Multi Processing

Introduction

1. All modern processors have multiple cores and typically also have more than one thread per core

Flynn's Taxonomy of Parallel Machines

1. How many instruction streams?
2. How many data streams?
3. Notation
    - SISD: Single instruction single data
    - SIMD: Single instruction multiple data
        - Vector instruction (SSE/MMX)
    - MISD: Multiple instruction single data
        - Stream processor
        - This isn't used much
    - MIMD: Multiple instruction multiple data
        - Normal multiprocessor comercially available today
        - Lecture focuses here

	Instruction Streams	Data Streams
SISD	1	1
SIMD	1	> 1
MISD	> 1	1
MIMD	> 1	> 1

Why Multiprocessors

1. Uniprocessor is already ~4-wide
    - Diminishing returns from making the processor wider
    - Must increase clock frequency and voltage -> Power increases cubically
    - But, Moore's Law continues
        - 2x transistors every 18 months
        - Instead, 2x cores every 18 months
2. This assumes we can use all of the cores
    - More cores on a single-threaded program doesn't provide any advantage

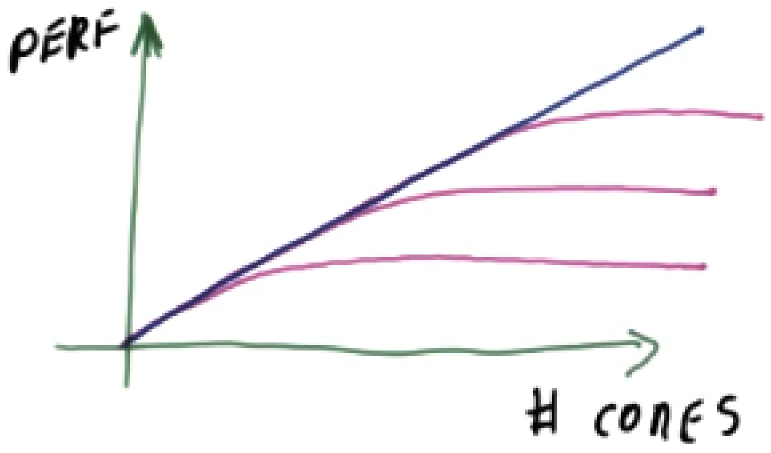
Multicore vs Single Core Quiz

1. Consider the following processors:
    - Old processor: 2 cm^2, IPC = 2.5, 100W @ 2 GHz
    - New processor: 1 cm^2, IPC = 2.5, 50W @ 2 GHz
2. Better single core processor:
    - 2 cm^2, IPC = 3.5, 75W @ 2 GHz
3. If we allow the processor to use all 100W, what is the frequency and speedup vs the old processor?
    - fnew = 1.33 ^ (1/3) * fold = 1.33 ^ (1/3) * 2 = 2.2 GHz
        - P ~ V ^ 2 * f ~ f ^ 3
        - Increasing frequency means increasing voltage as well, so power scales according to the frequency cubed
    - Speedup = 2.2 / 2 * 3.5 / 2.5 = 1.1 * 1.4 = 1.54
4. Multi-core processor:
    - 2 cm^2, IPC = 2.5 on each core, 2 * 50W @ 2 GHz
5. What is the speedup vs the old processor?

