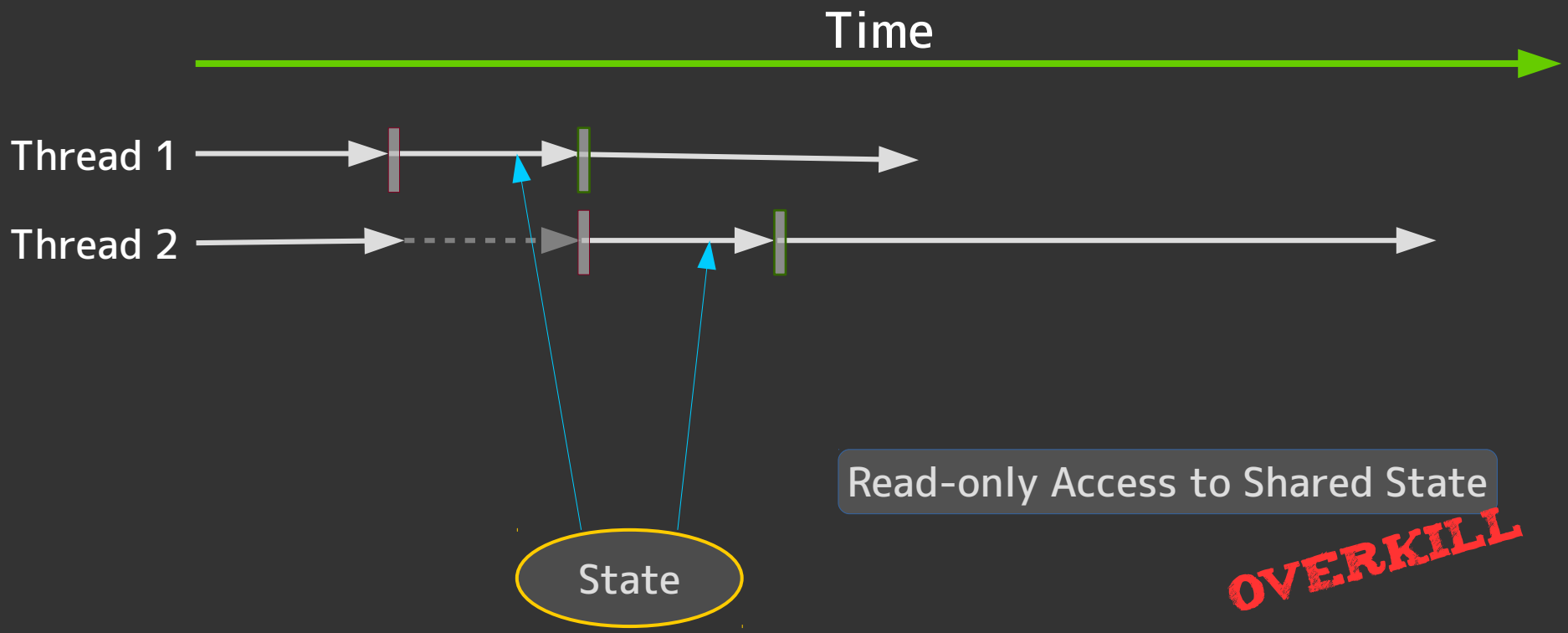
Explicit Concurrency Control

- Requires explicit synchronization
- Must be pessimistic about necessity



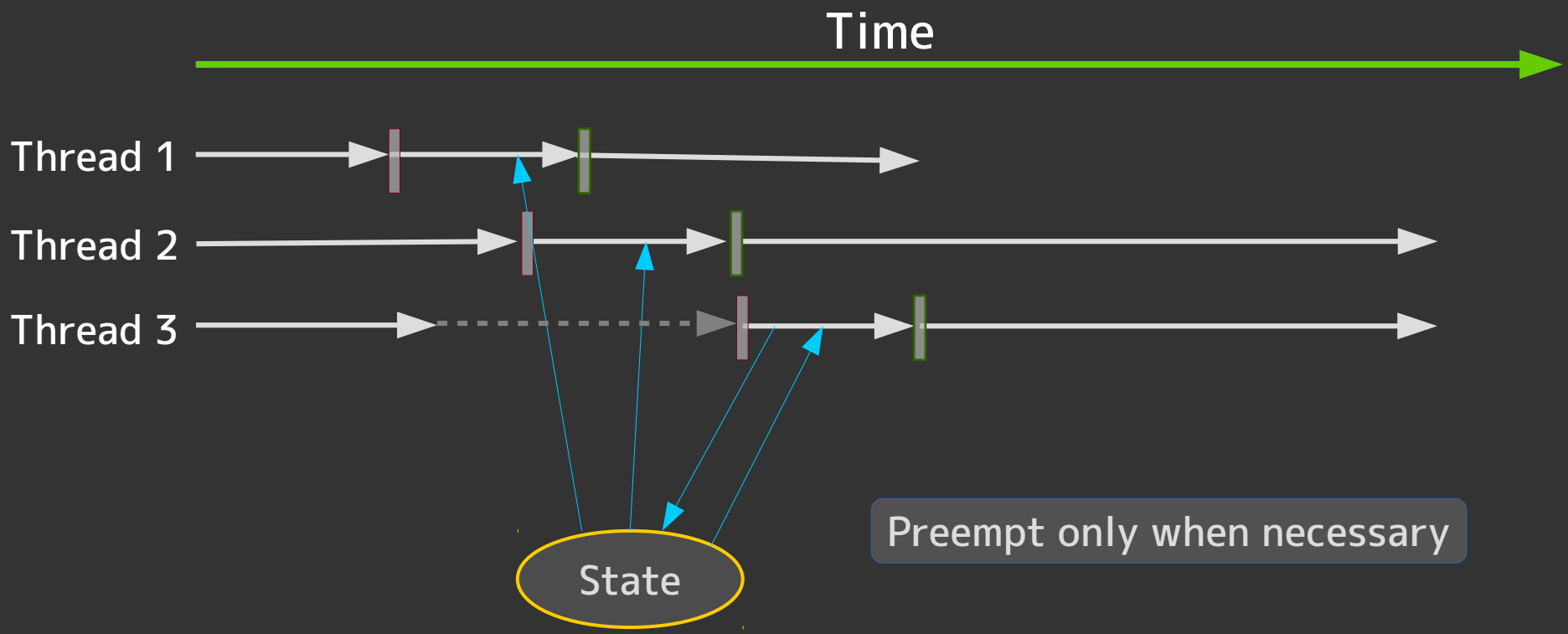
Explicit Concurrency Control

- Requires explicit synchronization
- Must be pessimistic about necessity

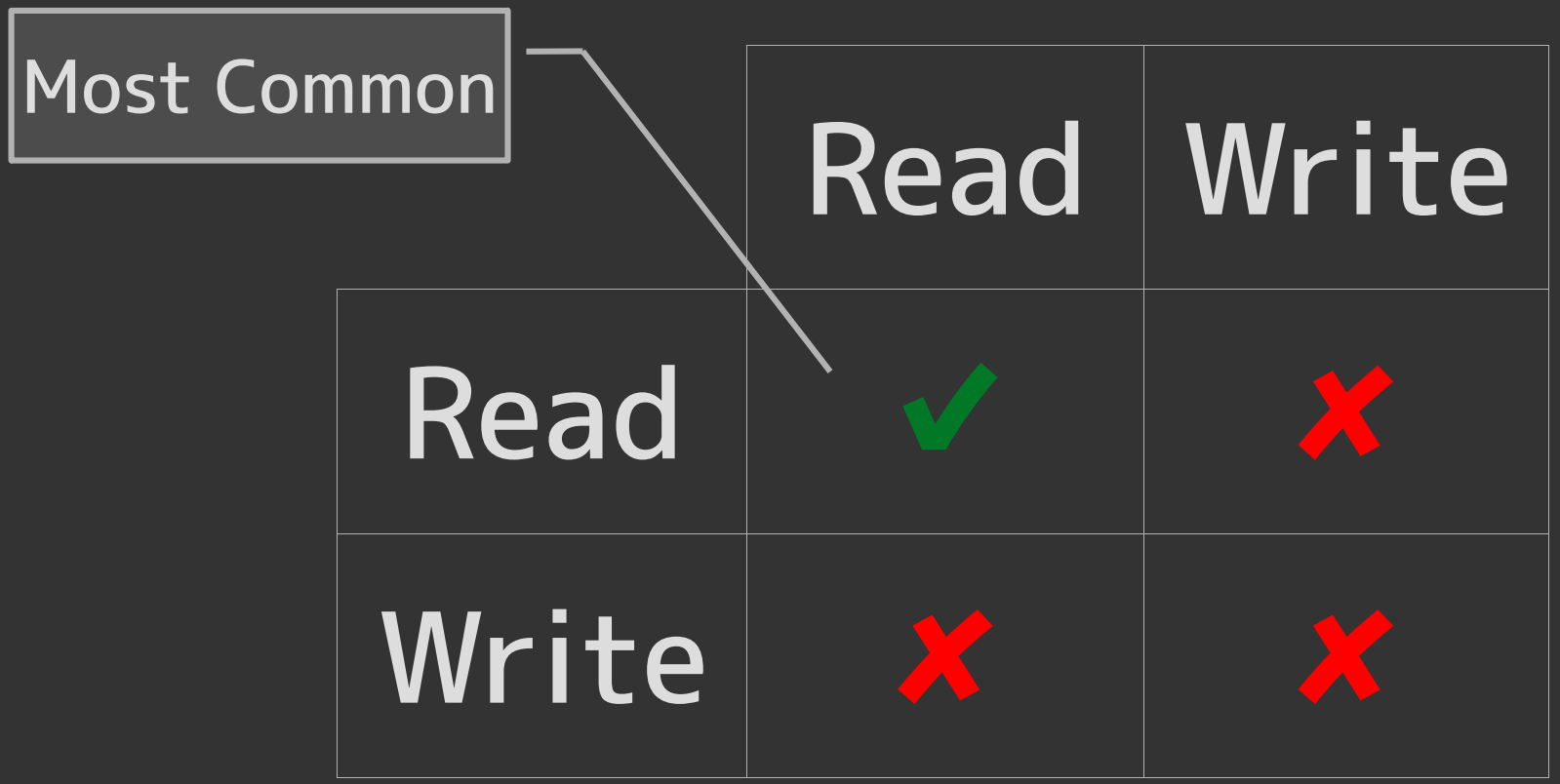


Explicit Concurrency Control

- Use more information:
  - Reader/writer locks

