CPSC 4240/5240: Parallel Programming Techniques

Lecture 1: Introduction

Lecturer: Quanquan C. Liu

Spring 2026

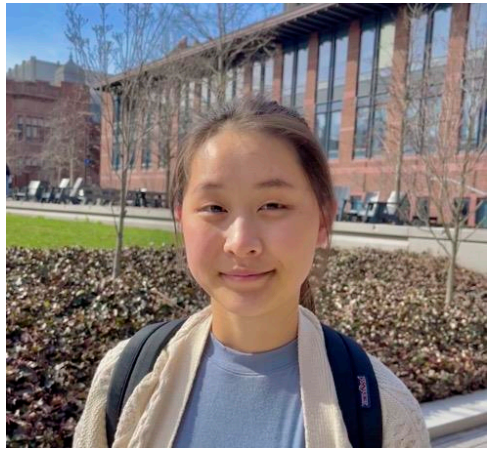
Special thanks to **MIT 6.106** and the FASTCODE Community for this part of this lecture's content!

CPSC 424 Staff

TF:

Pranay Mundra

ULAs:



Jewon Im

Bill Qian

Vishak Srikanth

Course Materials

- All course announcements and assignments will be on **Canvas**
- First homework will be released **this Thursday** (after class) and due on **Thursday, Jan. 22nd at 11:59pm EST**
  - Do this **homework ASAP**
  - Helps with setup for the rest of the course
- Syllabus posted under resources
- See announcements for office hours
  - Prof. Liu's office hours: **Wednesdays 3-4pm**

Course Structure

- Half theory and half practice
- Elements of parallel algorithms, programming, and software performance engineering
- Geared to help you write **faster** and **more performant programs**!
  - Requires both algorithms and coding skills
- Today's data is **massive** and we need new techniques to be able to analyze it

Course Structure

- Find the following information in the syllabus
  - 30% Programming Assignments Gradescope (biweekly)
  - 25% Written Homework Assignments (biweekly)
  - 5% Performance against the entire class/participation
  - 10% Midterm 1 (Feb. 10; multiple choice)
  - 10% Midterm 2 (March 3; multiple choice)
  - 20% Final Project (1-3 students; March 31 - April 23)

5% for Performance or Participation

- In real-life, **performance** can have a large impact on your company's well-being
- In parallel programming or performance engineering research, you are **comparing your code with the state-of-the-art code**
- Both require comparing your code against other people's code!
- We will simulate this aspect of performance engineering in this class

5% for Performance or Participation

- For 5% of your grade, you can choose to either have the 5% be **participation** grade:
  - Come to lecture
  - Answer questions on Ed Discussion
- Or participate in our performance engineering challenge!
  - We will run your code and compare against everyone else's code
  - **100%** if your code is within **10%** of the fastest runtime (or is the fastest)

5% for Performance or Participation

- Or participate in our performance engineering challenge!
  - We will run your code and compare against everyone else's code
  - **100%** if your code is within **10%** of the fastest runtime (or is the fastest)
    - **95%** if your code is within **20%** of the fastest runtime
    - **90%** if your code is within **50%** of the fastest runtime
    - **85%** if your code is within **70%** of the fastest runtime
    - **80%** if your code is within **2x** of the fastest runtime
    - **70%** otherwise

Parallel Programming and Software Performance Engineering

Why do we care?

Is Performance Important?

- What properties should software have that are important besides performance?
  - Correctness
  - Functionality
  - Security
  - What else?

Is Performance Important?

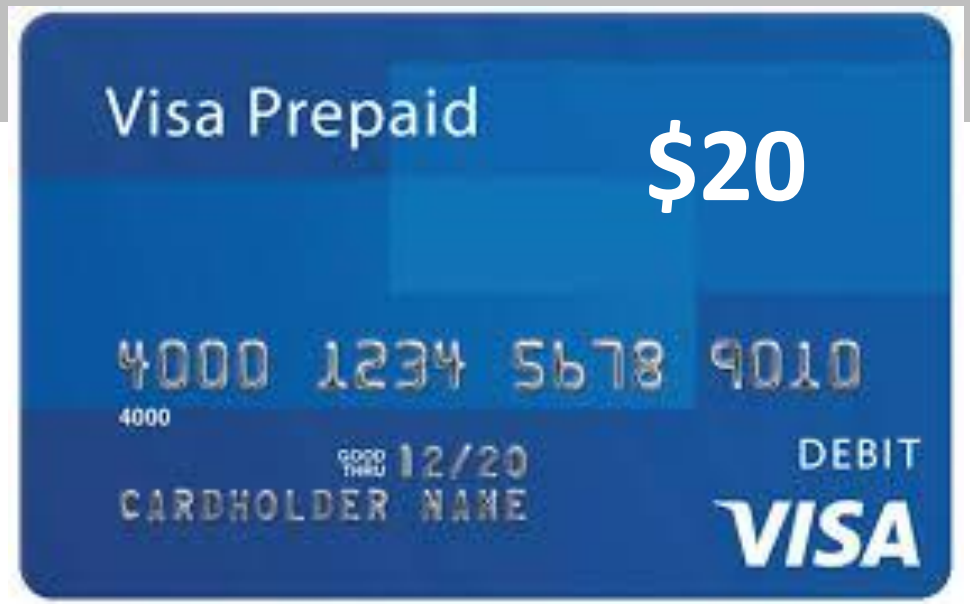
What properties should software have that are important besides performance?

- Correctness
- Functionality
- Security
- Compatibility
- Clarity
- Debuggability
- Functionality
- Maintainability
- Modularity
- Portability
- Reliability
- Robustness
- Security
- Usability

Why study performance when so many other things are also important?

Which would you choose?

or



Object 1

Object 2

Which would you choose?

<

Object 1

Object 2

Which would you choose?

Performance is like money!

Is Performance Important?

What properties should software have that are important besides performance?

- Correctness
- Functionality
- Security
- Compatibility
- Clarity
- Debuggability
- Functionality
- Maintainability
- Modularity
- Portability
- Reliability
- Robustness
- Security
- Usability

Why study performance when so many other things are also important?

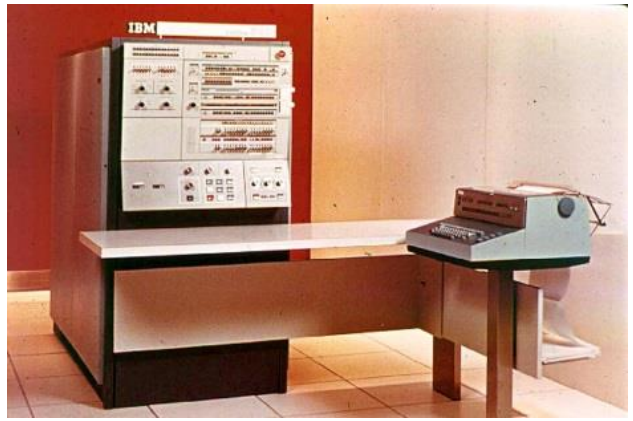
Performance is the **currency** for computing. You can **buy needed properties** with performance.

A Very Brief History

Of how we are here today…

Computer Programming in the Early Days

IBM System/360



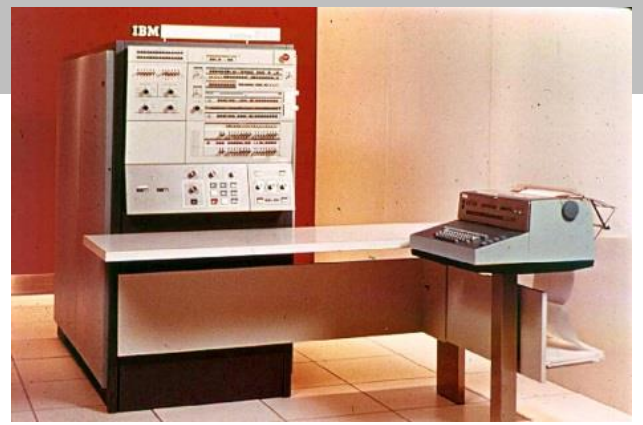
DEC PDP-11

Apple II

Machines had limited memory and compute power so performance was a big deal!

