2009 linux.conf.au Kernel Miniconf

Is Parallel Programming Hard, And If So, Why?

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

January 19, 2009

Credits

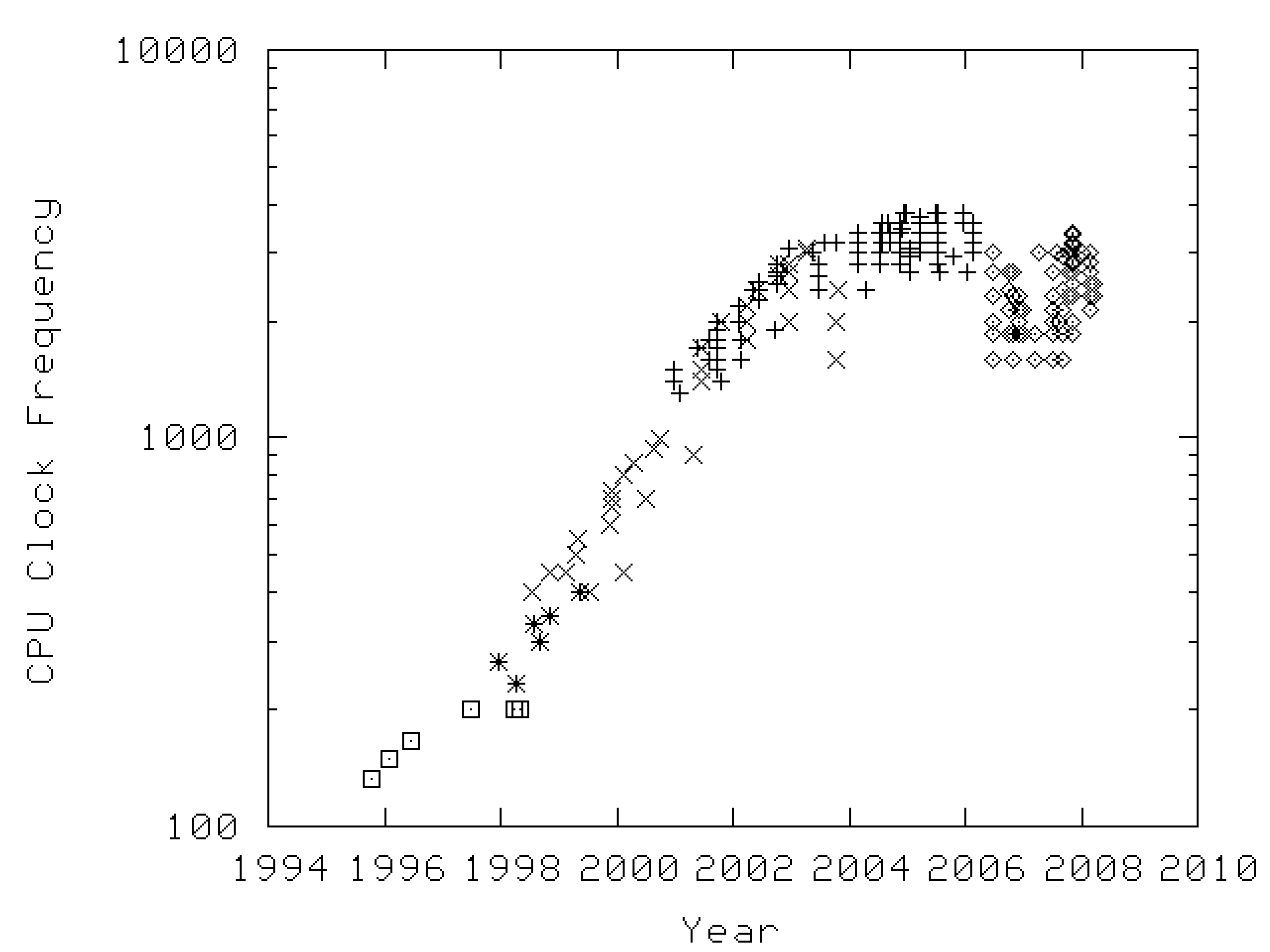
Joint work with Manish Gupta, Maged Michael, Phil Howard, Joshua Triplett, and Jonathan Walpole

Overview

- Why Parallel Programming?
- Parallel Programming Goals
- Parallel Programming Tasks
- Performance of Synchronization Operations
- Do “Tasks” Relate to Real-World Software?
- Conclusions

Why Parallel Programming?

Why Parallel Programming? (Party Line)

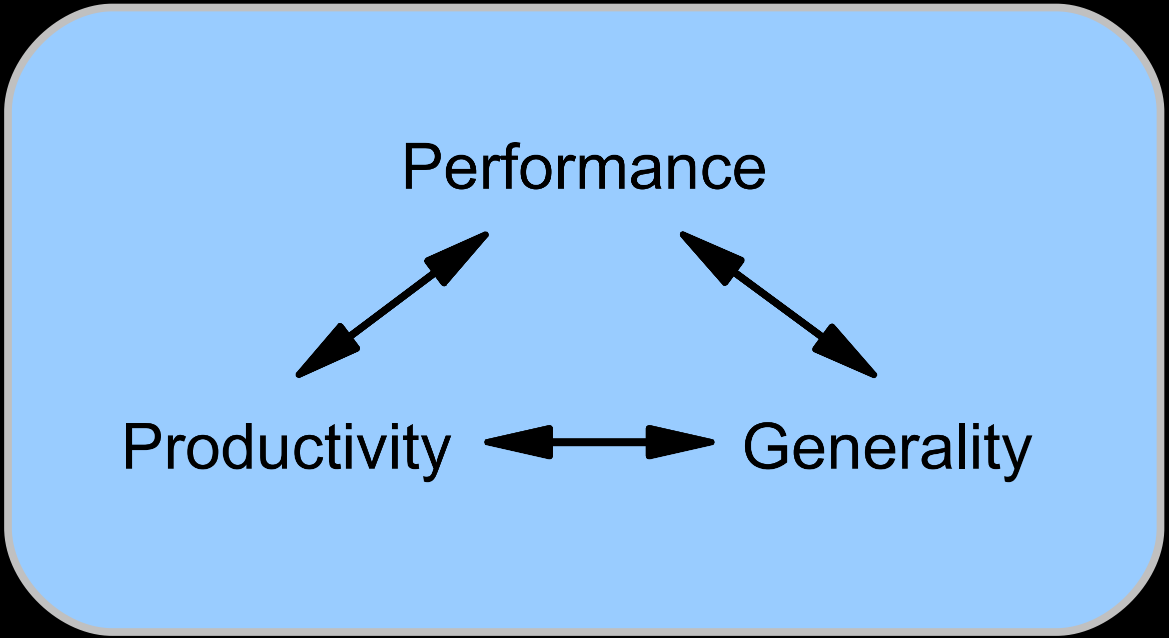


Why Parallel Programming? (Reality)

- 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, ...
  - More on this later!

Parallel Programming Goals

Parallel Programming Goals



Parallel Programming Goals: Why Performance?

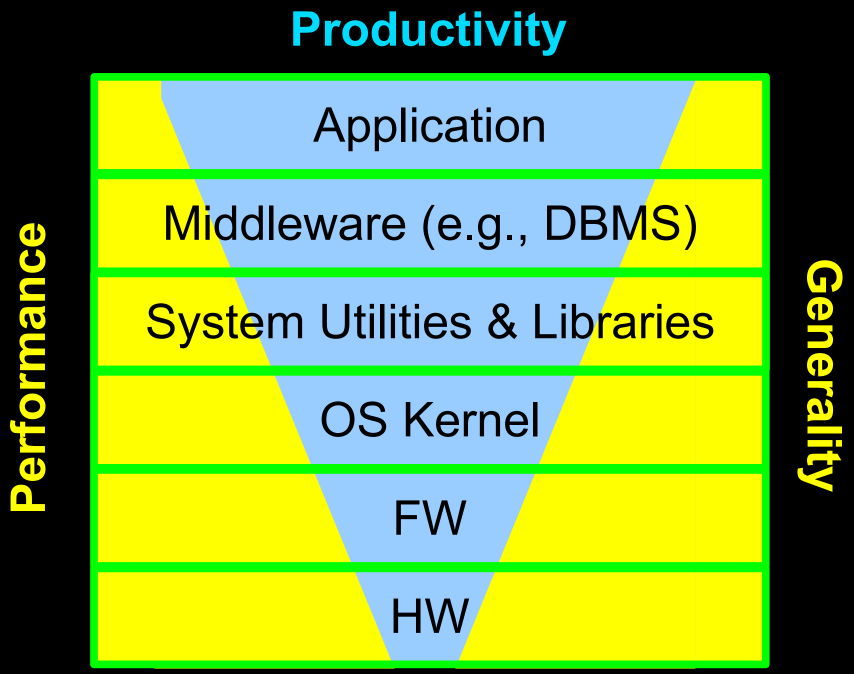
- (Performance often expressed as scalability or normalized as in performance per watt)
- If you don't care about performance, *why* are you bothering with parallelism???
  - Just run single threaded and be happy!!!
- But what about:
  - All the multi-core systems out there?
  - Efficient use of resources?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

Parallel Programming Goals: Why Productivity?

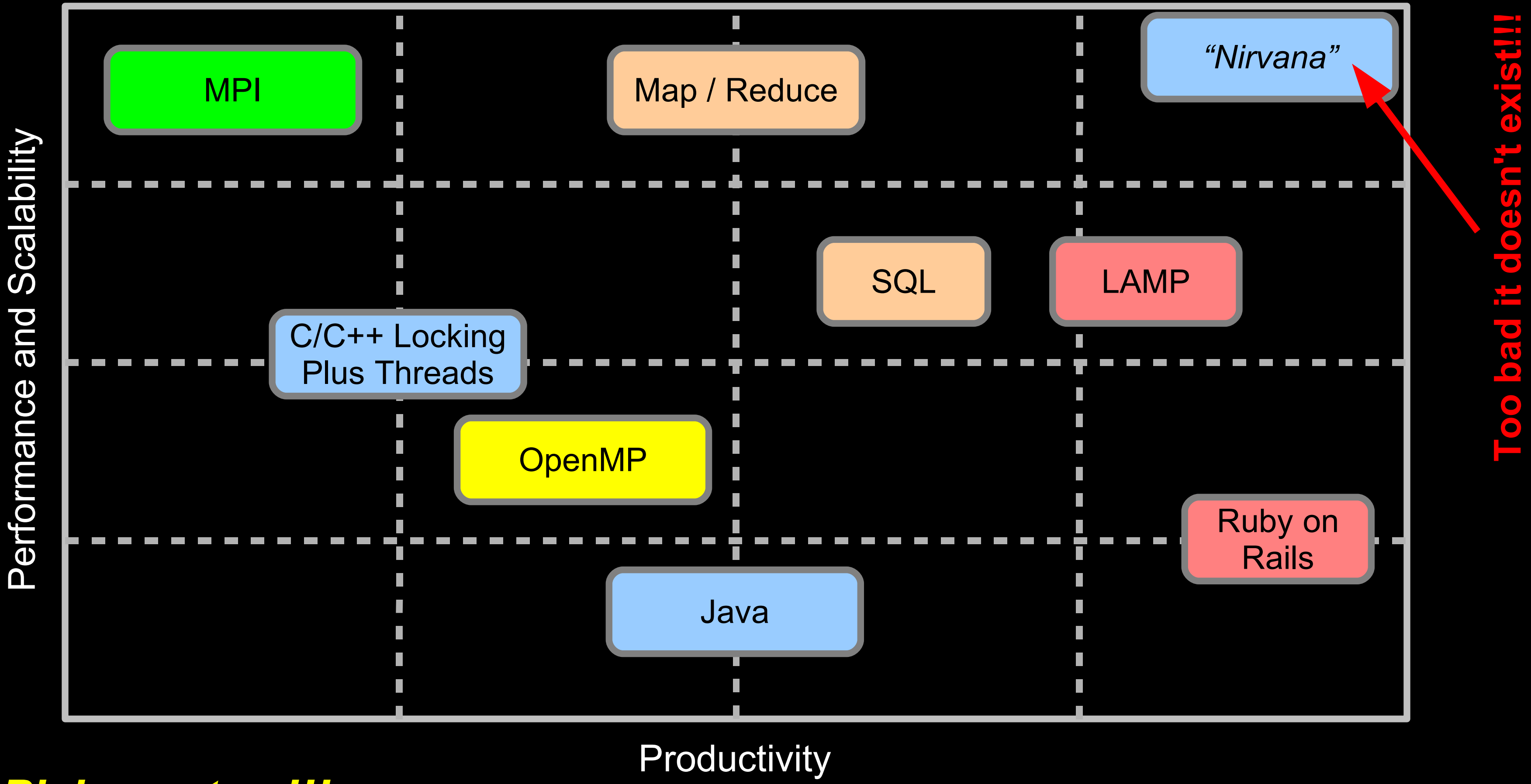
- 1948 CSIRAC (oldest intact computer)
  - 2,000 vacuum tubes, 768 20-bit words of memory
  - $10M AU construction price
  - 1955 technical salaries: $3-5K/year
  - Makes business sense to dedicate 10-person team to increasing performance by 10%
- 2008 z80 (popular 8-bit microprocessor)
  - 8,500 transistors, 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

