Today

- Quick review?
- Parallelism
- Wrap-up ☹

What is parallel processing?

When can we execute things in parallel?

What is parallel processing?

- **We will briefly introduce the key ideas behind parallel processing**
  - instruction level parallelism
  - data-level parallelism
  - thread-level parallelism

Exploiting Parallelism

- **Of the computing problems for which performance is important, many have inherent parallelism**

- **Best example: computer games**
  - Graphics, physics, sound, AI etc. can be done separately
  - Furthermore, there is often parallelism within each of these:
    - Each pixel on the screen's color can be computed independently
    - Non-contacting objects can be updated/simulated independently
    - Artificial intelligence of non-human entities done independently

- **Another example: Google queries**
  - Every query is independent
  - Google searches are (ehm, pretty much) read-only!!

Parallelism at the Instruction Level

```
add     $2 <- $3, $4
or      $2 <- $2, $4
lw      $6 <- 0($4)
addi    $7 <- $6, 0x5
sub     $8 <- $8, $4
```

Dependences?
RAW
WAW
WAR

When can we reorder instructions?

When should we reorder instructions?

```
add     $2 <- $3, $4
or      $5 <- $2, $4
lw      $6 <- 0($4)
sub     $8 <- $8, $4
addi    $7 <- $6, 0x5
```

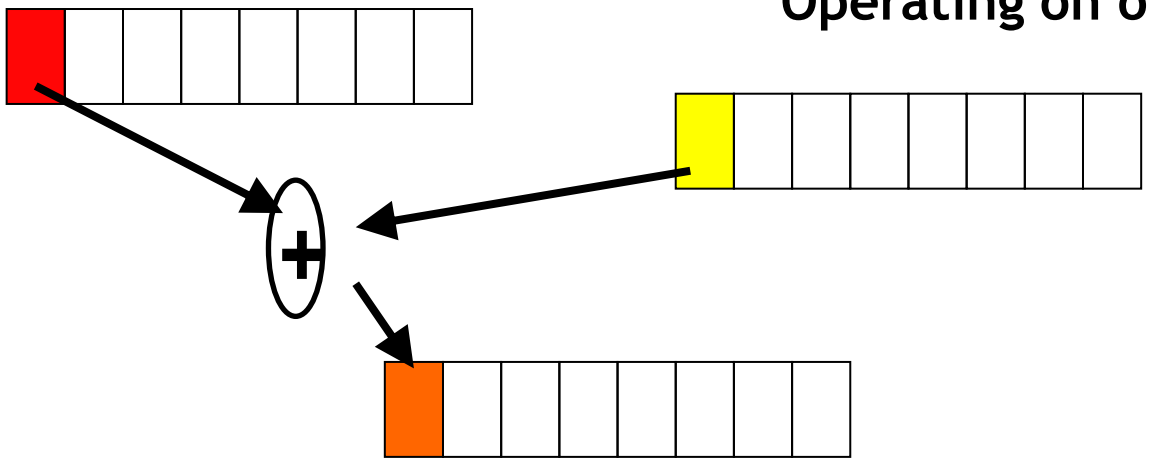
Superscalar Processors:
Multiple instructions executing in parallel at *same* stage

Exploiting Parallelism at the Data Level

- **Consider adding together two arrays:**

```
void
array_add(int A[], int B[], int C[], int length) {
  int i;
  for (i = 0 ; i < length ; ++ i) {
    C[i] = A[i] + B[i];
  }
}
```

