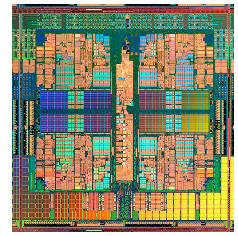
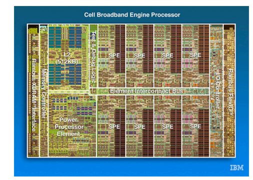
Lecture 28
Parallel Processing

General context: Multiprocessors

- Multiprocessor is any computer with several processors

Lemieux cluster, Pittsburgh supercomputing center





Multiprocessing

- Flynn's Taxonomy of Parallel Machines
  - How many Instruction streams?
  - How many Data streams?
- 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
  - Where is SIMD common?

Flynn's Taxonomy

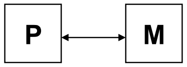
- MISD: Multiple I, Single D Stream
  - Not used much
- MIMD: Multiple I, Multiple D Streams
  - Each processor executes its own instructions and operates on its own data
  - This is your typical off-the-shelf multiprocessor (made using a bunch of "normal" processors)
    - Not superscalar
    - Each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

} What's superscalar?

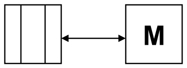
```
lw r2, 0(R1)       # Page fault
add r3, r2, r2     # waits
sub r6, r7, r8     # start?
```

In Pictures:

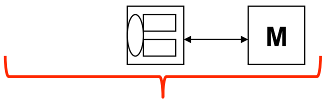
- Uni:



- Pipelined

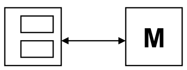


- Superscalar

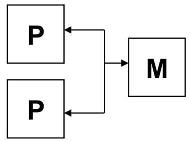


SISD

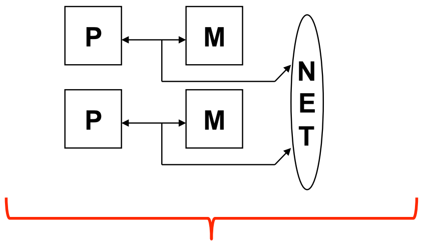
- VLIW/"EPIC"



- Centralized Shared Memory



- Distributed Shared Memory



MIMD

Multiprocessors

- Why did we need multiprocessors?
  - Uniprocessor speed improved fast
  - But there are things that needed even more speed
    - Wait for a few years for Moore's law to catch up?
    - Or use multiple processors and do it sooner?
    - (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

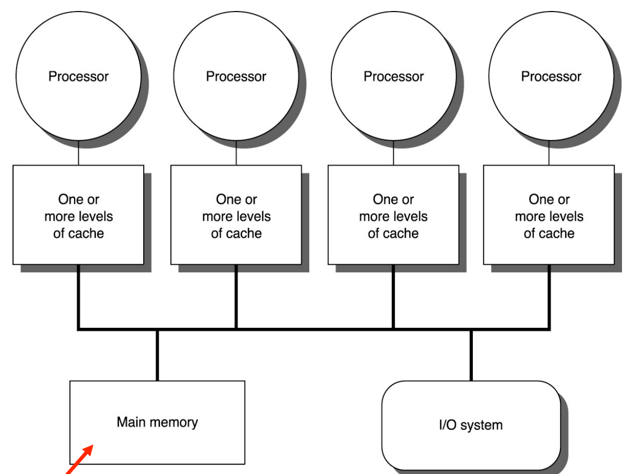
Let's look at a few MIMD example configurations…

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

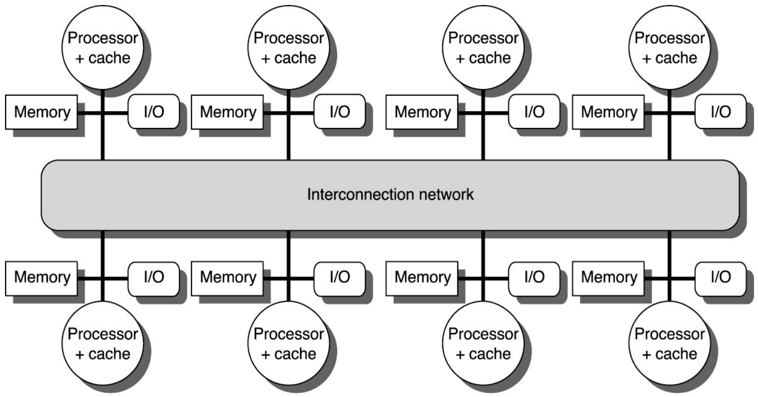
Centralized Shared Memory



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).

