High Performance Computing for Science and Engineering I

Lecturers: Petros Koumoutsakos & Matthias Troyer

TA's: *Panagiotis Chatzidoukas* - Christian Conti - Michael Dolfi - Andreas Hehn - Jonas Sukys

CREDITS

- SOME NOTES ARE TAKEN FROM:
- THE EXCELLENT WEB TUTORIAL OF **BLAISE BARNEY**
  - https://computing.llnl.gov/tutorials/parallel_comp/
- LECTURES BY **JAMES DEMMEL** (UC BERKELEY)

OUTLINE

- What and Why High Performance Computing (HPC) ?
- Large/**All** Computational Science and Engineering problems require HPC
- Why this course OR Why writing parallel programs is hard
- Structure of the course

Units of Measure

- High Performance Computing (HPC) units are:
  - **Flop:** floating point operation, usually double precision unless noted
  - **Flop/s:** floating point operations per second
  - **Bytes:** size of data (a double precision floating point number is 8 bytes)
- Typical sizes are millions, billions, trillions…
  - Mega Mflop/s = 10^6 flop/sec Mbyte = 2^20 = 1048576 ~ 10^6 bytes
  - Giga Gflop/s = 10^9 flop/sec Gbyte = 2^30 ~ 10^9 bytes
  - **Tera Tflop/s = 10^12 flop/sec Tbyte = 2^40 ~ 10^12 bytes**
  - **Peta Pflop/s = 10^15 flop/sec Pbyte = 2^50 ~ 10^15 bytes**
  - **Exa Eflop/s = 10^18 flop/sec Ebyte = 2^60 ~ 10^18 bytes**
  - Zetta Zflop/s = 10^21 flop/sec Zbyte = 2^70 ~ 10^21 bytes
  - Yotta Yflop/s = 10^24 flop/sec Ybyte = 2^80 ~ 10^24 bytes
- Current fastest (public) machine ~ 55 Pflop/s, 3.1M cores
  - Up-to-date list at www.top500.org

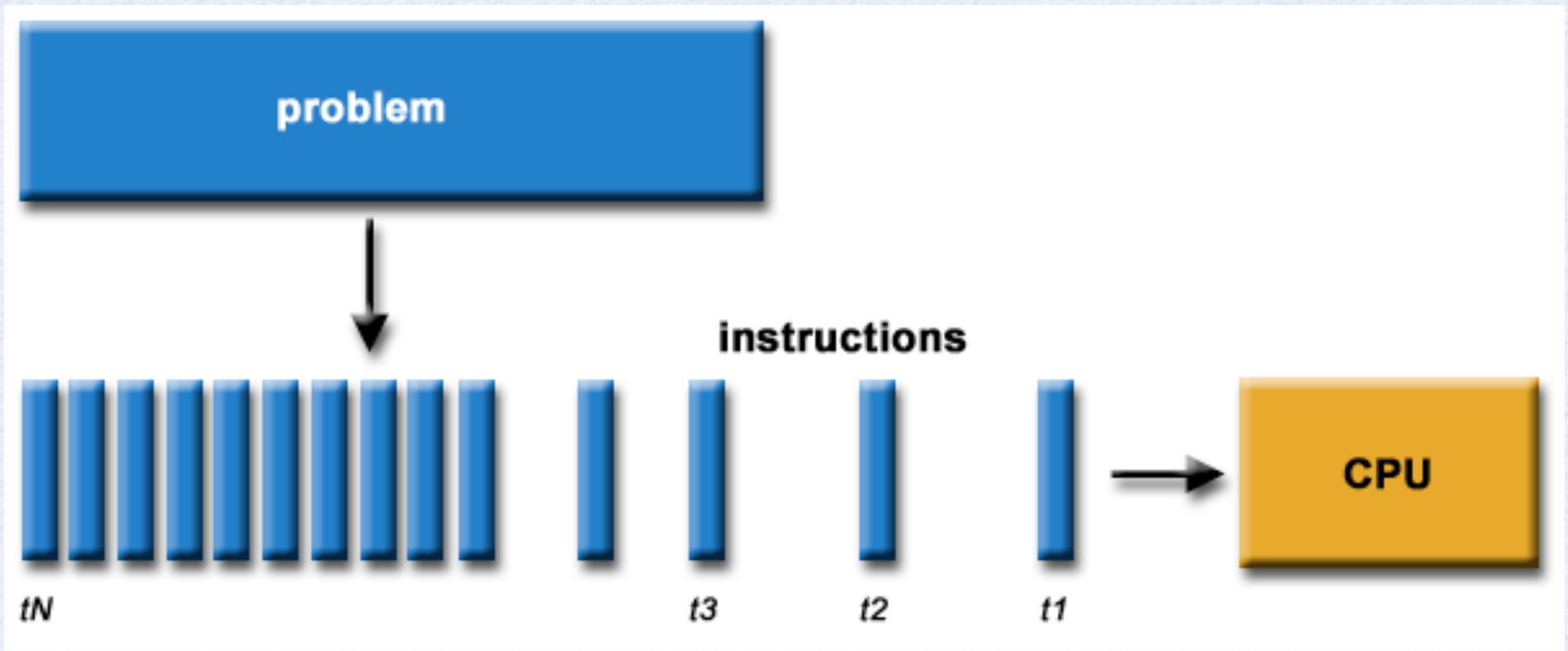
Why powerful ~~computers~~ *all* are parallel (2007 -)

circa 1991-2006

CREDIT: J. Demmel

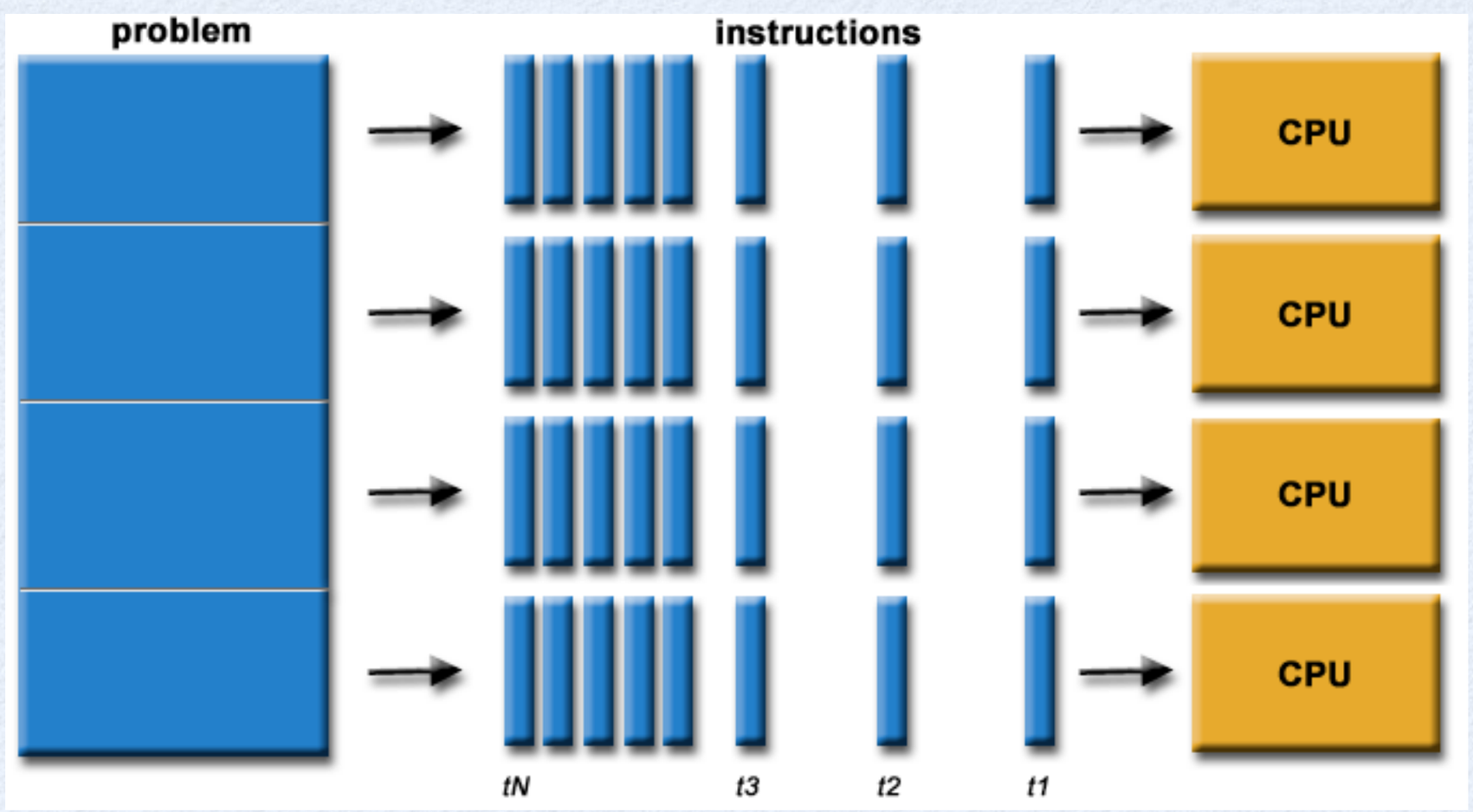
Serial Computing

- Serial Implementation of Programs for single CPUs
  - Program has series of Instructions
  - Executed one after the other
  - One instruction at a time



Parallel Computing

- Program is executed using Multiple Machines
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different machines (here : CPUs)



When do I need Parallelism ?

A program needs to be:

- correct
- solve an important problem
- provide a useful interface (to people and other programs)