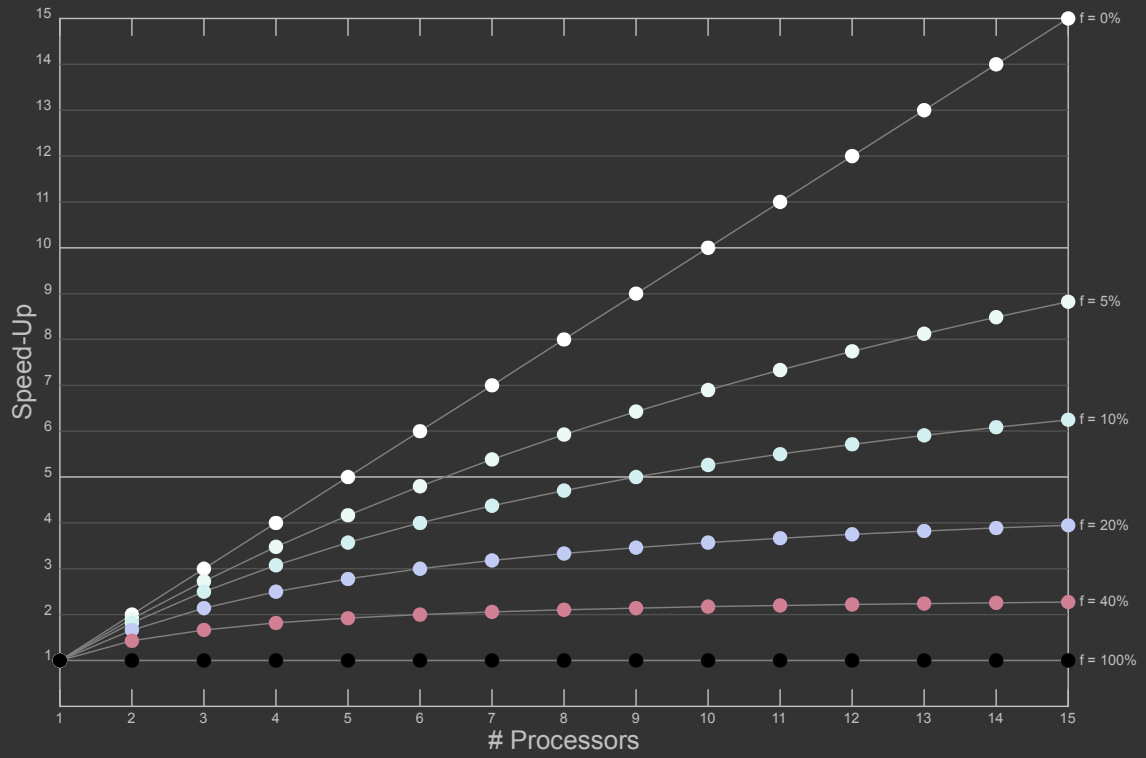


- Possible Concurrent Uses:



Fine Grained Synchronization

- Versus few locks
  - Helps keeping serialization factor *f* down
  - Remember:



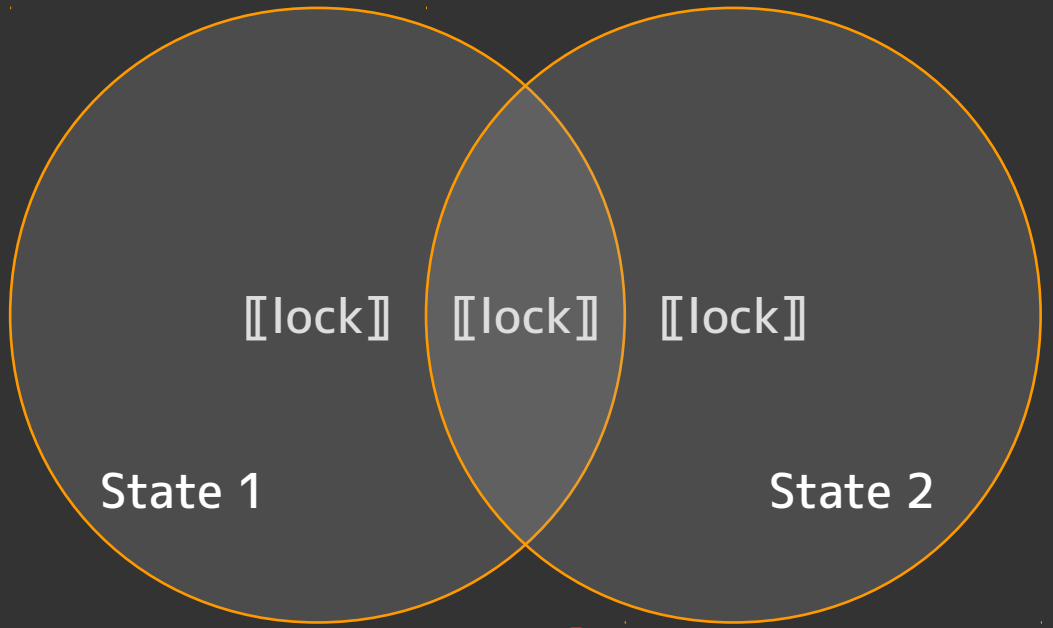
Many States

- Keep track of locks
- Not used together:

Easy!

Two locks – two states

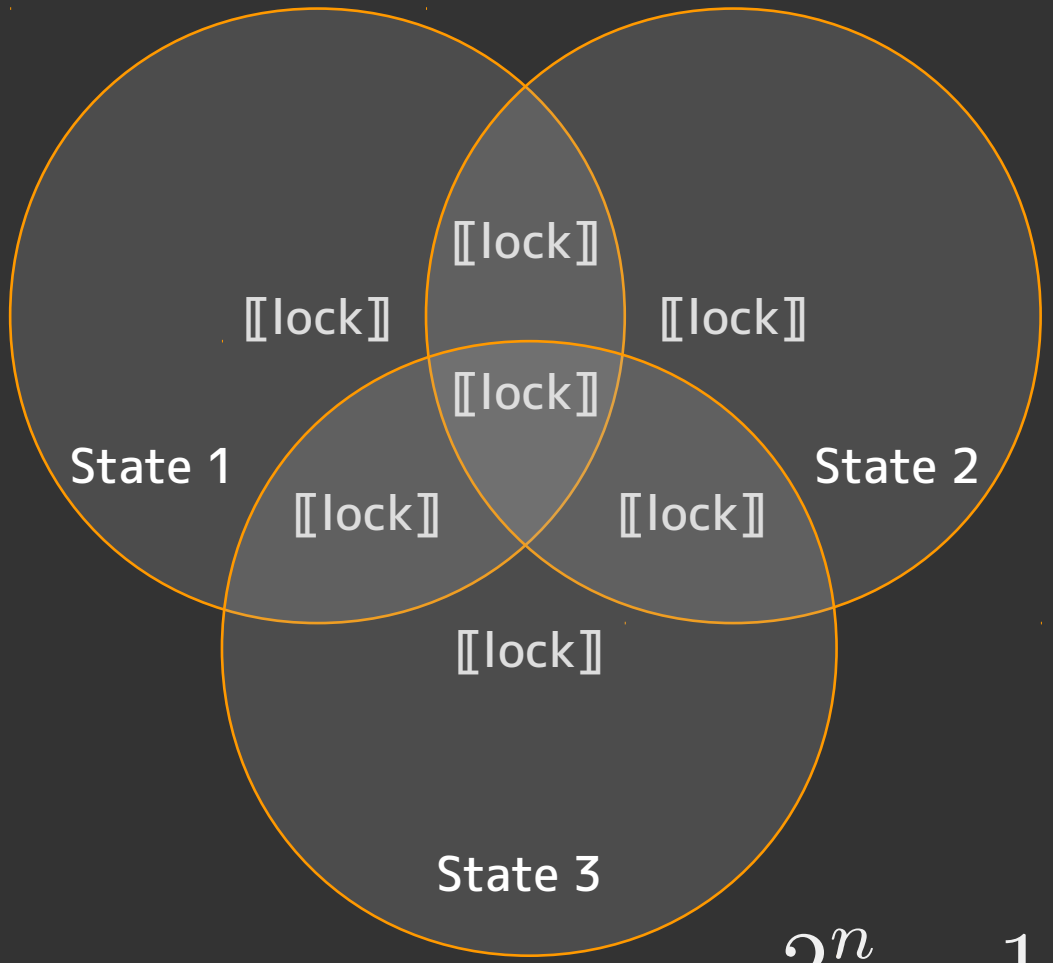
- Keep track of locks
- Used together:



Ugh!

