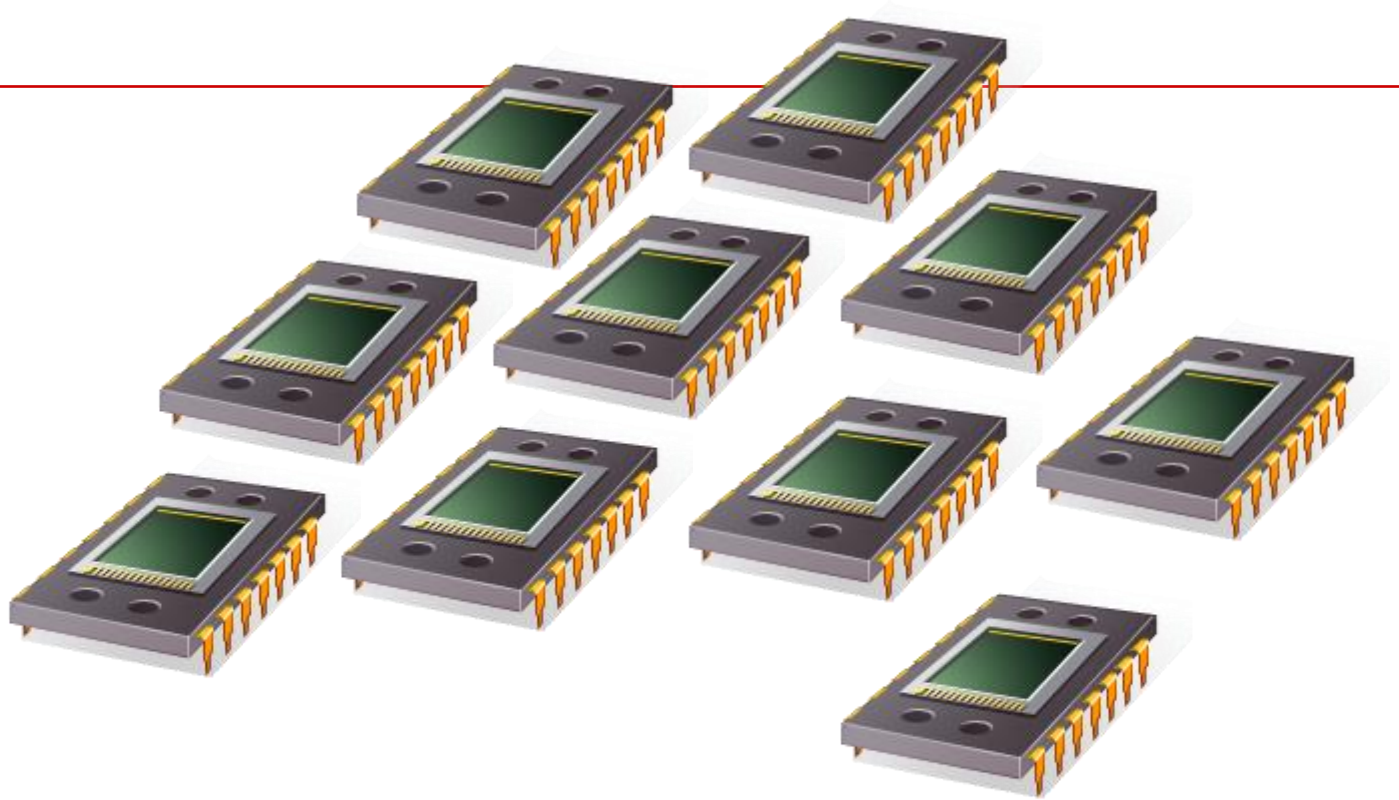


Parallel Architectures

UCSB CS140, T. Yang

Outline

- **Parallel architectures for high performance computing**
  - SIMD
  - Cluster computing and cloud
  - Shared memory architecture with cache coherence
- **Reference**
  - Chapter 2 of An Introduction to Parallel Programming" by Peter Pacheco, 2011, Morgan Kaufmann Publishers

Flynn’s Taxonomy

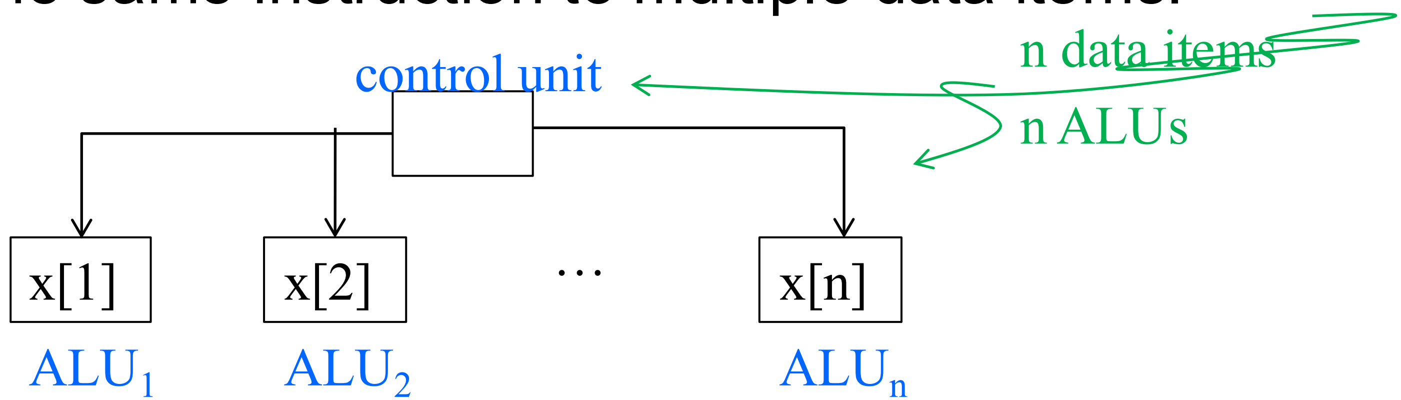
classic von Neumann

SISD Single instruction stream Single data stream	(SIMD) Single instruction stream Multiple data stream
MISD Multiple instruction stream Single data stream	(MIMD) Multiple instruction stream Multiple data stream

not covered

SIMD for Data Parallelism

- **Parallelism achieved by dividing data among processors.**
  - Applies the same instruction to multiple data items.



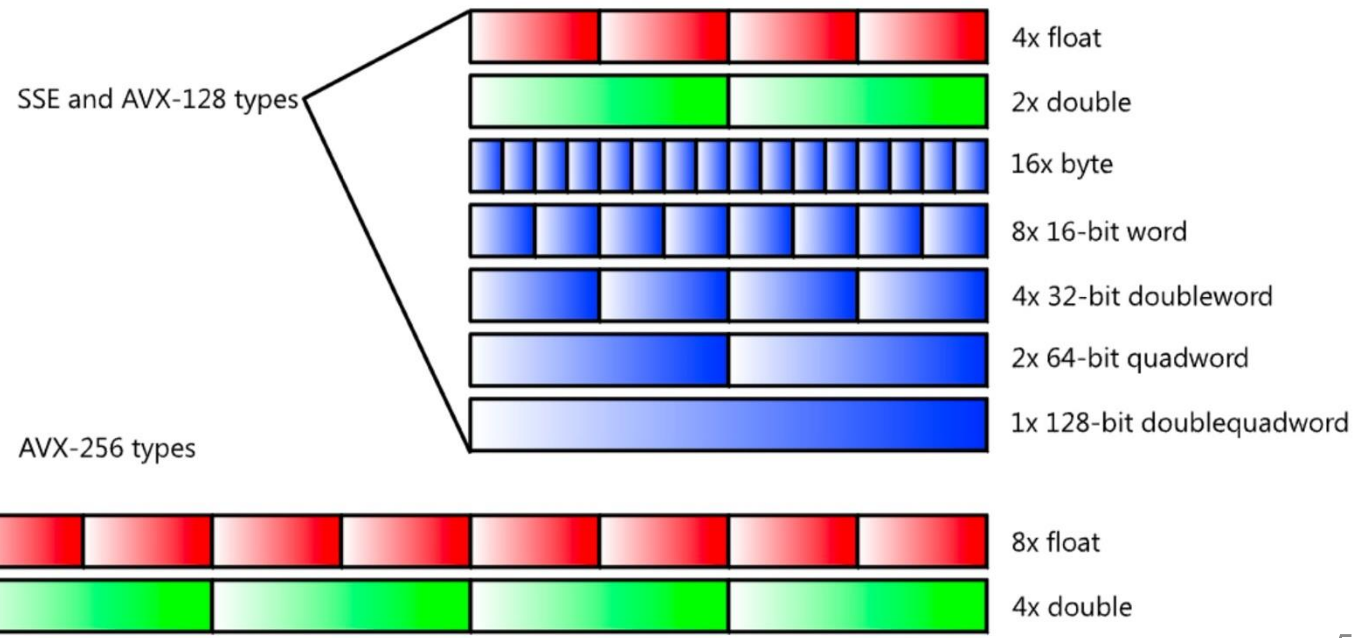

```
for (i = 0; i < n; i++)
    x[i] += y[i];
```

- **Drawbacks**
  - All ALUs are required to execute the same instruction simultaneously, or remain idle.
  - Efficient for large data parallel problems, but not flexible for more complex parallel problems.

