Lecture 28
Introduction to Parallel Processing and some Architectural Ramifications

General context: Multiprocessors

- Multiprocessor is any computer with several processors
- SIMD
  - Single instruction, multiple data
  - Modern graphics cards
- MIMD
  - Multiple instructions, multiple data

Lemieux cluster, Pittsburgh supercomputing center

Multiprocessing

- Flynn's Taxonomy of Parallel Machines
  - How many Instruction streams?
  - How many Data streams?

(note: we'll spend most time talking about just 1 class…)

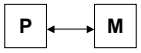
- SISD: Single I Stream, Single D Stream
  - A uniprocessor
- SIMD: Single I, Multiple D Streams
  - Each "processor" works on its own data
  - But all execute the same instrs in lockstep
  - E.g. a vector processor or MMX

Flynn's Taxonomy

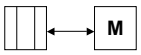
- MISD: Multiple I, Single D Stream
  - Not used much
- MIMD: Multiple I, Multiple D Streams
  - Each processor executes its own instructions and operates on its own data
  - This is your typical off-the-shelf multiprocessor (made using a bunch of "normal" processors)
    - Not superscalar
    - Each node is superscalar
    - Lessons will apply to multi-core too!

Or…in pictures!

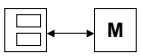
- Uni:



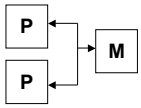
- Pipelined
- Superscalar



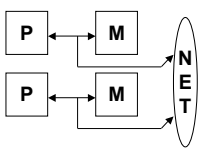
- VLIW/"EPIC"



- SMP ("Symmetric")
  - (Also "CSM")



- Distributed



Multiprocessors

- Why do we need multiprocessors?
  - Uniprocessor speed improving fast
  - But there are things that need even more speed
    - Wait for a few years for Moore's law to catch up?
    - Or use multiple processors and do it now?
    - (Is Moore's Law still catching up? M/C?)
- Multiprocessor software problem
  - Most code is sequential (for uniprocessors)
    - MUCH easier to write and debug
  - Correct parallel code very, very difficult to write
    - Efficient and correct is much more difficult
    - Debugging even more difficult

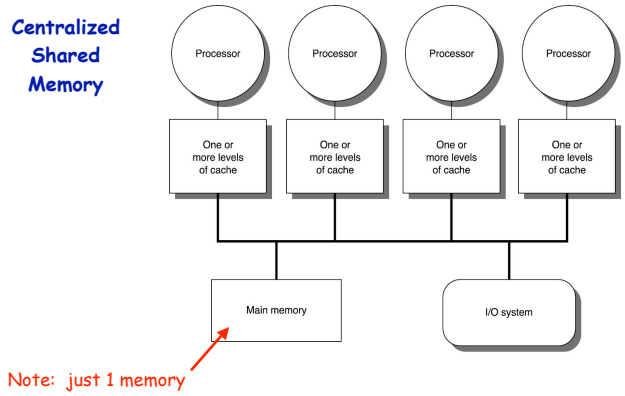
Let's look at a few MIMD example configurations…

Multiprocessor memory types

- Shared memory:
  In this model, there is one (large) common shared memory for all processors
- Distributed memory:
  In this model, each processor has its own (small) local memory, and its content is not replicated anywhere else

MIMD Multiprocessors

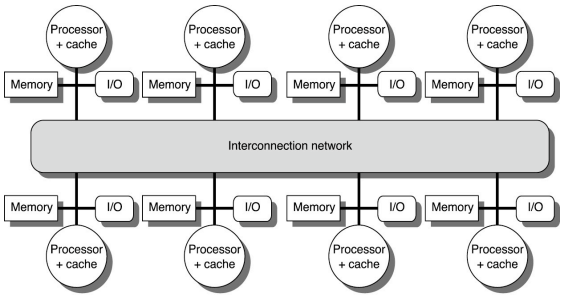
Centralized Shared Memory



Note: just 1 memory

MIMD Multiprocessors

Distributed Memory



Multiple, distributed memories here.

Before, we did parallel processing by chaining together separate processors.

