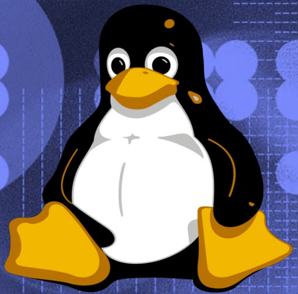
Scuola Superiore Sant'Anna RETIS

# Synchronization and Scalability in the Macho Multicore Era

**Paul E. McKenney**
**IBM Distinguished Engineer & CTO Linux**
**Linux Technology Center**



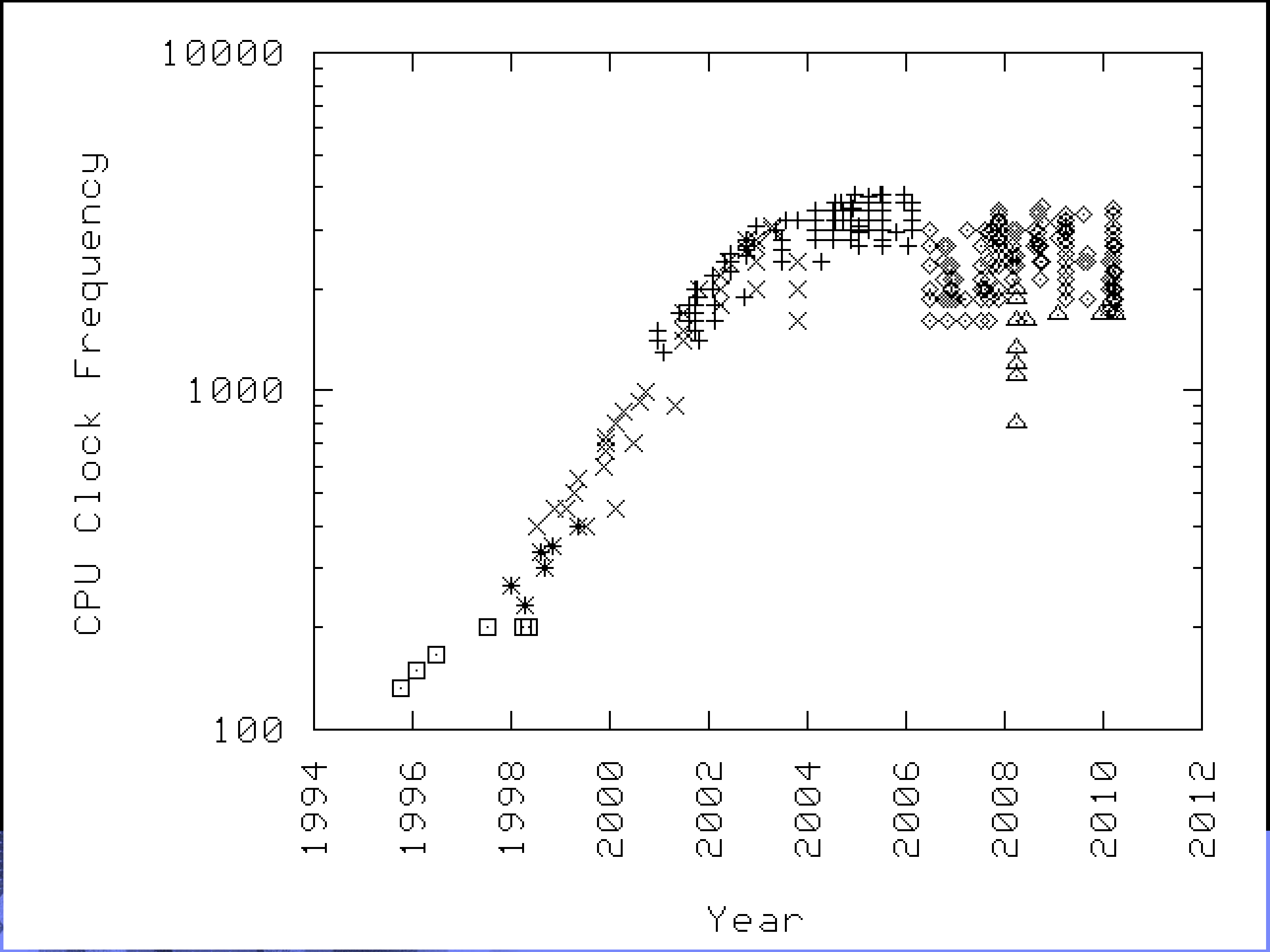
April 21, 2010

## Overview

- Why Parallel Programming?
- A Brief History of Parallel Programming
- Trends in Parallelism
- Performance of Synchronization Mechanisms
- System Hardware Structure
- Parallel Programming Principles
- Parallel Programming Exercise
- The Role of Non-Technical Issues
- Summary and Conclusions

## Why Parallel Programming?

## Why Parallel Programming? (Party Line)



## Why Parallel Programming? (Reality)

- Parallelism is one performance-optimization technique of many
  - Hashing, search trees, parsers, cordic algorithms, ...
- But the kernel is special
  - In-kernel performance and scalability losses cannot be made up by user-level code
  - Therefore, if any user application is to be fast and scalable, the portion of the kernel used by that application must be fast and scalable
- System libraries and utilities can also be special
- As can database kernels, web servers, ...

## Why Parallel Programming? (More Reality)

- There Are Other Uses For Transistors
  - Cache
  - DRAM
  - Accelerators: FP, crypto, compression, XML, ...
  - Networking hardware
  - Storage hardware: Flash, CD/DVD, disk, ...
  - Graphical display hardware
  - Audio/video input hardware
  - GPS hardware]
- Or the chips could get smaller

## Why Parallel Programming? (Even More Reality)

- If computer systems don't improve rapidly, computers not be replaced as frequently
  - This is a matter of serious concern for companies whose revenue is driven by sales of new computers
- Replacement was driven by CPU clock rate
- Macho multicore seen by some as new driver of computer sales
- Other possible scenarios:
  - Power efficiency drives new sales (new laptop!)
  - New applications and form factors drive new sales
  - Computers become a durable good

## A Brief History of Parallel Programming

## A Brief History of Parallel Programming

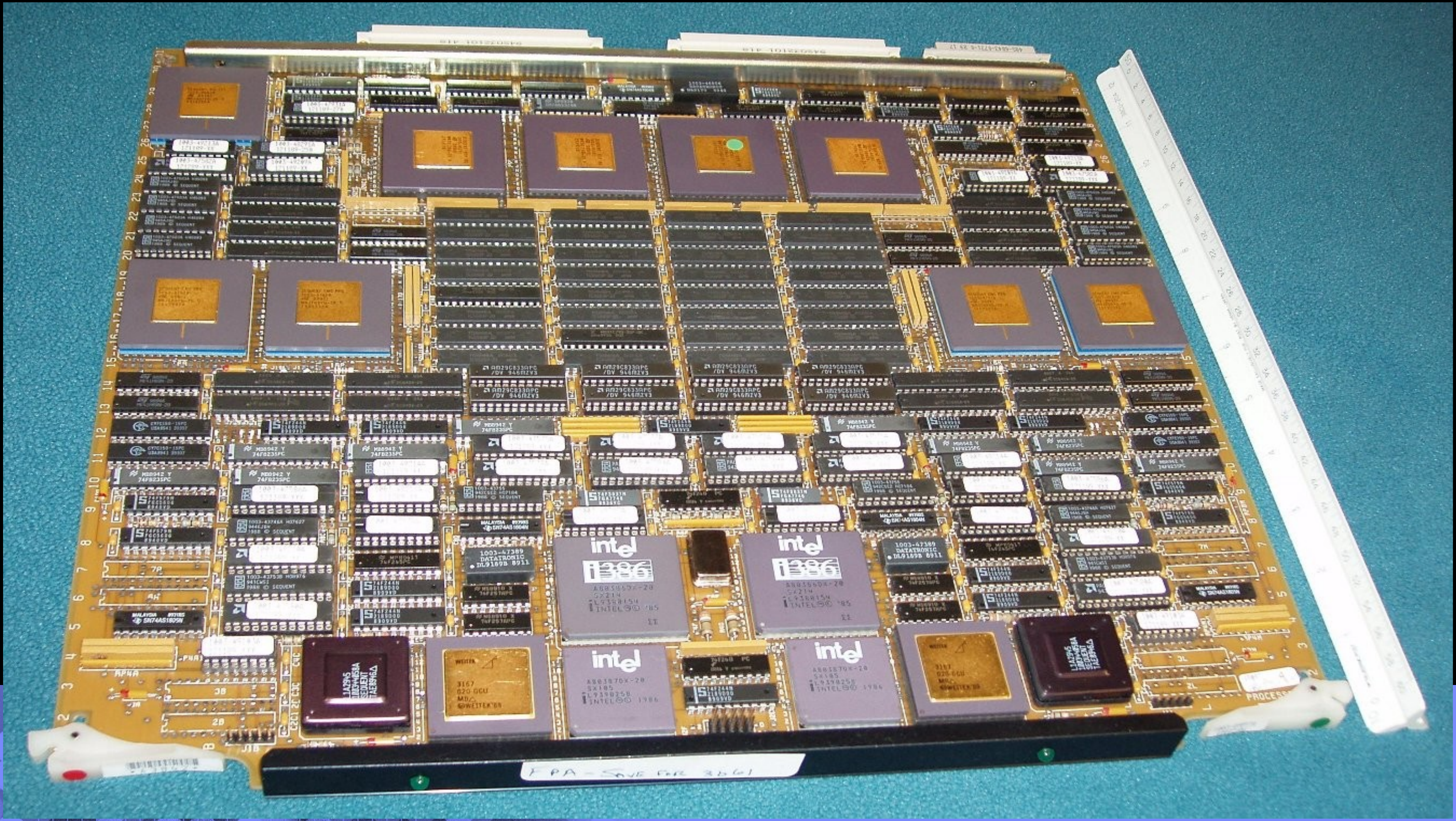
- Analog computers inherently parallel is 50s and earlier
- CDC3300 had RAD and SDL instructions in 60s
- IBM Mainframe had “I/O channels” in 60s
- CDC6600 has PPUs in 60s
- Dijkstra's locking algorithm in 60s
- Dijkstra's CSP in 60s
- Dijkstra's “Dining Philosophers Problem” in 70s
- Courtois, Hymans, & Parnas rwlock in 70s
- Hoare monitors in 70s
- Lamport's locking algorithm in 70s
- Relational database research in 70s
- Production parallel systems in 80s (driven by HPC and databases)
- Data locking in 80s
- pthreads in 80s and 90s
- Queued locks for high contention in 90s (not good for low contention)
- Efficient parallel memory allocators in 90s
- RCU in 90s
- NUMA-aware locking in 90s
- More than 200 new parallel-programming languages/environments in 90s (!!!)
- Adaptive simple/NUMA-aware locking in 00s
- Production parallel realtime operating systems in 00s
- Realtime RCU in 00s