Intel x86 SIMD Intrinsics

- Intrinsics are C functions and procedures for inserting SSE instructions into C code. Supported also in AMD CPUs
- **Data types of SSE registers**: __mm128i, __mm128, _mm128d

holding 4 32-bit integers, 4 32-bit single precision floats, and 2 64-bit double precision floats, respectively



SSE: 8 128-bit registers. **AVX2:** 16 256-bit registers. **AVX-512:** 32 512-bit registers

Use of Intel SIMD SSE Intrinsics

```
int x[4],  y[4];
__m128i sum4=__mm_setzero_si128();
__m128i tempx=_mm_loadu_si128 (& x[0]) ;
__m128i tempy=_mm_loadu_si128 (& y[0])
__m128i sum4 = __mm_add_epi32(tempx, tempy);
```

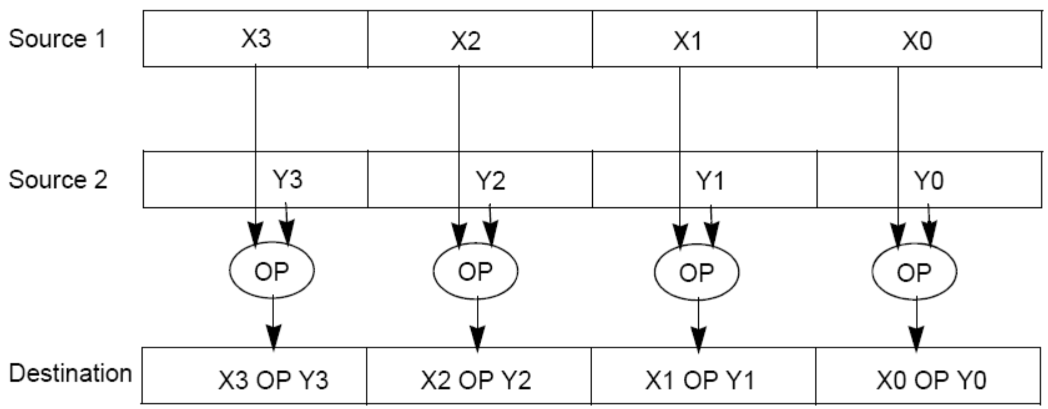
Load data from memory to a 128-bit register

Add 4 numbers in parallel

tempx

tempy

sum4



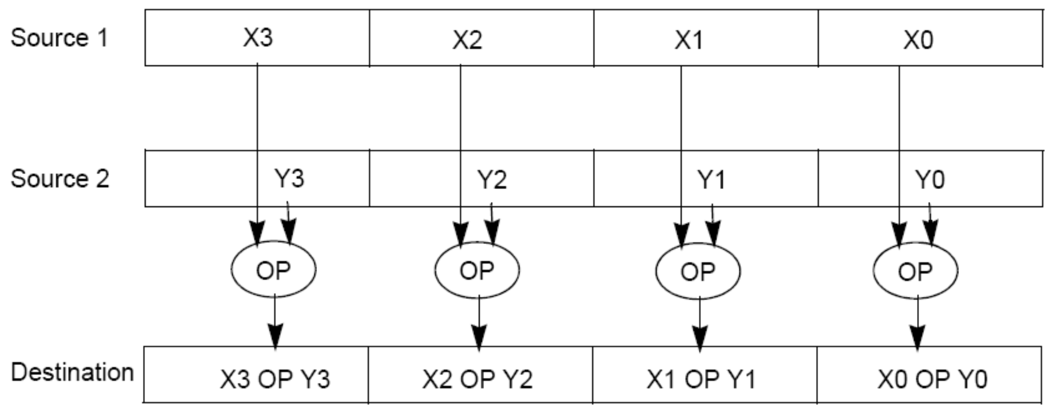
Arrays x and y may not be aligned with a 16-byte boundary in memory. Better SIMD performance if aligned during allocation

```
int x[4] __attribute__((aligned(16)));
```

AVX2 vectorization supports 8 number SIMD operations.

Related SSE 128-bit Intrinsics

__m128i _mm_setzero_si128( )	returns 128-bit zero vector
__m128i _mm_loadu_si128(__m128i *p)	Load data stored at pointer p of memory to a 128bit vector, returns this vector.
__m128i _mm_add_epi32(__m128i x, __m128i y)	returns vector (x_0+y_0, x_1+y_1, x_2+y_2, x_3+y_3) with 4 integers
void _mm_storeu_si128(__m128i *p, __m128i a)	stores content of 128-bit vector ”a” to memory starting at pointer p



Compiler Optimization with SIMD vectorization

Running time on CSIL	gcc	gcc -O	gcc –O2	gcc –O3
for (i=0; i<7780; i++) sum = sum+ a[i];	19µs	2.51µs	2.28µs	0.59µs
for (i=0;i <7780; i=i+4) Add a[i], a[i+1], a[i+2], a[i+3] with SIMD	8.9µs	0.54µs	0.50µs	0.49 µs

- **Optimization level of gcc compiler**
  - gcc → -O0 (default, no optimizations)
  - –O → -O1 (moderate optimization)
  - -02 → More optimization, e.g. SIMD vectorization
  - -03 → Aggressive optimization e.g. SIMD/loop unrolling
- Manual vectorization of code outperforms compiler optimization if the compiler cannot recognize data parallelism

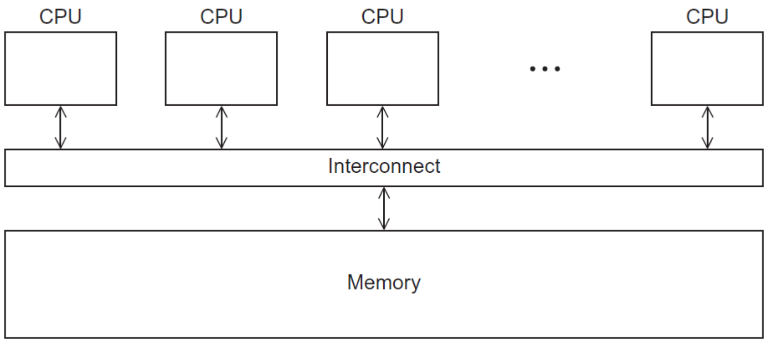
Graphics Processing Units (GPU)

- **Generalization from SIMD**
  - Single Instruction Multiple Threads (SIMT)
- GPU is popular for gaming and graphic applications, and now for AI/machine learning applications
- **Key Market Players: NVIDA, AMD, Intel**

MIMD

- Supports multiple simultaneous instruction streams operating on multiple data streams.
- Typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.
- **Types of MIMD systems**
  - **Shared-memory systems**
    - Most popular ones use multicore processors.
      - (multiple CPU’s or cores on a single chip)
  - **Distributed-memory systems**
    - Computer clusters are the most popular