Computer Programming in the Early Days

IBM System/360



DEC PDP-11

Apple II

- Memory was measured in **kilobytes (or even bytes!)**
- **Processing power was limited**, requiring efficient use of every CPU cycle
- Programs written in **low-level languages** (like assembly) to squeeze out performance

Moore's Law and the Evolution of Computing Performance

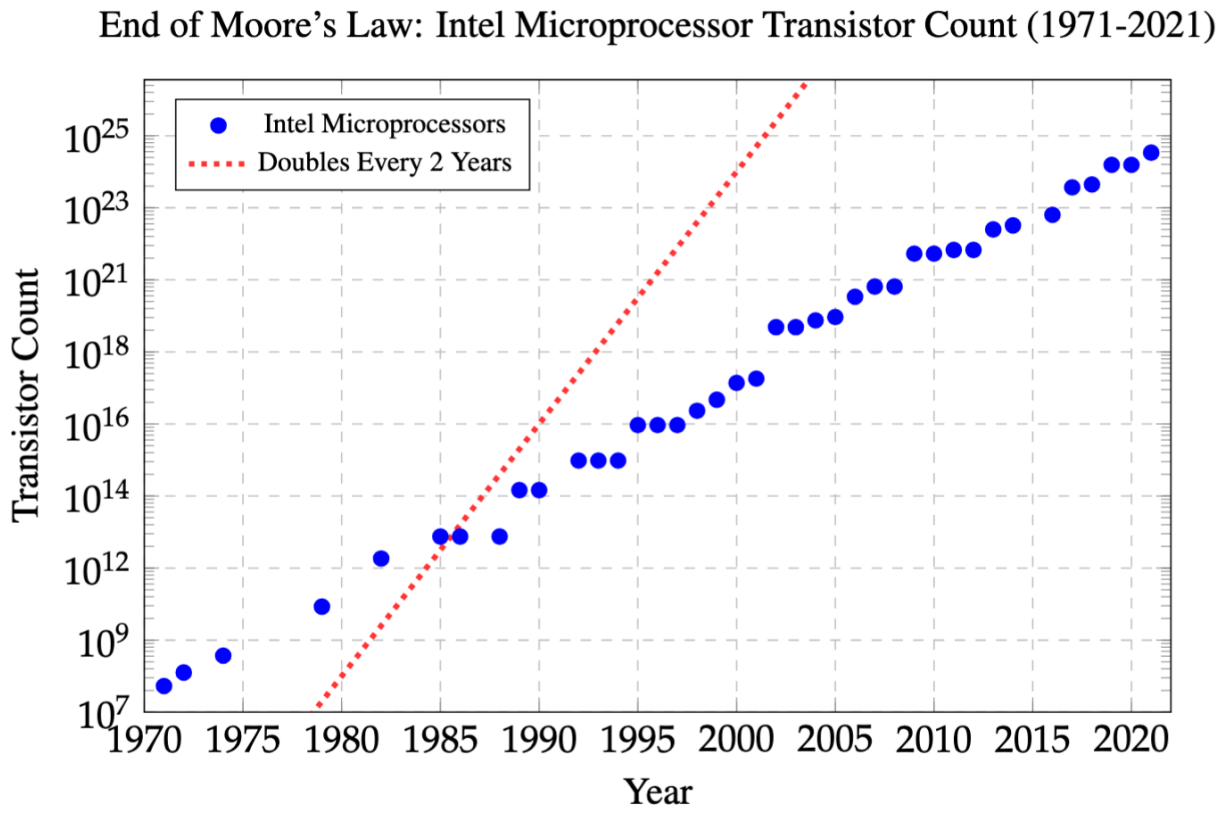
- Gordon E. Moore (co-founder of Intel) 1965 paper observing **number of transistors** on integrated circuit (roughly) doubles **every 18-24 months**
- More transistors means more:
  - Processing power: faster clock speeds
  - Memory capacity: larger and cheaper memory
  - Decreasing cost per transistor: more advanced engineering technology

Moore's Law and the Evolution of Computing Performance

- **Additional impacts:**
  - **Performance**: CPUs becoming exponentially faster
  - **Resource abundance**: less resource constraints enabling richer software with higher-level abstractions
  - **Shift in focus**: However, single-core clock speed has essentially plateaued (due to power and heat limitations)

Moore's Law (1965-2000s) has ended!

Moore's Law and the Evolution of Computing Performance

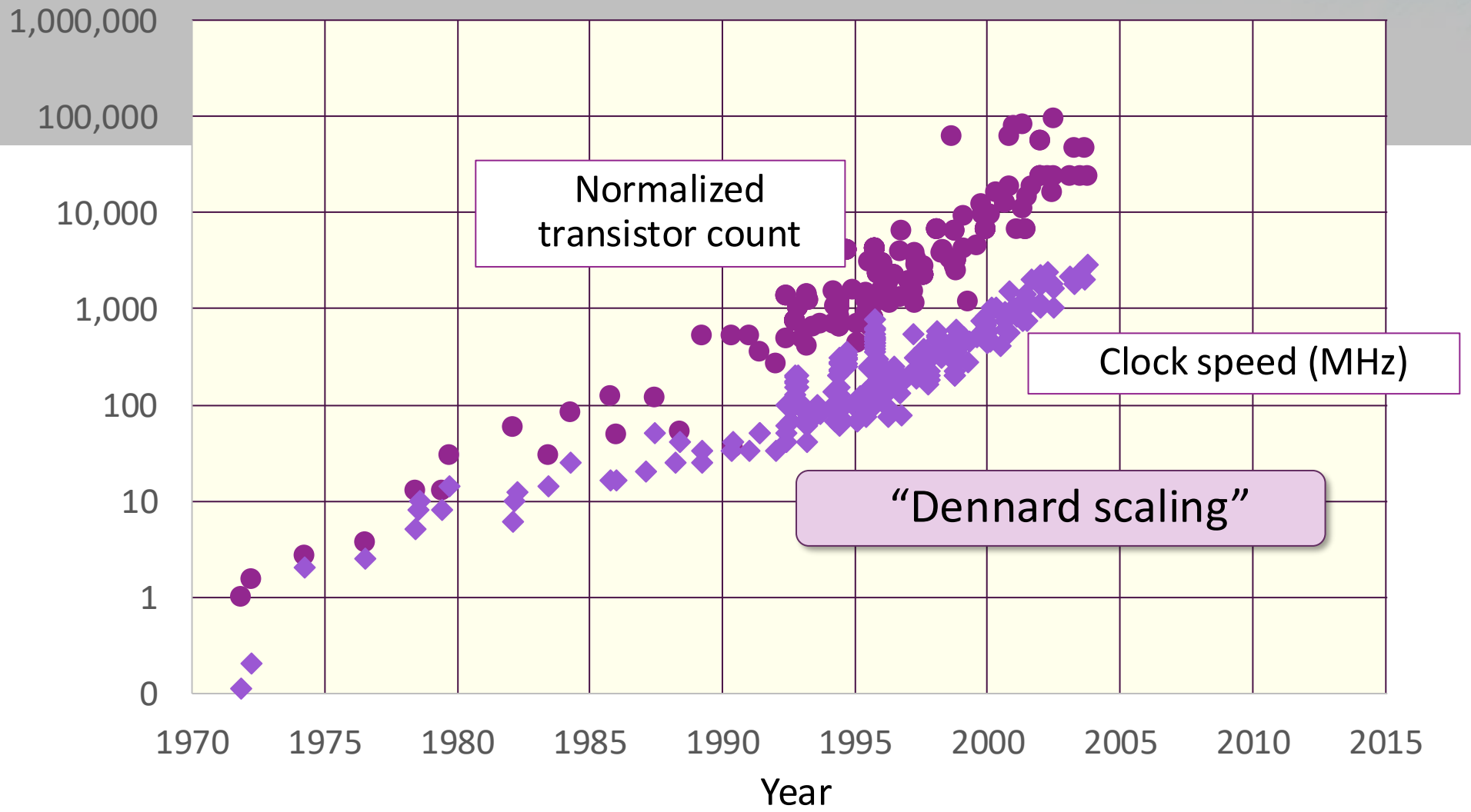


Moore's Law (1965-2000s) has ended!