## A Brief History of Parallel Programming

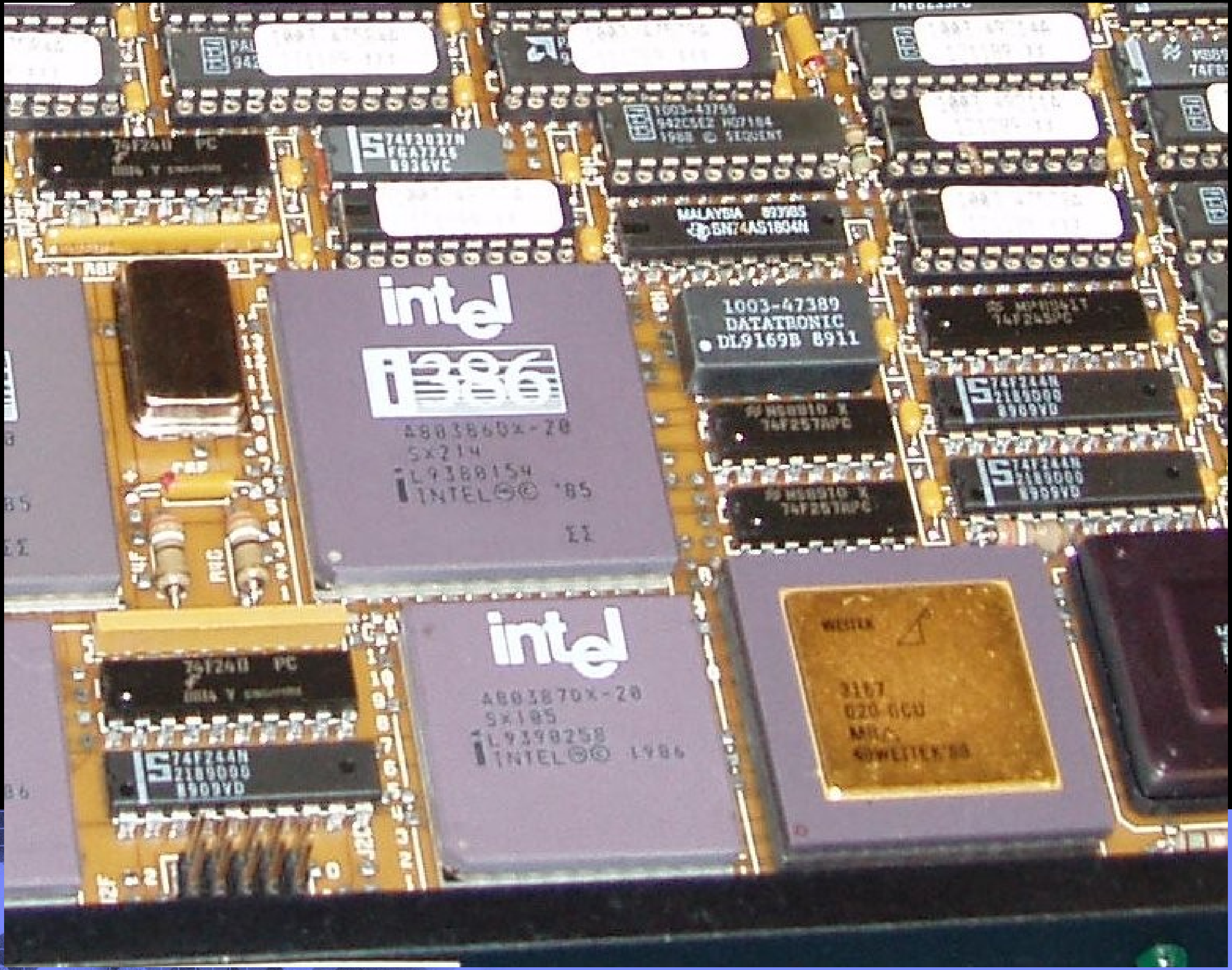
- **How could there *possibly* be anything new to discover in the decades-old field of parallel processing???**

## Trends in Parallelism

## 1989 Sequent Symmetry Model C CPU Board



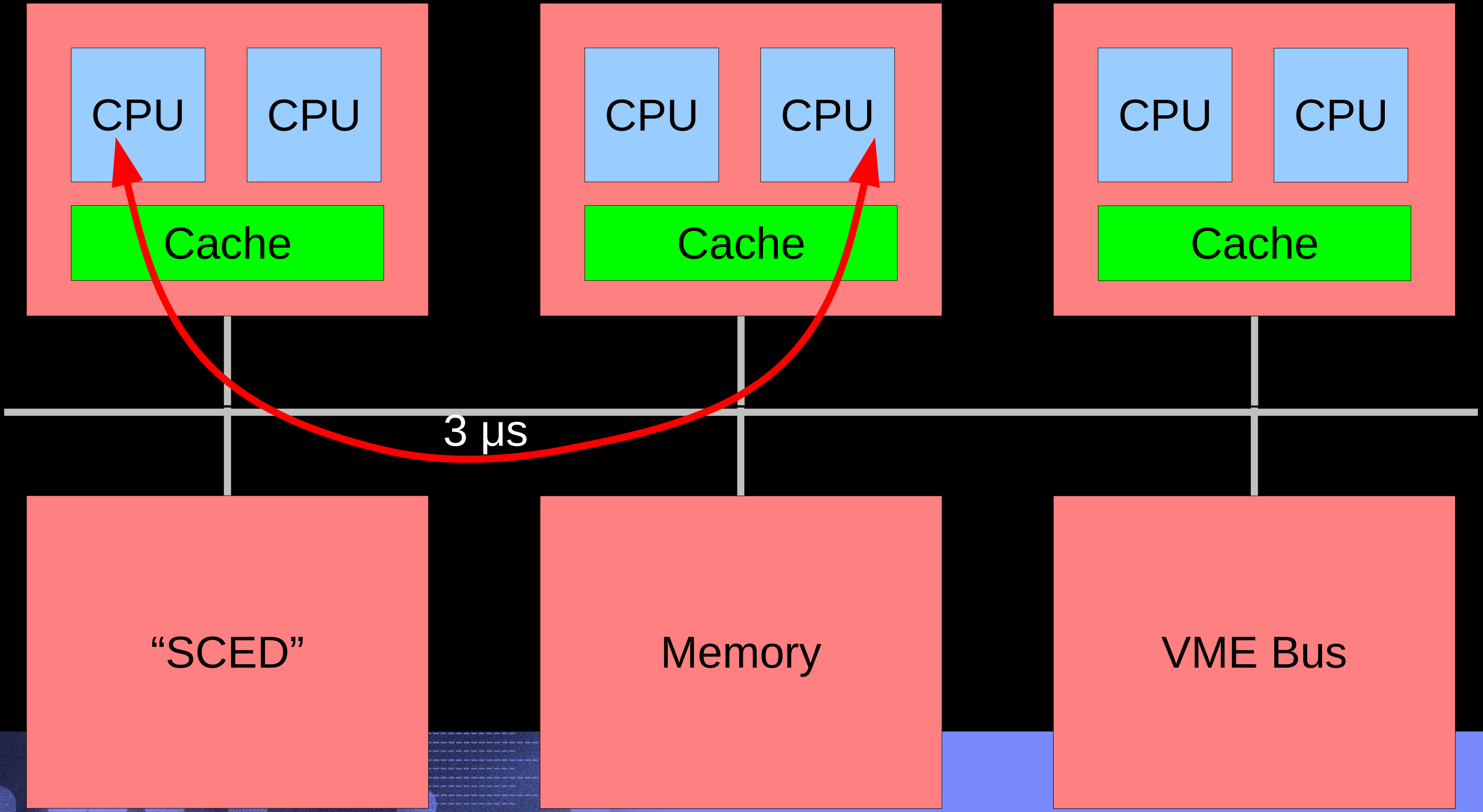
## 1989 Sequent Symmetry Model C 20MHz CPU



## 1989 Sequent Symmetry Computer System

- Two 20MHz 80386 CPUs per CPU board
  - No cmpxchg instruction, no xadd instruction
  - Cheapest instructions consume three cycles
  - Separate Weitek FPAs deliver 1MFLOP each
  - List price roughly $60K per board
    - 5x price/performance advantage over competitors
- Off-chip cache
- 53MB/s common bus
- 10Mbps Ethernet
- Tens of MB of memory
- Tens of GB of disk storage in full rack

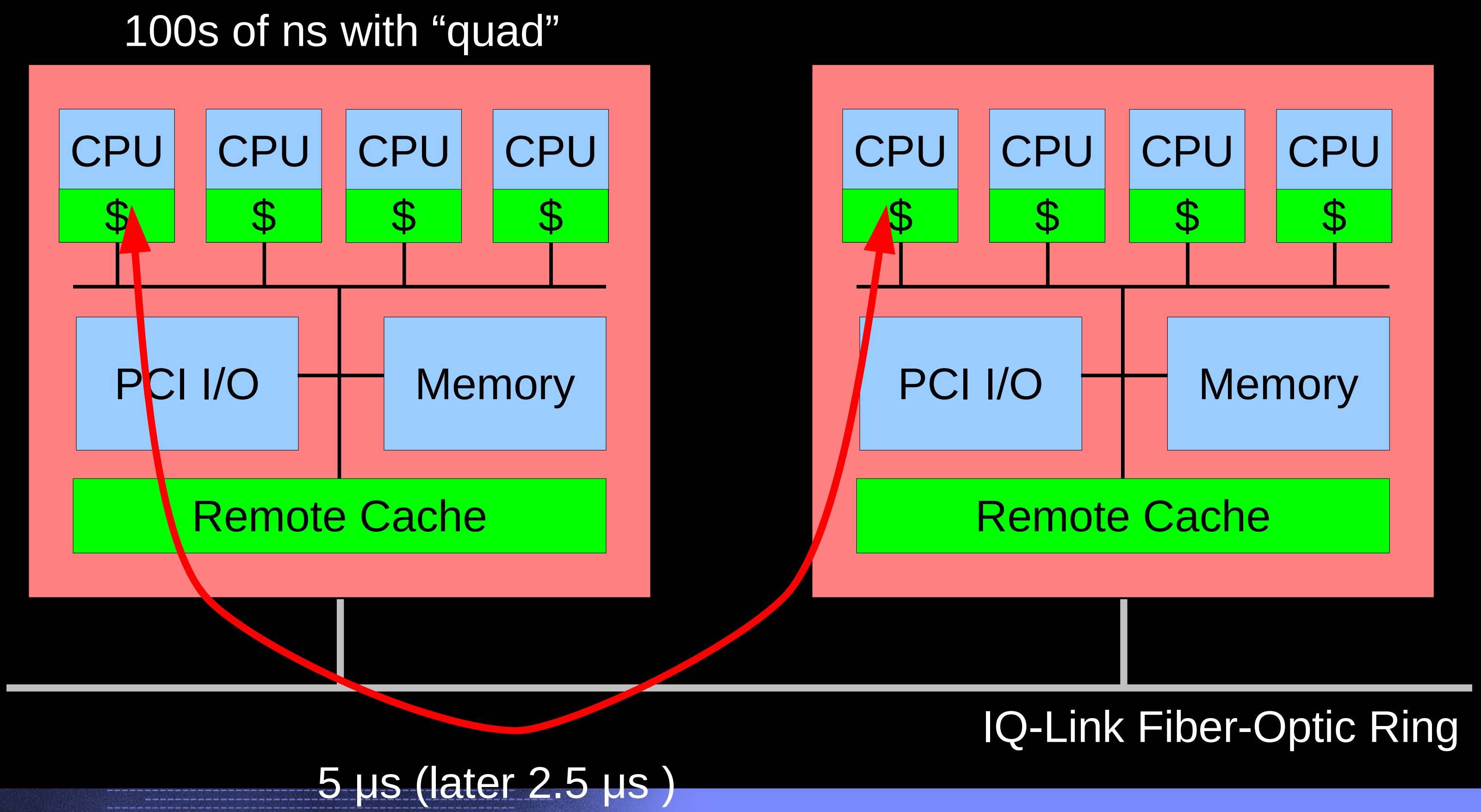
## 1989 Sequent Symmetry Architecture



## 1996 Sequent NUMA-Q Computer System

- Four 180MHz Pentium Pro CPUs per “Quad”
  - cmpxchg, xadd, cmpxchg8b, ...
  - Single-cycle instructions
  - On-chip floating-point
  - On-chip cache (2MB)
- Off-chip remote cache (128MB)
- Gbps SCI fiber-optic ring interconnect
- 100Mbps Ethernet
- Tens of GB of memory
- Hundreds of GB of disk storage in full rack

