Introduction to Performance, Scalability, and Real-Time Issues on Modern Multicore Hardware: *Is Parallel Programming Hard, And If So, Why?*

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

Course Objectives and Goals

- **Introduction to Performance, Scalability, and Real-Time Issues on Modern Multicore Hardware: Is Parallel Programming Hard, And If So, Why?**
- Performance and Scalability Technologies in the Linux Kernel
- Creating Performant and Scalable Linux Applications
- Real-Time Technologies in the Linux Kernel
- Creating Real-Time Linux Applications

- "I hear and I forget"
- "I see and I remember"
- "I do and I understand"
  - But unfortunately, we won't have time to get here this week
  - And need to go beyond understanding to habit formation
    - See "Outliers" by Malcolm Gladwell

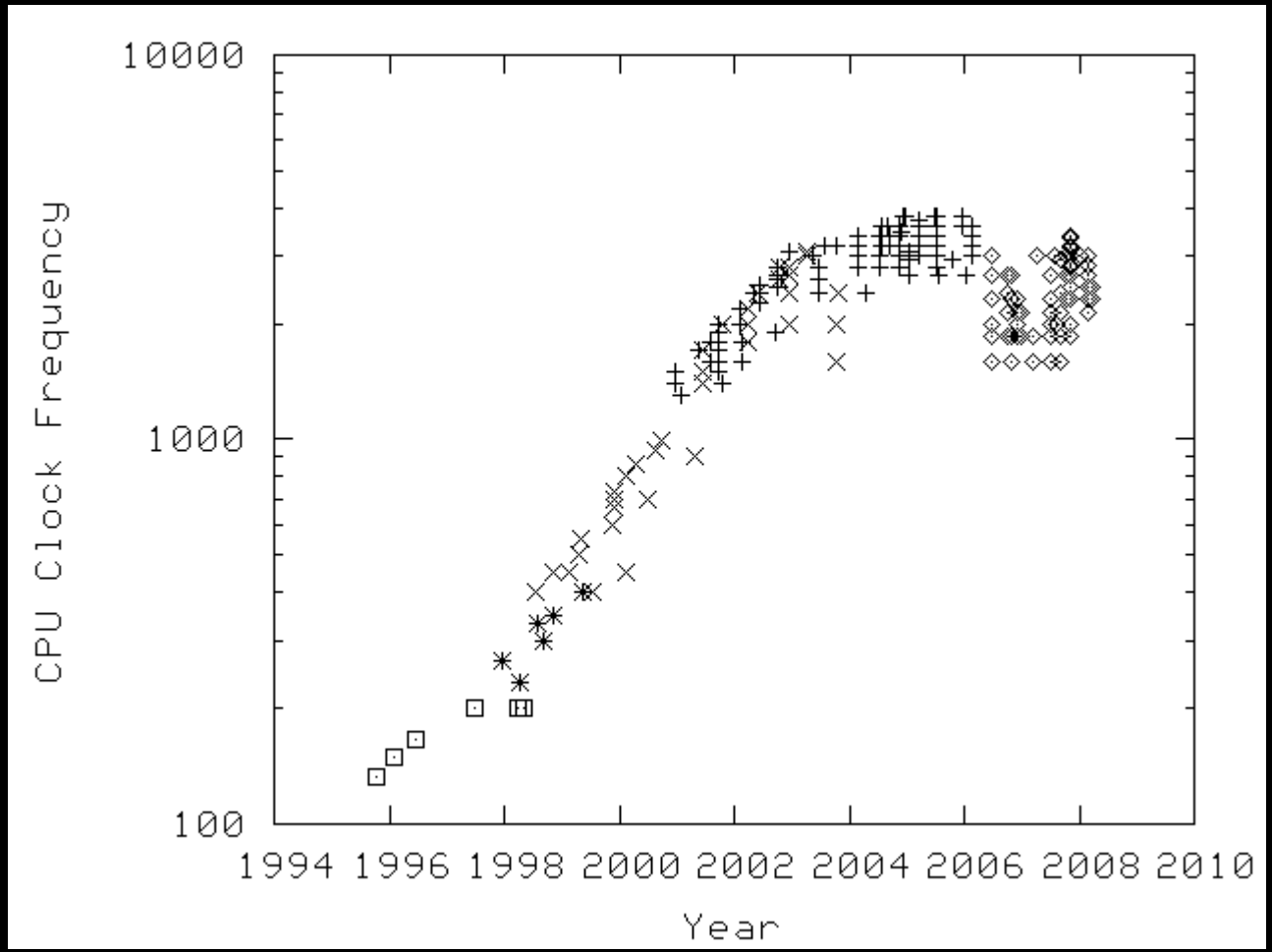
- **What decades-old parallel programming environment permits parallel-programming novices to keep large multi-core computers usefully busy?**

Overview

- Why Parallel Programming?
- Why Real-Time Programming?
- Parallel Programming Goals
- Parallel Programming Tasks
- Performance of Synchronization Operations
- Conclusions

Why Parallel Programming?

Why Parallel Programming? (Party Line)



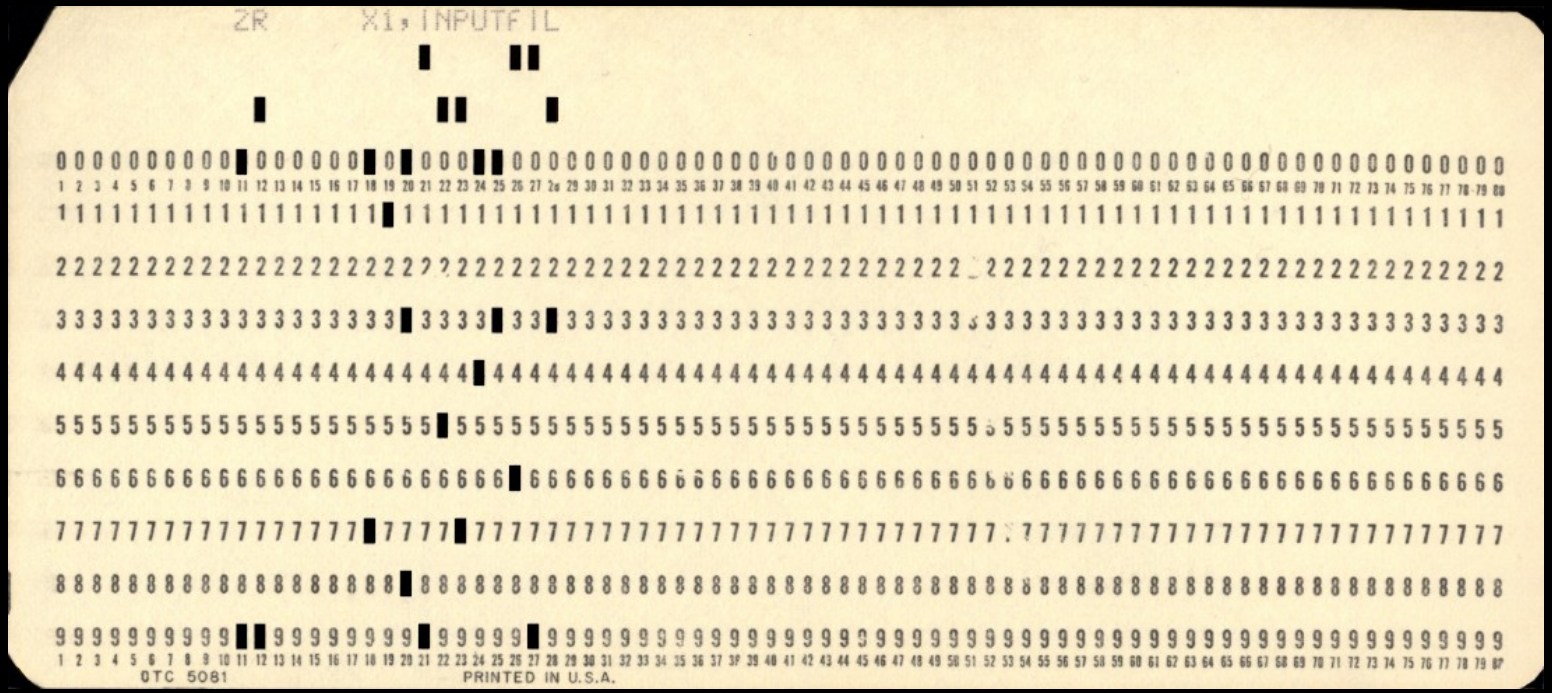
Why Parallel Programming? (Reality)

- Parallelism is one performance-optimization technique of many
  - Hashing, search trees, parsers, cordic algorithms, ...
- But operating-system kernels are 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, ...
  - More on this later!

Why Real-Time Programming?

Why Real-Time Programming?

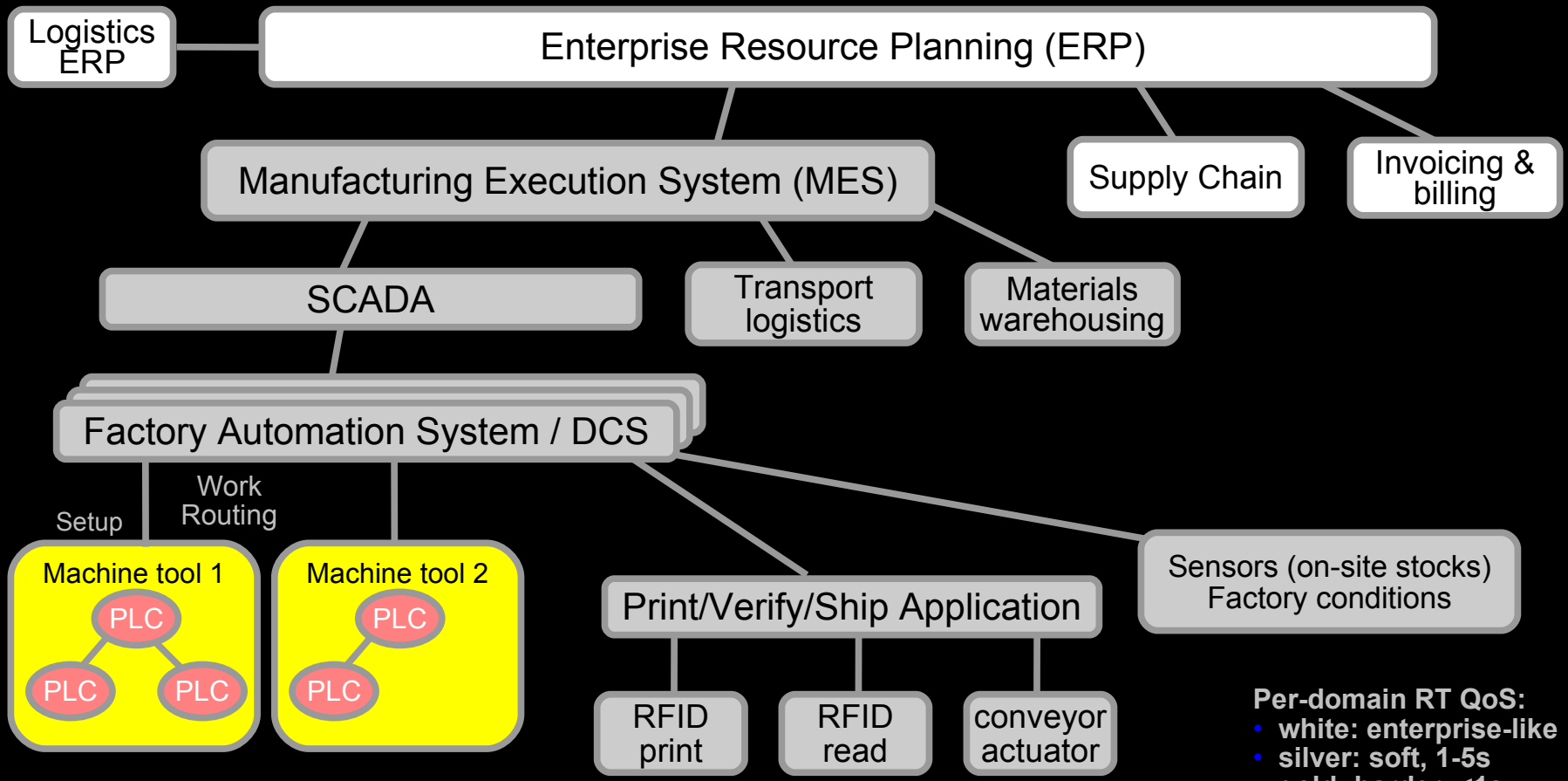
- Computer input, early-mid 1970s
  - Turn-around time ranged from minutes to days



Why Real-Time Programming?

