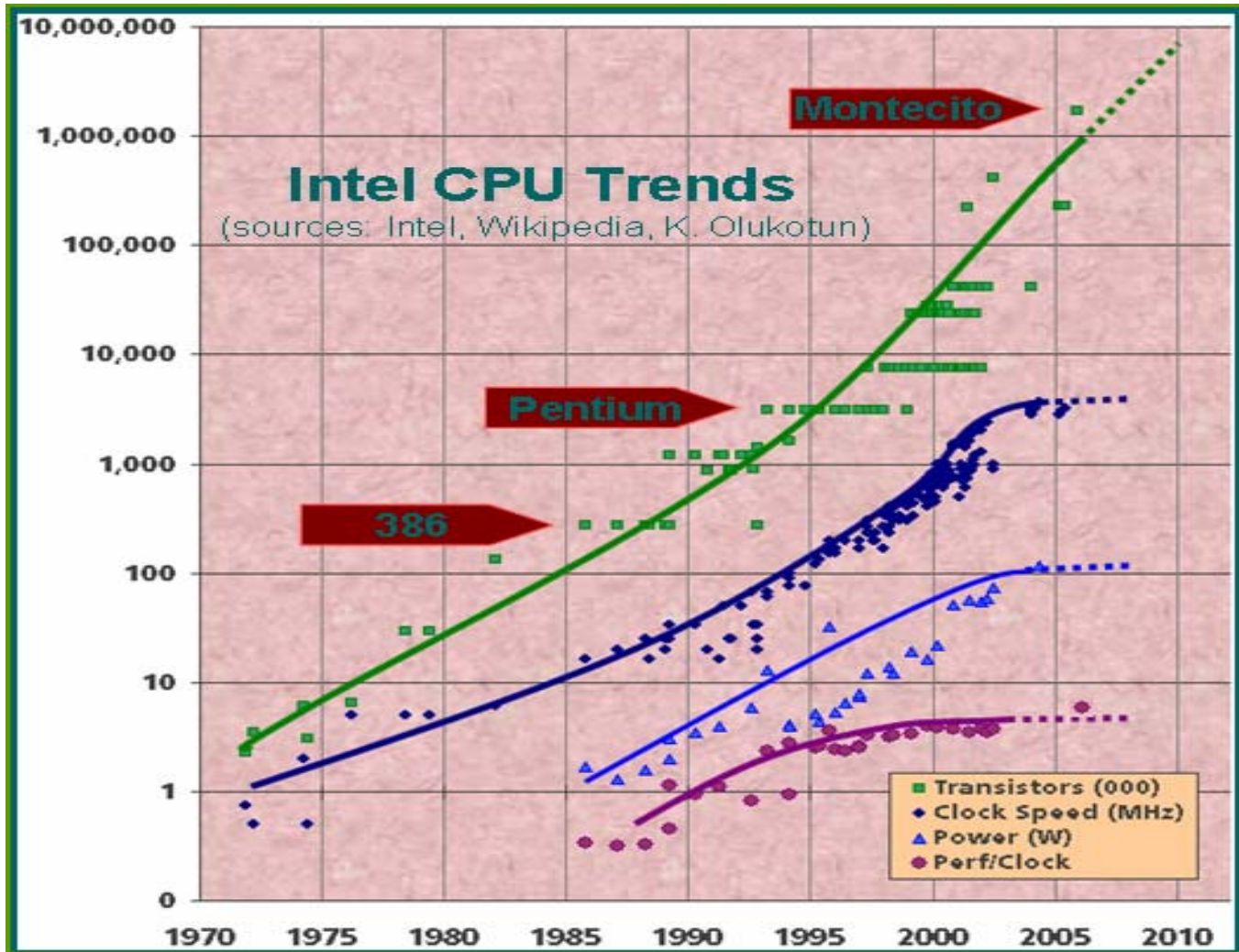


# *Reinventing Computing*

Burton Smith  
Technical Fellow  
Advanced Strategies and Policy

*Microsoft®*

# Times are Changing



[Thanks to Herb Sutter of Microsoft]

Microsoft®

# *Parallel Computing is Upon Us*

---

- Uniprocessor performance is leveling off
  - Instruction-level parallelism is near its limit (the ILP Wall)
  - Power per chip is getting painfully high (the Power Wall)
  - Caches show diminishing returns (the Memory Wall)
- Meanwhile, logic cost (\$ per gate-Hz) continues to fall
  - How are we going to use all that hardware?
- We expect new “killer apps” will need more performance
  - Semantic analysis and query
  - Improved human-computer interfaces (*e.g.* speech, vision)
  - Games!
- Microprocessors are now multi-core and/or multithreaded
  - But so far, it’s just “more of the same” architecturally
  - How are we going to program such systems?