## 1996 Sequent NUMA-Q Architecture



## 1999 Sequent NUMA-Q Computer System

- Four 900MHz Pentium CPUs per “Quad”
  - cmpxchg, xadd, cmpxchg8b, ...
  - Single-cycle instructions, on-chip floating-point
    - ~2,000 CPU cycles per remote cache miss
  - On-chip cache (2MB)
  - 3GB/s I/O bandwidth per quad full DBMS processing
  - Two EMC Symmetrix boxes per quad to keep up
- Off-chip remote cache (128MB)
- Gbps SCI fiber-optic ring interconnect
- 100Mbps Ethernet
- Tens of GB of memory
- TB of disk storage in full rack

## 2010 IBM Power 7 Computer System

- 8 cores per octant, 4.4GHz, 4 threads/core
  - larx/stcx, isync, lwsync, eieio, sync
  - Single-cycle instructions, on-chip floating-point
  - 32 octants per system for 1024 CPUs to Linux
  - NUCA architecture
- Gbps SCI fiber-optic ring interconnect
- 10Gbps Ethernet (100s of adapters)
- Tens of TB of memory
- More disk than you can shake a stick at

## But Much More Important...

- This laptop is a multiprocessor!!!
- I can now do parallel computing on airplanes
  - And not just due to the availability of WiFi
- Everyone can now afford a multiprocessor
- From more than the cost of a house to less than the cost of a bicycle in less than 20 years
- Why is this so important???

## But Much More Important...

- This laptop is a multiprocessor!!!
- I can now do parallel computing on airplanes
  - And not just due to the availability of WiFi
- Everyone can now afford a multiprocessor
- From more than the cost of a house to less than the cost of a bicycle in less than 20 years
- Why is this so important???
  - DYNIX/ptx: tens of developers, manual selection
  - AIX: hundreds of developers, automatic selection
  - Linux: thousands of developers, low contention

## Performance of Synchronization Mechanisms

## Performance of Synchronization Mechanisms

**4-CPU 1.8GHz AMD Opteron 844 system**

| Operation | Cost (ns) | Ratio |
|---|---|---|
| Clock period | 0.6 | 1 |
| Best-case CAS | 37.9 | 63.2 |
| Best-case lock | 65.6 | 109.3 |
| Single cache miss | 139.5 | 232.5 |
| CAS cache miss | 306.0 | 510.0 |

Need to be here! (Partitioning/RCU)

Heavily optimized reader-writer lock might get here for readers (but too bad about those poor writers...)

Typical synchronization mechanisms do this a lot

## Performance of Synchronization Mechanisms

**4-CPU 1.8GHz AMD Opteron 844 system**

| Operation | Cost (ns) | Ratio |
|---|---|---|
| Clock period | 0.6 | 1 |
| Best-case CAS | 37.9 | 63.2 |
| Best-case lock | 65.6 | 109.3 |
| Single cache miss | 139.5 | 232.5 |
| CAS cache miss | 306.0 | 510.0 |

Need to be here! (Partitioning/RCU)

Heavily optimized reader-writer lock might get here for readers (but too bad about those poor writers...)

Typical synchronization mechanisms do this a lot

**But this is an old system...**

## Performance of Synchronization Mechanisms

**4-CPU 1.8GHz AMD Opteron 844 system**

| Operation | Cost (ns) | Ratio |
|---|---|---|
| Clock period | 0.6 | 1 |
| Best-case CAS | 37.9 | 63.2 |
| Best-case lock | 65.6 | 109.3 |
| Single cache miss | 139.5 | 232.5 |
| CAS cache miss | 306.0 | 510.0 |

Need to be here! (Partitioning/RCU)

Heavily optimized reader-writer lock might get here for readers (but too bad about those poor writers...)

Typical synchronization mechanisms do this a lot

**But this is an old system...**

**And why low-level details???**

## Why All These Low-Level Details???

- Would you trust a bridge designed by someone who did not understand strengths of materials?
  - Or a ship designed by someone who did not understand the steel-alloy transition temperatures?
  - Or a house designed by someone who did not understand that unfinished wood rots when wet?
  - Or a car designed by someone who did not understand the corrosion properties of the metals used in the exhaust system?
  - Or a space shuttle designed by someone who did not understand the temperature limitations of O-rings?
- So why trust algorithms from someone ignorant of the properties of the underlying hardware???

## But Isn't Hardware Just Getting Faster?

## Performance of Synchronization Mechanisms

**16-CPU 2.8GHz Intel X5550 (Nehalem) System**

| Operation | Cost (ns) | Ratio |
|---|---:|---:|
| Clock period | 0.4 | 1 |
| “Best-case” CAS | 12.2 | 33.8 |
| Best-case lock | 25.6 | 71.2 |
| Single cache miss | 12.9 | 35.8 |
| CAS cache miss | 7.0 | 19.4 |

**What a difference a few years can make!!!**

## Performance of Synchronization Mechanisms

**16-CPU 2.8GHz Intel X5550 (Nehalem) System**

| Operation | Cost (ns) | Ratio |
|---|---:|---:|
| Clock period | 0.4 | 1 |
| "Best-case" CAS | 12.2 | 33.8 |
| Best-case lock | 25.6 | 71.2 |
| Single cache "miss" | 12.9 | 35.8 |
| CAS cache "miss" | 7.0 | 19.4 |
| Single cache miss **(off-core)** | 31.2 | 86.6 |
| CAS cache miss **(off-core)** | 31.2 | 86.5 |

**Not *quite* so good... But still a 6x improvement!!!**

## Performance of Synchronization Mechanisms

16-CPU 2.8GHz Intel X5550 (Nehalem) System

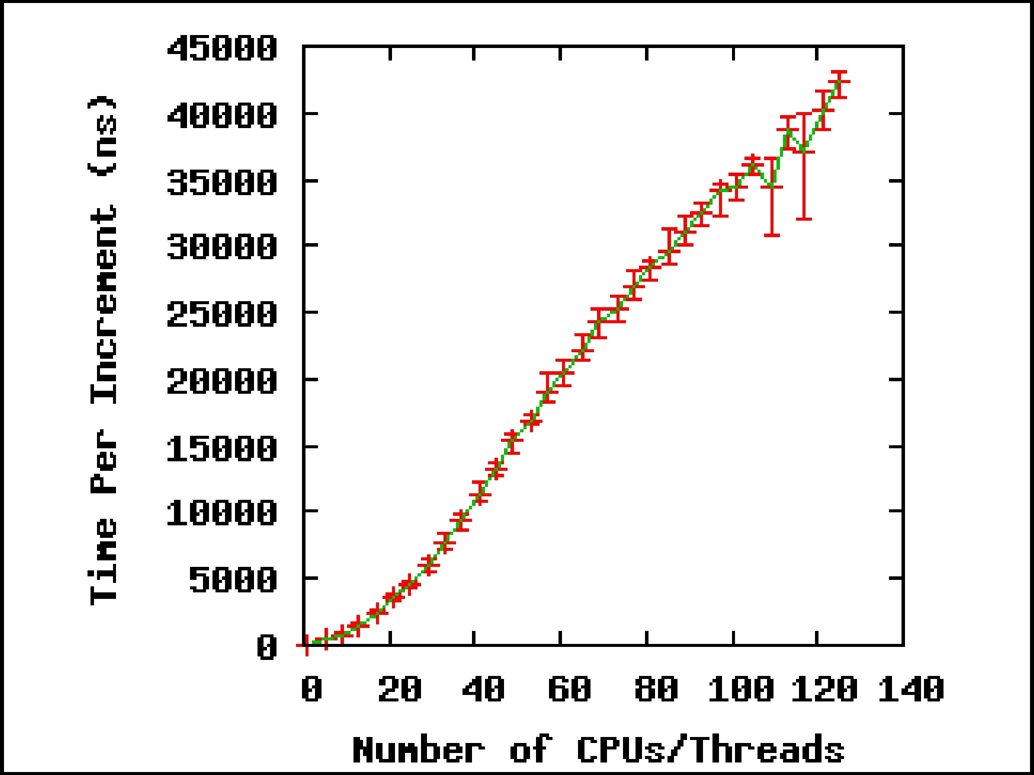
| Operation | Cost (ns) | Ratio |
|---|---|---|
| Clock period | 0.4 | 1 |
| “Best-case” CAS | 12.2 | 33.8 |
| Best-case lock | 25.6 | 71.2 |
| Single cache miss | 12.9 | 35.8 |
| CAS cache miss | 7.0 | 19.4 |
| Single cache miss **(off-core)** | 31.2 | 86.6 |
| CAS cache miss **(off-core)** | 31.2 | 86.5 |
| Single cache miss **(off-socket)** | 92.4 | 256.7 |
| CAS cache miss **(off-socket)** | 95.9 | 266.4 |

**Maybe not such a big difference after all...**
**And these are best-case values!!! (Why?)**

## Visual Demonstration of Instruction Overhead

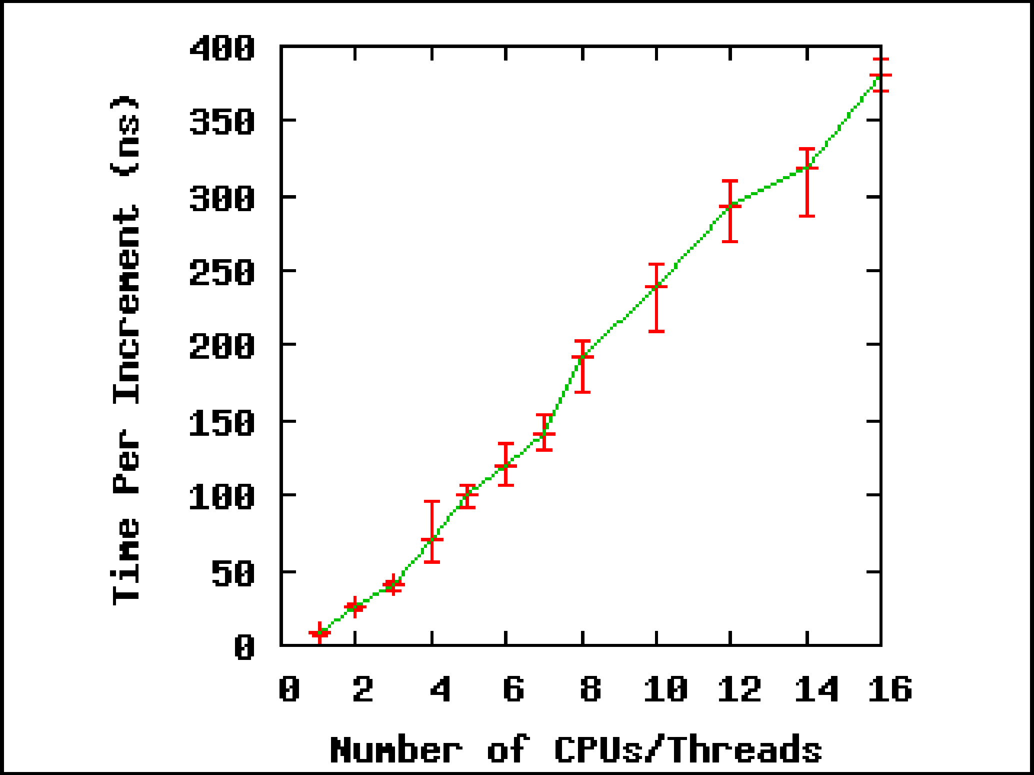
**The Bogroll Demonstration**

# Performance of Synchronization Mechanisms



If you thought a *single* atomic operation was slow, try lots of them!!!
(Atomic increment of single variable on 1.9GHz Power 5 system)