OK
Sequential

- Fast
- Throughput
- UQ
- Optimization

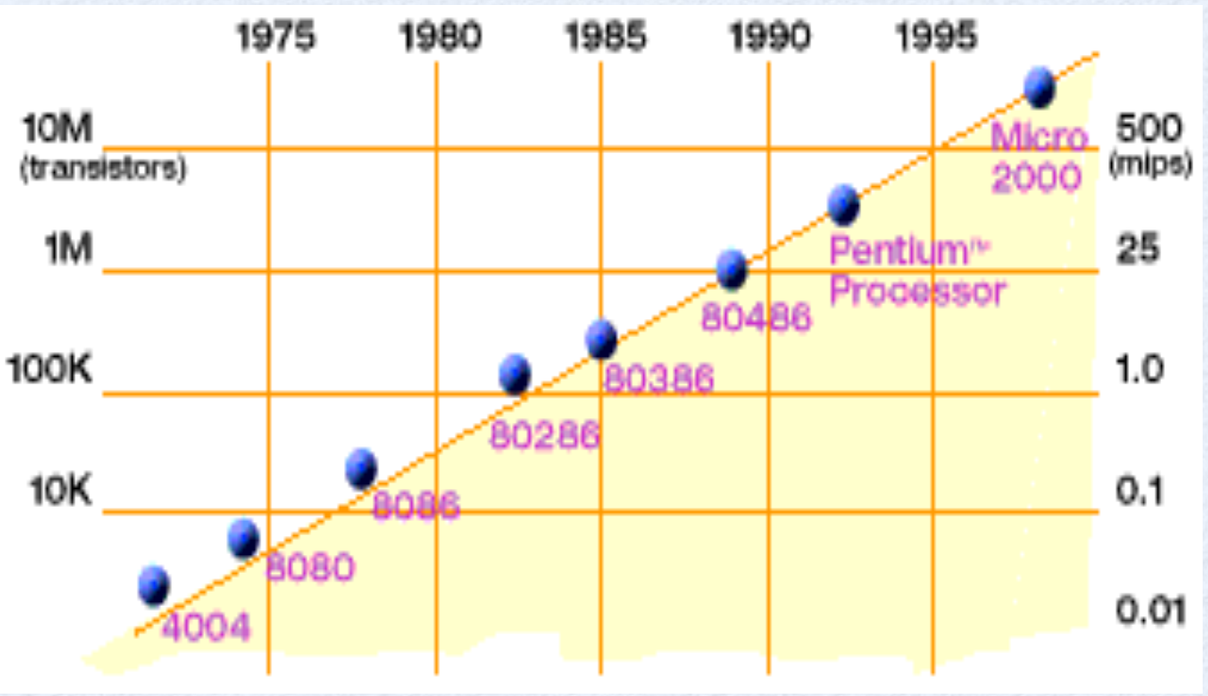
Only
Parallel

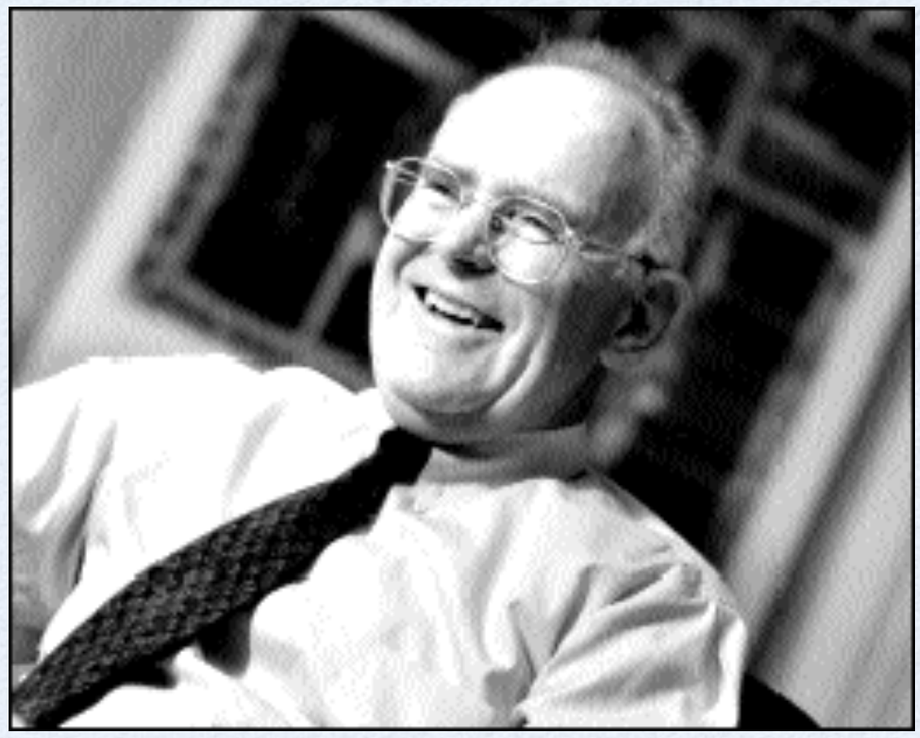
Tunnel Vision by Experts

- "I think there is a world market for maybe five computers." Thomas Watson, chairman of IBM, 1943.
- "There is no reason for any individual to have a computer in their home" Ken Olson, president and founder of Digital Equipment Corporation, 1977.
- "640K [of memory] ought to be enough for anybody." Bill Gates, chairman of Microsoft,1981.
- "On several recent occasions, I have been asked whether parallel computing will soon be relegated to the trash heap reserved for promising technologies that never quite make it." Ken Kennedy, CRPC Directory, 1994

Slide source: Warfield et al.

Technology Trends: Microprocessor Capacity





2X transistors/Chip Every 1.5 years
Called "Moore's Law"

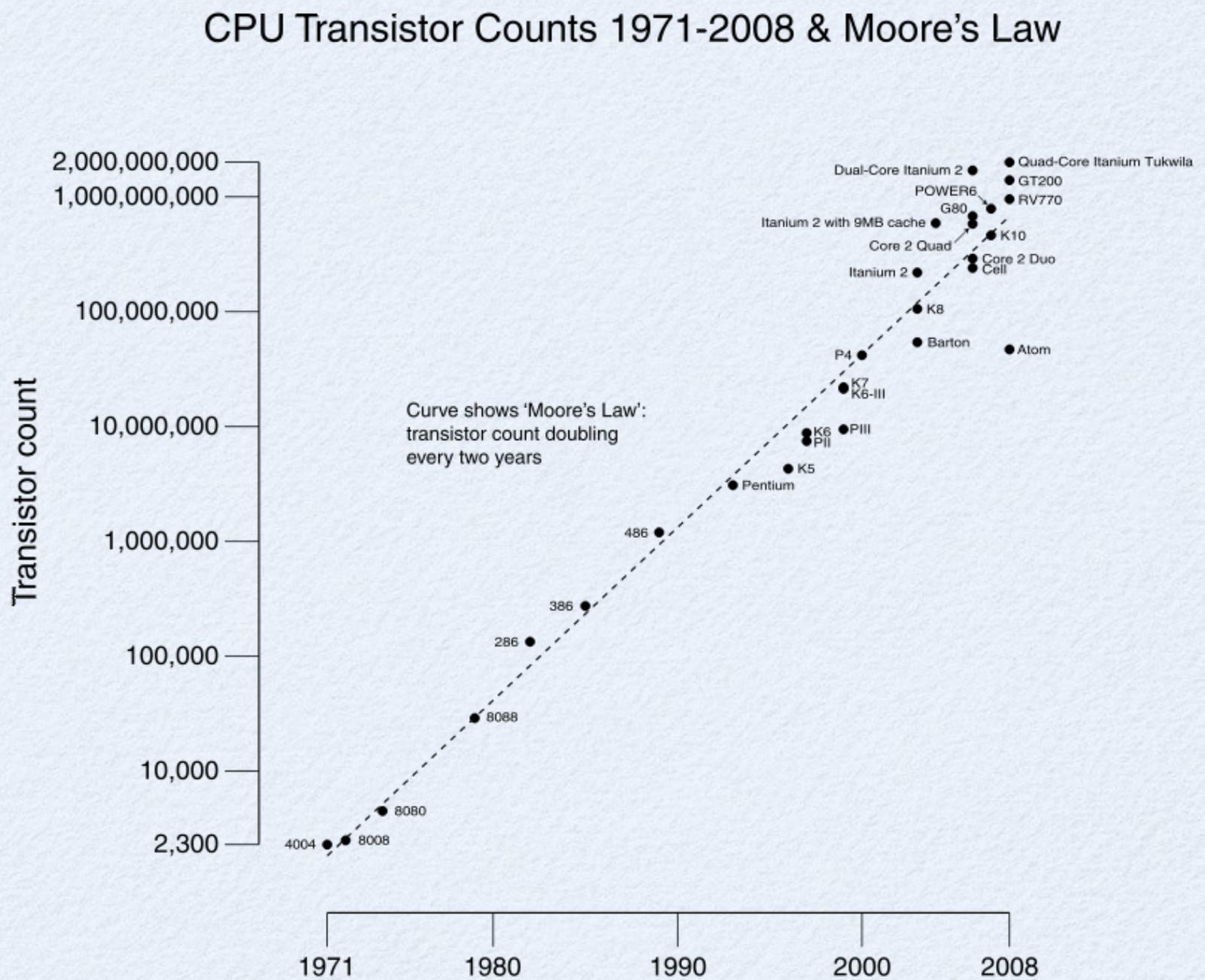
Microprocessors have become smaller, denser, and more powerful.

Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

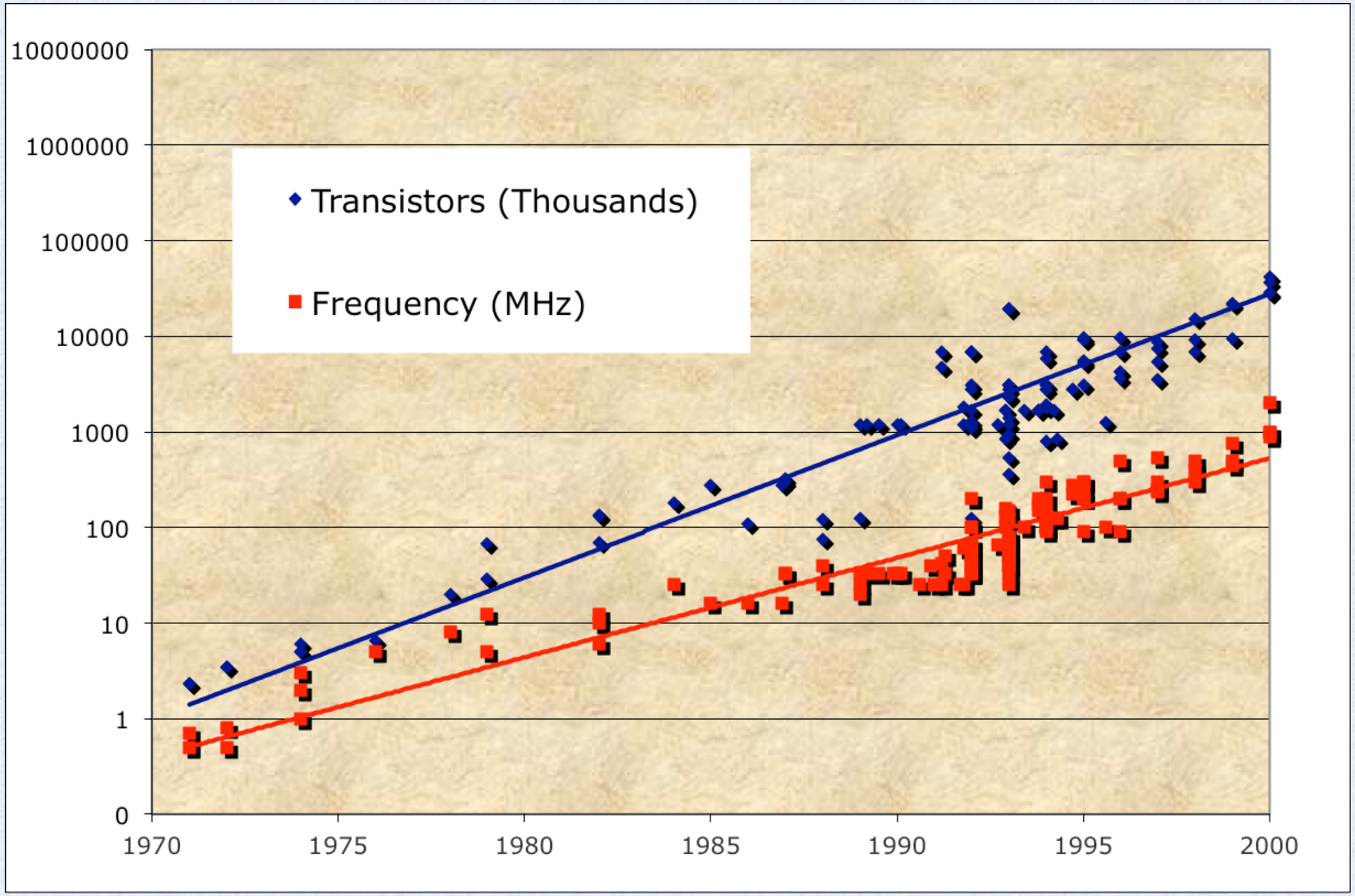
Slide source: Jack Dongarra

Moore's Law (still alive and well)

The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years



Microprocessor Transistors / Clock (1970–2000)



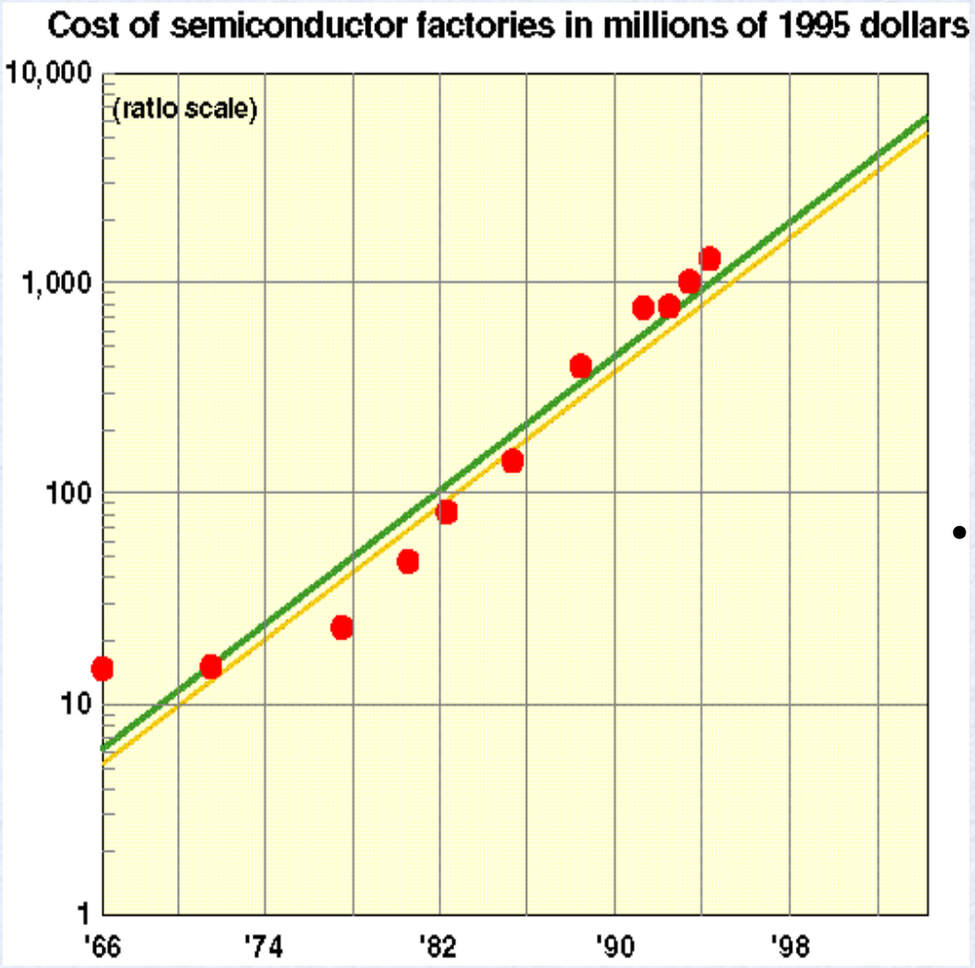
Impact of Device Shrinkage

What happens when the feature size (transistor size) shrinks by a factor of x ?