# *The ILP Wall*

---

- There have been two popular approaches to ILP:
  - Vector instructions, including SSE and the like
  - The HPS<sup>†</sup> canon: out-of-order issue, in-order retirement, register renaming, branch prediction, speculation, ...
- Neither scheme generates much concurrency given a lot of:
  - Control-dependent computation
  - Data-dependent memory addressing (*e.g.* pointer-chasing)
- In practice, we are limited to a few instructions/clock
  - If you doubt this, ask your neighborhood computer architect
- Parallel computing is necessary for higher performance

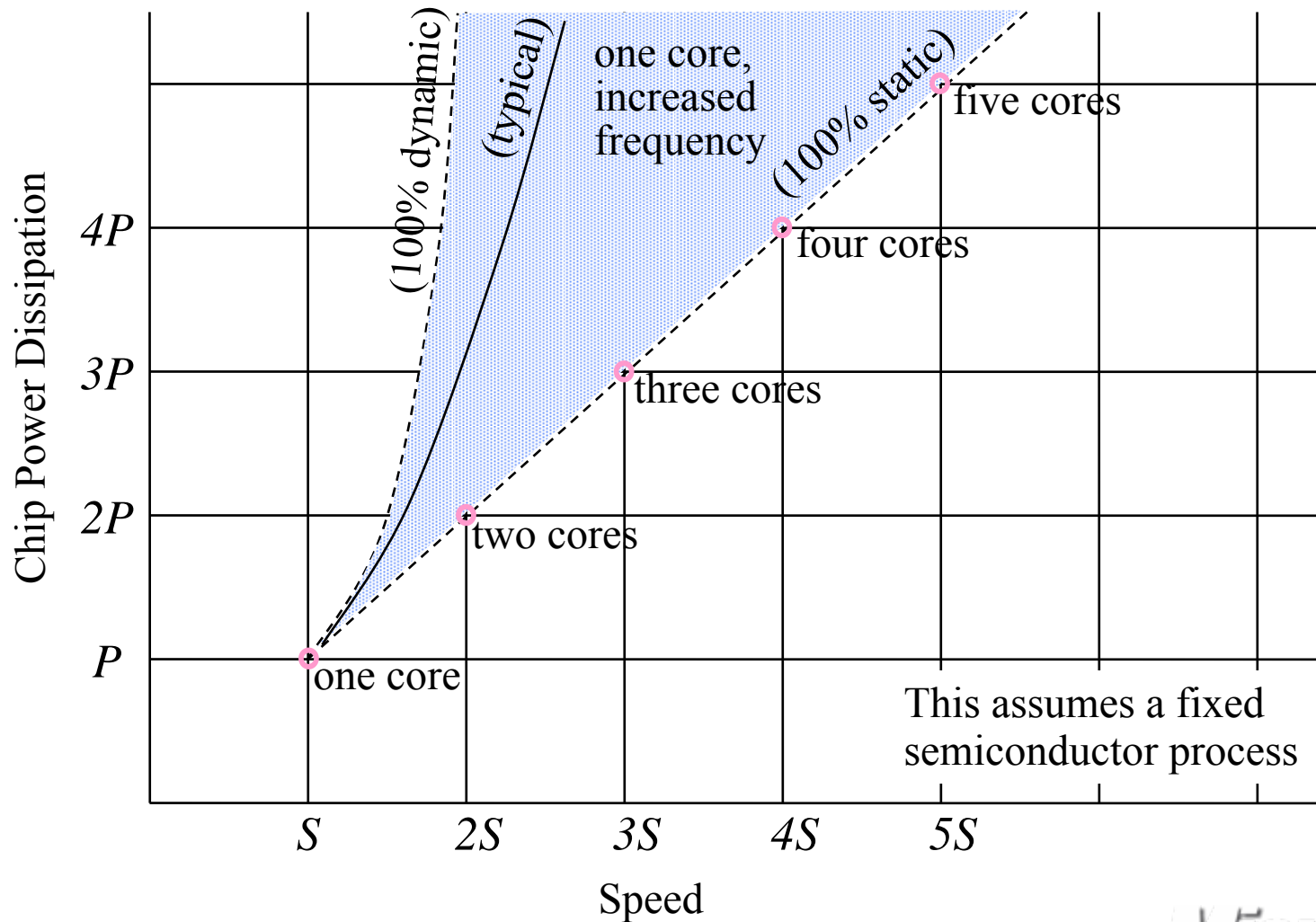
<sup>†</sup> Y.N. Patt et al., "*Critical Issues Regarding HPS, a High Performance Microarchitecture*," Proc. 18th Ann. ACM/IEEE Int'l Symp. on Microarchitecture, 1985, pp. 109–116.