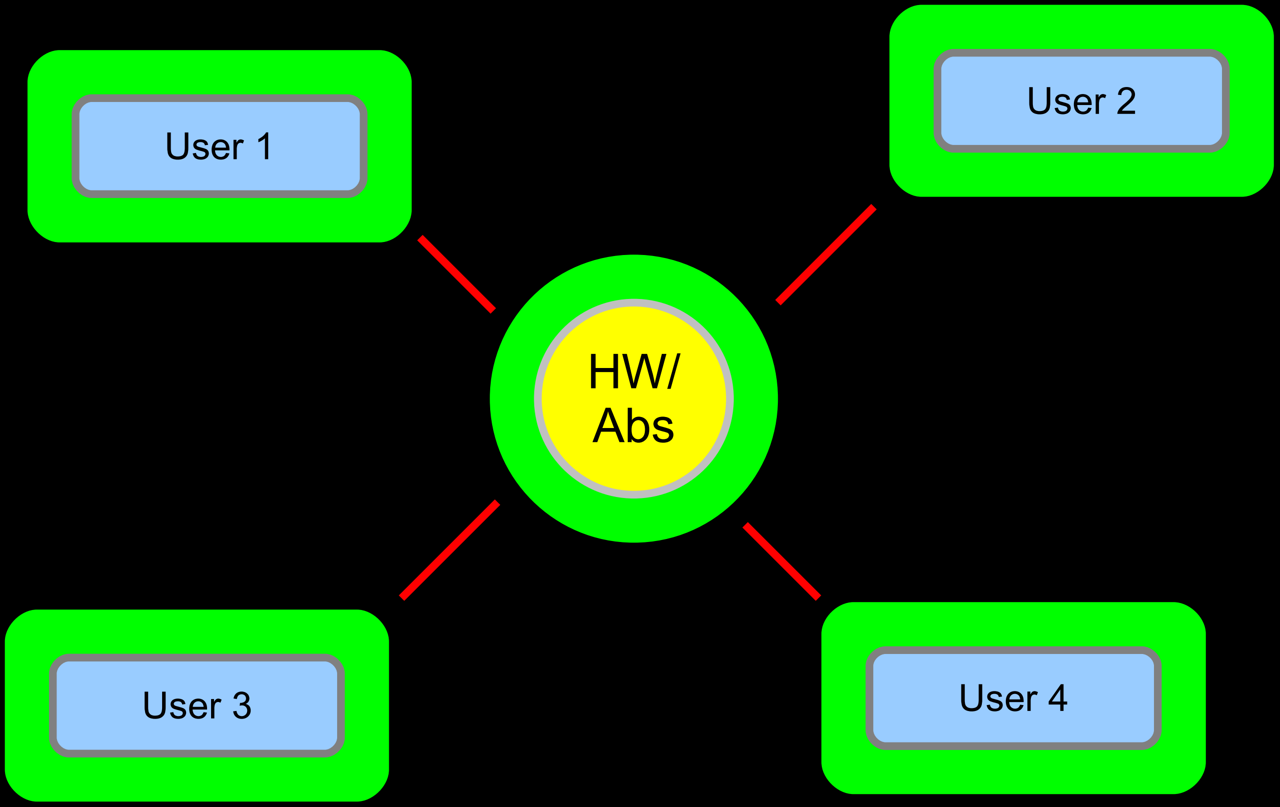

MPI
Map / Reduce
"Nirvana"
SQL
LAMP
C/C++ Locking Plus Threads
OpenMP
Ruby on Rails
Java
Performance and Scalability
Productivity
Too bad it doesn't exist!!!

Pick any two!!!

Why Are Environments Specialized?

- C/C++ Locking Plus Threads
  - General purpose (and the only one useful for Linux kernel work)
- Java
  - General purpose
- MPI
  - Theoretically general purpose, but used primarily for HPC
- OpenMP
  - Parallel loops, primarily HPC (parallelize single control flow)
- SQL
  - Relational database (not good for tree/graph-structured data)
- Map/Reduce
  - “Shardable” applications with no cross-shard dependencies
- LAMP
  - Relational database with web presence
- Ruby on Rails
  - Relational database with web presence without legacy database

Why Are Environments Specialized?