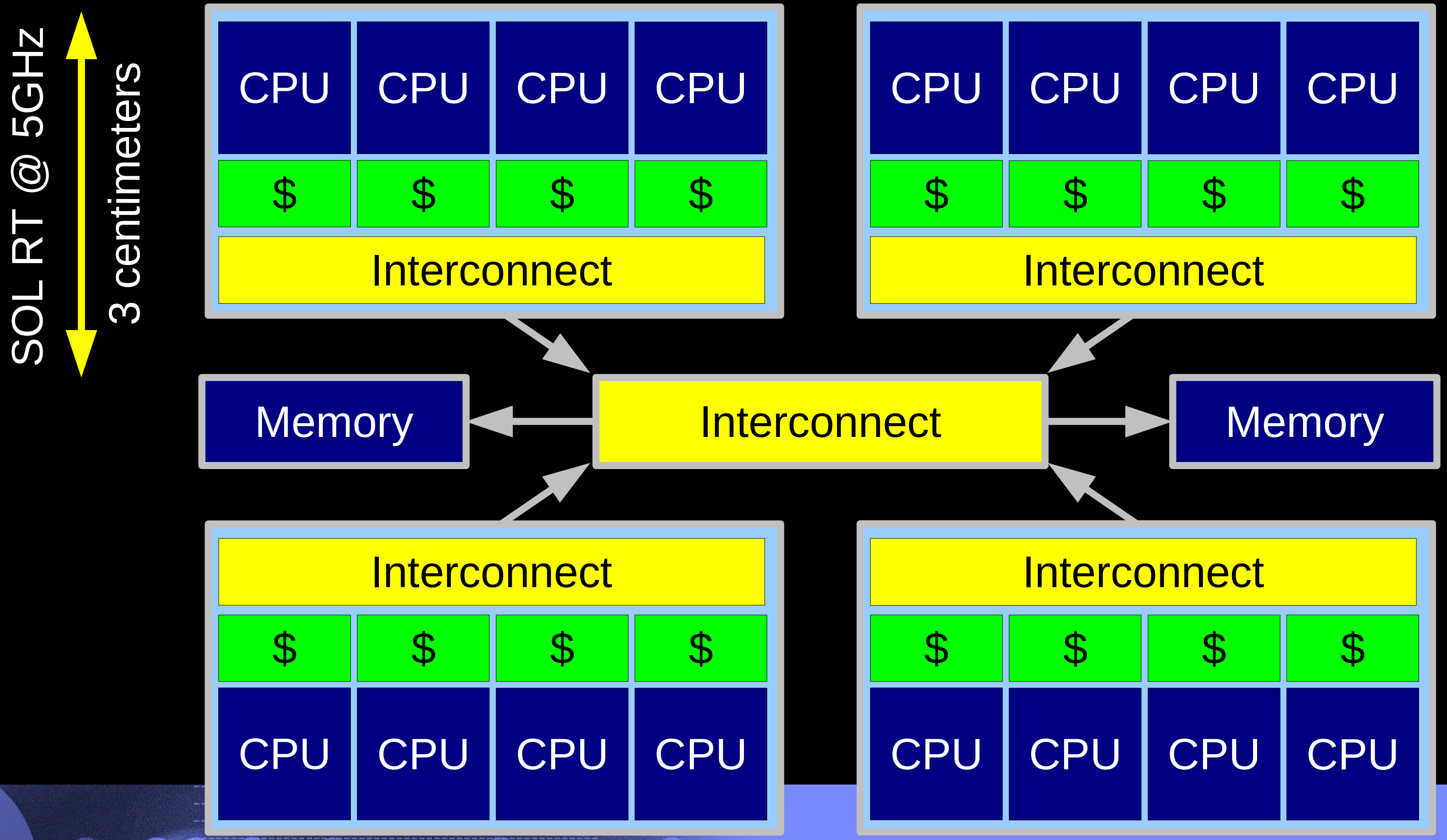
## Performance of Synchronization Mechanisms



Same effect on a 16-CPU 2.8GHz Intel X5550 (Nehalem) system

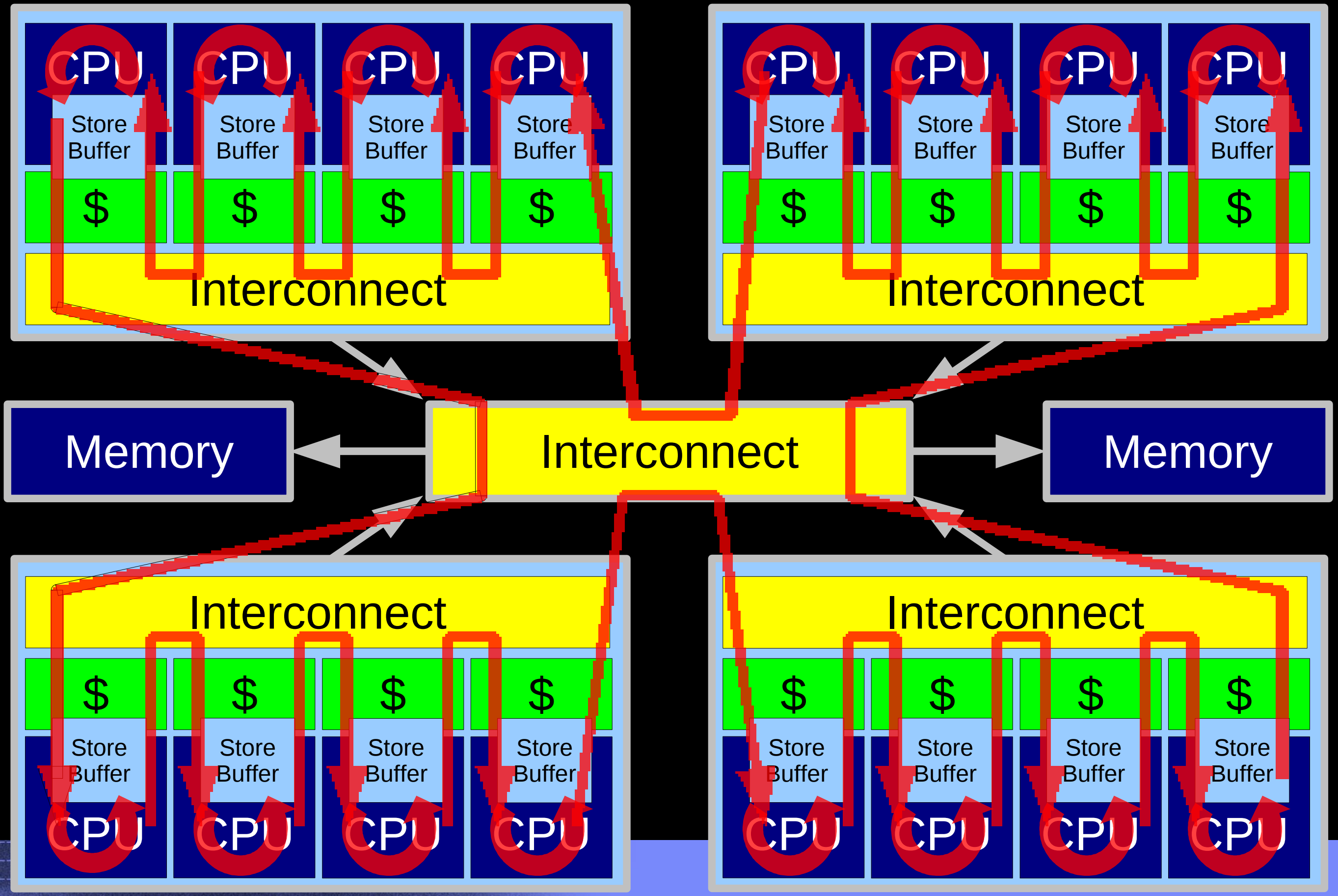
## System Hardware Structure

## System Hardware Structure



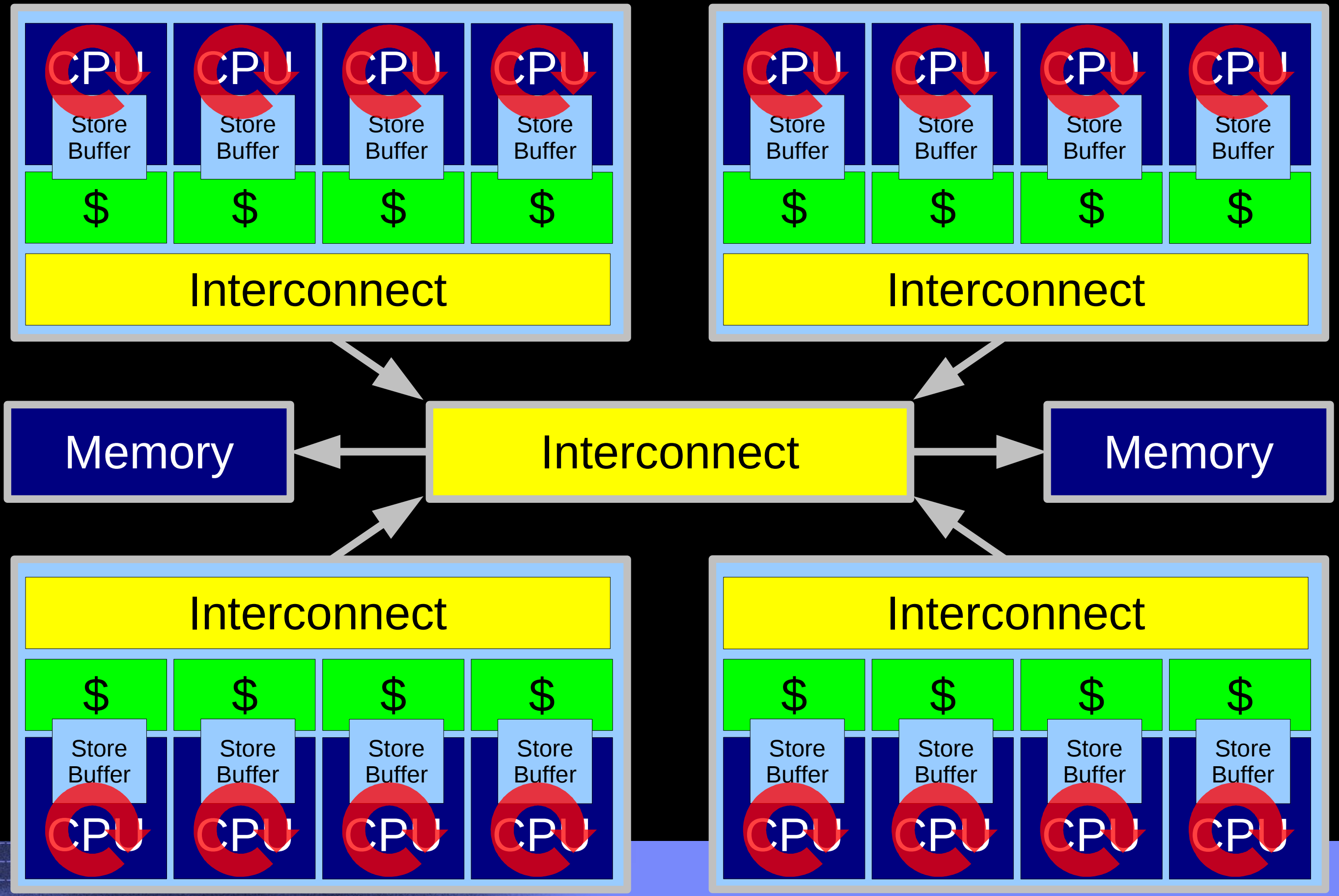
Electrons move at 0.03C to 0.3C in transistors and, so lots of waiting.