Operating on one element at a time

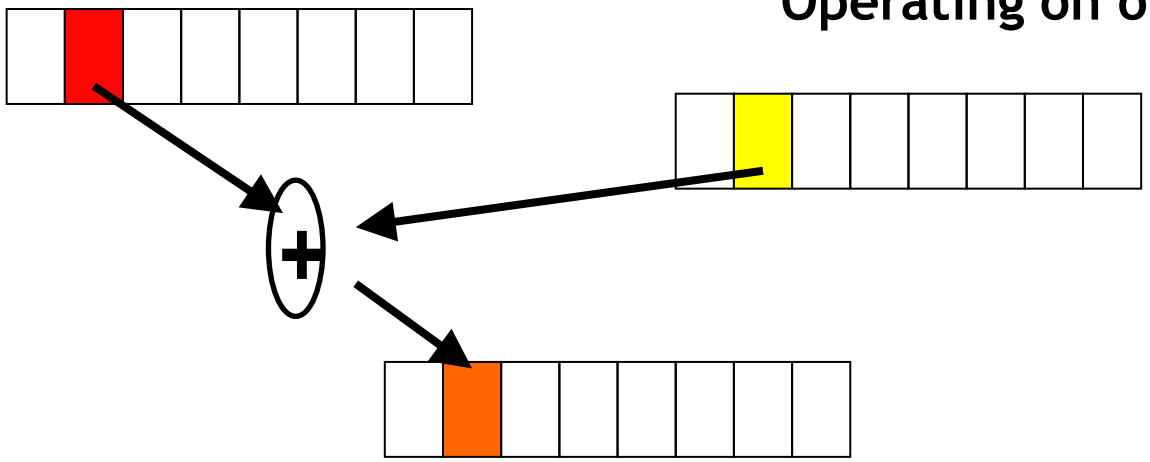


Exploiting Parallelism at the Data Level

- **Consider adding together two arrays:**

```
void
array_add(int A[], int B[], int C[], int length) {
  int i;
  for (i = 0 ; i < length ; ++ i) {
    C[i] = A[i] + B[i];
  }
}
```

Operating on one element at a time

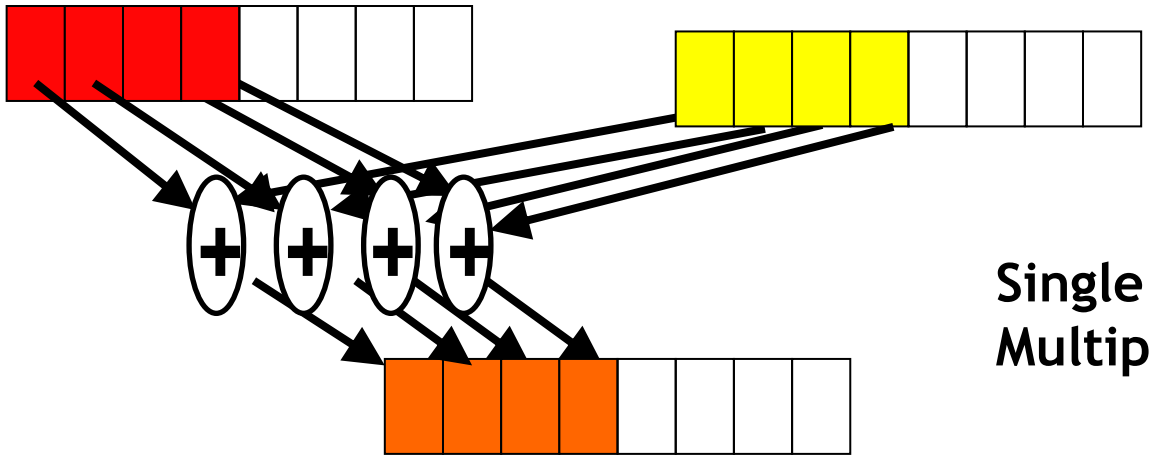


Exploiting Parallelism at the Data Level (SIMD)

- Consider adding together two arrays:

```
void
array_add(int A[], int B[], int C[], int length) {
  int i;
  for (i = 0 ; i < length ; ++ i) {
    C[i] = A[i] + B[i];
  }
}
```

Operate on MULTIPLE elements



Single Instruction, Multiple Data (SIMD)

Is it always that easy?

- Not always… a more challenging example:

```
unsigned
sum_array(unsigned *array, int length) {
  int total = 0;
  for (int i = 0 ; i < length ; ++ i) {
        total += array[i];
  }
  return total;
}
```

- Is there parallelism here?