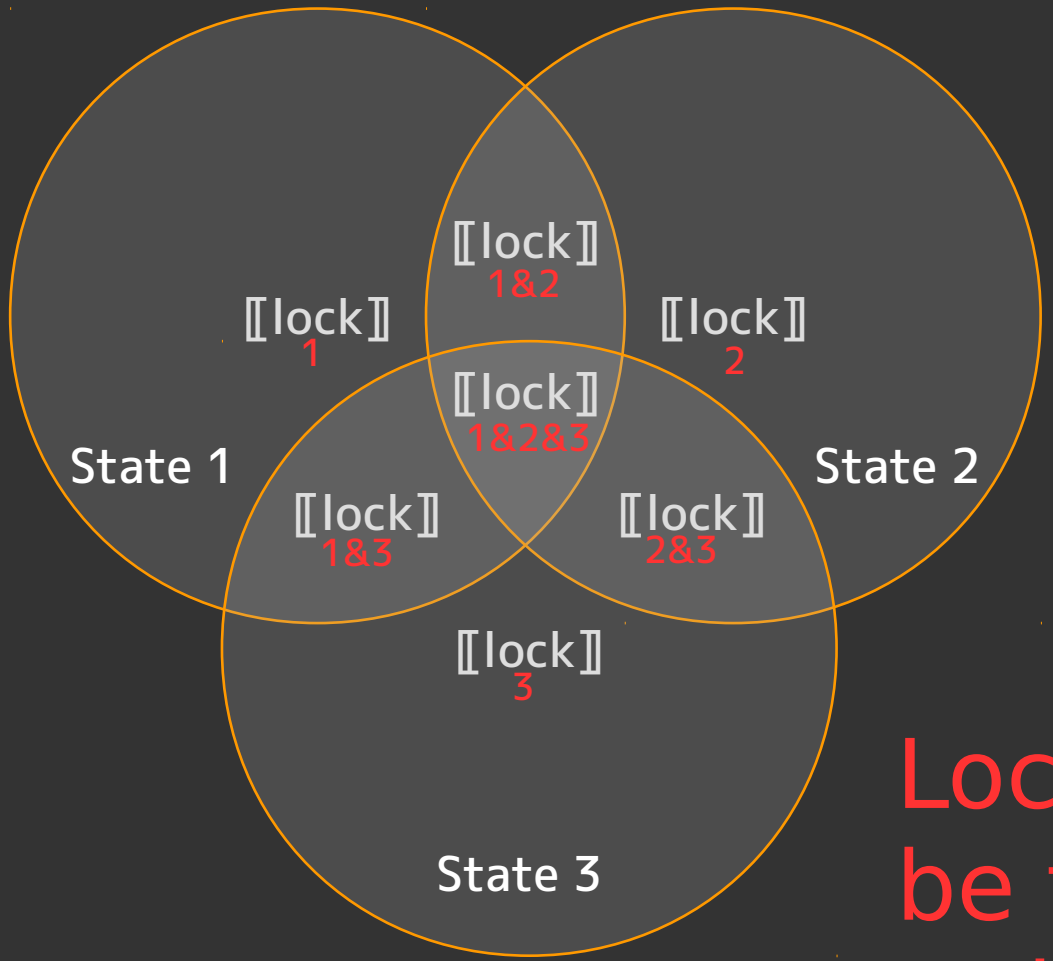
Three locks needed…

- Numbers rise:



$2^n - 1$ locks needed

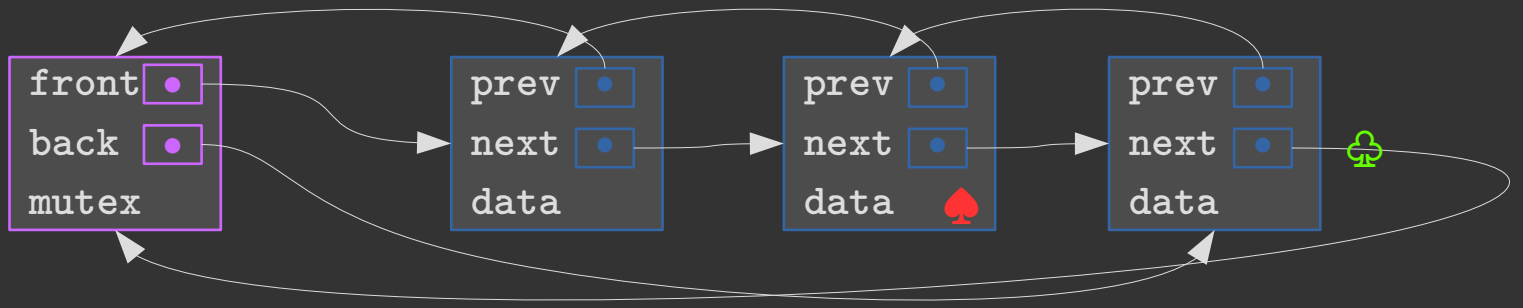
- Alternative not better:



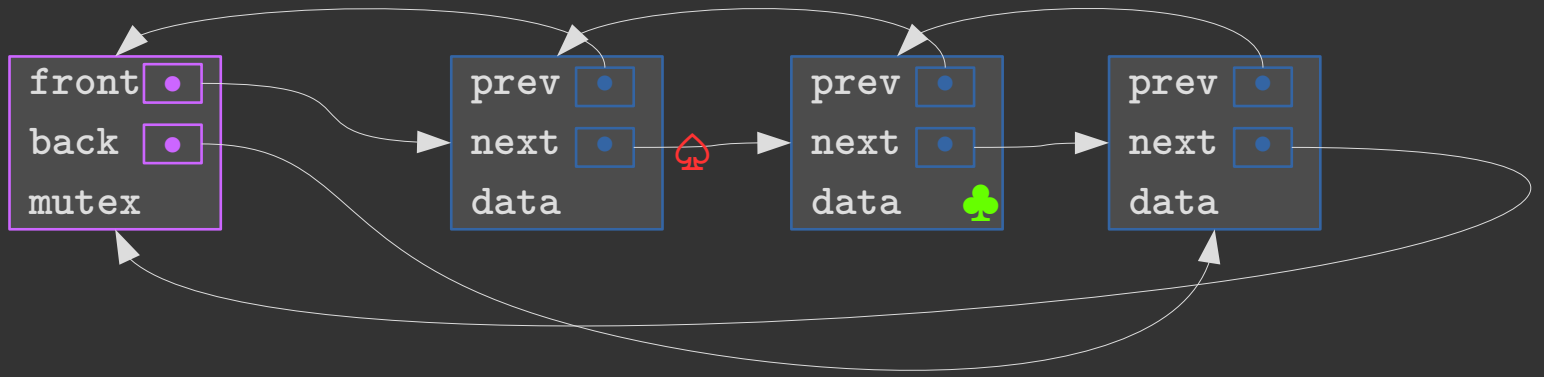
Locks must be taken in order!!!