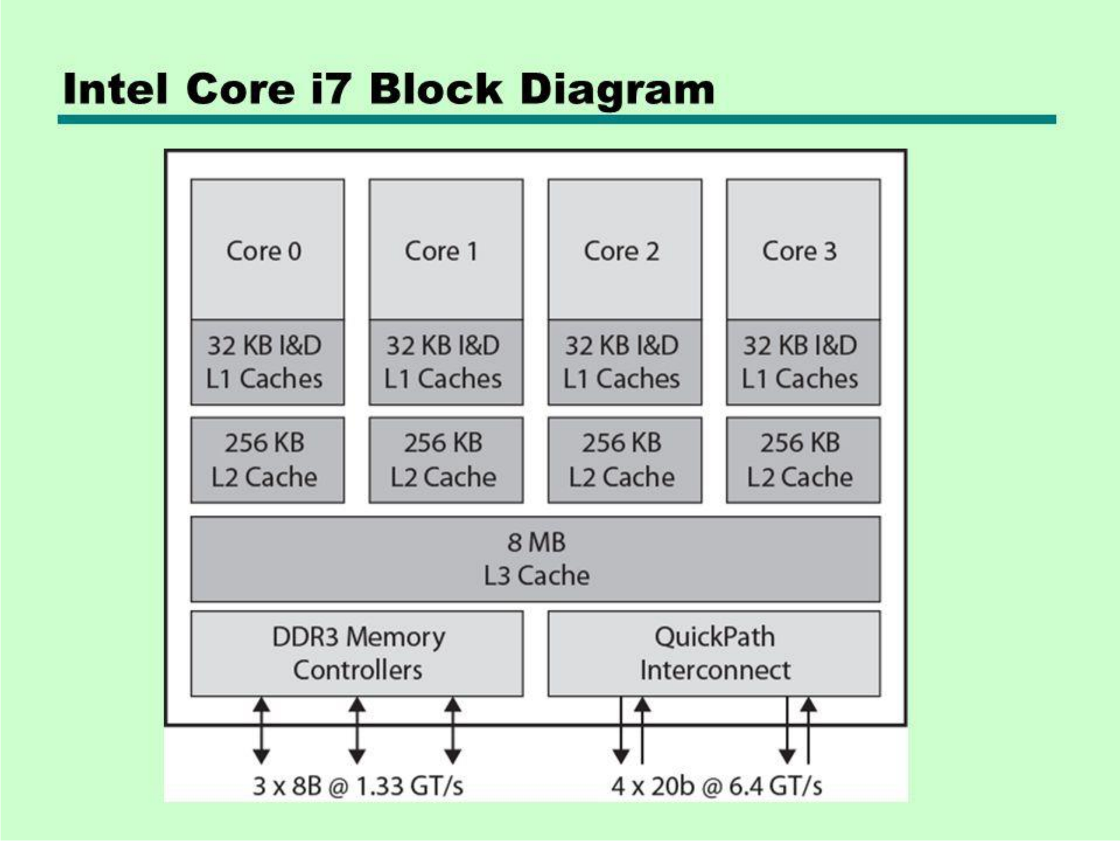
Shared Memory Systems



Each processor can access each memory location.

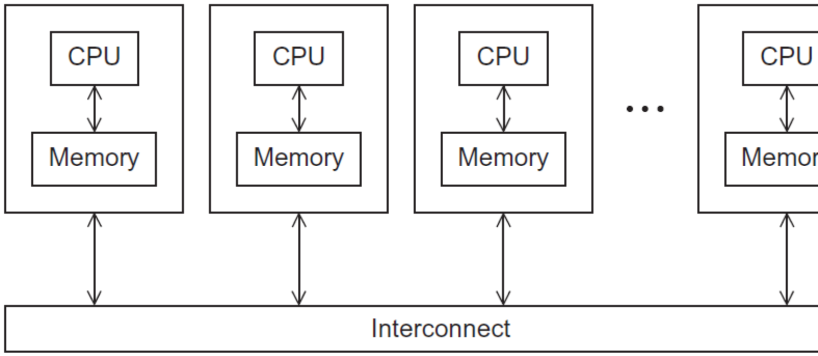
- Processors communicate implicitly by accessing shared data structures

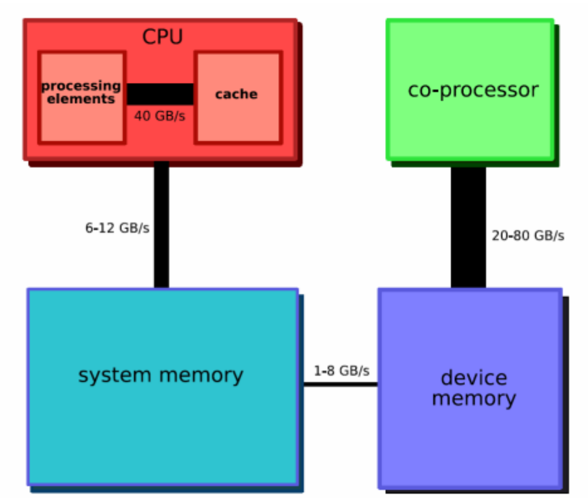
Example



Distributed Memory Systems

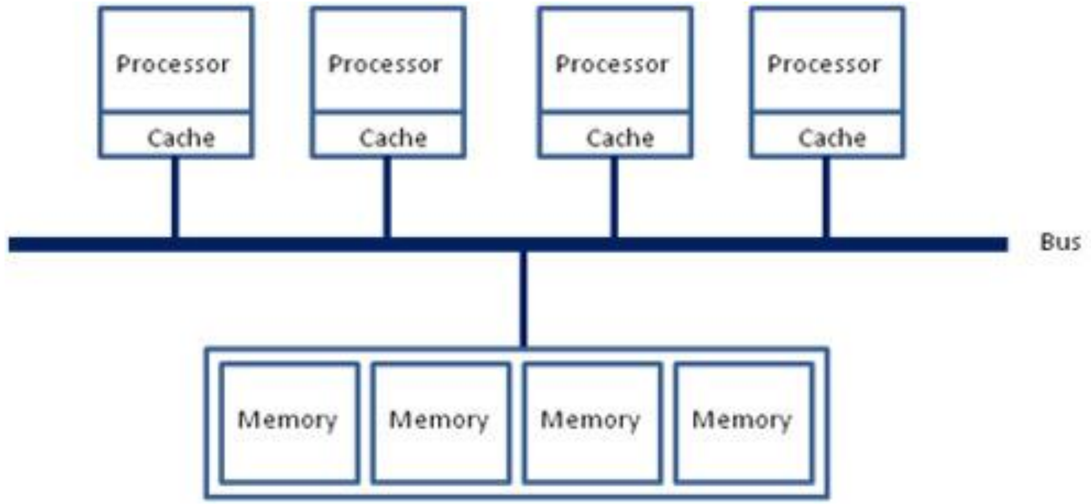
- **Clusters (most popular)**
  - A collection of commodity systems.
  - Connected by a commodity interconnection network.
  - Each node may contain both CPU/GPUs



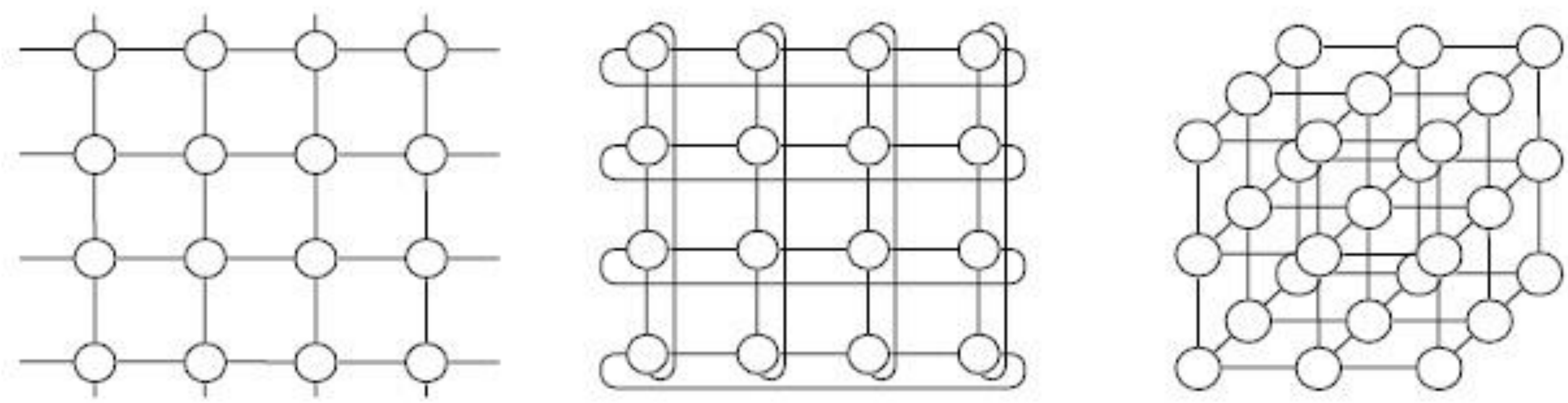


Interconnection networks

- **Two categories:**
  - Shared memory interconnects
  - Distributed memory interconnects
- **Shared memory interconnects: bus**
  - Parallel communication wires together with some hardware that controls access to the bus.
  - As the number of devices connected to the bus increases, contention for shared bus use increases, and performance decreases.



Examples of distributed memory interconnects

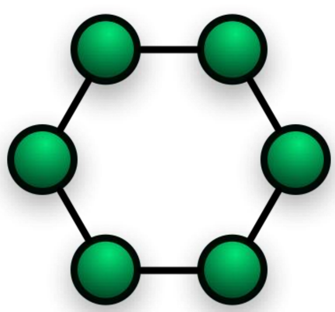


2D mesh

2D torus

(toroidal mesh)

3D mesh

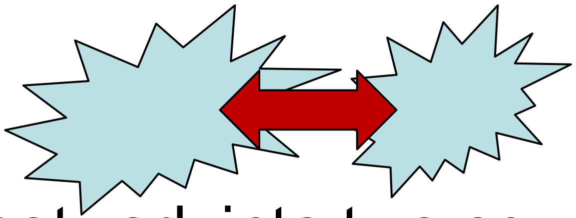


Ring

A network of computers and each node is a machine

How to measure network quality?