Moore's Law and Dennard Scaling

- **Dennard Scaling (1971–2004):**
  - As transistors shrank, **power density stayed constant**
  - Enabled **increasing clock frequencies** generation after generation
    - Miniaturizing transistors and other integrated-circuit components
    - **Clock speed could be increased without substantially increasing power**
  - Clock frequency increase annual **rate of 25% to 30%**

Dennard Scaling (1971-2004)



Processor data from Stanford’s CPU DB [DKM12]. Slide courtesy of MIT 6.106.

Dennard Scaling (1971-2004)

Similarly priced Apple computers from 1977 to 2004

Apple II

Launched:	1977
Clock rate:	1 MHz
Data path:	8 bits
Memory:	48 KB
Cost:	$1,395

Power Macintosh G4

Launched:	2000
Clock rate:	400 MHz
Data path:	32 bits
Memory:	64 MB
Cost:	$1,599

Power Macintosh G5

Launched:	2004
Clock rate:	1.8 GHz
Data path:	64 bits
Memory:	256 MB
Cost:	$1,499

Slide courtesy of MIT 6.106.

Dennard Scaling (1971-2004)

Similarly priced Apple computers from 1977 to 2004

Revolution in Hardware Performance!

Appl					sh G5
Launched:					004
Clock rate:					.8 GHz
Data path:					4 bits
Memory:					56 MB
Cost:	$1,395	Cost:	$1,599	Cost:	$1,499

Slide courtesy of MIT 6.106.

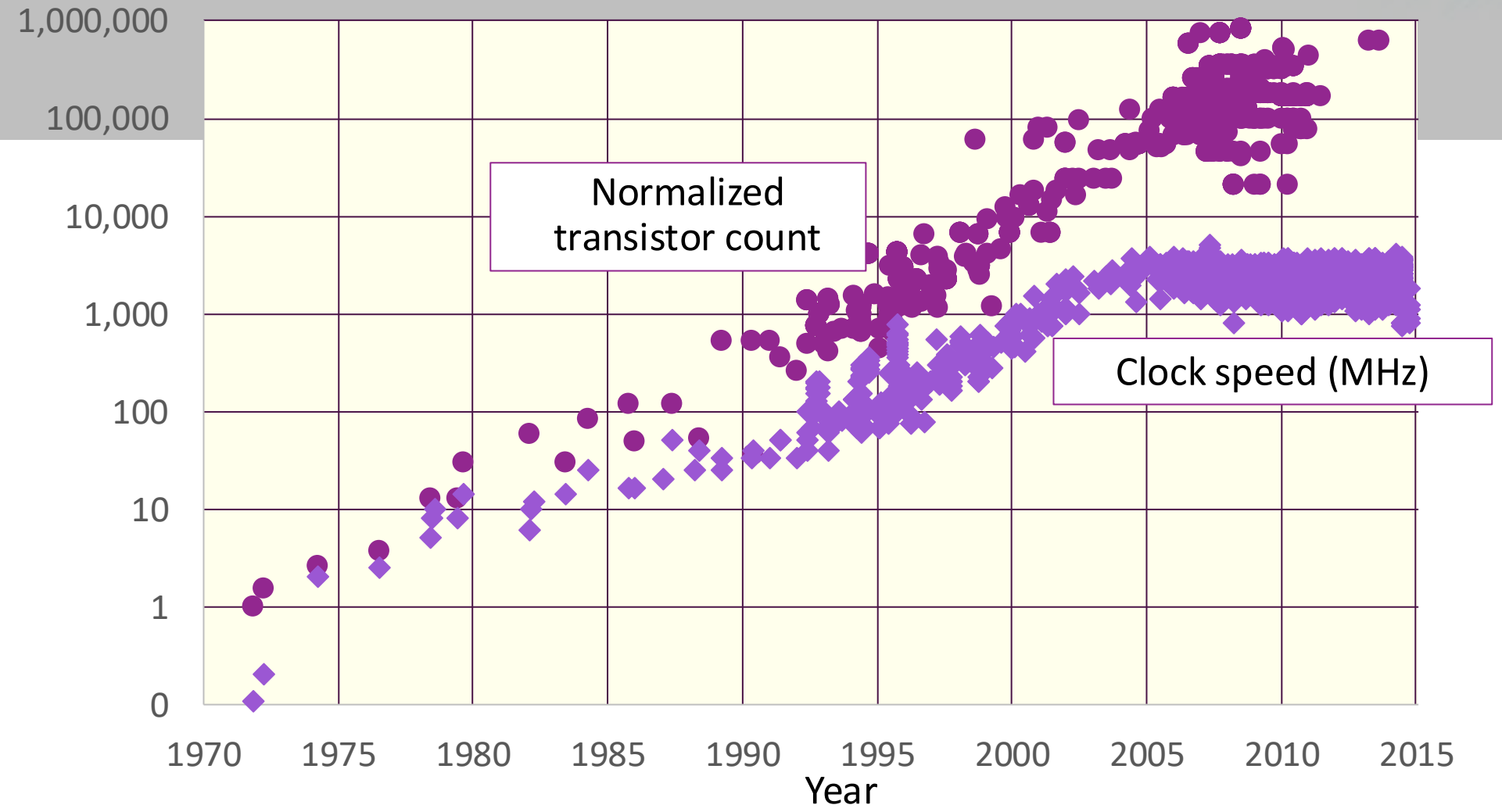
Until 2004

Moore's Law and Dennard Scaling

= printing press for the currency of performance.

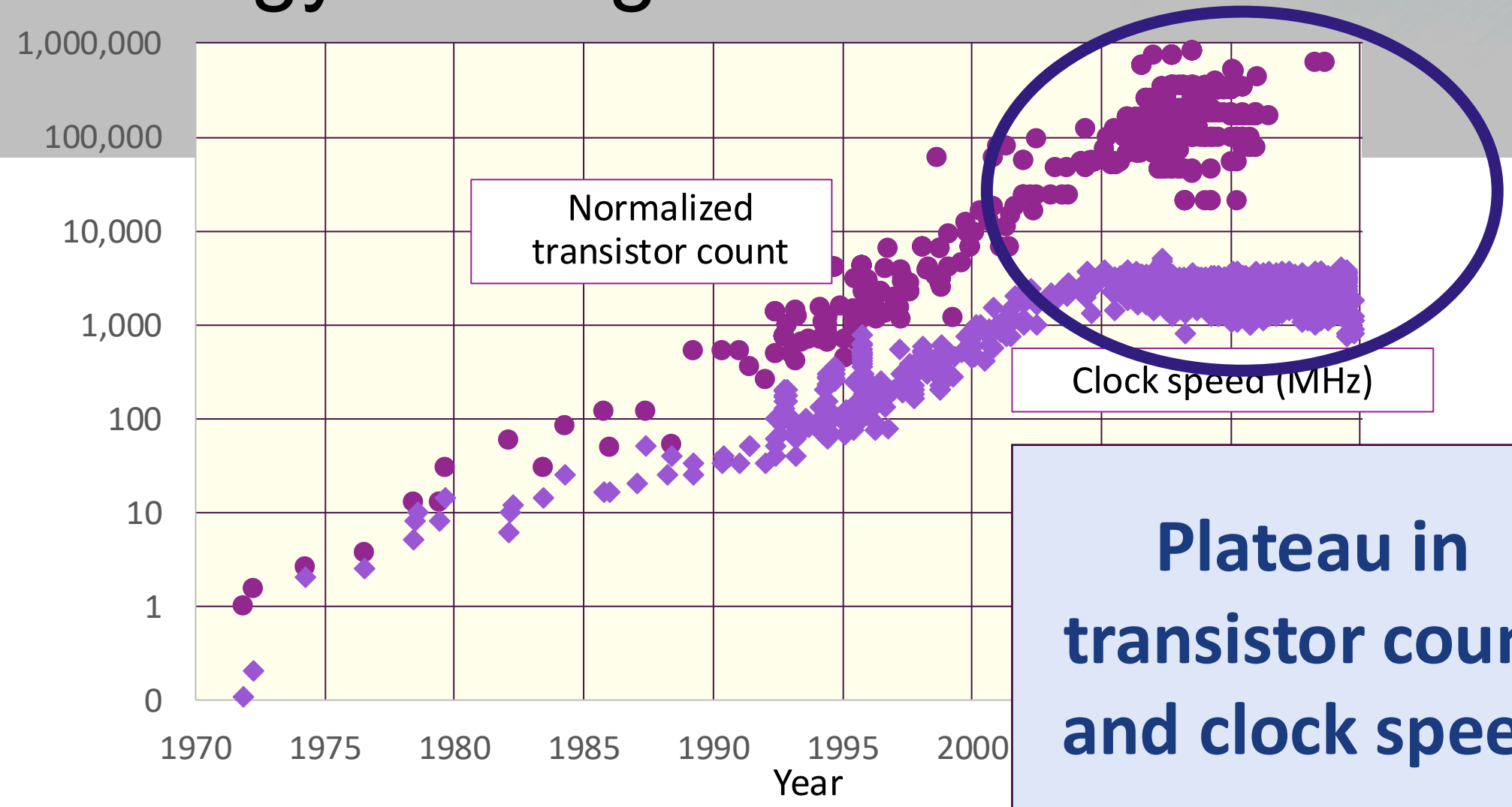
Slide courtesy of MIT 6.106.