- Clock rate goes up by x because wires are shorter
  - actually less than x, because of power consumption
- Transistors per unit area goes up by x^2
- Die size also tends to increase
  - typically another factor of ~x
- Raw computing power of the chip goes up by ~ x^4 !
  - typically x^3 is devoted to either on-chip
    - parallelism: hidden parallelism such as ILP
    - locality: caches
- So most programs x^3 times faster, without changing them

CREDIT: J. Demmel

PROBLEMS : Costs of Manufacturing



CREDIT: J. Demmel

Manufacturing costs and yield problems limit use of density

- **Yield: What percentage of the chips are usable?**
  - E.g., Cell processor (PS3) was sold with 7 out of 8 “on” to improve yield

The power wall

- Dominant technology for integrated circuits is called CMOS (Complementary Metal-Oxide Semiconductors)

- For CMOS the primary source of power dissipation is the so-called dynamic power - power consumed in switching

Power = Capacitive Load per Transistor x Voltage^2 x Frequency switched

Function of number of transistors connected to an output (fanout) and technology that determines the capacitance of both wires and transistors

This is a function of the Clock Rate

The power wall

- How can power grow by 30x while clock rates by 10^3x ?
  Reduce the voltage! (~15% per generation)

- Further lowering the voltage makes the transistors too leaky (like water faucets) that cannot shut off

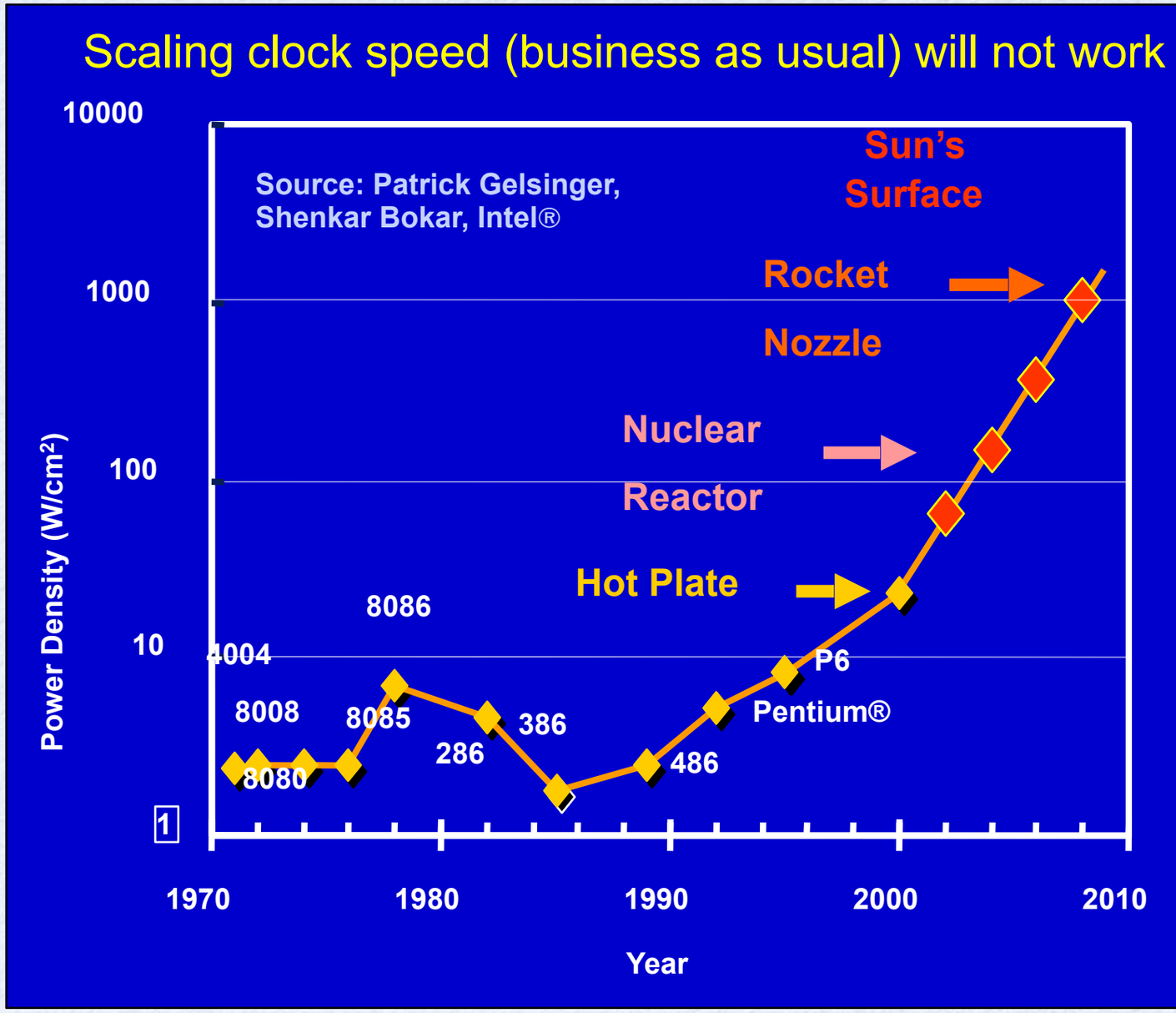
- Even today 40% of consumption is due to leakage (**leakage current**, gate leakage)

- Alternative is to take that power and use it more efficiently or to cool the computers

After hitting the wall ⟶ MULTICORE

Power Density Limits Serial Performance

- Concurrent systems are more power efficient
  - Dynamic power is proportional to V^2fC
  - Increasing frequency (f) also increases supply voltage (V) → cubic effect
  - Increasing cores increases capacitance (C) but only linearly
  - Save power by lowering clock speed



CREDIT: J. Demmel

- High performance serial processors waste power
  - Speculation, dynamic dependence checking, etc. burn power
  - Implicit parallelism discovery
- **More transistors, but not faster serial processors**

Multicore

- The sea change: since 2006 all microprocessor companies are shipping computers with multiple cores per chip
- New Moore: Double the number of cores per microprocessor per semi-conductor technology generation every two years

OLD	⟶	Hardware, Architecture, Compilers	⟶ Speed
NEW	⟶	Parallel programming	⟶ Speed

Single processor performance is no longer tracking Moore's Law

Moore's Law ran smoothly until 2002, when the gap between performance and gate count started to appear.