AB/BA Locking Problem

List 1

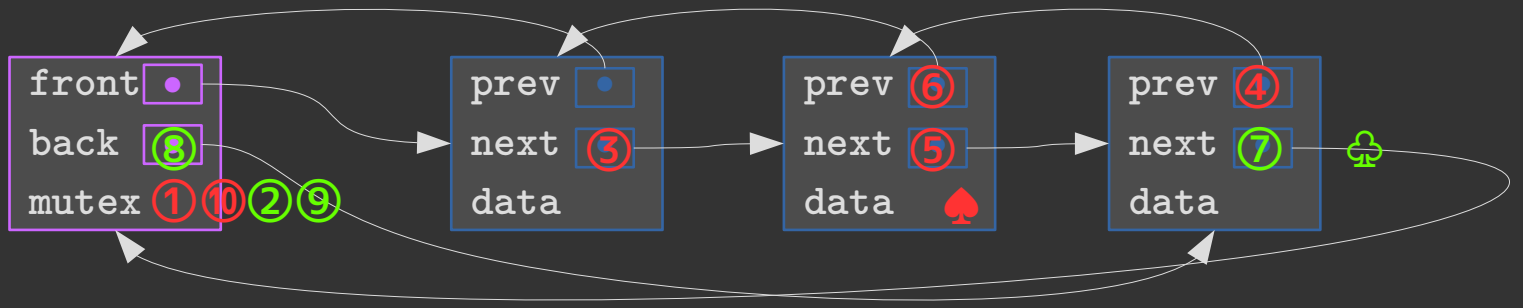


List 2

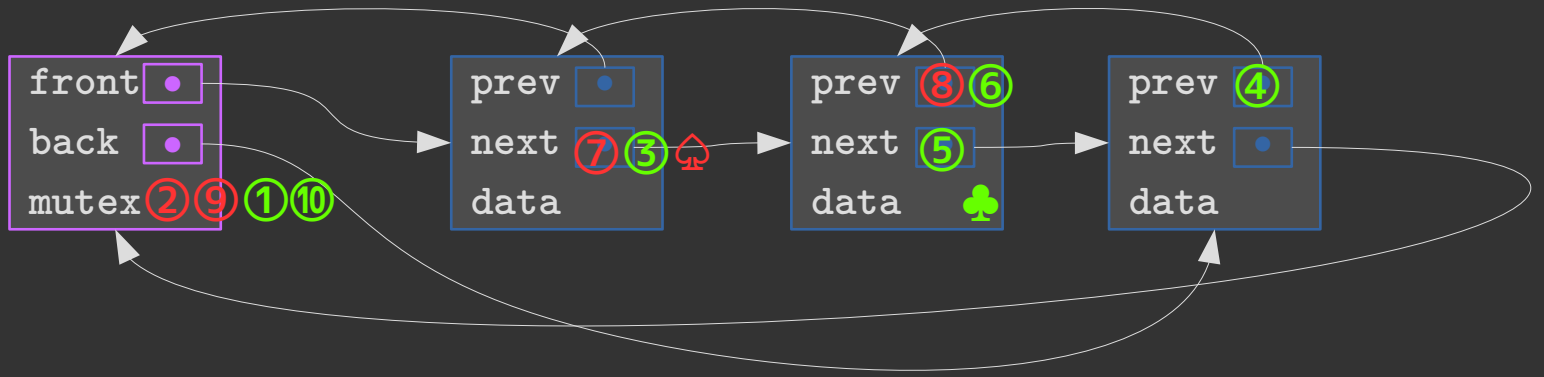


Move ♠ to list 2 and concurrently ♣ to list 1

List 1



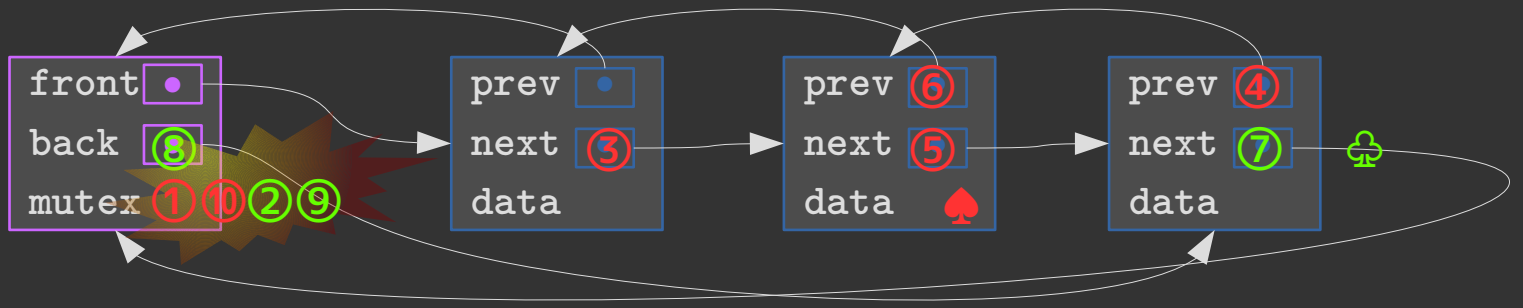
List 2



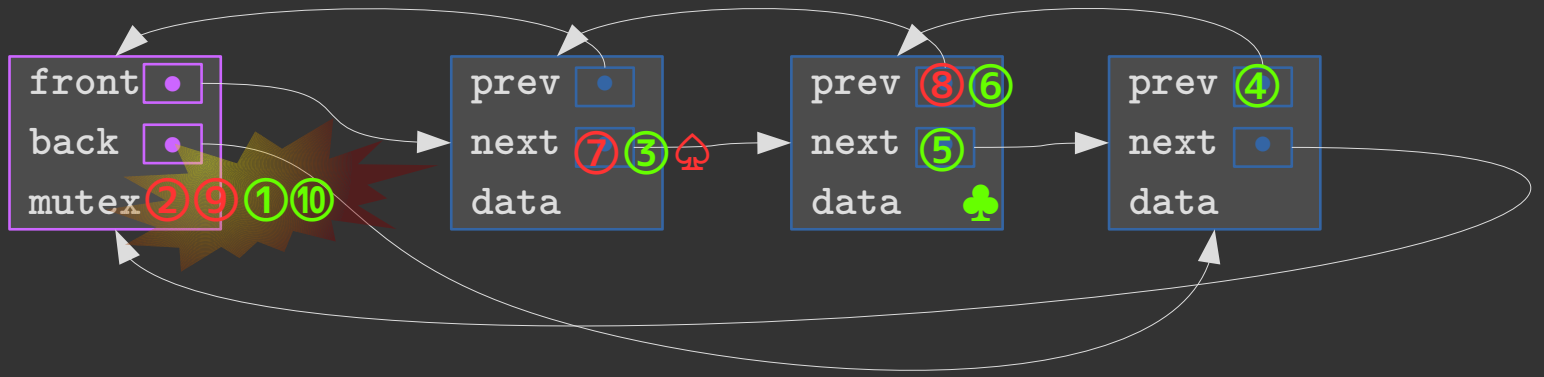
Thread 1 ①→②→③→④→⑤→⑥→⑦→⑧→⑨→⑩

Thread 2 ①→②→③→④→⑤→⑥→⑦→⑧→⑨→⑩

List 1



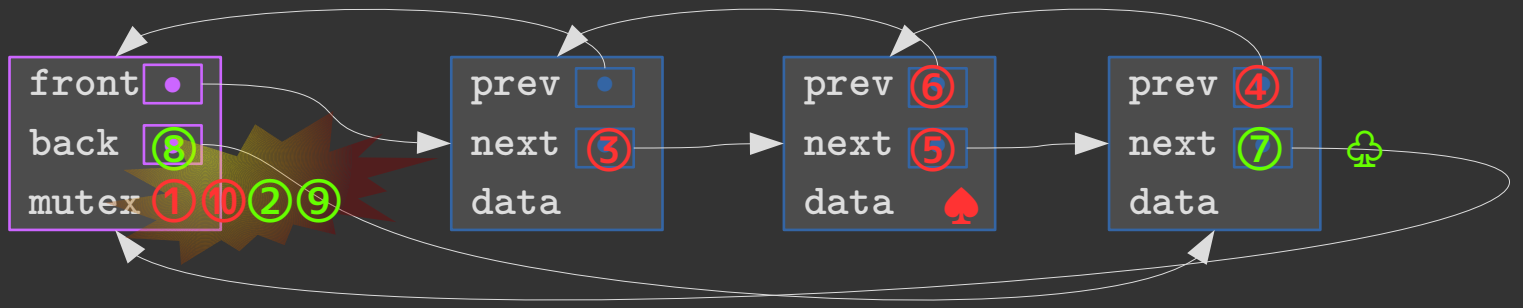
List 2



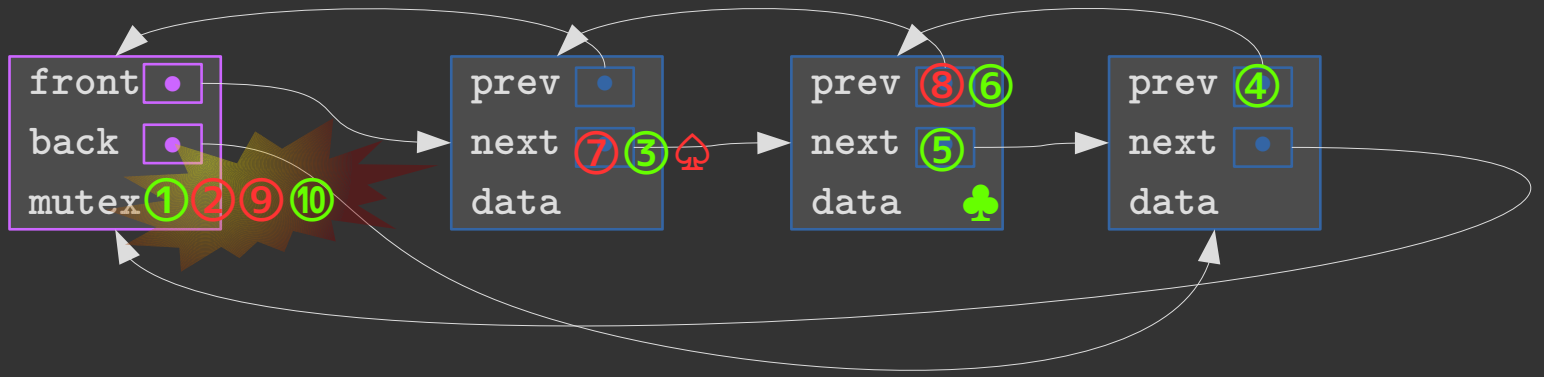
Thread 1 ①→②→③→④→⑤→⑥→⑦→⑧→⑨→⑩

Thread 2 ①→②→③→④→⑤→⑥→⑦→⑧→⑨→⑩

List 1



List 2

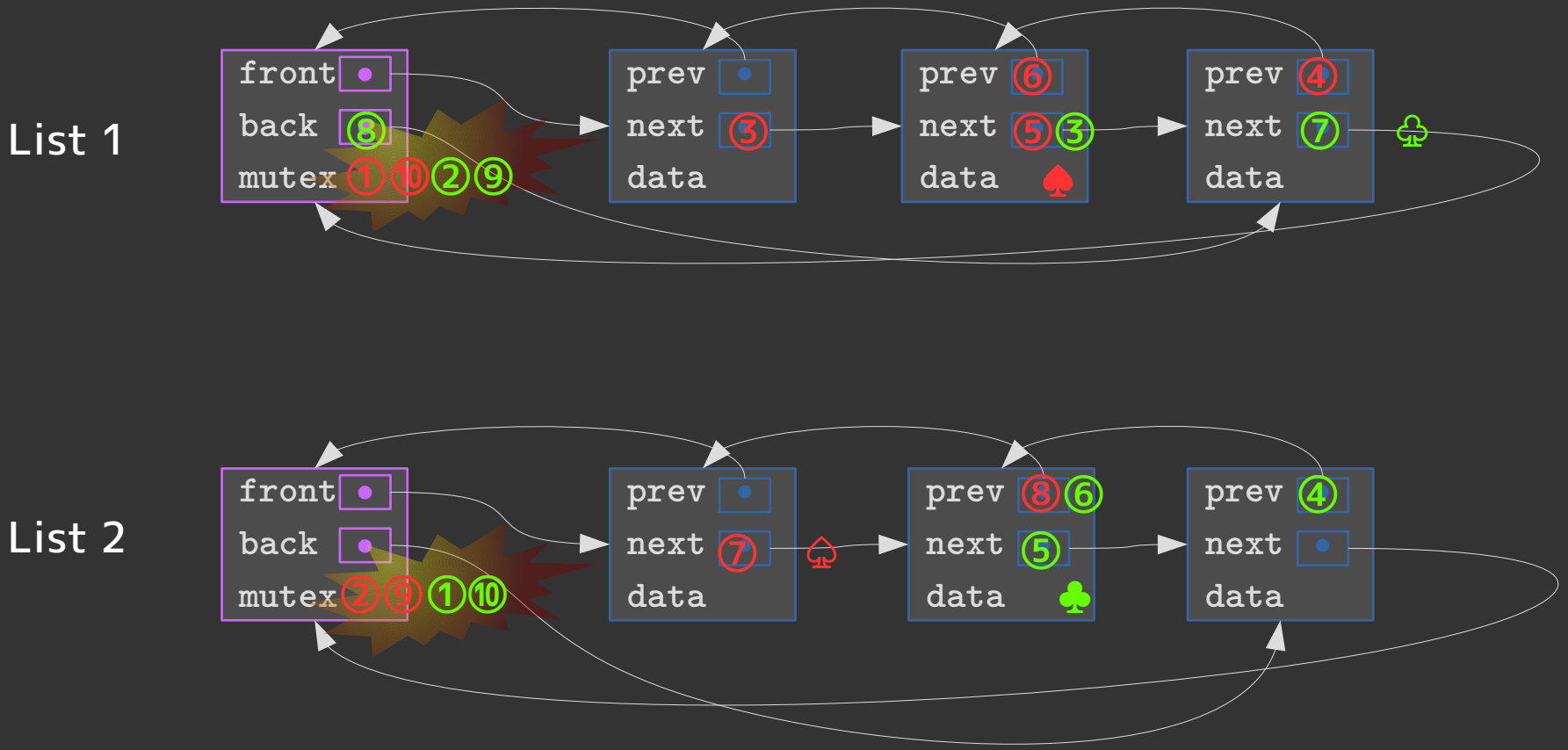


Thread 1 ①→ ②→⋯

Thread 2 ①→ ②→⋯

DEADLOCK!

Transactional Approach



```
if (from == to)
 lists[from].lock();
else if (from < to) {
 lists[from].lock();
 lists[to].lock();
} else {
 lists[to].lock();
 lists[from].lock();
}
```

AB/BA-controlled Locking

```
if (from != to) {
 auto it1 = lists[from].begin();
 auto it2 = lists[to].begin();
 advance(it1, fromidx);
 advance(it2, toidx);
 std::swap(it1, it2);
}
```

Exchange elements from two lists

```
if (from == to)
 lists[from].unlock();
else if (from < to) {
 lists[from].unlock();
 lists[to].unlock();
} else {
 lists[to].unlock();
 lists[from].unlock();
}
```

AB/BA-controlled Unlocking

```
if (from == to)
 lists[from].lock();
else if (from < to) {
 lists[from].lock();
 lists[to].lock();
} else {
 lists[to].lock();
 lists[from].lock();
}
```

AB/BA-controlled Locking

Works with and without HLE support

```
__transaction_atomic {
```

Transaction Start

```
if (from != to) {
 auto it1 = lists[from].begin();
 auto it2 = lists[to].begin();
 advance(it1, fromidx);
 advance(it2, toidx);
 std::swap(it1, it2);
}
```

```
if (from != to) {
 auto it1 = lists[from].begin();
 auto it2 = lists[to].begin();
 advance(it1, fromidx);
 advance(it2, toidx);
