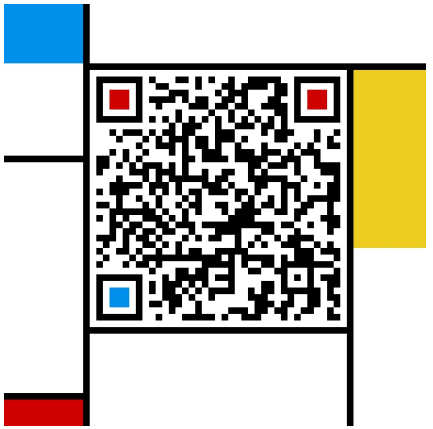
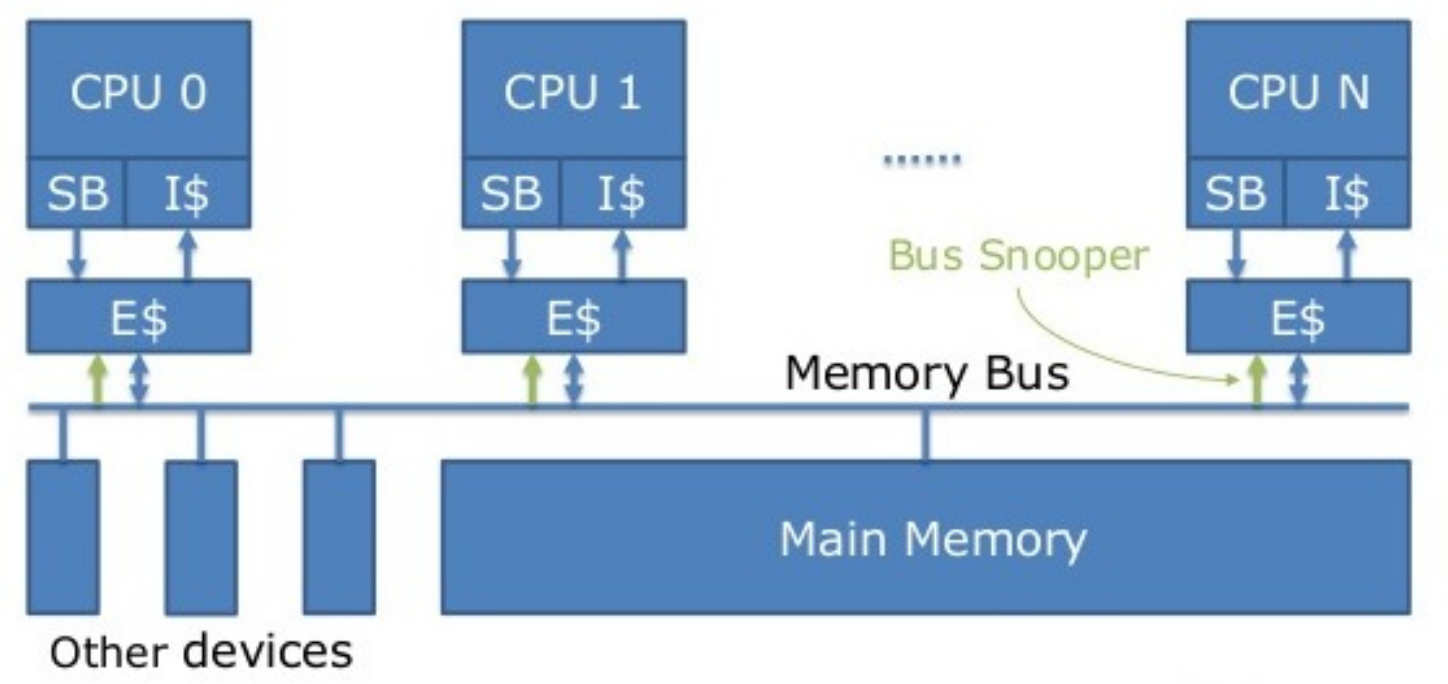
计算机组成与系统结构
Computer Organization & System Architecture

Huang Kejie (黄科杰) 百人计划研究员
Office: 玉泉校区老生仪楼 304
Email address: huangkejie@zju.edu.cn
HP: 17706443800



(Chip) Multicore Multiprocessor

- SMP: (Shared Memory) Symmetric Multiprocessor
  - Two or more identical CPUs/Cores
  - Single shared *coherent* memory

