Paul E. McKenney – IBM Distinguished Engineer, Linux Technology Center

29 Jan 2011 linux.conf.au Multicore and Parallel Computing MiniConf

Is Parallel Programming Hard?

And, If So, Why?

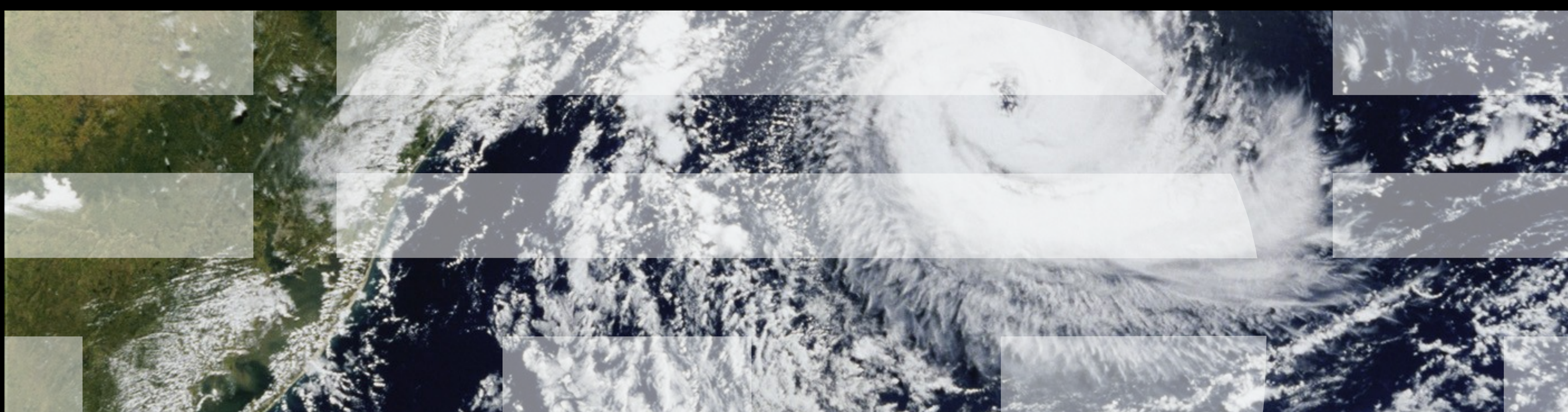


Table of contents

Early Experiences With Parallelism

Early Experiences With Parallelism

- In the mid-1970s, the doorbell rang
  - And like a fool, I answered it...

Nor Were the Quints' Parents Unprecedented

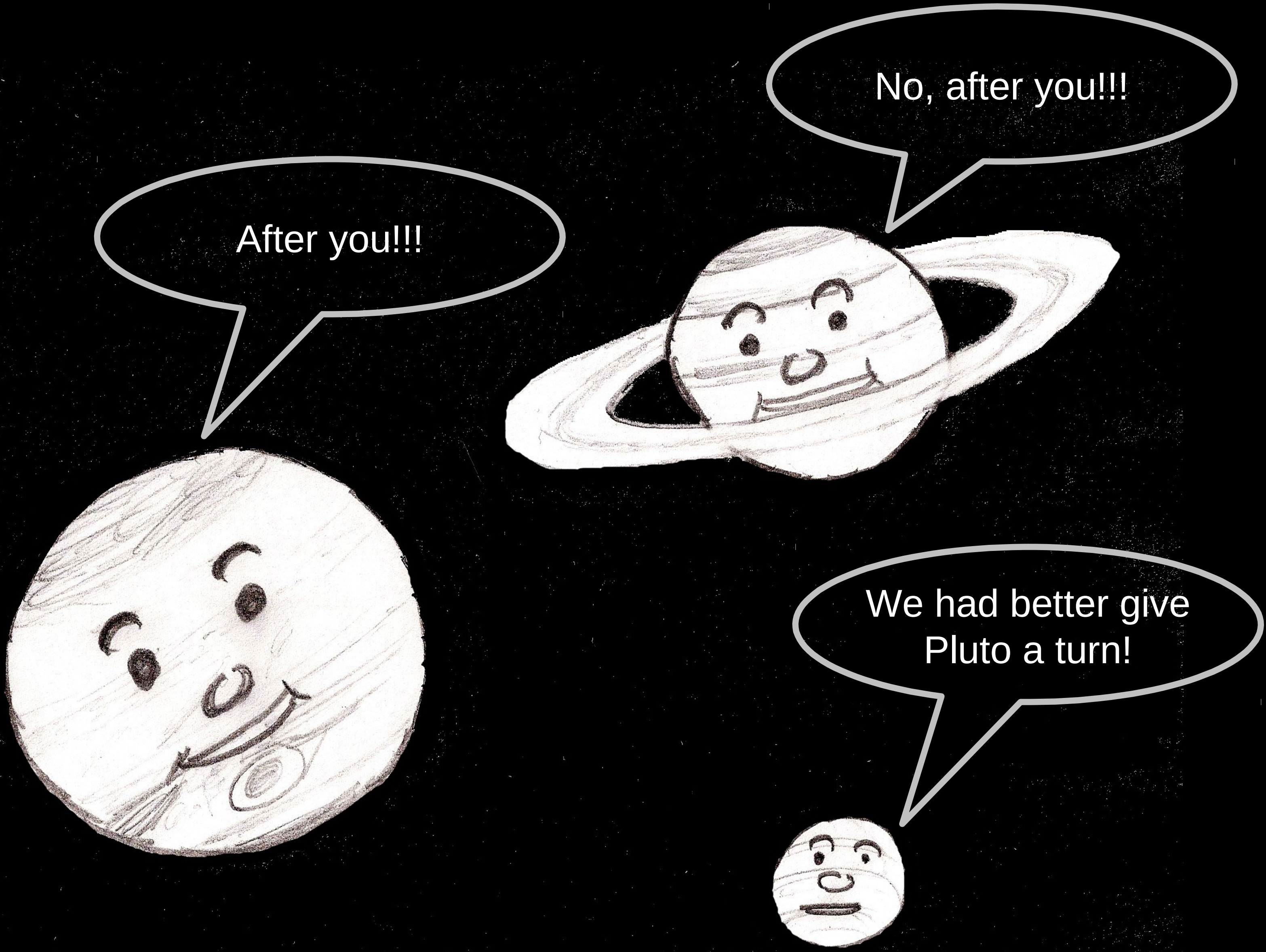
Nor Were the Quints' Parents Unprecedented