- Assuming old work can simply be divided among cores
- Speedup = 2

Multiprocessor Needs Parallel Programs

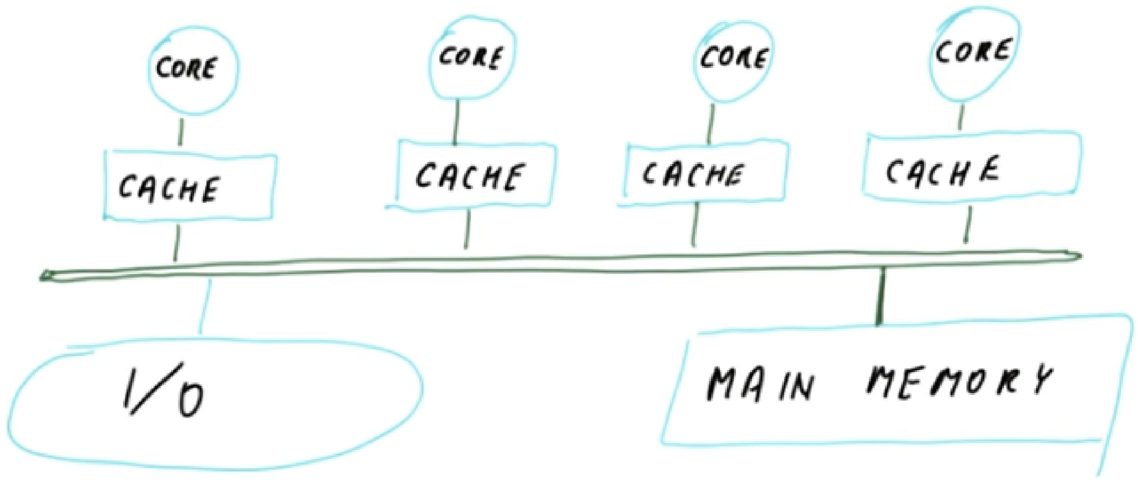
1. Sequential (single-thread) code is much easier to develop
2. Debugging parallel code is much more difficult
3. Performance scaling is very hard to achieve
    - Performance scaling: As we increase the number of cores, performance also continues to increase



Performance Scaling

Centralized Shared Memory

1. Cores are all connected to main memory and I/O devices through a shared bus
    - UMA: Uniform memory access (time)
        - Time to access main memory is identical
    - SMP: Symmetric multiprocessor
        - Any core/cache looks like any other core/cache



Centralized Memory

Multicore Quiz

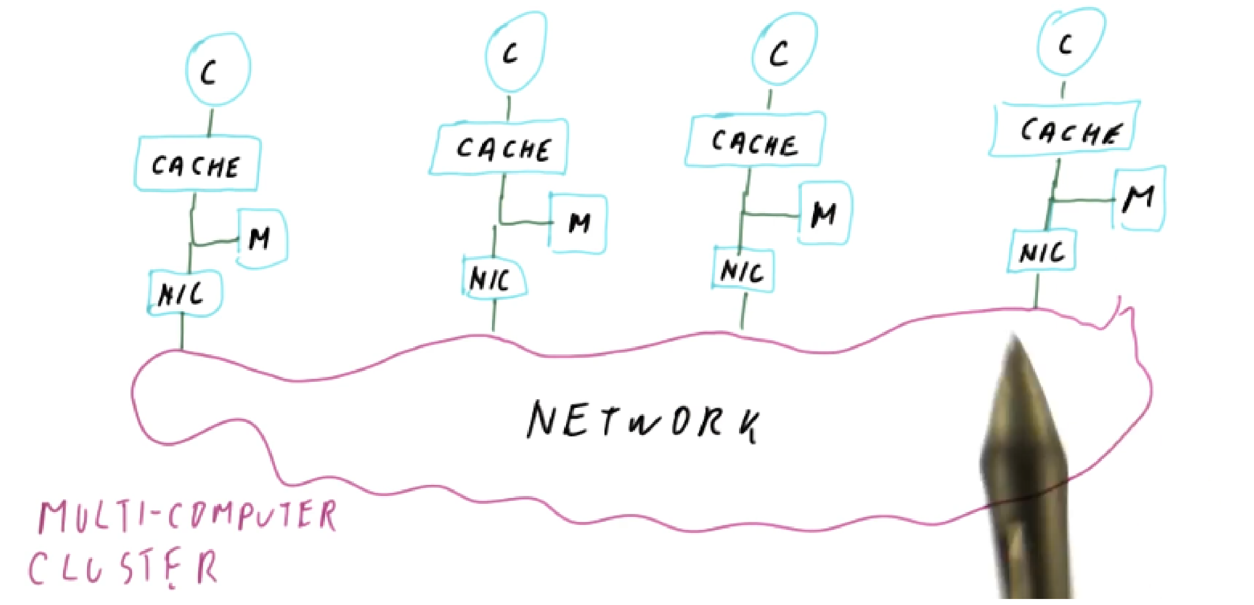
1. Want lots of cores with centralized shared memory
2. Which of these are problems?
    - Memory needs to be large (and slow) (true)
    - Memory gets too many accesses/second (true)
    - We'll need a lot of small memory modules (false)
    - Caches will have very small block size (false)
    - Pages for virtual memory become too small (false)