Technology Scaling After 2004



Processor data from Stanford's CPU DB [DKM12]. Slide courtesy of MIT 6.106.

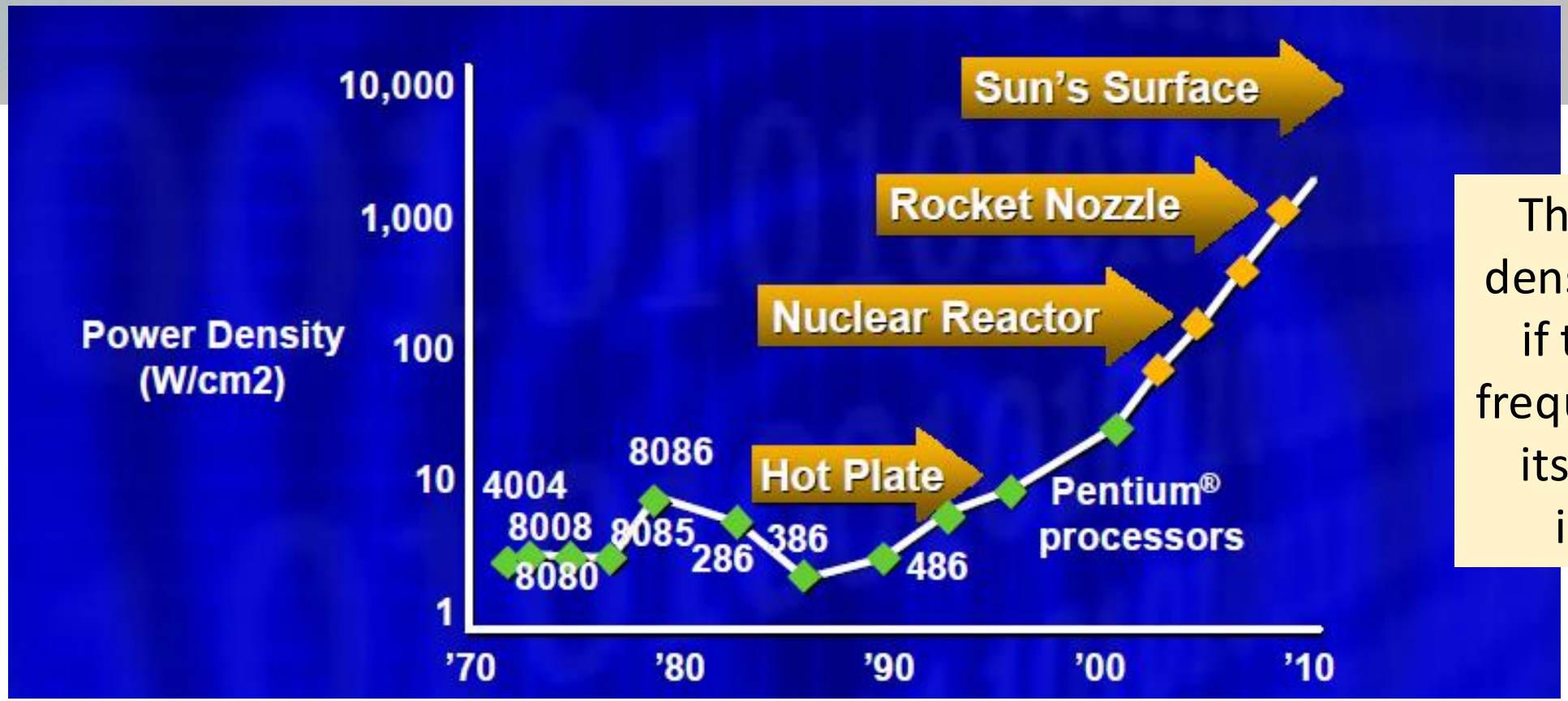
Technology Scaling After 2004



Plateau in transistor count and clock speed

Processor data from Stanford's CPU DB [DKM12]. Slide courtesy of MIT 6.106.

Power Density

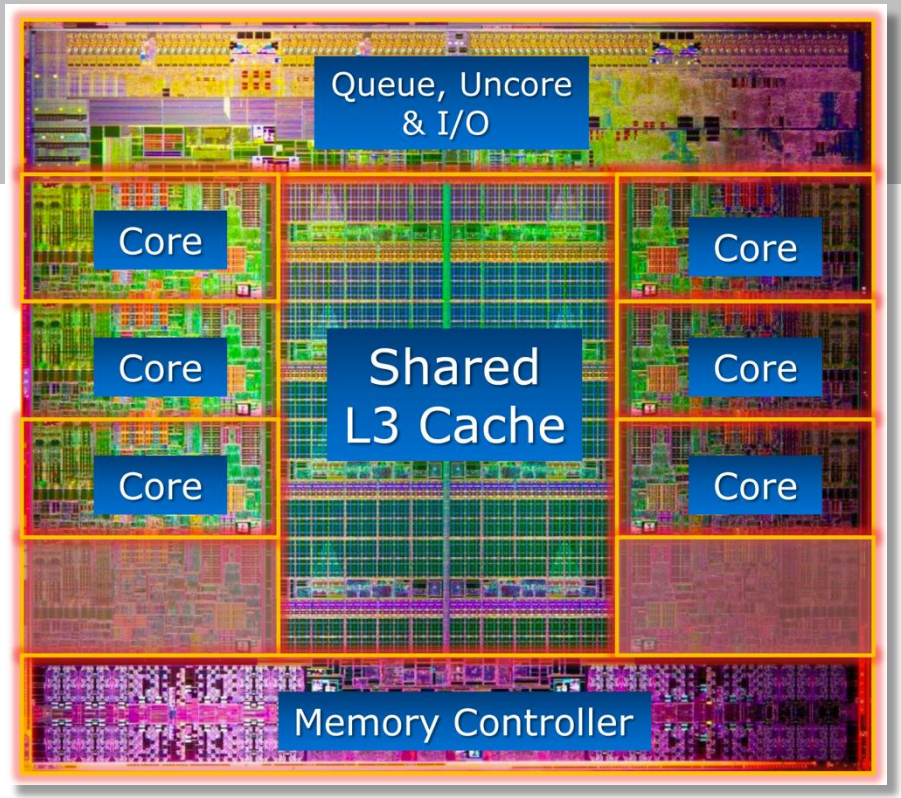


The growth of power density, as seen in 2004, if the scaling of clock frequency had continued its trend of **25%-30%** increase per year.