- Commercial computing response time
  - -1970s: punched-card-based batch processing
    - Response time: minutes-days
    - One computer
  - 1960s-1980: time sharing
    - Response time: seconds-minutes
    - Two computers (host and terminal)
  - 1980s-present: workstation/desktop/laptop/...
    - Response time: milliseconds-minutes
    - One computer
  - 1990s-present: web computing
    - Response time: milliseconds-seconds
    - Lots of computers, routers, gateways, web front-ends, ...
- Industrial computing response time
  - Microseconds-milliseconds, but now interconnected

Why Real-Time Programming?



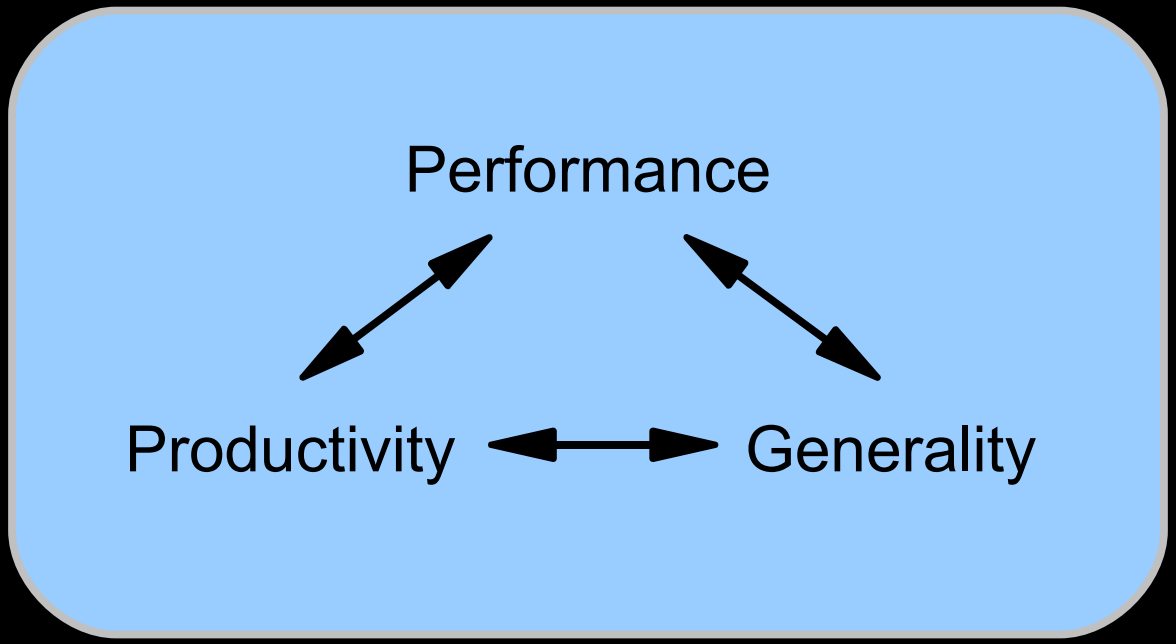
Per-domain RT QoS:

- **white: enterprise-like**
- **silver: soft, 1-5s**
- **gold: harder, <1s**
- **red: hard, sub-reflex**

SCADA: supervisory/system control and data acquisition

Parallel Programming Goals

Parallel Programming Goals



Parallel Programming Goals: Why Performance?

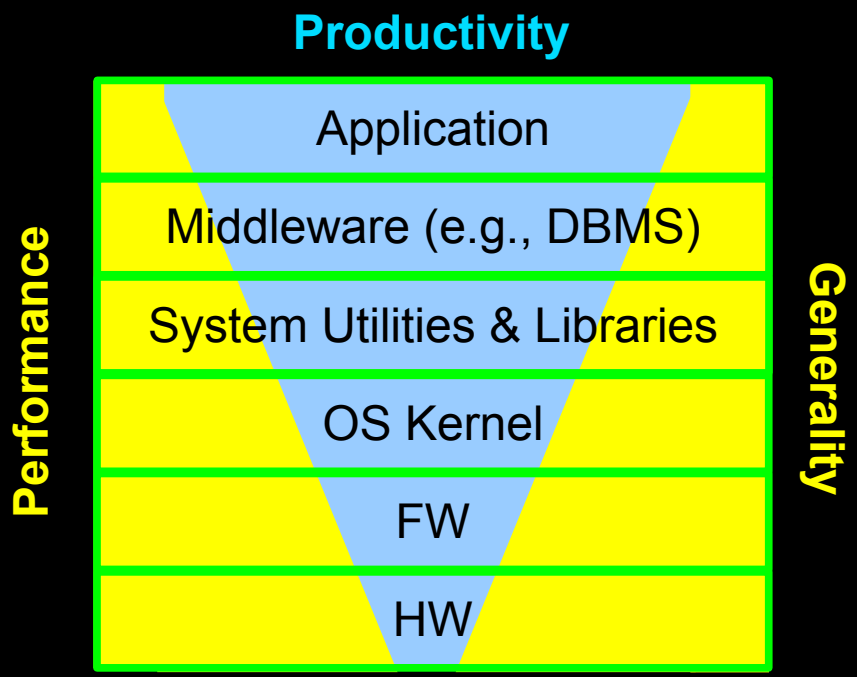
- (Performance often expressed as scalability or normalized as in performance per watt)
- If you don't care about performance, *why* are you bothering with parallelism???
  - Just run single threaded and be happy!!!
- But what about:
  - All the multi-core systems out there?
  - Efficient use of resources?
  - Everyone saying parallel programming is crucial?
- Parallel Programming: one optimization of many
- CPU: one potential bottleneck of many

Parallel Programming Goals: Why Productivity?

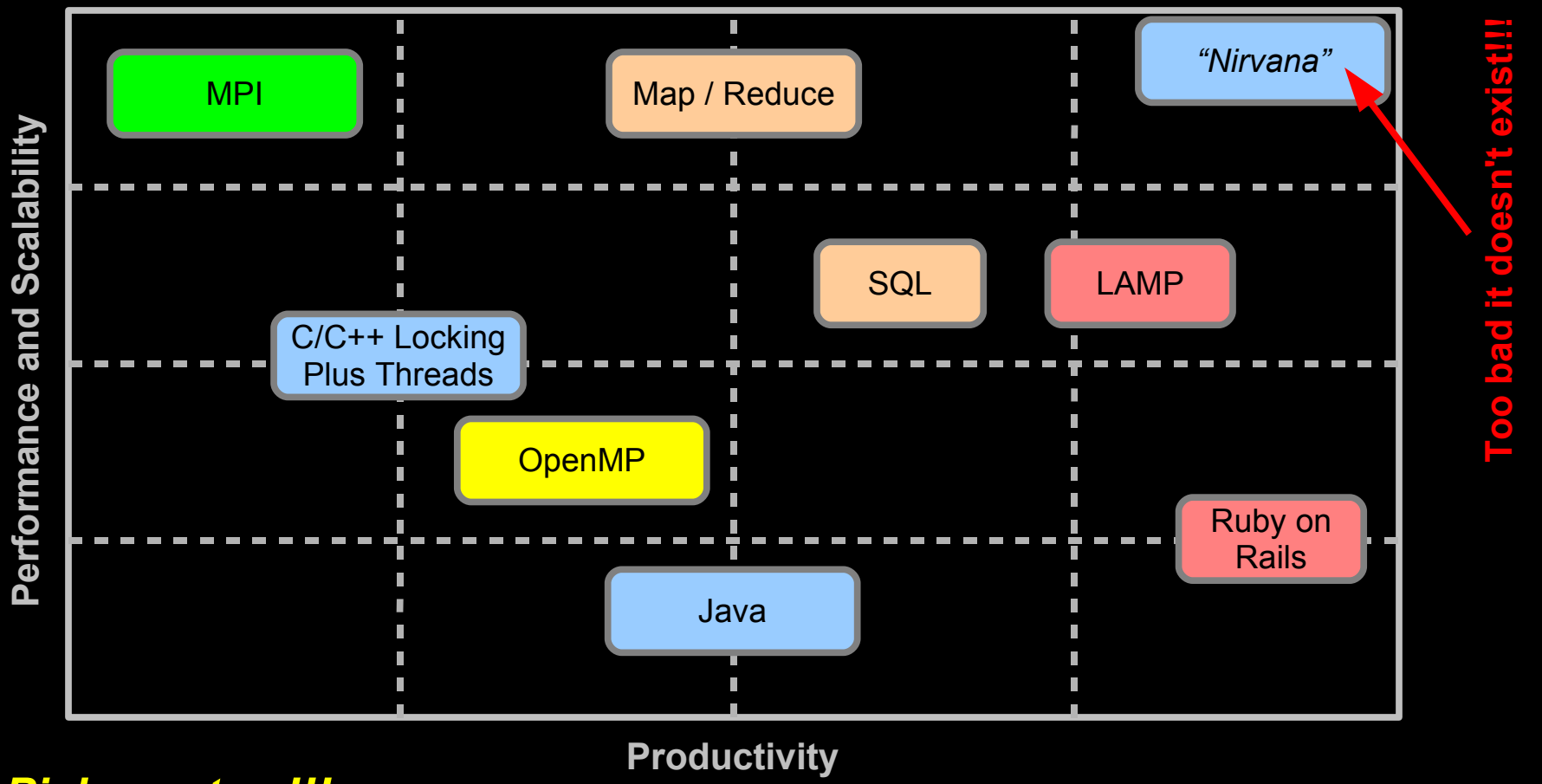
- 1948 CSIRAC (oldest intact computer)
  - 2,000 vacuum tubes, 768 20-bit words of memory
  - $10M AU construction price
  - 1955 technical salaries: $3-5K/year
  - Makes business sense to dedicate 10-person team to increasing performance by 10%
- 2008 z80 (popular 8-bit microprocessor)
  - 8,500 transistors, 64K 8-bit works of memory
  - $1.36 per CPU in quantity 1,000 (7 OOM decrease)
  - 2008 SW starting salaries: $50-95K/year US (1 OOM increase)
  - Need 1M CPUs to break even on a one-person-year investment to gain 10% performance!
    - Or 10% more performance must be blazingly important
    - Or you are doing this as a hobby... In which case, do what you want!

Parallel Programming Goals: Why Generality?

- The more general the solution, the more users to spread the cost over.