 std::swap(it1, it2);
}
```

```
if (from == to)
 lists[from].unlock();
else if (from < to) {
 lists[from].unlock();
 lists[to].unlock();
} else {
 lists[to].unlock();
 lists[from].unlock();
}
```

AB/BA-controlled Unlocking

```
}
```

Transaction End

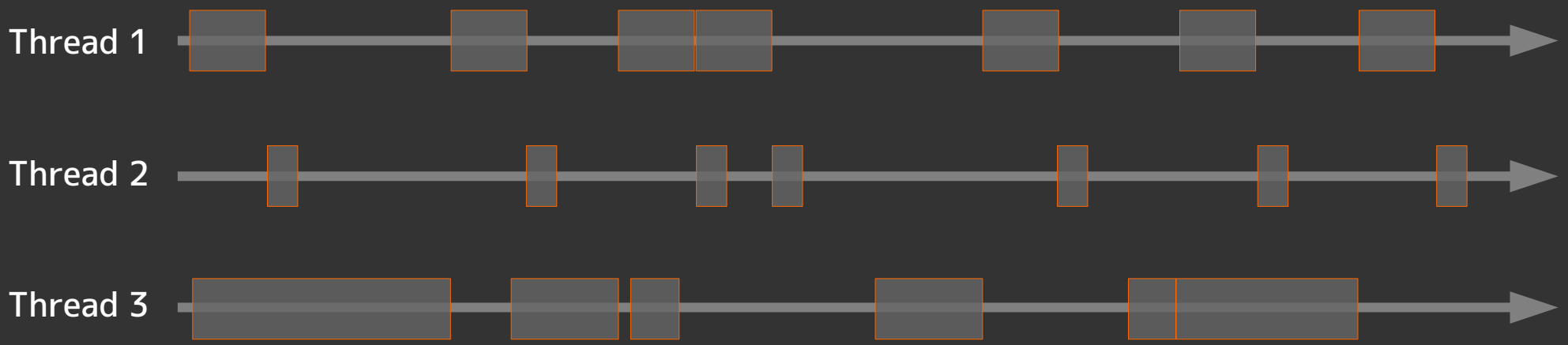
	0.5	0.2	0.1	...	0.0005	0.0002	0.0001
fastrwlock	51,480,108,625	6,273,517,692	4,463,060,823		2,402,605,983	2,548,314,560	2,484,088,100
fastrwlock_hle	53,090,237,967	24,260,399,475	4,288,488,921		911,502,618	1,724,446,655	918,704,241
fastrwlock_nocheck	52,468,287,048	22,124,187,852	4,448,158,733		2,457,996,585	2,571,891,030	2,410,316,807
fastrwlock_nocheck_hle	21,227,164,646	802,029,192	1,092,841,711	...	446,033,670	1,275,359,233	575,373,313
futex	3,529,717,186	2,856,581,277	2,984,845,701		2,685,044,105	2,802,146,684	2,564,183,641
futex_hle	546,008,603	394,687,068	381,043,932		346,546,816	360,030,896	344,344,698
mutex	6,282,423,300	5,718,409,931	5,583,734,371		5,617,024,609	5,392,923,259	5,005,785,005
rwlock	53,708,141,989	47,708,113,002	12,101,444,247		9,147,748,366	9,072,847,111	8,271,694,730

	50%	20%	10%	...	0.05%	0.02%	0.01%
fastrwlock	4.15%	86.85%	63.12%		73.74%	71.91%	69.97%
fastrwlock_hle	1.15%	49.15%	64.56%		90.04%	80.99%	88.89%
fastrwlock_nocheck	2.31%	53.63%	63.24%		73.13%	71.65%	70.86%
fastrwlock_nocheck_hle	60.48%	98.32%	90.97%	...	95.12%	85.94%	93.04%
futex	93.43%	94.01%	75.33%		70.65%	69.12%	69.00%
futex_hle	98.98%	99.17%	96.85%		96.21%	96.03%	95.84%
mutex	88.30%	88.01%	53.65%		38.60%	40.56%	39.48%

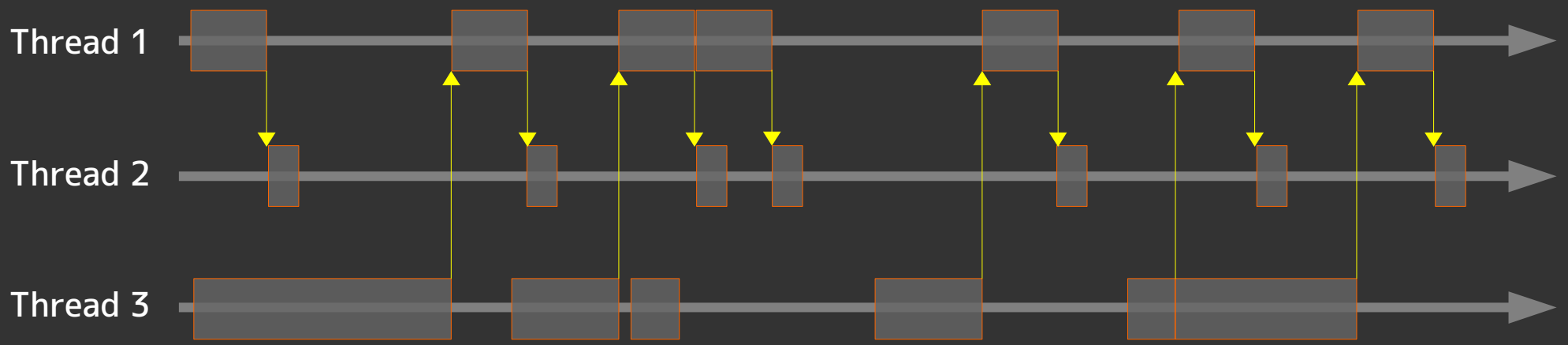
HLE Speed-Up	50%	20%	10%	...	0.05%	0.02%	0.01%
fastrwlock	-3.13%	-286.71%	3.91%		62.06%	32.33%	63.02%
fastrwlock_nocheck	59.54%	96.37%	75.43%	...	81.85%	50.41%	76.13%
futex	84.53%	86.18%	87.23%		87.09%	87.15%	86.57%