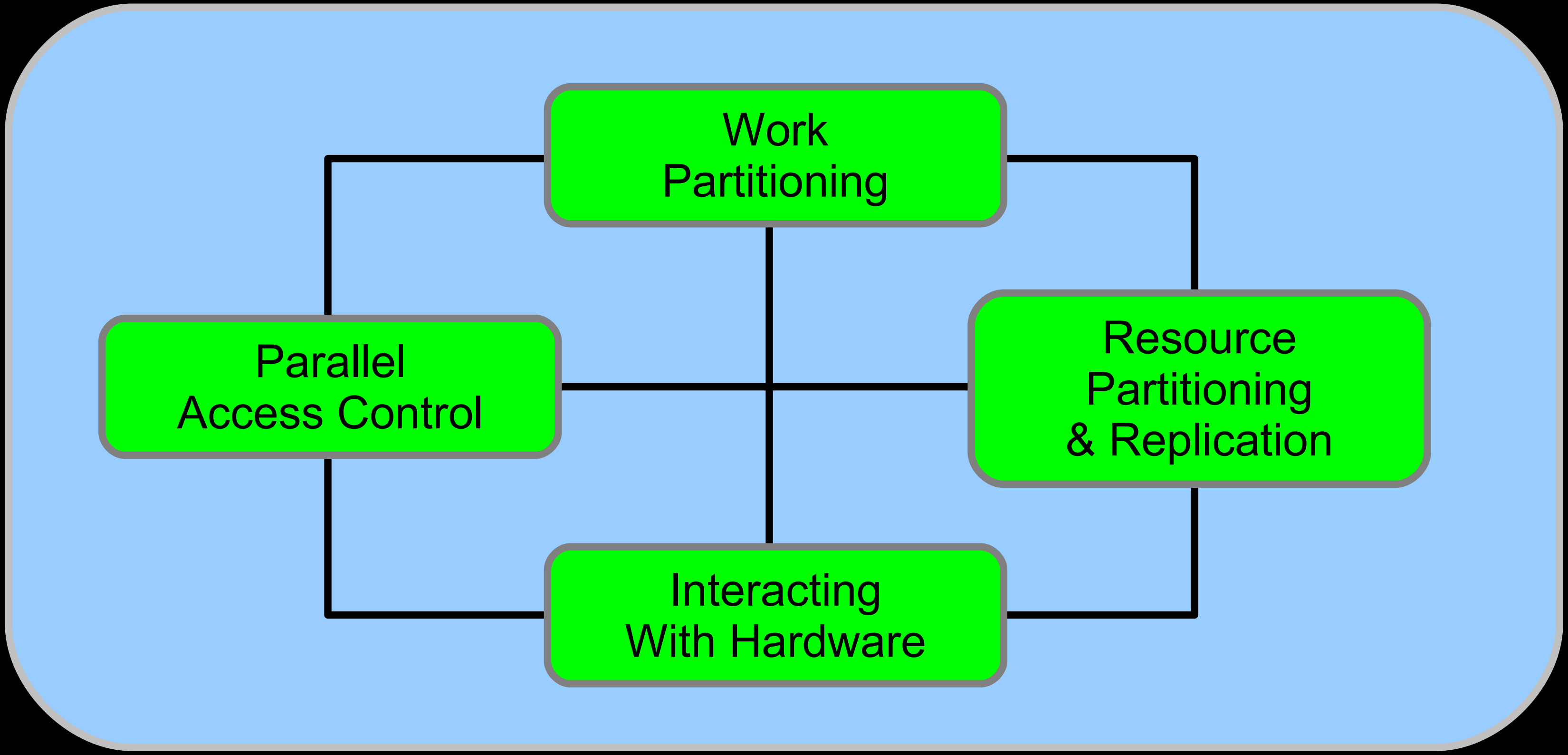


Parallel Programming Tasks

Parallel Programming Tasks

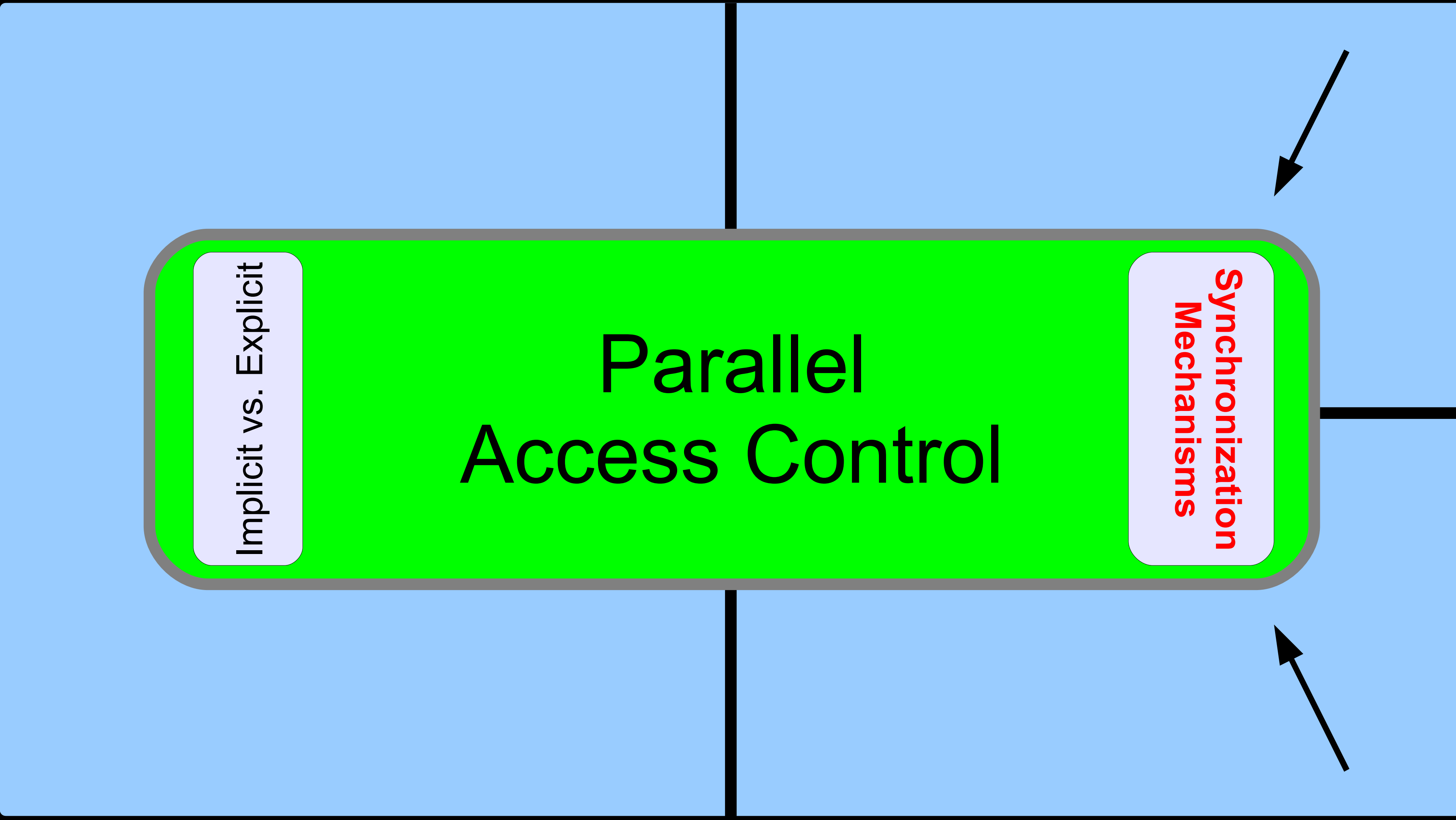
- Parallel Programming Only Partly Technical
  - Human element is extremely important
  - What can a human being easily construct and read?
    - Similar to stylized English used in emergency situations
    - Clarity, concision, and unambiguity trump style and grace
- In a perfect world, use human-factors 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

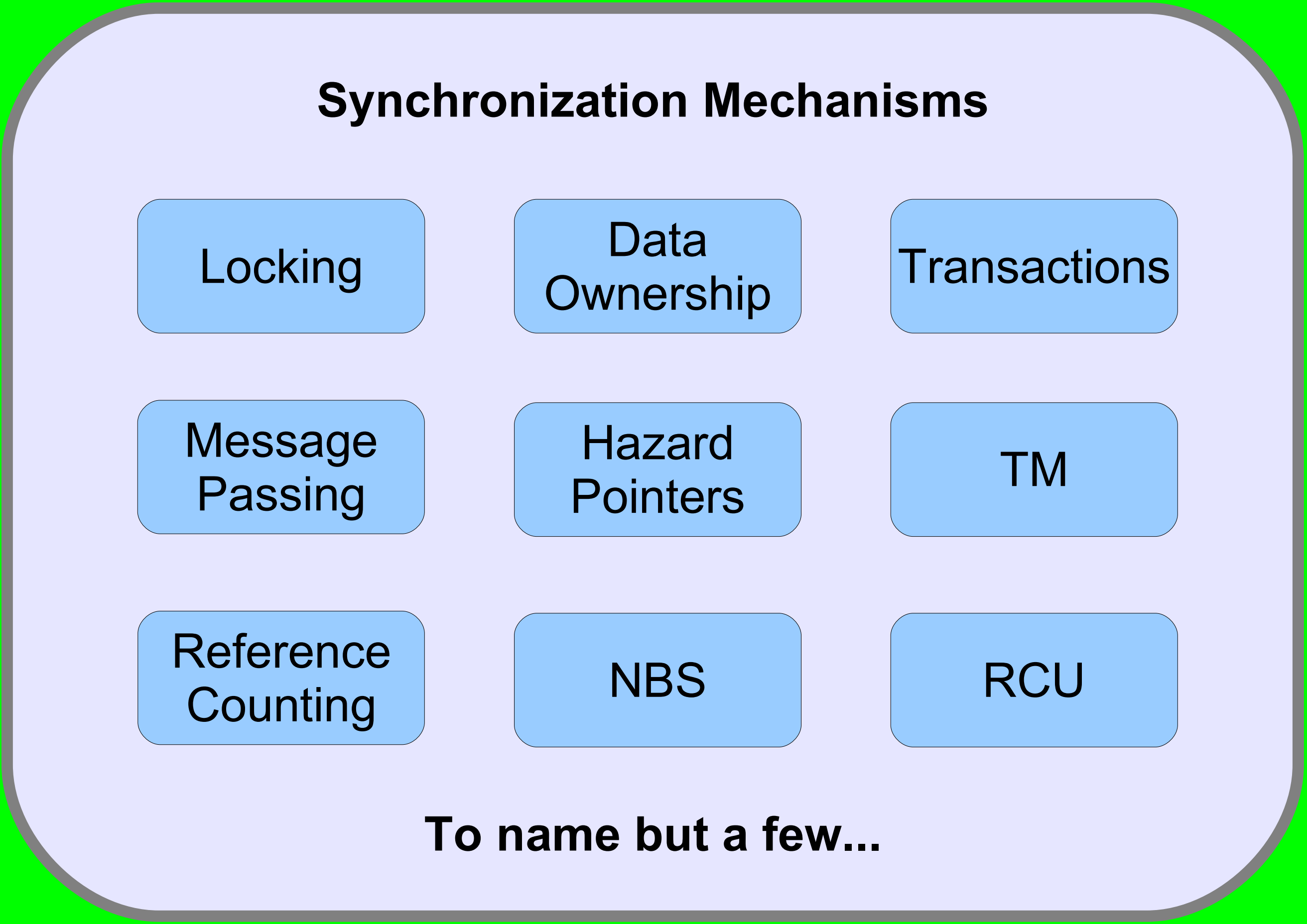

Work Partitioning
Parallel Access Control
Resource Partitioning & Replication
Interacting With Hardware

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

TM

Reference Counting

NBS

RCU

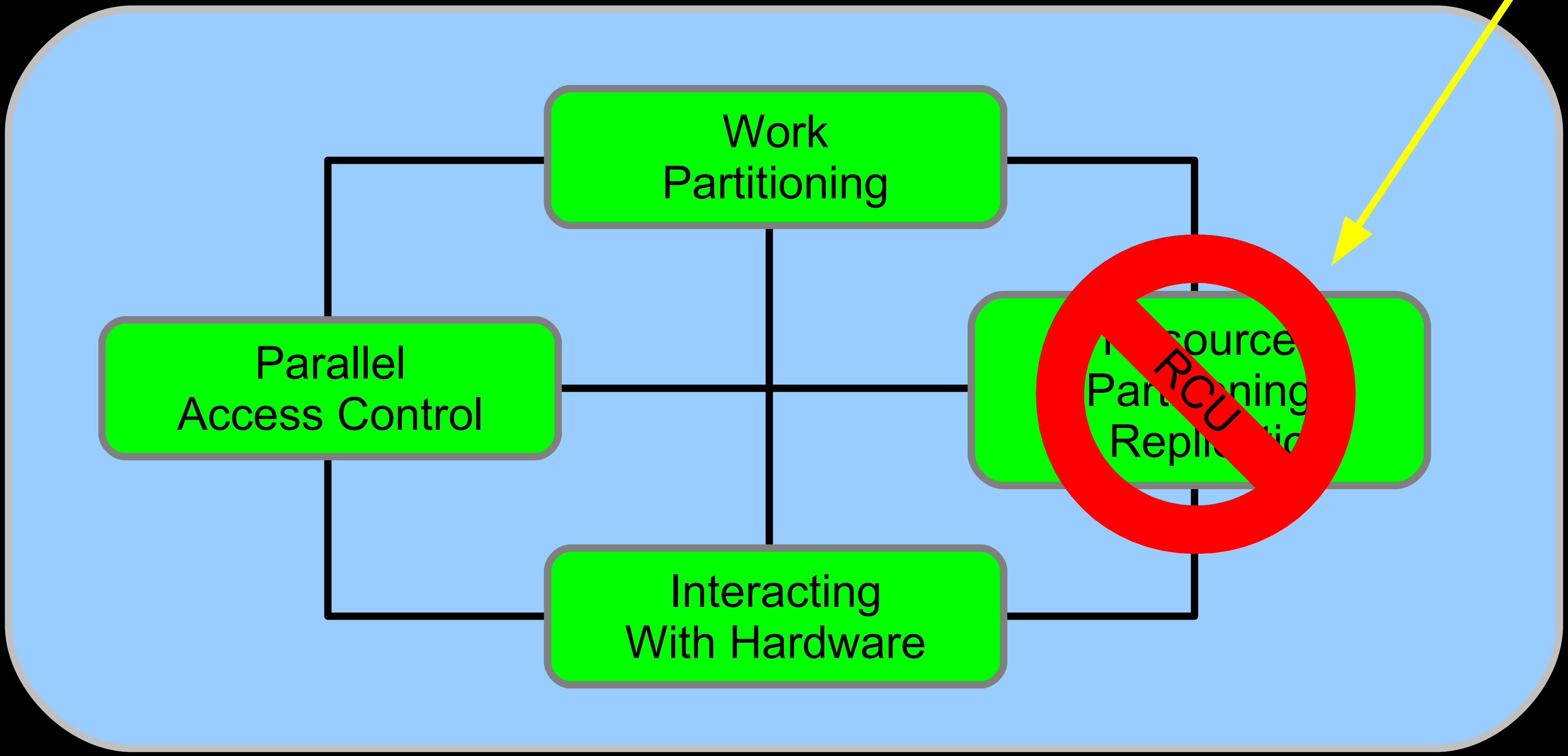
To name but a few...

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....