Source: Patrick Gelsinger, *Intel Developer's Forum*, Intel Corporation, 2004.

Slide courtesy of MIT 6.106.

Vendor Solution: Multicore Chips!



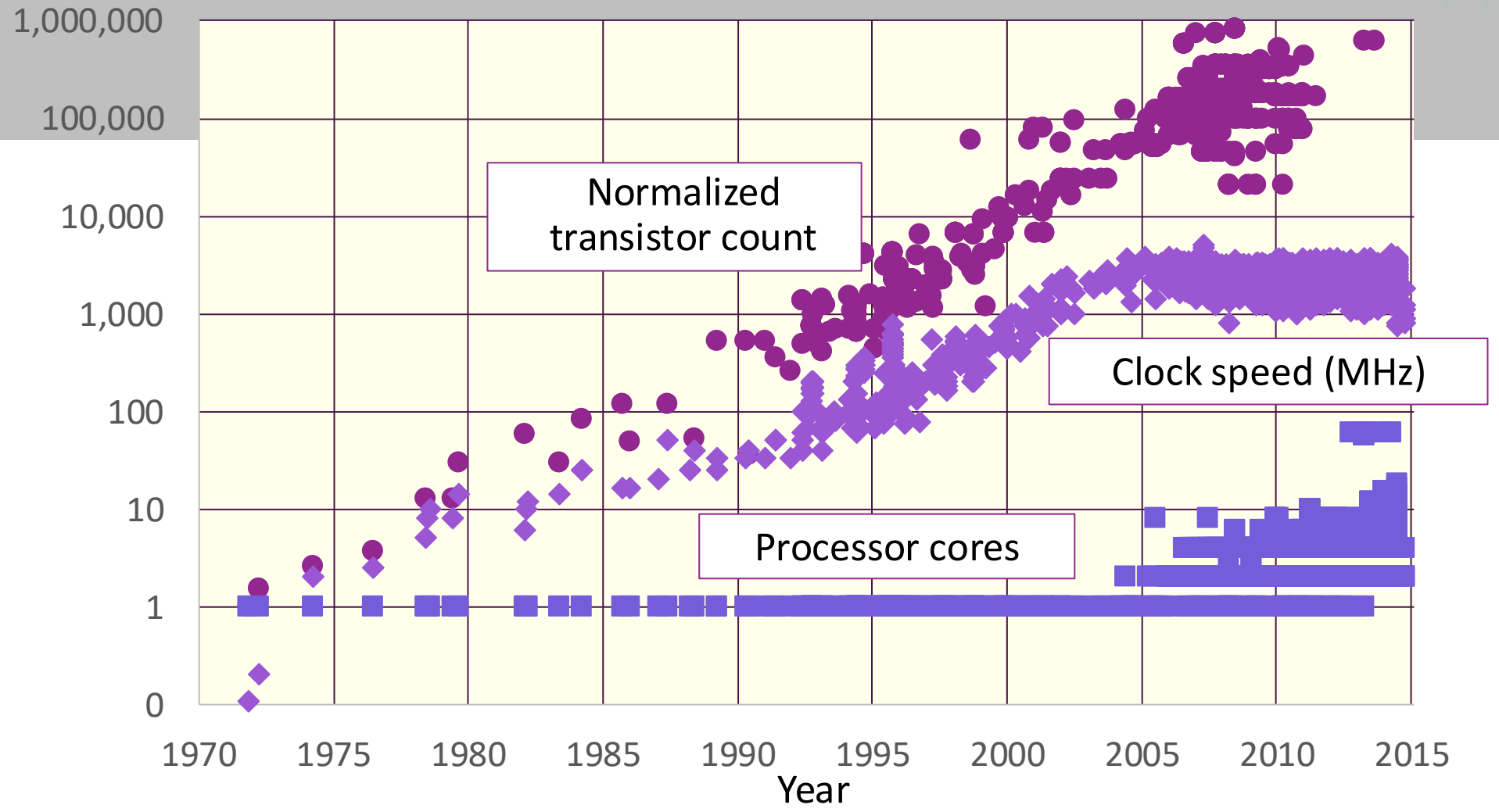
Intel Core i7 3960X (Sandy Bridge E), 2011

- 6 cores
- 3.3 GHz
- 15-MB L3 cache

To scale performance, processor manufacturers put **many processing cores** on the same microprocessor chip

Slide courtesy of MIT 6.106.

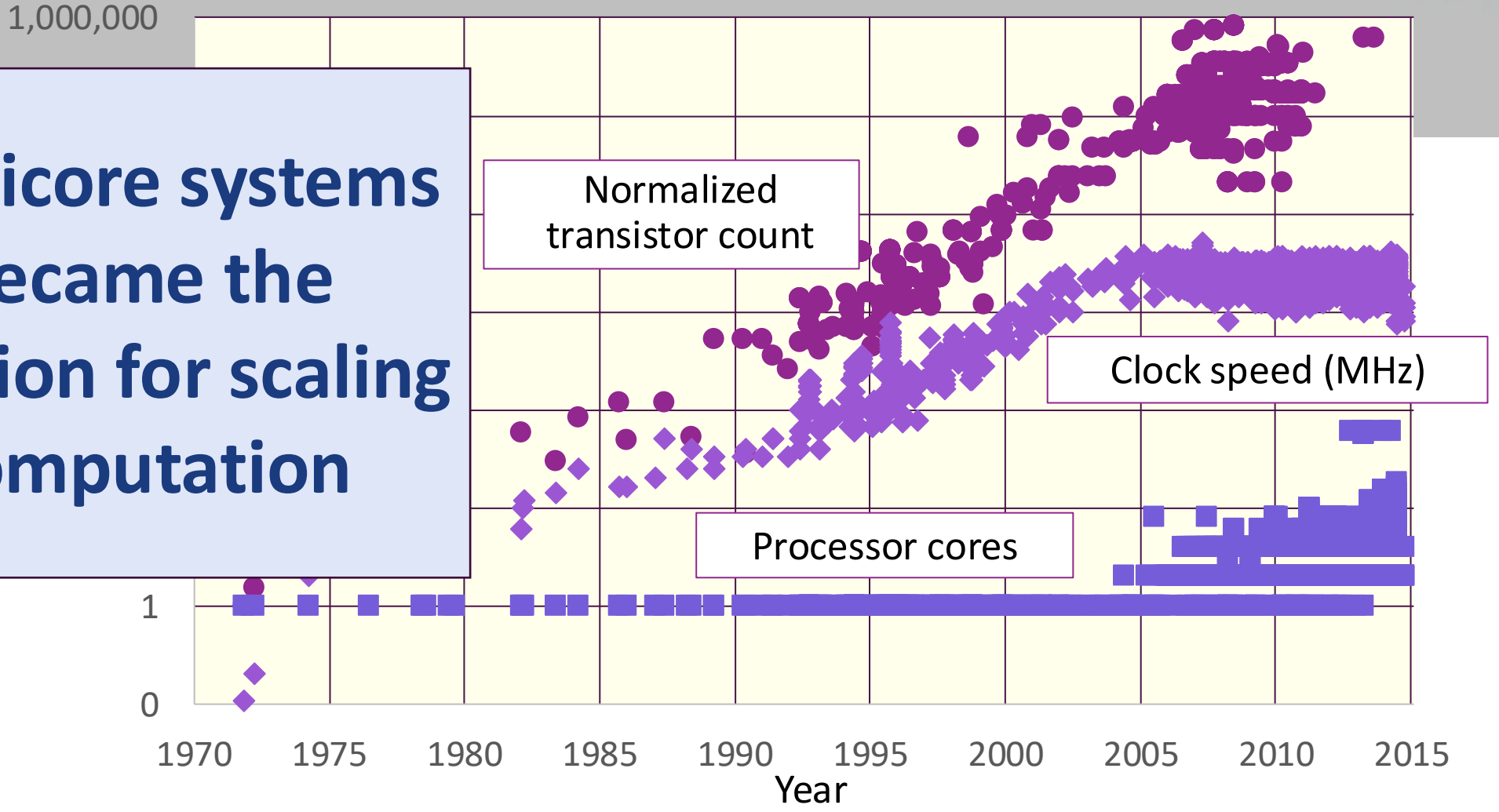
Technology Scaling



Processor data from Stanford's CPU DB [DKM12]. Slide courtesy of MIT 6.106.

