What is parallel processing?

When can we execute things in parallel?

Parallelism:
Use extra resources to solve a problem faster

Concurrency:
Correctly and efficiently manage access to shared resources

thanks to Dan Grossman for the succinct definitions
What is parallel processing?

Briefly introduction to key ideas of 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

- 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

- search engine queries
  - Every query is independent
  - Searches are (ehm, pretty much) read-only!!
Instruction-Level Parallelism

\[
\begin{align*}
\text{add} & \quad %r2 \leftarrow %r3, %r4 \\
\text{or} & \quad %r5 \leftarrow %r2, %r4 \\
\text{lw} & \quad %r6 \leftarrow 0(\%r4) \\
\text{addi} & \quad %r7 \leftarrow %r6, 0x5 \\
\text{sub} & \quad %r8 \leftarrow %r8, %r4
\end{align*}
\]

Dependences?

- RAW – read after write
- WAW – write after write
- WAR – write after read

When can we reorder instructions?

When should we reorder instructions?

Superscalar Processors:

Multiple instructions executing in parallel at *same* stage

Take 352 to learn more.
**Data Parallelism**

- Consider adding together two arrays:

```c
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
Data Parallelism

Consider adding together two arrays:

```c
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
Consider adding together two arrays:

```c
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];
  }
}
```

Data Parallelism with SIMD
- Operate on MULTIPLE elements
  - Single Instruction, Multiple Data (SIMD)
Is it always that easy?

- Not always... a more challenging example:

```c
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?
- Each loop iteration uses data from previous iteration.
Restructure the code for SIMD...

// one option...
unsigned sum_array2(unsigned *array, int length) {
    unsigned total, i;
    unsigned temp[4] = {0, 0, 0, 0};
    // chunks of 4 at a time
    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];
    }
    // add the 4 sub-totals
    // add the non-4-aligned parts
    for ( ; i < length ; ++ i) {
        total += array[i];
    }
    return total;
}
What are threads?

- Independent “thread of control” within process
- Like multiple processes within one process, but sharing the same virtual address space.
  - logical control flow
    - program counter
    - stack
  - shared virtual address space
    - all threads in process use same virtual address space

- Lighter-weight than processes
  - faster context switching
  - system can support more threads
Thread-level parallelism: Multicore Processors

- Two (or more) complete processors, fabricated on the same silicon chip
- Execute instructions from two (or more) programs/threads at same time
Multicores are everywhere. (circa 2013)

- **Laptops, desktops, servers**
  - Most any machine from the past few years has at least 2 cores

- **Game consoles:**
  - Xbox 360: 3 PowerPC cores; Xbox One: 8 AMD cores
  - PS3: 9 Cell cores (1 master; 8 special SIMD cores);
    PS4: 8 custom AMD x86-64 cores
  - Wii U: 2 Power cores

- **Smartphones**
  - iPhone 4S, 5: dual-core ARM CPUs
  - Galaxy S II, III, IV: dual-core ARM or Snapdragon
  - ...
Why Multicores Now?

- Number of transistors we can put on a chip growing exponentially...
- But performance is no longer growing along with transistor count.
- So let’s use those transistors to add more cores to do more at once...
As programmers, do we care?

■ What happens if we run this program on a multicore?

```c
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 one 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:
  
  $$\text{speedup}(p) = \frac{\text{time for best serial version}}{\text{time for version with } p \text{ processors}}$$

- What can we realistically expect?
Reason #1: Amdahl’s Law

- In general, the whole computation is not (easily) parallelizable
- Serial regions limit the potential parallel speedup.
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?

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

- 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

\[
\text{New Execution Time} = \frac{1-s}{p} + s + \text{overhead}(P)
\]
Multicore: what should worry us?

- **Concurrency**: what if we’re sharing resources, memory, etc.?

- **Cache Coherence**
  - What if two cores have the same data in their own caches? How do we keep those copies in sync?

- **Memory Consistency, Ordering, Interleaving, Synchronization**...
  - With multiple cores, we can have *truly* concurrent execution of threads. In what order do their memory accesses appear to happen? Do the orders seen by different cores/threads agree?

- **Concurrency Bugs**
  - When it all goes wrong...
  - Hard to reproduce, hard to debug
  - [http://cacm.acm.org/magazines/2012/2/145414-you-dont-know-jack-about-shared-variables-or-memory-models/fulltext](http://cacm.acm.org/magazines/2012/2/145414-you-dont-know-jack-about-shared-variables-or-memory-models/fulltext)
Summary

- **Multicore: more than one processor on the same chip.**
  - Almost all devices now have multicore processors
  - Results from Moore’s law and power constraint

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

- **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

- **Take 332, 352, 451 to learn more!**
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

- Application
- Algorithm
- Programming Language
- Compiler
- Managed Runtime
- Operating System/Virtual Machine
- Instruction Set Architecture (ISA)
- Microarchitecture
- Gates/Register-Transfer Level (RTL)
- Circuits
- Devices
- Physics
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

From 1st lecture

The HW/SW Interface
Underlying principles linking hardware and software