Performance, Scalability, and Generality



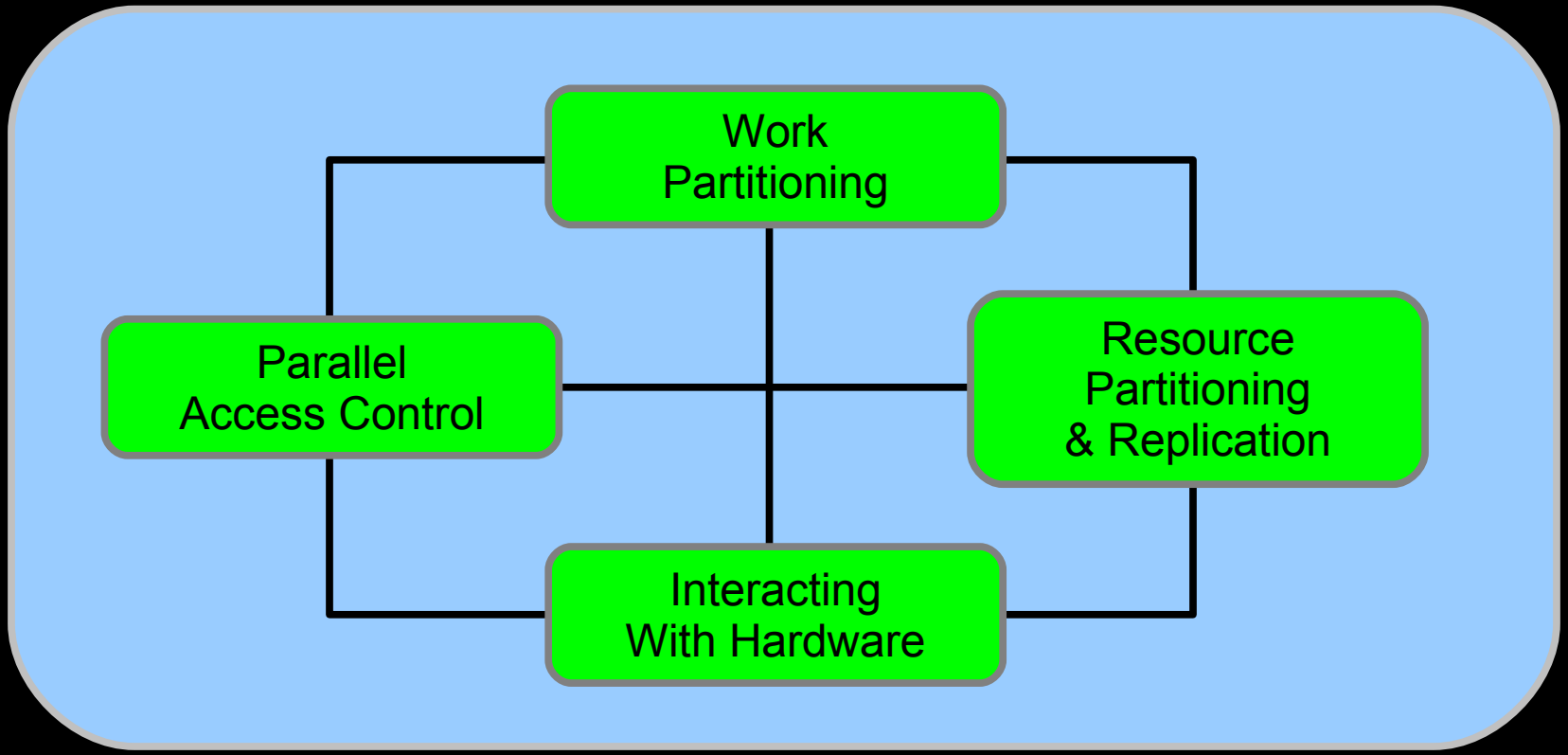
Pick any two!!!

Parallel Programming Tasks

Parallel Programming Tasks

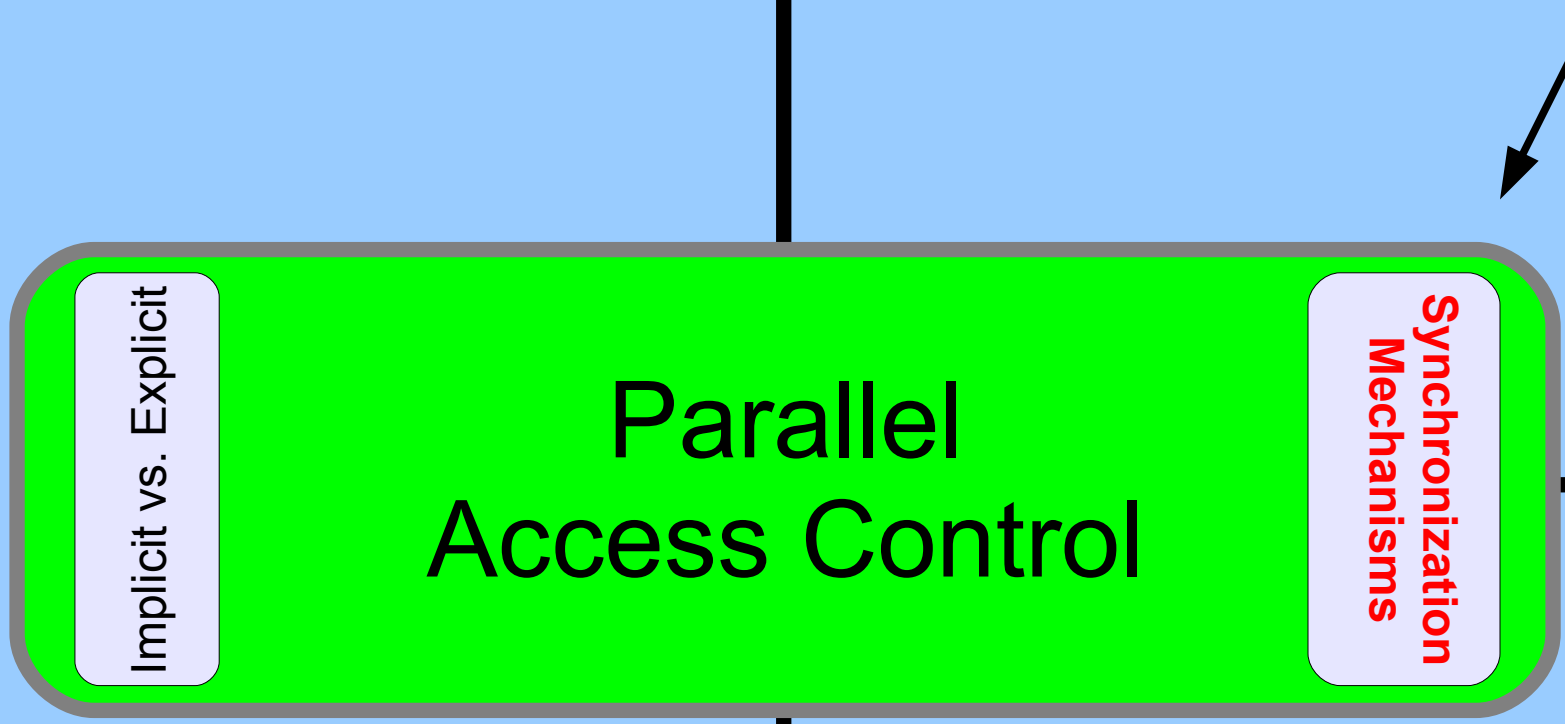
- Parallel Programming Only Partly Technical
  - Human element is extremely important
  - What can a human being easily construct and read?
    - Similar to stylized English used in emergency situations
    - Clarity, concision, and unambiguity trump style and grace
- In a perfect world, use human-factors studies
  - But few very narrow parallel human-factors studies
  - And programmers vary by orders of magnitude
  - < 3-4 OOM benefit is invisible to affordable study
- Therefore, look at tasks that must be performed for parallel programs that need not be for sequential programs

Parallel Programming Tasks



Data-parallel approach: first partition resources, then partition work, and only then worry about parallel access control. Lather, rinse, and repeat.

Parallel Programming Tasks (Close-Up View)



Parallel Programming Tasks (Even Closer View)

Synchronization Mechanisms

Locking	Data Ownership	Transactions
Message Passing	Hazard Pointers	SeqLocks
Reference Counting	NBS	RCU

To name but a few...

Performance of Synchronization Mechanisms

Performance of Synchronization Mechanisms

4-CPU 1.8GHz AMD Opteron 844 system

Need to be here! (Partitioning/RCU)

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

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

Need to be here!
(Partitioning/RCU)

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

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???

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!!!

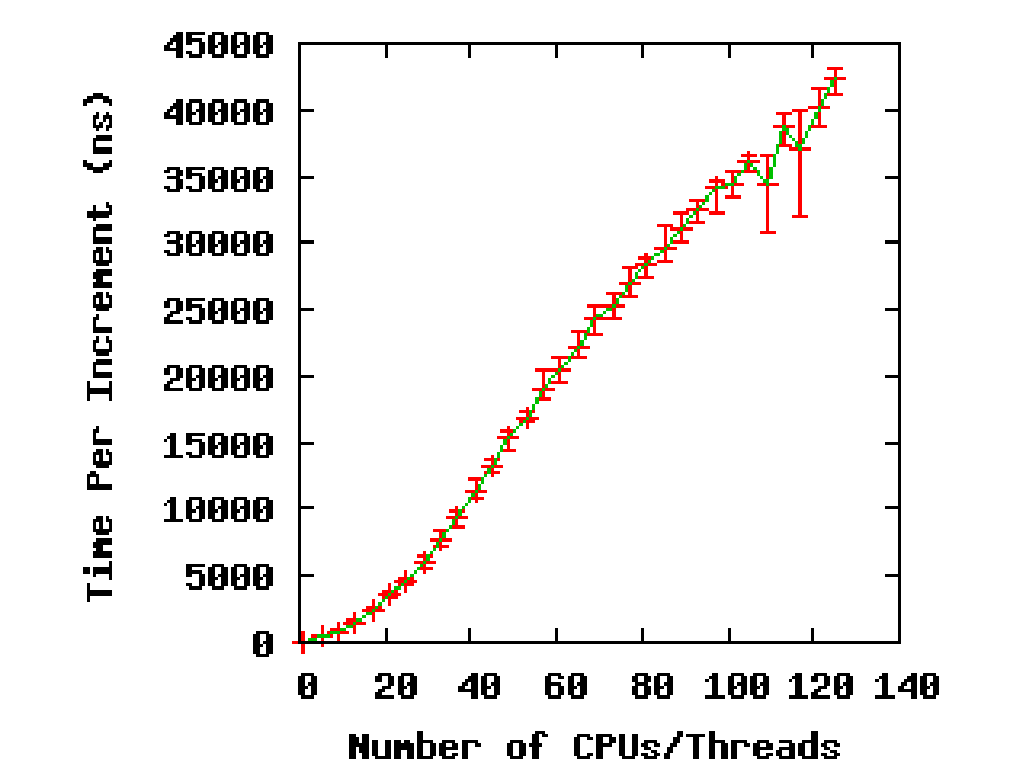
Performance of Synchronization Mechanisms

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

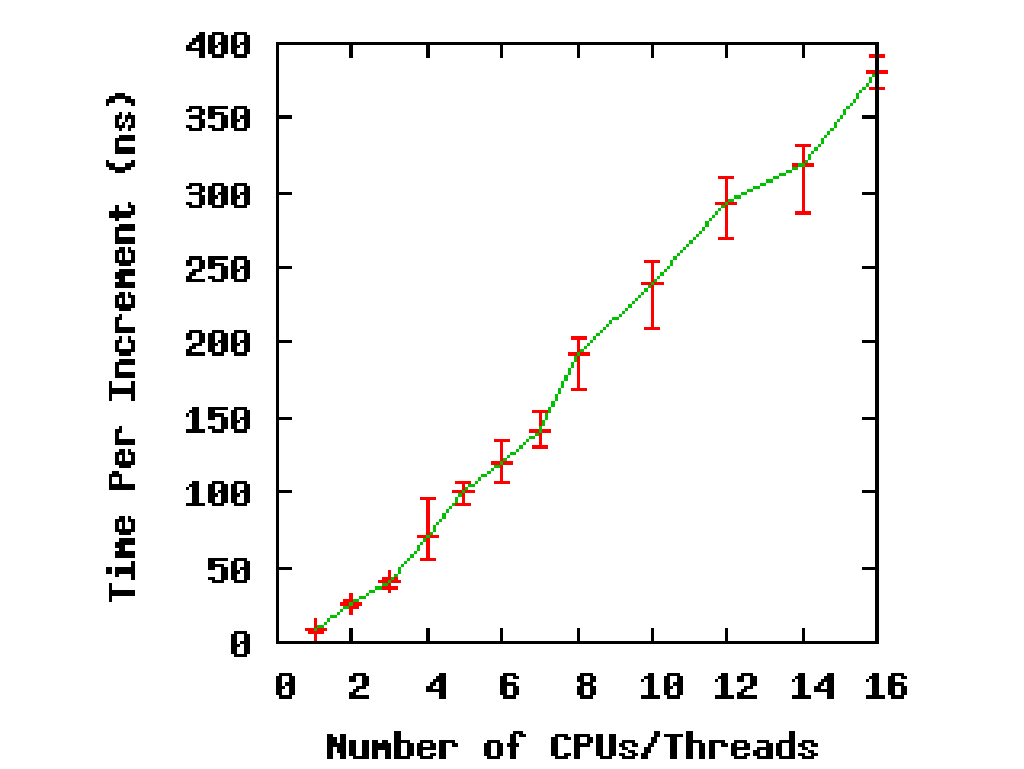
Maybe not such a big difference after all...
And these are best-case values!!! (Why?)