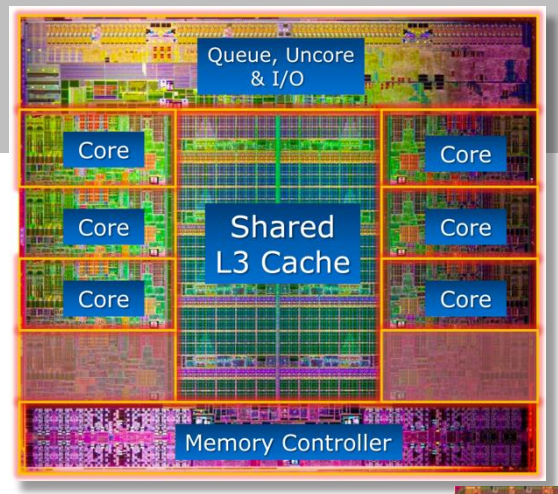
Technology Scaling

Multicore systems became the solution for scaling computation

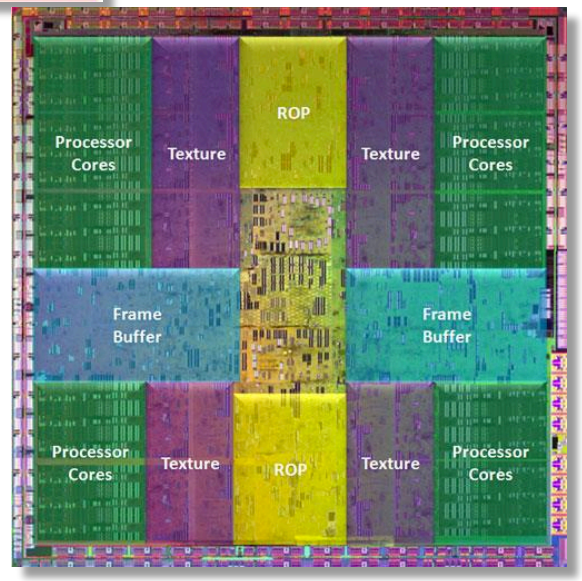


Processor data from Stanford's CPU DB [DKM12]. Slide courtesy of MIT 6.106.

Performance Is No Longer Free



2011 Intel Skylake processor

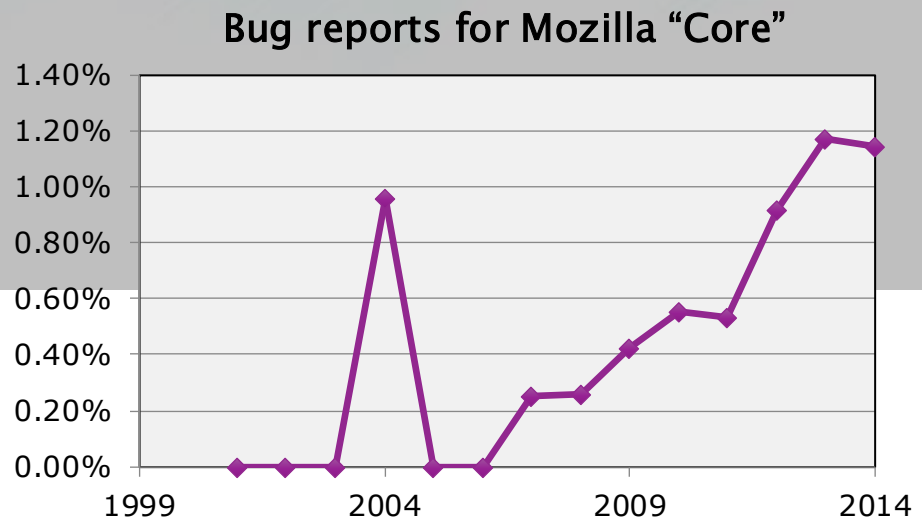


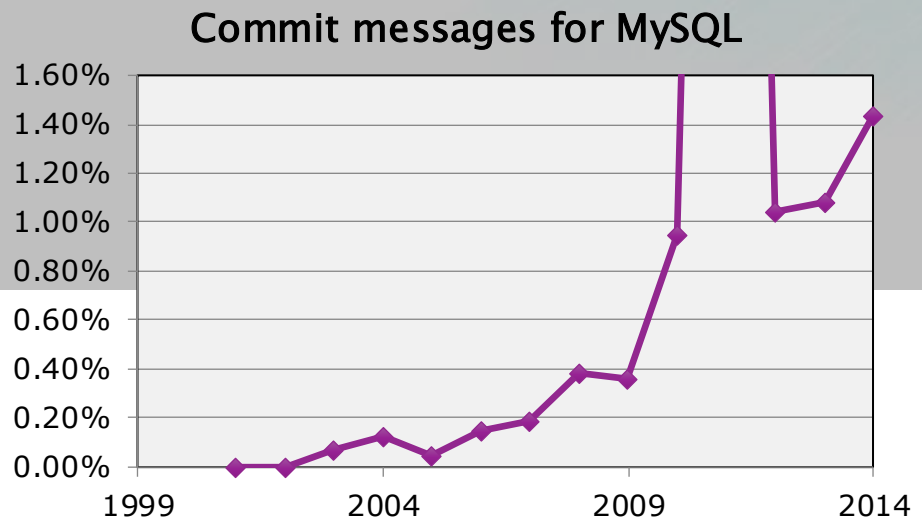
2008 NVIDIA GT200 GPU

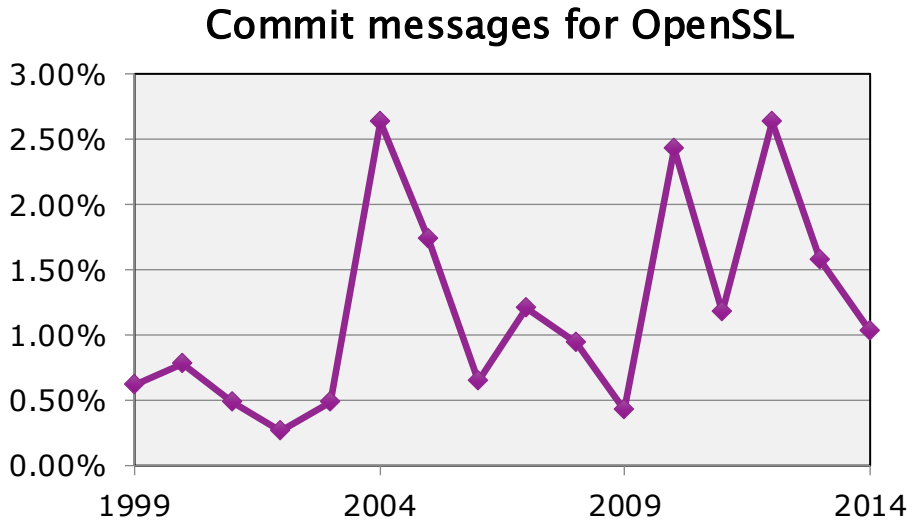
- Performance was available in the form of multicore processors with complex cache hierarchies, wide vector units, GPU's, FPGA's, etc.
- Generally, software must be **rewritten** to utilize this hardware efficiently!
- Need to learn **parallel programming**!