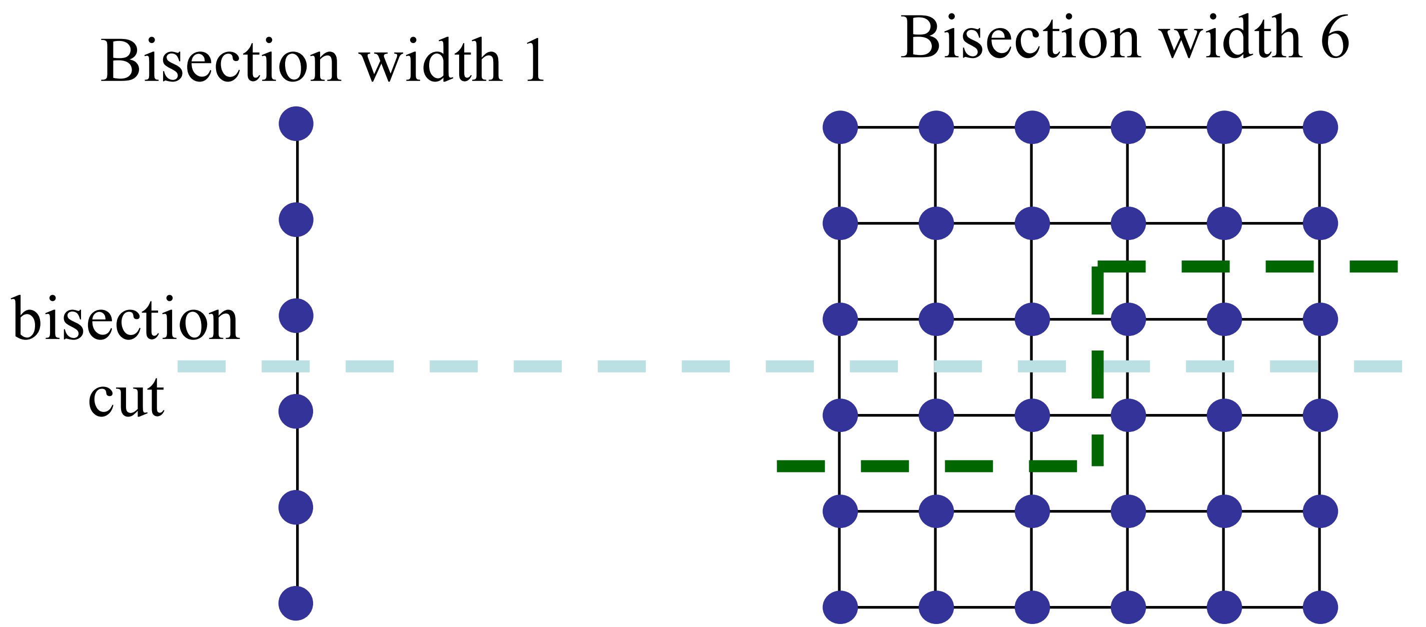
- **Bandwidth of each link**
  - The rate at which a link can transmit data. E.g. 1GB/s.
- **Bisection width of the network**
  - A measure of “number of simultaneous communications” between two subnetworks within a network



  - Typically divide a network into two equal halves by a single line/plane or curve (or two node sets that differ by at most 1 node in size)
    - There are many ways to partition
    - Each partitioning removes links that connect two halves
  - Find the minimum one among all possible partitionings
    - The minimum number of links that must be removed to partition the network into two equal halves

Bisection Width and Bisection Bandwidth

- **Example of bisection width**



Bisection bandwidth (different from bisection width)

- **Add bandwidth** of links that cut the network into two equal halves (or two sets that differ by at most 1 node in size)
- Choose the minimum bandwidth sum as the answer after above cutting for all possible ways of partitioning.

More definitions on network performance

- **Any time data is transmitted, we're interested in how long it will take for the data to reach its destination.**
- **Latency**
  - The time that elapses between the source's beginning to transmit the data and the destination's starting to receive the first byte.
- ***Startup cost*** The startup time required to handle a message at the sending and receiving nodes
- **Bandwidth**

  - The rate at which the destination receives data after it has started to receive the first byte.

Network transmission cost

Message transmission time = $\alpha + m\,\beta$

latency (seconds)

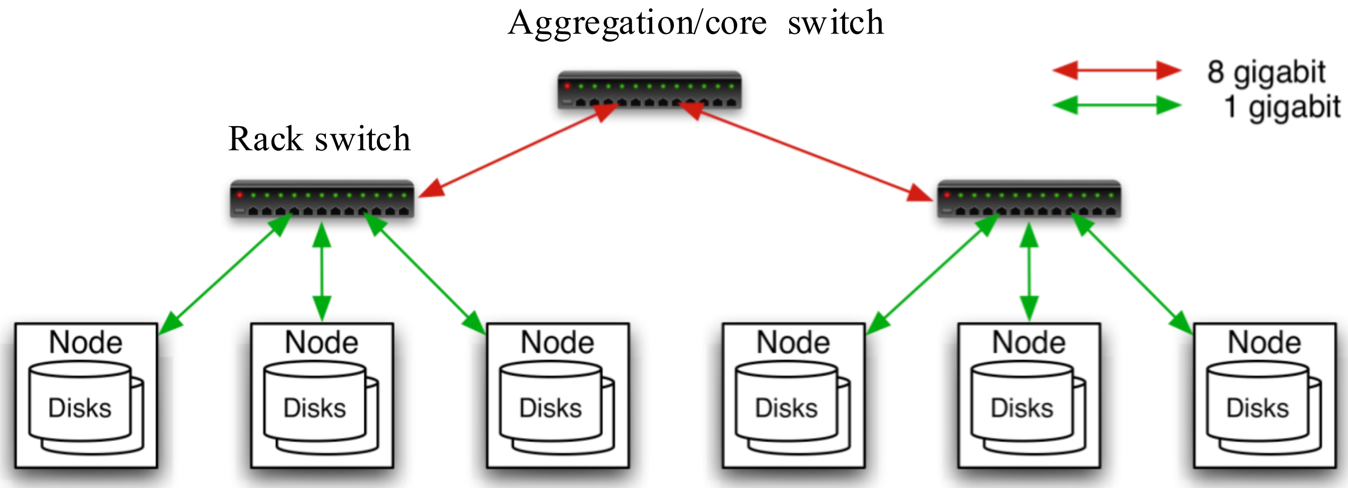
length of message (bytes)

1/bandwidth (bytes per second)

Typical latency/startup cost α : Tens of microseconds ~ 1 millisecond

Typical bandwidth β: 100 MB ~ 1GB per second

Typical network for a cluster

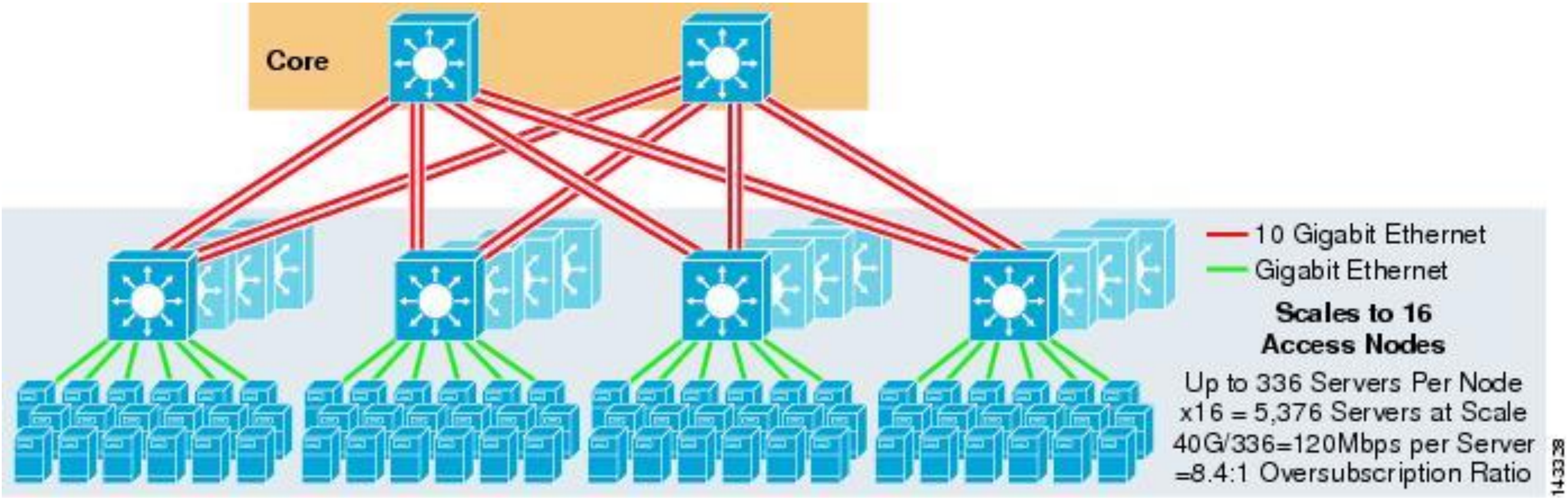


- Example: 40 nodes/rack.