## Atomic Increment of Global Variable



Lots and Lots of Latency!!!

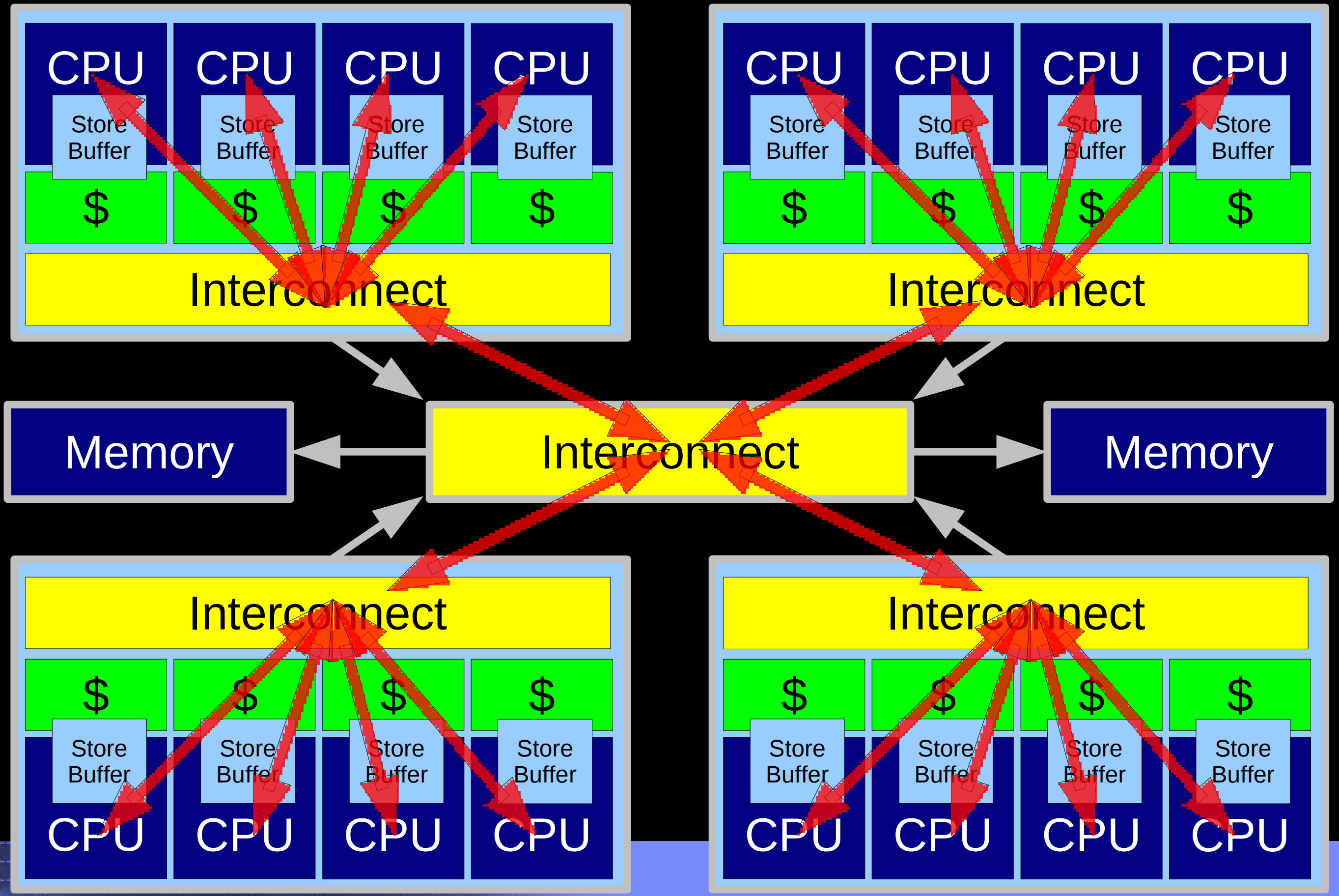
## Atomic Increment of Per-CPU Variable



Little Latency, Lots of Increments at Core Clock Rate

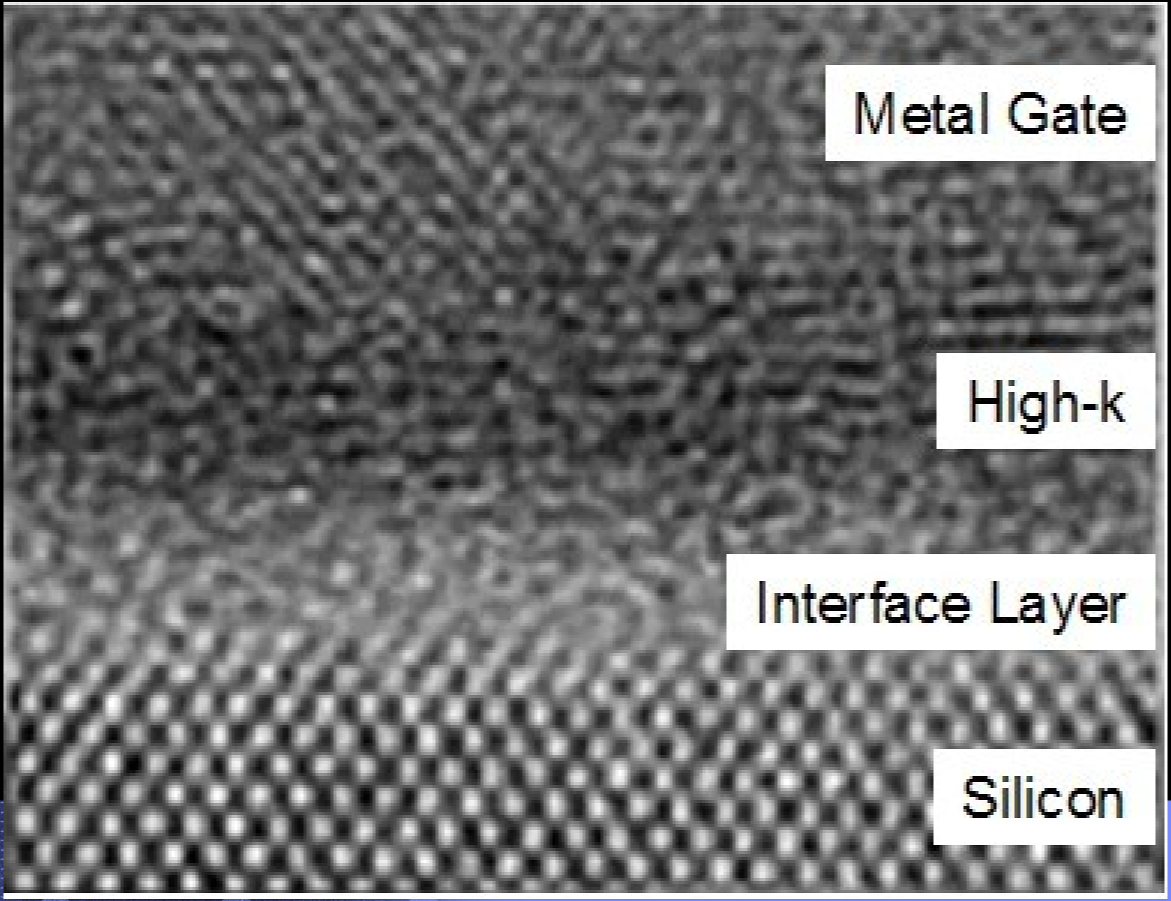
## Is There A Better HW XADD Implementation?

## HW-Assist Atomic Increment of Global Variable



Better than current hardware, but still much worse than per-thread variables!
Parallel software design can be a powerful tool: use it!!!