Concurrency comes naturally to human beings

Other Examples of Human Tolerance of Concurrency

- Team sports:
  - Basketball: Nine other players plus referees
  - American football: 21 other players plus referees
  - Football/soccer: 21 other players plus referees
  - Ice Hockey: 11 other players plus referees
  - Massively Multiplayer Online Gaming: lots of other players
- Teaching: tens of students
- Construction: tens of workers
- Driving in congested conditions: many other drivers
  - Hopefully also paying attention to pedestrians and bicyclists!
- Air-traffic control: many aircraft
- Emergency services: large numbers of people

Let's Face It: Concurrency Comes Naturally To The Entire Universe



But Just Because Concurrency Comes Naturally Does Not Necessarily Mean That Concurrent ***Programming*** Comes Naturally

More on this later...

Early Experiences With Parallel Computing

- 1989: distributed simulation on network of workstations
- 1990-1999: DYNIX/ptx parallel UNIX kernel
- 2000: AIX parallel UNIX kernel
- 2001-present: Linux UNIX kernel

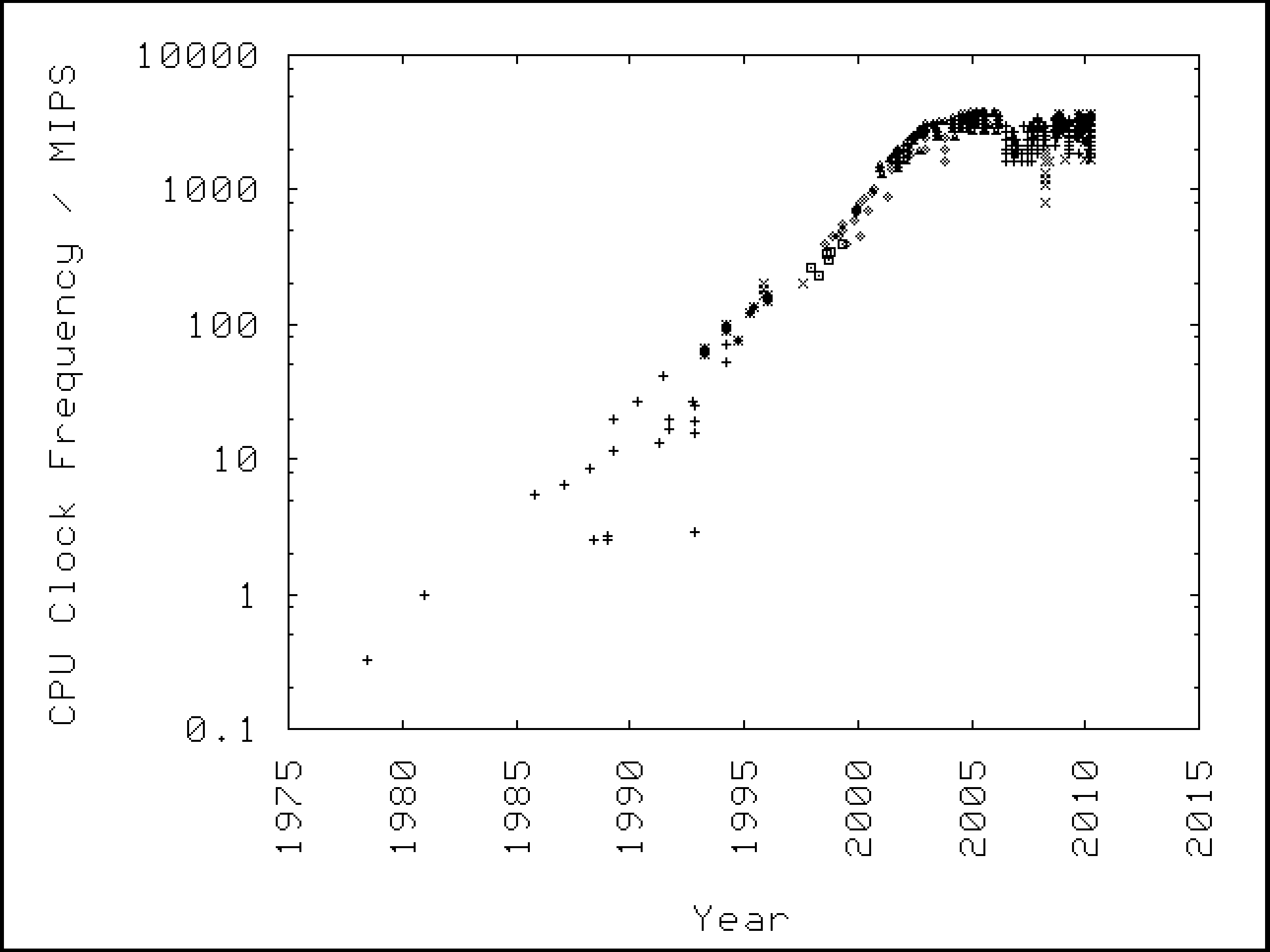
Early Experiences With Parallel Computing

- 1989: distributed simulation on network of workstations
- 1990-1999: DYNIX/ptx parallel UNIX kernel
- 2000: AIX parallel UNIX kernel
- 2001-present: Linux UNIX kernel

- So why all the excitement after all these decades?