We first need to restructure the code

```
unsigned
sum_array2(unsigned *array, int length) {
  unsigned total, i;
  unsigned temp[4] = {0, 0, 0, 0};
  for (i = 0 ; i < length & ~0x3 ; i += 4) {
    temp[0] += array[i];
    temp[1] += array[i+1];
    temp[2] += array[i+2];
    temp[3] += array[i+3];
  }
  total = temp[0] + temp[1] + temp[2] + temp[3];
  for ( ; i < length ; ++ i) {
    total += array[i];
  }
  return total;
}
```

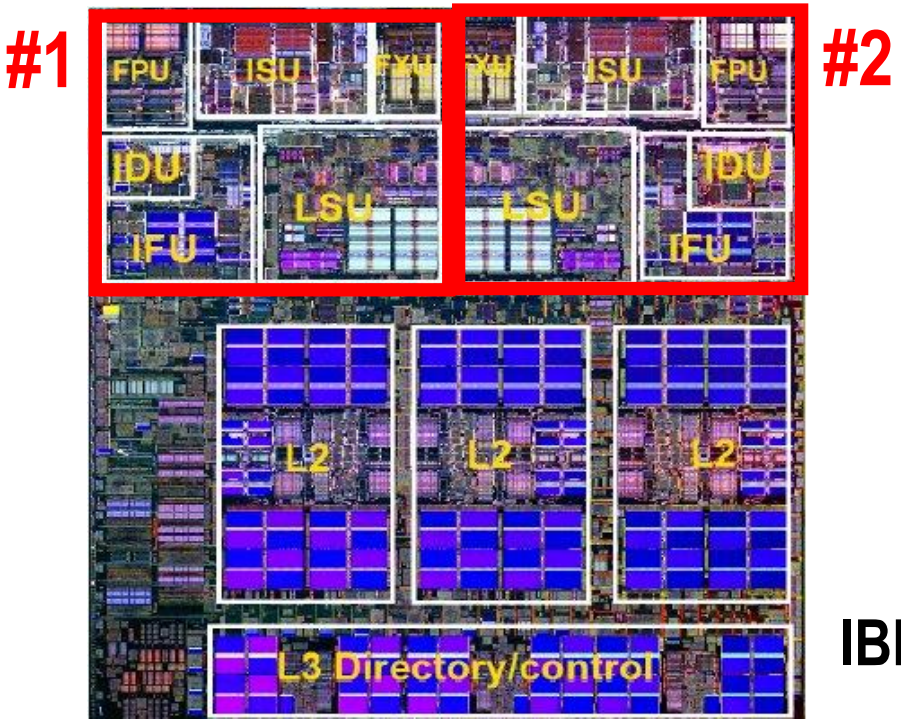
Then we can write SIMD code for the hot part

```
unsigned
sum_array2(unsigned *array, int length) {
  unsigned total, i;
  unsigned temp[4] = {0, 0, 0, 0};
  for (i = 0 ; i < length & ~0x3 ; i += 4) {
    temp[0] += array[i];
    temp[1] += array[i+1];
    temp[2] += array[i+2];
    temp[3] += array[i+3];
  }
  total = temp[0] + temp[1] + temp[2] + temp[3];
  for ( ; i < length ; ++ i) {
    total += array[i];
  }
  return total;
}
```

What's a thread?

Thread level parallelism: Multi-Core Processors

- Two (or more) complete processors, fabricated on the same silicon chip
- Execute instructions from two (or more) programs/threads at same time



IBM Power5

Multi-Cores are Everywhere

Intel Core Duo in Macs, etc.: 2 x86 processors on same chip

XBox360: 3 PowerPC cores

Sony Playstation 3: Cell processor, an asymmetric multi-core with 9 cores (1 general-purpose, 8 special purpose SIMD processors)

Why Multi-cores Now?