# The Power Wall

---

- There are two ways to scale speed by a factor  $\sigma$ :
  - Scale the number of (running) cores by  $\sigma$ 
    - Power will scale by the same factor  $\sigma$
  - Scale the clock frequency  $f$  and voltage  $V$  by  $\sigma$ 
    - Dynamic power will scale by  $\sigma^3 (CV^2f)$
    - Static power will scale by  $\sigma (Vi_{\text{leakage}})$
    - Total power lies somewhere in between
- Clock scaling is worse when  $\sigma > 1$ 
  - This is part of the reason times are changing!
- Clock scaling is better when  $\sigma < 1$ 
  - Moral: if your multiprocessor is fully used but too hot, scale down voltage and frequency rather than processors
- Parallel computing is necessary to save power

# Power vs. Speed



# *The Memory Wall*

---

- We can build bigger caches from more plentiful transistors
  - Does this suffice, or is there a problem scaling up?
- To deliver twice the performance with the same aggregate DRAM bandwidth, the cache miss rate must be cut in half
- How much bigger does the cache have to be?<sup>†</sup>
  - For dense matrix-matrix multiply or dense LU, 4x bigger
  - For sorting or FFTs, the square of its former size
  - For sparse or dense matrix-vector multiply, forget it
- Faster clocks and deeper interconnect increase miss latency
  - Higher performance makes higher latency inevitable
- Latency and bandwidth are closely related

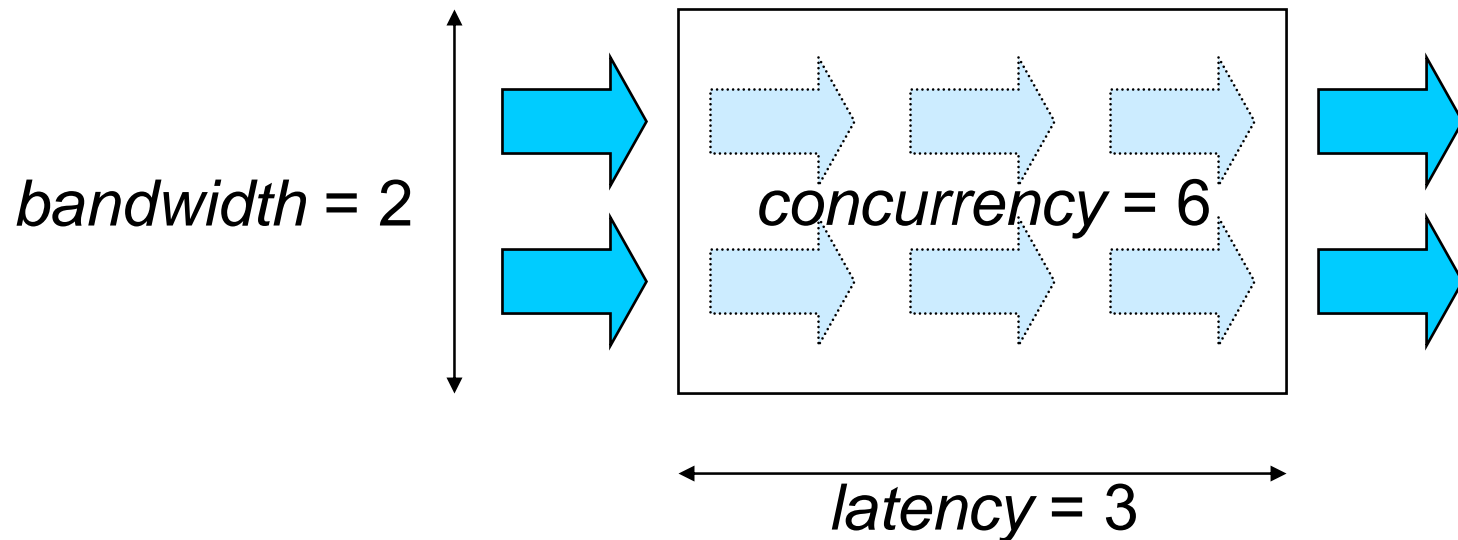
<sup>†</sup> H.T. Kung, “*Memory requirements for balanced computer architectures,*”  
13<sup>th</sup> International Symposium on Computer Architecture, 1986, pp. 49–54.