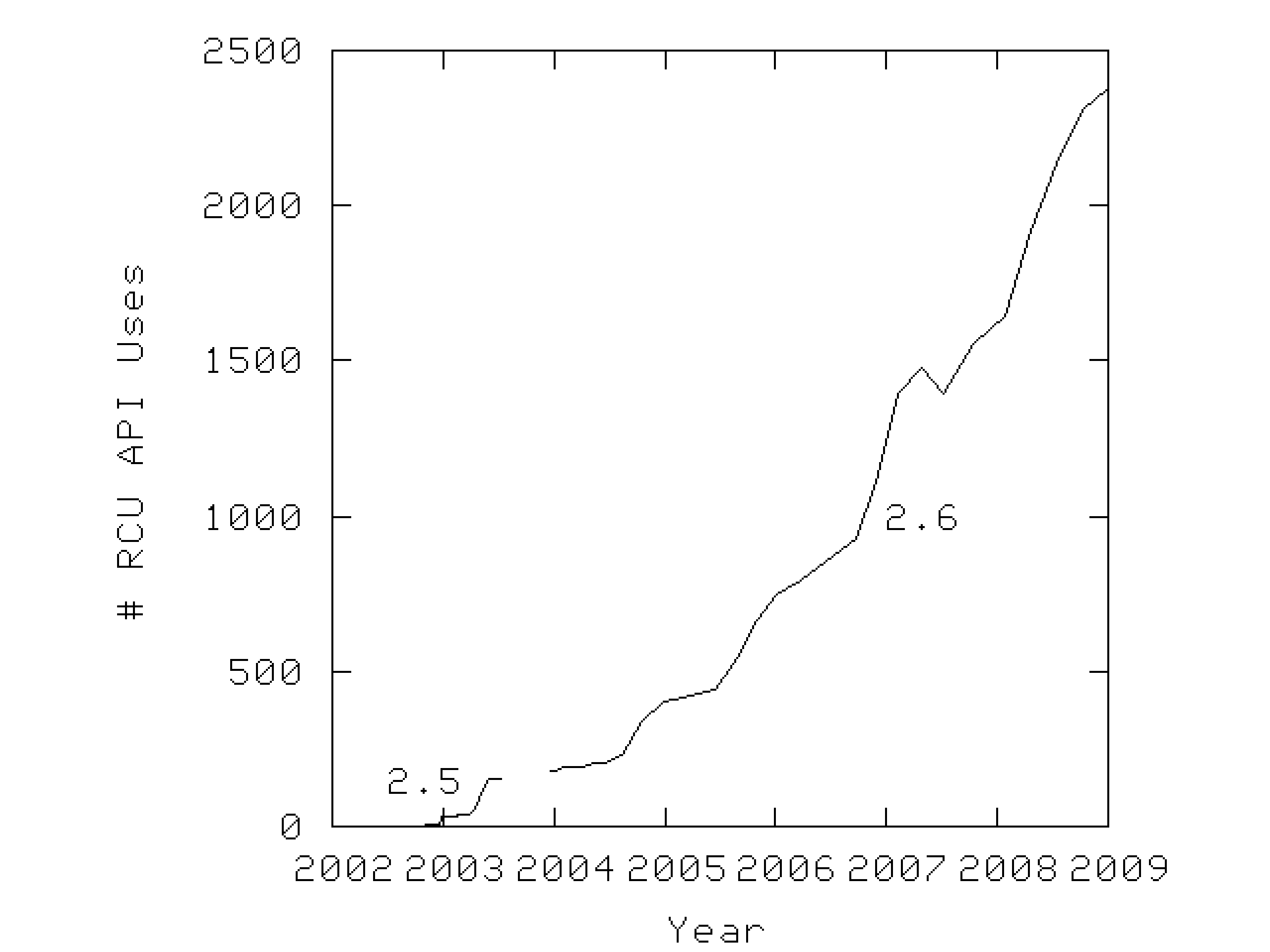
RCU for Read-Mostly Data Structures

Almost...

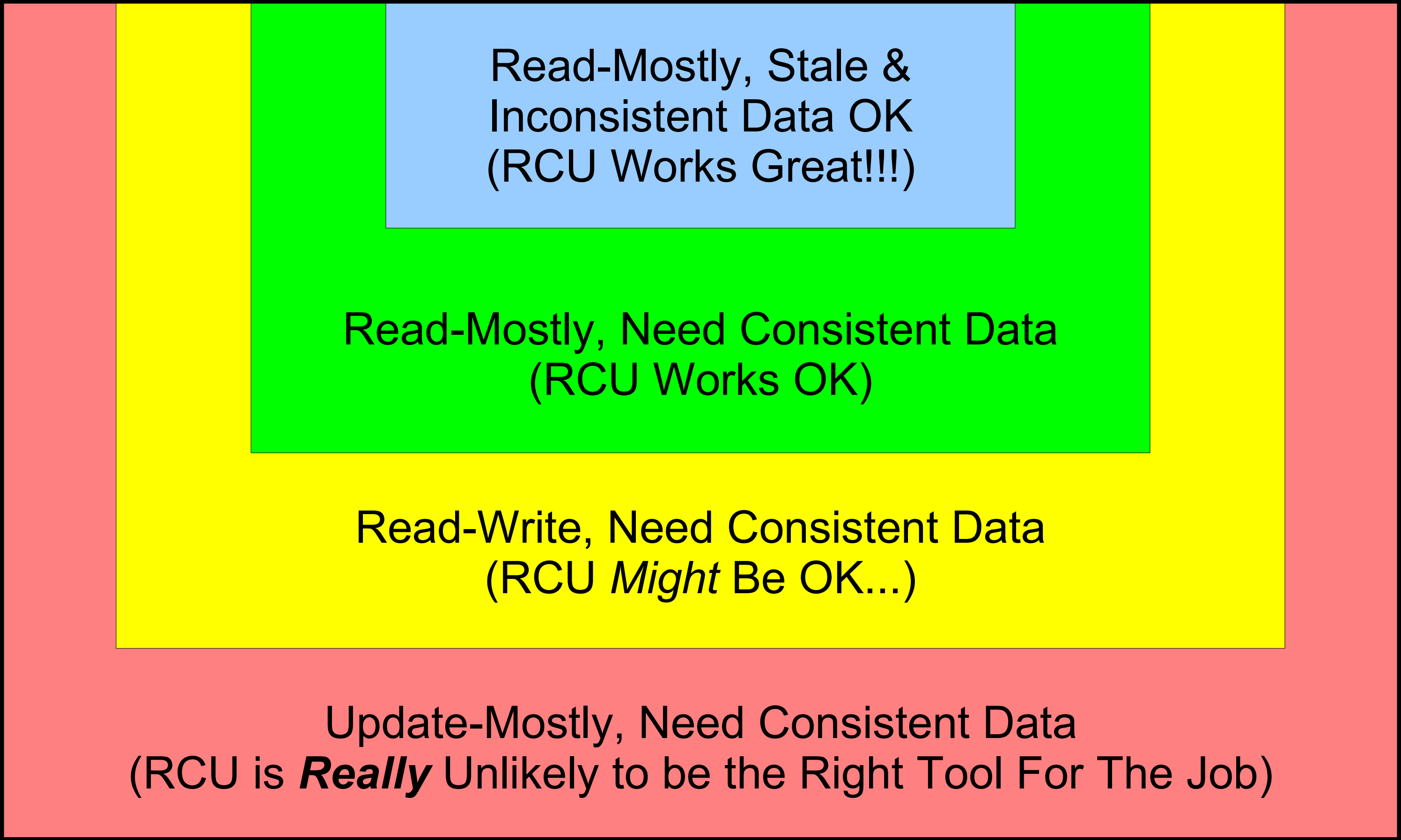


RCU data-parallel approach: ~~first partition resources~~, then partition work, and only then worry about parallel access control, and only for updates.

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)

Performance of Synchronization Mechanisms

Performance of Synchronization Mechanisms

Need to be here!
(Partitioning/RCU)

