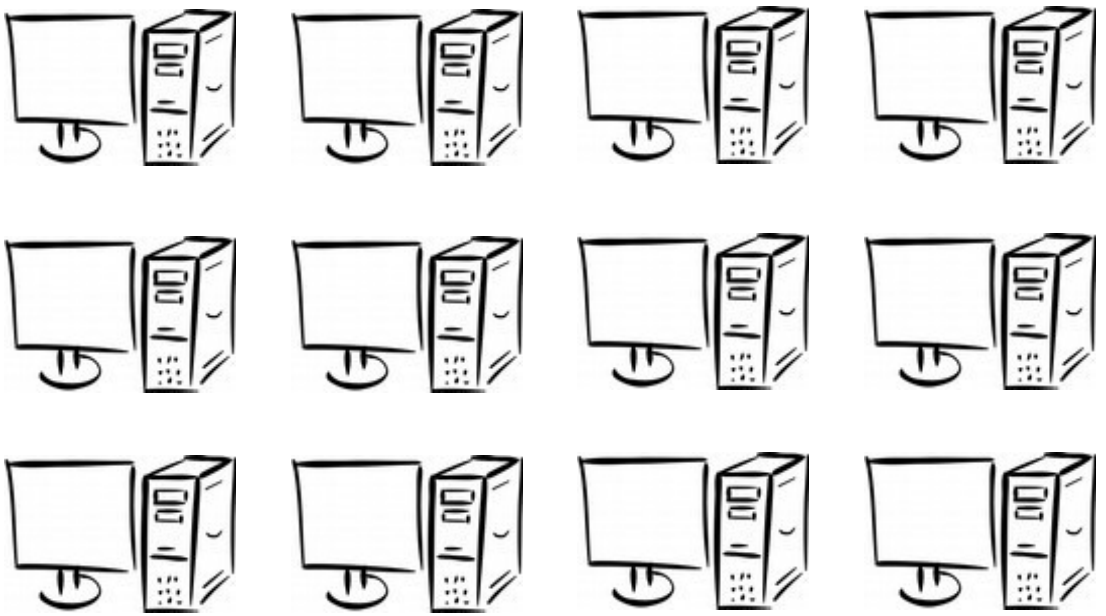
CS 470
Spring 2017

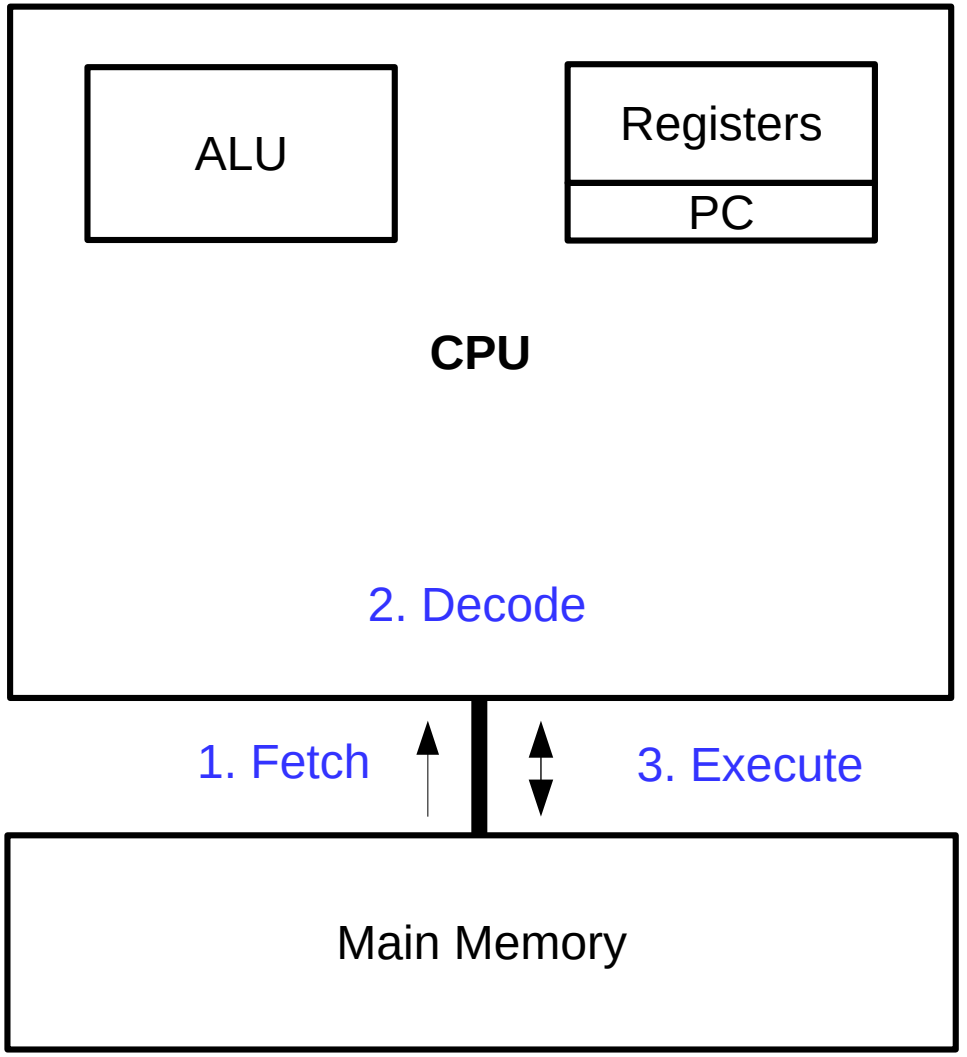
Mike Lam, Professor



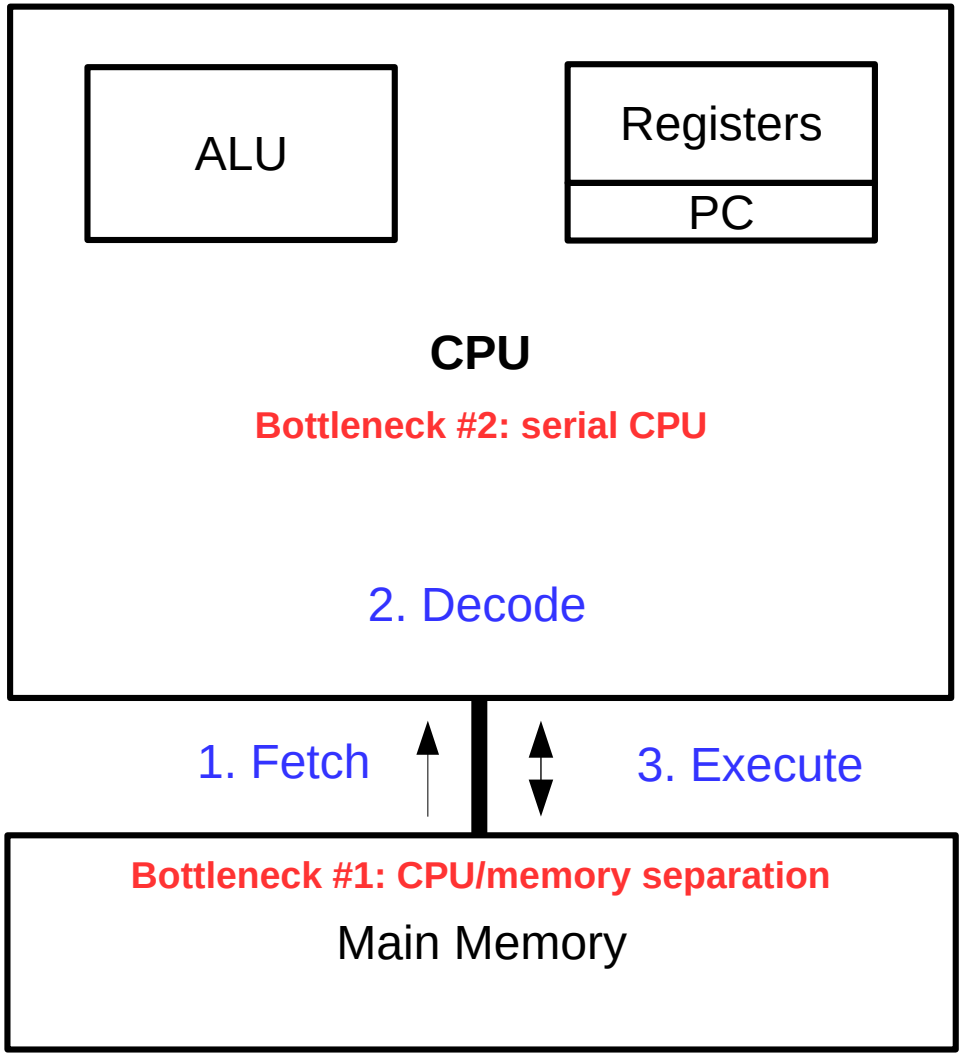
Parallel Computing

(a.k.a. "CS 470 in a nutshell")