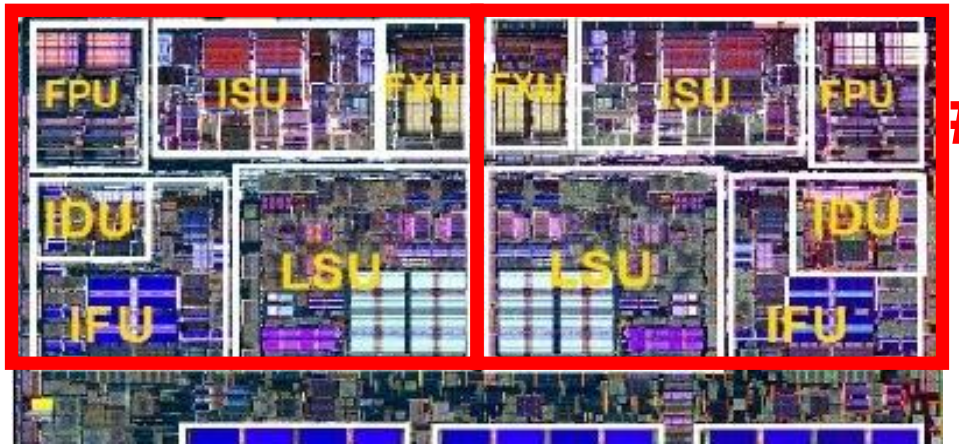
- Number of transistors we can put on a chip growing exponentially…
- But is performance growing too?

As programmers, do we care?

- What happens if we run this program on a multi-core?

```
void
array_add(int A[], int B[], int C[], int length) {
  int i;
  for (i = 0 ; i < length ; ++i) {
   C[i] = A[i] + B[i];
  }
}
```

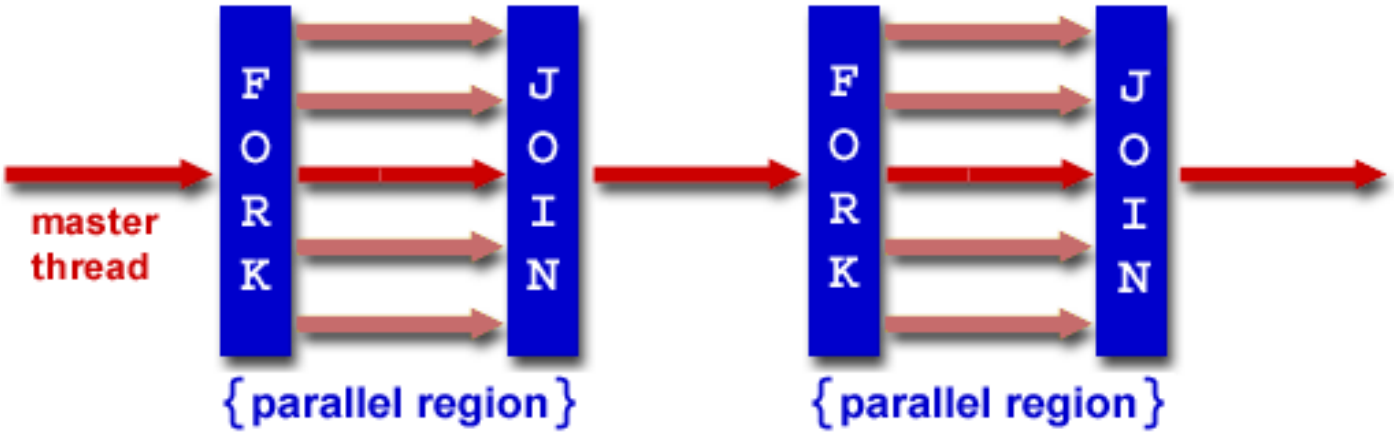
#1



#2

What if we want a program to run on multiple processors (cores)?