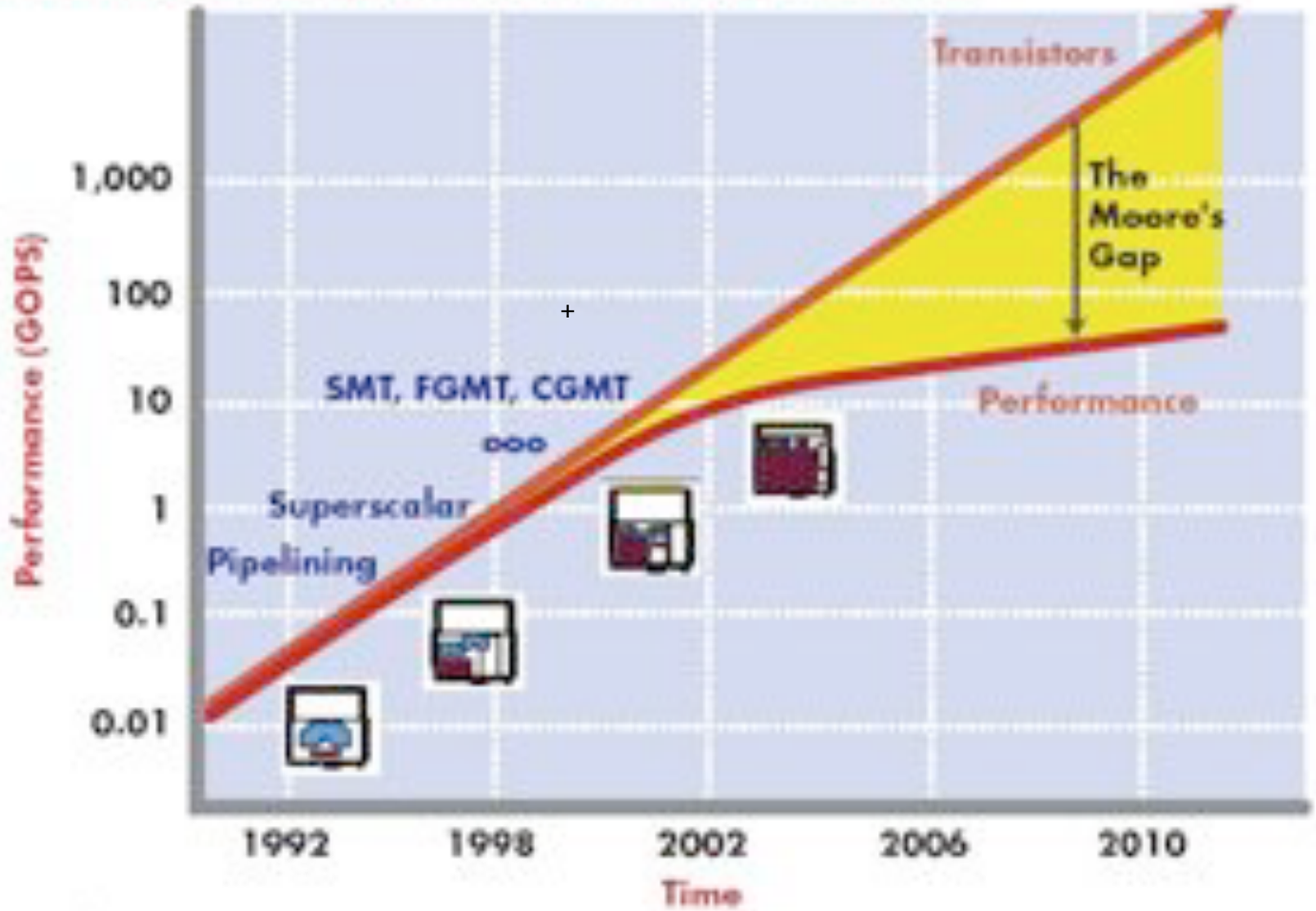
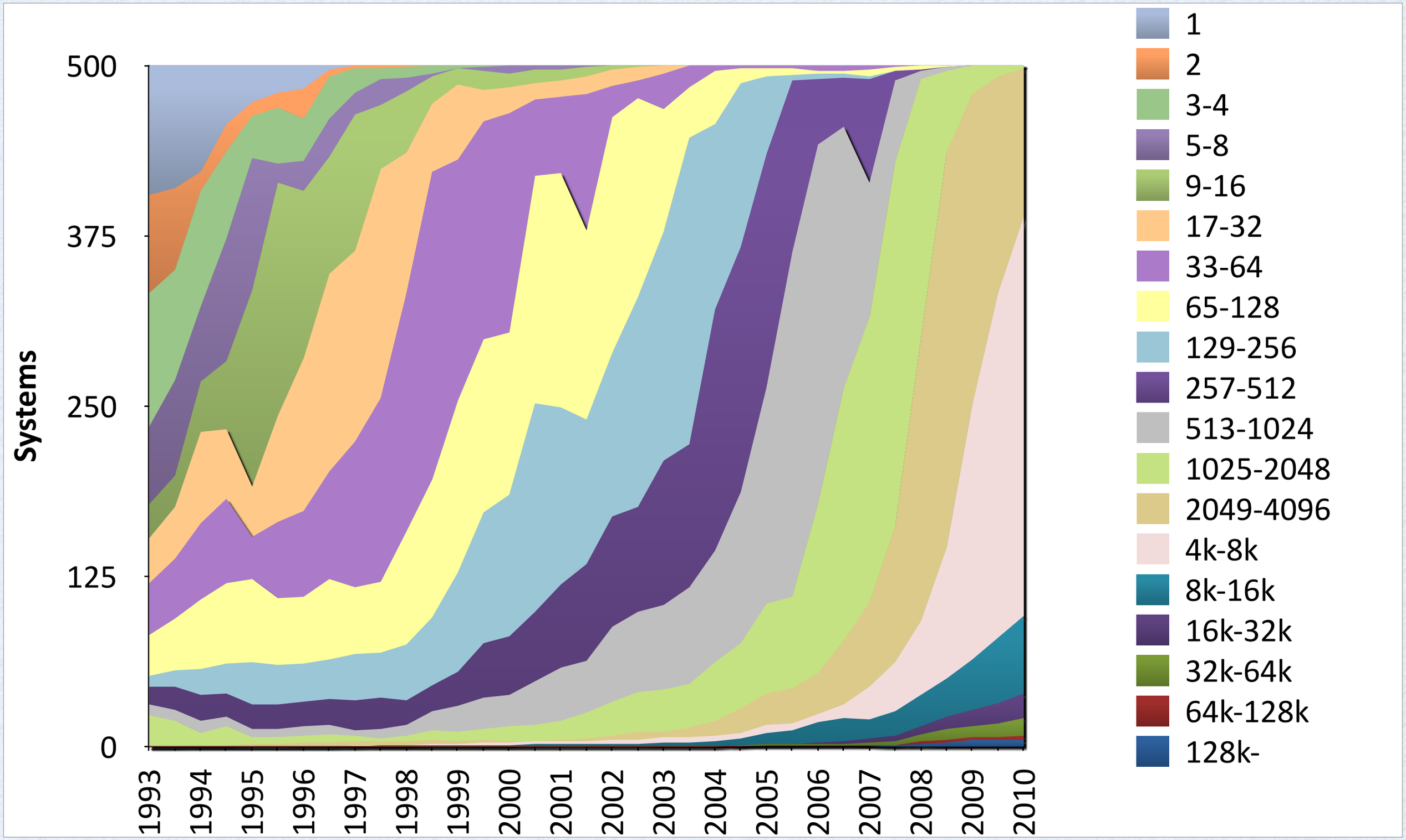
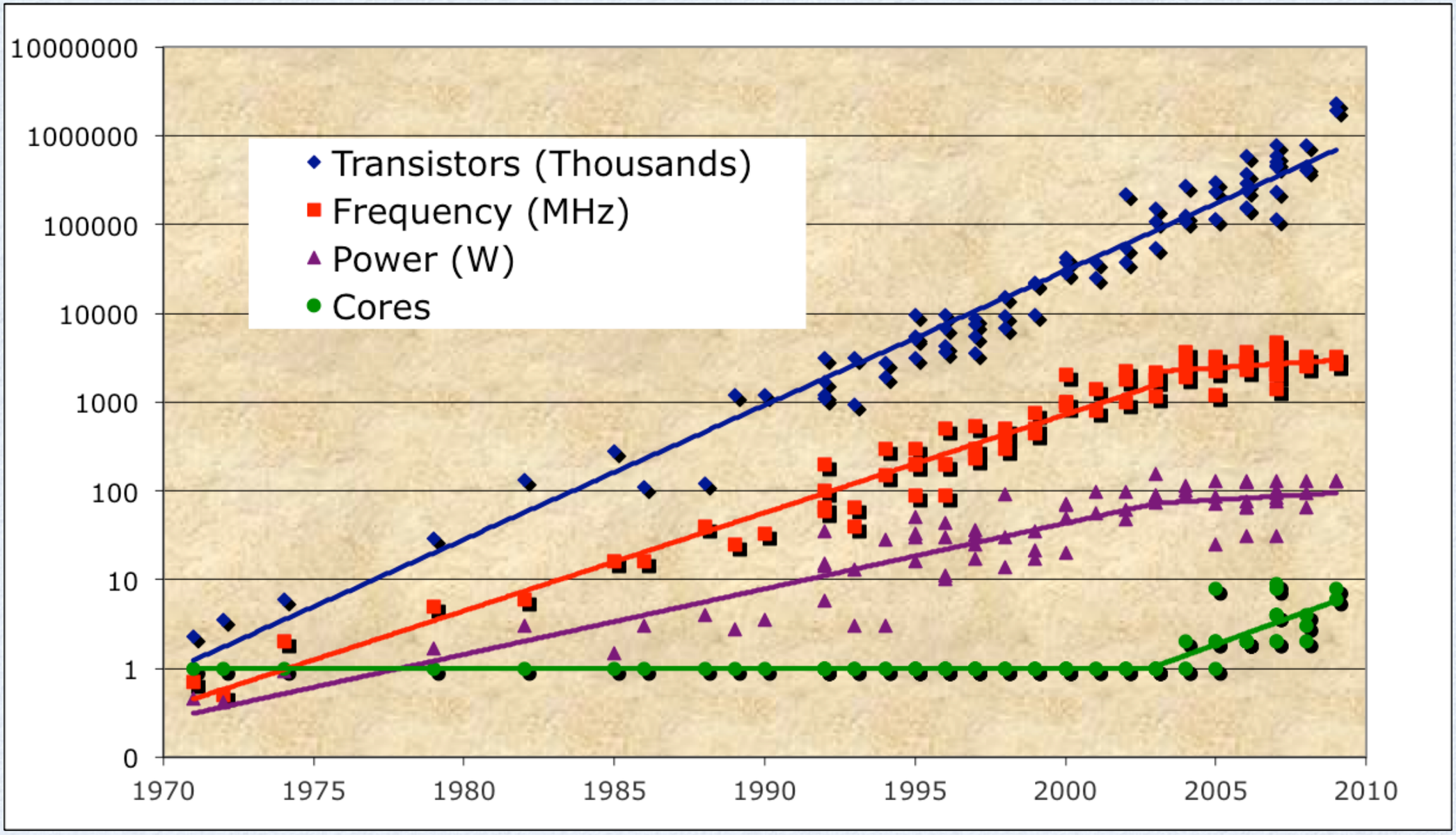


Figure 2

Core Count


500
375
250
125
0
Systems
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
1
2
3-4
5-8
9-16
17-32
33-64
65-128
129-256
257-512
513-1024
1025-2048
2049-4096
4k-8k
8k-16k
16k-32k
32k-64k
64k-128k
128k-

Revolution in Processors



- Chip density is continuing increase ~2x every 2 years
- Clock speed is not
- Number of processor cores may double instead
- Power is under control, no longer growing

Parallelism in 2014?

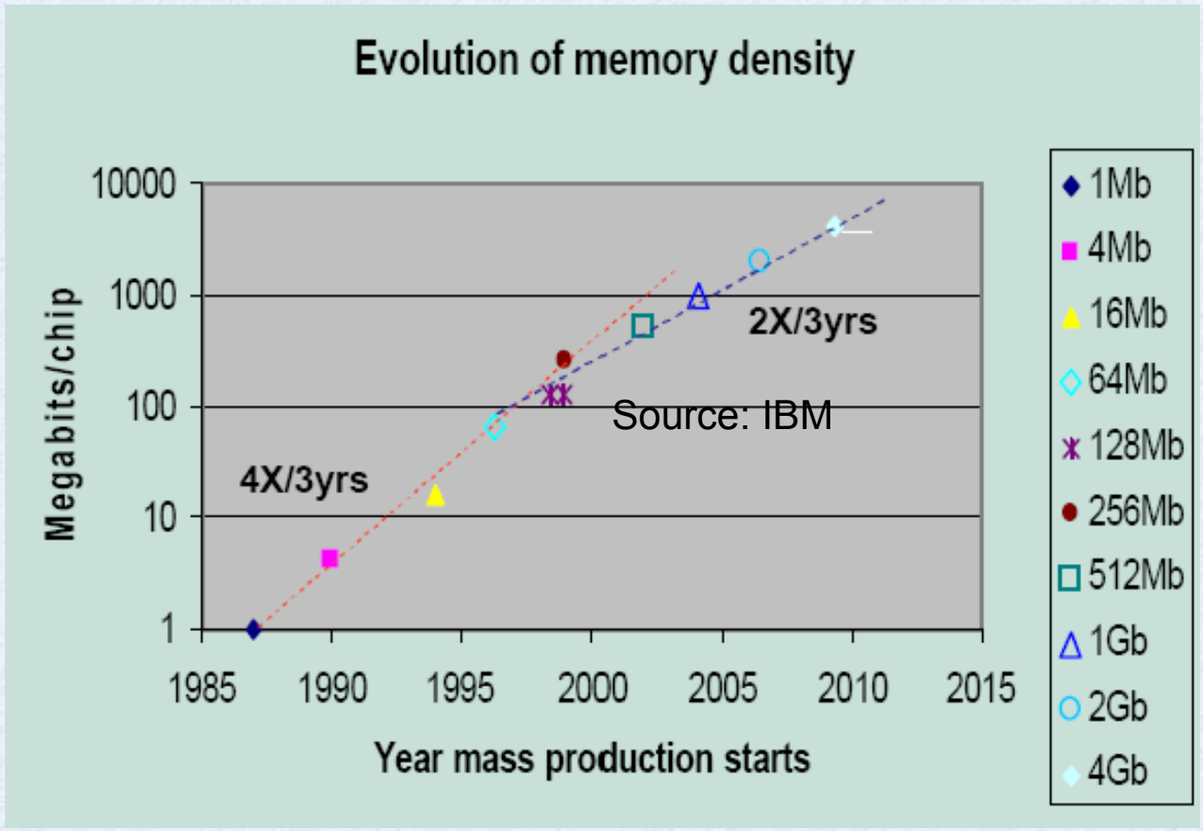
- These arguments are no longer theoretical
- All major processor vendors are producing *multicore* chips
  - Every machine will soon be a parallel machine
  - To keep doubling performance, parallelism must double
- Which (commercial) applications can use this parallelism?
  - Do they have to be rewritten from scratch?
- Will all programmers have to be parallel programmers?
  - New software model needed
  - Try to hide complexity from most programmers – eventually
  - In the meantime, need to understand it
- Computer industry betting on this big change, but does not have all the answers

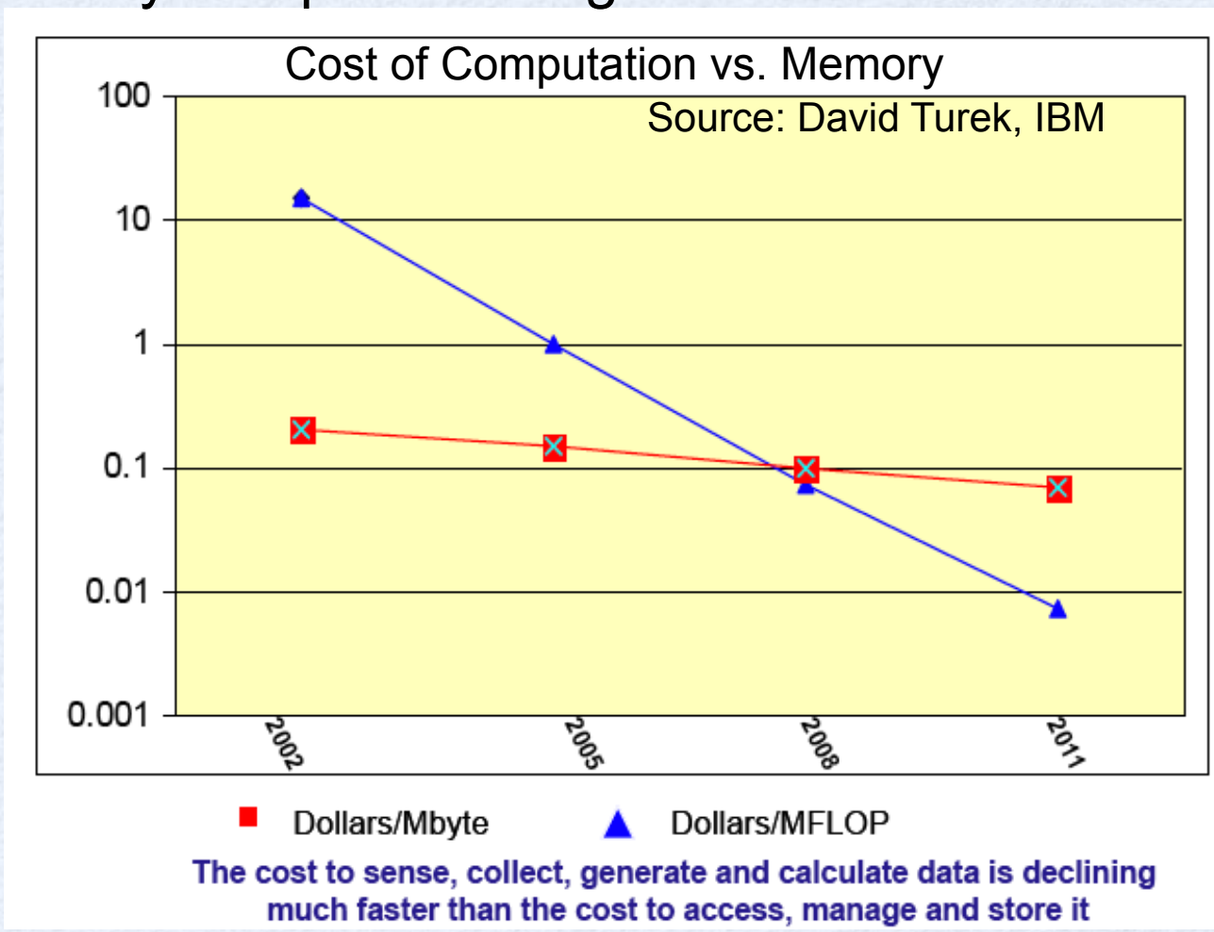
CREDIT: J. Demmel

Memory is Not Keeping Pace

Technology trends against a constant or increasing memory per core

- Memory density is doubling every three years; processor logic is every two
- Storage costs (dollars/Mbyte) are dropping gradually compared to logic costs





CREDIT: J. Demmel

QUESTION: Can you double concurrency without doubling memory?