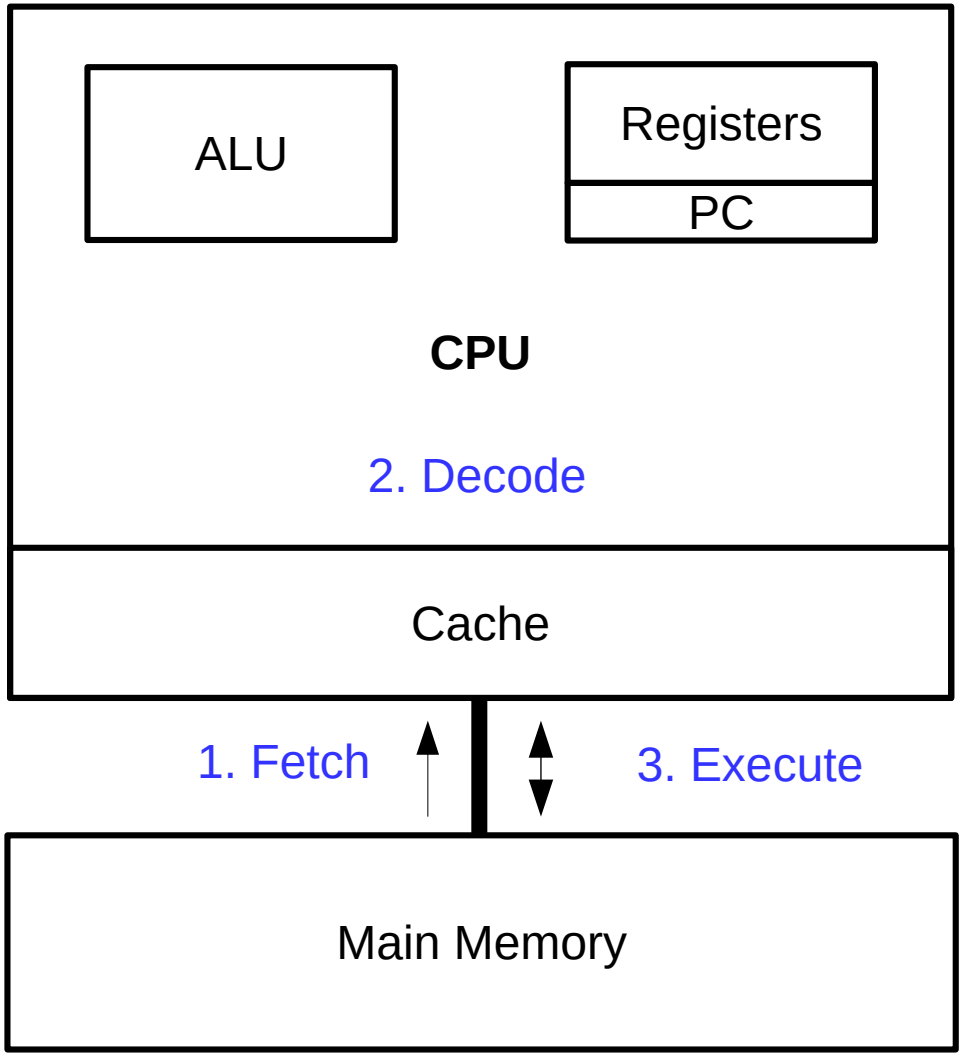
von Neumann architecture



von Neumann architecture



von Neumann architecture



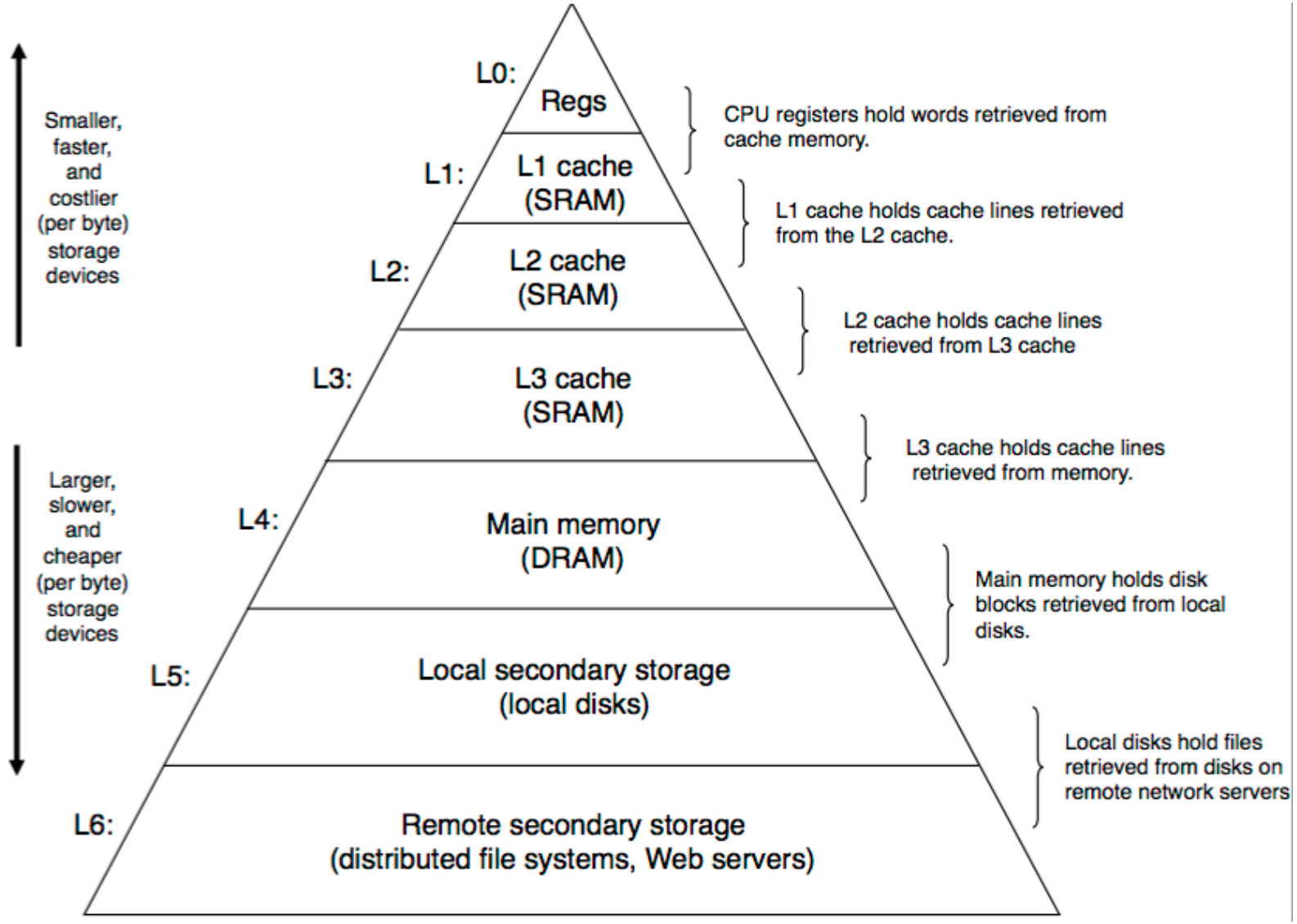
Locality

- Temporal locality: frequently-accessed items will continue to be accessed in the future
  - Theme: **repetition is common**
- Spatial locality: nearby addresses are more likely to be accessed soon
  - Theme: **sequential access is common**
- Why do we care?
  - *Programs with good locality can perform better*

Caching

- A cache is a small, fast memory that acts as a buffer or staging area for a larger, slower memory
  - Takes advantage of good locality
  - Data is transferred to/from the cache in blocks
    - For CPUs, also called cache lines
  - Cache hit vs. cache miss
    - Did the cache have to load from slower memory?
  - Cache misses require blocks to be replaced or evicted
    - Random replacement
    - Least recently used
    - Least frequently used