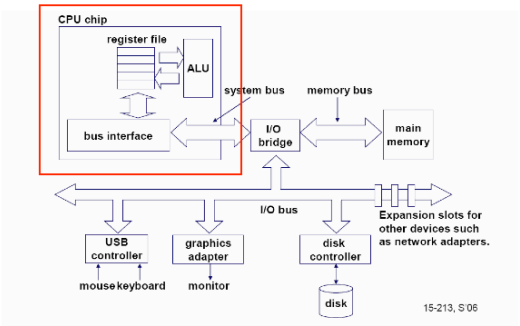
Now we can do it on the same chip.

Multi-core processor is a special kind of a multiprocessor:

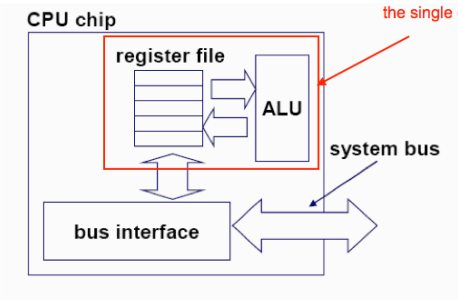
All processors are on the same chip

- Multi-core processors are MIMD:
  Different cores execute different threads (Multiple Instructions), operating on different parts of memory (Multiple Data).
- Multi-core is a shared memory multiprocessor:
  All cores share the same memory

Single-core computer

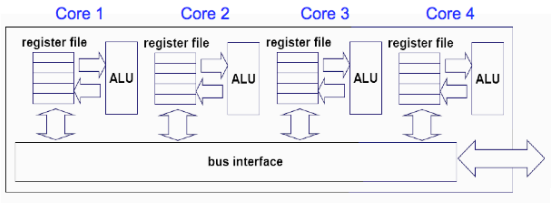


Single-core CPU chip



Multi-core architectures

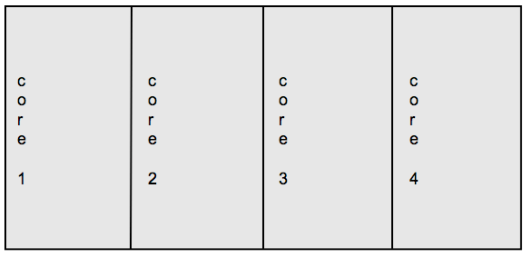
- This lecture is about a new trend in computer architecture:
  Replicate multiple processor cores on a single die.



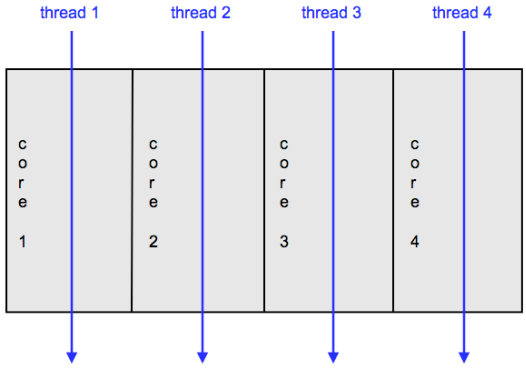
Multi-core CPU chip

Multi-core CPU chip

- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)

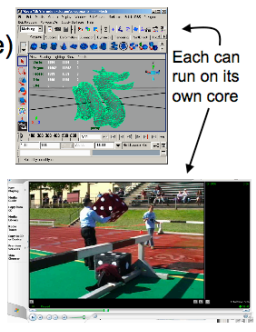


The cores run in parallel



What applications benefit from multi-core?

- Database servers
- Web servers (Web commerce)
- Compilers
- Multimedia applications
- Scientific applications, CAD/CAM
- In general, applications with *Thread-level parallelism* (as opposed to instruction-level parallelism)



More examples

- Editing a photo while recording a TV show through a digital video recorder
- Downloading software while running an anti-virus program
- "Anything that can be threaded today will map efficiently to multi-core"
- BUT: some applications difficult to parallelize

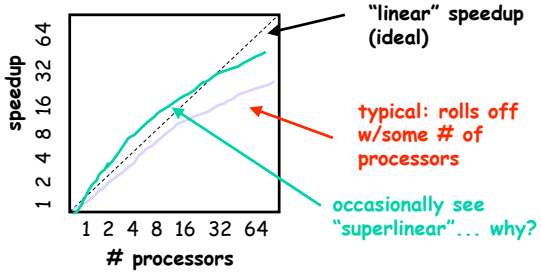
Ok, after all of that, what does parallel processing really do for performance?