Few thousand nodes in a cluster

- 1 Gbps bandwidth in rack, 8 Gbps out of rack
- Node specs :
  32+ cores, 64-256 GB RAM, 16 TB disks

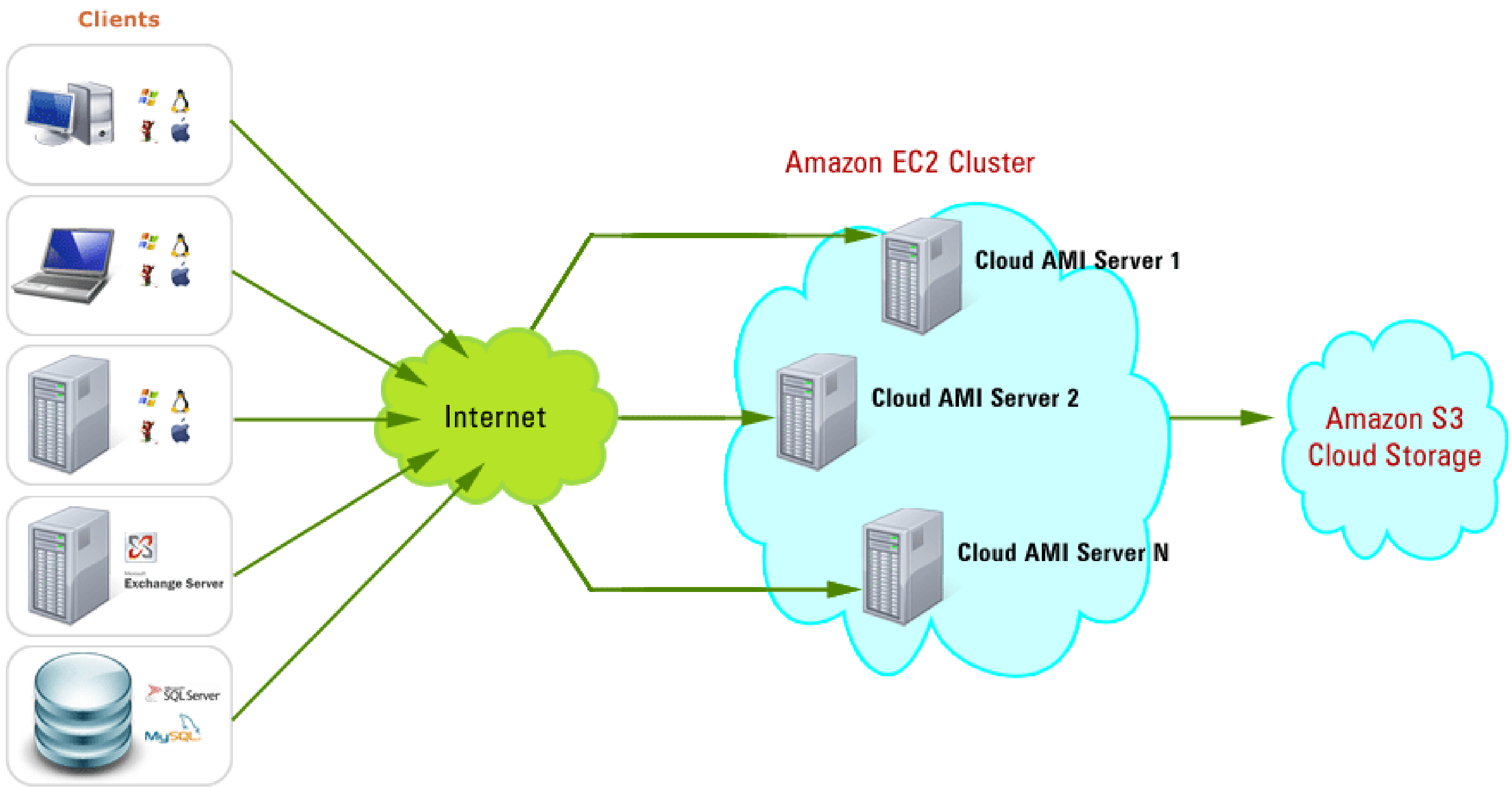
Layered Network in Clustered Machines

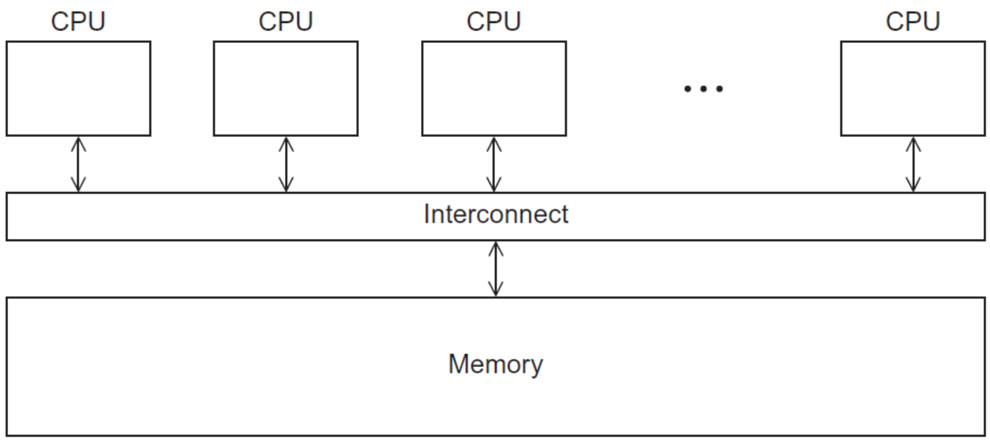


- A layered example from Cisco: core, aggregation, the edge or top-of-rack switch
- http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_Infra2_5/DCInfra_3a.html

Cloud Computing with Amazon EC2

- **On-demand elastic computing**
  - Allocate a Linux or windows cluster only when you need.
    - Pay based on time usage of computing instance/storage
  - Expandable or shrinkable