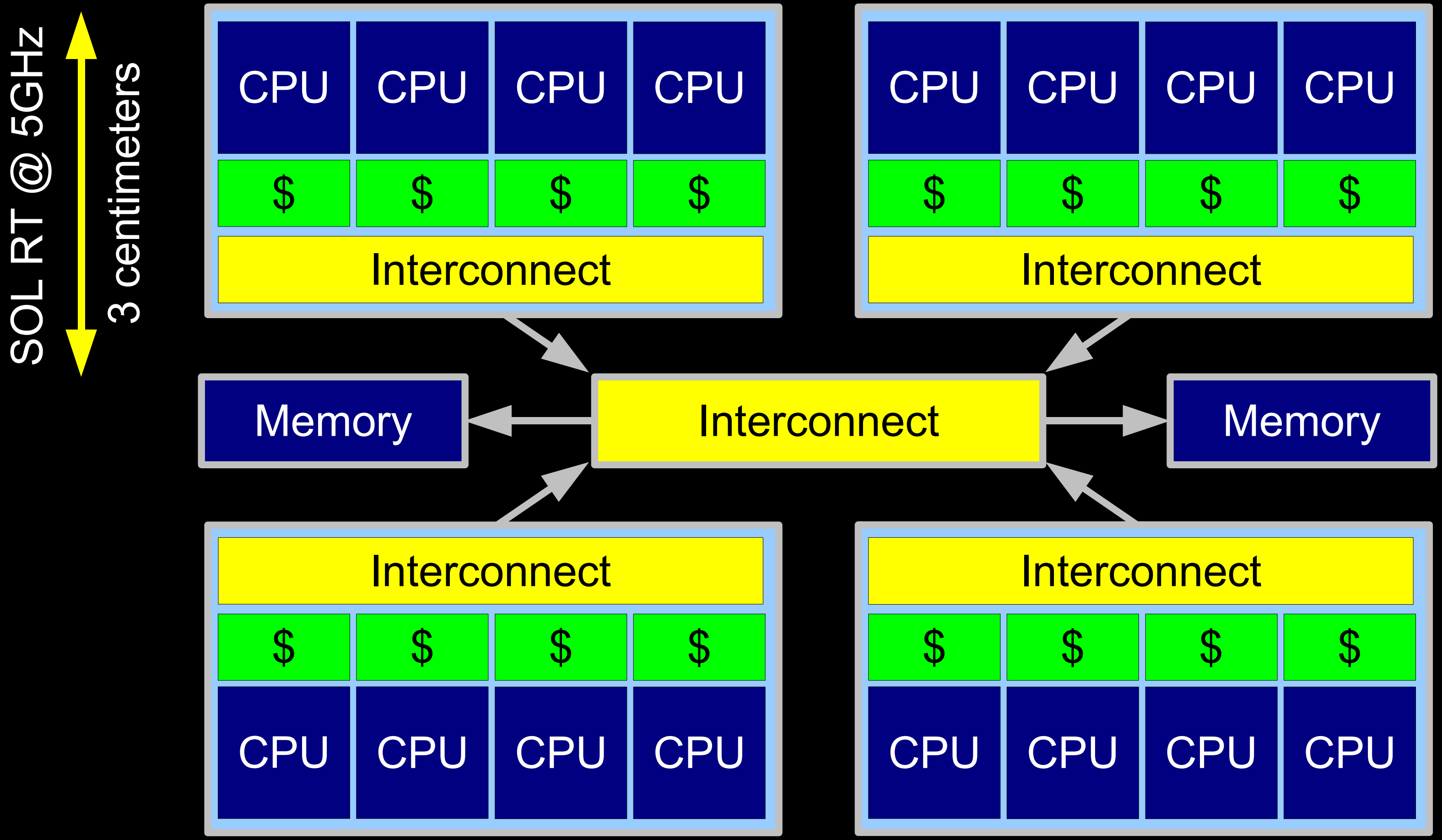
4-CPU 1.8GHz AMD Opteron 844 system

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

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

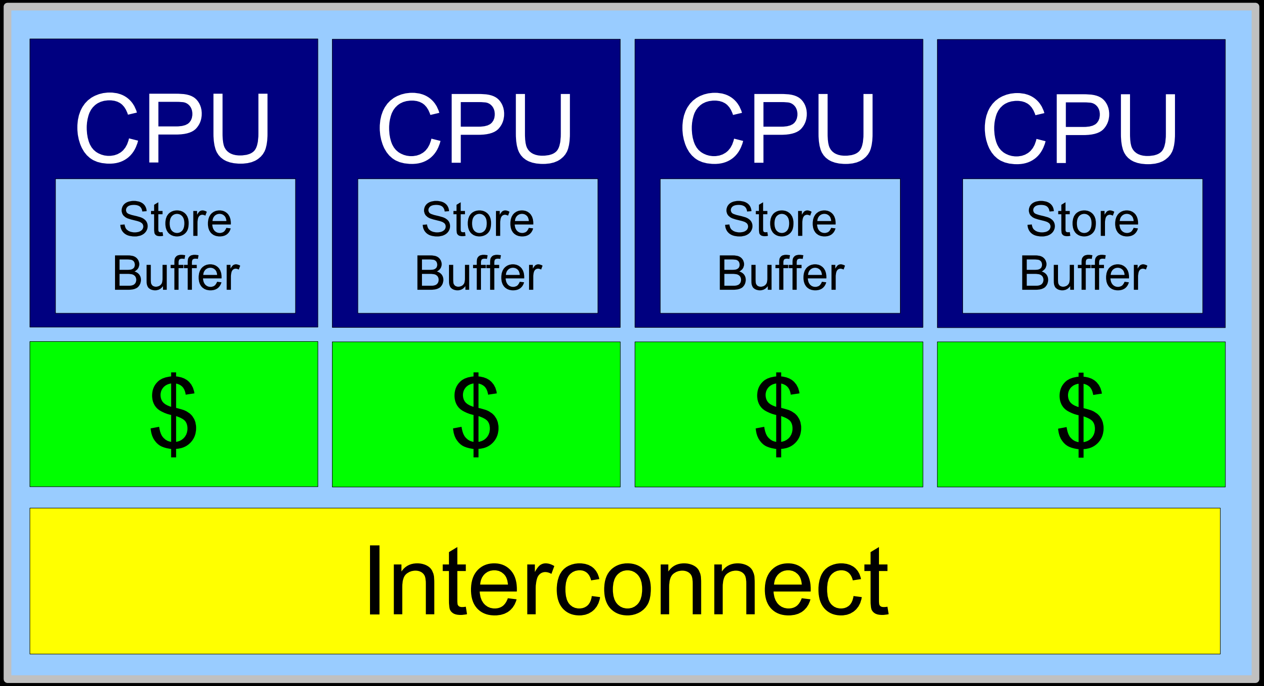
Typical synchronization mechanisms do this a lot

System Hardware Structure

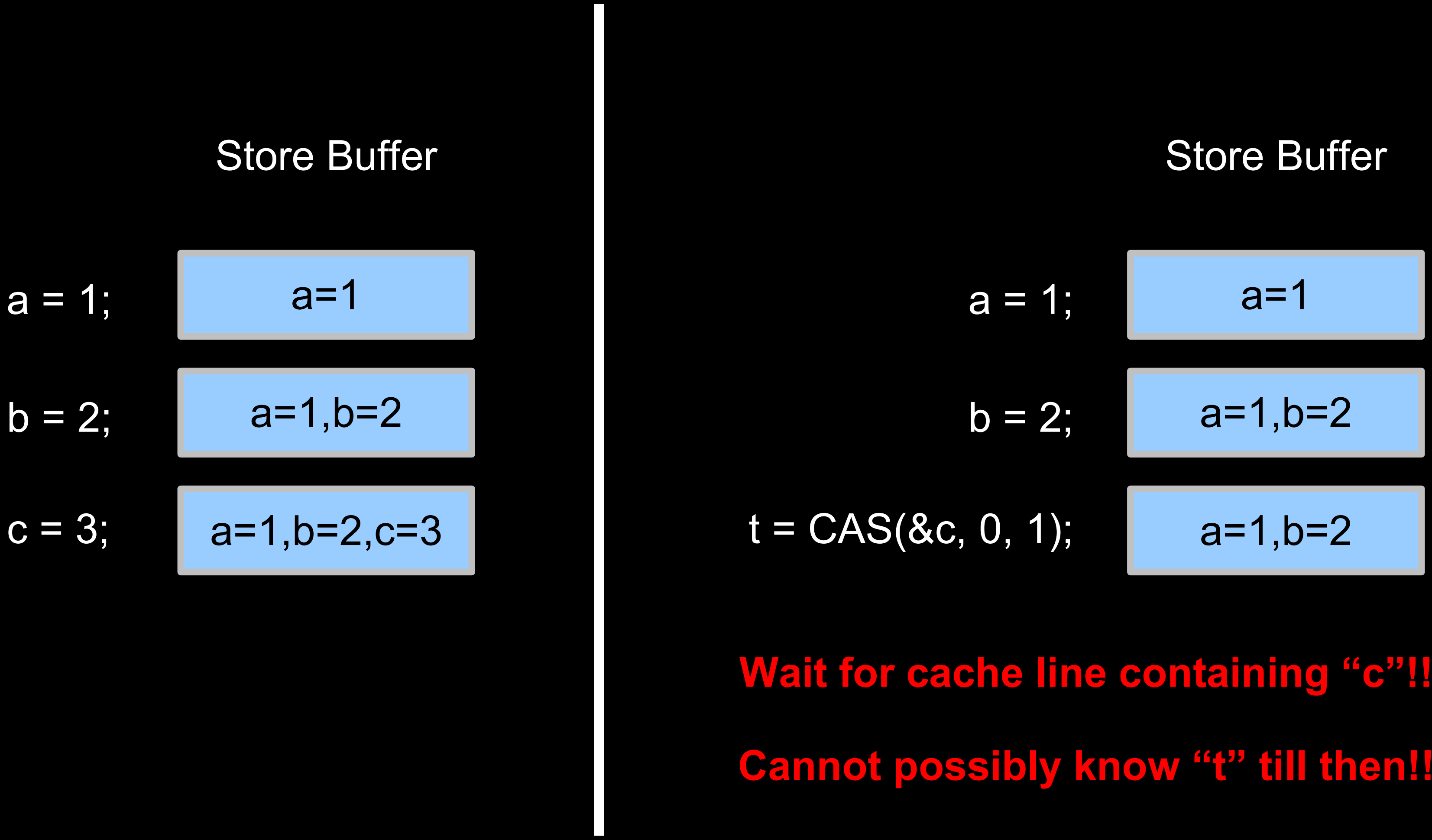


Electrons move at 0.03C to 0.3C in transistors and, so lots of waiting. 3D to the rescue?

CPU Hardware Structure



Why Aren't All Instructions Created Equal?



Store Buffer

a = 1; a=1

b = 2; a=1,b=2

c = 3; a=1,b=2,c=3

Store Buffer

a = 1; a=1

b = 2; a=1,b=2

t = CAS(&c, 0, 1); a=1,b=2

Wait for cache line containing "c"!!!

Cannot possibly know "t" till then!!!

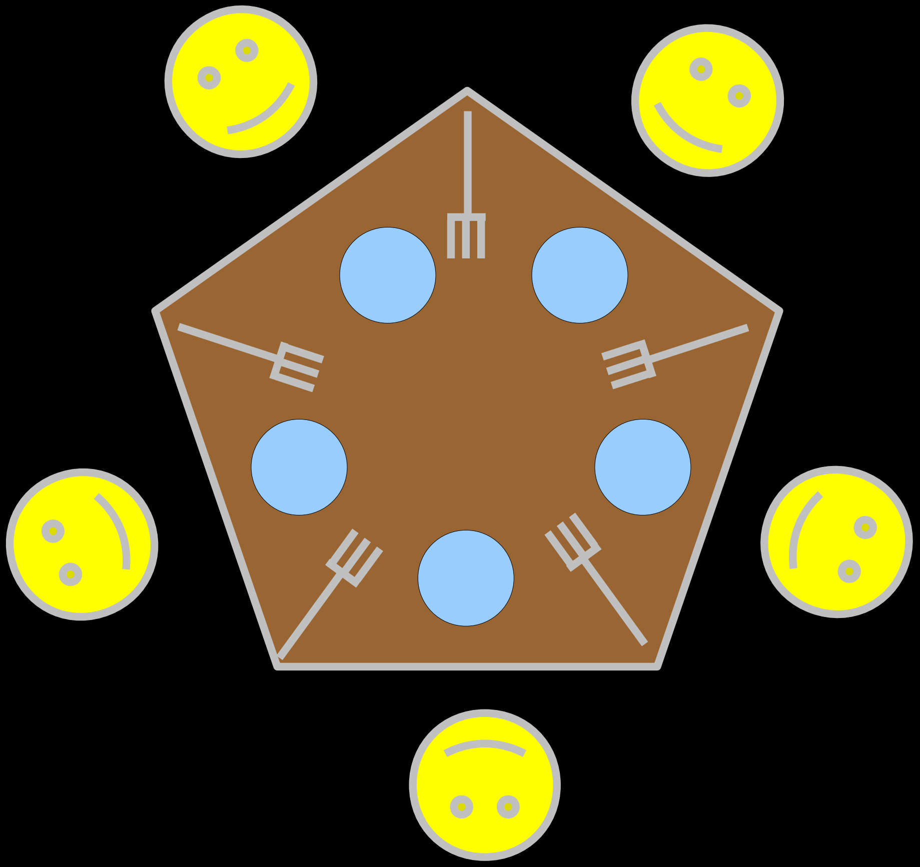
There are many tricks the HW guys play – otherwise the latencies would be *much* worse.