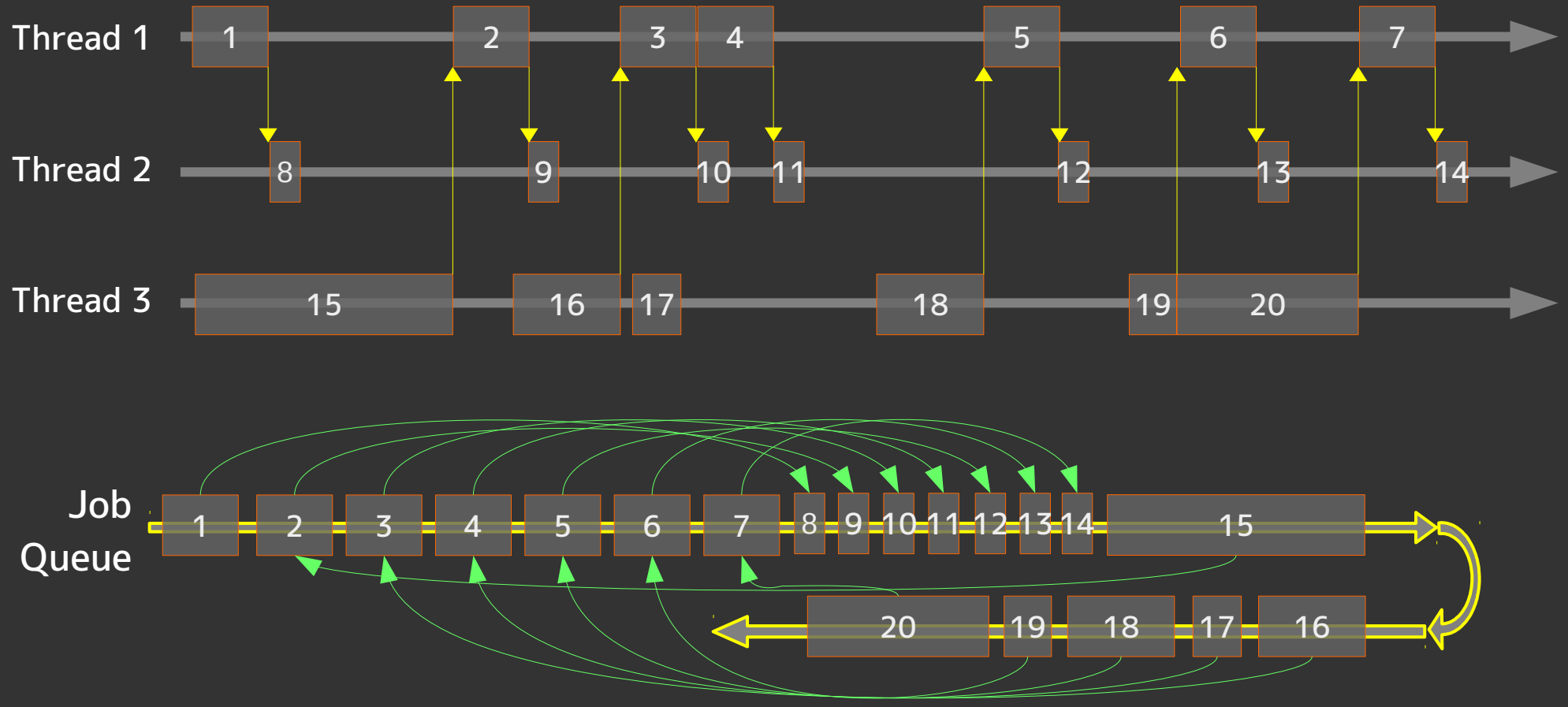
Complete table at http://www.akkadia.org/drepper/hsw.html

Expressing Parallelism



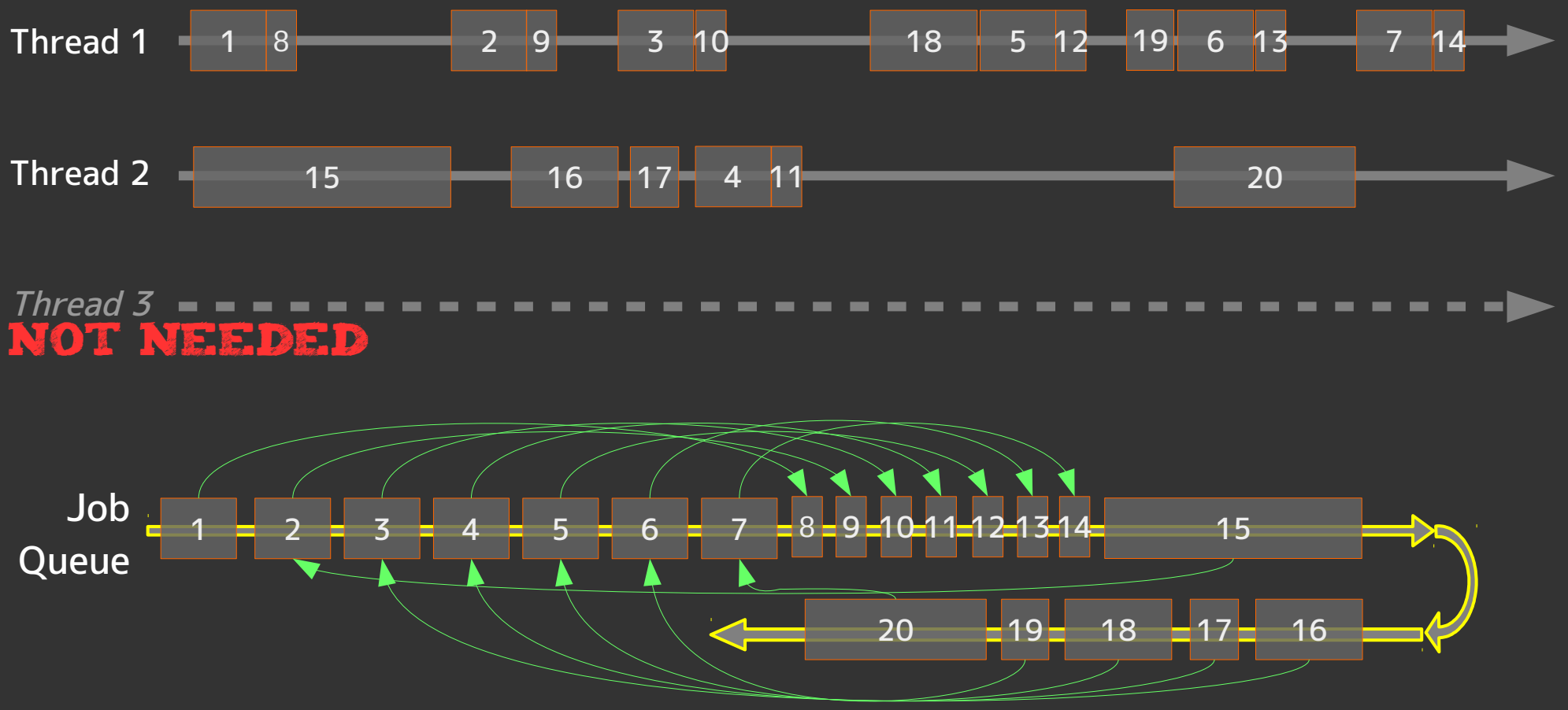
- Threads explicitly created
- Started based on event or availability
- Stopped on contention or missing resource


Thread 1
Thread 2
Thread 3



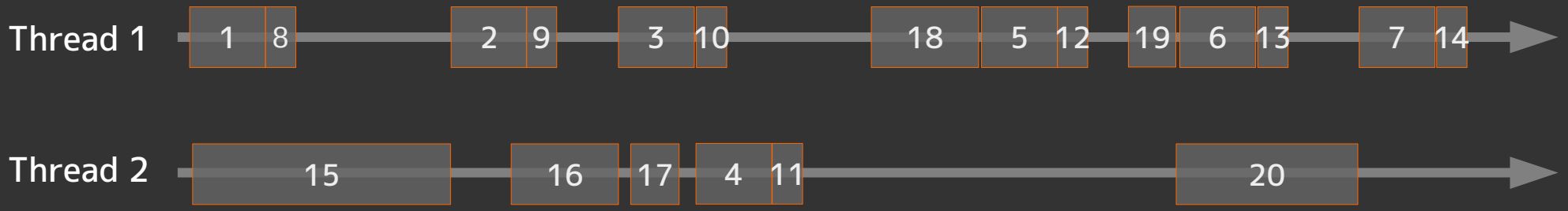
Queue up and record dependencies

Schedule by taking from the job queue:



Queue up and record dependencies

Benefits



- No (explicit) locking needed
- Parallelism created implicitly via thread pool
- More efficient utilization
  - Can be much better with more information and special scheduler

Haskell Approach

- Lazy Evaluation:

  `[1..]` → `[1,2,3,4,5,6,` and so on⋯`]`

- Infinite list
- Example use:

```
Prelude> let mystery = 0 : scanl (+) 1 mystery

head (drop 13) mystery
```