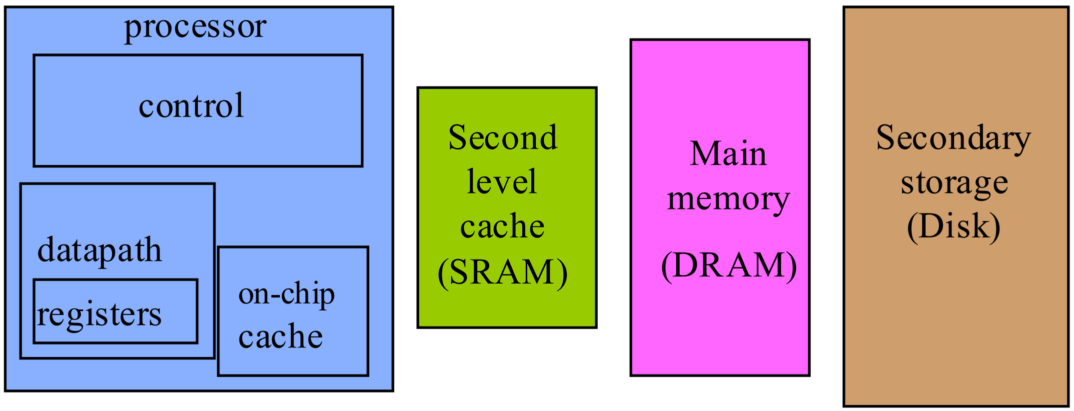




SHARED MEMORY ARCHITECTURES WITH CACHE COHERENCE

Memory Hierarchy and Performance

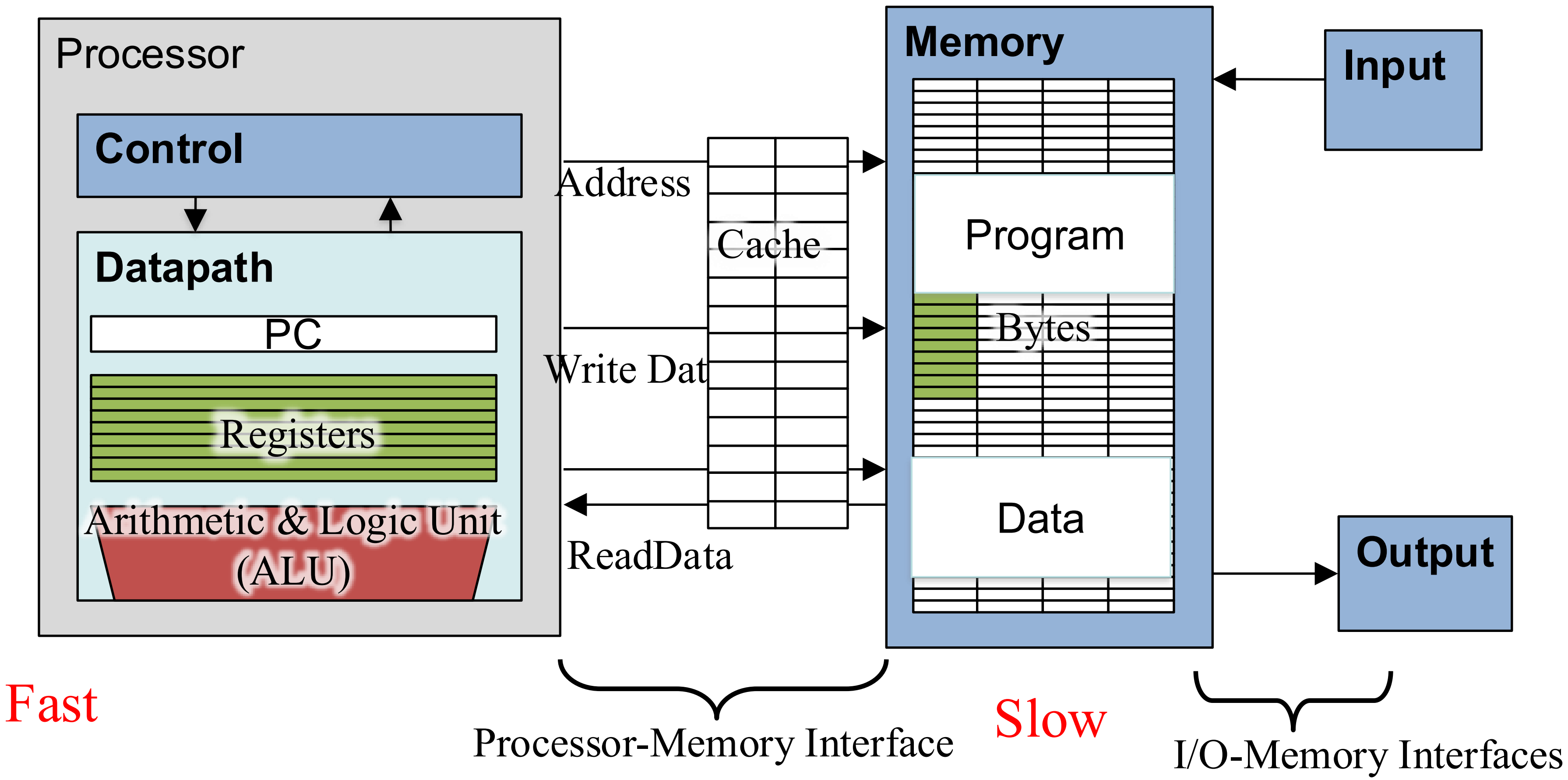
- **Most programs have a high degree of locality in their accesses**
  - **spatial locality:** accessing things nearby previous accesses
  - **temporal locality:** reusing an item that was previously accessed
- **Memory hierarchy tries to exploit locality to improve average**



Speed	1ns	10ns	60-100ns	0.1-10ms
Size	KB	MB	GB	TB

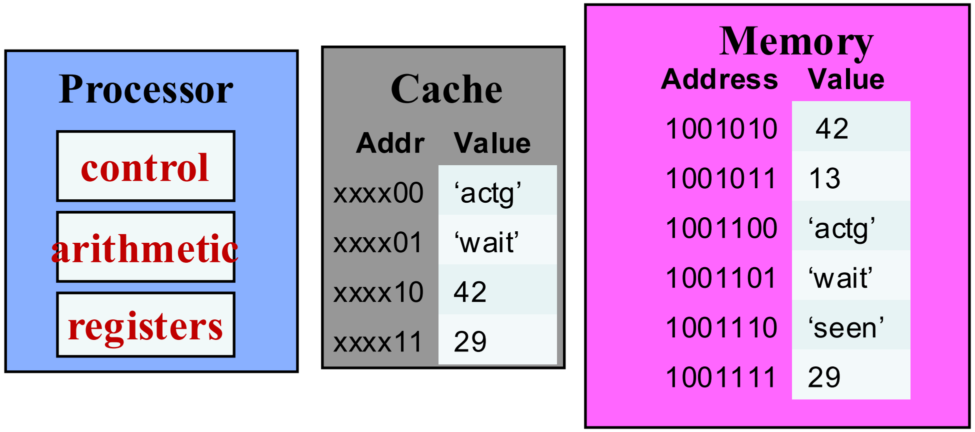
Uniprocessors in the Real World

- Have **caches (small amounts of fast memory) storing values of recently used or nearby data**
  - different memory ops can have different costs



Cache Basics

- Cache is fast (expensive) memory which keeps copy of data; it is hidden from software
  - Simplest example: data at memory address xxxxxx10 is stored at cache location 10

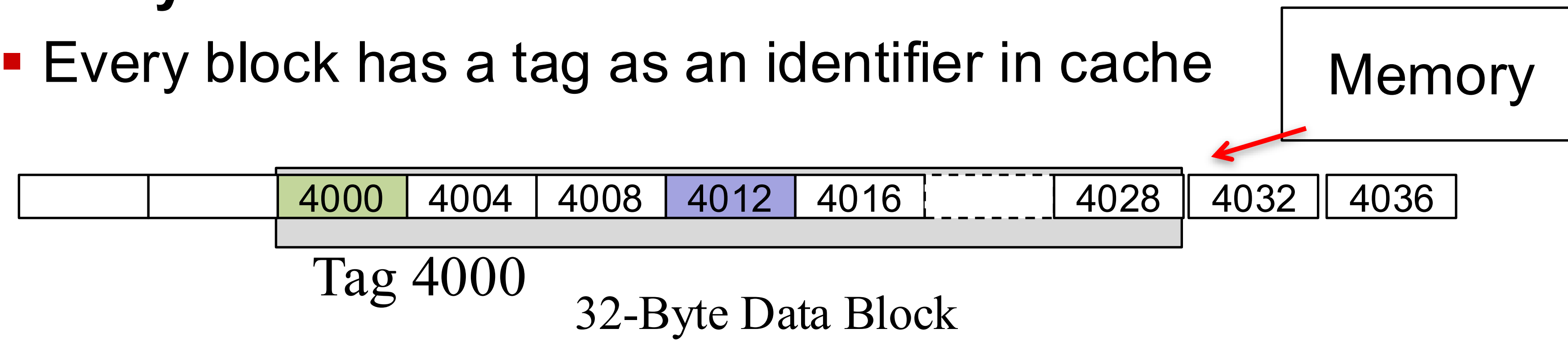


- Cache hit: in-cache memory access—cheap
- Cache miss: non-cached memory access—expensive
  - Need to access next, slower level of memory

from Berkeley CS267 Lecture Bulux/Demmel

Cache Blocks (or called Cache Lines)

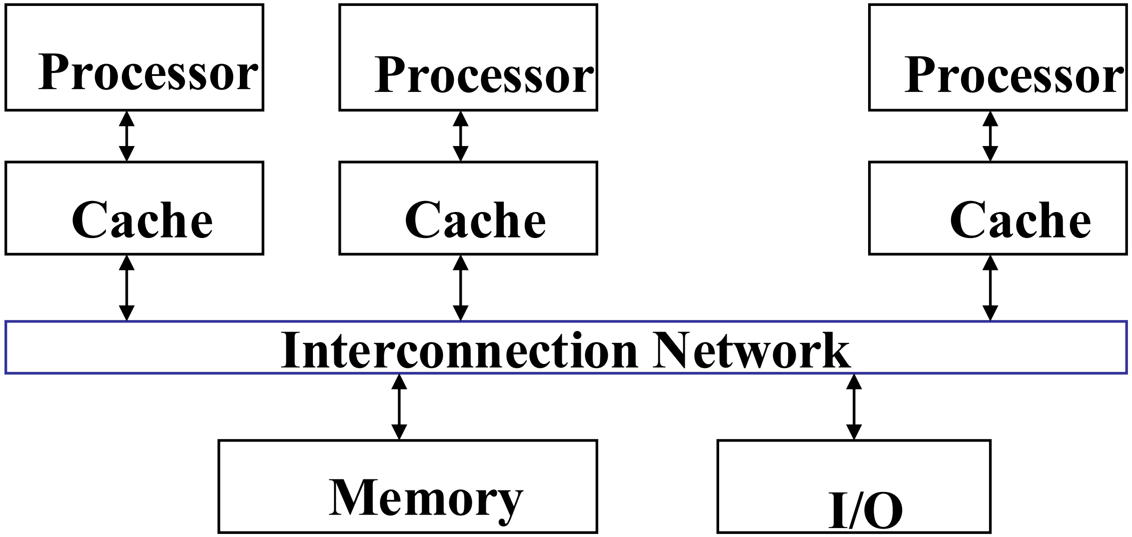
- **Memory data is divided into blocks**
  - Every block has a tag as an identifier in cache