- **We have to explicitly tell the machine exactly how to do this**
  - This is called parallel programming or concurrent programming
- **There are many parallel/concurrent programming models**
  - We will look at a relatively simple one: **fork-join parallelism**

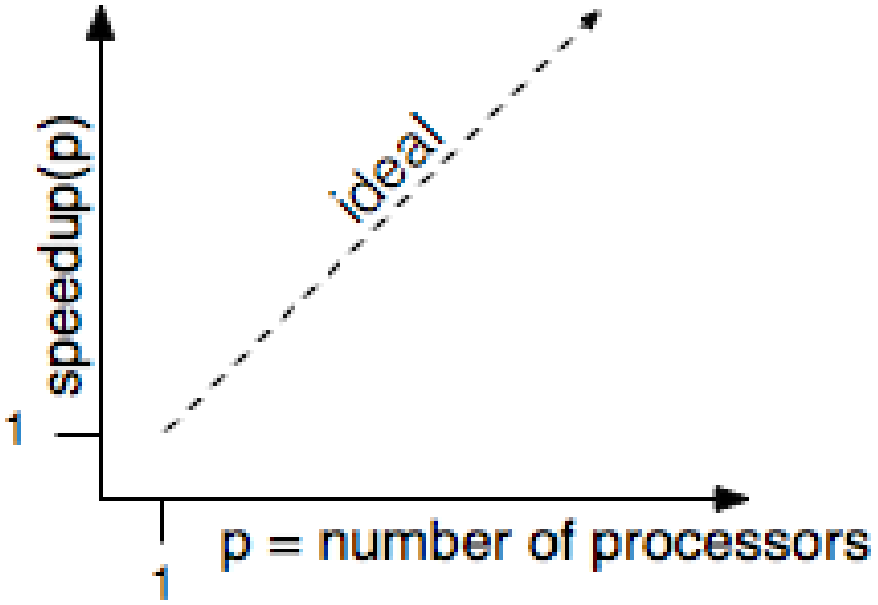


How does this help performance?

- **Parallel speedup measures improvement from parallelization:**

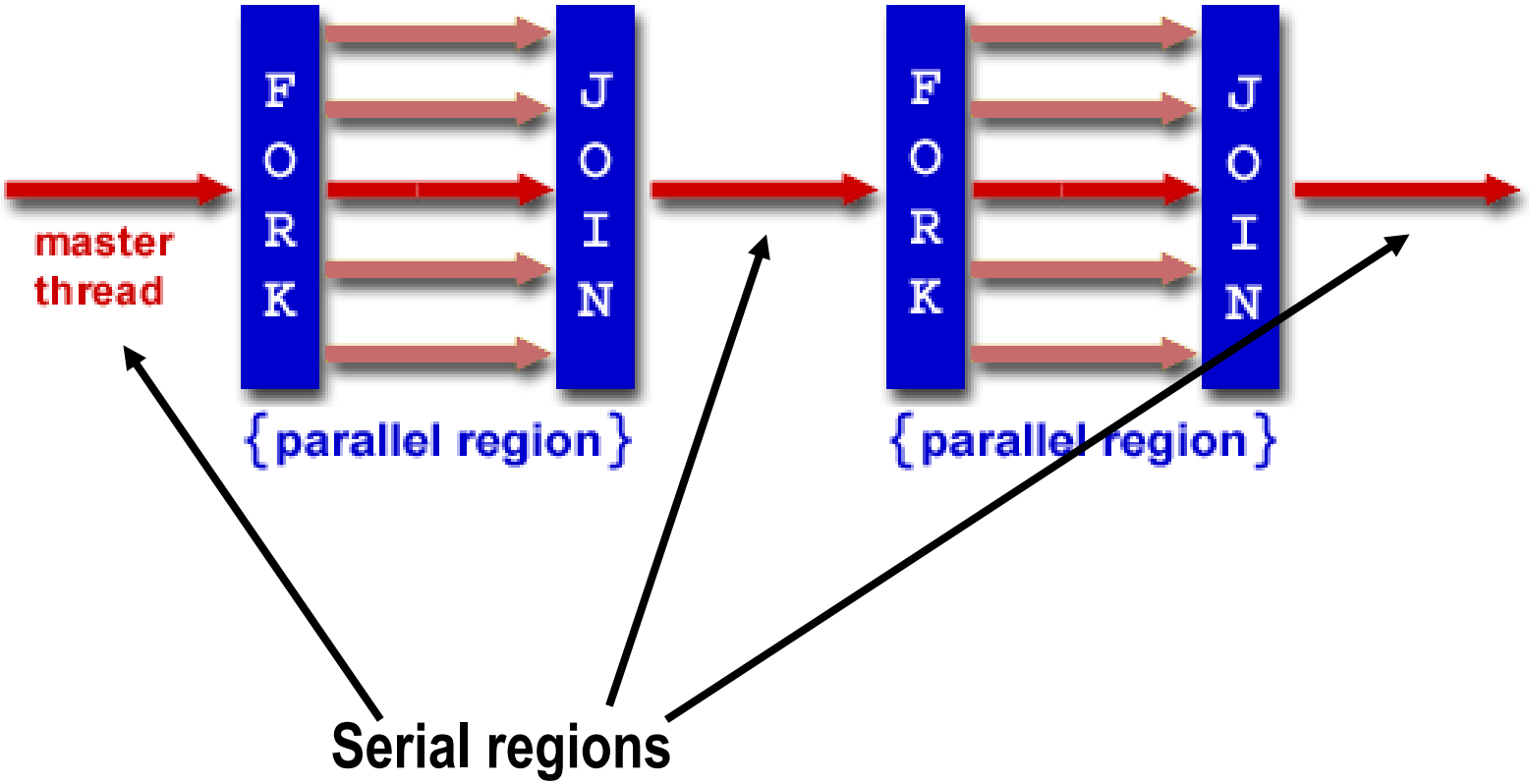
$$\text{speedup}(p) = \frac{\text{time for best serial version}}{\text{time for version with } \mathbf{p} \text{ processors}}$$