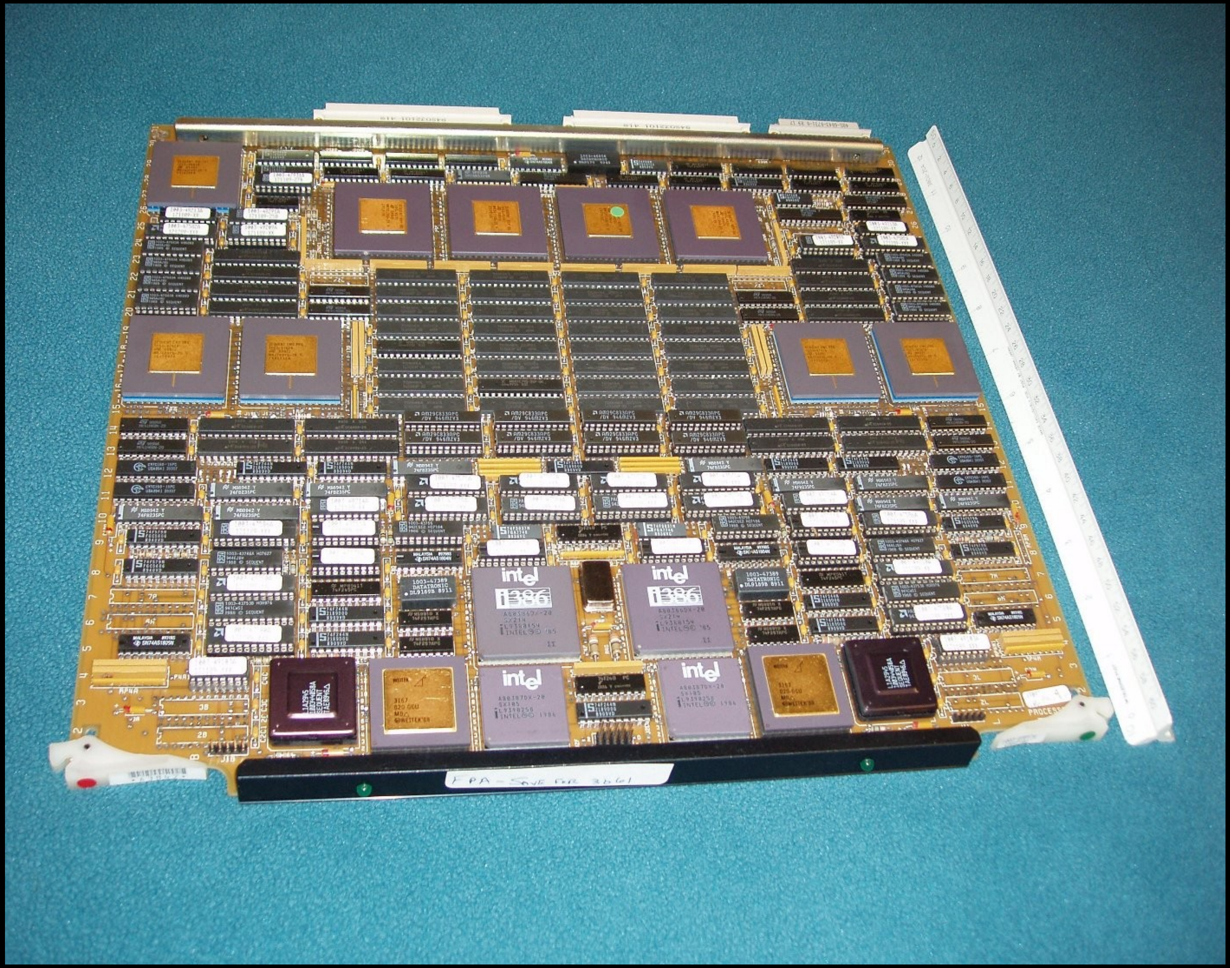
Why Excitement About Parallelism After All These Decades?

Why Excitement About Parallelism After All These Decades?



The Party Line

Simple Economics!



Why Excitement About Parallelism After All These Decades? Simple Economics!

- 1990: “Low-cost” multiprocessors system >> $100K
- 2006: Grad student buys dual-core Mac on whim
- 2011: Multiprocessor systems << $1K

IBM

Why Excitement About Parallelism After All These Decades? Simple Economics!

- 1990: “Low-cost” multiprocessors system >> $100K
- 2006: Grad student buys dual-core Mac on whim
- 2011: Multiprocessor systems << $1K

- Suddenly, multiprocessor systems can be used ***everywhere***
- Suddenly there is an acute shortage of parallel programmers

Why Excitement About Parallelism After All These Decades? Simple Economics!

- 1990: “Low-cost” multiprocessors system >> $100K
- 2006: Grad student buys dual-core Mac on whim
- 2011: Multiprocessor systems << $1K

- Suddenly, multiprocessor systems can be used ***everywhere***
- Suddenly there is an acute shortage of parallel programmers
  – But we have been here before...

The Great Software Crisis

History And Causes

- 1960s: “Low-cost” computer system >> $100K
- 1970: Minicomputers for $25K
- Late 1970s: Microcomputers << $1K

- Suddenly, computer systems can be used ***everywhere***
- Suddenly there is an acute shortage of programmers
  - But somehow the problem was solved. How?