- **Strong scaling:** fixed problem size, increase number of processors
- **Weak scaling:** grow problem size proportionally to number of processors

Moore's Law reinterpreted

- **Number of cores per chip can double every two years**
- **Clock speed will not increase (possibly decrease)**
- **Need to deal with systems with millions of concurrent threads**
- **Need to deal with inter-chip parallelism as well as intra-chip parallelism**

CREDIT: J. Demmel

The TOP500 Project

- Listing the 500 most powerful computers in the world
- Yardstick: Rmax of Linpack
  - Solve Ax=b, dense problem, matrix is random
  - Dominated by dense matrix-matrix multiply
- Updated twice a year:
  - ISC'xy in June in Germany
  - SCxy in November in the U.S.
- TOP500 web site at: www.top500.org

42st List: The TOP9

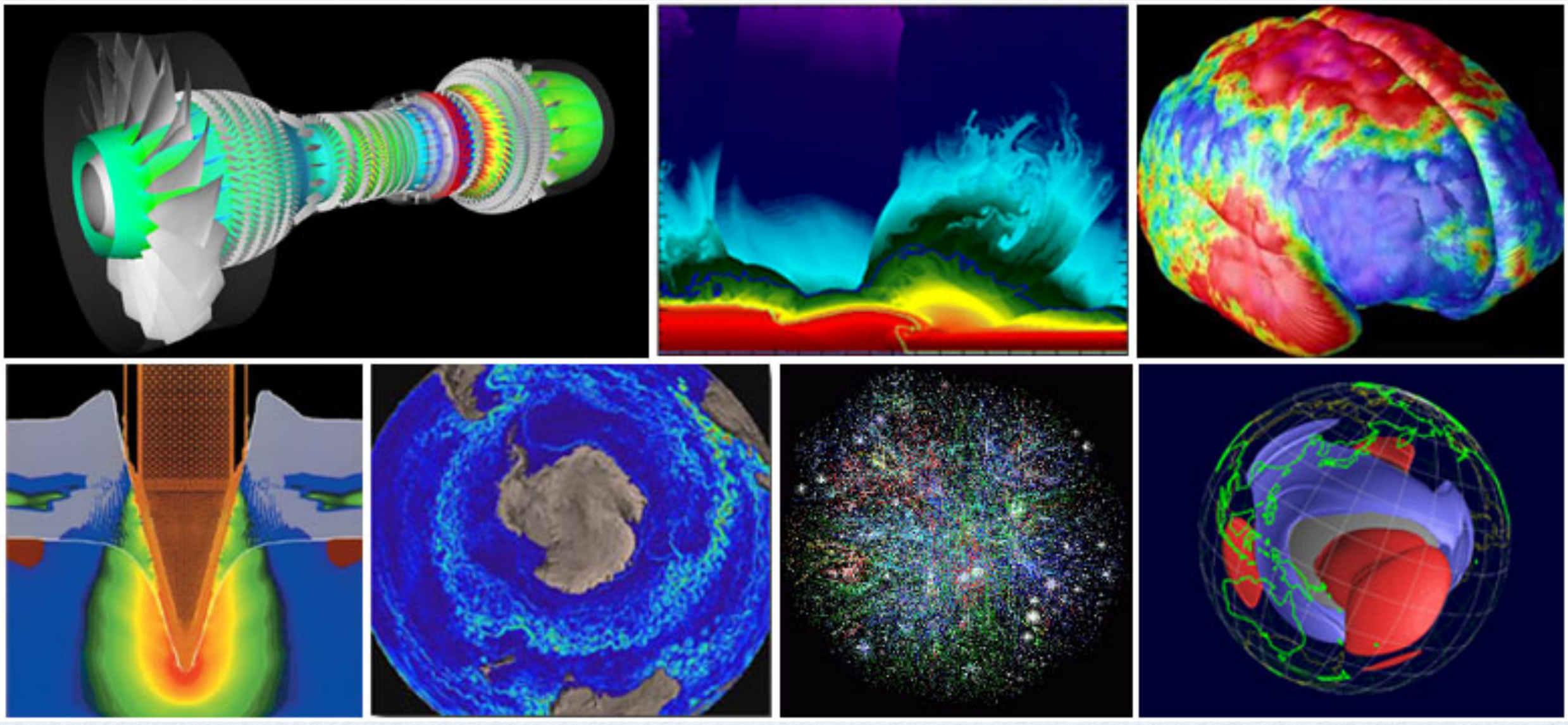
	Site	Manufacturer	Computer	Country	Cores	Rmax [Pflops]	Power [MW]
1	National University of Defense Technology	NUDT	Tianhe-2 NUDT TH-IVB-FEP, Xeon 12C 2.2GHz, IntelXeon Phi	China	3,120,000	33.9	17.8
2	Oak Ridge National Laboratory	Cray	Titan Cray XK7, Opteron 16C 2.2GHz, Gemini, NVIDIA K20x	USA	560,640	17.6	8.21
3	Lawrence Livermore National Laboratory	IBM	Sequoia BlueGene/Q, Power BQC 16C 1.6GHz, Custom	USA	1,572,864	17.2	7.89
4	RIKEN Advanced Institute for Computational Science	Fujitsu	K Computer SPARC64 VIIIfx 2.0GHz, Tofu Interconnect	Japan	795,024	10.5	12.7
5	Argonne National Laboratory	IBM	Mira BlueGene/Q, Power BQC 16C 1.6GHz, Custom	USA	786,432	8.59	3.95
6	Swiss National Supercomputing Centre (CSCS)	Cray	Piz Daint Cray XC30, Xeon E5 8C 2.6GHz, Aries, NVIDIA K20x	Switzerland	115,984	6.27	2.33
7	Texas Advanced Computing Center/ UT	Dell	Stampede PowerEdge C8220, Xeon E5 8C 2.7GHz, Intel Xeon Phi	USA	462,462	5.17	4.51
8	Forschungszentrum Juelich (FZJ)	IBM	JuQUEEN BlueGene/Q, Power BQC 16C 1.6GHz, Custom	Germany	458,752	5.01	2.30
9	Lawrence Livermore National Laboratory	IBM	Vulcan BlueGene/Q, Power BQC 16C 1.6GHz, Custom	USA	393,216	4.29	1.97

Computational Science – News

"An important development in sciences is occurring at the intersection of computer science and the sciences that has the potential to have a profound impact on science. It is a leap from the application of computing … to the *integration of computer science concepts, tools, and theorems* into the very fabric of science." -*Science* 2020 Report, March 2006

PAST : Parallel Computing = High End Science

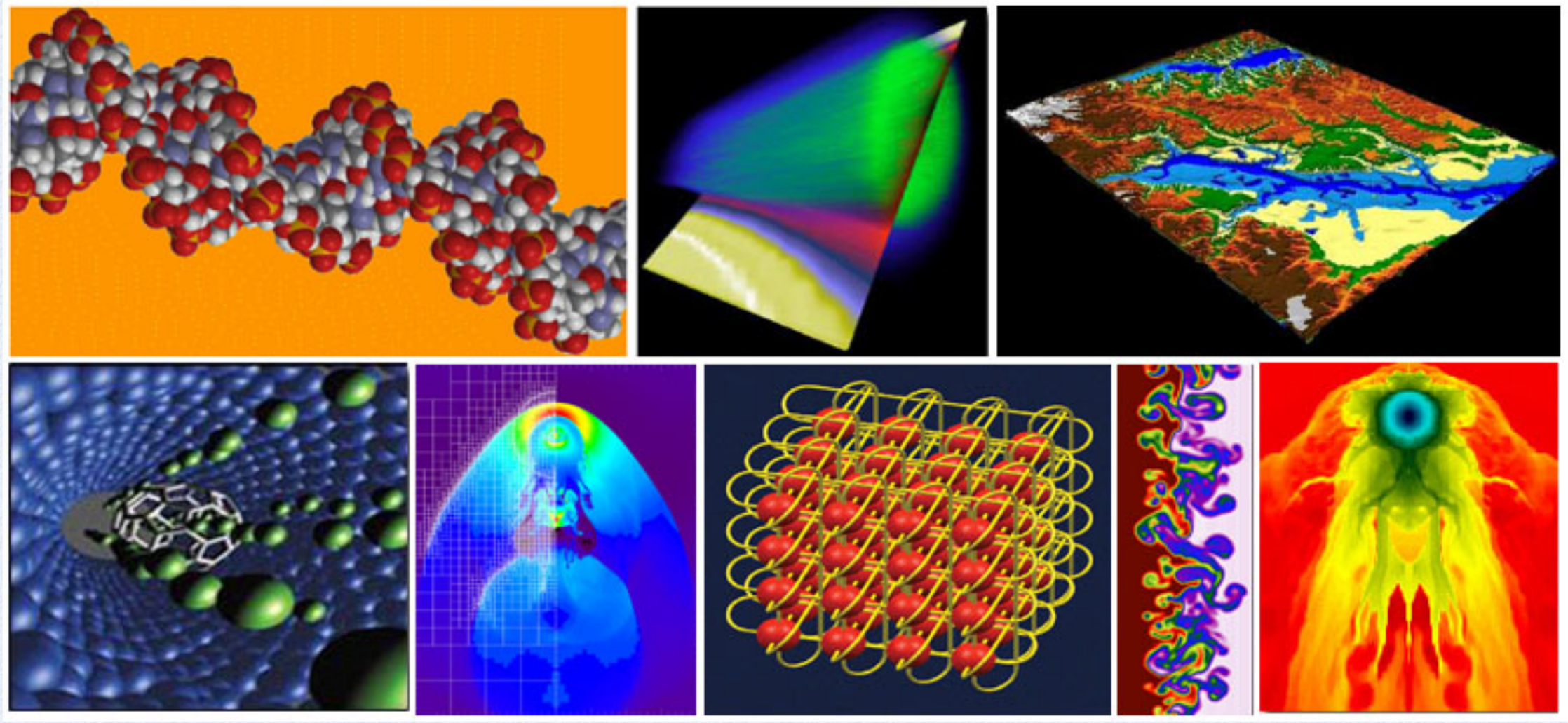
- Atmosphere, Earth, Environment
- Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Science



- Geology, Seismology
- Mechanical Engineering - from prosthetics to spacecraft
- Electrical Engineering, Circuit Design, Microelectronics
- Computer Science, Mathematics

TODAY : Parallel Computing
= Everyday (industry, academia) Computing

- Pharmaceutical design
- Medical imaging and diagnosis
- Oil exploration
- Product Design
- Databases, data mining
- Web search engines, web based business services