- **What can we realistically expect?**



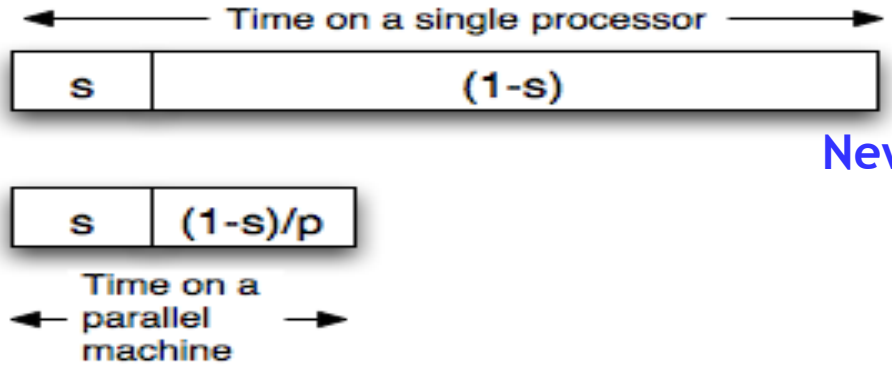
Reason #1: Amdahl's Law

- In general, the whole computation is not (easily) parallelizable



Reason #1: Amdahl's Law

- Suppose a program takes 1 unit of time to execute serially
- A fraction of the program, s, is inherently serial (unparallelizable)



$$\text{New Execution Time} = \frac{1-s}{p} + s$$

- For example, consider a program that, when executing on one processor, spends 10% of its time in a non-parallelizable region. How much faster will this program run on a 3-processor system?

$$\text{New Execution Time} = \frac{.9T}{3} + .1T = \qquad \text{Speedup} =$$

- What is the maximum speedup from parallelization?