Centralized Main Memory Problems

1. Memory size
    - Need large memory -> slow memory (far from all cores)
2. Memory bandwidth
    - Miss from all cores -> memory bandwidth contention
    - Accesses are serialized
3. Centralized main memory works well only for smaller machines
    - 2, 4, 8, 16 (maybe) cores

Distributed Shared Memory

1. Actually only distributed memory (not shared)
2. One core can access a memory slice
3. A core must create a message to send a request to another core and retrieve the data
    - In distributed shared memory, we use shared memory for communication
    - In distributed memory, we must use message passing
4. Also called a multi-computer or cluster
    - Equivalent to writing a program across separate computers
    - Scale to a large number of processors
        - Programmer is forced to explicitly think about communicating efficiently



Distributed Memory

Numa Memory Allocation Quiz

1. The operating system should:
   - Put the stack pages for core N in the memory slice N (true)
   - Put all data pages ever touched by core N in the memory slice N (false)
   - Put all data pages mostly accessed by core N in the memory slice N (true)

A Message Passing Program

```
#define ASIZE 1024
#define NUMPROC 4

double myArray[ASIZE/NUMPROC]; // each processor gets 1/4 of array
double mySum = 0;

for(int i = 0; i < ASIZE/NUMPROC; i++)
{
    mySum += myArray[i];
}
if(myPID == 0)
{
    for(int p = 1; p < NUMPROC; p++)
    {
        int pSum;
        recv(p, pSum); // need to align sends and receives
        mySum += pSum;
    }
    printf("Sum: %lf\n", mySum);
}
else
{
    send(0, mySum); // need to align sends and receives
}
```

A Shared Memory Program

```
#define ASIZE 1024
#define NUMPROC 4

shared double myArray[ASIZE]; // array is in shared memory
shared double allSum = 0;
shared mutex sumLock;

double mySum = 0;

// sum up its "own" part of the array
for(int i = myPID*ASIZE/NUMPROC; i < (myPID+1) * ASIZE/NUMPROC; i++)
{
    mySum += array[i];
}

// synchronize access to allSum through critical section
lock(sumLock);
allSum += mySum;
unlock(sumLock);

// need a barrier here

if(myPID == 0)
{
    printf("Sum: %lf\n", allSum);
}
```

Message Passing vs Shared Memory

	Message Passing	Shared Memory
Communication	Programmer	Automatic
Data Distribution	Manual	Automatic
Hardware Support	Simple	Extensive
Programming		
Correctness	Difficult	Less difficult
Performance	Difficult	Very difficult