Memory hierarchy



Cache performance impact

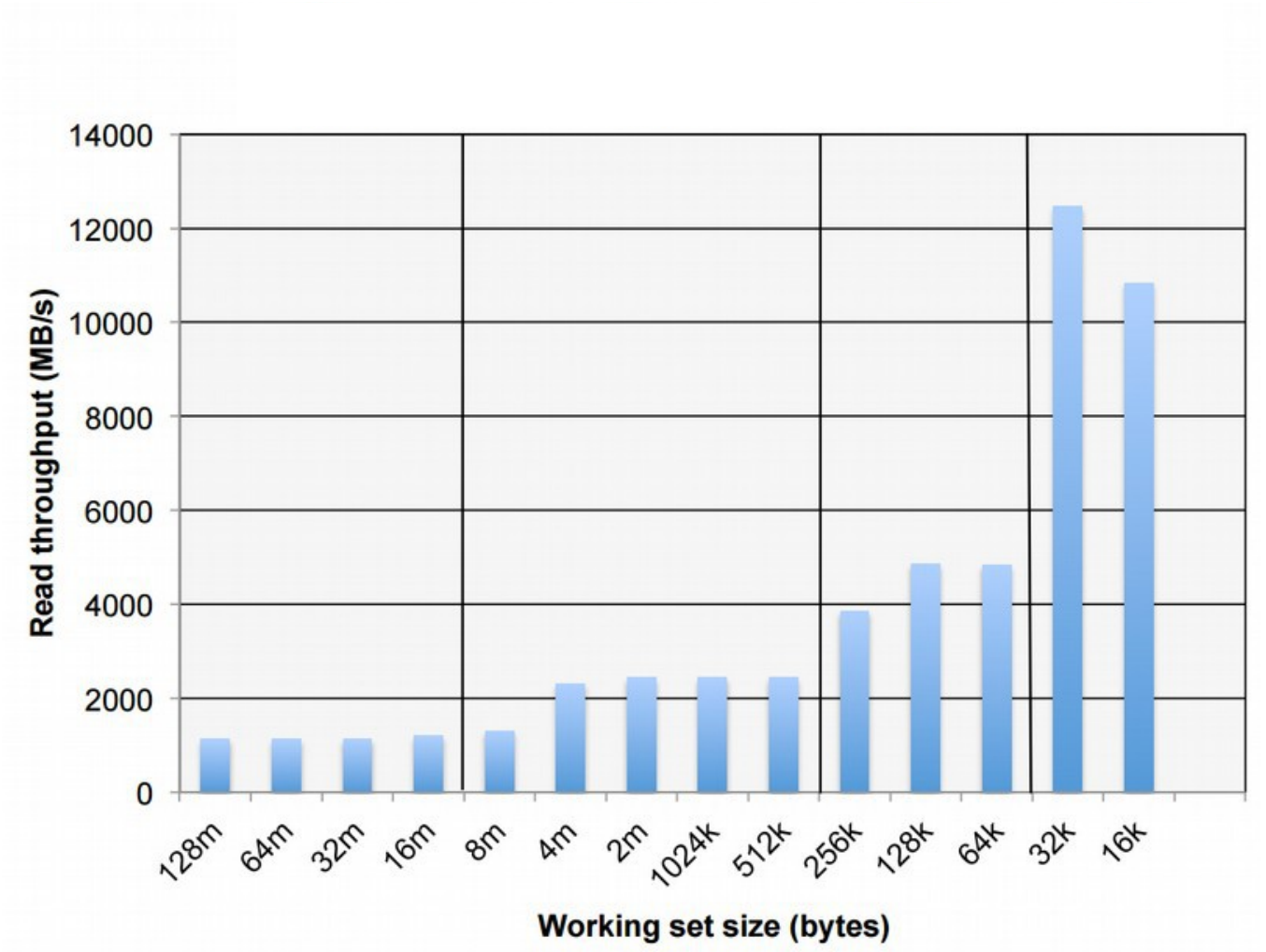
- Metrics
  - Miss rate: # misses / # memory accesses
  - Hit rate: 1 – miss rate
  - Hit time: delay in accessing data for a cache hit
  - Miss penalty: delay in loading data for a cache miss
  - Read throughput (or "bandwidth"): the rate that a program reads data from a memory system
- General observations:
  - Larger cache = higher hit rate but higher hit time
  - Lower miss rates = higher read throughput

Cache misses

- A cache always begins cold (empty)
  - Every request will be a miss initially
- As the cache loads data, it is warmed up
  - This effect can cause performance measurement variation during experiments if not controlled for
- A working set is a collection of elements needed repeatedly for a particular computation
  - If the working set doesn't fit in cache, this is called a capacity miss

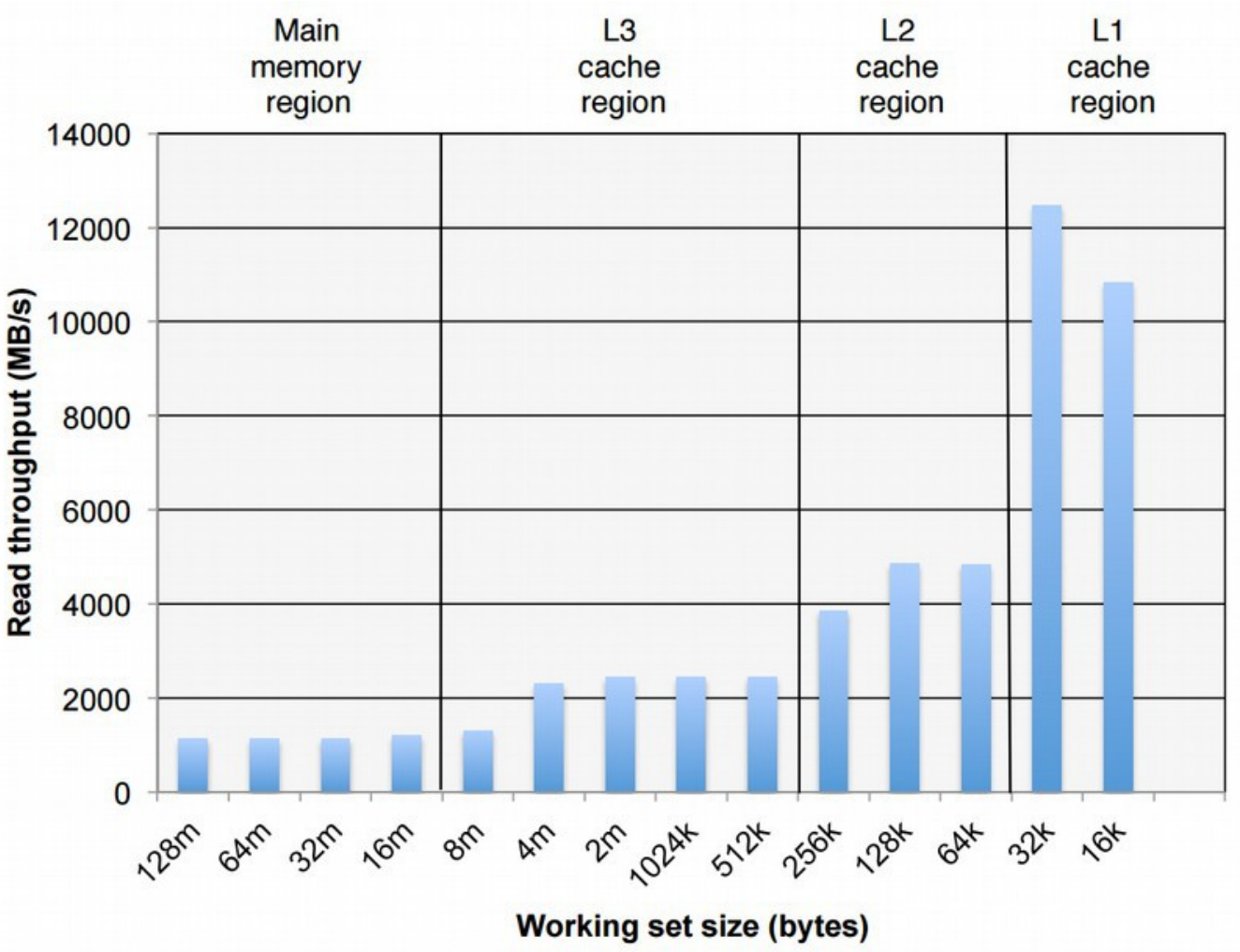
Temporal locality

- Working set size vs. throughput



Temporal locality

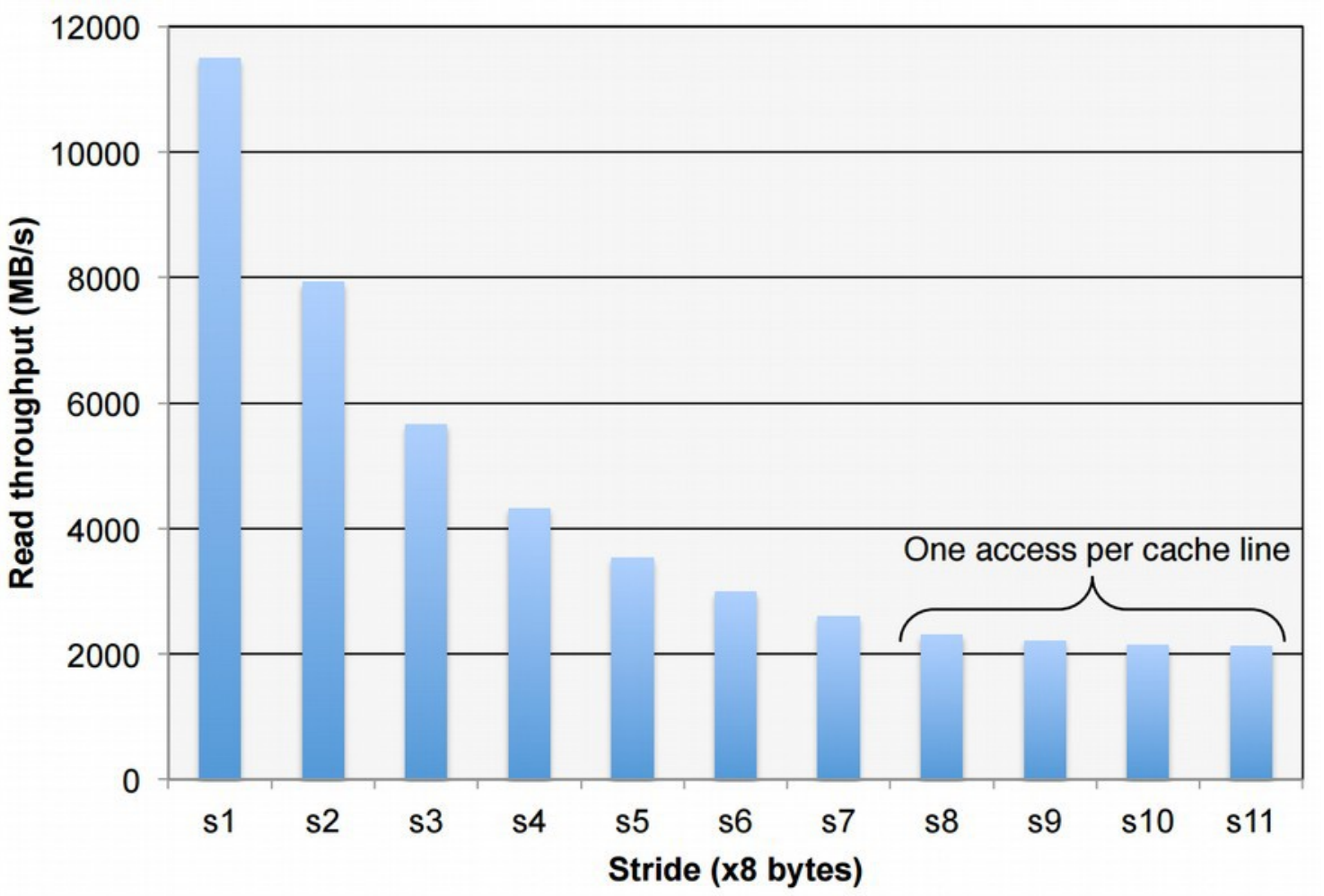
- Working set size vs. throughput



Possible source of superlinear speedup!