- **When a program accesses some bytes, CPU checks its corresponding block**
  - e.g. CPU reads an integer of 4 bytes from address 4012
  - CPU checks if the cache block tagged with 4000 is in cache
  - If not in cache (called cache miss), fetch the entire block of 32 bytes from memory
- **Write some bytes**
  - Write an integer @ 4012 → update the cache block from address 4000 to 4028
  - May have to update memory block immediate (write-through protocol)

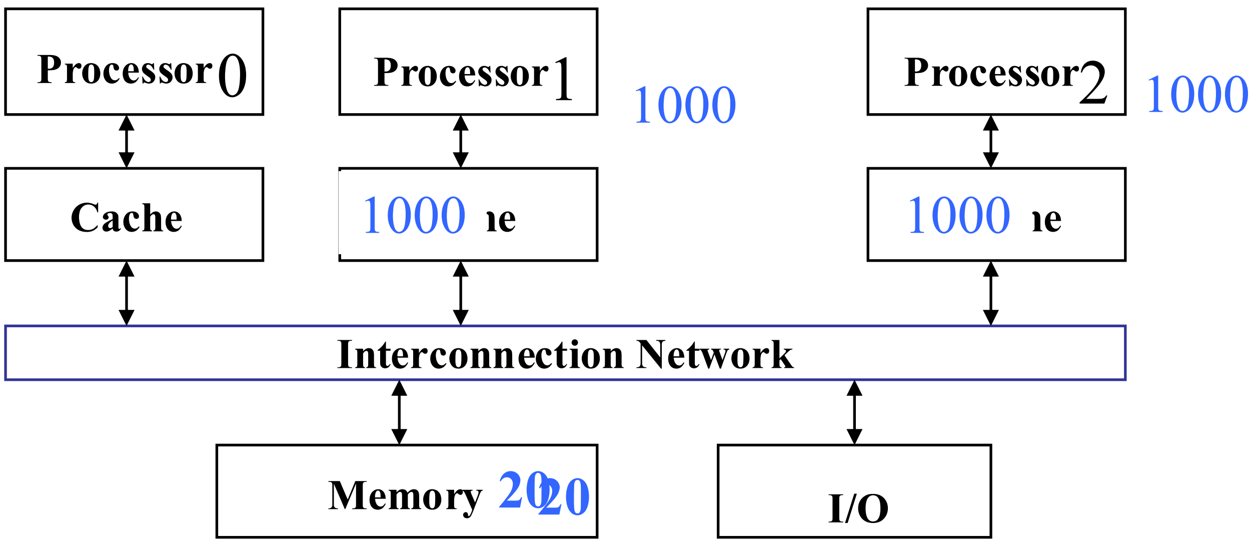
Shared Memory Architectures with Cache Coherence

- Memory is a performance bottleneck even with one processor
- Use caches to reduce bandwidth demands on main memory
- Each core has a local private cache holding data it has accessed recently
- Only cache misses have to access the shared common memory



Shared Memory and Caches

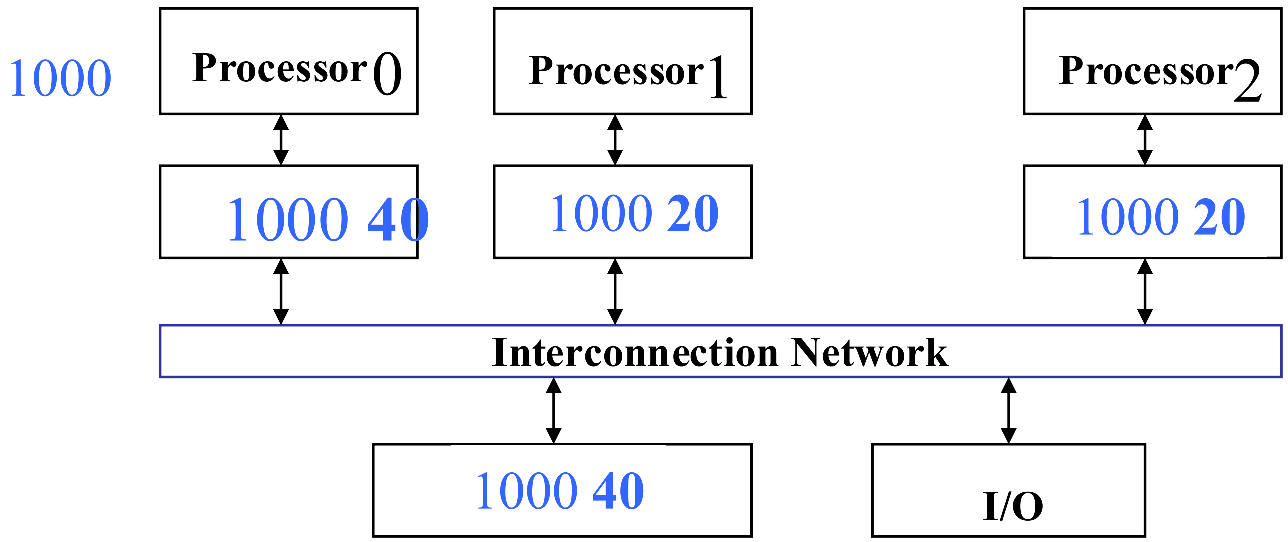
- **What if?**
  - Processors 1 and 2 read Memory[1000] (value 20)



Each cache of Processors 1 and 2 has a copy of memory[1000]

Shared Memory and Caches

- **Now:**
  - Processor 0 writes Memory[1000] with 40



Problem?

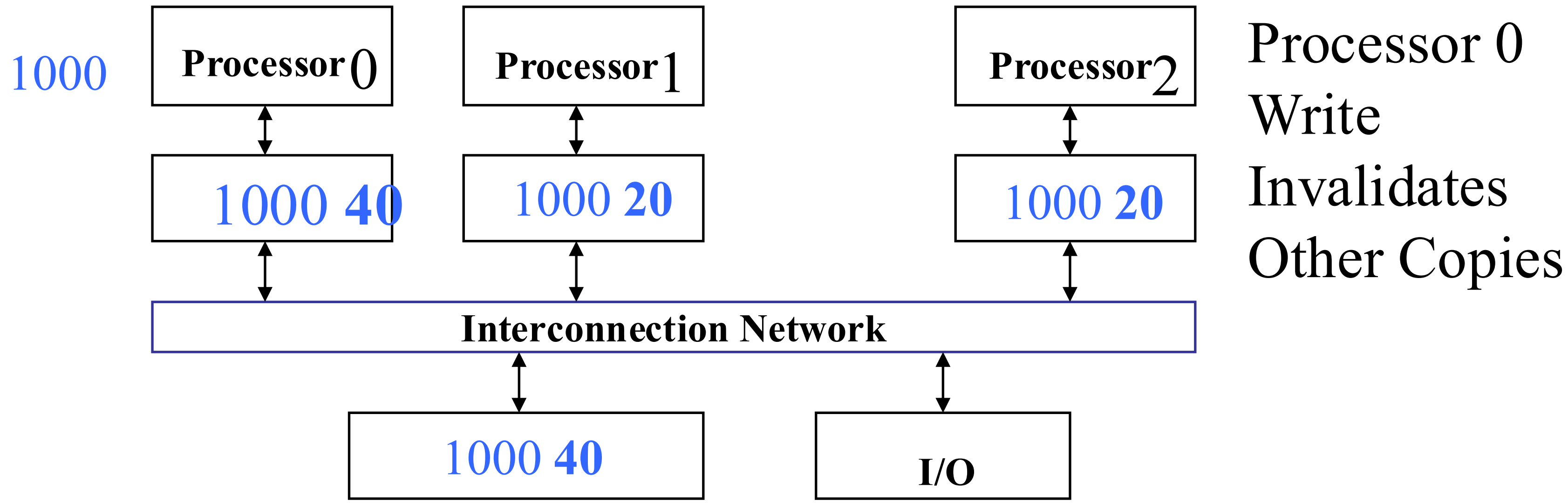
Cache data in Processor 2 is not coherent from Processor 1 even memory is updated

Keeping Multiple Caches Coherent

- **Architect's job: shared memory**
  => keep cache values coherent
- **Idea:**
  - When any processor has cache miss, fetch data from main shared memory.
  - When a processor writes, invalidate any cached copies in other processors. The corresponding data block in main memory will get updated.
    - Assume write-through with immediate update of entire cache line
- **How to detect data block is modified and where to invalidate?**