Reason #2: Overhead

- Forking and joining is not instantaneous
  - Involves communicating between processors
  - May involve calls into the operating system
    - Depends on the implementation

$$\text{New Execution Time} = \frac{1-s}{P} + s + \text{overhead}(P)$$

Summary

- **Multi-core is having more than one processor on the same chip.**
  - Almost all devices now have multicore processors
  - Results from Moore's law and power constraint

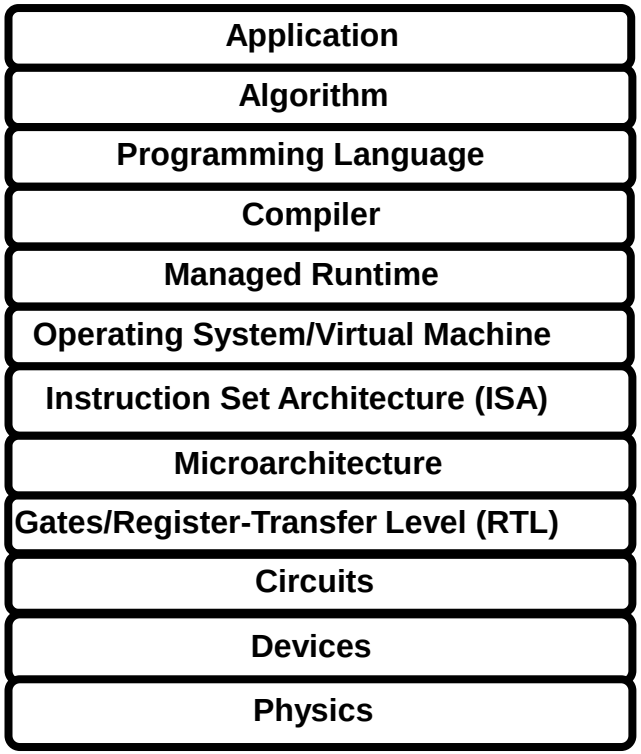
- **Exploiting multi-core requires parallel programming**
  - Automatically extracting parallelism too hard for compiler, in general.
  - But, can have compiler do much of the bookkeeping for us
  - OpenMP

- **Fork-Join model of parallelism**
  - At parallel region, fork a bunch of threads, do the work in parallel, and then join, continuing with just one thread
  - Expect a speedup of less than P on P processors
    - Amdahl's Law: speedup limited by serial portion of program
    - Overhead: forking and joining are not free

The Big Theme

- **THE HARDWARE/SOFTWARE INTERFACE**
- **How does the hardware (0s and 1s, processor executing instructions) relate to the software (Java programs)?**
- **Computing is about abstractions (but don't forget reality)**
- **What are the abstractions that we use?**
- **What do YOU need to know about them?**
  - When do they break down and you have to peek under the hood?
  - What assumptions are being made that may or may not hold in a new context or for a new technology?
  - What bugs can they cause and how do you find them?
- **Become a better programmer and begin to understand the thought processes that go into building computer systems**

The system stack!