# *Latency, Bandwidth, & Concurrency*

- In any system that transports items from input to output without creating or destroying them,

$$\textit{latency} \times \textit{bandwidth} = \textit{concurrency}$$

- Queueing theory calls this result *Little's Law*



# *Overcoming the Memory Wall*

---

- Provide more memory bandwidth
  - Increase aggregate DRAM bandwidth per gigabyte
  - Increase the bandwidth of the chip pins
- Use multithreaded cores to tolerate memory latency
  - When latency increases, just increase the number of threads
  - Significantly, this does not change the programming model
- Use caches to improve bandwidth as well as latency
  - Make it easier for compilers to optimize locality
  - Keep cache lines short
  - Avoid mis-speculation in all its forms
- Parallel computing is necessary for memory balance

# *The von Neumann Assumption*

---

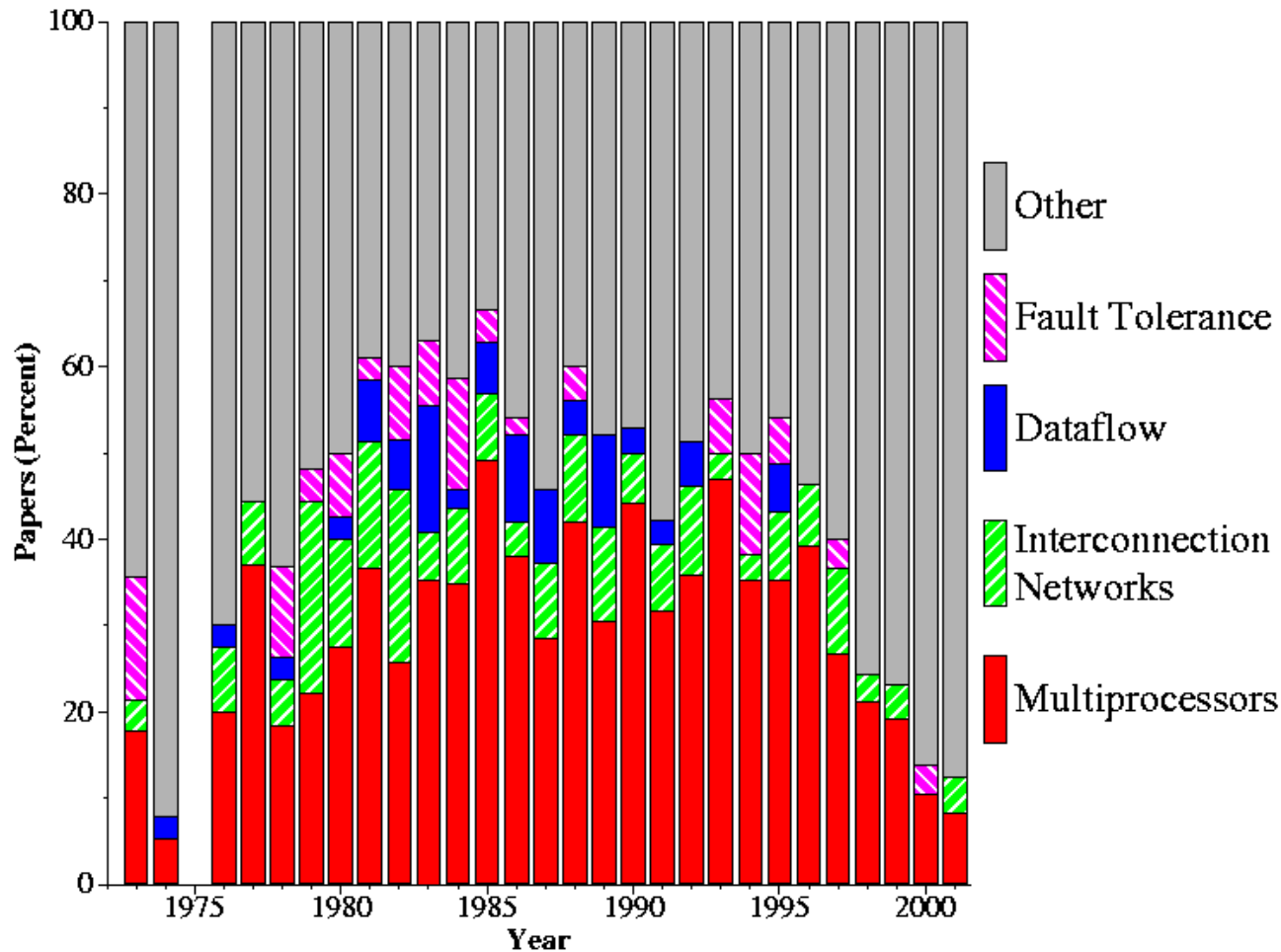
- We have relied on it for some 60 years
- Now it (and some things it brought along) must change
  - Serial execution lets programs *schedule values into variables*
  - Parallel execution makes this scheme hazardous
- Serial programming is easier than parallel programming
  - Serial programs are now becoming slow programs
- We need parallel programming paradigms that will make everyone who writes programs successful
- The stakes for our field's vitality are high
- Computing must be reinvented