## Shrinking Transistors Won't Save Us

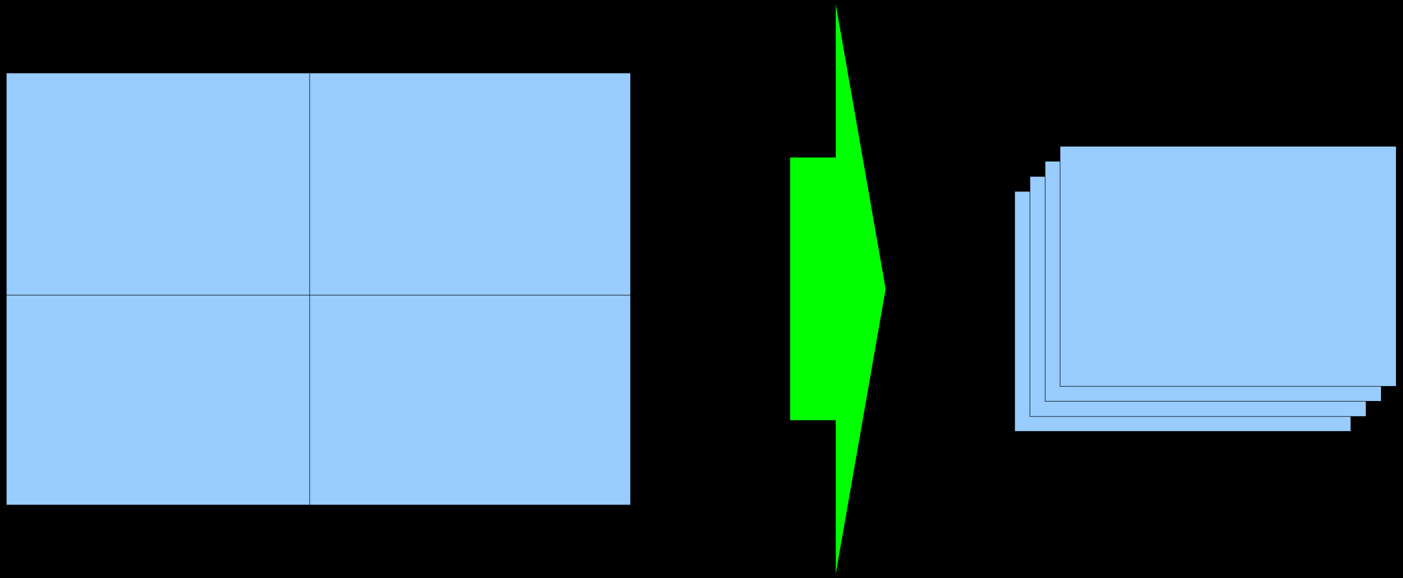


## Who is Gordon Moore Quoting?

**Gentlemen, you have two fundamental problems: (1) the finite speed of light and (2) the atomic nature of matter.**

## Is There Any HW Help To Be Had???

## Is There Any HW Help To Be Had??? Maybe...



## Parallel Programming Principles

## *IMPORTANT*

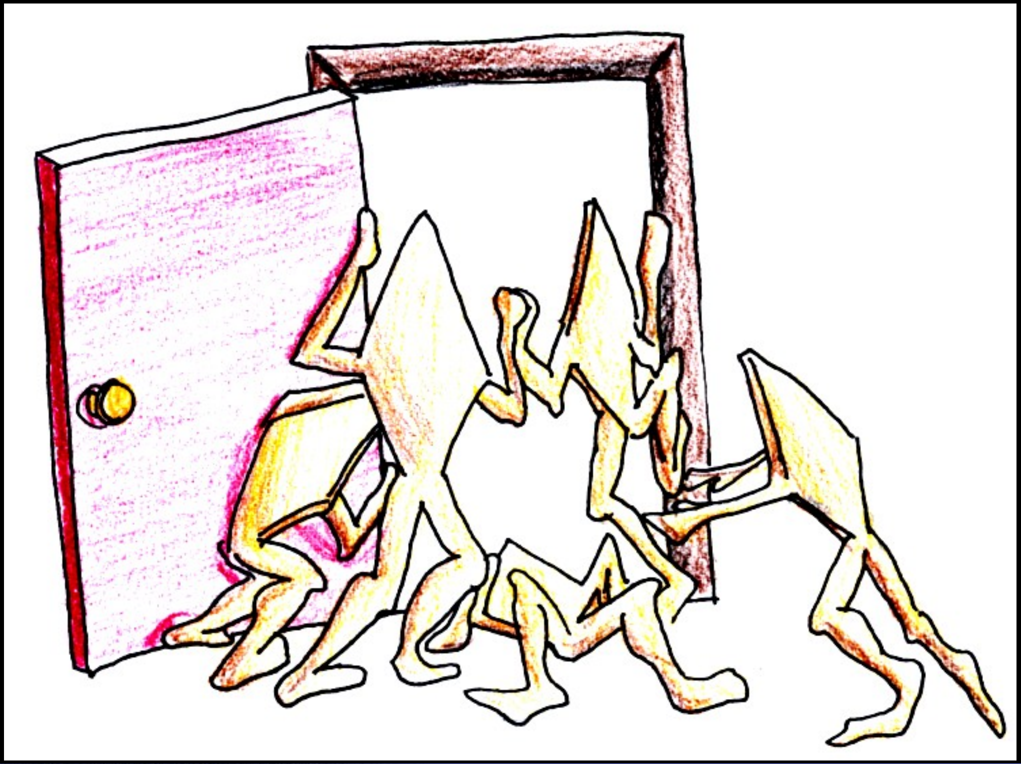
**Work *with* the hardware!!!**
***Not* against it!!!**

**Locality of Reference is Golden**

## Do Parallelism via Design, Not Implementation

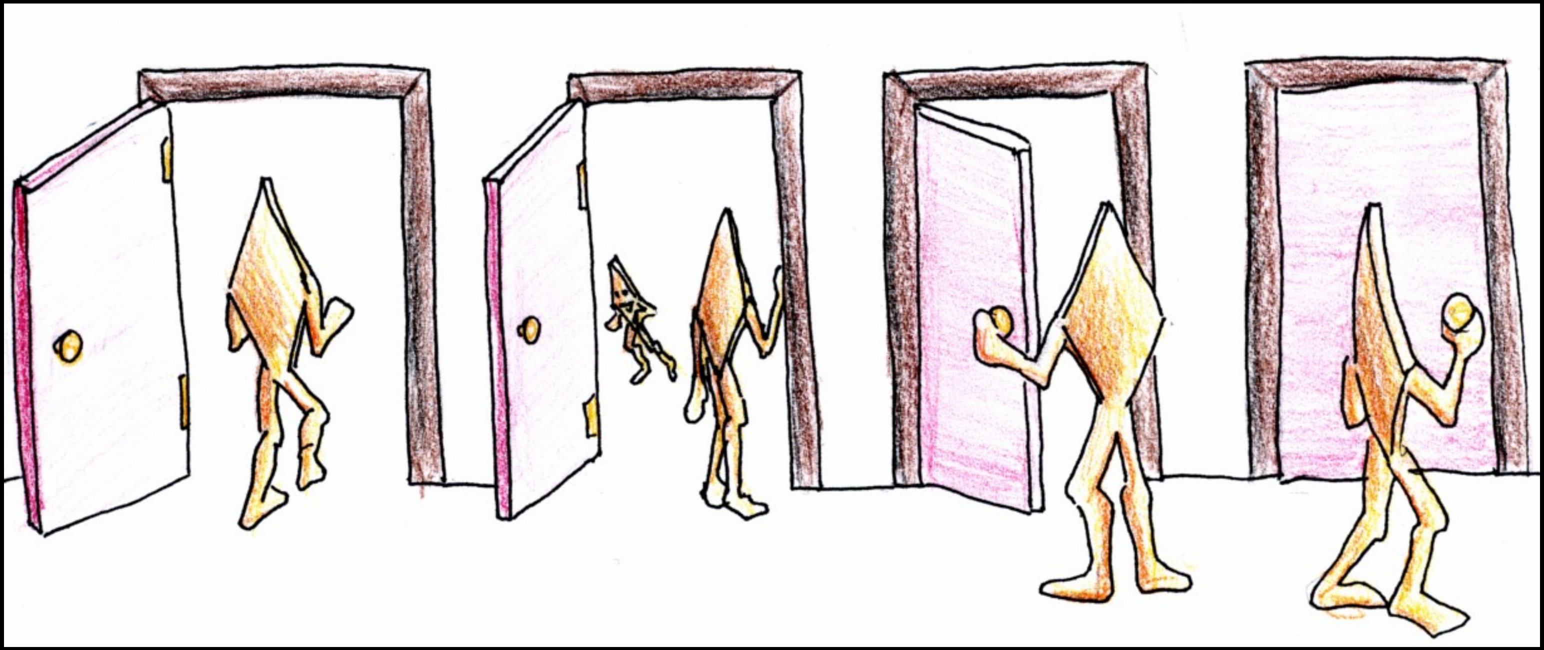
- Traditional synchronization primitives require global agreement
  - Global agreement is inherently slow on today's HW
- Traditional synchronization therefore requires coarse-grained parallelism
  - Otherwise cost of synchronization dominates
- Partitioning decisions required at high level
  - Low-level partitioning is ineffective
- Much fear of parallelism stems from ill-advised attempts to do low-level partitioning

## Design Principle: Avoid Bottlenecks



**Only one of something: bad for performance and scalability**

## Design Principle: Avoid Bottlenecks



Many instances of something: great for performance and scalability!
Any exceptions to this rule?

## Parallel Programming Exercise

## Parallel Programming Exercise

- **July 2010 IEEE Spectrum "The Trouble With Multicore" by David Patterson page 31 – calculating π:**
  - **A sequential approach:**
    - **Π/4 = 1-1/3+1/5-1/7+1/9-...**
  - **A parallel approach:**
    - **Generate a pair of random real numbers in range [-1,1]**
    - **If the pair forms a coordinate within the unit circle, count it**
    - **Π/4 = count/trials**
- **Are these good algorithms?**
  - **If so, why?**
  - **If not, what would be better?**

## Evaluation of Sequential Algorithm for π

| Iteration | Π/4 | π | Error |
|---|---|---|---|
| 0 | 1.0000 | 4.0000 | 0.8584 |
| 1 | 0.6667 | 2.6667 | -0.4749 |
| 2 | 0.8667 | 3.4667 | 0.3251 |
| 3 | 0.7238 | 2.8952 | -0.2464 |
| 4 | 0.8349 | 3.3397 | 0.1981 |
| 5 | 0.7440 | 2.9760 | -0.1655 |
| 6 | 0.8209 | 3.2837 | 0.1421 |
| 7 | 0.7543 | 3.0171 | -0.1245 |
| 8 | 0.8131 | 3.2524 | 0.1108 |
| 9 | 0.7605 | 3.0418 | -0.0998 |
| 10 | 0.8081 | 3.2323 | 0.0907 |

## Better Sequential Algorithm for π

$$\frac{\pi}{4} = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239}$$

John Machin

$$\arctan x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{2k+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

http://en.wikipedia.org/wiki/Numerical_approximations_of_%CF%80#Efficient_methods

## Better Sequential Algorithm for π

| | Term 1 | Term 2 | Π/4 | π | Error |
|---|---|---|---|---|---|
| 0 | 0.200000 | 0.004184 | 0.795816 | 3.183263598 | 4.17E-02 |
| 1 | 0.197333 | 0.004184 | 0.785149 | 3.140597029 | -9.96E-04 |
| 2 | 0.197397 | 0.004184 | 0.785405 | 3.141621029 | 2.84E-05 |
| 3 | 0.197396 | 0.004184 | 0.785398 | 3.141591772 | -8.81E-07 |
| 4 | 0.197396 | 0.004184 | 0.785398 | 3.141592682 | 2.88E-08 |
| 5 | 0.197396 | 0.004184 | 0.785398 | 3.141592653 | -9.74E-10 |
| 6 | 0.197396 | 0.004184 | 0.785398 | 3.141592654 | 3.46E-11 |
| 7 | 0.197396 | 0.004184 | 0.785398 | 3.141592654 | -3.97E-13 |
| 8 | 0.197396 | 0.004184 | 0.785398 | 3.141592654 | 8.36E-13 |
| 9 | 0.197396 | 0.004184 | 0.785398 | 3.141592654 | 7.92E-13 |
| 10 | 0.197396 | 0.004184 | 0.785398 | 3.141592654 | 7.94E-13 |

## Even Better Sequential Algorithms for π

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103+26390k)}{(k!)^4 396^{4k}}$$

Srinivasa Ramanujan

$$\frac{1}{\pi} = 12 \sum_{k=0}^{\infty} \frac{(-1)^k (6k)!(13591409+545140134k)}{(3k)!(k!)^3 640320^{3k+3/2}}$$

David Chudnovsky and Gregory Chudnovsky

http://en.wikipedia.org/wiki/Numerical_approximations_of_%CF%80#Efficient_methods

## Evaluation of Parallel Algorithm for π

## Evaluation of Parallel Algorithm for π

| Number of Trials | Number of Digits |
|---|---|
| 1 | 0 |
| 10 | 1 |
| 100 | 2 |
| 1,000 | 3 |
| 10,000 | 4 |
| 100,000 | 5 |
| 1,000,000 | 6 |
| 10,000,000 | 7 |
| 100,000,000 | 8 |
| 1,000,000,000 | 9 |

What should you do instead???