The Solution To The Great Software Crisis

- Low-cost PCs meant that lots of people could afford them
- Lots of people bought PCs and other computers
  - Both for themselves and for their children
- As a result, lots of people gained experience with computers and with software
- These people produced the software that allows anyone to make good use of computers
  - Even my grandparents used computers

- The advent of low-cost multicore systems will solve the Great Multicore Software Crisis

The Solution To The Great Software Crisis

- Low-cost PCs meant that lots of people could afford them
- Lots of people bought PCs and other computers
  - Both for themselves and for their children
- As a result, lots of people gained experience with computers and with software
- These people produced the software that allows anyone to make good use of computers
  - Even my grandparents used computers

- The advent of low-cost multicore systems will solve the Great Multicore Software Crisis... Eventually...

Three Classes Of Attempted Solutions To The Great Software Crisis

- The Good
  - Orders of magnitude improvement in productivity
  - Orders of magnitude increase in people able to use computers
  - Preferably both simultaneously
- The Fad
  - Lots of excitement at the time, but long forgotten
- The Ugly
  - Was in use then, still in use now
  - To ugly to die

- Your nominations for these categories?
  - If you remember the 1980s...

Three Classes Of Attempted Solutions To The Great Software Crisis

- The Good
  - Spreadsheet
  - Presentation manager and word processor
  - Computer-aided engineering
- The Fad
  - An amazingly large number of long-forgotten languages
- The Ugly
  - The C language
  - sed, awk, perl, Visual BASIC, …

- There will be the same three classes of attempted solutions to the Great Multicore Crisis

What Is Hard About Programming?

Erroneous Expectations

- People expect anything that seems intelligent to have some degree of common sense
- People expect intelligent beings to understand their intent
- People expect to be successful despite fragmentary and incomplete plans

Erroneous Expectations

- People expect anything that seems intelligent to have some degree of common sense
- People expect intelligent beings to understand their intent
- People expect to be successful despite fragmentary and incomplete plans

- Most of this has little to do with parallelism

Managing Erroneous Expectations

- People expect anything that seems intelligent to have some degree of common sense
  - Computers are usually marketed as tools rather than beings
  - Eliza, Aibo, and Watson notwithstanding
- People expect intelligent beings to understand their intent
  - Computers are used in situations where intent is known implicitly
  - GPS units, web browsers, autopilots, ...
- People expect to be successful despite fragmentary and incomplete plans
  - And this one is the most relevant to parallel programming
  - Deadlocks, livelocks, and data races are ***planning failures***
  - Solution: Let the computer do the planning!!! Tools – Lots of tools!!!

Pathways To Multicore Software Success

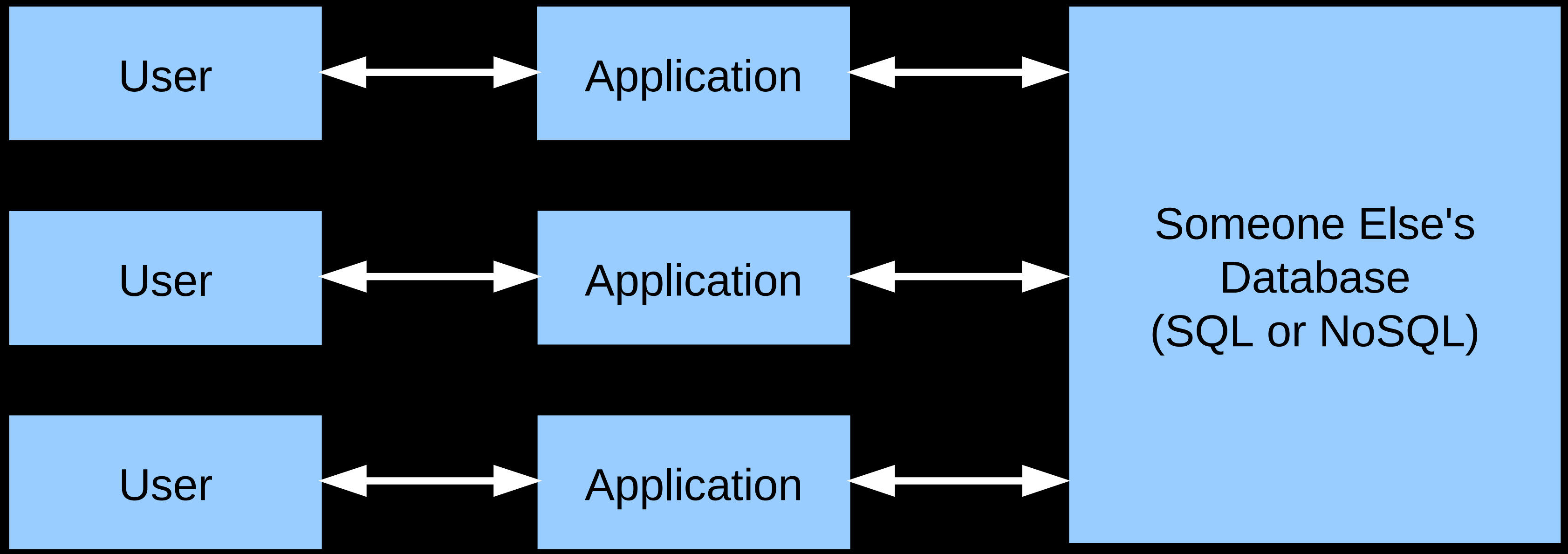
What Has Worked In The Past?