C++ Approach

- Futures:

```
template<class F, class... Args>
future<typename result_of<F(Args...)>::type>
async(F&& f, Args&&... args);
```

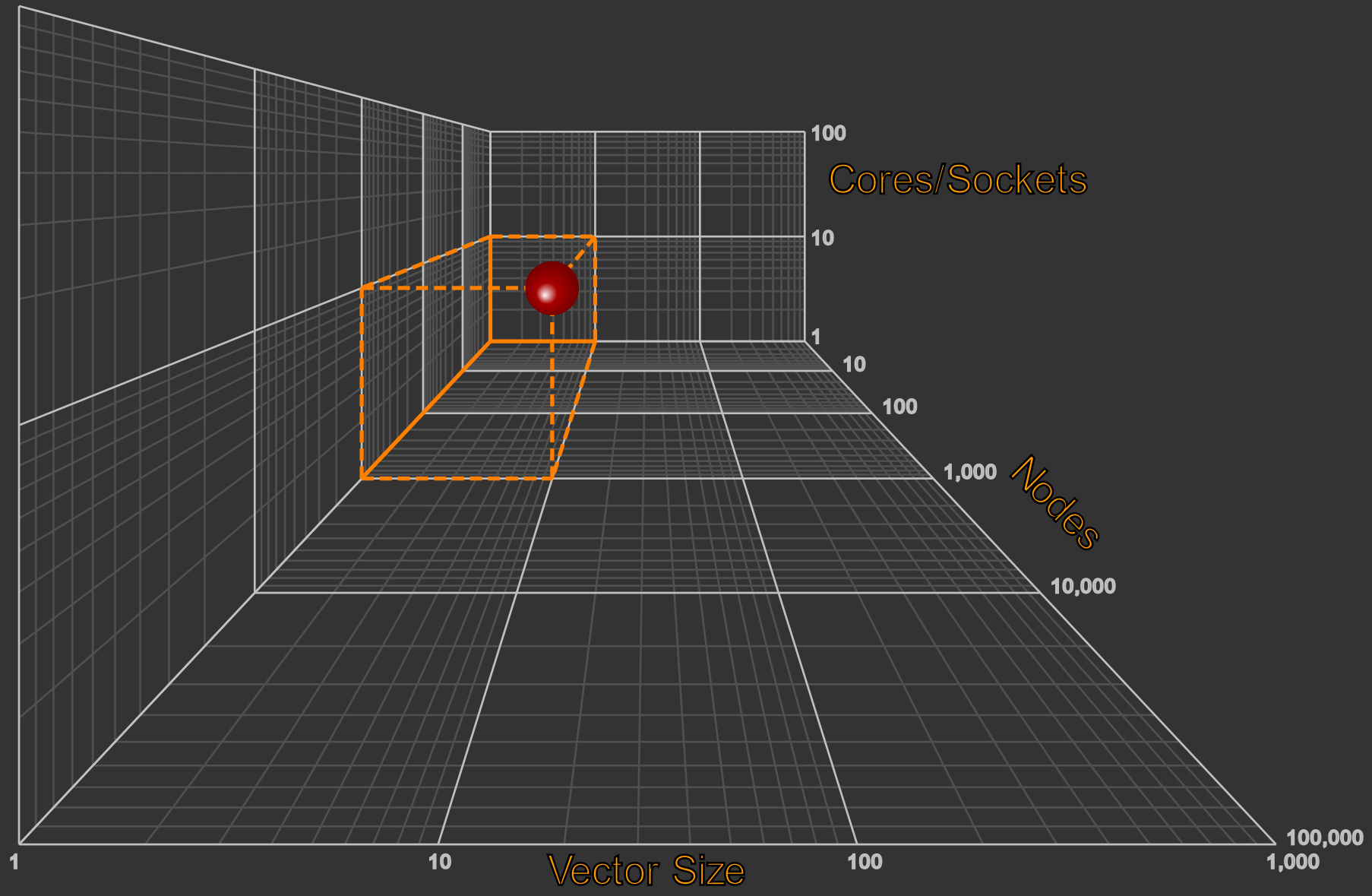
- Function `f` submitted to be executed
  - Self-contained, arguments plus potentially closure
  - State represented by `future` object
  - Result available with `future::get()`

- Finer control:

```
template<class F, class... Args>
future<typename result_of<F(Args...)>::type>
async(launch policy, F&& f, Args&&... args);
```

- Extra argument for fine control
  - `launch::async` — run as-if in separate thread
  - `launch::deferred` — do not run until needed

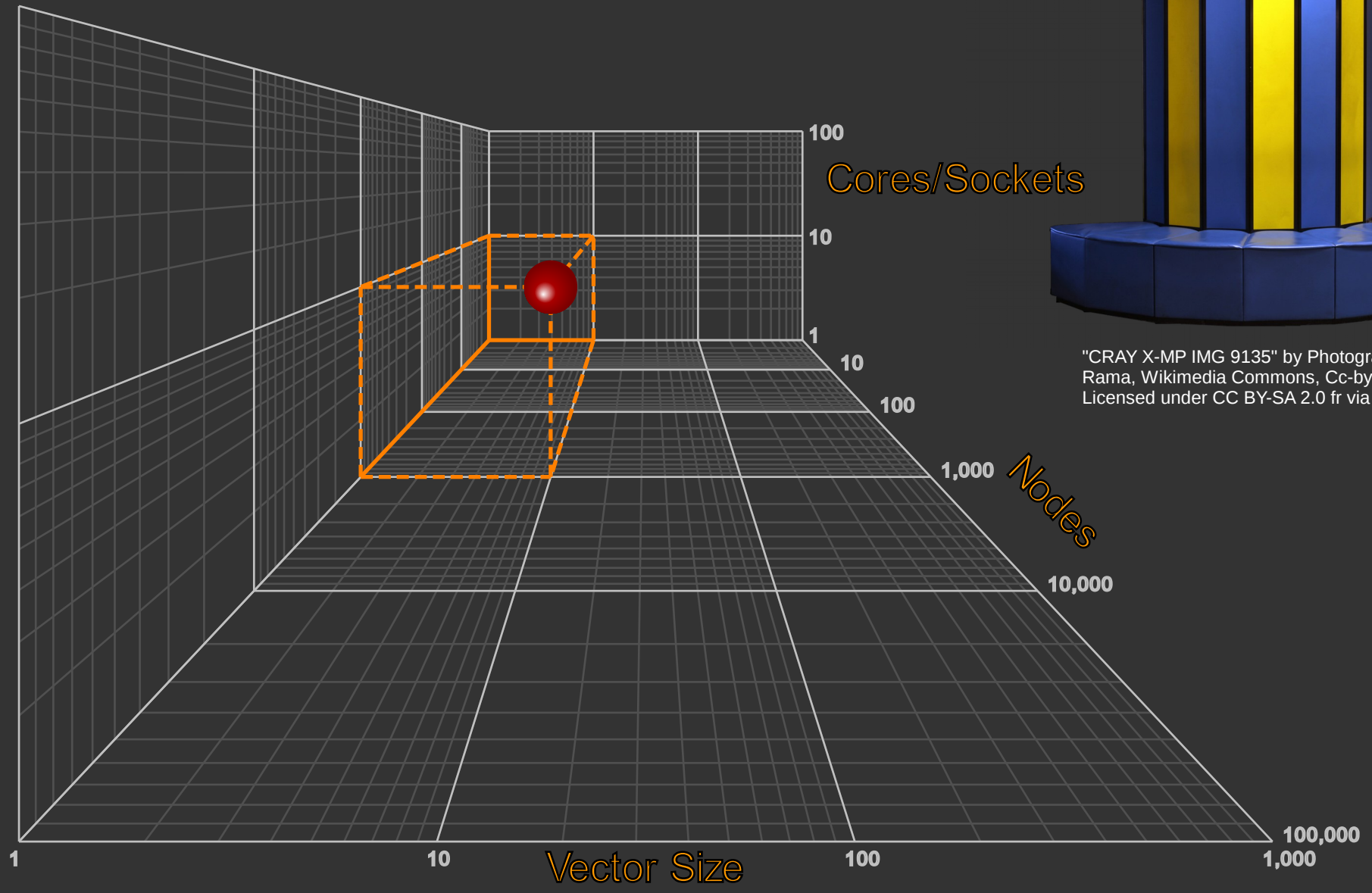
Third Dimension



History I

"CDC 6600.jc" by Jitze Couperus - Flickr: Supercomputer - The Beginnings. Licensed under CC BY 2.0 via Commons