Message Passing vs Shared Memory Quiz

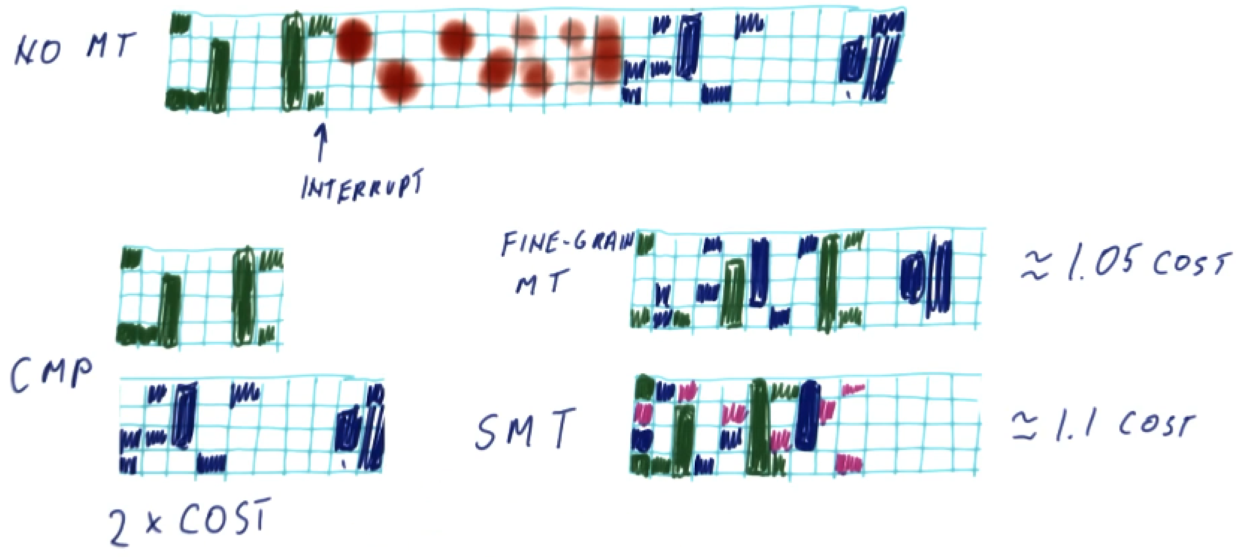
1. One core initializes an array, then all others read the entire array
2. Data distribution for lines of code
    - Message Passing > Shared Memory
3. Synchronization adds how many lines?
    - In message passing? 0
    - In shared memory? Greater than 0
4. This is typical
    - Message passing needs to distribute data, but synchronization is free
    - Shared memory doesn't need to distribute data, but synchronization is required

Shared Memory Hardware

1. Multiple cores share same physical address space
    - UMA, NUMA
2. Multi-threading by time-sharing a core
    - Same core -> same physical memory
3. Hardware multithreading in a core
    - Coarse-grain: Change thread every few cycles
    - Fine-grain: Change thread every cycle
    - Simultaneous multi-threading: Completing instructions from different threads in any given cycle (hyperthreading)

Multi Threading Performance

1. No MT: Single core, OS intervenes to switch contexts between processes and provide the illusion of simultaneous processing
2. CMP: Two separate physical cores with double the cost
3. Fine-grain MT: Instructions from different processes alternate between cycles
4. SMT: Instructions from different processes can be executed in the same cycle