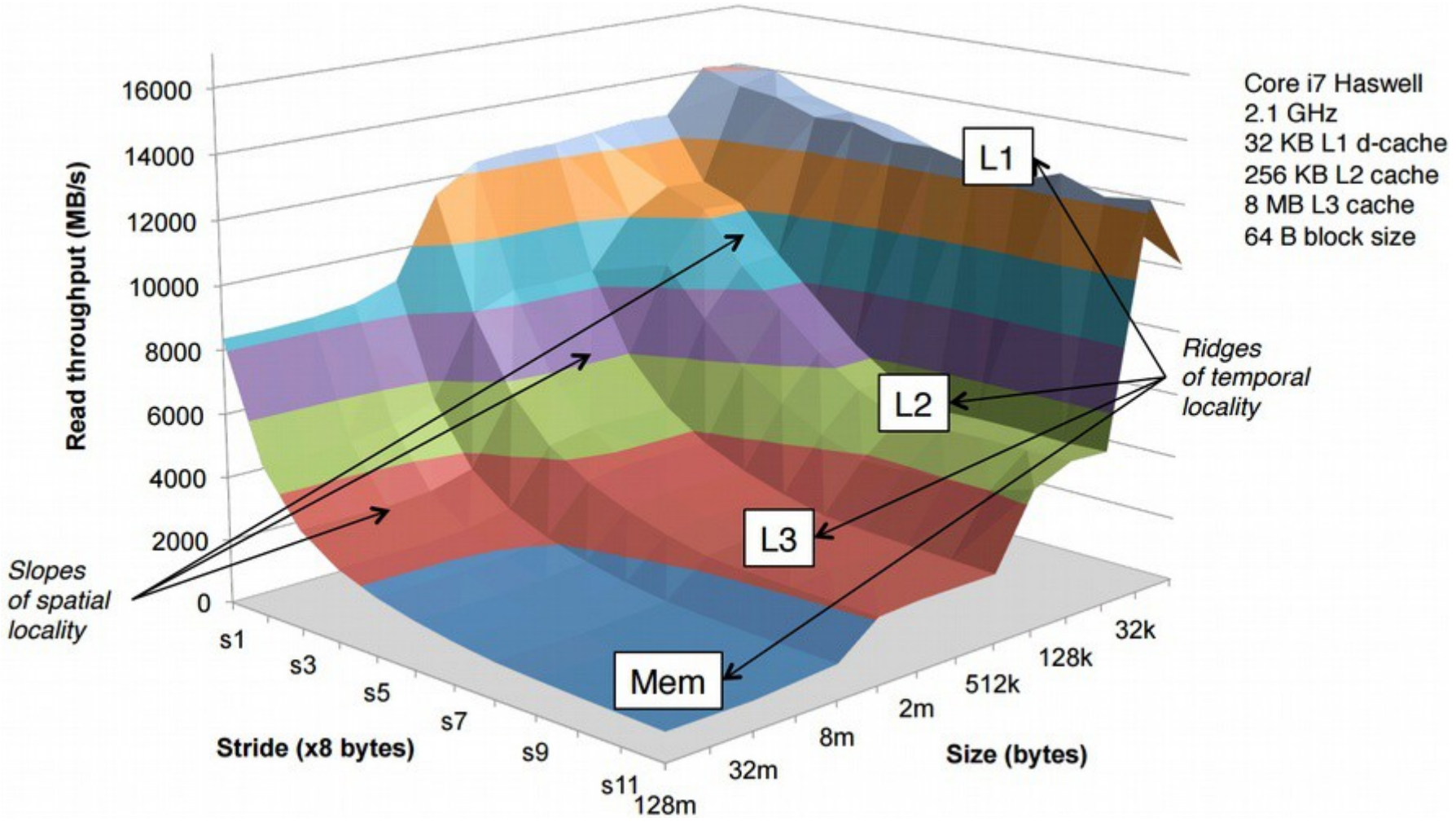
Spatial locality

- Stride vs. throughput



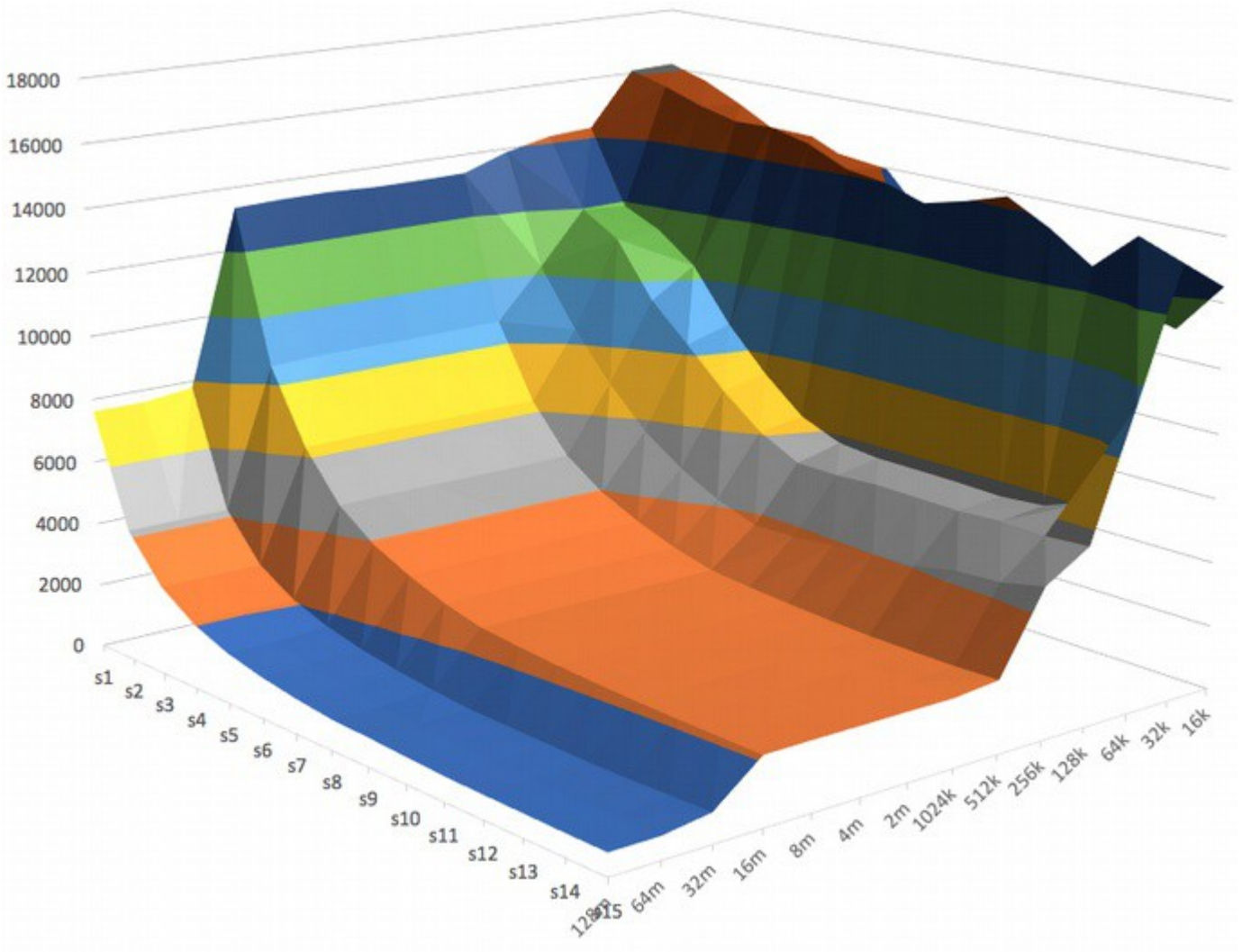
Memory mountain

- Stride and WSS vs. read throughput



Memory mountain

- Stride and WSS vs. read throughput

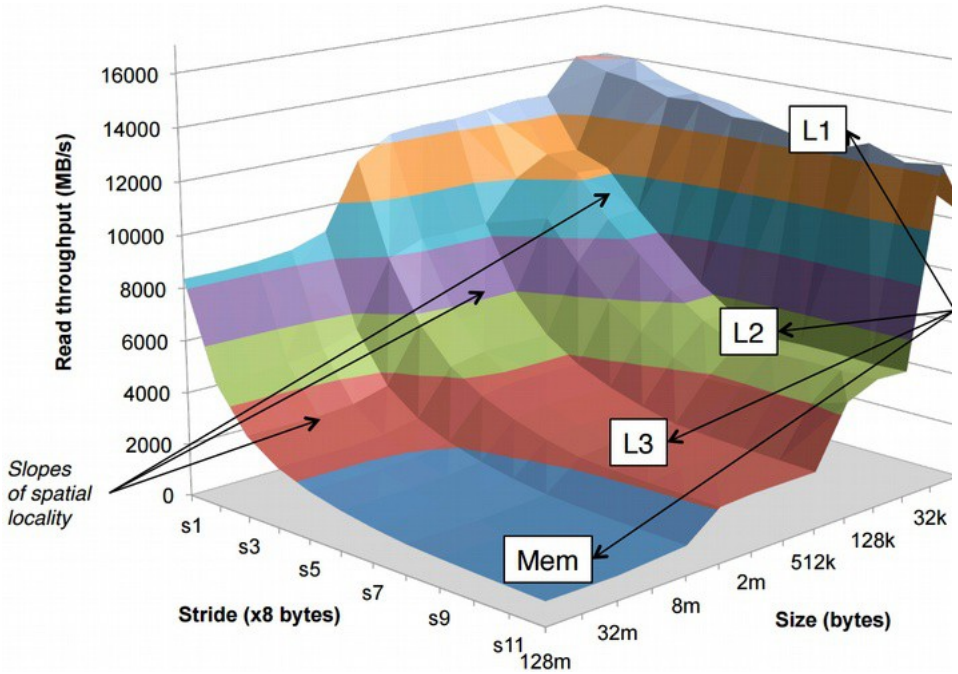


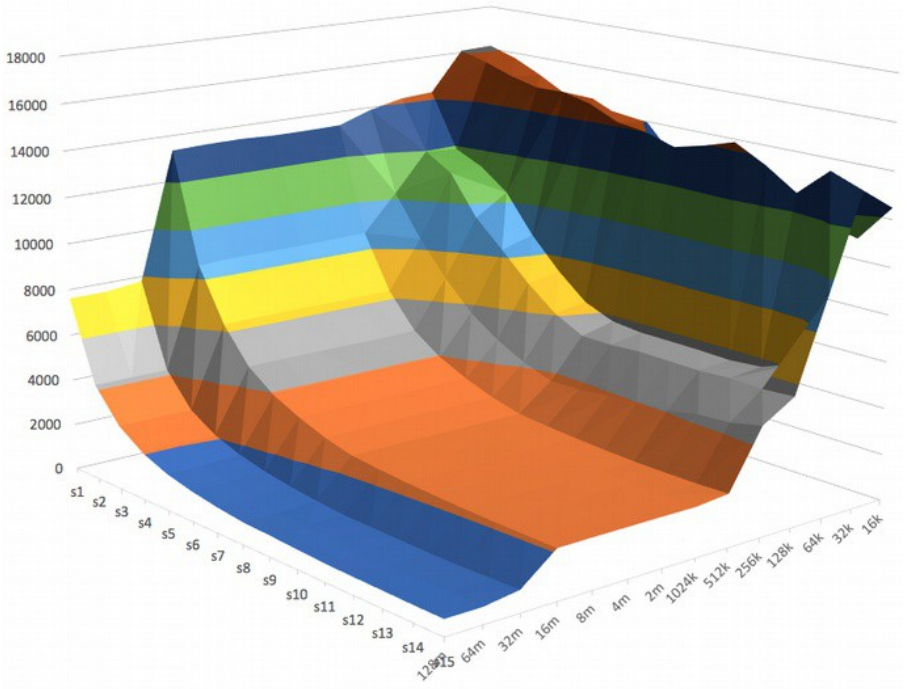
(our cluster)

Core i7 Haswell
2.4 GHz
32K L1 d-cache
256K L2 cache
20MB L3 cache
64 B block

Memory mountain