## Better Parallelization of Computation of π

- If you really need millions of digits, parallel arithmetic?
  - Need carry propagation for addition, but unlikely to carry very far
  - Multiplication can be block-evaluated
  - And this solution would have other uses
    - To the extent that huge-number exact computation is useful

## But What Would Be Even Faster???

## But What Would Be Even Faster???

**Pi =** 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745028410270193852110555964462294895493038196442881097566593344612847564823378678316527120190914564856692346034861045432664821339360726024914127372458700660631558817488152092096282925409171536436789259036001133053054882046652138414695194151160943305727036575959195309218611738193261179310511854807446237996274956735188575272489122793818301194912983367336244065664308602139494639522473719070217986094370277053921717629317675238467481846766940513200056812714526356082778577134275778960917363717872146844090122495343014654958537105079227968925892354201995611212902196086403441815981362977477130996051870721134999999837297804995105973173281609631859502445945534690830264252230825334468503526193118817101000313783875288658753320838142061717766 ...

# The Role of Non-Technical Issues

## Non-Technical Issues Can Cause Trouble...

- **Potential Obstacles:**
  - **Project based on inherently sequential algorithm**
  - **Project has multiple proprietary plugins sharing a single address space, owned by different players**
  - **Currently staffed by “software janitors” incapable of “big animal” changes**
    - **Nothing against SW janitors, but use the right guy for the job!**
  - **Project unable to fund/support “big animal” changes**
  - **APIs designed without regard to parallelism**
  - **Implemented without regard to parallelism**
  - **Implemented without regard to good software-development practice**

## Preventing Non-Technical Interference

- For parallel programming to be easy, you need:
  - Easy access to parallel hardware
  - Access to all source code sharing address space
  - Enlightened design and coding standards
  - Vigorous enforcement of said standards
  - Experienced developers to review designs and code
  - For existing non-parallel projects, sufficiently many developers ready, willing, and able to make big-animal changes
- Numerous projects, both proprietary and open-source, demonstrate what is possible
- Then again, parallelism is one optimization of many: use the right tool for the job!!!

## Conclusions

## Summary and Conclusions

- Why Parallel Programming?
- A Brief History of Parallel Programming
- Trends in Parallelism
- Performance of Synchronization Mechanisms
- System Hardware Structure
- Parallel Programming Principles
- Parallel Programming Exercise
- The Role of Non-Technical Issues
- Summary and Conclusions

## To Design Great Parallel Software

- Work with the hardware, not against it
- Introduce parallelism into high-level design
- Avoid bottlenecks
- Don't ignore non-technical obstacles
- Learn from the past, but design for the present

# Summary and Conclusions

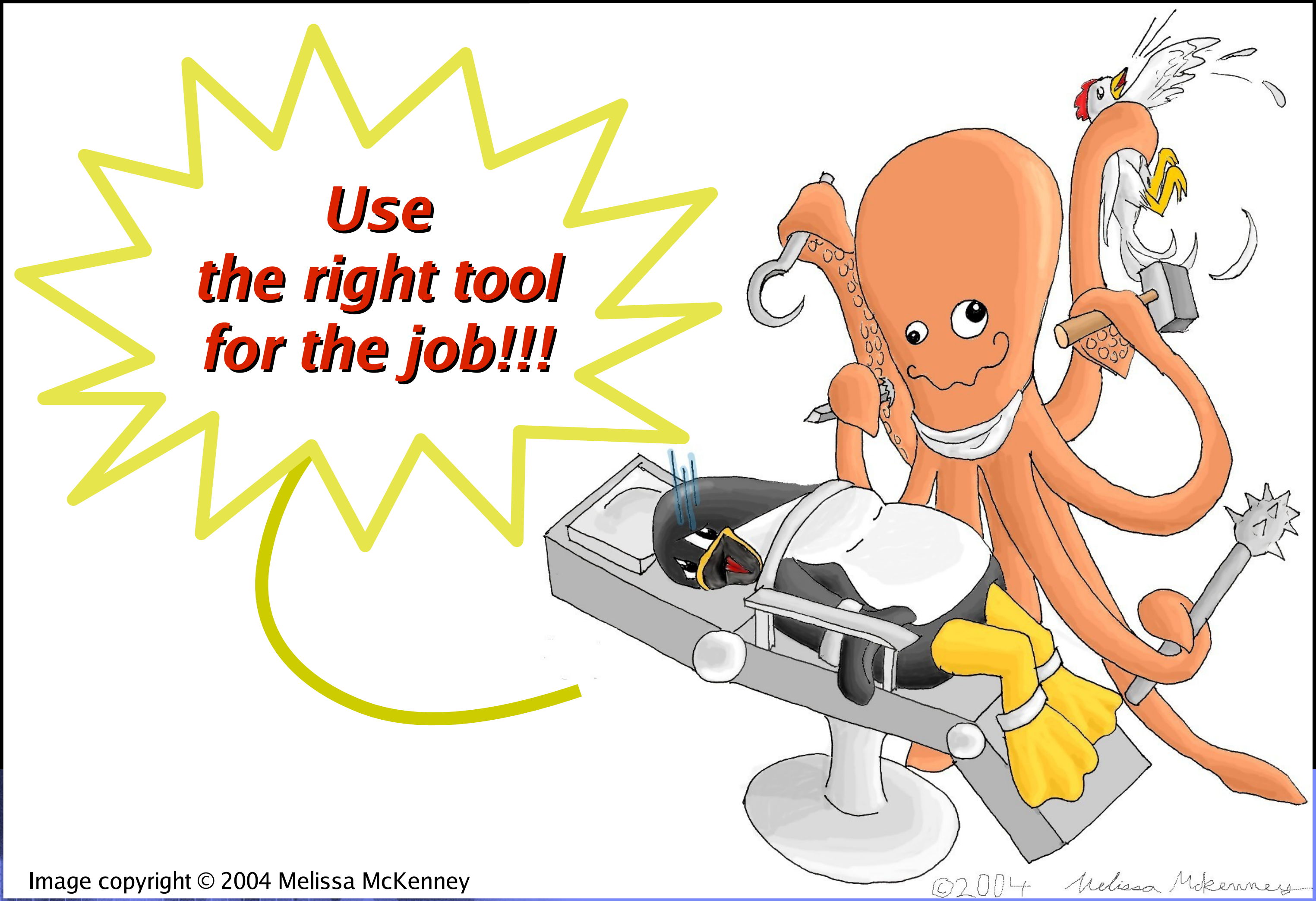


Image copyright © 2004 Melissa McKenney

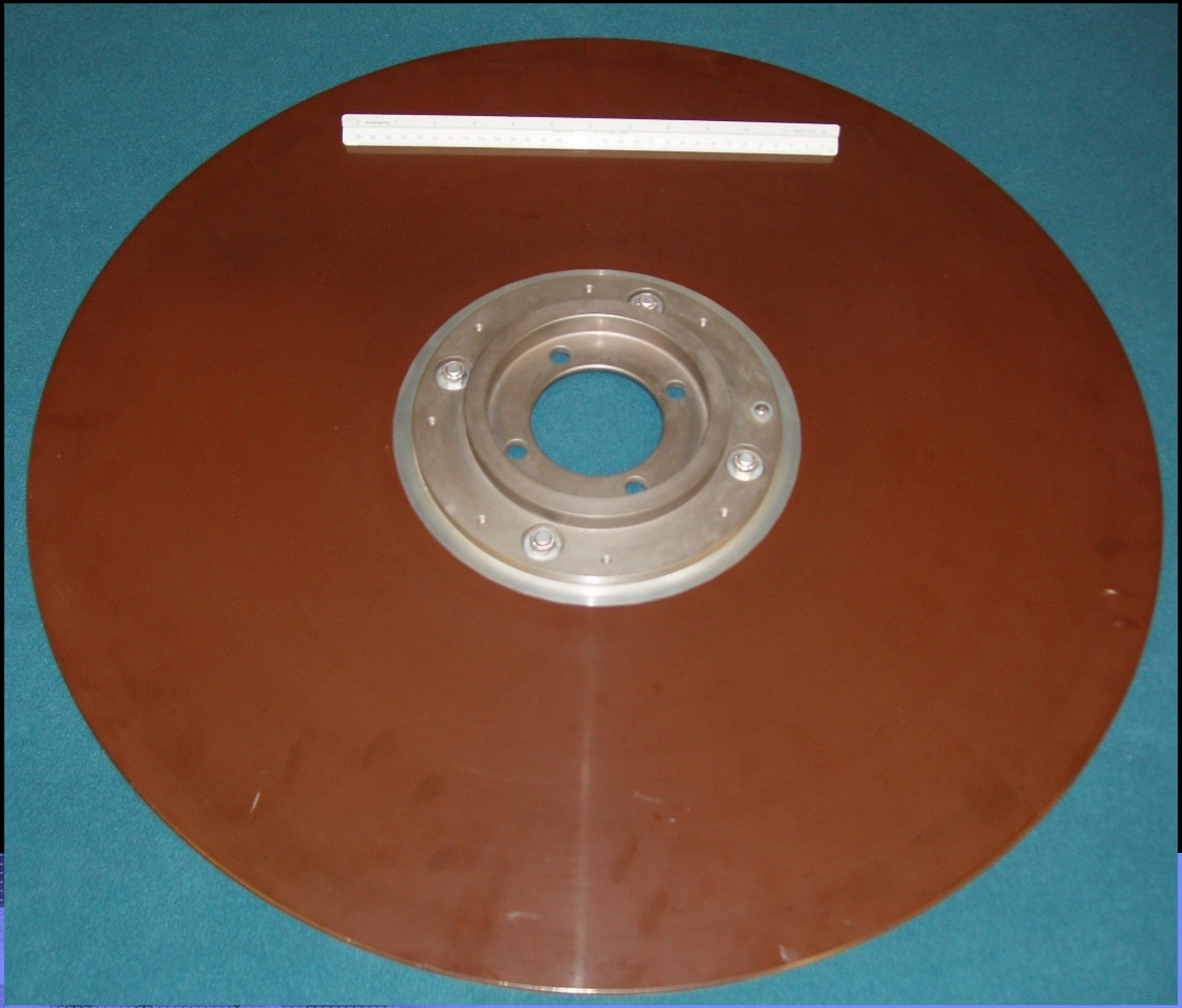
## If There Is No Right Tool, Invent It!!!

## Legal Statement

- This work represents the view of the author and does not necessarily represent the view of IBM.
- IBM and IBM (logo) are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.