- Apprenticeship approach
  - Pair newbie with experienced parallel programmers
    - Sequent Computer Systems
    - Linux kernel community
  - Very effective: very ordinary engineers will produce competent parallel code within a few months
- Learn from existing parallel open-source projects
  - Linux kernel, PostgreSQL, Samba, …
  - Find the one that most closely matches your needs
- Exploit embarrassing parallelism
  - Transform your problem into an embarrassing one if need be
- Take validation seriously, from the ground up

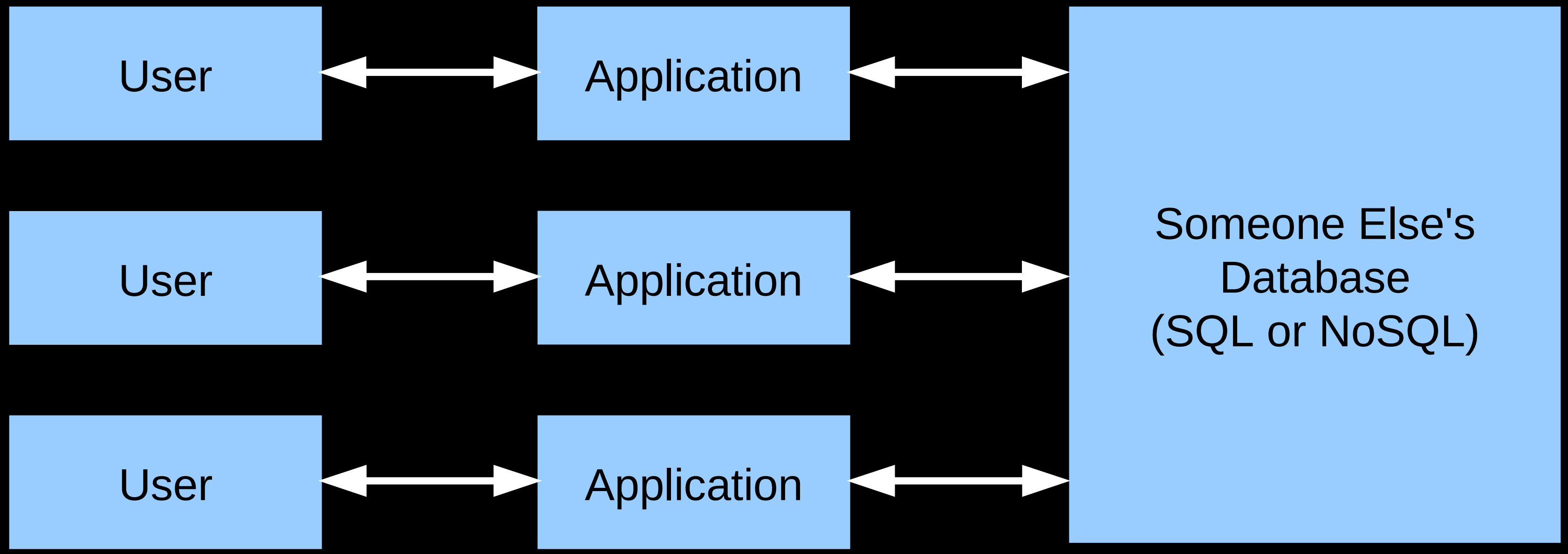
What Has Worked In The Past?

- Work out what you need up front
  - If single-threaded software is fast enough, ignore parallelism!
  - Avoid the N+1 trap
    - “Do a single-threaded implementation.” “Good, now do a 2-CPU implementation.” “Great, now make it handle 3 CPUs.” …
    - Easier to do it once for (say) 32 CPUs than to rewrite it 32 times!!!
- Make sure you have a solid core of experienced engineers
  - Trust me, you don't want the blind mentoring the blind!!!
- Make sure all engineers have access to parallel hardware
  - And that they understand its properties and capabilities
- Make sure all engineers have access to all source code
- Make sure your team can deliver decent software...

Pattern for Success: Let Someone Else Be Parallel

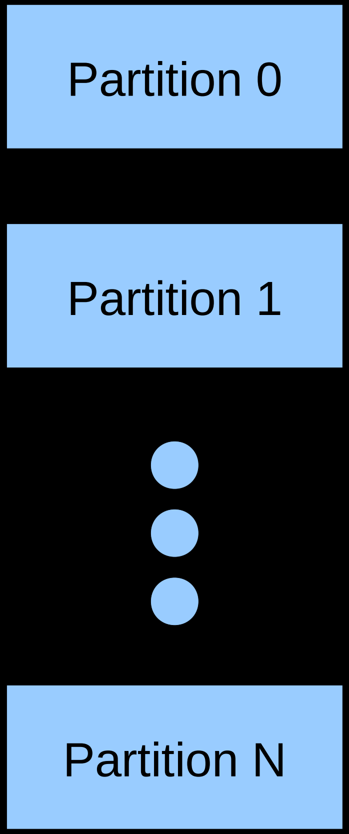


Pattern for Success: Let Someone Else Be Parallel

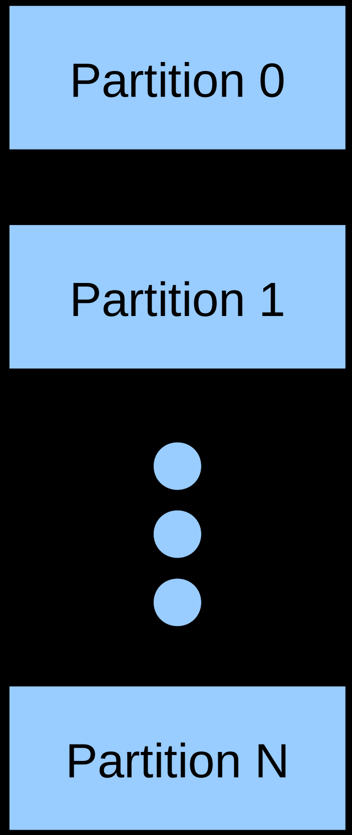


You can use an in-memory database, for example, Samba's TDB

Pattern for Success: Treasure Trivial Solutions



Pattern for Success: Treasure Trivial Solutions



Which is one step away from map-reduce

Pattern for Success: Stick With Single-Threaded Code!!!