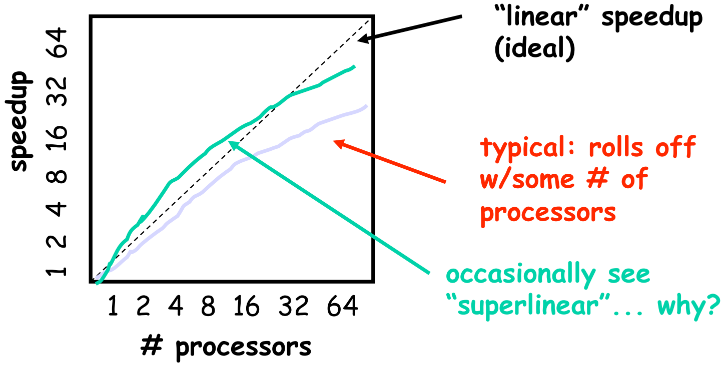
- Multi-core is a shared memory multiprocessor:
  All cores share the same memory

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.



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

See Examples 1 & 2

Other Problems: Cache Coherence

- 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, final value ends up wrong

If # of nodes so much as moderate, write-through not practical.

Other Problems: Contention

- Contention for access to shared resources - esp. memory banks or remote elements - may dominate overall system scalability
  - The problem:
    - Neither network technology nor chip memory bandwidth has grown at the same rate as the processor execution rate or data access demands
  - With success of Moore's Law:
    - Amt. of data per memory chip is growing such that it takes an increasing # of CCs to touch all bytes per chip at least once
      - Imposes a fundamental bound on system scalability
      - Is a significant contributor to single digit performance efficiencies by many of today's large scale apps.

Other Problems: Latency

- …is already a major source of performance degradation
  - Architecture charged with hiding local latency
    - (that's why we talked about registers, caches, IC, etc.)
  - Hiding global latency is task of programmer
    - (I.e. manual resource allocation)
- Today:
  - multiple clock cycles to cross chip
  - access to DRAM in 100s of CCs
  - round trip remote access in 1000s of CCs
- In spite of progress in NW technology, *increases* in clock rate may cause delays to reach 1,000,000s of CCs in worst cases

Other problems…

- Reliability:
  - Improving yield and achieving high "up time"
    - (think about how performance might suffer if one of the 1 million nodes fails every x number of seconds or minutes…)
  - Solve with checkpointing, other techniques
- Programming languages, environments, & methodologies:
  - Need simple semantics and syntax that can also expose computational properties to be exploited by large-scale architectures

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

Look back to Lecture 01

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)

Why parallel processing?

- Need more high performance supercomputing capability
  - FLOP History:
    - MegaFLOPS 1970s
    - GigaFLOPS 1980s
    - TeraFLOPS 1990s (1994)
    - PetaFLOPS (2010) (ish)
  - About 3 orders of magnitude every 12 years…
- Next target
  - ExaFLOPS (10^{18} FLOPS)
- Ultimate limit?
  - ZettaFLOPS (10^{21} FLOPS)
- Today's lecture: can we get to 10^{21} FLOPS? And Why?

Parallel processing enables solutions to important problems.

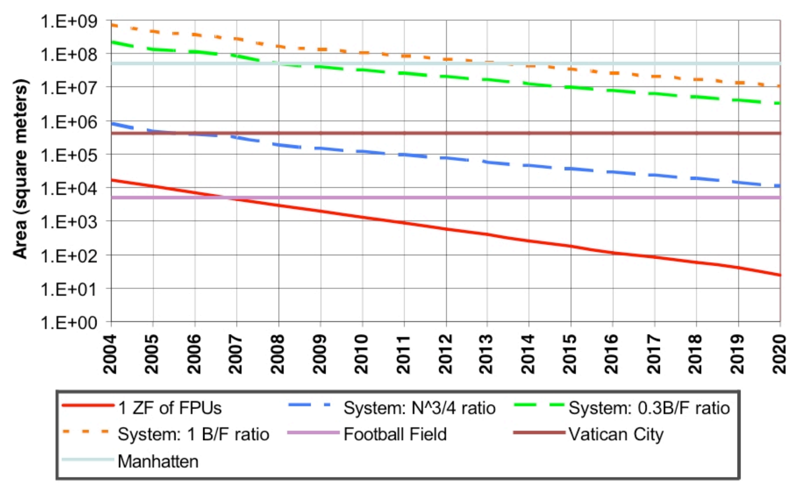
- Why zettaFLOPS?
  - More computational capacity needed for science, national defense, society as a whole…
- Good example:
  - Climate modeling…
    - Climate modelers want - actually should say *need* - 10^{21} FLOPS to do accurate modeling…
    - …Ideally with a program-based model…