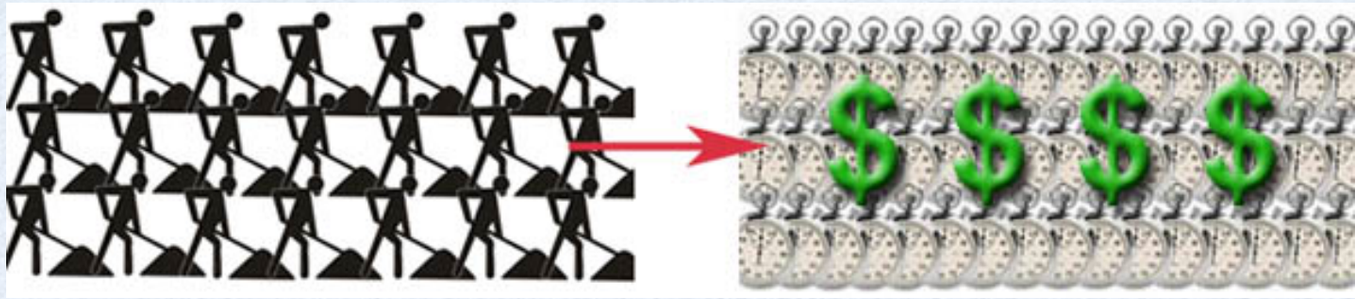


- Management of national and multi-national corporations
- Financial and economic modeling
- Advanced graphics and virtual reality(entertainment industry)
- Networked video and multi-media technologies
- Collaborative work environments

I. Why Parallel Computing ?

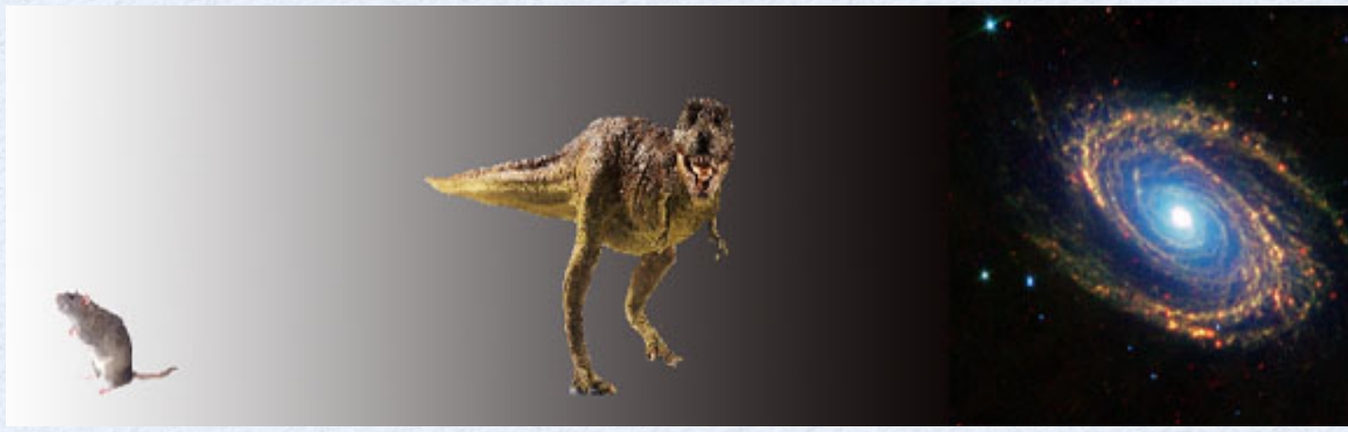
Save time and/or money:

Shorten Time to Completion - Parallel Computers built from commodity components



Solve Larger Problems

- "Grand Challenge" (en.wikipedia.org/wiki/Grand_Challenge) problems requiring PetaFLOPS and PetaBytes of computing resources.
- Web search engines/databases processing millions of transactions per second



Provide Concurrency

- A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.
- For example, the Access Grid (www.accessgrid.org) provides a global collaboration network where people from around the world can meet and conduct work "virtually".

Use Non-Local Resources

- SETI@home (setiathome.berkeley.edu) uses 2.9 million computers in 253 countries. (July 2011)
- Folding@home (folding.stanford.edu) uses over 450,000 cpus globally (July 2011)

II. Why Parallel Computing ?

Physical/Practical constraints for even faster serial computers:

Transmission speeds - the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.

Limits to miniaturization - processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.

- **Economic limitations** - it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

Energy Limits - limits imposed by cooling needs for chips and supercomputers.

- **Current computer architectures are increasingly relying upon hardware level parallelism to improve performance:**
  - **Multiple execution units**
  - **Pipelined instructions**
  - **Multi-core**

Economic Impact of HPC

- **Airlines:**
  - System-wide logistics optimization systems on parallel systems.
  - Savings: approx. $100 million per airline per year.
- **Automotive design:**
  - Major automotive companies use large systems (500+ CPUs) for:
    - CAD-CAM, crash testing, structural integrity and aerodynamics.
    - One company has 500+ CPU parallel system.
  - Savings: approx. $1 billion per company per year.
- **Semiconductor industry:**
  - Semiconductor firms use large systems (500+ CPUs) for device electronics simulation and logic validation
  - Savings: approx. $1 billion per company per year.
- **Energy**
  - Computational modeling improved performance of current nuclear power plants, equivalent to building two new power plants.

CREDIT: J. Demmel

HPC across Science/Technology

Can we find widely used patterns ?

Common patterns of communication and computation

1. Embedded Computing (EEMBC benchmark)
2. Desktop/Server Computing (SPEC2006)
3. Data Base / Text Mining Software
4. Games/Graphics/Vision
5. Machine Learning
6. **High Performance Computing (Original "7 Dwarfs")**

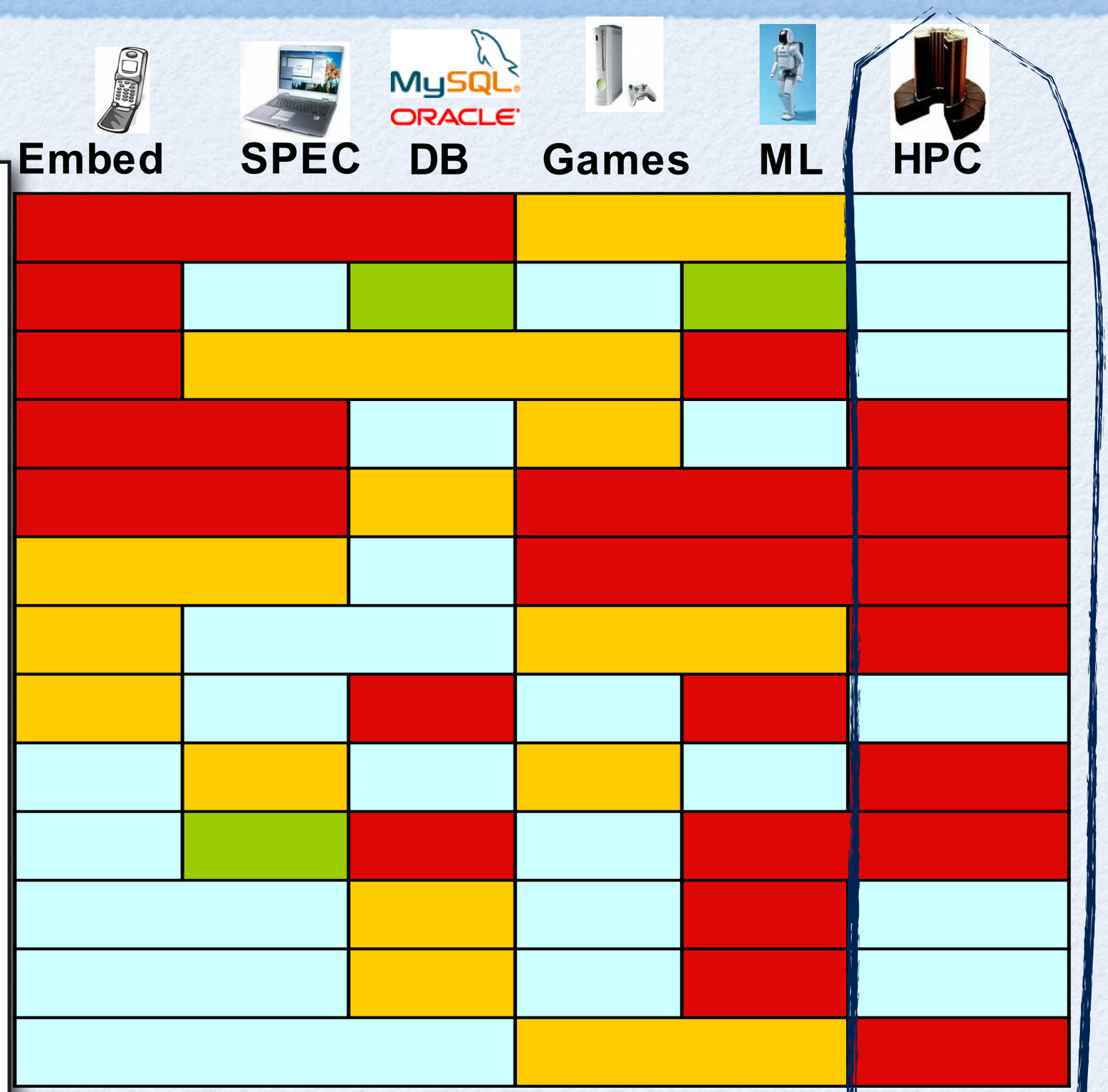
- Result: 13 "Dwarfs"

CREDIT: J. Demmel

Common Patterns = "Dwarfs" (Collela)

"Dwarfs"

1 Finite State Mach.
2 Combinational
3 Graph Traversal
4 Structured Grid
5. Dense Matrix
6. Sparse Matrix
7. Spectral (FFT)
8. Dynamic Prog
9. N-Body
10. MapReduce
11. Backtrack/ B&B
12. Graphical Models
13. Unstructured Grid



CREDIT: J. Demmel

Why writing fast parallel programs is hard

- Essential to know the hardware to get the best out of software
- KEY ISSUE: Understand in thsi cointext alternatives between algorithms

Principles of Parallel Computing

- Finding enough parallelism (Amdahl's Law)
- Granularity – how big should each parallel task be
- Locality – moving data costs more than arithmetic
- Load balance – don't want 1K processors to wait for one slow one
- Coordination and synchronization – sharing data safely
- Performance modeling/debugging/tuning

All of these things makes parallel programming even harder than sequential programming.

"Automatic" Parallelism in Modern Machines

- **Bit level parallelism**
  - within floating point operations, etc.
- **Instruction level parallelism (ILP)**
  - multiple instructions execute per clock cycle
- **Memory system parallelism**
  - overlap of memory operations with computation
- **OS parallelism**
  - multiple jobs run in parallel on commodity SMPs

Limits to all of these --

for very high performance, need user to identify, schedule and coordinate parallel tasks

Finding Enough Parallelism

- Suppose only part of an application seems parallel
- Amdahl's law
  - let **s** be the fraction of work done **sequentially**, so (1-s) is fraction parallelizable
  - **P** = number of processors

$$\text{Speedup(P)} = \frac{Time(1)}{Time(P)} = \frac{T(1)}{T(1) * s + \frac{1-s}{P} T(1)}$$

$$= \frac{1}{s + \frac{1-s}{P}} \sim \frac{1}{s}$$

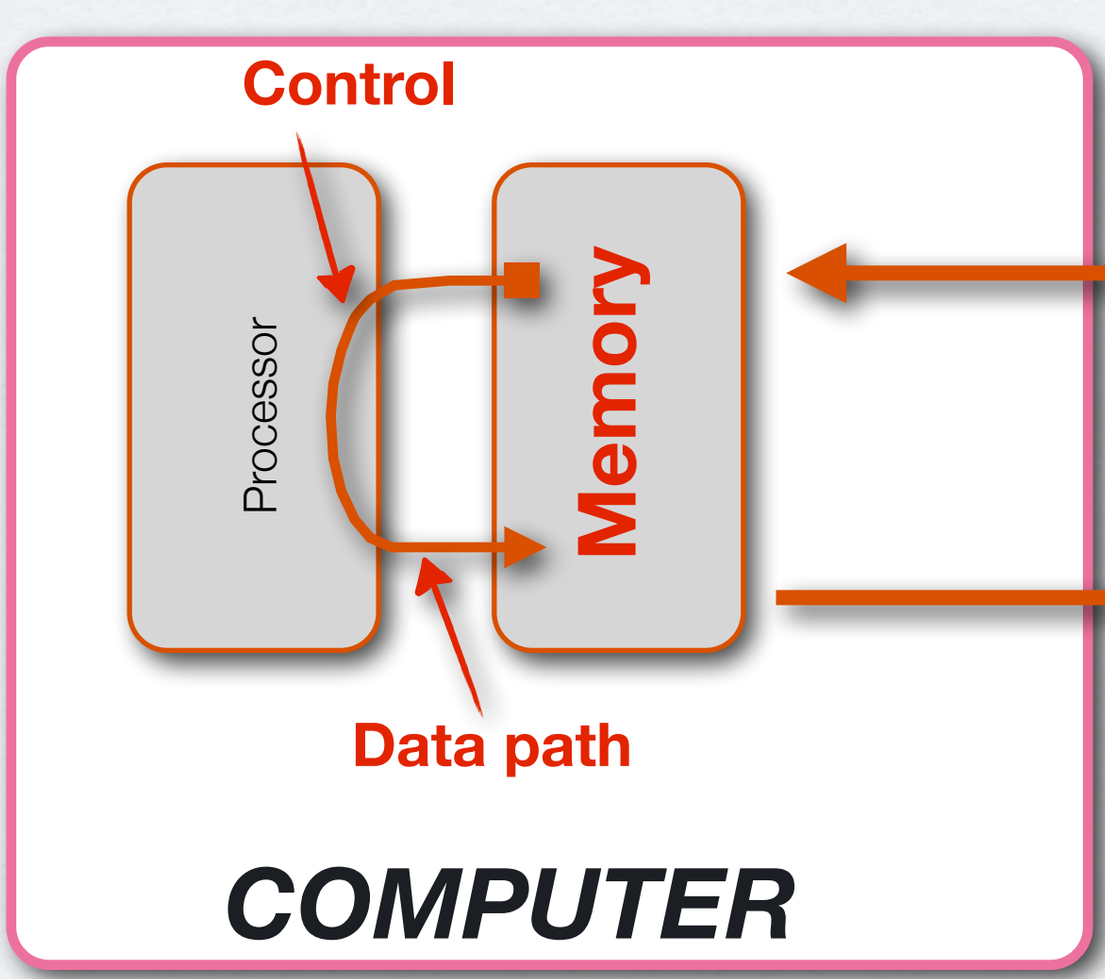
- Even if the parallel part speeds up perfectly performance is limited by the sequential part
- Top500 list: currently fastest machine has P~3.1M; 2nd fastest has ~560K