If single-threaded is fast enough, why bother with parallelism???

Don't Forget Simple Techniques

- Partitioning is simple, but can be extremely effective
- Batching is simple, but amortizes synchronization overhead
- Sequential execution is simple, and should be used when the resulting performance is sufficient
- Pipelining is simple, but can greatly reduce synchronization overhead
- Never be afraid to exploit important special cases:
  - Read-only and read-most situations, partitionable common-case execution, privatizable data, …
- Finding bottlenecks should be simple, but often isn't

Traps and Pitfalls

Traps and Pitfalls

Not that sequential code has any traps and pitfalls...

Parallelizing An Existing Single-Threaded Project

- Single-Threaded Design, Code, and APIs
- Sequential Staff
- Darwin Strikes Again!!!
- Weak or Non-Existent Validation
- Failing to Understand Underlying Software and Hardware

Single-Threaded Design, Code, and APIs

- Things that are cheap and easy for single-threaded code:
  - Singleton pattern/objects through which all control passes
    - Global counters
    - Hoare monitors
    - Global transaction IDs
  - Ordering guarantees
  - Stop-the-world processing
  - Global locks
  - Strongly non-commutative APIs that guarantee determinism and linearizability (Attiya et al. "Laws of Order")
- These are all quite expensive for parallel code: Often fatal!!!
- Parallelizing your single-threaded software may require big-animal changes.

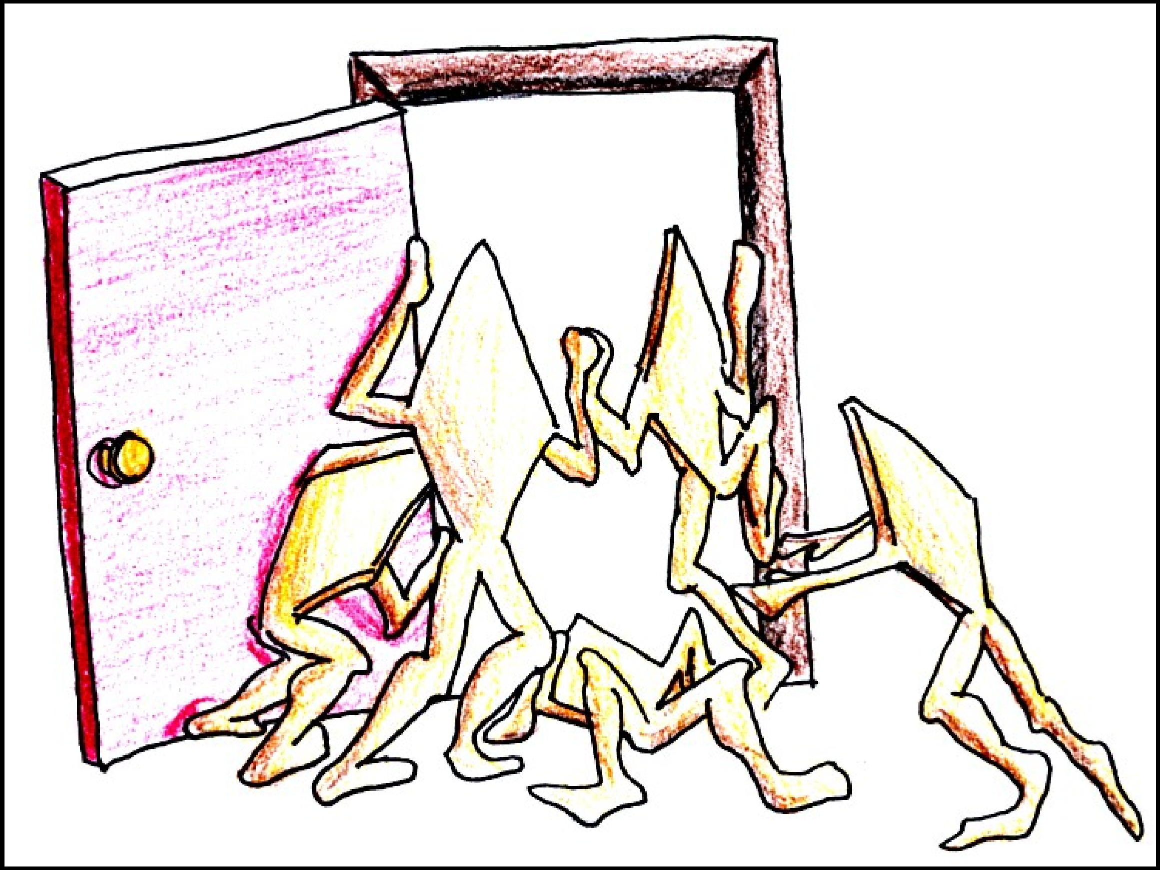
Sequential Staff

- If you have an existing sequential software project, you probably already have people working on it
  - Who probably know nothing about parallel software
- You might be able to attract (or hire) an experienced parallel programmer (preferably in the required type of parallelism)
- You will then need:
  - Readily available parallel/multicore hardware
  - Large body of high-quality parallel code for review and tinkering
  - Easy access to parallel-programming experts
    - Preferably with a wide variety of viewpoints
  - Immediate “frank and open” expert feedback for multicore newbies
- What if you can't get an experienced parallel programmer?