Performance of Synchronization Mechanisms



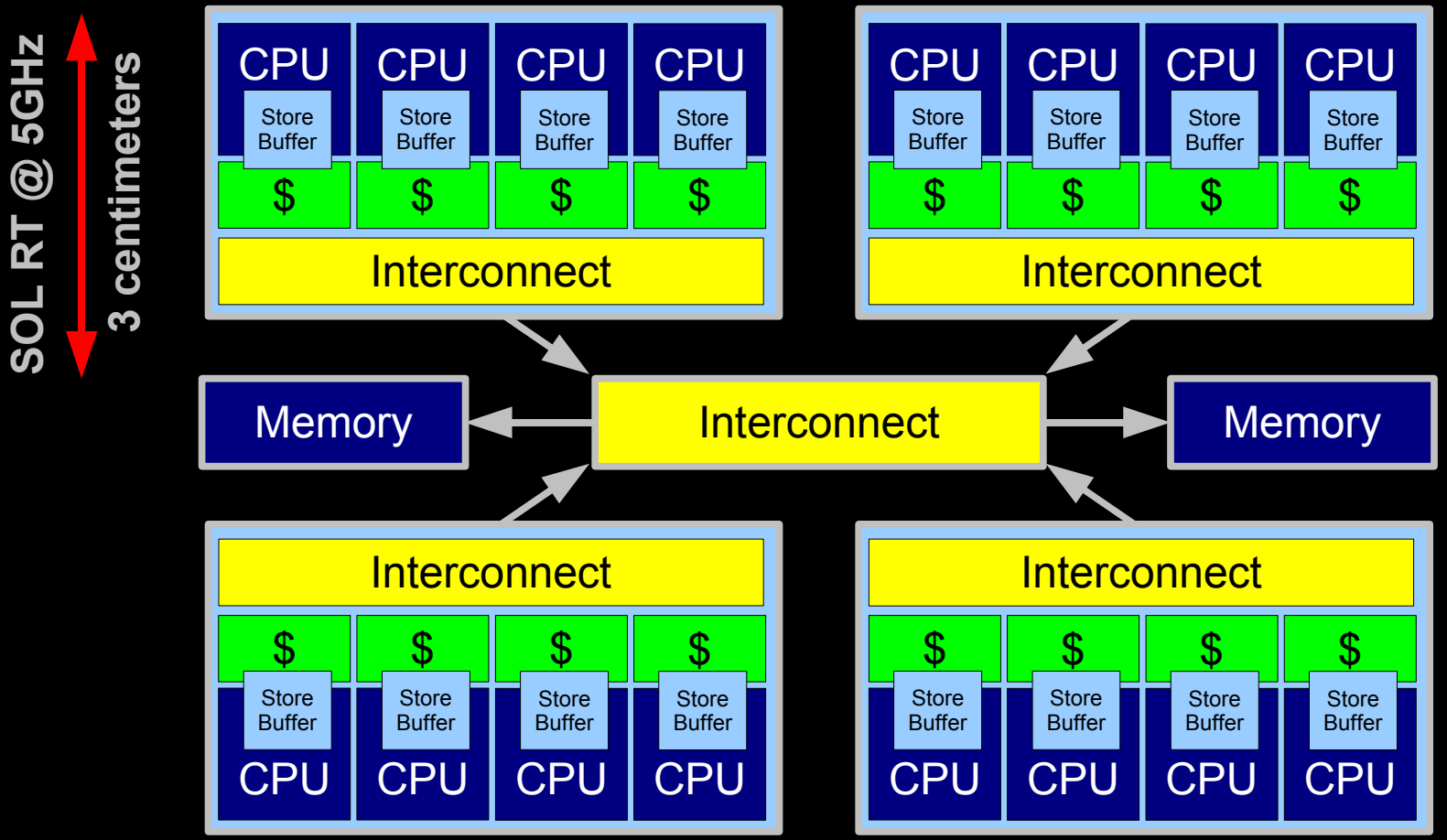
If you thought a *single* atomic operation was slow, try lots of them!!!
(Parallel atomic increment of single variable on 1.9GHz Power 5 system)

Performance of Synchronization Mechanisms



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

System Hardware Structure

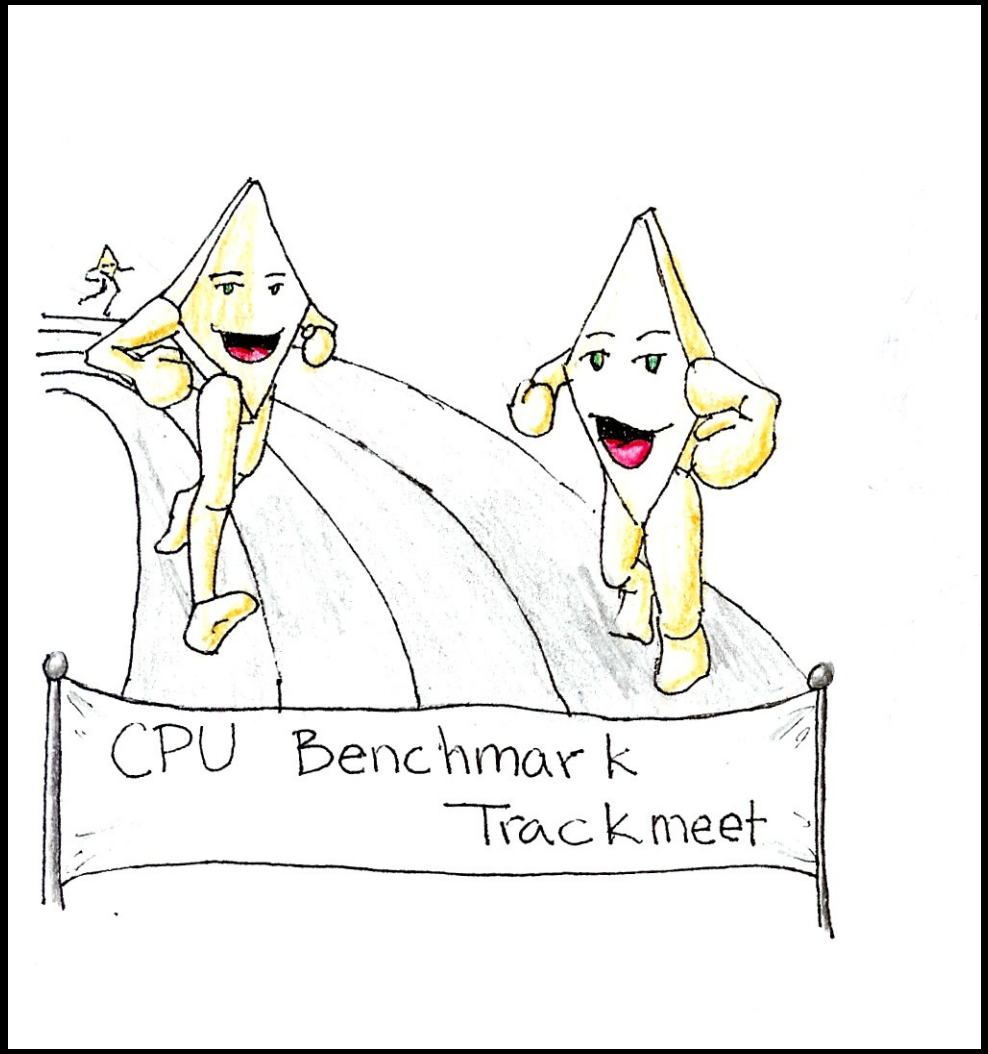


Electrons move at 0.03C to 0.3C in transistors and, so lots of waiting. 3D???