- **Theirs** (CS261 textbook) vs **ours** (CS470 cluster)

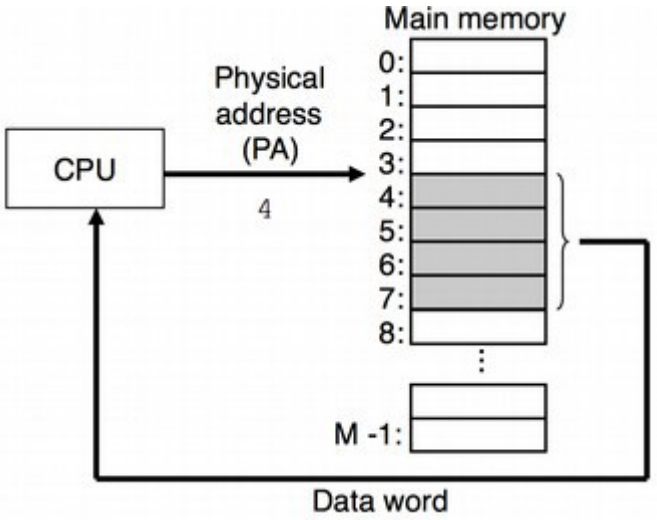




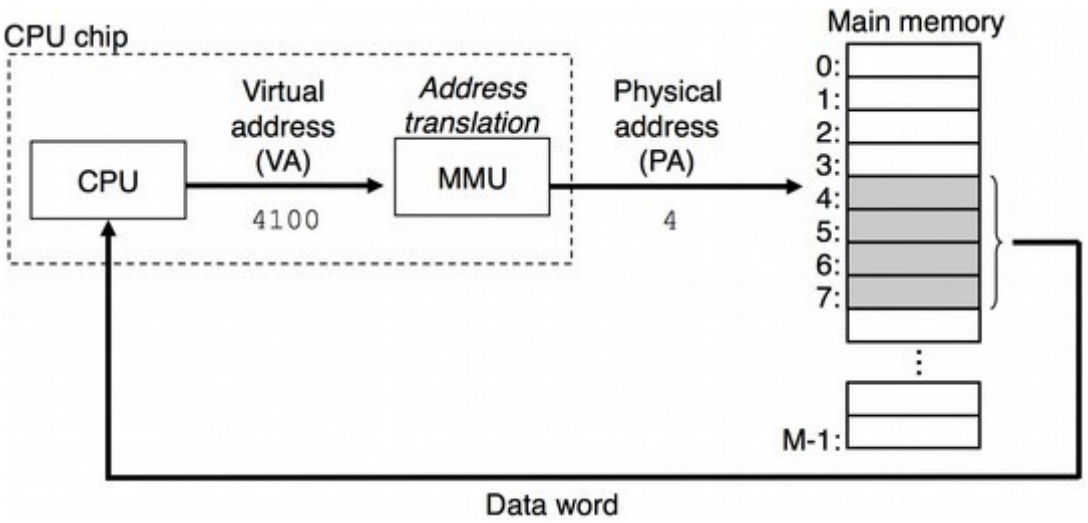
Test your own machine! `http://csapp.cs.cmu.edu/3e/mountain.tar`

Virtual memory

- Kernel translates between virtual and physical addresses
- Goals:
  - Use main memory as a cache for disks
  - Provide every process with a uniform view of memory
  - Protect processes from interference