# *How did we get into this fix?*

---

- Microprocessors kept getting faster at a tremendous pace
  - Better than 1000-fold in the last 20 years
- HPC was drawn into a spiral of specialization
  - “HPC applications are those things HPC systems do well”
  - The DARPA HPCS program is a response to this tendency
- University research on parallel systems dried up
  - No interest?
  - No ideas?
  - No need?
  - No money?
- A resurgence of interest in parallel computing is likely

# Architecture Conference Papers†



ISCA papers 1973-2001

† Mark Hill and Ravi Rajwar, *The Rise and Fall, etc.*

<http://pages.cs.wisc.edu/~markhill/mp2001.html>

Microsoft®

# *Lessons From the Past*

---

- A great deal is already known about parallel computing
  - Programming languages
  - Compiler optimization
  - Debugging and performance tuning
  - Operating systems
  - Architecture
- Most prior work was done with HPC in mind
  - Some ideas were more successful than others
  - Technical success doesn't always imply commercial success

# *Parallel Programming Languages*

---

- There are (at least) two promising approaches:
  - Functional programming
  - Atomic memory transactions
- Neither is completely satisfactory by itself
  - Functional programs don't allow mutable state
  - Transactional programs implement dependence awkwardly
- Data base applications show the synergy of the two ideas
  - SQL is a “mostly functional” language
  - Transactions allow updates with atomicity and isolation
- Many people think functional languages are inefficient
  - Sisal and NESL are excellent counterexamples
  - Both competed strongly with Fortran on Cray systems
- Others believe the same is true of memory transactions
  - This remains to be seen; we have only begun to optimize

*Microsoft*

# Transactions and Invariants

---

- Invariants are a program's *conservation laws*
  - Relationships among values in iteration and recursion
  - Rules of data structure (state) integrity
- If statements  $p$  and  $q$  preserve the invariant  $I$  and they do not “interfere”, their parallel composition  $\{ p \parallel q \}$  also preserves  $I$ <sup>†</sup>
- If  $p$  and  $q$  are performed atomically, *i.e.* as transactions, then they will not interfere<sup>‡</sup>
- Although operations seldom commute with respect to state, transactions give us *commutativity with respect to the invariant*
- It would be nice if the invariants were available to the compiler
  - Can we get programmers to supply them?

<sup>†</sup> Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. CACM **19**(5):279–285, May 1976.

<sup>‡</sup> Leslie Lamport and Fred Schneider. The “Hoare Logic” of CSP, And All That. ACM TOPLAS **6**(2):281–296, Apr. 1984.