## Questions?

## Questions?

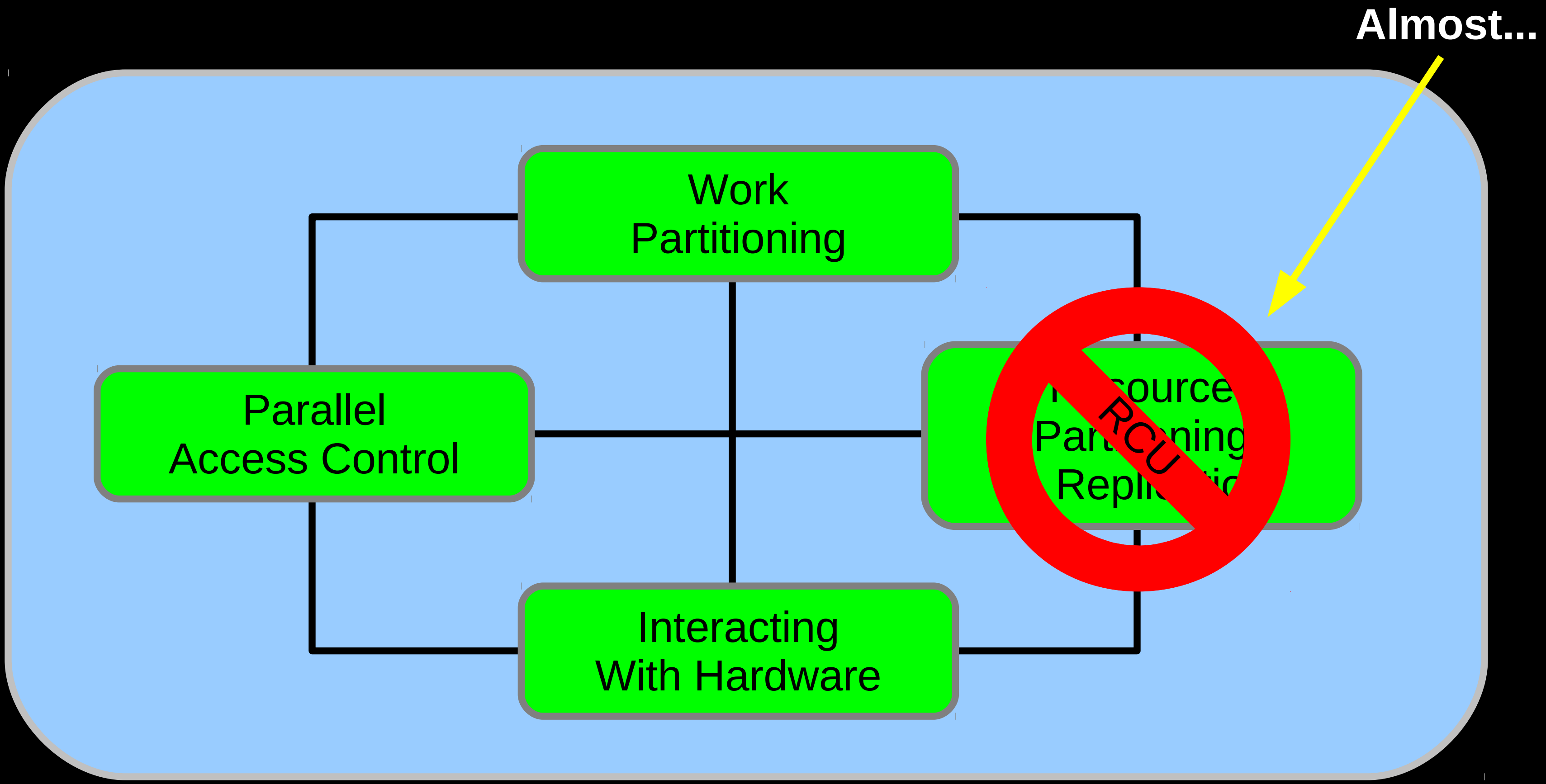


## Backup

## Parallel Programming Tasks: RCU

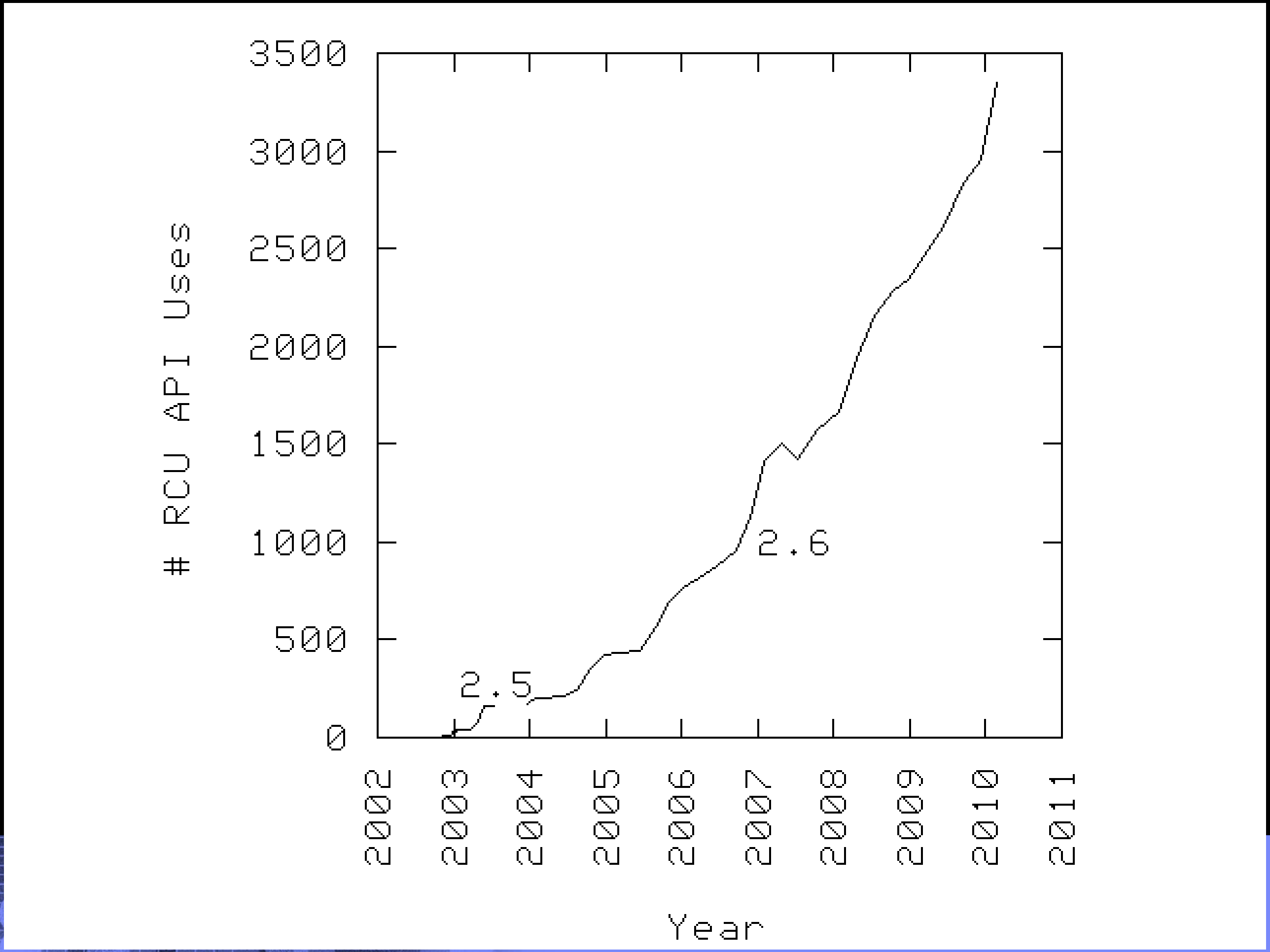
- For read-mostly data structures, RCU provides the benefits of the data-parallel model
  - But without the need to actually partition or replicate the RCU-protected data structures
  - Readers access data without needing to exclude each others or updates
    - Extremely lightweight read-side primitives
- And RCU provides additional read-side performance and scalability benefits
  - With a few limitations and restrictions....

## RCU for Read-Mostly Data Structures

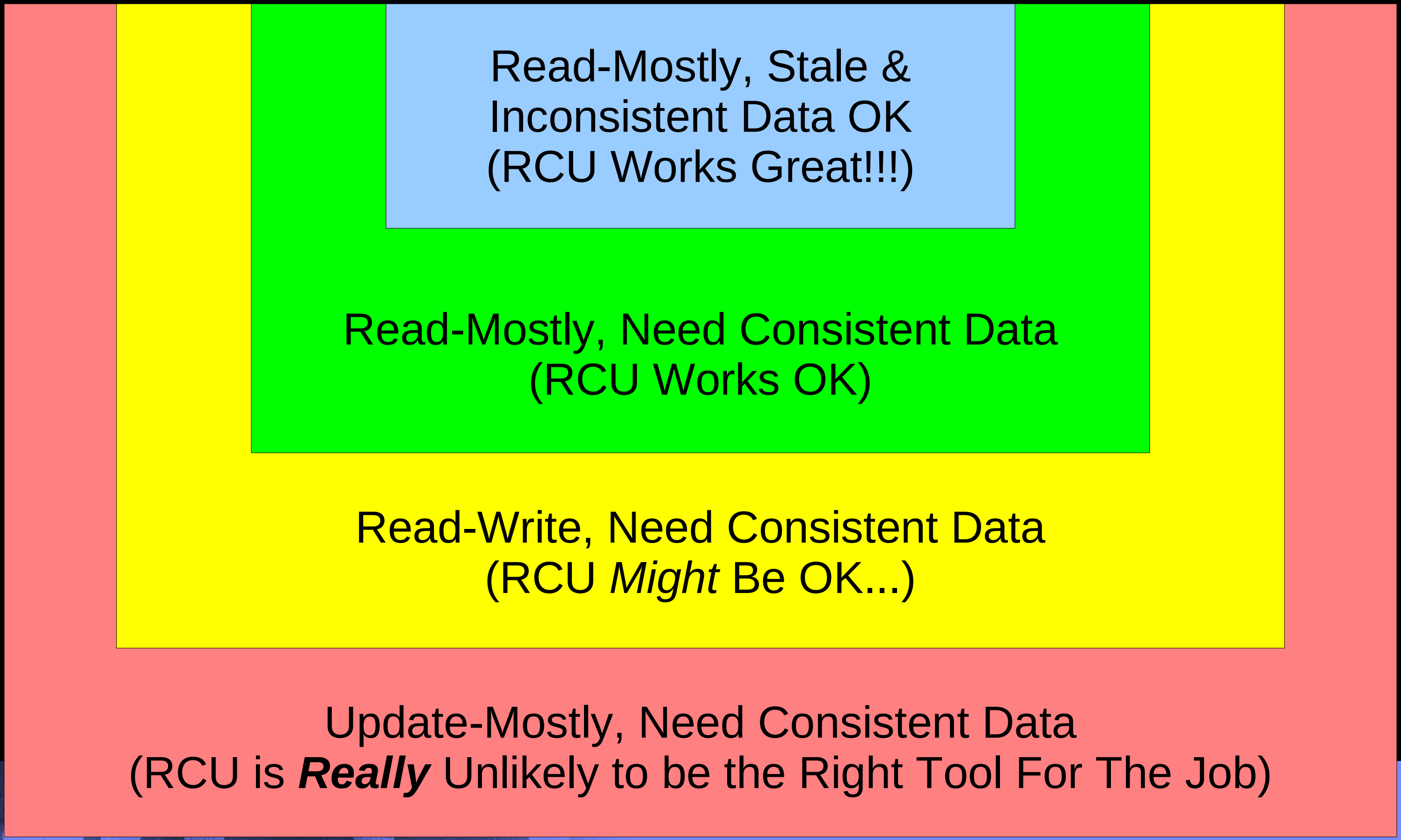


**RCU data-parallel approach: ~~first partition resources~~, then partition work, and only then worry about parallel access control, and only for updates.**

## RCU Usage in the Linux Kernel



## RCU Area of Applicability


Read-Mostly, Stale & Inconsistent Data OK (RCU Works Great!!!)
Read-Mostly, Need Consistent Data (RCU Works OK)
Read-Write, Need Consistent Data (RCU *Might* Be OK...)
Update-Mostly, Need Consistent Data (RCU is ***Really*** Unlikely to be the Right Tool For The Job)