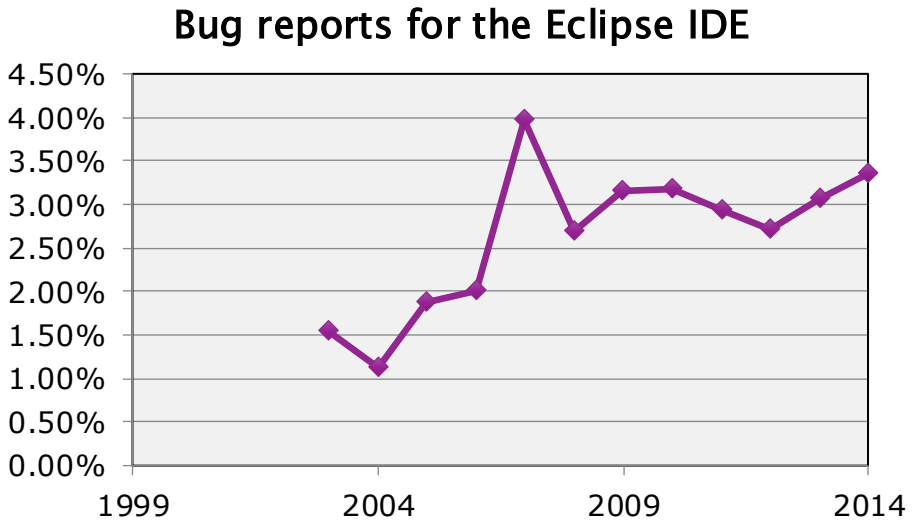
Slide courtesy of MIT 6.106.

Software Bugs Mentioning “Performance”



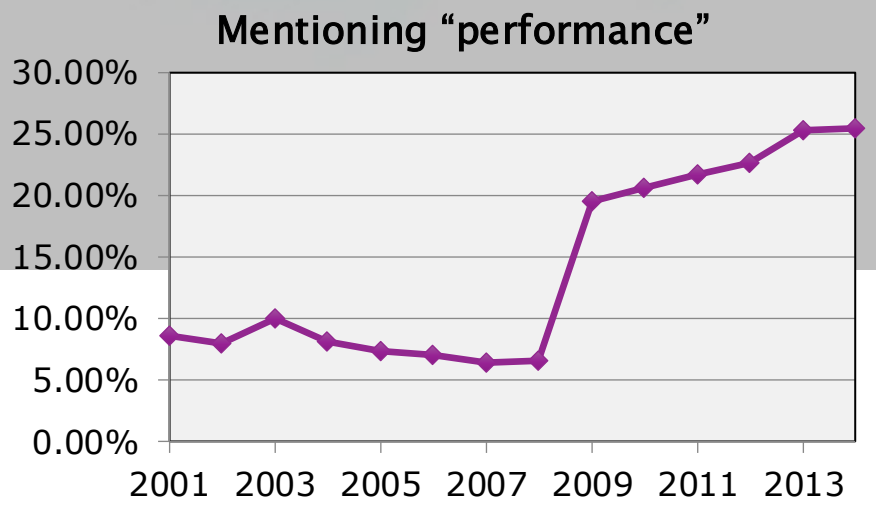


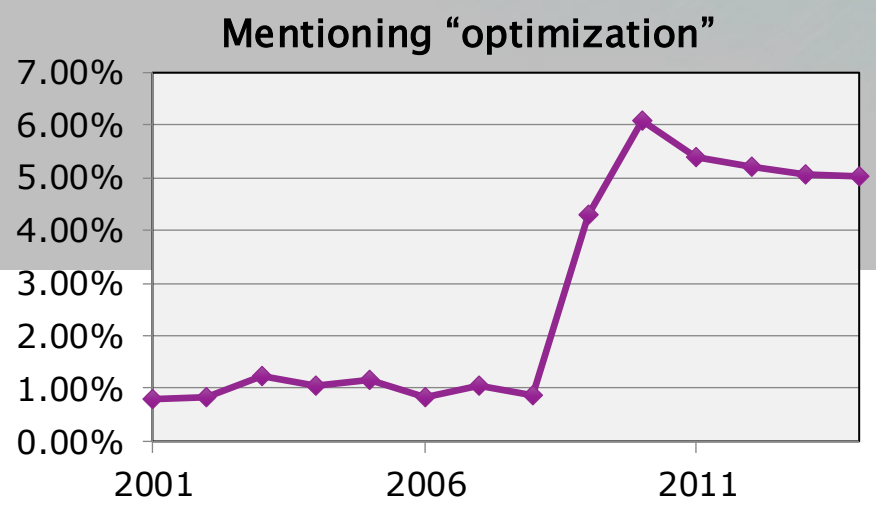


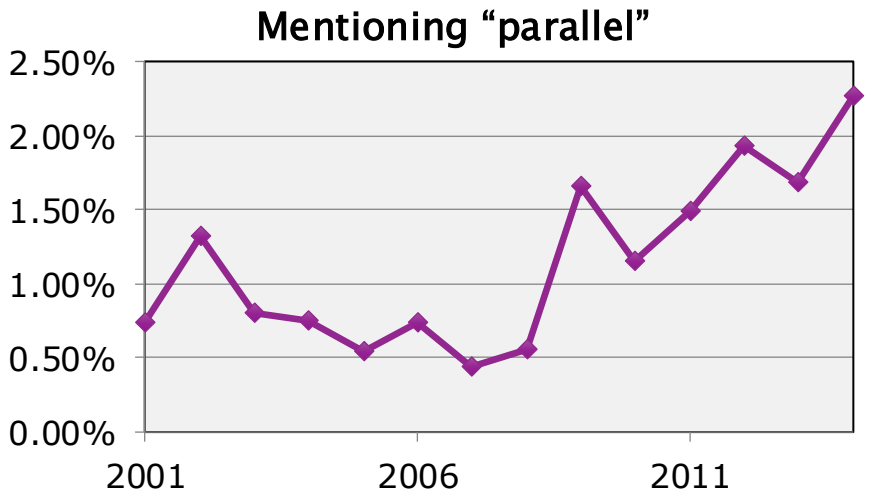


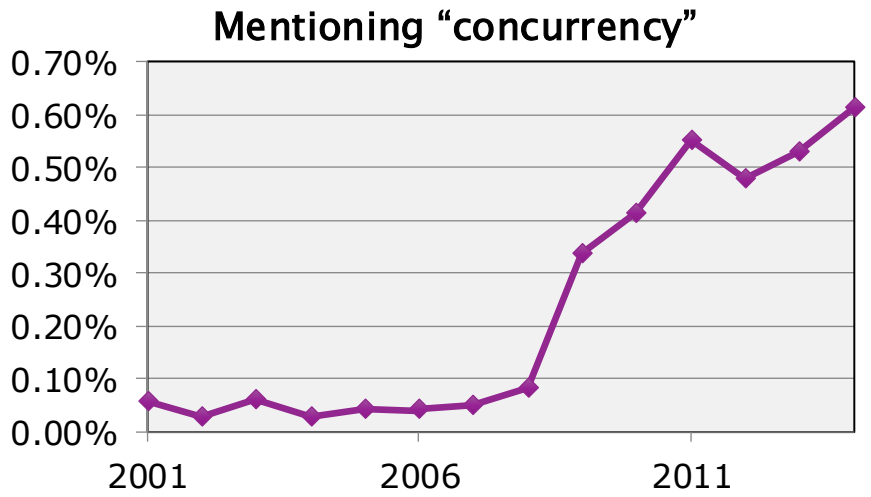
Slide courtesy of MIT 6.106.