# A LINQ Example<sup>†</sup>

- LINQ stands for “Language Integrated Query”
  - It is a new enhancement to C# and Visual Basic (F# soon)
  - It operates on data in memory or in an external database

```
public void Linq93() {  
    double startBalance = 100.0;  
  
    int[] attemptedWithdrawals = { 20, 10, 40, 50, 10, 70, 30 };  
  
    double endBalance =  
        attemptedWithdrawals.Fold(startBalance,  
            (balance, nextWithdrawal) =>  
                ( nextWithdrawal <= balance ) ? ( balance - nextWithdrawal ) : balance );  
  
    Console.WriteLine("Ending balance: {0}", endBalance);  
}
```

**Result:** Ending balance: 20

- PLINQ might need a transaction within the lambda body

<sup>†</sup> From “101 LINQ Samples”, [msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx](http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx)

# *Styles of Parallelism*

---

- We need to support multiple programming styles
  - Both functional and transactional
  - Both data parallel and task parallel
  - Both message passing and shared memory
  - Both declarative and imperative
  - Both implicit and explicit
- We may need several languages to accomplish this
  - After all, we do use multiple languages today
  - Language interoperability (*e.g.* .NET) will help greatly
- It is essential that parallelism be exposed to the compiler
  - So that the compiler can adapt it to the target system
- It is also essential that locality be exposed to the compiler
  - For the same reason

# *Compiler Optimization for Parallelism*

---

- Some say automatic parallelization is a demonstrated failure
  - Vectorizing and parallelizing compilers (especially for the right architecture) have been a tremendous success
  - They have enabled machine-independent languages
  - What they do can be termed *parallelism packaging*
  - Even manifestly parallel programs need it
- What failed is *parallelism discovery*, especially in-the-large
  - Dependence analysis is chiefly a local success
- *Locality discovery* in-the-large has also been a non-starter
  - Locality analysis is another word for dependence analysis
- The jury is still out on large-scale *locality packaging*

# *Parallel Debugging and Tuning*

---

- Today, debugging relies on single-stepping and `printf()`
  - Single-stepping a parallel program is much less effective
- Conditional data breakpoints have proven to be valuable
  - To stop when an invariant fails to be true
- Support for ad-hoc data perusal is also very important
  - This is a kind of data mining application
- Serial program tuning tries to discover where the program counter spends most of its time
  - The answer is usually discovered by sampling
- In contrast, parallel program tuning tries to discover places where there is insufficient parallelism
  - A proven approach has been event logging with timestamps

# *Operating Systems for Parallelism*

---

- Operating systems must stop trying to schedule processors
  - Their job should be allocating processors and other resources
  - Resource changes should be negotiated with the user runtime
- Work should be scheduled at user level
  - There's no need for a change of privilege
  - Locality can be better preserved
  - Optimization becomes much more possible
  - Blocked computations can become first-class
- Quality of service has become important for some uses
  - Deadlines are more relevant than priorities in such cases
- Demand paging is a bad idea for most parallel applications
  - Everything ends up waiting on the faulting computation

# *Parallel Architecture*

---

- Hardware has had a head start at parallelism
  - That doesn't mean it's way ahead!
- Artifacts of the von Neumann assumption abound
  - Interrupts, for example
  - Most of these are pretty easy to repair
- A bigger issue is support for fine-grain parallelism
  - Thread granularity depends on the amount of state per thread and on how much it costs to swap it when the thread blocks
- Another is whether all processors should look the same
  - There are options for heterogeneity
    - Heterogeneous architectures or heterogeneous implementations
  - Shared memory can be used to communicate among them
  - Homogeneous architectural data types will help performance
- The biggest issue may be how to maintain system balance

# *Consequences for HPC*

---

- HPC has been a lonely parallel outpost in a serial world
- Now parallel computing is quickly becoming mainstream
- Consequences for HPC are likely to be:
  - Adaptation and use of mainstream software for HPC
  - Routine combining of shared memory and message passing
  - A broadening spectrum of programming language choices
- Successful HPC product offerings might include:
  - HPC editions of emerging client applications and tools
  - HPC services that enable or accelerate client applications
  - HPC systems that scale up the client architectural model
- HPC will need to be reinvented along with everything else

# *Conclusions*

---

- We must now rethink many of the basics of computing
- There is lots of work for everyone to do
  - I've left some subjects out, especially applications
- We have significant valuable experience with parallelism
  - Much of it will be applicable going forward