Visual Demonstration of Instruction Overhead

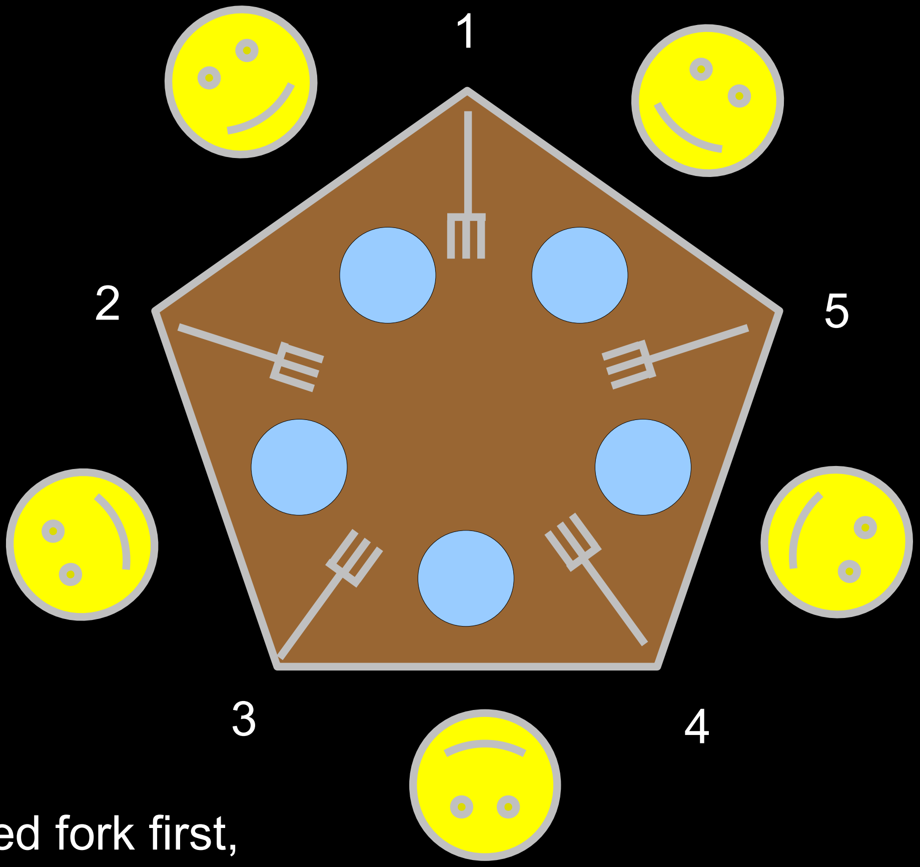
The Bogroll Demonstration

Exercise: Dining Philosophers Problem

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



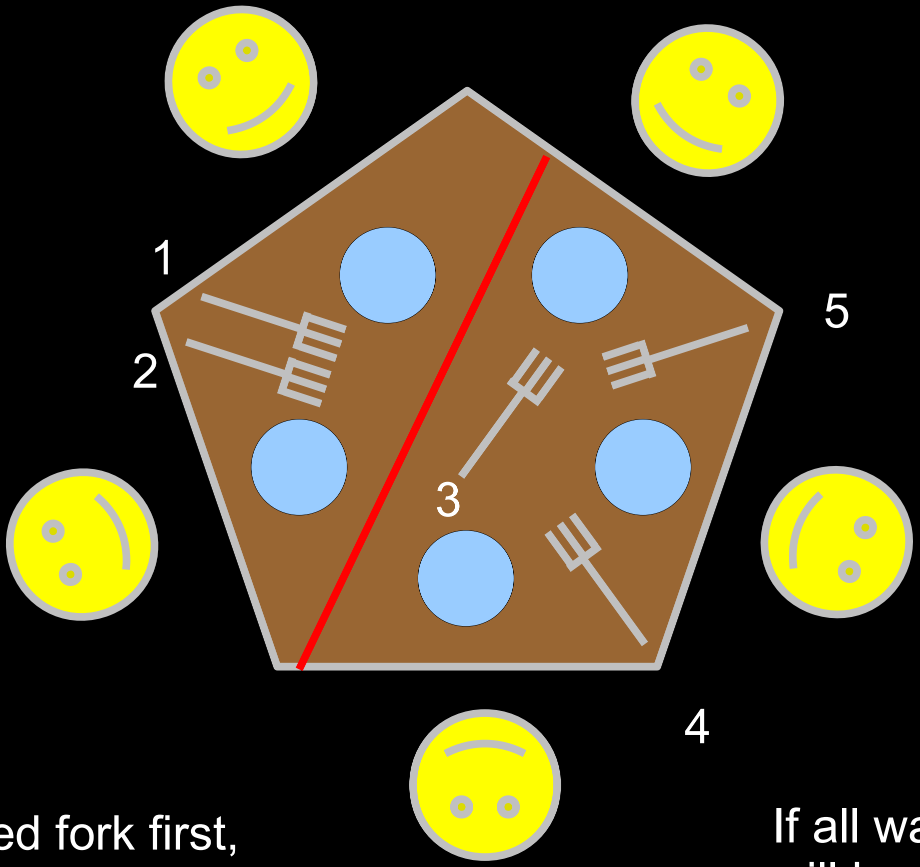
Exercise: Dining Philosophers Solution #1



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

Is this a good solution???

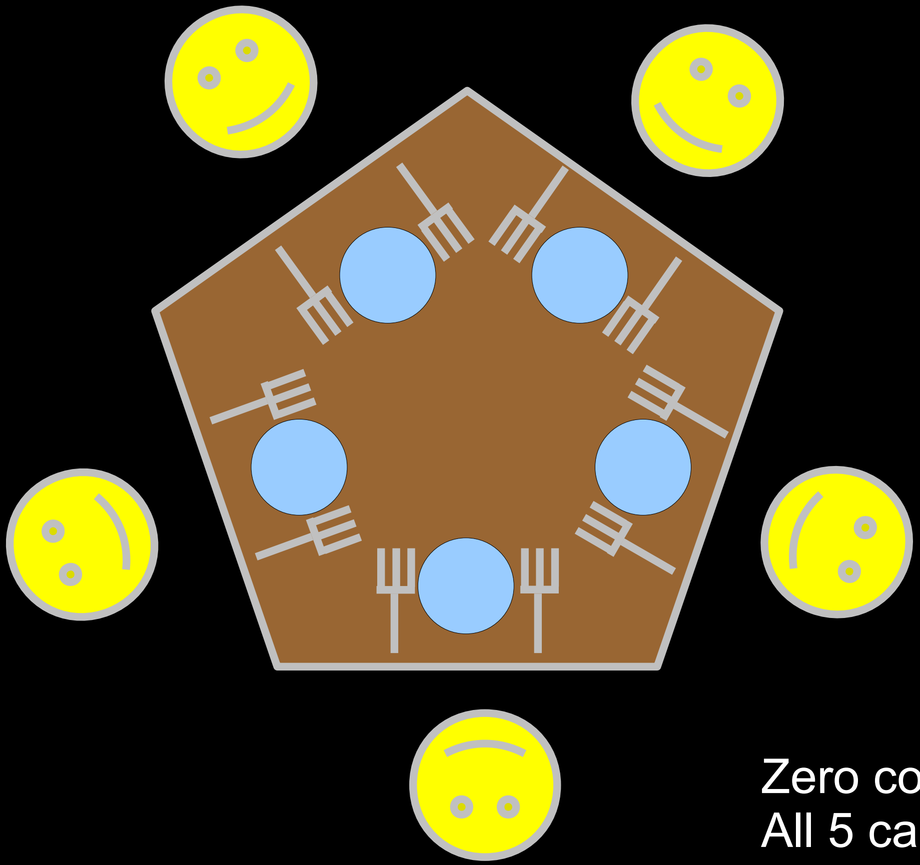
Exercise: Dining Philosophers Solution #2



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...

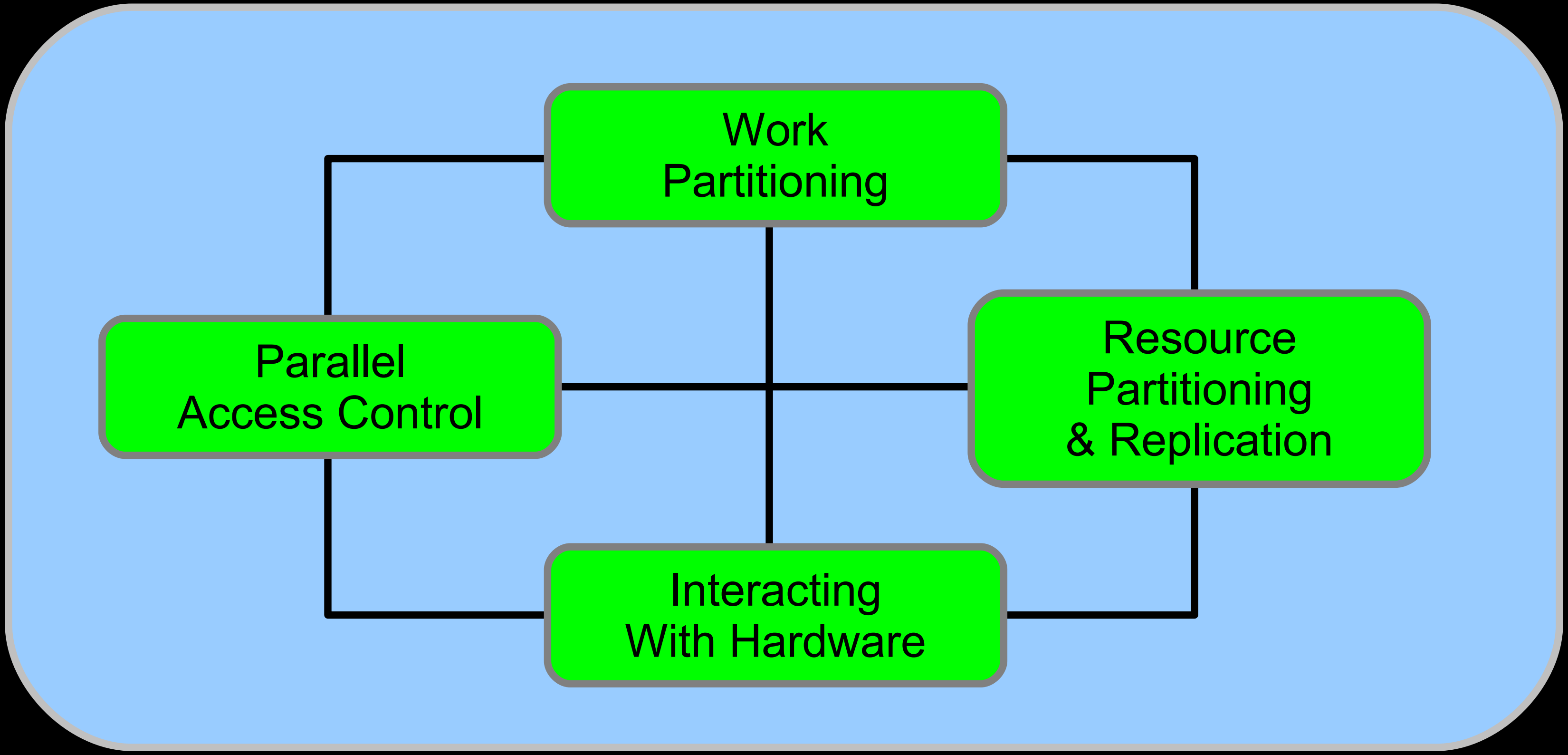
- What do you do for a problem that is inherently fine-grained (so that synchronization primitives such as locking, TM, NBS, &c are inefficient) and update-heavy (so that RCU is not helpful)?

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!!!

Do “Tasks” Relate to Real-World Software?

Parallel Programming Tasks


Work Partitioning
Parallel Access Control
Resource Partitioning & Replication
Interacting With Hardware

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

Do “Tasks” Relate to Real-World Software?

- Three Real-World Production Environments:
  - “Locking plus threads” (L+T)
    - Linux kernel, Pthreads, Windows Threads, ...
    - Often augmented: TM, RCU, ...
  - Message Passing Interface (MPI)
    - Environment of choice for high-end scientific computing
  - Structured Query Language (SQL)
    - Decades-old RDBMS workhorse
- All three have excellent performance
  - Look primarily at productivity

Do “Tasks” Relate to Real-World Software?

	L+T	MPI	SQL
Work Partitioning	m	m	A
Error Processing	A	A	A
Global Processing	m	m	A
Thread Load Balancing	M	M	A
Work Item Load Balancing	m	m	A
Affinity to Resources	m	m	A
Control of Utilization	m	M	A

Do “Tasks” Relate to Real-World Software?

Parallel Access Control	L+T	MPI	SQL
Implicit vs. Explicit	I	e	I
Message Passing	m	m	A
Locking	m		A
Transactions			m
Reference Counting	m		A
Shared Variables	m		A
Ownership	m	A	A

Do “Tasks” Relate to Real-World Software?