Speedup

metric for performance on latency-sensitive applications

- Time(1) / Time(P) for P processors
  - note: must use the best *sequential* algorithm for Time(1) -- the parallel algorithm may be different.

B



Parallel Performance

- Serial sections
  - Very difficult to parallelize the entire app
  - Amdahl's law

$$\text{Speedup}_{\text{Overall}} = \frac{1}{(1-F_{\text{Parallel}}) + \frac{F_{\text{Parallel}}}{\text{Speedup}_{\text{Parallel}}}} \Rightarrow \begin{matrix}\text{Speedup}_{\text{Parallel}} = 1024 \\ F_{\text{Parallel}} = 0.5 \\ \text{Speedup}_{\text{Overall}} = 1.998\end{matrix} \quad \begin{matrix}\text{Speedup}_{\text{Parallel}} = 1024 \\ F_{\text{Parallel}} = 0.99 \\ \text{Speedup}_{\text{Overall}} = 91.2\end{matrix}$$

C Let's look at an example in context of CPI

Parallel Programming

- Parallel software is the problem
- Need to get significant performance improvement
  - Otherwise, just use a faster uniprocessor, since it's easier!
- Difficulties
  - Partitioning
  - Coordination
  - Communications overhead

Amdahl's Law

- Sequential part can limit speedup
- Example: 100 processors, 90× speedup?
  - $T_{new} = T_{parallelizable}/100 + T_{sequential}$
  - $$\text{Speedup} = \frac{1}{(1-F_{\text{parallelizable}}) + F_{\text{parallelizable}}/100} = 90$$
  - Solving: $F_{parallelizable} = 0.999$
- Need sequential part to be 0.1% of original time

Scaling Example

- Workload: sum of 10 scalars, and 10 × 10 matrix sum
  - Speed up from 10 to 100 processors
- Single processor: Time = (10 + 100) × t_{add}
- 10 processors
  - Time = 10 × t_{add} + 100/10 × t_{add} = 20 × t_{add}
  - Speedup = 110/20 = 5.5 (55% of potential)
- 100 processors
  - Time = 10 × t_{add} + 100/100 × t_{add} = 11 × t_{add}
  - Speedup = 110/11 = 10 (10% of potential)
- Assumes load can be balanced across processors

Scaling Example (cont)

- What if matrix size is 100 × 100?
- Single processor: Time = (10 + 10000) × t_{add}
- 10 processors
  - Time = 10 × t_{add} + 10000/10 × t_{add} = 1010 × t_{add}
  - Speedup = 10010/1010 = 9.9 (99% of potential)
- 100 processors
  - Time = 10 × t_{add} + 10000/100 × t_{add} = 110 × t_{add}
  - Speedup = 10010/110 = 91 (91% of potential)
- Assuming load balanced

Speedup Challenge

- To get full benefit of parallelism need to be able to parallelize the entire program!
- Amdahl's Law
  - $Time_{after}$ = ($Time_{affected}$/Improvement)+$Time_{unaffected}$
  - Example: We want 100 times speedup with 100 processors
  - $Time_{unaffected}$ = 0!!!

Cache Coherence Problem

- Shared memory easy with no caches
  - P1 writes, P2 can read
  - Only one copy of data exists (in memory)
- Caches store their own copies of the data
  - Those copies can easily get inconsistent
  - Classical example: adding to a sum
    - P1 loads allSum, adds its mySum, stores new allSum
    - P1's cache now has dirty data, but memory not updated
    - P2 loads allSum from memory, adds its mySum, stores allSum
    - P2's cache also has dirty data
    - Eventually P1 and P2's cached data will go to memory
    - Regardless of write-back order, the final value ends up wrong

Seems like lots of trouble.
Why do it?
Because we sort of have to…

Why multi-core ?

- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
  - heat problems
  - speed of light problems
  - difficult design and verification
  - large design teams necessary
  - server farms need expensive air-conditioning
- Many new applications are multithreaded
- General trend in computer architecture (shift towards more parallelism)

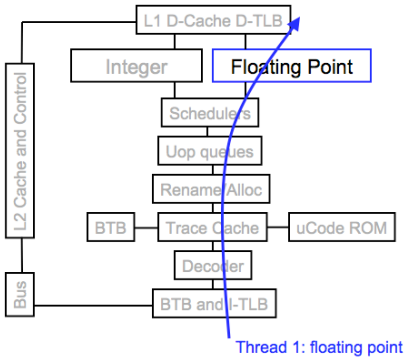
Let's look back to Lecture 01

There's another kind of parallelism too.

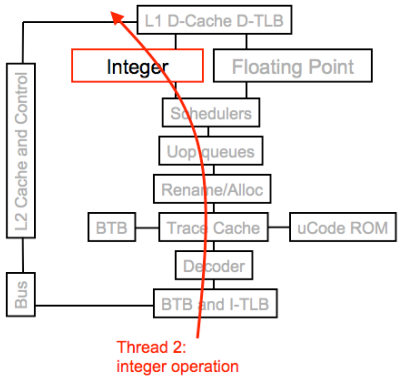
Simultaneous multithreading (SMT)

- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple "threads" on the same core
- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

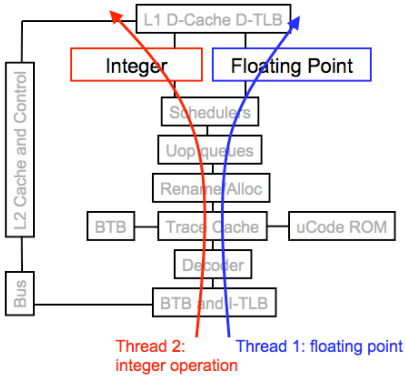
Without SMT, only a single thread can run at any given time



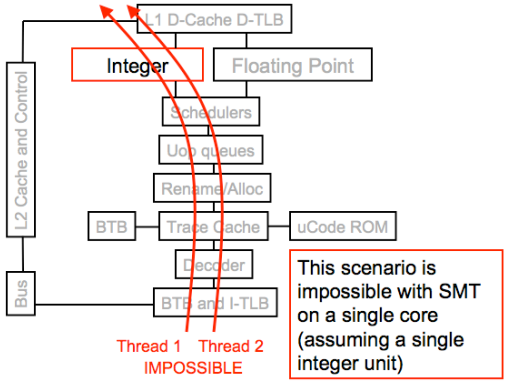
Without SMT, only a single thread can run at any given time



SMT processor: both threads can run concurrently



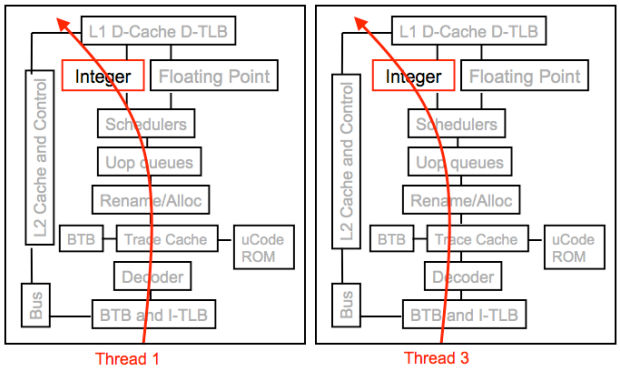
But: Can't simultaneously use the same functional unit