Sequential Staff With No Experts?

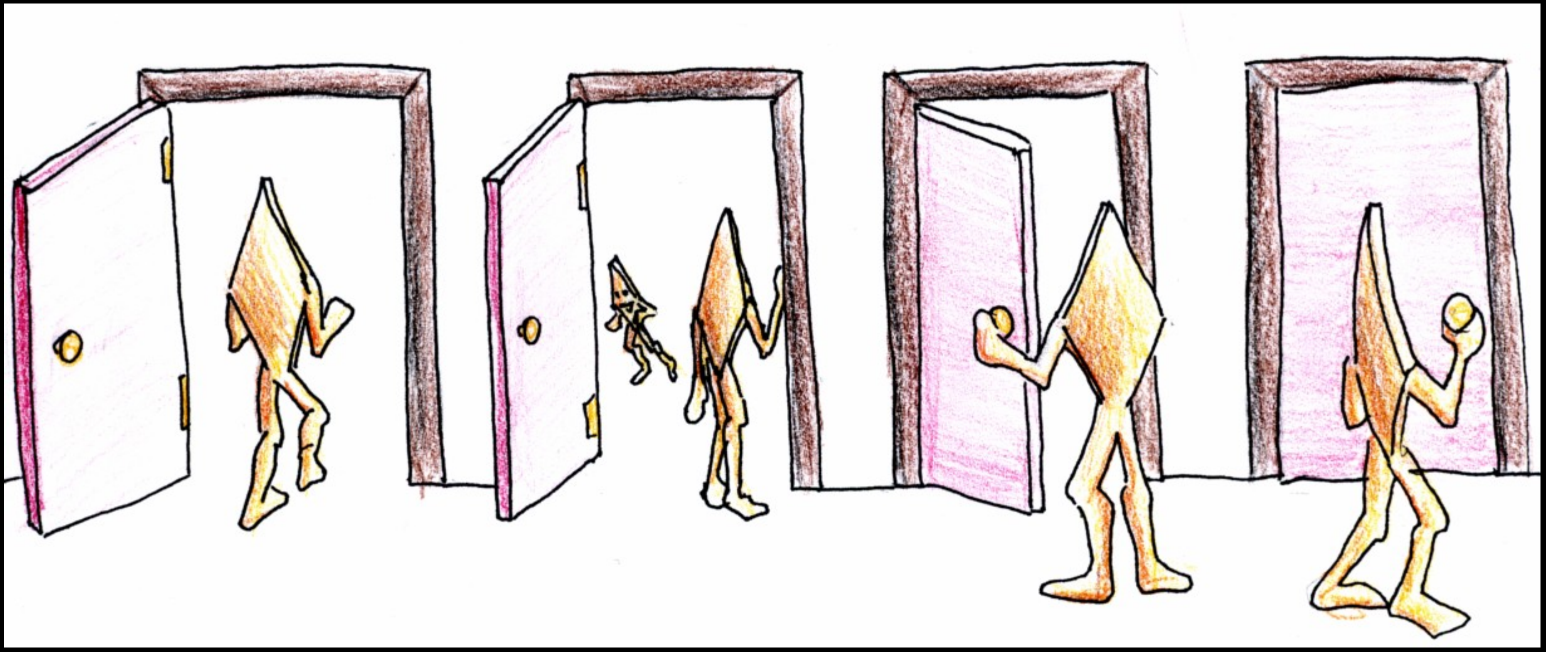
- Decent classes are becoming available
  - But I cannot judge: I learned this stuff directly from the hardware
  - But I do know the single most important lesson!!!
- Any number of parallel open-source projects are out there
  - One quick (if brutal) way to get employee training!
  - If your project has an incompatible license, use carefully crafted procedures to avoid contamination
  - But it might be easier to select a project with a compatible license

Sequential Staff: The Most Important Parallel/Multicore Lesson:



Avoid having only one of something on the fastpath!!!

Sequential Staff: The Most Important Parallel/Multicore Lesson:



Avoid having only one of something on the fastpath!!!

Sequential Staff With No Experts?

- Decent classes are becoming available
  - But I cannot judge: I learned this stuff directly from the hardware
  - But I do know the single most important lesson!!!
- Any number of parallel open-source projects are out there
  - One quick (if brutal) way to get employee training!
  - If your project has an incompatible license, use carefully crafted procedures to avoid contamination
  - But it might be easier to select a project with a compatible license

- But how about just hiring a bunch of really smart people?

Sequential Staff With Really Smart Newbies?

- This can work if the newbies have access to hardware, software, experts, and feedback
- Otherwise, added intelligence is a negative
  - The smarter you are, the deeper a hole you will dig for yourself before you realize that you are in trouble
  - There are a ***very*** small number of exceptions to this rule, and I am ***not*** one of them
- Fortunately, the increasing quantities of readily available multicore hardware and parallel software is increasing the odds that newbies will have parallel/multicore experience
- Just as with the Great Software Crisis!

Sequential Staff: One More Question...

- There is a good chance that your single-threaded software base will need big-animal changes
- So can your existing developers make big-animal changes in your current code base?

Sequential Staff: One More Question...

- There is a good chance that your single-threaded software base will need big-animal changes
- So can your existing developers make big-animal changes in your current code base?
- Or have your original developers long since departed?

Sequential Staff: Software Janitors?

Your multicore strategy must take into account available skills!

Darwin Strikes Again!!!