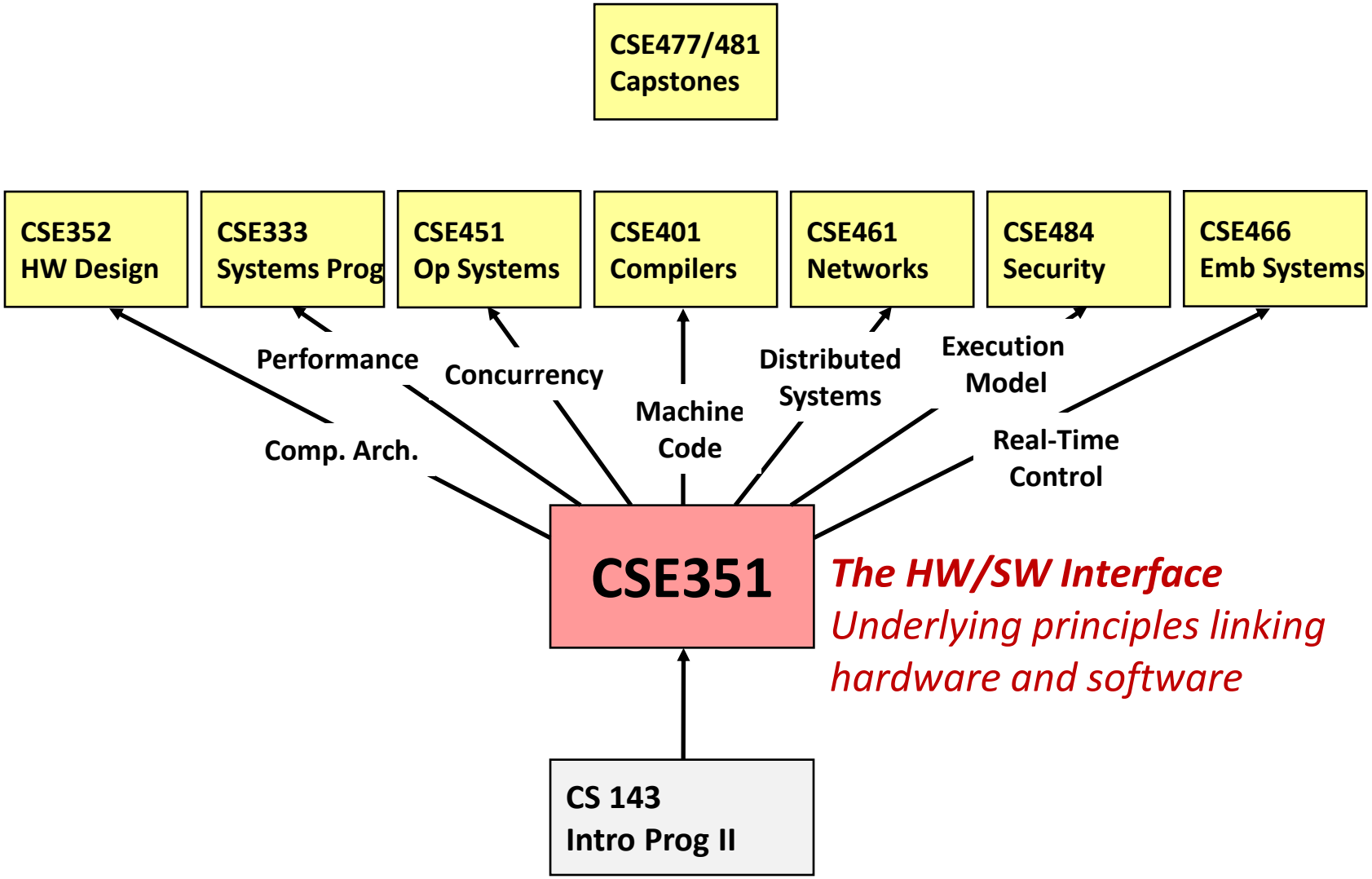
From 1st lecture

Course Outcomes

- **Foundation: basics of high-level programming**
- **Understanding of some of the abstractions that exist between programs and the hardware they run on, why they exist, and how they build upon each other**
- **Knowledge of some of the details of underlying implementations**
- **Become more effective programmers**
  - More efficient at finding and eliminating bugs
  - Understand the many factors that influence program performance
  - Facility with some of the many languages that we use to describe programs and data
- **Prepare for later classes in CSE**

From 1st lecture

CSE351's place in new CSE Curriculum



The HW/SW Interface

Underlying principles linking hardware and software

The Hard Things to Evaluate

- **What will you remember in going on to next core courses?**
- **What will you remember in senior year, for later courses?**
- **Will this have an impact on ability to get internships/jobs?**
- **Will this enable deeper participation in range of research?**

- **This will take several years to assess**
- **CSE351 will likely evolve over time – moving target**
- **Negotiation of content with follow-on courses**
  - e.g., use of X86/Y86 for implementation in 352
  - e.g., overlap with topics in 333 (systems programming), 390A (unix)
  - e.g., different background entering 401, 484, 451, others
  - e.g., sufficiency of background from 142/143