Performance

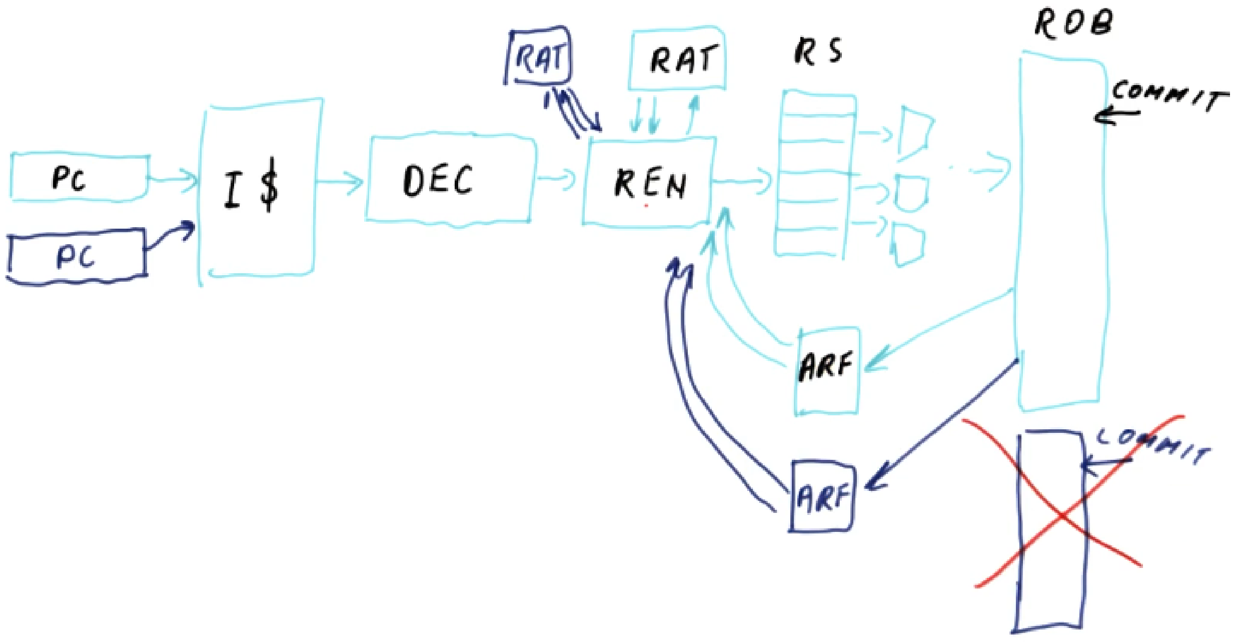
SMT vs Dual Core Quiz

1. One floating point intensive thread
2. One integer-only thread
3. 4-issue core, 2 FP + 2 integer / cycle
4. Do we get better performance per dollar with SMT or two cores?
    - SMT because the two threads are operating on separate hardware, so there's no need to have multiple units. We won't gain additional performance from this additional cost
    - Two core would be better if both threads were integer-dominant

SMT Hardware Changes

1. Most processors use fine-grained multithreading to handle fetching instructions

    - Gather instructions from one thread one cycle and a different one the next
2. Decoder does the same thing regardless
3. Renamer does the same thing, but we need multiple RATs for different threads
4. We don't duplicate the ROB; instead, we wait for all instructions for one cycle to be done before committing
5. Need separate register files per thread
6. Most of the cost of the processor is in the instruction/data cache and ROB
    - Adding PC, RAT, and ARF doesn't add much to the cost



Hardware

SMT - Data Cache - TLB

1. If we access the data cache with a VIVT, we can get the wrong data because virtual addresses can be the same between threads, but reference different physical memory locations
2. VIPT or PIPT resolve this issue due to the physical tag
    - TLB must be thread-aware
        - This is done by adding a bit for each entry to determine which thread this entry is for

SMT and Cache Performance

1. Cache shared by all SMT threads
2. Fast data sharing eliminates the need for shared memory
    - Thread 0: SW A
    - Thread 1: LW A
3. Cache capacity (and associativity) shared
    - WS(t0) + WS(t1) - WS(t0,t1) > Size of data cache
        - This causes cache thrashing
4. If WS(t0) < D$ size -> SMT is worse than one-at-a-time
    - We might lose more performance due to cache misses than we gained by overlapping execution
    - There's an underlying assumption that cache performance stays the same

SMT and Cache Quiz

1. Program A: Working set is 10 kB
2. Program B: Working set is 4 kB
3. Program C: Working set is 1 kB
4. Processor is 40way SMT, 8 kB data L1 cache
5. We should run:
    - A, B, C (no)
    - A, B then C (no)
    - A then B, C (yes)
    - A, C then B (no)
    - A then B then C (no)

Conclusion

1. Message passing vs shared memory for parallel programs
2. Next lecture covers caching when multiple cores are sharing memory