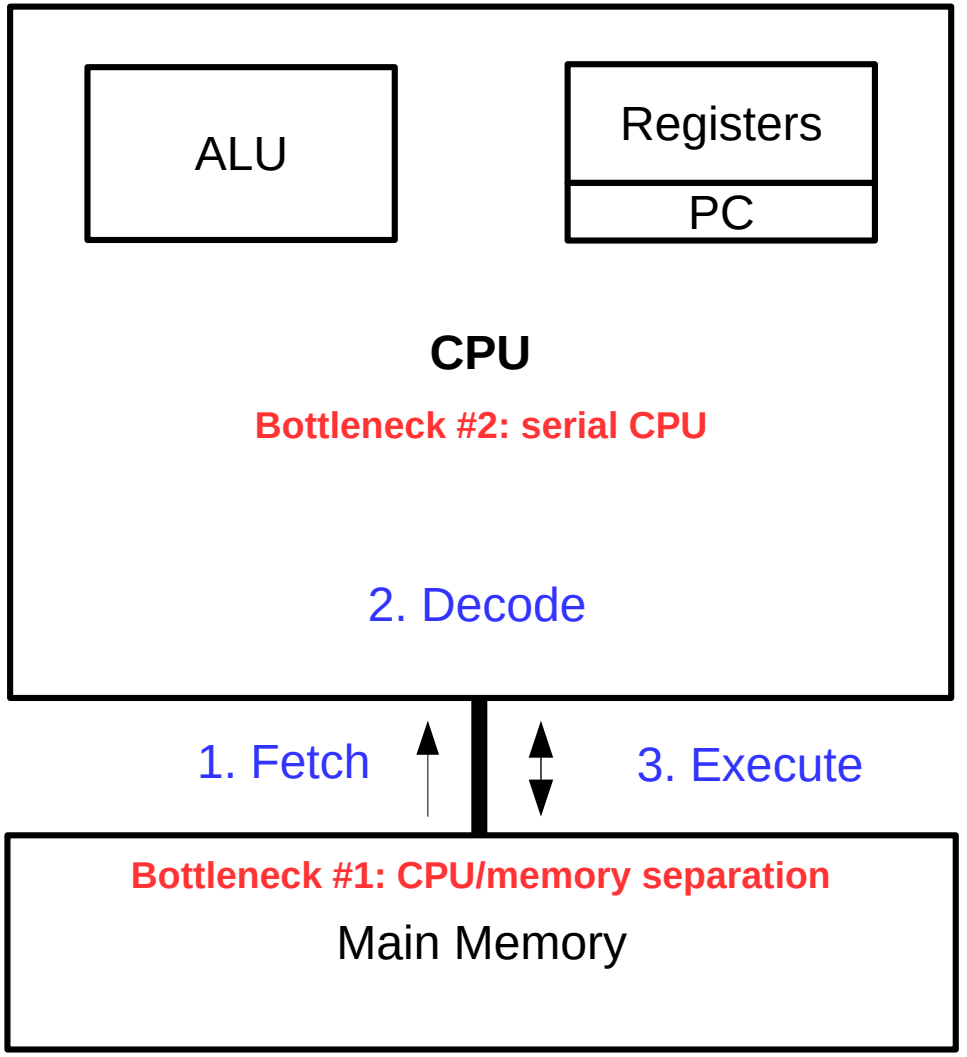


No virtual memory



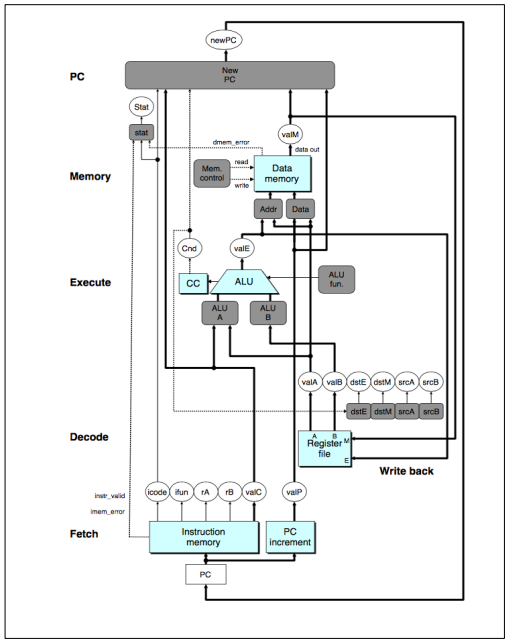
With virtual memory

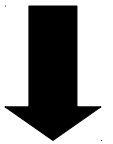
von Neumann architecture

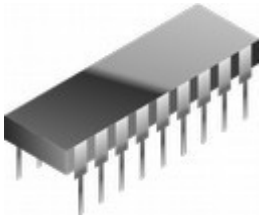


Instruction-level parallelism

- Original CPU design was serial
  - One instruction executes at a time
  - Only way to improve is to run faster!
  - Limited by speed of light
- One approach: make it smaller
  - Shorter circuit = faster circuit
  - Limited by manufacturing technology

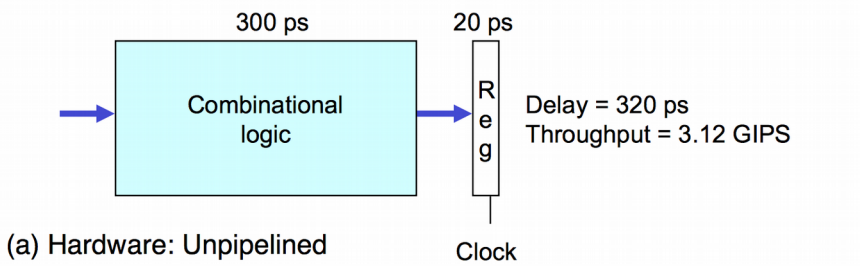




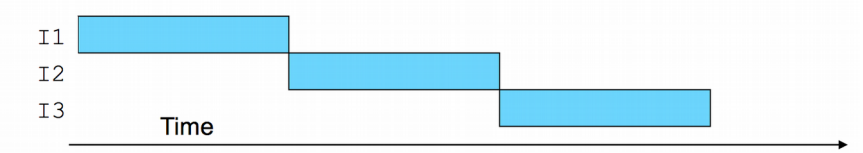


Instruction-level parallelism

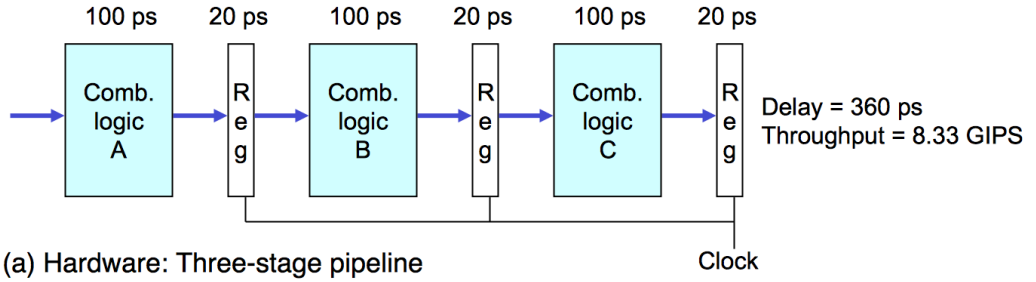
- Pipelined CPU design (multiple issue)
  - Multiple instructions execute simultaneously
  - Instruction-level parallelism at the core level (below threading)
  - Split CPU logic into stages and connect stages with clocked registers
  - Execution will stall if the next instruction cannot be executed yet



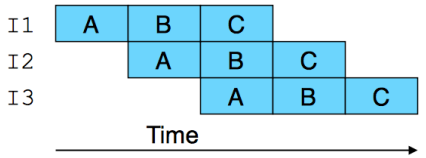
(a) Hardware: Unpipelined



(b) Pipeline diagram



(a) Hardware: Three-stage pipeline



(b) Pipeline diagram

Instruction-level parallelism

- Limitation: dependencies
  - The effect of one instruction depends on the result of another
  - Both data and control dependencies
  - Causes stalls and obstructs pipelining

Data dependency:

```
irmovq $8, %rax
addq %rax, %rbx
mrmovq 0x300(%rbx), %rdx
```

Control dependency:

```
loop:
    subq %rdx, %rbx
    jne loop
    irmovq $10, %rdx
```

Instruction-level parallelism

- Limitation: dependencies
  - The effect of one instruction depends on the result of another
  - Both data and control dependencies
  - Causes stalls and obstructs pipelining

Data dependency:

```
irmovq $8, %rax
addq %rax, %rbx
mrmovq 0x300(%rbx), %rdx
```

Control dependency:

```
loop:
    subq %rdx, %rbx
    jne loop
    irmovq $10, %rdx
```

Instruction-level parallelism

- Out-of-order execution
  - Increase saturation of instruction pipeline
  - Can alleviate data dependency stalls
  - Not always possible

Assumption: memory accesses require 3 cycles, all other instructions require 1 cycle

Original instructions: (**14 cycles**)

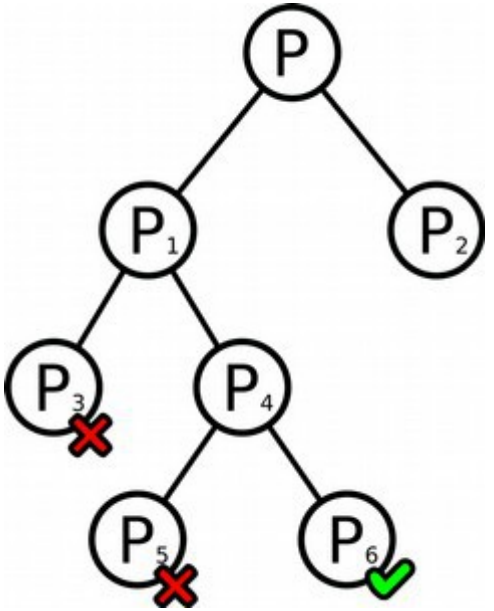
```
irmovq 0, %rsi
mrmovq 8(%rbp), %rax
addq %rax, %rsi
mrmovq 16(%rbp), %rcx
addq %rcx, %rsi
mrmovq 24(%rbp), %rdx
addq %rdx, %rsi
```

Out-of-order schedule: (**8 cycles**)

```
mrmovq 8(%rbp), %rax
mrmovq 16(%rbp), %rcx
mrmovq 24(%rbp), %rdx
irmovq 0, %rsi
addq %rax, %rsi
addq %rcx, %rsi
addq %rdx, %rsi
```

Instruction-level parallelism

- Speculative execution
  - Predict which branch will be taken
    - Roll back changes if we guessed wrong
  - Can alleviate control dependency stalls
  - High cost of incorrect prediction
  - Eager variant: execute **both** branches
    - Discard the wrong one once we figure it out



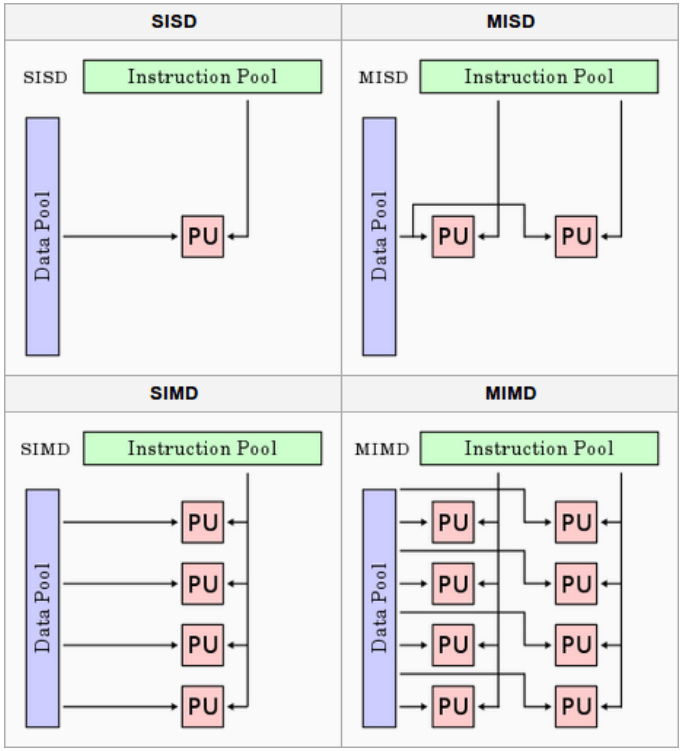
Beyond von Neumann

- These improvements can only get us so far
  - Need to move beyond single-CPU systems
  - Parallel computing: multiple simultaneous CPUs or threads/processes
  - Many different possible architectures