Resource Partitioning	L+T	MPI	SQL
Over Systems		m	H
Over NUMA Nodes	m	m	A
Over CPUs/Dies/Cores	A	A	A
Over Critical Sections	m		A
Over Synchronization Primitives	m		H
Over Storage Devices	m	m	h
Over Pages and Cache Lines	m	m	A

Conclusions

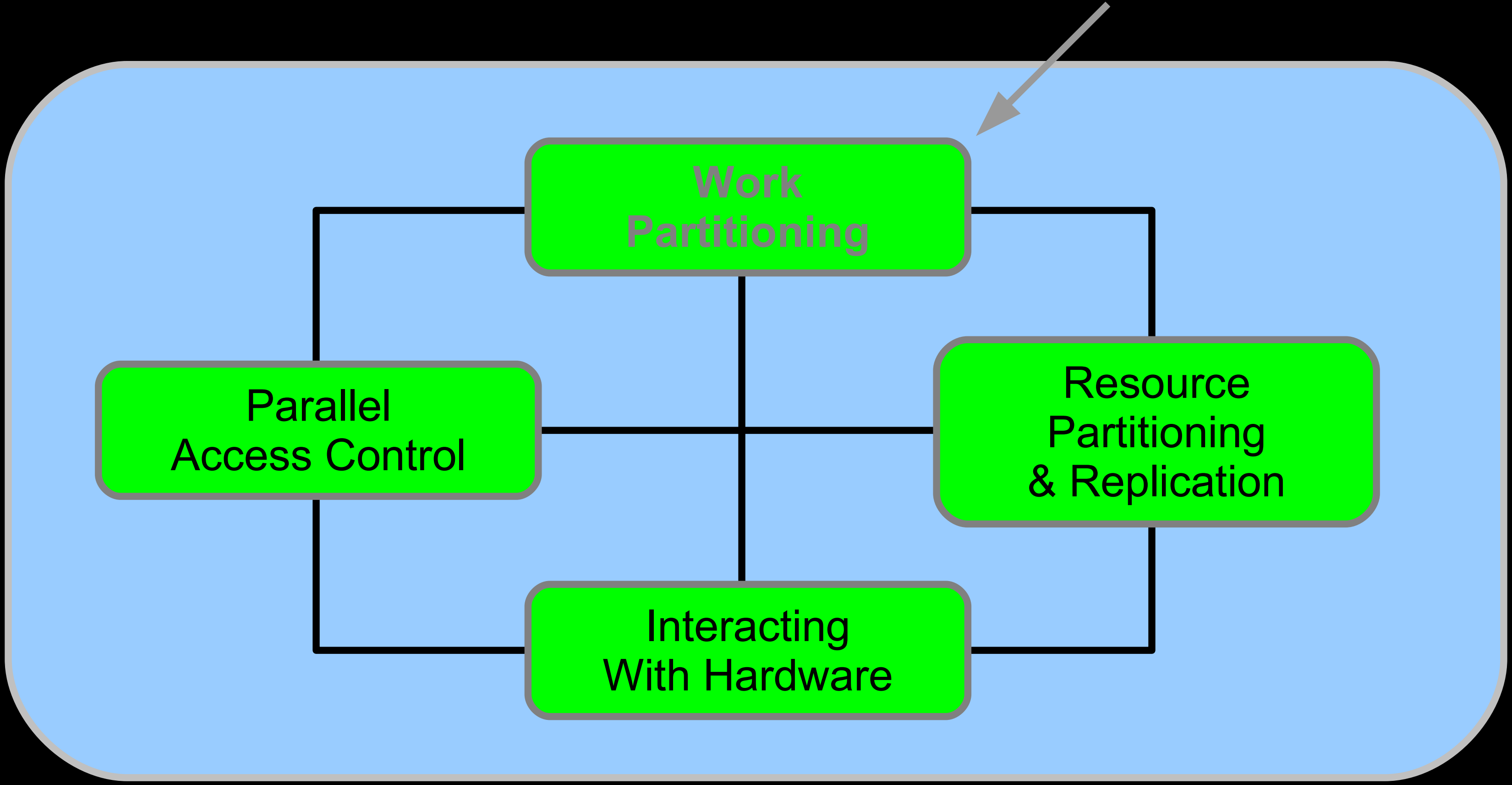
Summary and Problem Statement

- SQL Offers Impressive Example of Pervasive Parallel Automation With High Performance
  - Unfortunately, quite specialized
- L+T and MPI are General With High Performance
  - Too bad about that low productivity!!!
- So: use SQL Where it Makes Sense, Else L+T or MPI
  - MPI scales higher than does L+T, but harder to convert
- 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 Statement #1: Parallel Pitfall

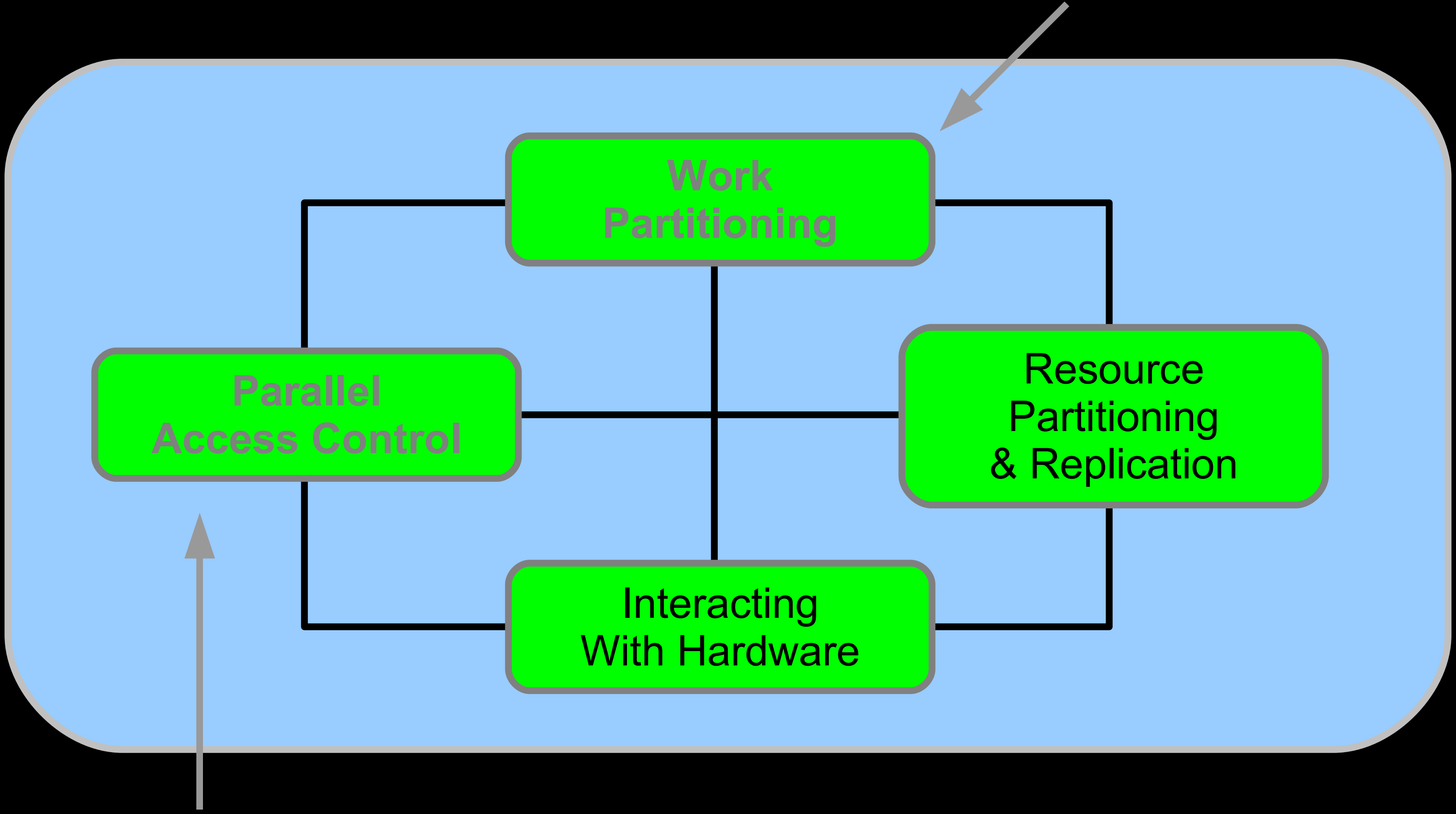
Problem Statement #1

Start with preconceived algorithmic work breakdown


Work Partitioning
Parallel Access Control
Resource Partitioning & Replication
Interacting With Hardware

Problem Statement #1

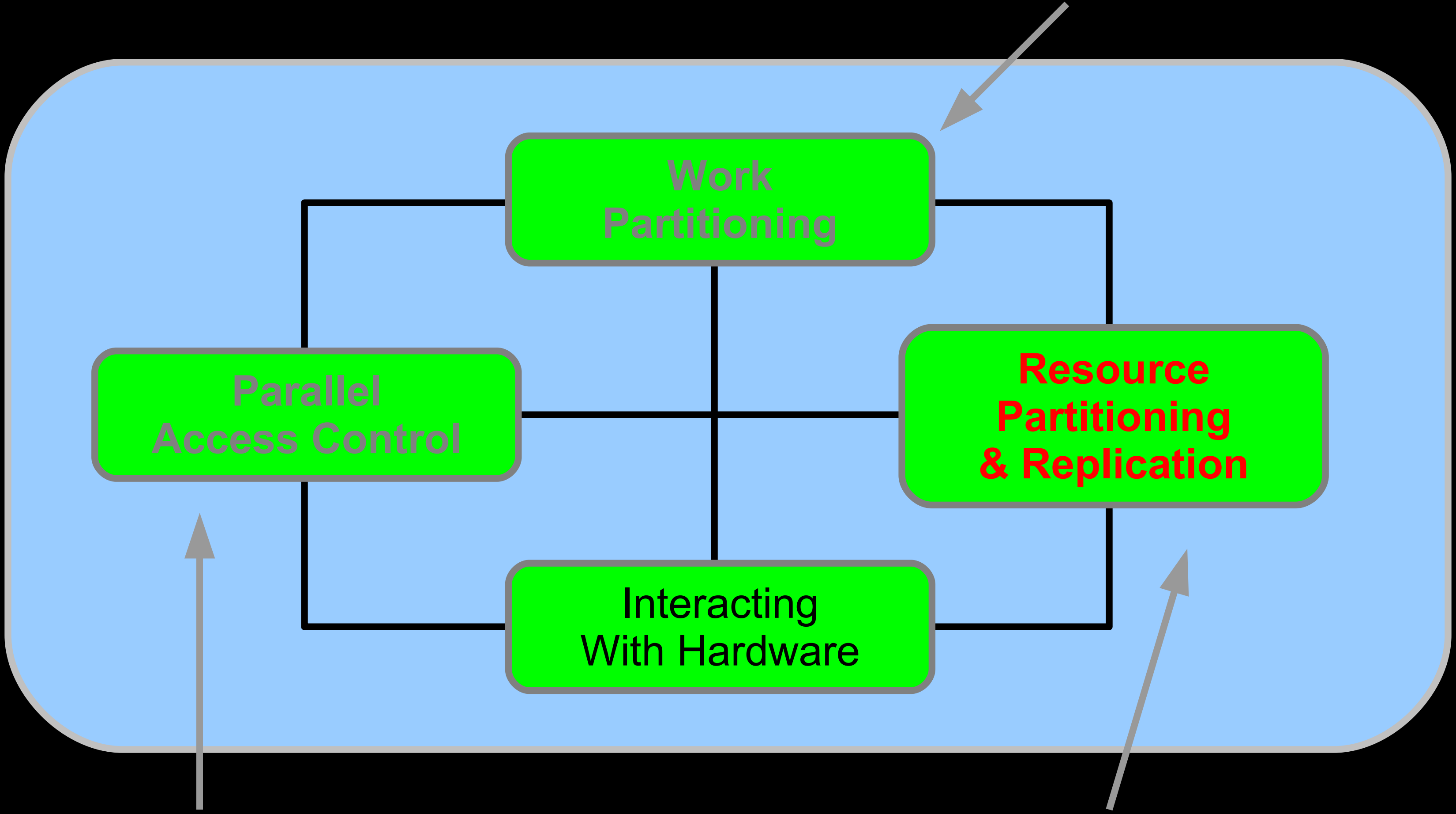
Start with preconceived algorithmic work breakdown


Work Partitioning
Parallel Access Control
Resource Partitioning & Replication
Interacting With Hardware

Choose synchronization mechanism

Problem Statement #1

Start with preconceived algorithmic work breakdown


Work Partitioning
Parallel Access Control
Resource Partitioning & Replication
Interacting With Hardware