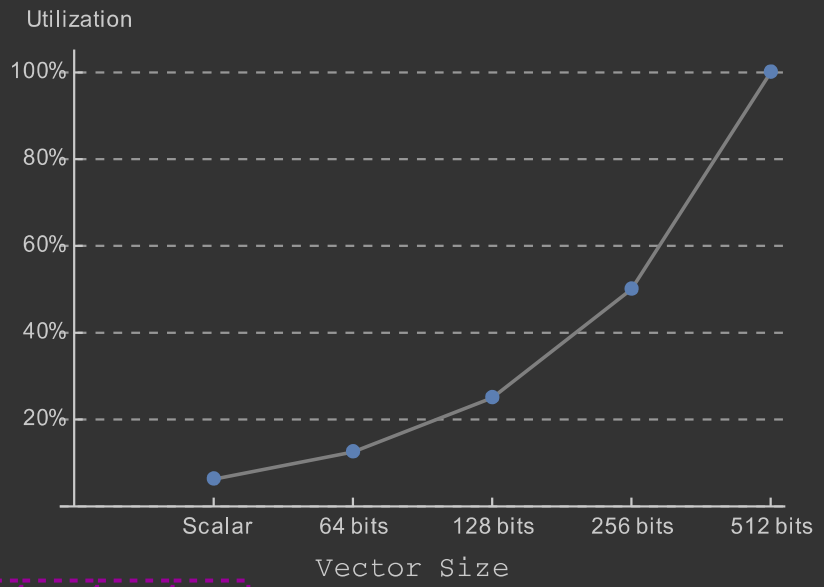
History II

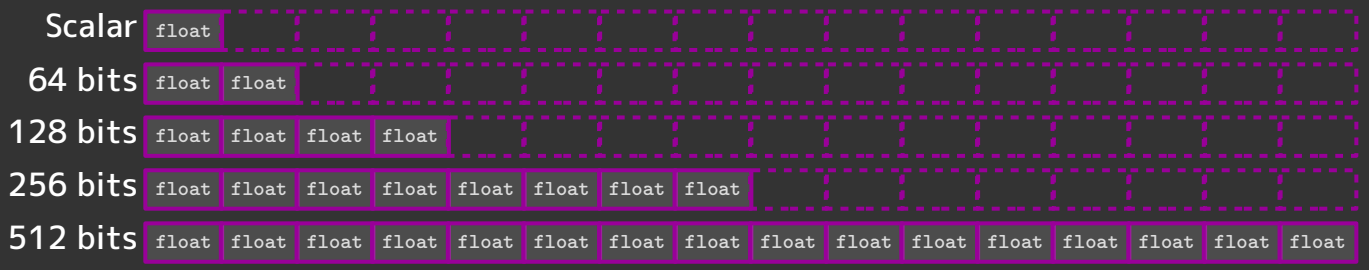


"CRAY X-MP IMG 9135" by Photograph by Rama, Wikimedia Commons, Cc-by-sa-2.0-fr. Licensed under CC BY-SA 2.0 fr via Commons

Scalar vs SIMD

- All modern processors provide SIMD capabilities





Effects of Vectorization

Operation	Energy	Scale
8-bit int add	0.03pJ	1
32-bit int add	0.10pJ	3
8-bit int mult	0.20pJ	6
32-bit int mult	3.00pJ	100
16-bit FP mult	1.00pJ	30
32-bit FP mult	4.00pJ	133
RISC instruction	>70.00pJ	>2300

The PicoSecond is Dead Long Live the PicoJoule, Christos Kozyrakis

Name	Energy	Time	Power
crafty	6.94%	6.99%	-0.05%
parser	0.12%	0.08%	0.04%
mcf	0.58%	0.63%	-0.05%
vortex	-3.72%	-3.25%	-0.49%
gcc	-1.17%	-1.51%	0.34%
bzip2	-2.34%	-1.82%	-0.53%
art	0.66%	0.68%	-0.02%
ammp	-0.13%	-0.03%	-0.10%
equake	-0.01%	-0.94%	0.94%

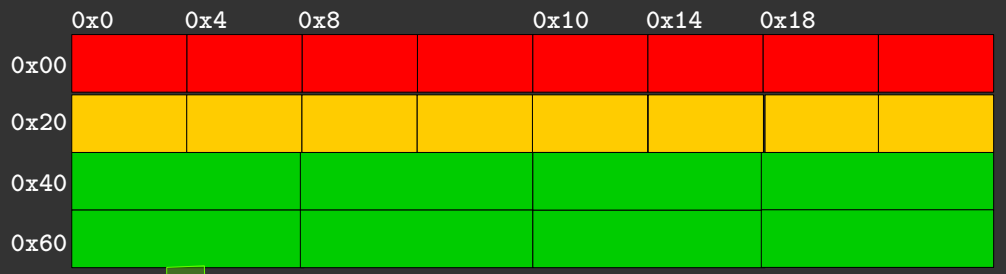
The Effect of Compiler Optimization on Pentium 4 Power Consumption, J.S. Seng and D.M. Tullsen

SIMD Memory Access

Vector of structures vs structure of vectors

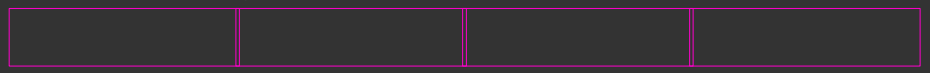
```
struct transaction
{
 unsigned id;
 time_t time;
 double total;
};
```

```
move VR,(AR)
```



Linear Load

Vector Register



Address Register

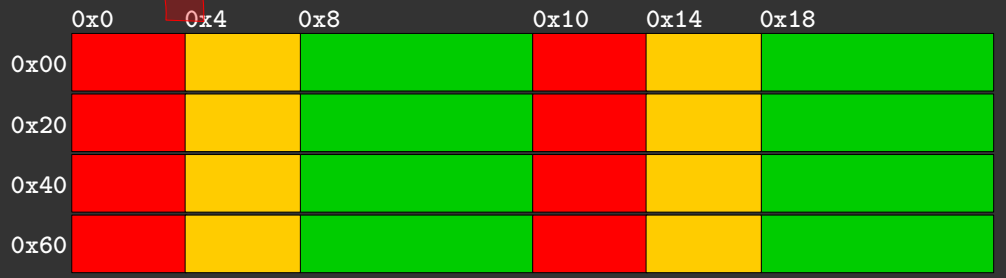
0x00

Index Register

0x8	0x18	0x28	0x38

Gather Load

```
move VR,(AR,IR)
```



Conditional Expressions

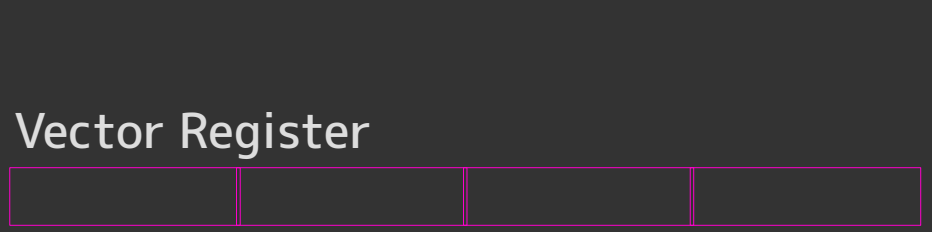
```
double value(const vector<postion_t>& ps, double r) {
 double res = 0.0;
 for (auto& p : ps) {
  auto d1 = (log(p.S / p.K) + (r + p.std * p.std) * p.t) / (p.std * sqrt(p.t));
  auto p1 = N(d1);
  auto p2 = N(d1 - p.std * sqrt(p.t));
  auto dK = p.K * exp(-r * p.t);
  if (p.put)
   res += dK * (1.0 - p2) - p.S * (1.0 - p1);
  else
   res += p.S * p1 - dK * p2;
 }
 return res;
}
```

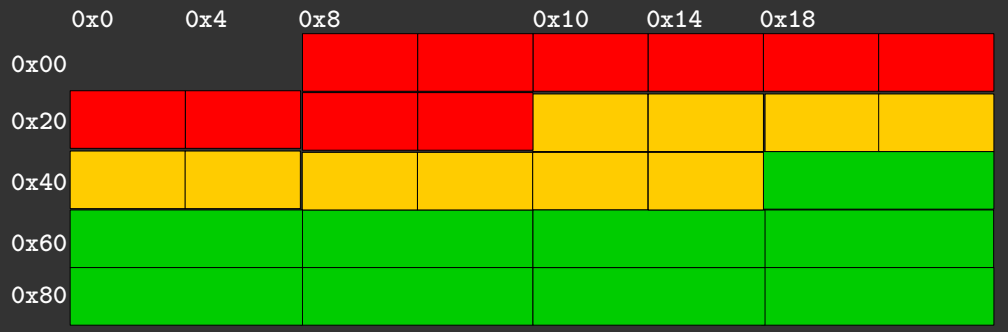
- Vector versions of log, exp, and sqrt available
- Condition might be different for each loop iteration
  - Rewrite without conditional

```
double value(const vector<postion_t>& ps, double r) {
 double res = 0.0;
 for (auto& p : ps) {
  auto d1 = (log(p.S / p.K) + (r + p.std * p.std) * p.t) / (p.std * sqrt(p.t));
  auto p1 = N(d1);
  auto p2 = N(d1 - p.std * sqrt(p.t));
  auto dK = p.K * exp(-r * p.t);
  mask_t mask = p.put;
  auto cp1 = p1 - ((mask & 1.0) | (!mask & 0.0));
  auto cp2 = p2 - ((mask & 1.0) | (!mask & 0.0));
  res += p.S * cp1 - dK * cp2;
 }
 return res;
}
```

- Mask operations part of all vector processors
  - This is pseudo code

Startup and Finish Costs





- Avoid unaligned memory accesses
- Vectors of size not multiple of register size
- Handle start and/or end in pieces

Programming Differences

Arm:

```
float32x4_t vaddq_f32(float32x4_t a, float32x4_t b)
```

x86:

```
__m128 _add_ps(__m128 a, __m128 b)
```

Power:

```
vector float vec_add(vector float a, vector float b)
```

gcc:

```
typedef float floatx4_t __attribute__((vector_size(16)));


floatx4_t add(floatx4_t a, floatx4_t b) {
 return a + b;
}
```

Summary

- Parallelism essential
- The “old ways” do not work/scale
- Hardware details important
- Extensive evaluation of results needed
- Modern programming technologies
  - Help using all kinds of hardware
  - While preserving single code base

Questions?