Computer: the BIG picture

SOFTWARE

Compiler

Performance evaluation



Input

Devices:

keyboard, mouse, etc.

Output

screen, printer, etc.

Computer: the BIG picture

Five classic components:

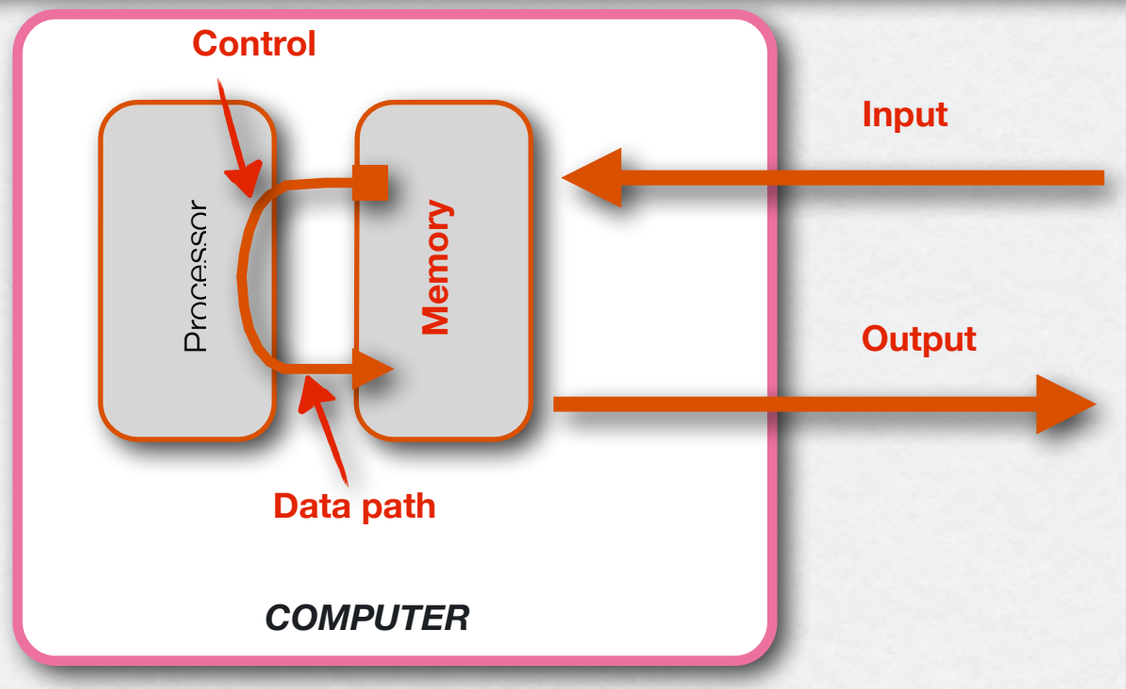
I.input

II.output

III.memory

IV.data path

V.control.



Memory: program and data storage → DRAM chips (some cost no matter what portion accessed)

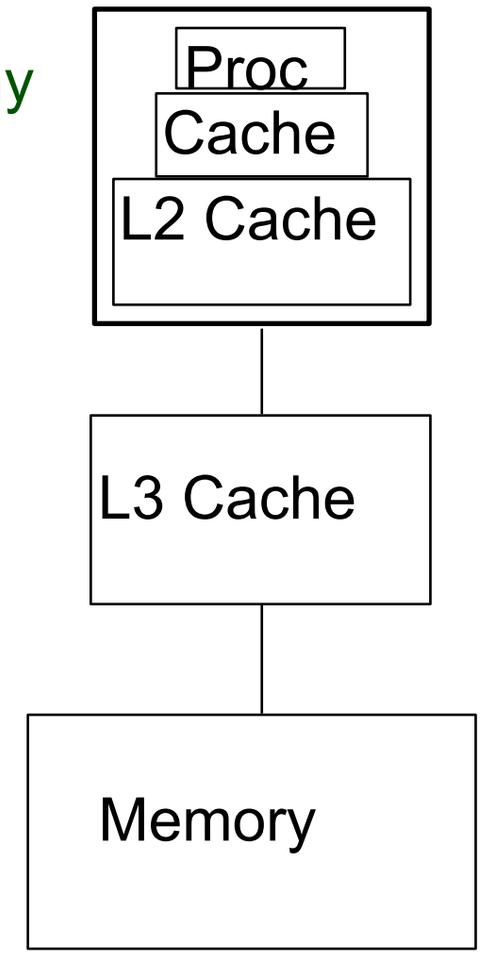
Processor: (iv) data path + (v) control + cache memory

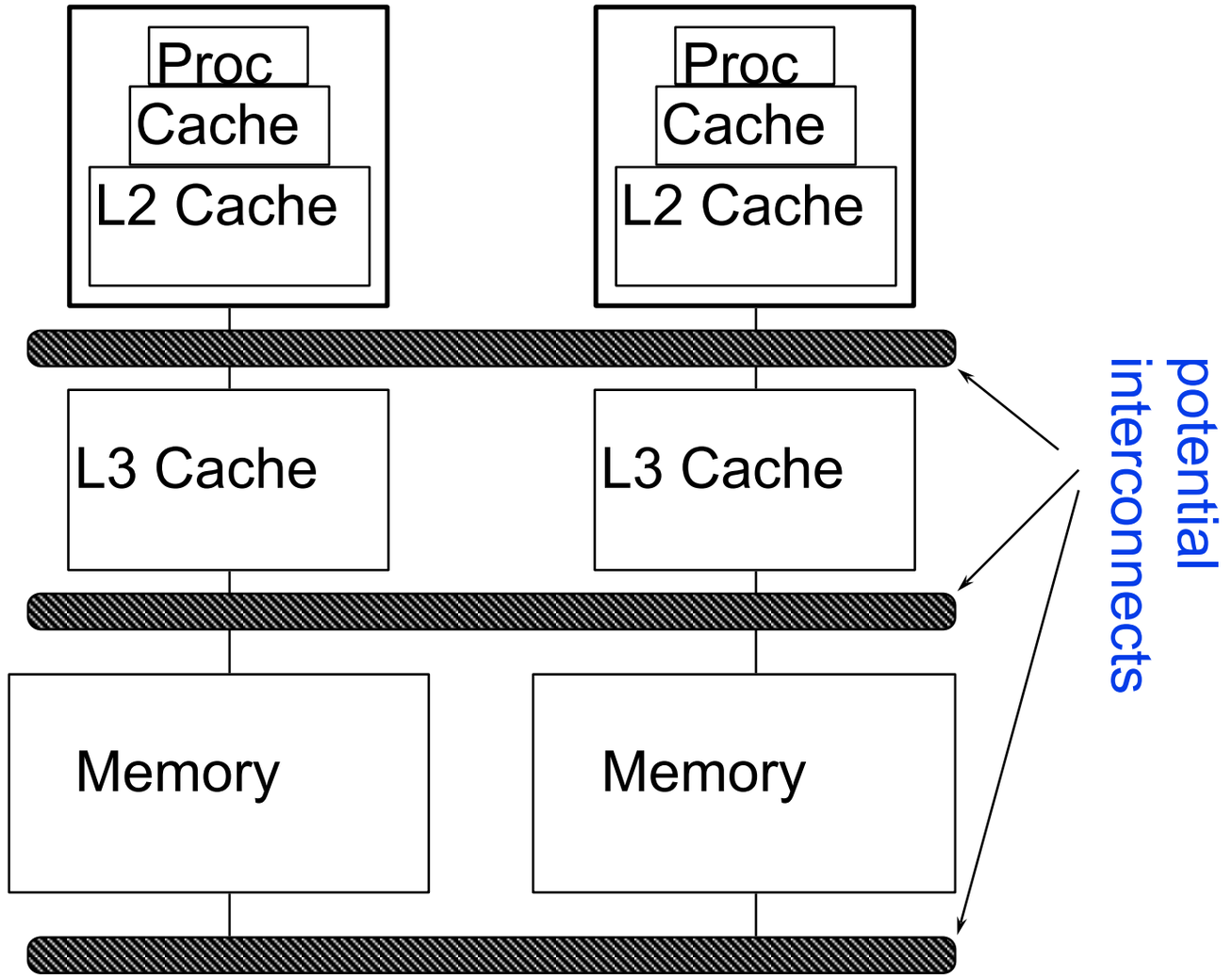
- (iv) data path: performs arithmetic operations
- (v) control: tells the data path memory and I/O what to do according to the program direction
- cache memory: buffer for DRAM memory

Communication between computers for: communicating information, resource sharing, nonlocal access

Locality and Parallelism

Conventional Storage Hierarchy



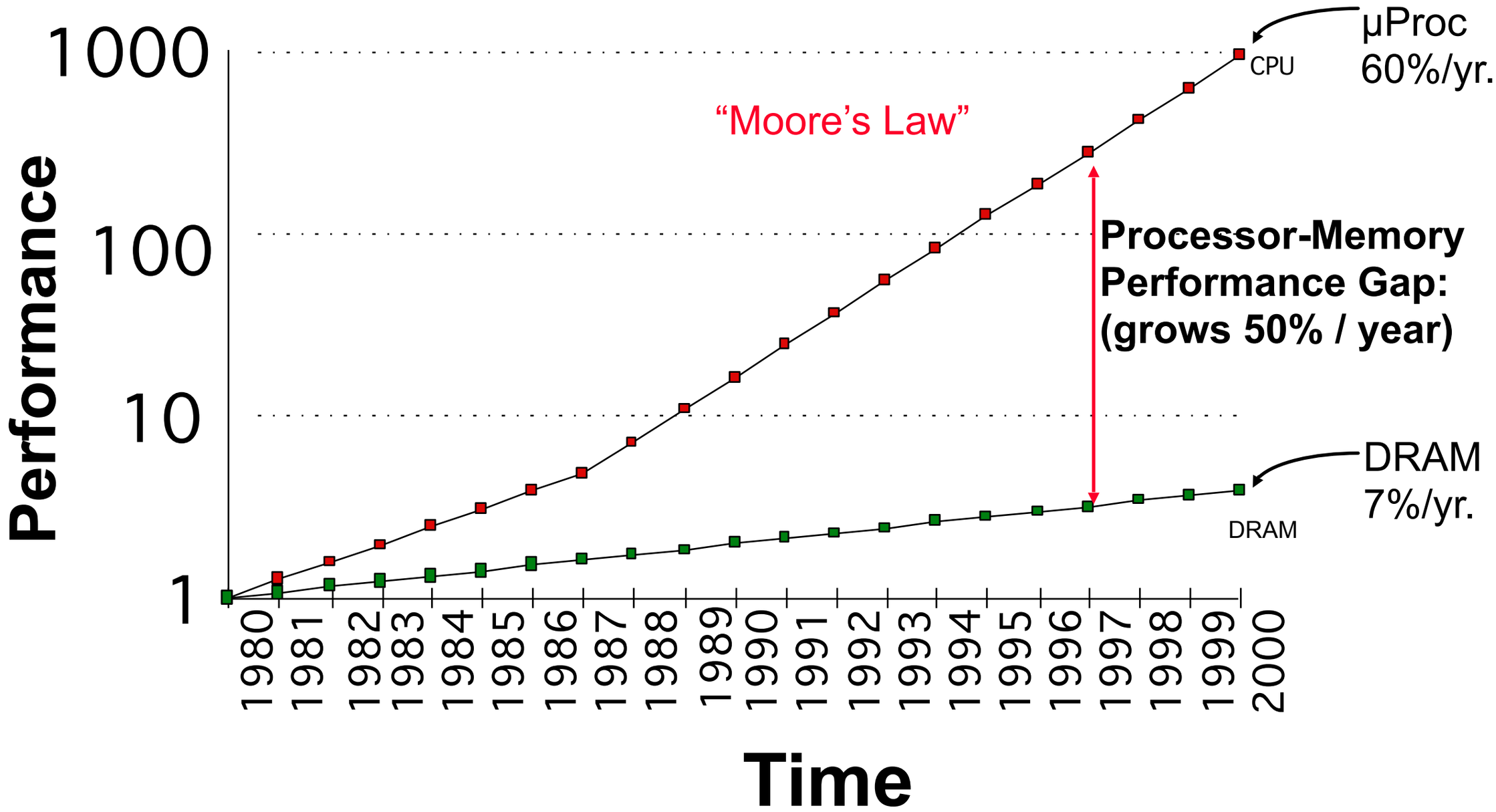


- Large memories are slow, fast memories are small
- Storage hierarchies are large and fast <u>on average</u>
- Parallel processors, collectively, have large, fast cache
  - the slow accesses to “remote” data we call “communication”
- Algorithm should do most work on local data