System architectures

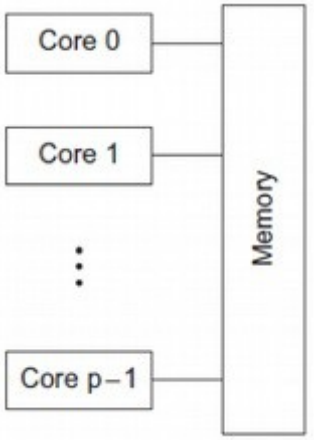
- Flynn's Taxonomy
  - Single Instruction, Single Data (SISD)
    - Traditional von Neumann
    - Increasingly insufficient!
  - Single Instruction, Multiple Data (SIMD)
    - Vector instructions (SSE/AVX)
    - GPUs and other accelerators
  - Multiple Instruction, Multiple Data (MIMD)
    - Single Program, Multiple Data (SPMD)
    - Shared memory
    - Distributed memory

Trend: higher number of slower, more energy-efficient processors

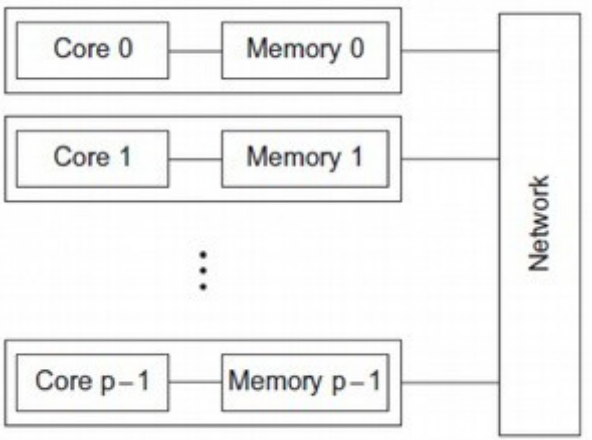


MIMD system architectures

- Shared memory
  - Idea: add more **CPUs**
  - Paradigm: threads
  - Technologies: Pthreads, OpenMP
  - Issue: synchronization



- Distributed memory
  - Idea: add more **computers**
  - Paradigm: message passing
  - Technologies: MPI, PGAS
  - Issue: data movement



Shared memory software

- Threading libraries
  - Low-level multiprocessing programming
  - Independent threads of execution; shared variables
  - Synchronization mechanisms (locks, semaphores, conditions, barriers)
    - Prevents data races and enforces thread safety
  - Libraries: Pthreads, Java Threads, Boost Threads

- Language extensions
  - Write one program that is both serial and parallel
  - Use pragmas to annotate the program with parallelism guidelines
  - Threading and synchronization added automatically (usually by compiler)
  - Languages: OpenMP, OpenACC

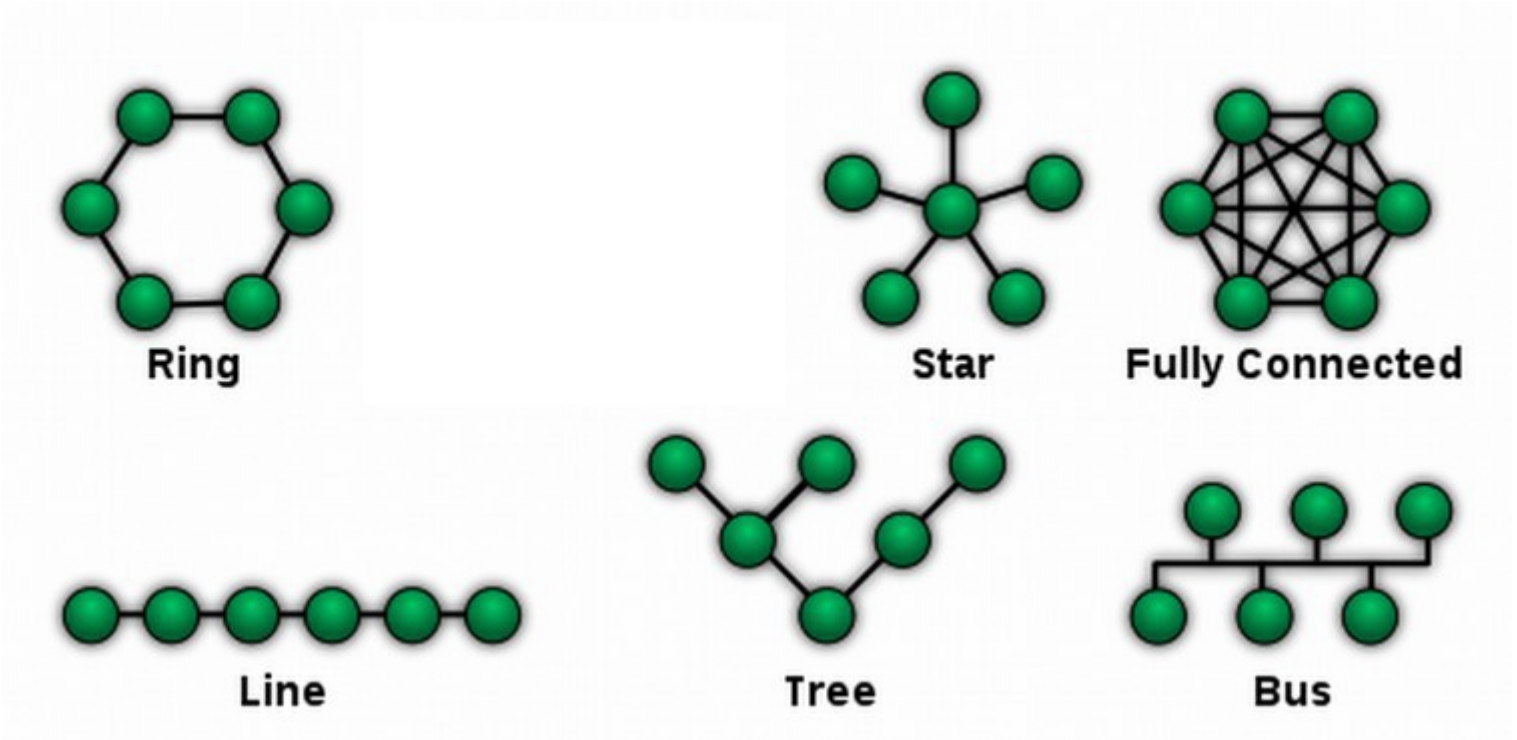
Distributed memory software

- Message-Passing Interface (MPI)
  - Low-level message-passing programming
  - Point-to-point operations (Send / Receive)
  - Collective operations (Broadcast / Reduce)
    - Allow MPI implementations to optimize data movement
  - Libraries: OpenMPI, MPICH, MVAPICH

- Partitioned Global Address Space (PGAS)
  - Make distributed memory look and act “like” shared memory
  - Split address space among all processes
  - Message passing is added automatically (usually by compiler)
  - Languages: Chapel, X10, Fortress

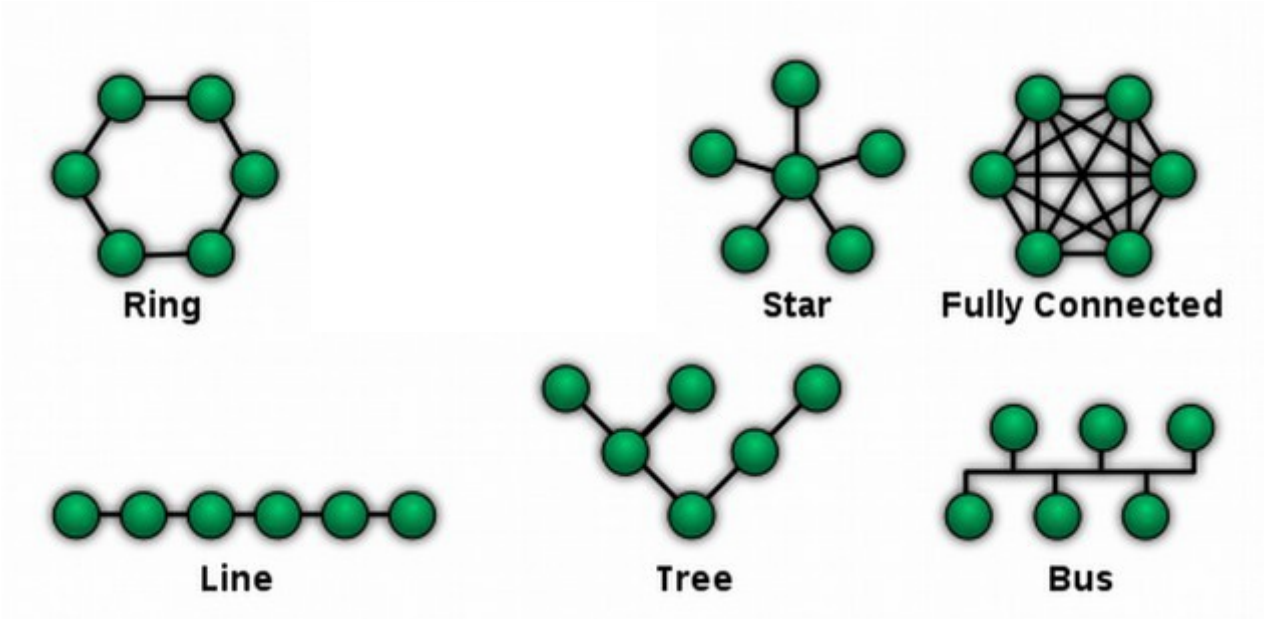
Distributed networks

- A topology is an arrangement of nodes in a system
  - Impacts communication efficiency in a distributed system
  - Non-uniform memory access (NUMA) costs