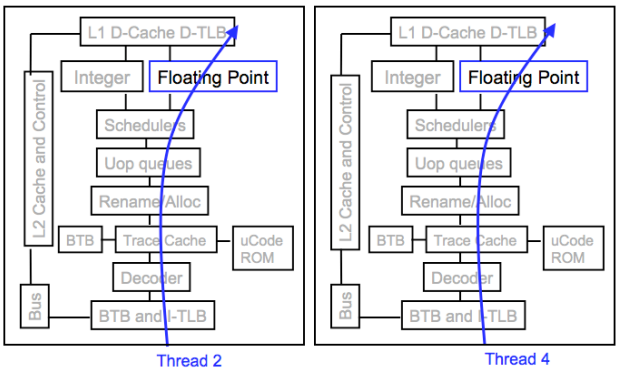
SMT not a "true" parallel processor

- Enables better threading (e.g. up to 30%)
- OS and applications perceive each simultaneous thread as a separate "virtual processor"
- The chip has only a single copy of each resource
- Compare to multi-core: each core has its own copy of resources

Multi-core: threads can run on separate cores



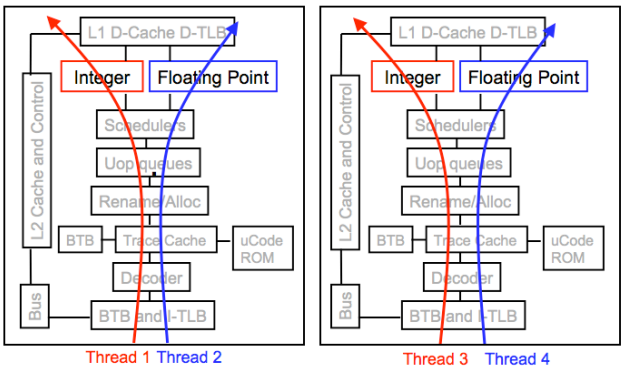
Multi-core: threads can run on separate cores



Combining Multi-core and SMT

- Cores can be SMT-enabled (or not)
- The different combinations:
  - Single-core, non-SMT: standard uniprocessor
  - Single-core, with SMT
  - Multi-core, non-SMT
  - Multi-core, with SMT:
- The number of SMT threads: 2, 4, or sometimes 8 simultaneous threads
- Intel calls them "hyper-threads"

SMT Dual-core: all four threads can run concurrently



Comparison: multi-core vs SMT

- Advantages/disadvantages?

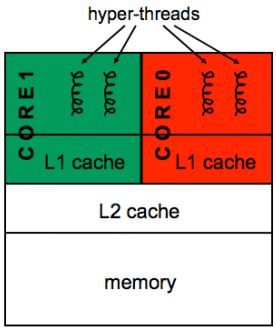
Comparison: multi-core vs SMT

- Multi-core:
  - Since there are several cores, each is smaller and not as powerful (but also easier to design and manufacture)
  - However, great with thread-level parallelism
- SMT
  - Can have one large and fast superscalar core
  - Great performance on a single thread
  - Mostly still only exploits instruction-level parallelism

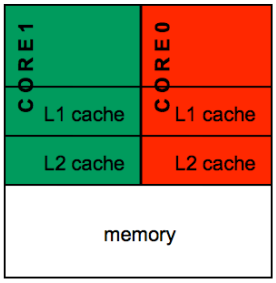
The memory hierarchy

- If simultaneous multithreading only:
  - all caches shared
- Multi-core chips:
  - L1 caches private
  - L2 caches private in some architectures and shared in others
- Memory is always shared

- Dual-core Intel Xeon processors
- Each core is hyper-threaded
- Private L1 caches
- Shared L2 caches

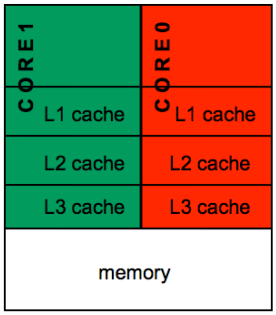


Designs with private L2 caches



Both L1 and L2 are private

Examples: AMD Opteron, AMD Athlon, Intel Pentium D



A design with L3 caches

Example: Intel Itanium 2

§7.5 Hardware Multithreading

Multithreading

- Performing multiple threads of execution in parallel
  - Replicate registers, PC, etc.
  - Fast switching between threads
- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed
- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn't hide short stalls (eg, data hazards)

Examples