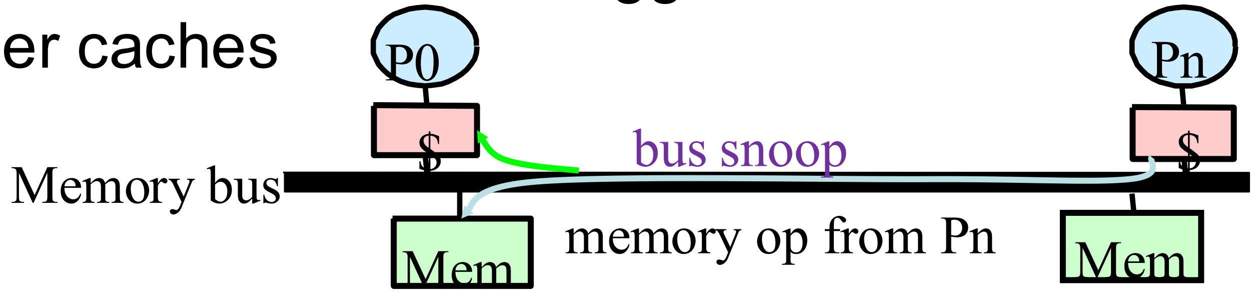
Shared Memory with Snooping Caches

- **Bus keeps track of what is written to memory and invalidates entries cached in other processors.**
- **For example, now with cache coherence**
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



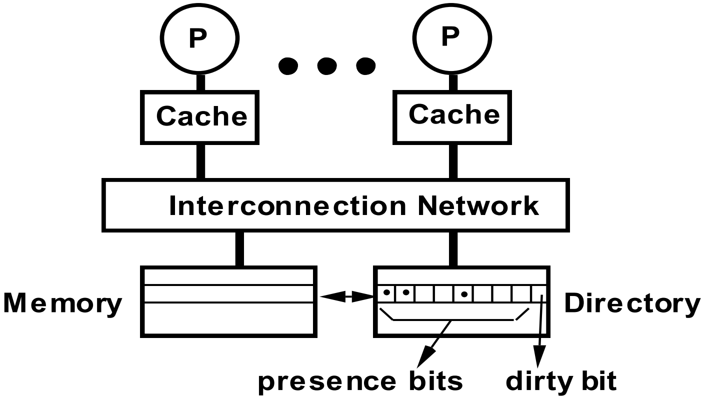
Cache-Coherence Protocols

- **Snoopy cache-coherence protocol**
  - Cache controller "snoops" all transactions on the bus, considering memory bus as a broadcast medium
  - Bus monitors a write transaction and triggers invalidation in other caches

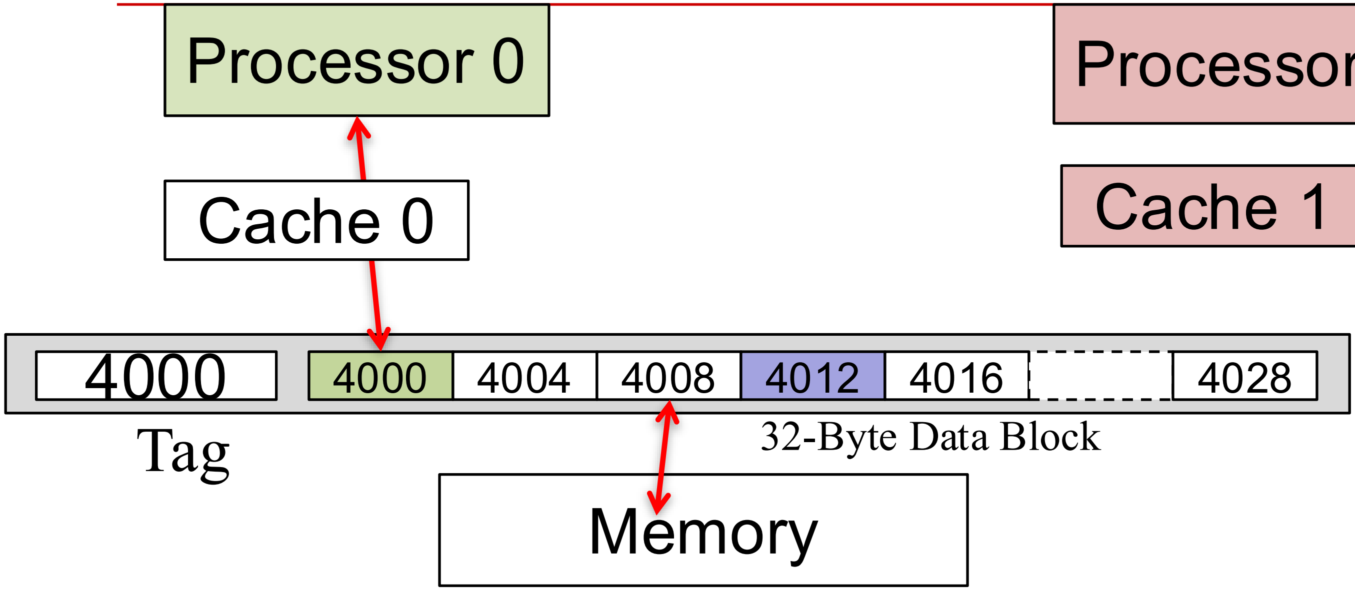


  - Not scalable for a large number of processors

- More scalable solution with lookup directories for larger systems. More complex/overhead

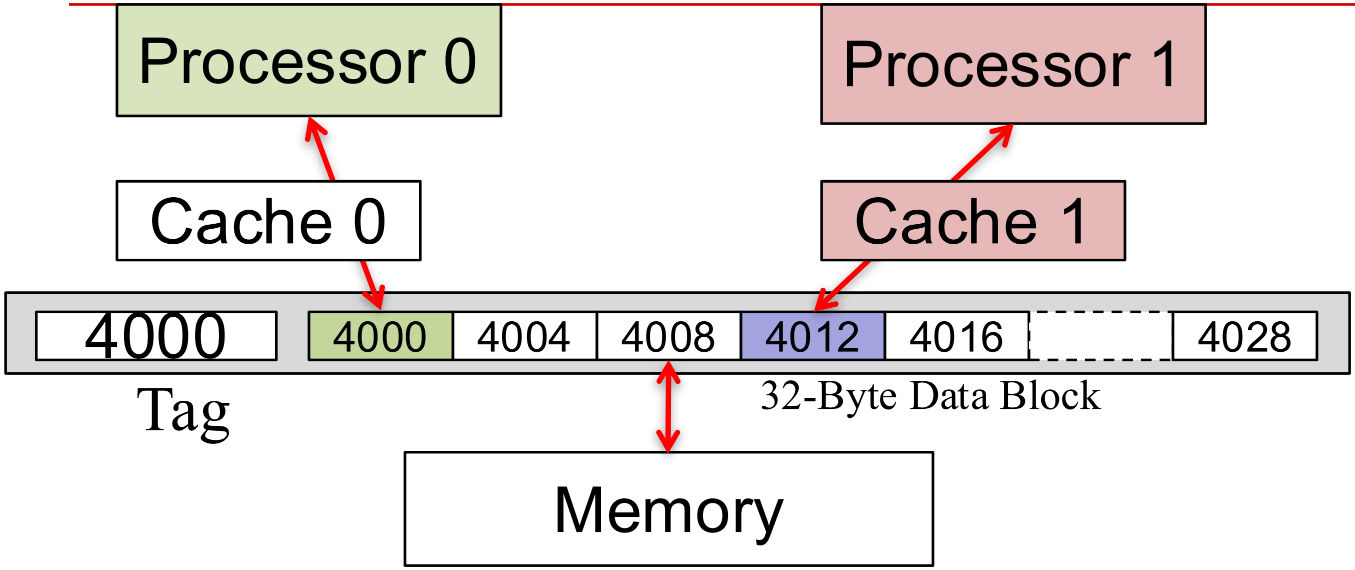


Takeaway from shared memory architectures



- Read a number → cause caching of a data block
  - P0: Read an integer at address 4000 → read/cache a block from address 4000 to 4028
- Write a number → invalidate a block in other caches
- Frequent invalidation of other caches is bad for performance!

False Sharing: Cache Coherency Tracked by Block



- Suppose block size is 32 bytes (i.e. cache line= 32 bytes)
- Suppose Processor 0 reads and writes variable X, Processor 1 reads and writes variable Y
- X is at memory address 4000. Y is at 4012
- What will happen? P0 writes X → invalidate a block in Cache 1 that holds Y for P1

Block invalidation in a ping-pong manner between two caches even though processors are accessing disjoint variables

False Sharing

- **Shared data within the same cache line (cache block) is modified by multiple processors.**
  - This updating occurs very frequently (for example, in a tight loop).
  - This effect is called *false sharing*
  - *It causes cache miss for every write, even they write to different locations.*
- **How can you prevent it for a higher cache hit ratio?**
  - Let parallel iterations write to different cache blocks (as much as possible)
    - allocate data used by each processor contiguously

    For i = 1 to 10000

    x[i]= 3

    It is bad if x[0] is modified by Proc 0 and x[1] is modified by Proc 1 as x[0] and x[1] are in the same cache block most likely.
  - Make use of private data for each thread as much as possible

Summary of Parallel Architecture

- **Important Concepts**
  - SIMD
  - MIMD
    - Shared memory machines
      - Cache coherence, false sharing
    - Distributed memory machines
    - Interconnection network
      - Topology, bisection width and bandwidth, networking cost.
  - Cluster computing and clouds