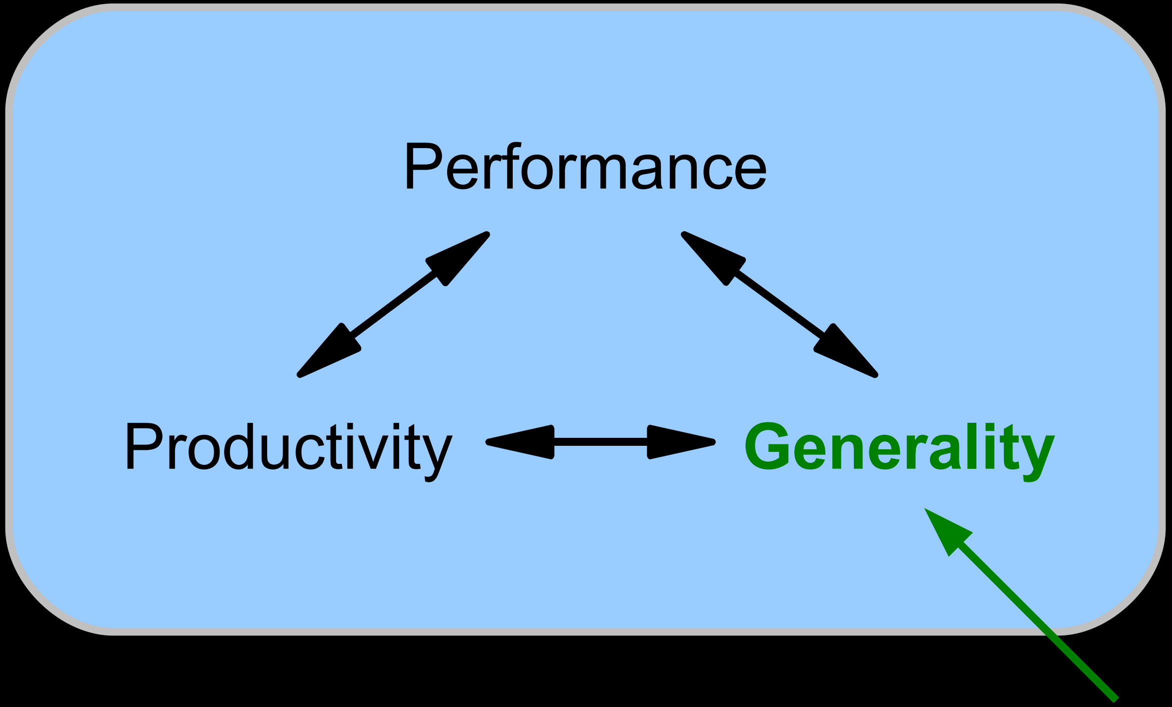
Choose synchronization mechanism

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

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

Narf!!!

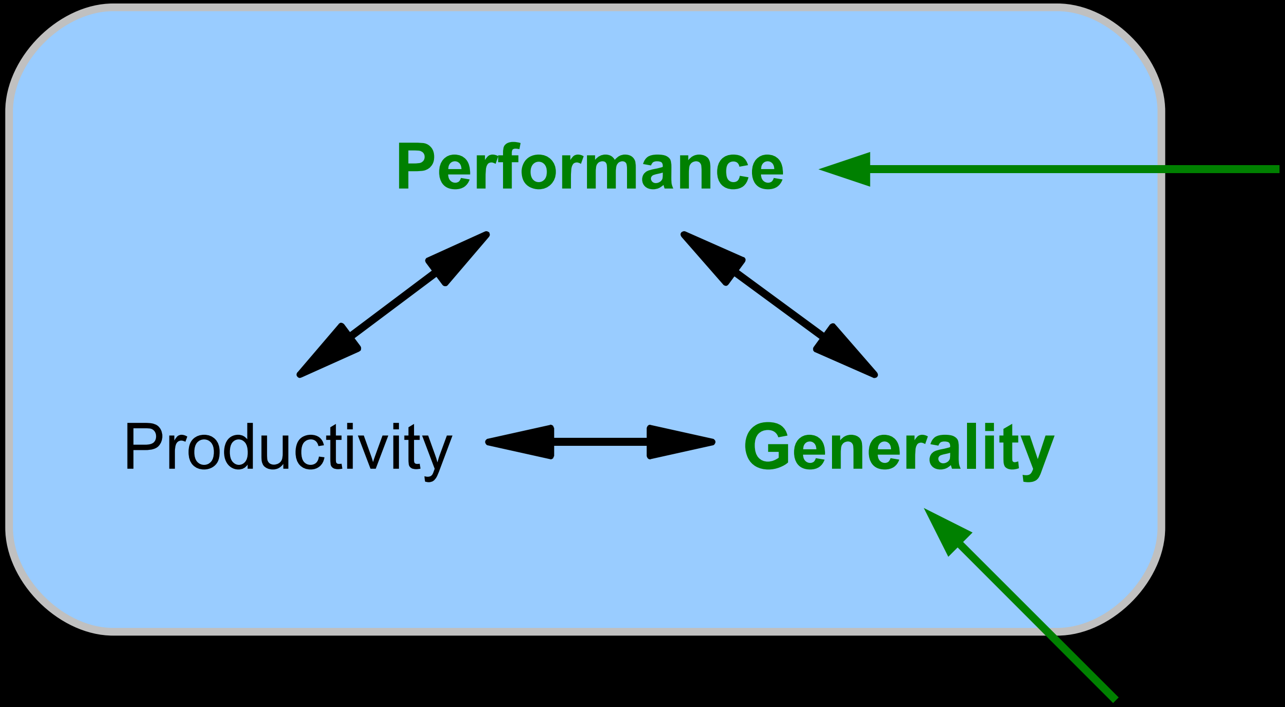
Problem Statement #2



Job #1 to “Take Over the World”

But now a choice: Performance? or Productivity?

Problem Statement #2

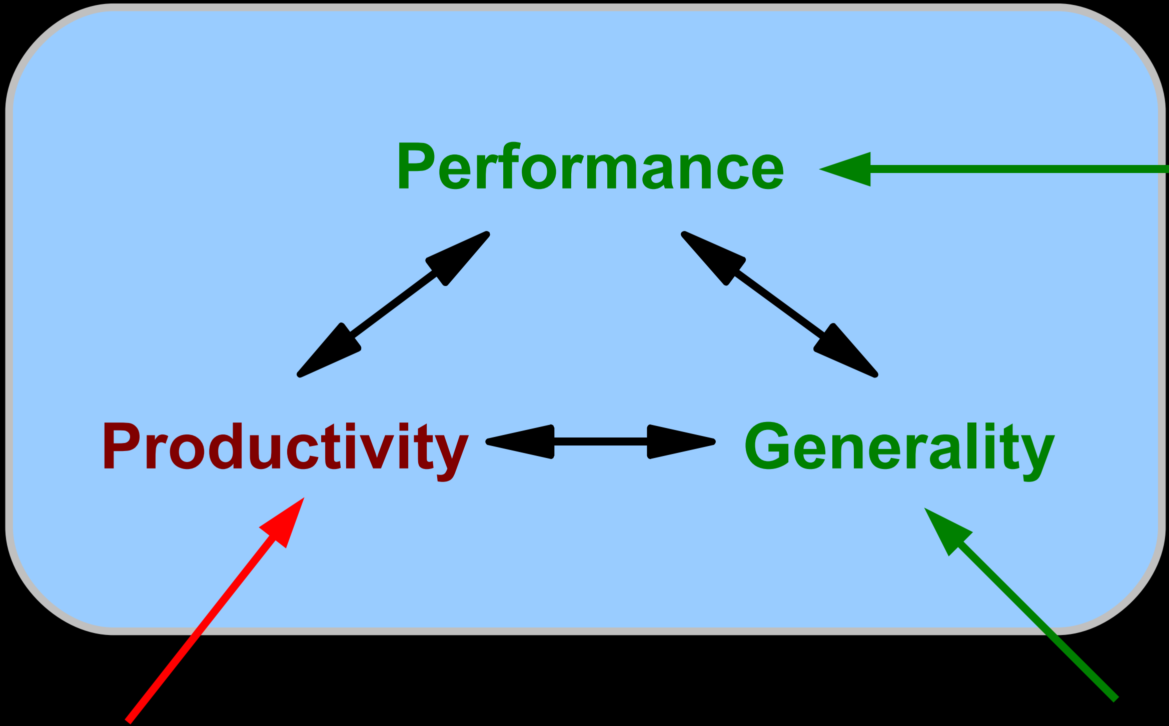


Also need to “Be the Best”

Job #1 to “Take Over the World”

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

Problem Statement #2

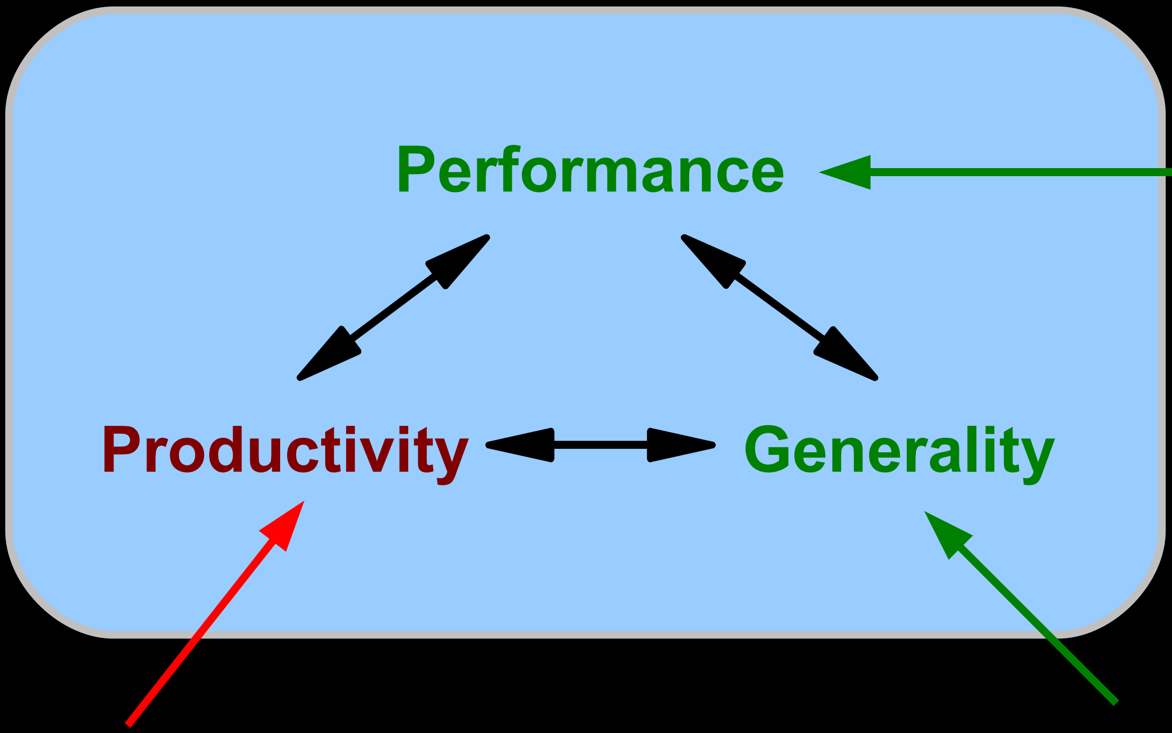


Also need to “Be the Best”

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.

Questions?