A silicon zettaflop won't be easy.

(technology still can limit)

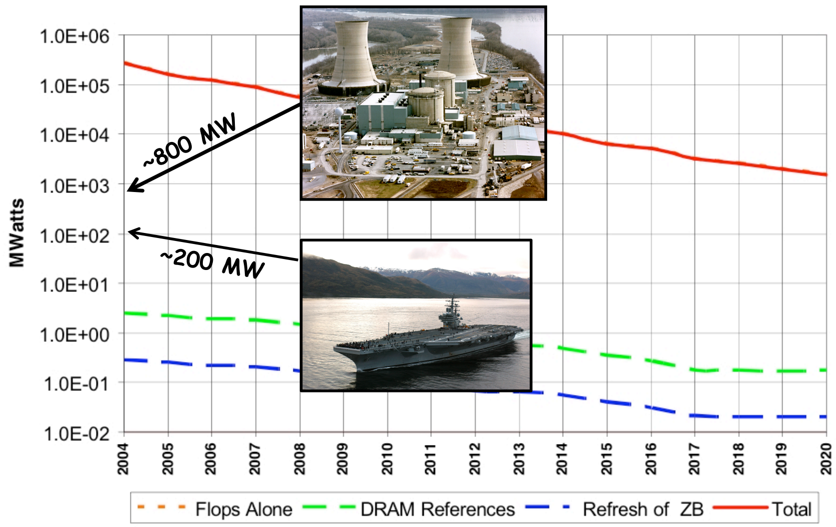
- As just mentioned, to get to a ZF, also need to consider storage?
- Thus, question #1: how much is enough?
  - Actually some debate about this…
    - Option 1: 1 ZB for 1 ZF
    - Option 2: 0.3 ZB / ZF
    - Option 3:
      - Gigabytes should equal at least sustained GF to the 3/4th power
      - This leads to 1 EB / ZF
  - We'll consider all of these…

It's big.



Note: this is the BEST case…

It's hot.



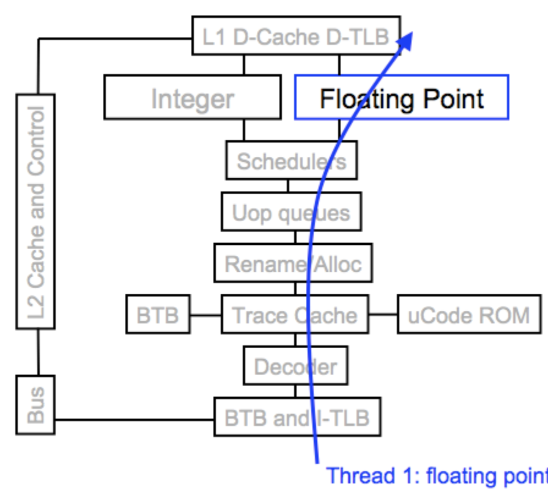
Another kind of parallelism

Simultaneous multithreading (SMT)

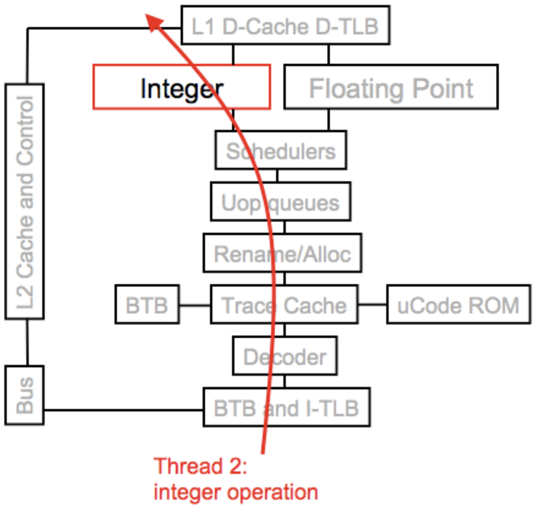
- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple "threads" on the same core

- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

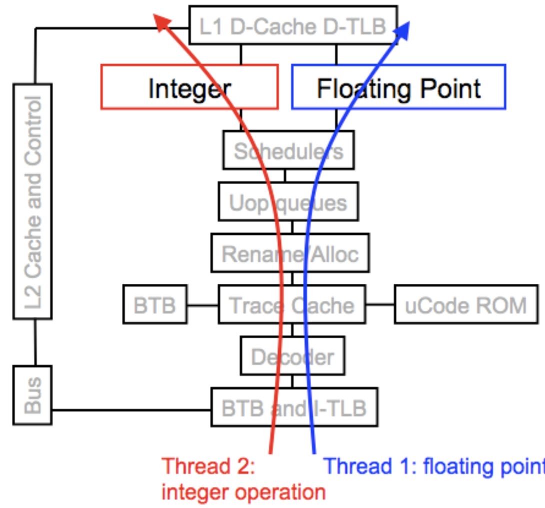
Without SMT, only a single thread can run at any given time



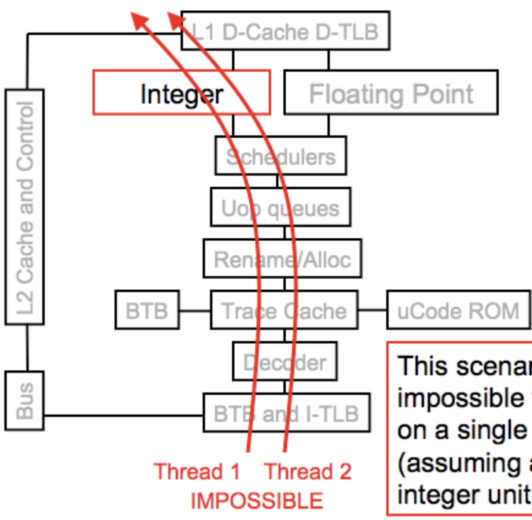
Without SMT, only a single thread can run at any given time



SMT processor: both threads can run concurrently



But: Can't simultaneously use the same functional unit



This scenario is impossible with SMT on a single core (assuming a single integer 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

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”

Comparison: multi-core vs SMT

- Advantages/disadvantages?

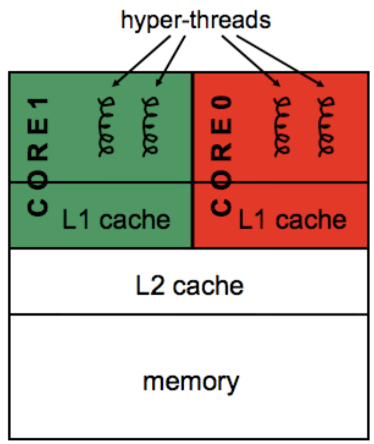
Comparison: multi-core vs SMT

- 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

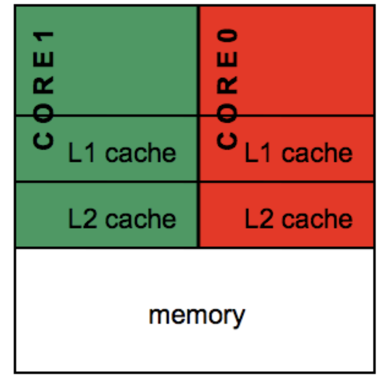
The memory hierarchy

- 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

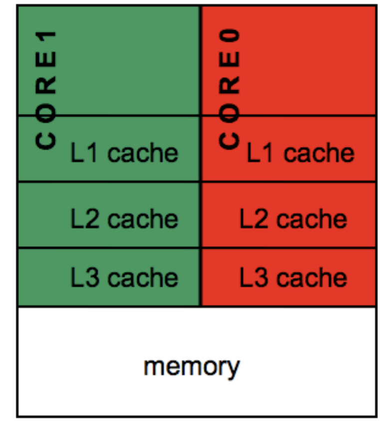


Designs with private L2 caches



Both L1 and L2 are private

Examples: AMD Opteron, AMD Athlon, Intel Pentium D



A design with L3 caches

Example: Intel Itanium 2

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)