CREDIT: James Demmel

Processor-DRAM Gap (latency)

Goal: find algorithms that minimize communication, not necessarily arithmetic



CREDIT: James Demmel

LOAD IMBALANCE

- Load imbalance is the time that some processors in the system are idle due to:
  - insufficient parallelism (during that phase)
  - unequal size tasks
- Examples of the latter
  - adapting to “interesting parts of a domain”
  - tree-structured computations
  - fundamentally unstructured problems
- Algorithm needs to balance load
  - Sometimes can determine work load, divide up evenly, before starting
    - “Static Load Balancing”
  - Sometimes work load changes dynamically, need to rebalance dynamically
    - “Dynamic Load Balancing,” eg work-stealing

Parallel Software Eventually

- 2 types of programmers ➔ 2 layers of software
- **Efficiency Layer** (10% of programmers)
  - Expert programmers build Libraries implementing kernels, "Frameworks", OS, ...
  - Highest fraction of peak performance possible
- **Productivity Layer** (90% of programmers)
  - Domain experts / Non-expert programmers productively build parallel applications by composing frameworks & libraries
  - Hide as many details of machine, parallelism as possible
  - Willing to sacrifice some performance for productive programming
- Expect students may want to work at either level
  - In the meantime, we all need to understand enough of the efficiency layer to use parallelism effectively

(credit: J. Demmel)

THIS COURSE

9/19 - Introduction

9/26 - C++ Threads

10/3 - Diffusion: Finite Differences

10/10 - OPEN-MP

10/17 - OPEN-MP + Stochastics

10/24 - Elements of Architectures

10/31 - Performance-The Roofline Model

11/7 - MPI

11/14 - Vortex Dynamics: Particles

11/21 - Linked Lists and Particle-Mesh

11/28 - Diffusion: Stochastic Particles

12/5 - Diffusion/Reaction: Deterministic Particles

12/12 - MPIvs OpenMP - Review

12/19 - FINAL EXAM

Programming Environments

OpenMP

- Basic principles
- Work sharing
- Synchronization
- Threads
- Performance and optimization

MPI

- Point to point and collective communications
- Optimizations
- Topologies
- Memory to memory copies
- Parallel input-output

HPCSE II: MPI and OpenMP (Deeper Cuts) + GPUs

ADVANCED MPI
Derived types
Point to point communication modes and comparison with collective communications
Collecting computations via communications
Problems of load balancing
Placing of processes

ADVANCED Open/MP
- Fine synchronisation between 2 threads among « n » threads
- « Fine grain » approach: advantages and disadvantages for performance, extendability, ease of programming, etc...

HYBRID MPI/OpenMP
- Motivation
- Presentation of general concepts
- Hybrid programming model on SMP cluster
- Feedback, presentation of results
- Practical implementation
- Tools

GPUs
Massively Parallel Computing
Programming with CUDA

4 “Dwarfs”

- Stochastic Simulations
- Partial Differential Equations
- N-Body Problem
- Mesh to Particle to Mesh Interpolations

Collaboration

NO PLAGIARISM (one incident = CLASS FAIL)

- Submitting as your own work the work or parts thereof of another person (whether it is from a book, the web, a program library or ANY other source) constitutes academic misconduct.

- If the source is clearly cited then it is OK, but then the homework credit will be adjusted to compensate for original work.

- You are encouraged to DISCUSS the homework problems with other colleagues

BOOKS

Introduction to High Performance Computing for Scientists and Engineers, Georg Hager and Gerhard Wellein, CRC Press, 2011

- **READING MATERIAL**
  - See Class Web Site
  - Computer Organization and Design, D.H. Patterson, J.L. Hennessy
  - Introduction to Parallel Programming : http://www-users.cs.umn.edu/~karypis/parbook/
  - CUDA by Example, J. Sanders and E. Kandrot

Links

HPC tutorials at LLNL - Blaise Barney
https://computing.llnl.gov/?set=training&page=index

Hardware and Concepts
https://hpcforge.org/docman/view.php/67/134/EssentialHardware.pdf
https://hpcforge.org/docman/view.php/67/135/CoreConcepts.pdf

Parallel Program Design - Ian Foster
http://www.mcs.anl.gov/~itf/dbpp/

Class website

http://www.cse-lab.ethz.ch/index.php/teaching/42-teaching/classes/615-hpcse1

D-DAY

- **GRADING -** FINAL EXAM

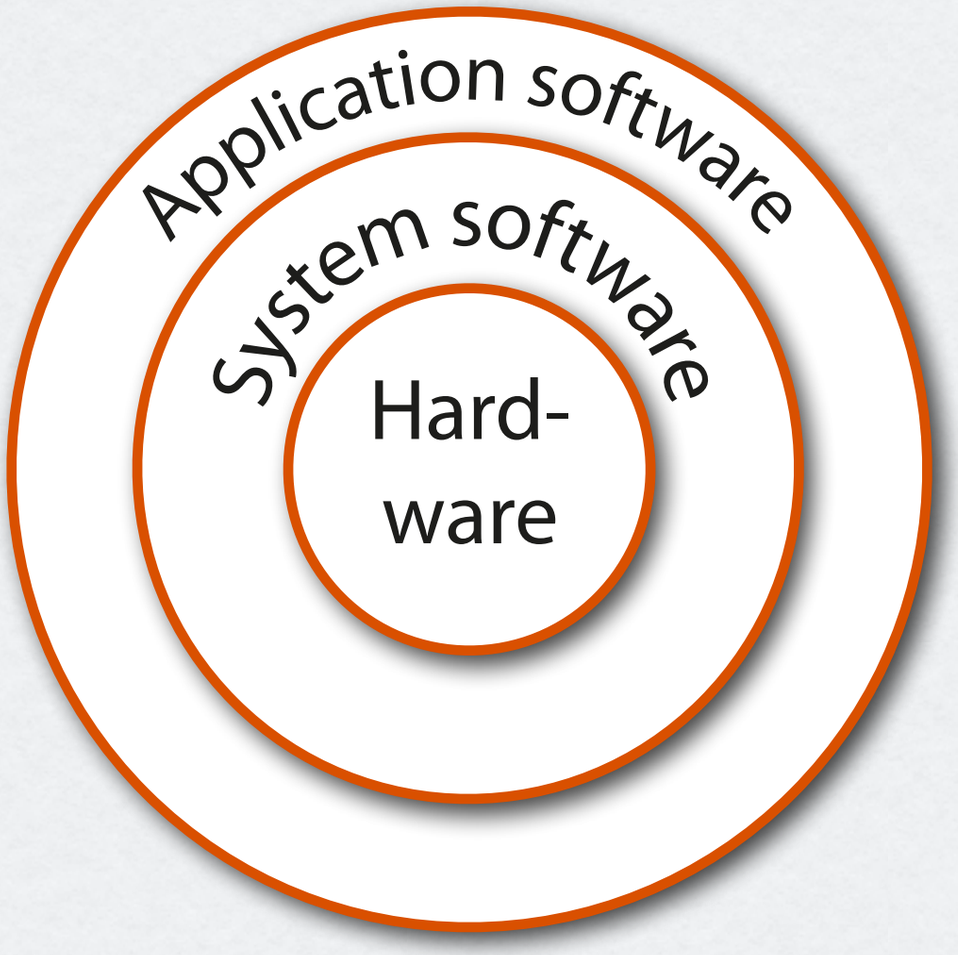
What YOU should get out of the course

- Understanding of computer hardware options from the HPC perspective
- When is parallel computing useful?
- Understanding of parallel computing hardware options
- Overview of OpenMP and MPI, and experience using them
- Elements of some important parallel applications and the algorithms
- Performance analysis and tuning
- Exposure to various open research questions

Hardware and software

- The **hardware** can execute extremely simple, low level instructions.

Complex application —multiple levels→ simple instructions



System Software

- *Operating System:*
  - I/O operations
  - Storage and memory
  - Protection among concurrent applications
- *Compilers*
  - Translate from a high level language to hardware instructions

High level language to hardware

- Computer understands “on” and “off”
- The symbols for these states are 0 and 1 and we commonly use a language based on binary numbers.
  We use numbers for instructions and data.
- First programmers communicated with 0 and 1. Later programs were developed to translate from symbolic notation to binary. The first was called assembly.
  E. g. a programmer writes: `add A, B` ← Assembly language

  assembly translates to: `1000110010100000` ← Machine language
- Advanced Languages are better than Assembly for:
  - programmer thinks in a more natural language
  - productivity of software development
  - portability

Performance

Response/Execution time: time between start and completion of a task

Throughput / Bandwidth: total amount of work done at a certain time

Performance and execution time:

$$Perf_x = \frac{1}{Exectime_x}$$

$$Perf_x > Perf_y \implies Exectime_y > Exectime_x$$

X is *n* times faster than Y to mean:

$$\frac{Perf_x}{Perf_y} = n = \frac{Exectime_y}{Exectime_x}$$

Time

Definitions of time

- **wall clock/response/elapsed:** total time including memory, CPU access, I/O, OS, everything
- **CPU execution time:** processor time spent on a task (system CPU/user CPU)
  - **user CPU time :** time spent in the program
  - **system CPU time :** time spent in the operating system performing tasks on behalf of the program.

CPU performance and its factors

$$\text{CPU exec. time} = \text{CPU}\ {}^{clock}_{cycles} \times \text{Clock cycle time}$$

$$= \frac{\text{CPU}\ {}^{clock}_{cycles}}{\text{Clock rate}}$$

Note: No reference to the number of instructions needed by the program

CPU performance and its factors

EXAMPLE

Program runs in 10s on a processor A which has a 2GHz clock rate. How to design a computer B to run this program in 6s?

Design consideration: Possible to increase in clock rate but then B will require 1.2 as many clock cycles as A for this program

$$\text{CPU}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_\text{A}} = \frac{\text{CPU clock cycles}_A}{2\text{GHz}} = 10\text{s}$$

$$\implies \text{CPU clock cycles}_A = 2 * 10^{10}\text{cycles.}$$

$$\text{CPU}_B = \frac{1.2\text{CPU clock cycles}_A}{\text{Clock rate}_\text{B}} = 6\text{s} \qquad \implies \frac{1.2 * 2 * 10^{10}}{6} = \text{Clock rate}_\text{B} = 4\text{GHz}$$

CPU performance and its factors

Compiler generates instructions to execute

Computer executes these instructions

Execution time depends on number of instructions

Number of clock cycles to run a program:

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \text{Average Clock Cycles Per Instruction (CPI)}$$

provides a way to compare different architectures

Note : Different arithmetic operations may have different numbers of instructions (e.g. x vs /)

CPU performance equation

$$\text{CPU time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock cycle time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock rate}}$$

Includes the 3 factors that affect performance.

Performance is measured in time:

$$\text{Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instructions}} \times \frac{\text{Seconds}}{\text{Clock cycles}}$$

Program performance