Software Developer Jobs





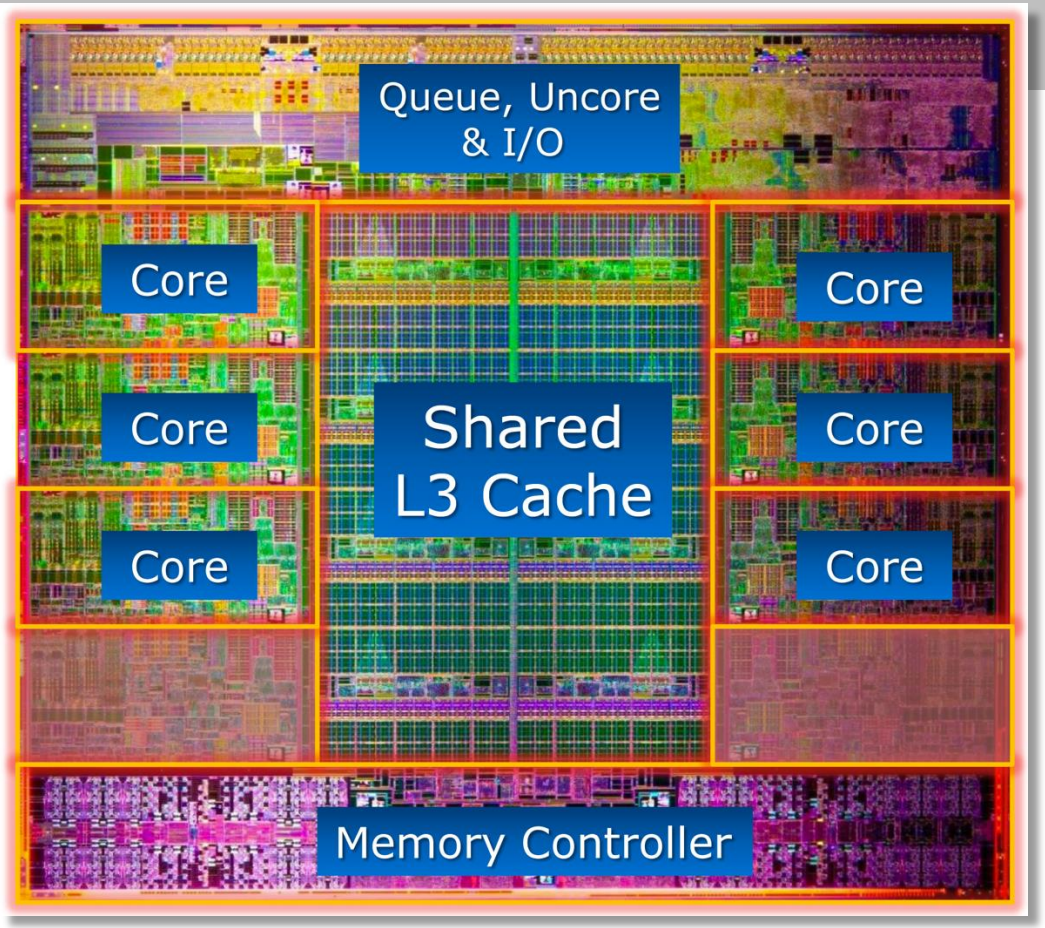




Source: Monster.com

Slide courtesy of MIT 6.106.

Parallel Programming Techniques



- A modern multicore desktop processor contains
  - parallel-processing cores,
  - vector units,
  - hierarchical caches,
  - instruction prefetchers,
  - GPU's,
  - simultaneous hyperthreading (SMT),
  - dynamic frequency scaling,
  - etc.
- These features can be challenging to use and require domain knowledge

Parallel Programming Techniques

Many things to keep in mind when performing parallel programming!

This class will teach you some **fundamental skills for parallel programming** for modern processors:

- Algorithms
- Implementation

- A modern multicore desktop processor contains
- parallel-processing cores,
- vector units,
- hierarchical caches,
- instruction prefetchers,
- GPU's,
- simultaneous hyperthreading (SMT),
- dynamic frequency scaling,
- etc.
- These features can be challenging to exploit.

Case Study

Matrix Multiplication

Square-Matrix Multiplication

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

C A B

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Assume for simplicity that $n = 2^k$.

Slide courtesy of MIT 6.106.

AWS c4.8xlarge Machine Specs

Feature	Specification
Microarchitecture	Haswell (Intel Xeon E5-2666 v3)
Clock frequency	2.9 GHz
Processor chips	2
Processing cores	9 per processor chip
Hyperthreading	2 way
Floating-point unit	8 double-precision operations, including fused-multiply-add, per core per cycle
Cache-line size	64 B
L1-icache	32 KB private 8-way set associative
L1-dcache	32 KB private 8-way set associative
L2-cache	256 KB private 8-way set associative
L3-cache (LLC)	25 MB shared 20-way set associative
DRAM	60 GB

Slide courtesy of MIT 6.106.

Nested Loops in Python

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
B = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
C = [[0 for row in xrange(n)]
     for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

print '%0.6f' % (end - start)
```

Slide courtesy of MIT 6.106.

Nested Loops in Python

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
B = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
C = [[0 for row in xrange(n)]
     for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

print '%0.6f' % (end - start)
```

Running time:

≈ 6 microseconds?

≈ 6 milliseconds?

≈ 6 seconds?

≈ 6 hours?

≈ 6 days?

Slide courtesy of MIT 6.106.

Nested Loops in Python

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
B = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
C = [[0 for row in xrange(n)]
     for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

print '%0.6f' % (end - start)
```

Running time:

≈ 6 microseconds?

≈ 6 milliseconds?

≈ 6 seconds?

≈ 6 hours?

≈ 6 days?

Slide courtesy of MIT 6.106.

Nested Loops in Python

Is this fast?

Should we expect more from our machine?

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
B = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
C = [[0 for row in xrange(n)]
     for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

print '%0.6f' % (end - start)
```

Running time:

≈ 6 microseconds?

≈ 6 milliseconds?

≈ 6 seconds?

≈ 6 hours?

≈ 6 days?

Slide courtesy of MIT 6.106.

Nested Loops in Python

Back-of-the-envelope calculation

$2n^3 = 2(2^{12})^3 = 2^{37}$ floating-point operations
Running time = 21042 seconds
∴ Python gets $2^{37}/21042 \approx 6.25$ MFLOPS
Peak ≈ 836 GFLOPS
Python gets ≈ 0.00075% of peak

Slide courtesy of MIT 6.106.

After Many Optimizations

53953x improvement!

Version	Implementation	Running time (s)	Relative speedup	Absolute Speedup	GFLOPS	Percent of peak
1	Python	21041.67	1.00	1	0.006	0.001
2	Java	2387.32	8.81	9	0.058	0.007
3	C	1155.77	2.07	18	0.118	0.014
4	+ interchange loops	177.68	6.50	118	0.774	0.093
5	+ optimization flags	54.63	3.25	385	2.516	0.301
6	Parallel loops	3.04	17.97	6,921	45.211	5.408
7	Parallel divide-and-conquer	1.30	1.38	16,197	105.722	12.646
8	+ compiler vectorization	0.70	1.87	30,272	196.341	23.486
9	+ AVX intrinsics	**0.39**	1.76	53,292	352.408	41.677
10	Intel MKL	**0.41**	0.97	51,497	335.217	40.098

Slide courtesy of MIT 6.106.

Parallel Programming Techniques

Banana slug:
0.02 mph

53953x

1079.06 mph

Parallel Programming Techniques

Banana slug:

0.02 mph

53953x

1079.06 mph

Parallel Programming Techniques

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
B = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
C = [[0 for row in xrange(n)]
     for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

print '%0.6f' % (end - start)
```

53953x

1079.06 mph

Parallel Programming Techniques

YOU!

```
import sys, random
from time import *

n = 4096

A = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
B = [[random.random()
      for row in xrange(n)]
     for col in xrange(n)]
C = [[0 for row in xrange(n)]
     for col in xrange(n)]

start = time()
for i in xrange(n):
    for j in xrange(n):
        for k in xrange(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

print '%0.6f' % (end - start)
```

53953x

1079.06 mph