Visual Demonstration of Instruction Overhead

The Bogroll Demonstration

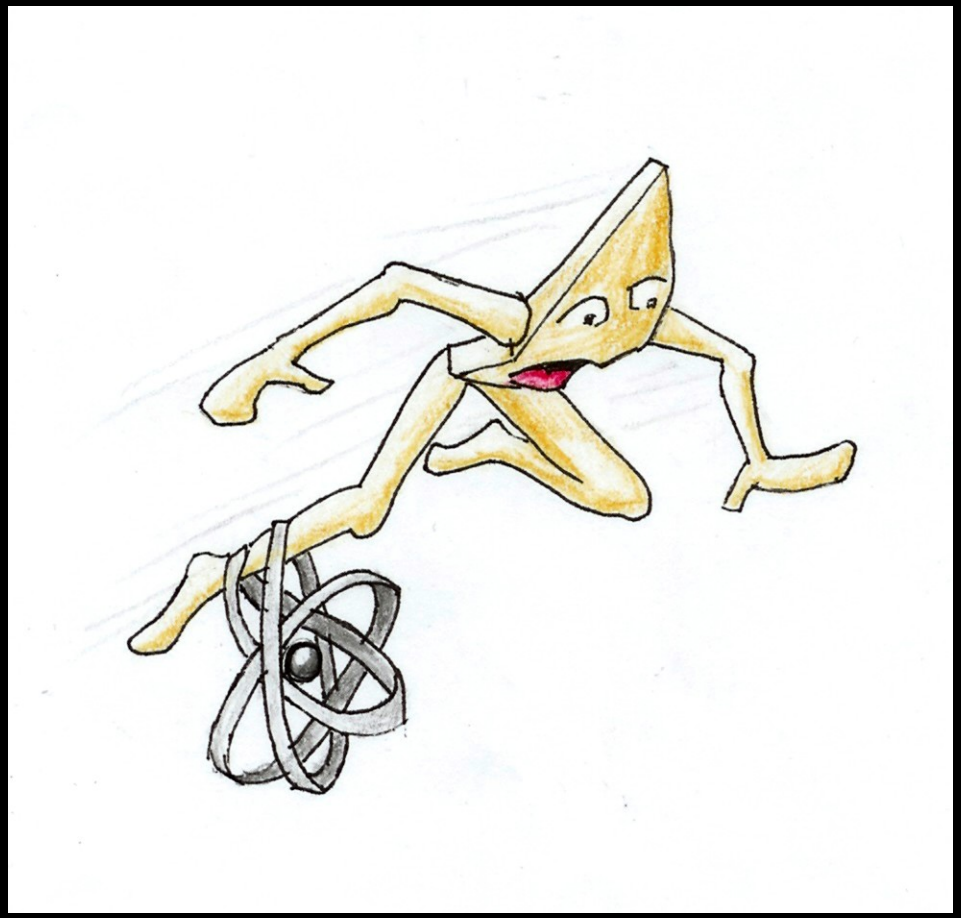
CPU Performance: The Marketing Pitch



CPU Performance: Memory References

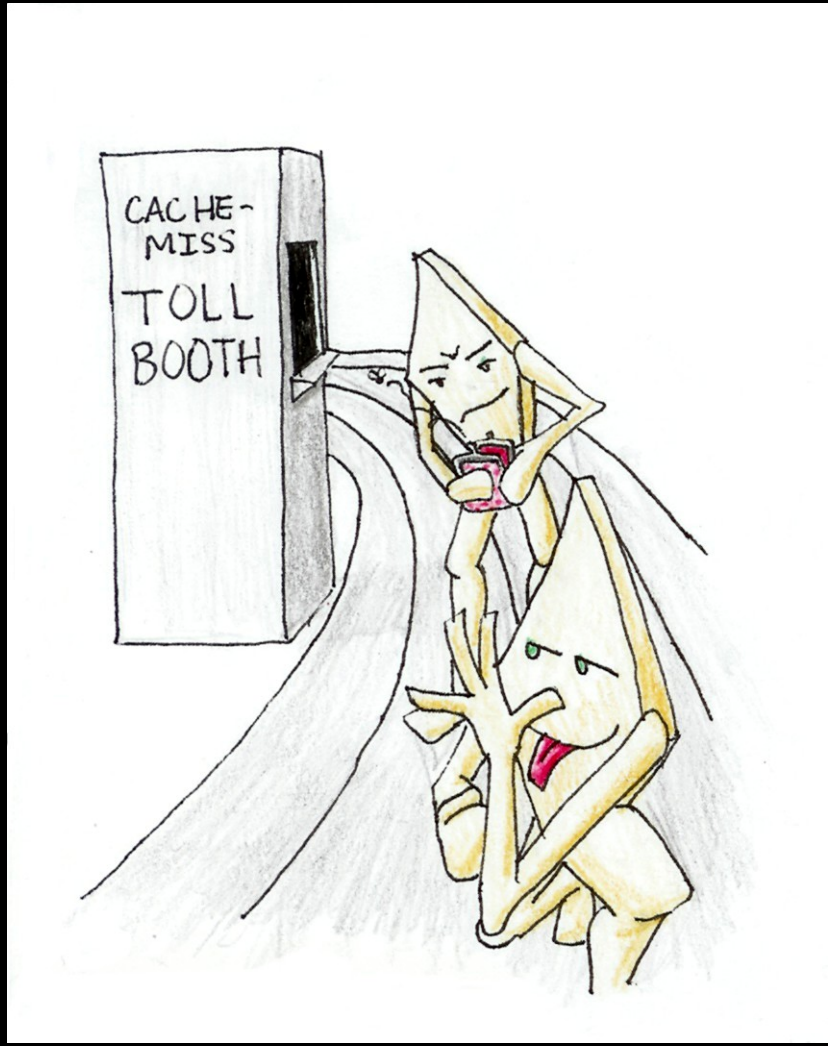
CPU Performance: Pipeline Flushes

CPU Performance: Atomic Instructions

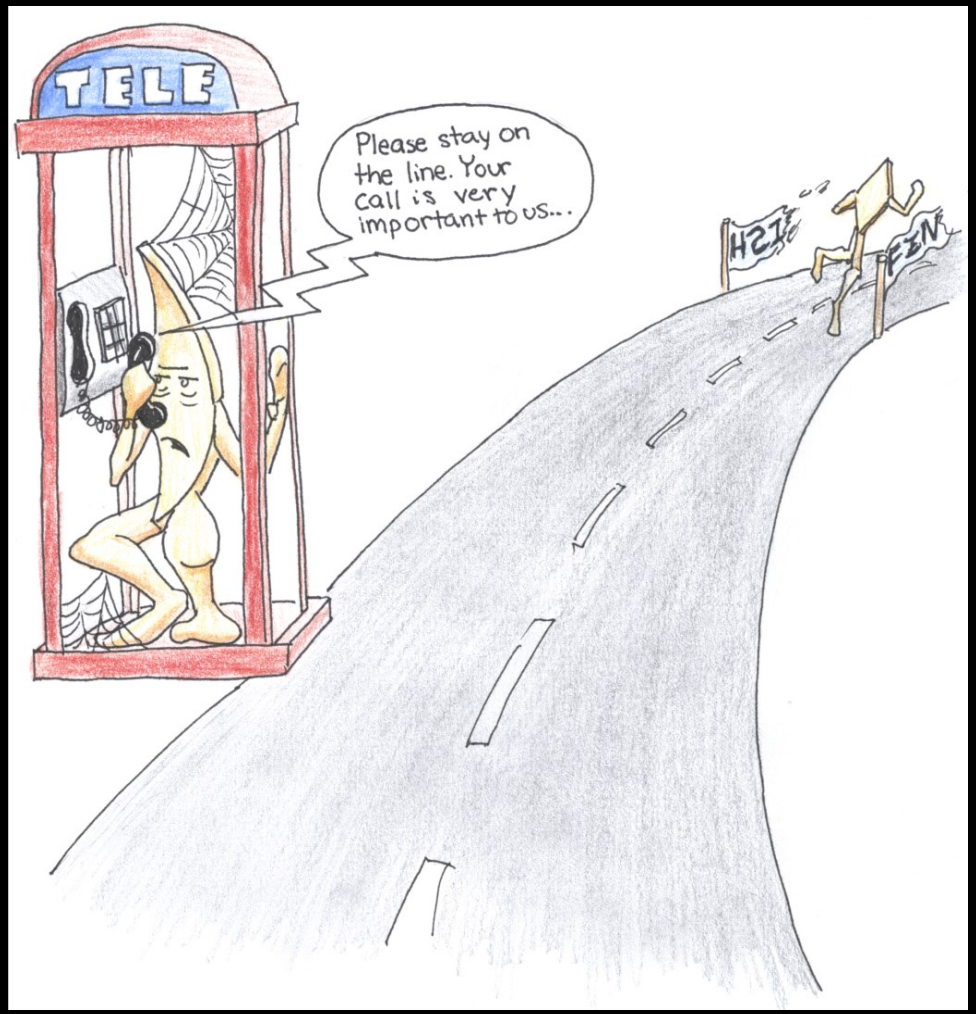


CPU Performance: Memory Barriers

CPU Performance: Cache Misses

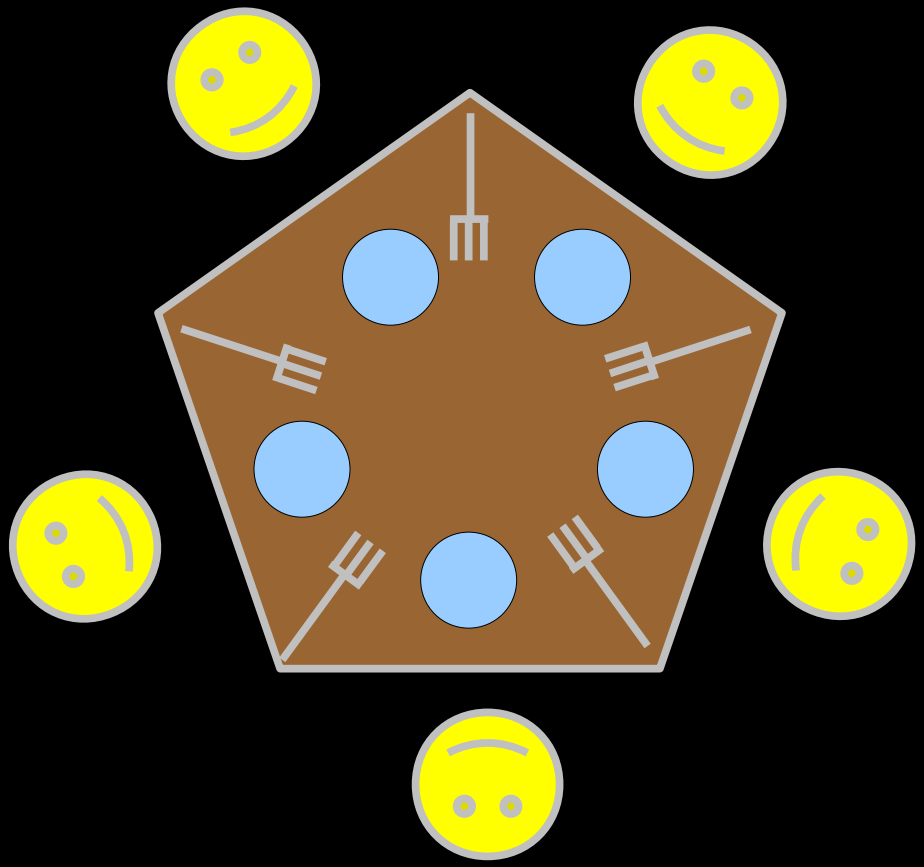


CPU Performance: I/O

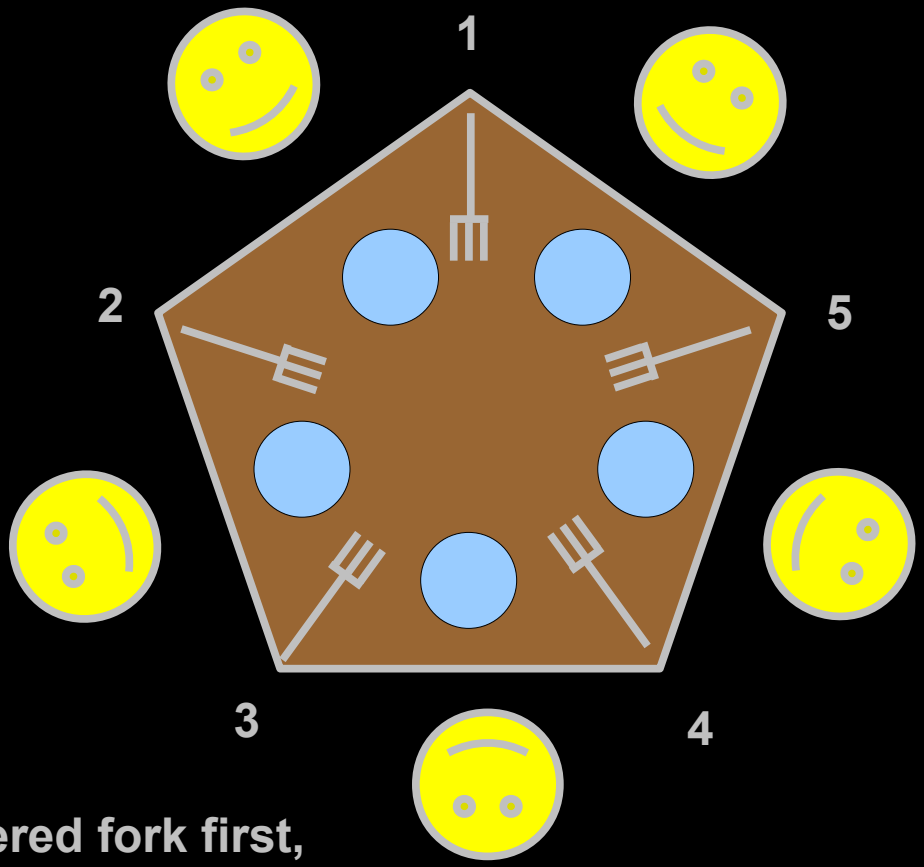


Exercise: Dining Philosophers Problem

Each philosopher requires two forks to eat.
Need to avoid starvation.



