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?

- Brief 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 (pretty much) read-only!!
Instruction-Level Parallelism

```
add %r2 <- %r3, %r4
or %r2 <- %r2, %r4
lw %r6 <- 0(%r4)
addi %r7 <- %r6, 0x5
sub %r8 <- %r8, %r4
```

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

When can we reorder instructions?

```
add %r2 <- %r3, %r4
or %r5 <- %r2, %r4
lw %r6 <- 0(%r4)
sub %r8 <- %r8, %r4
addi %r7 <- %r6, 0x5
```

When should we reorder instructions?

Superscalar Processors:
- Multiple instructions executing in parallel at *same* stage
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
Data Parallelism with SIMD

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];
  }
}
```

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: 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
Multi-cores 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
  - ...

Spring 2014

Wrap-up
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];
    }
}
```

As programmers, do we care?
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.

Diagram showing parallel and serial regions with a master thread forking and joining processes.
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!**