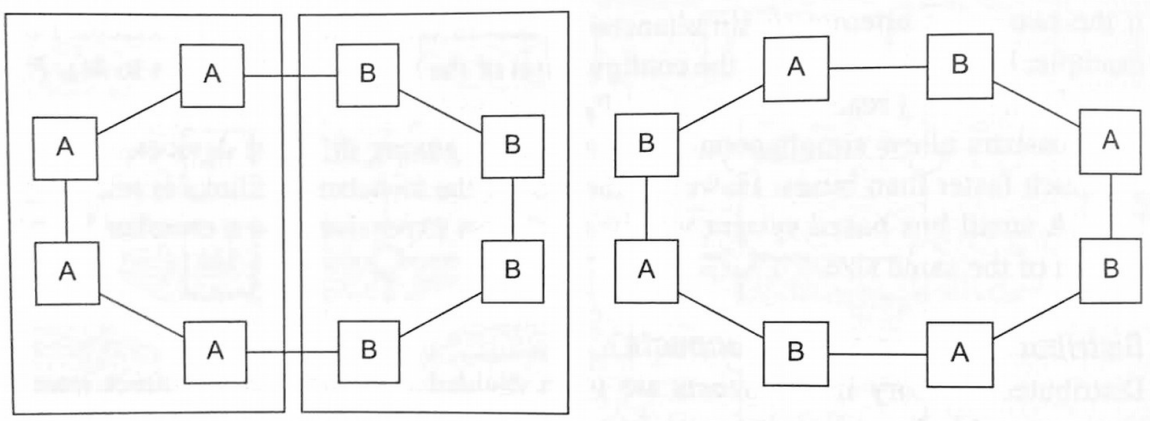
Distributed networks

- A topology is an arrangement of nodes in a system
  - Ring, star, line, and tree allow simultaneous connections but disallow some pairs of point-to-point communication
  - Fully connected and bus allow any-to-any communication but do not scale well



Distributed networks

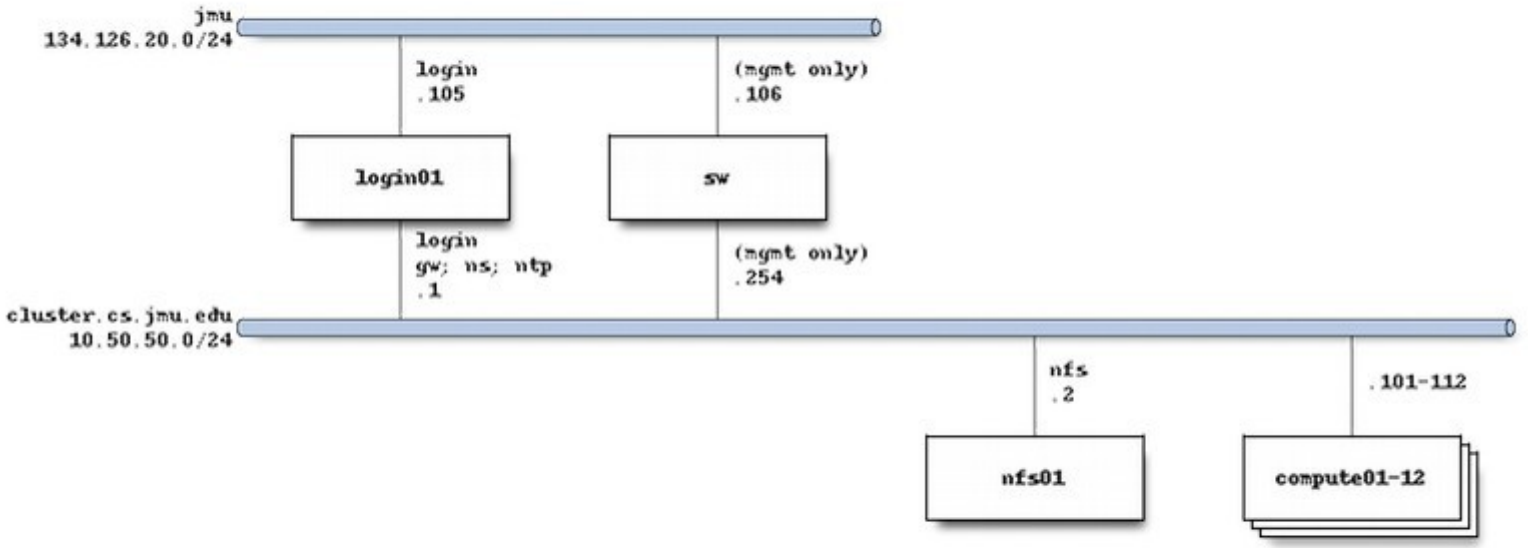
- Bandwidth: rate at which a link can transmit data
- Latency: time between send and receive
- Bisection: divide the network into two partitions
  - Bisection width: total possible simultaneous communications between the partitions
  - Bisection bandwidth: sum of bandwidth over all connections between the partitions
- **Important**: how do these metrics scale as you add nodes?



Two different bisections of a network

Our cluster

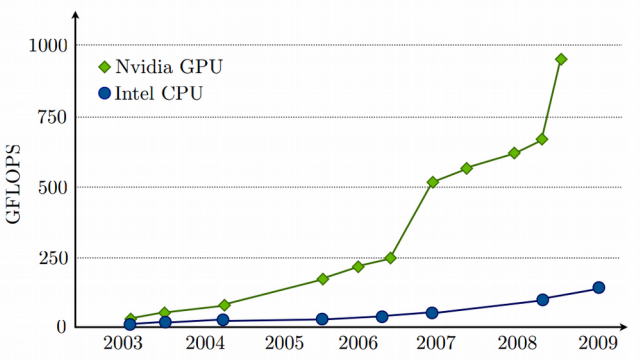
- **Compute nodes**: 12x Dell PowerEdge R430 w/ Xeon E5-2630v3 (8C, 2.4Ghz, HT) 32 GB
- **Login node**: Dell PowerEdge R430 w/ 2x Xeon E5-2630v3 (8C, 2.4Ghz, HT) 32 GB
- **File server**: Dell PowerEdge R730 w/ Xeon E5-2640v3 (8C, 2.6Ghz, HT) 32 GB
  - **Storage**: 8x 1.2TB 10K SAS HDD w/ RAID
- **Interconnect**: Dell N3024 Switch 24x1GbE, 2x10GbE SFP+ (212Gbps duplex)



Hybrid architectures

- Shared memory on the node
  - Hardware: CPU and coprocessor (e.g., GPU)
  - Enables energy-efficient strong scaling
  - Technologies: OpenMP, CUDA, OpenACC, OpenCL

- Distributed memory between nodes
  - Hardware: interconnect and distributed FS
  - Enables weak scaling w/ efficient I/O
  - Technologies: Infiniband, Lustre, HDFS, MPI



History of parallelism

- Uniprogramming / batch (1950s)
  - Traditional von Neumann, no parallelism
- Multiprogramming / time sharing (1960s)
  - Increased utilization, lower response time
- Multiprocessing / shared memory (1970s)
  - Increased throughput, strong scaling
- Distributed computing / distributed memory (1980s)
  - Larger problems, weak scaling
- Hybrid computing / heterogeneous (2000s)
  - Energy-efficient strong/weak scaling

Shared memory summary

- Shared memory systems can be very efficient
  - Low overhead for thread creation/switching
  - Uniform memory access times (symmetric multiprocessing)
- They also have significant issues
  - Limited scaling (# of cores) due to interconnect costs
  - Requires explicit thread management and synchronization
  - Caching problems can be difficult to diagnose
- Core design tradeoff: synchronization granularity
  - Higher granularity: simpler but slower
  - Lower granularity: more complex but faster
  - Paradigm: synchronization is expensive

Distributed memory summary

- Distributed systems can scale massively
  - Hundreds or thousands of nodes, petabytes of memory
  - Millions of cores, petaflops of computation capacity
- They also have significant issues
  - Non-uniform memory access (NUMA) costs
  - Requires explicit data movement between nodes
  - More difficult debugging and optimization
- Core design tradeoff: data distribution
  - How to partition and arrange the data; is any of it duplicated?
  - Goal: minimize data movement
  - Paradigm: computation is “free” but communication is not

Parallel & distributed systems

- Hardware architecture
- Software patterns & frameworks

First half of CS 470

- Interconnects, naming, and routing
- Clocks and synchronization
- Consistency and replication
- Fault tolerance and reliability
- File I/O and data archival
- Security

Second half of CS 470