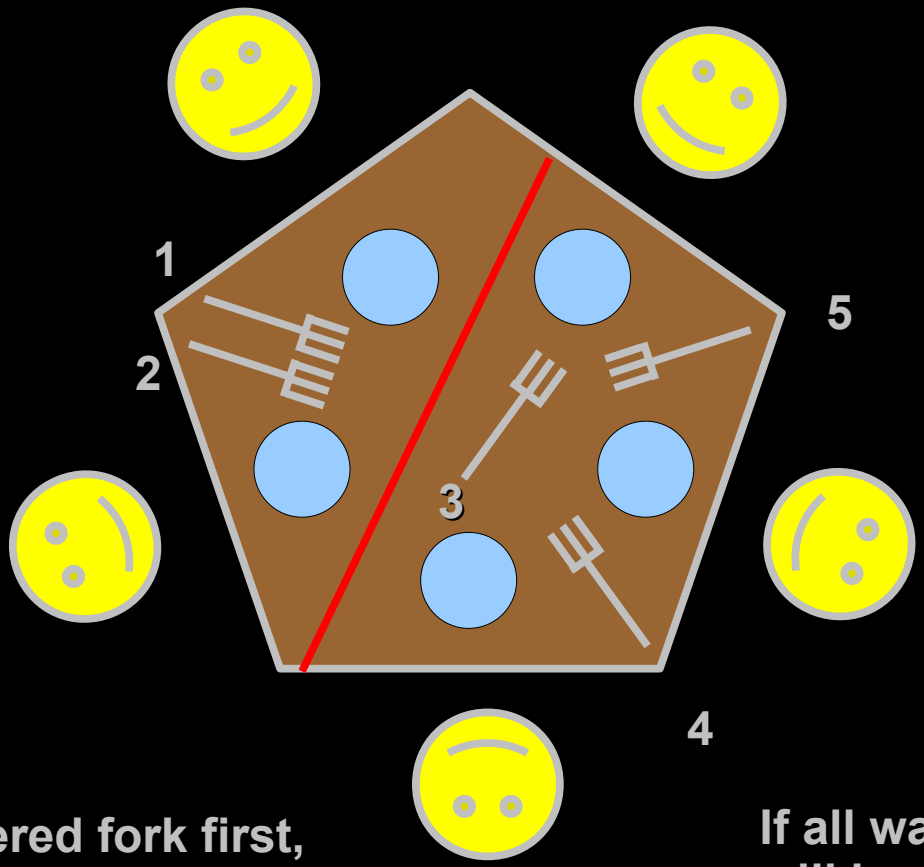
Exercise: Dining Philosophers Solution #1



Locking hierarchy.
Pick up low-numbered fork first, preventing deadlock.

Is this a good solution???

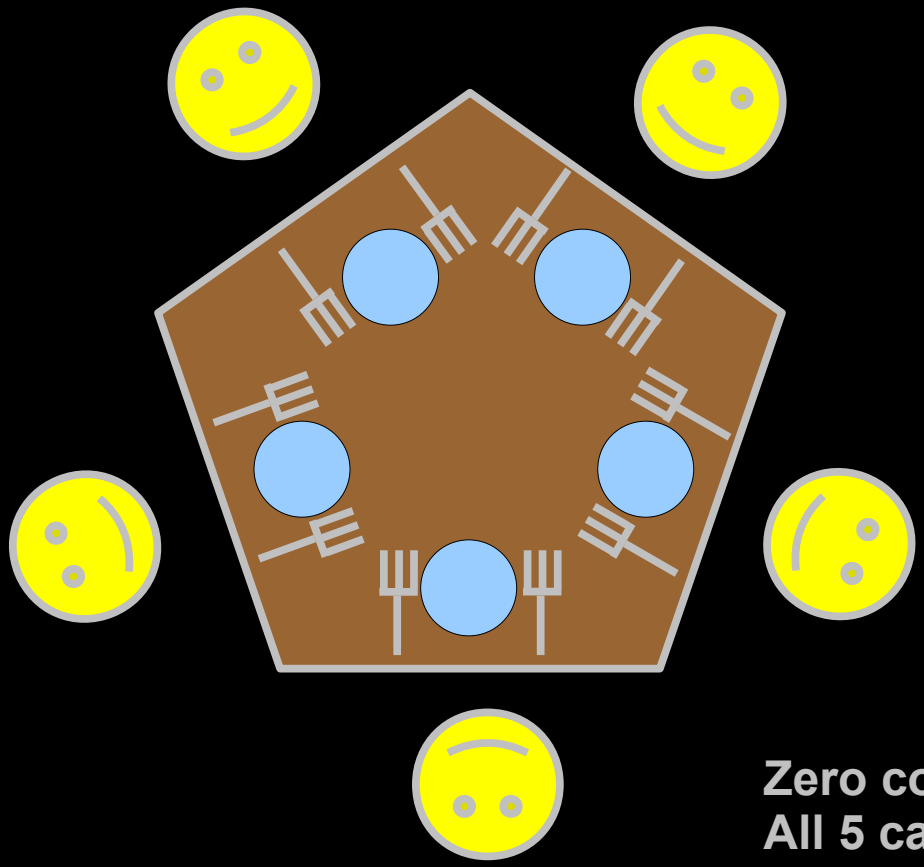
Exercise: Dining Philosophers Solution #2



Locking hierarchy.
Pick up low-numbered fork first, preventing deadlock.

If all want to eat, at least two will be able to do so.

Exercise: Dining Philosophers Solution #3



Zero contention.
All 5 can eat concurrently.
Excellent disease control.

Exercise: Dining Philosophers Solutions

- **Objections to solution #2 and #3:**
  - **"You can't just change the rules like that!!!"**
    - No rule against moving or adding forks!!!
  - **"Dining Philosophers Problem valuable lock-hierarchy teaching tool – #3 just destroyed it!!!"**
    - Lock hierarchy is indeed very valuable and widely used, so the restriction "there can only be five forks positioned as shown" does indeed have its place, even if it didn't appear in this instance of the Dining Philosophers Problem.
    - But the lesson of transforming the problem into perfectly partitionable form is also very valuable, and given the wide availability of cheap multiprocessors, most desperately needed.
  - **"But what if each fork cost a million dollars?"**
    - Then we make the philosophers eat with their fingers... ☺

But What To Do...

- What do you do for a problem that is inherently fine-grained (so that synchronization primitives such as locking, TM, NBS, &c are inefficient) and update-heavy (so that RCU is not helpful)?

But What To Do...

- What do you do for a problem that is inherently fine-grained (so that synchronization primitives such as locking, TM, NBS, &c are inefficient) and update-heavy (so that RCU is not helpful)?
  - Why not just write an optimized sequential program?
  - Or you can always invent something new!!!

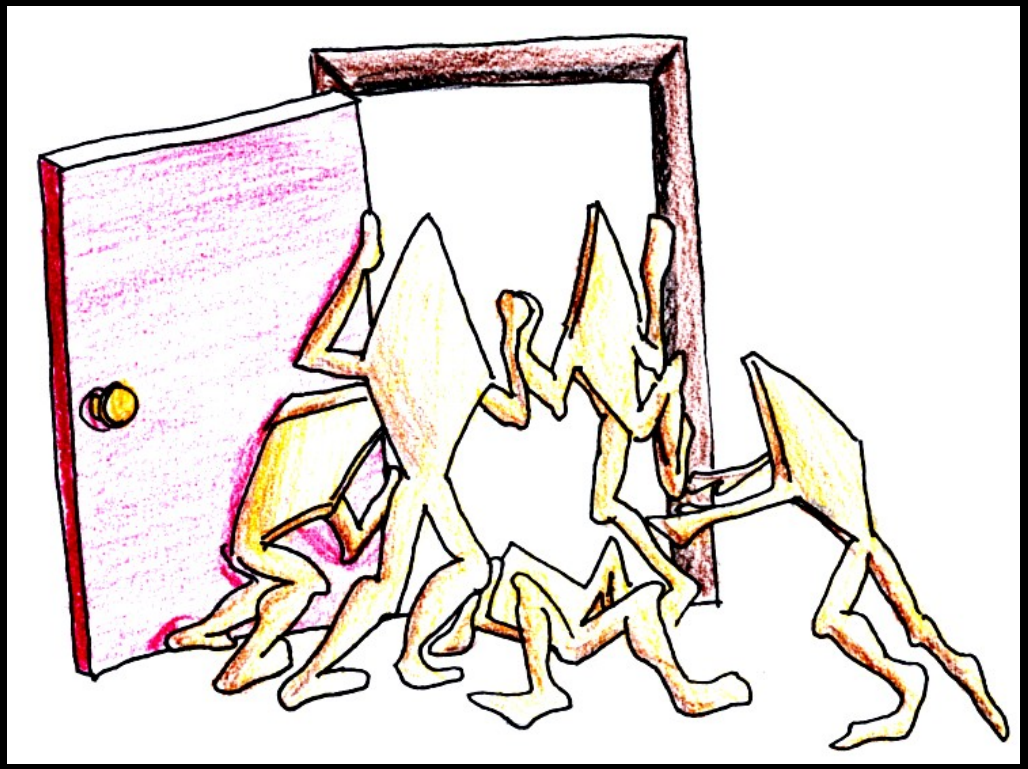
Exercise: High-Speed Concurrent Counting

- Need to maintain networking statistics
  - Packets transmitted and received
  - Bytes transmitted and received
- Packets might be transmitted or received on any CPU at any time
- Large machine capable of sending and receiving millions of packets per second
  - 40 microseconds per packet unacceptable!!!
- Systems-monitoring package reads out these statistics every five seconds
- Can you implement this?

Solution: High-Speed Concurrent Counting

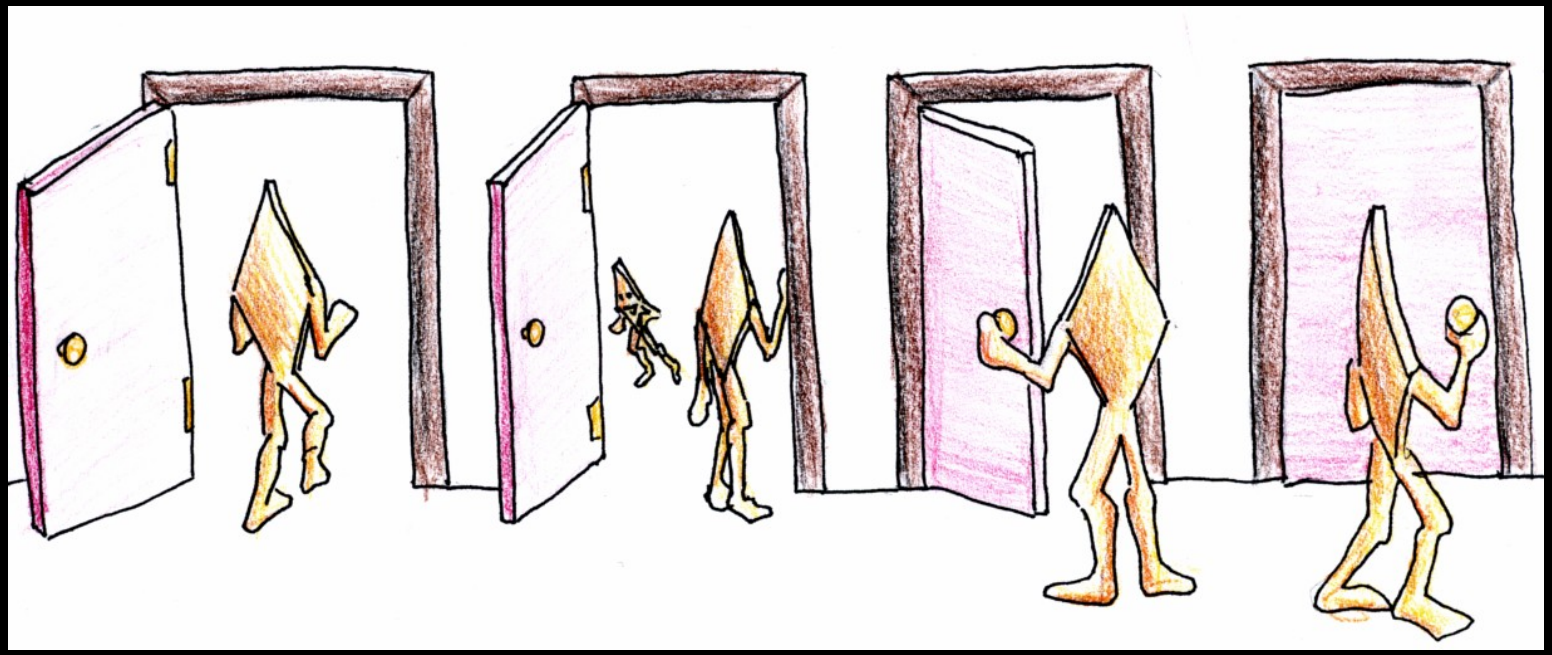
- Maintain per-CPU counters for each datum
  - Packets transmitted
  - Packets received
  - Bytes transmitted
  - Bytes received
- Each CPU updates its own counter
  - __get_cpu_var(packets_xmitted)++;
- To read out current value, sum all copies of the desired counter
  - Slow, but doesn't happen very often, so OK