- Through the 80s, 90s, and the first half of the 00s, parallel systems and programmers were rare and expensive
  - Extreme cost, itty bitty unit volumes
- With a very few exceptions, projects and products whose developers disliked parallelism were at a large advantage
  - Most of the market at reasonably low cost
- If your product or project is decades old, some attitude adjustments may be required...

Weak or Non-Existent Validation

- Parallelism adds failure modes, requiring tougher validation
- So get your validation in good shape before parallelizing!!!

Failing to Understand Underlying Software and Hardware

- Would you trust:
  - A bridge designed by someone who didn't understand that concrete, while strong in compression, is weak in tension?
  - A home heating system designed by someone who didn't understand that would houses burn?
  - A home in the rainy Pacific Northwest designed by someone who didn't understand that wood rots in temperate rain forests?
  - A space shuttle designed by someone who didn't understand the low-temperature properties of O-rings?

Failing to Understand Underlying Software and Hardware

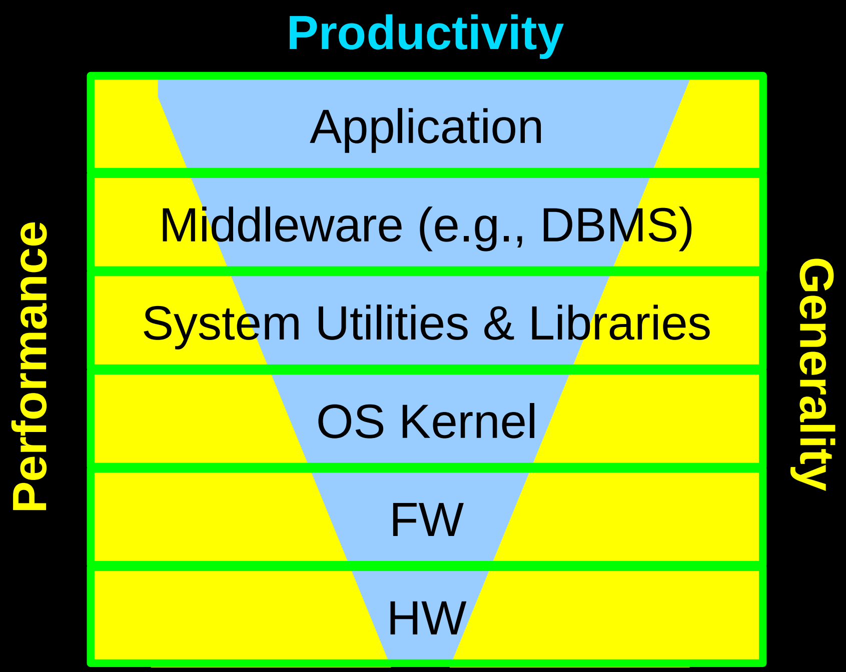
- Would you trust:
  - A bridge designed by someone who didn't understand that concrete, while strong in compression, is weak in tension?
  - A home heating system designed by someone who didn't understand that would houses burn?
  - A home in the rainy Pacific Northwest designed by someone who didn't understand that wood rots in temperate rain forests?
  - A space shuttle designed by someone who didn't understand the low-temperature properties of O-rings?
- If not, why would you trust an algorithm designed by someone who didn't understand hardware properties?

How Far Should You Take Parallelism?

How Far Should You Take Parallelism?

It Depends!

It Depends On Your Position In The Software Stack



There is great variety at the application level

How Does the Linux Kernel Community Cope?

Kernel-Community Approaches to Concurrency (Subset 1/2)

- Organizational mechanisms
  - Maintainers and quality assurance: recognition and responsibility
  - Informal apprenticeship/mentoring model
  - Design/code review required for acceptance
  - Aggressive pursuit of modularity and simplicity
- Use sane idioms and abstractions
  - Locking, sequence locking, sleep/wakeup, memory fences, RCU, ...
  - Conventional use of memory-ordering primitives, for example:
  - Needing to know too much about the underlying memory model indicates broken abstraction, broken design, or both

Kernel-Community Approaches to Concurrency (Subset 2/2)

- Static source-code analysis
  - “checkpatch.pl” to enforce coding standards
  - “sparse” static analyzer to check lock acquire/release mismatches
  - “coccinelle” to automate inspection and generation of bug fixes
- Dynamic analysis
  - “lockdep” deadlock detector (also checks for misuse of RCU)
  - Tracing and performance analysis
  - Assertions
- Aggressive automation
  - “git” source-code control system: from weeks to minutes for rebases and merges
- Testing
  - In-kernel test facilities such as rcutorture
  - Out-of-kernel test suites

Kernel-Community Approaches to Concurrency

- To err is human, and therefore...
  - People/organizational mechanisms are at least as important as concurrency technology
  - Use multiple error-detection mechanisms
  - For core of RCU, validation starts at the very beginning:
    - Write a design document: safety factors and conservative design
    - Consult with experts, update design as needed
    - Write code in pen on paper: Recopy until last two copies identical
    - Do proofs of correctness for anything non-obvious
    - Do full-up functional and stress testing
    - Document the resulting code (e.g., publish on LWN)
  - If I do all this, then there are probably only a few bugs left
    - And I detect those at least half the time

Summary

Summary: Many Promising Starts On Parallelism

- Linux kernel
- Apache
- MySQL (whichever fork you like)
- PostgreSQL
- Samba
- liburcu
- OpenMP
- POSIX threads
- C/C++ concurrency
- Intel TBB
- RapidMind
- Open Parallel
- Erlang
- Apple GCS
- CCAN
- Your project here...

Parallelism: Some fear is indeed warranted

Summary

But don't be a slave to your fear

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?