Solution: High-Speed Concurrent Counting



One variable bad

Solution: High-Speed Concurrent Counting



Many variables good

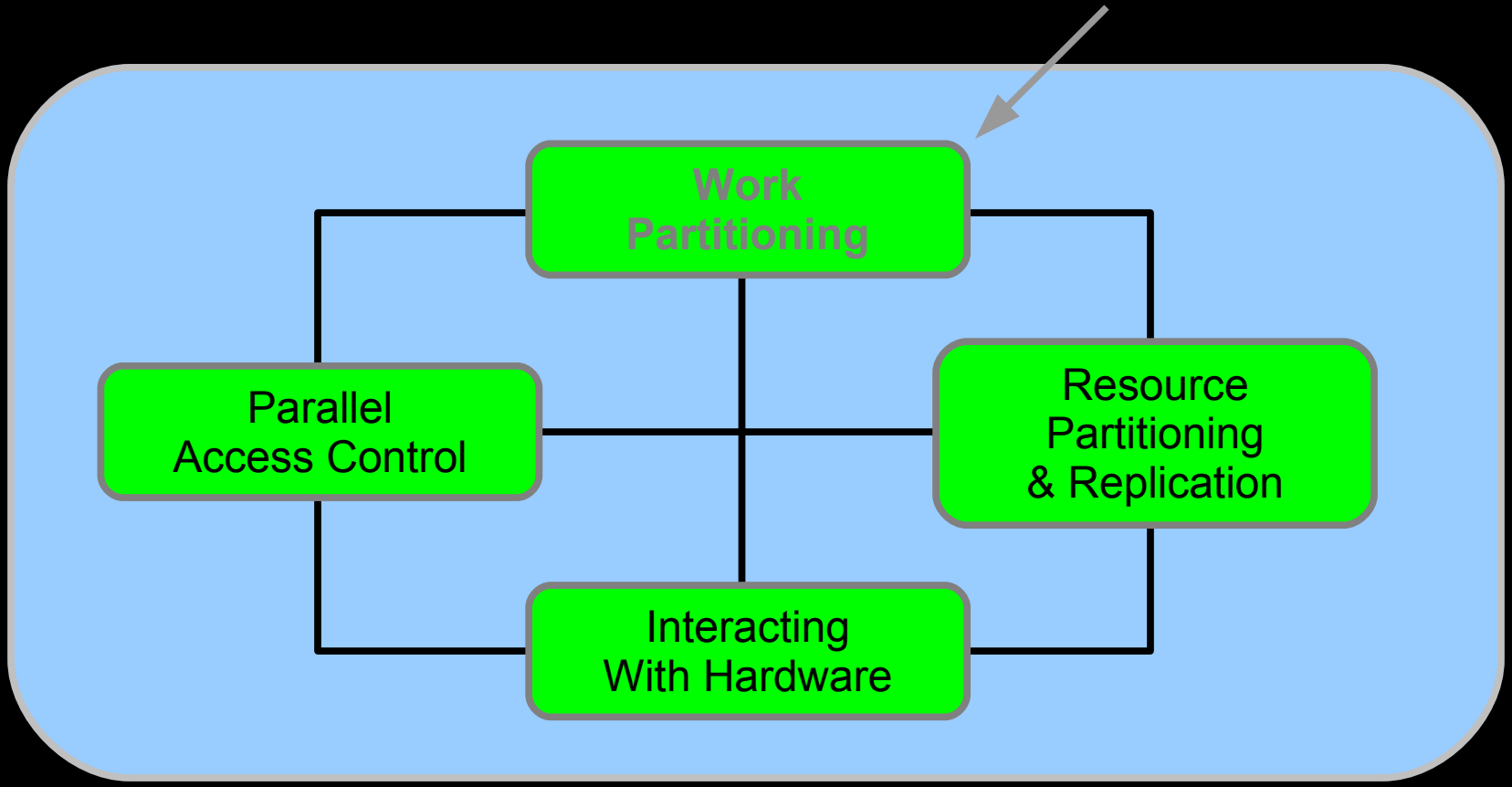
Conclusions

Summary and Problem Statement

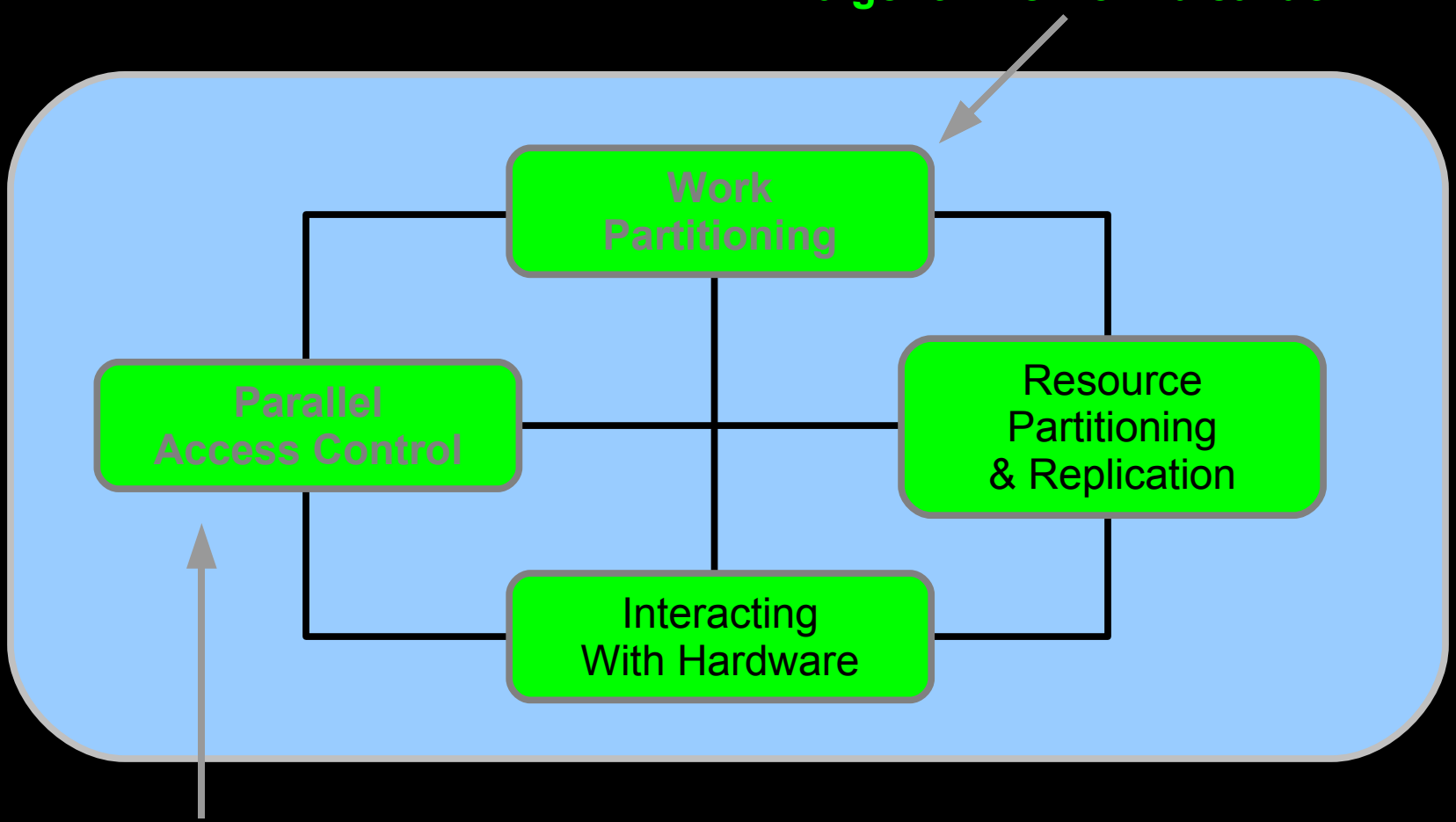
- Parallel Research and Development:
  - High productivity and high performance (specialized apps)
    - Remember what the spreadsheet did for the PC!!!
  - Generality and high performance (infrastructure)
    - For the experts developing the above apps
  - Generality and high productivity
    - But only if some advantage over sequential environment!!!
- Problem Statements:
  - Work breakdown, primitives, then partitioning
  - Generality, performance, then productivity

Problem Statement #1: Parallel Pitfall

Problem Statement #1

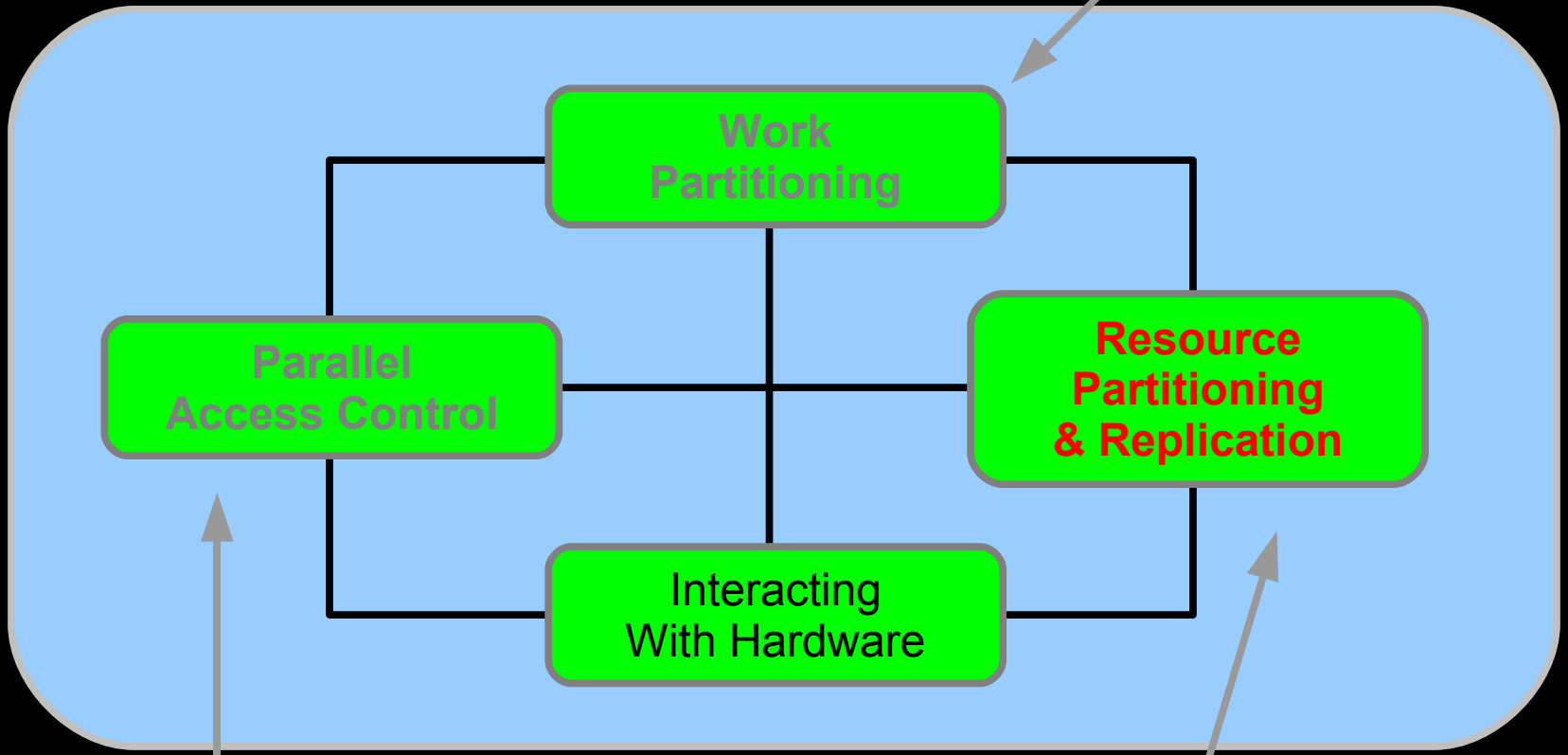


Problem Statement #1



Problem Statement #1

Start with preconceived algorithmic work breakdown



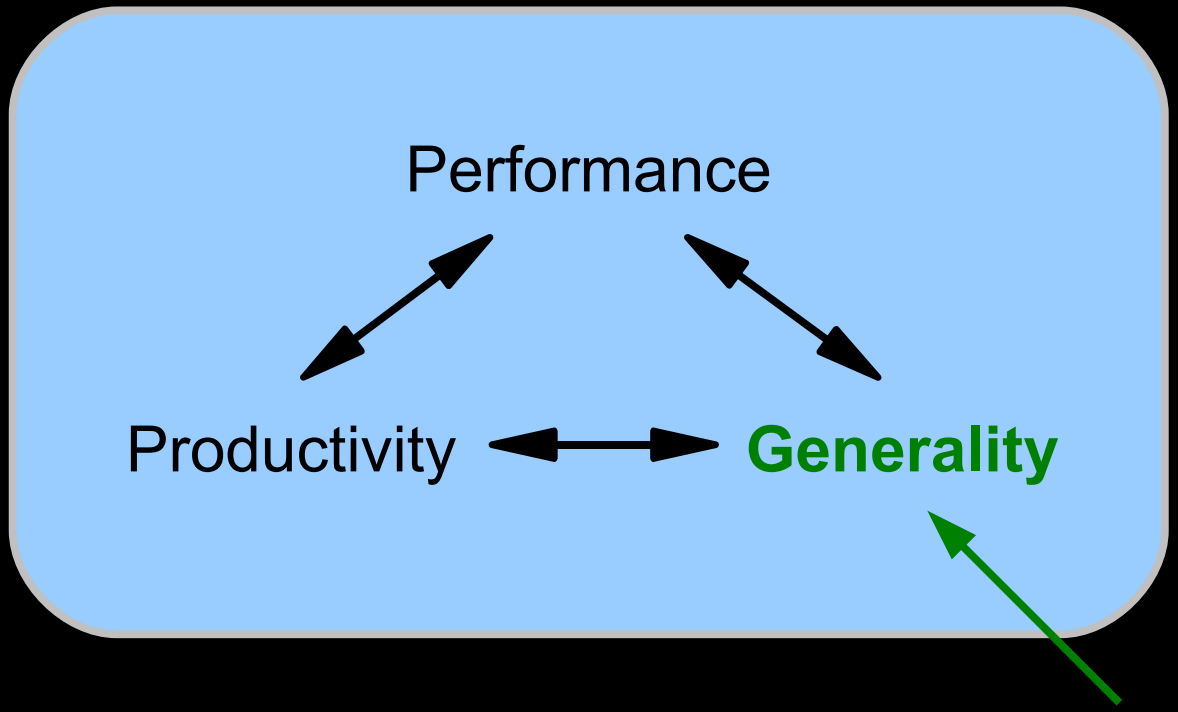
Choose synchronization mechanism

No attention to partitioning and replication: *Poor scalability and performance!!!*

Problem Statement #2: Take Over The World!!!

Narf!!!

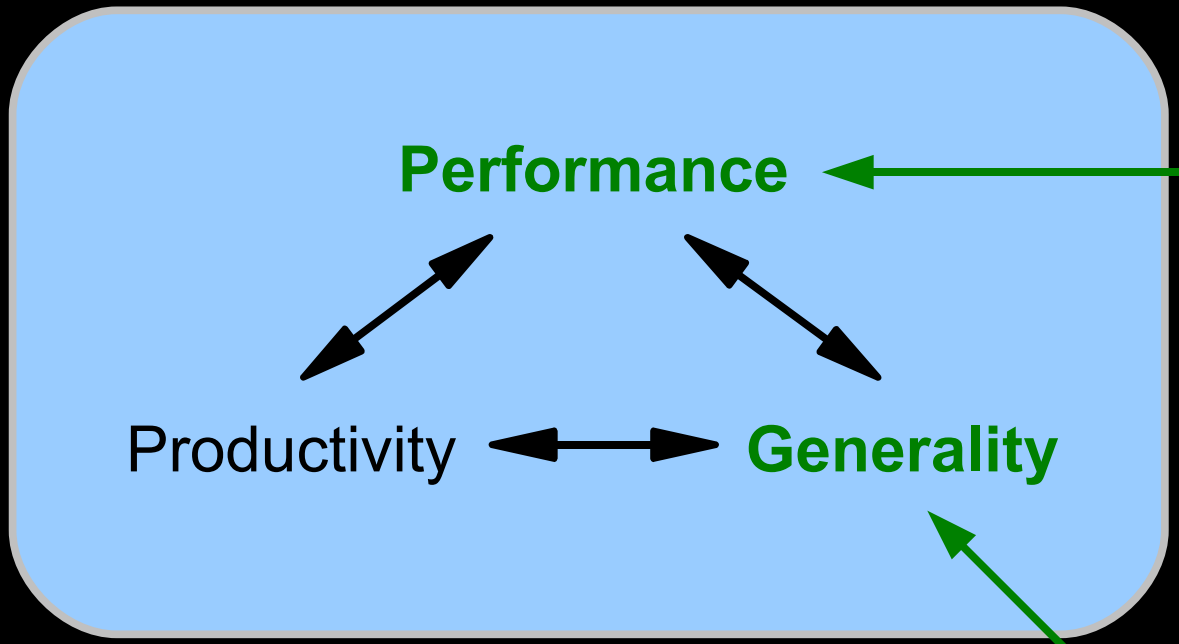
Problem Statement #2



Job #1 to "Take Over the World"

But now a choice: Performance? or Productivity?

Problem Statement #2

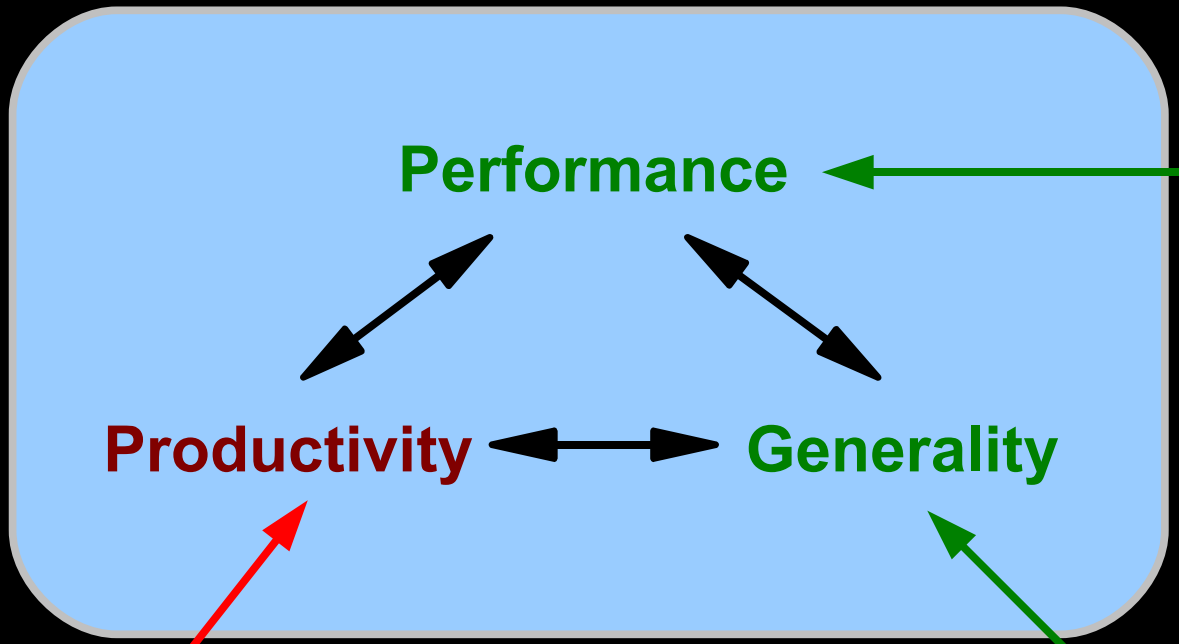


Also need to "Be the Best"

Job #1 to "Take Over the World"

After all, publishing performance improvements is much easier than publishing productivity results!

Problem Statement #2

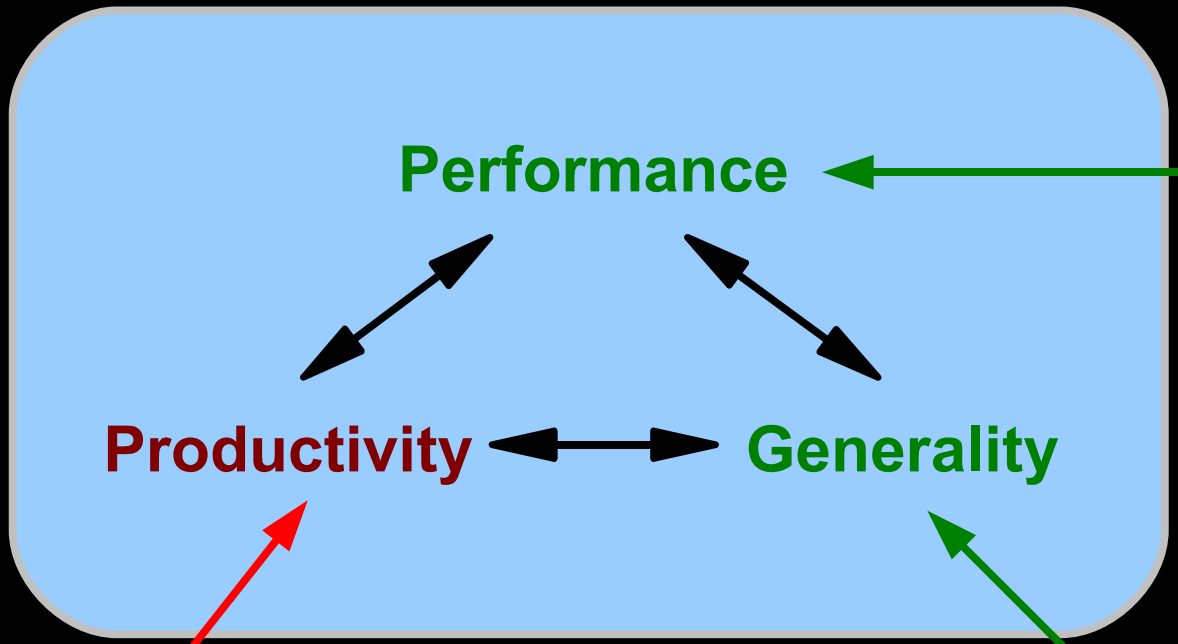


Also need to "Be the Best"

Which means poor productivity...

Job #1 to "Take Over the World"

Problem Statement #2



Also need to "Be the Best"

Which means poor productivity...

Job #1 to "Take Over the World"

And then these people have the gall to complain that parallel programming is hard!!!

If You Really Want to Take Over the World...

If You Really Want to Take Over the World...

Remember what the spreadsheet and word processor did for the personal computer.

If You Really Want to Take Over the World...

Remember what the spreadsheet and word processor did for the personal computer.

Then focus on solving a specific problem really well.

Sometimes, generality can be a shot in the foot!!!

Is Parallel Programming Hard, And If So, Why?

Parallel Programming is as Hard or as Easy as We Make It.

It is that hard (or that easy) because we make it that way!!!

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.
- This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851.
  - Joint work with Manish Gupta, Maged Michael, Phil Howard, Joshua Triplett, and Jonathan Walpole

Questions?

To probe further:

- Any physics text
- Any queuing-theory text
- Computer Architecture: A Quantitative Approach, Hennessy and Patterson

- But there is no substitute for running tests on real hardware!!!
  - For examples, see “CodeSamples” directory in:
    git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git