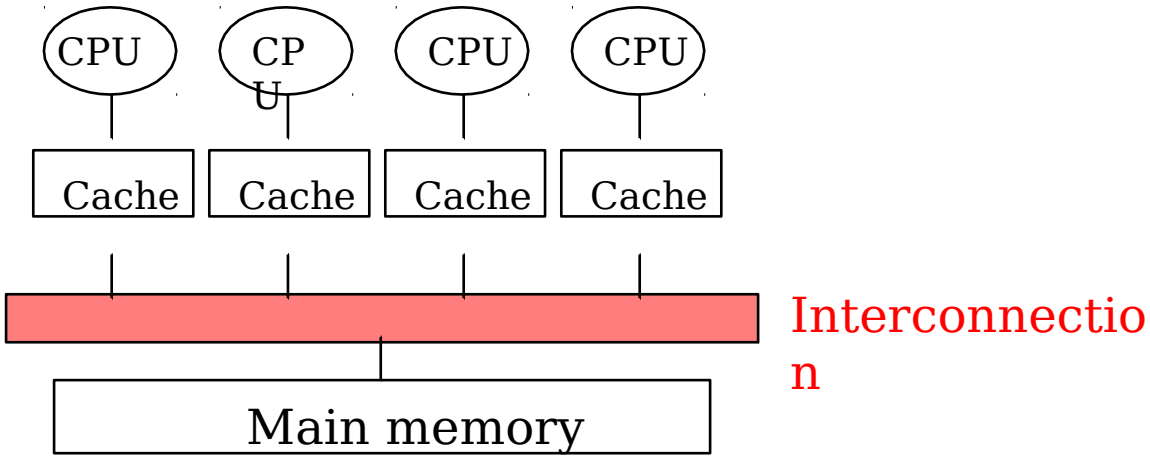


Multiprocessor Key Questions

- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How many processors can be supported?

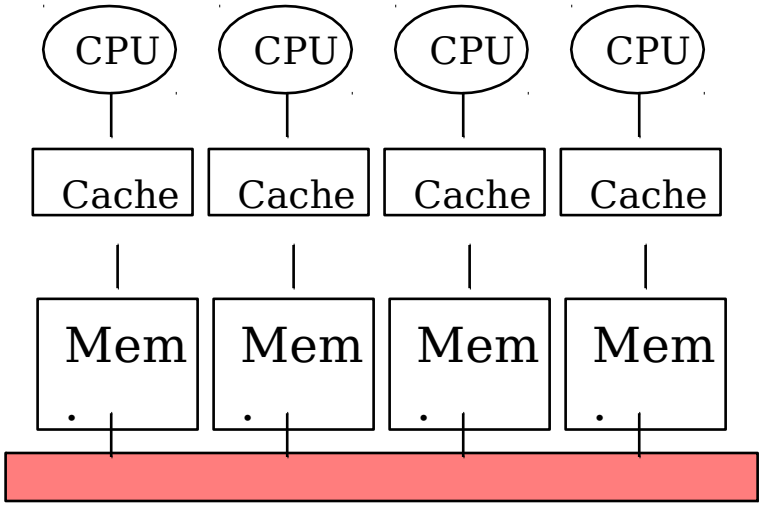
Taxonomy of Parallel Computers

- According to physical organization of processors and memory:
  - Physically centralized memory, uniform memory access (UMA)
    - All memory is allocated at same distance from all processors
    - Also called symmetric multiprocessors (SMP)
    - Memory bandwidth is fixed and must accommodate all processors → does not scale to large number of processors
    - Used in CMPs today (single-socket ones)

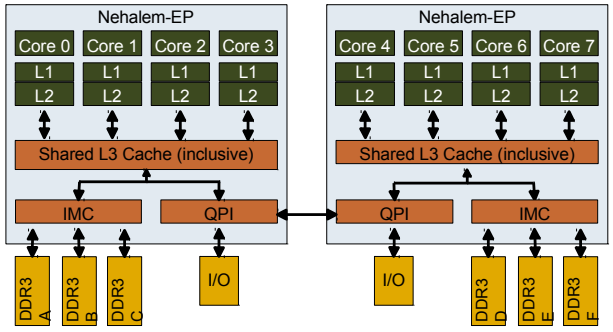


Taxonomy of Parallel Computers

- According to physical organization of processors and memory:
  - Physically distributed memory, non-uniform memory access (NUMA)
    - A portion of memory is allocated with each processor (node)
    - Accessing local memory is much faster than remote memory
    - If most accesses are to local memory than overall memory bandwidth increases linearly with the number of processors
    - Used in multi-socket CMPs E.g Intel Nehalem



Node



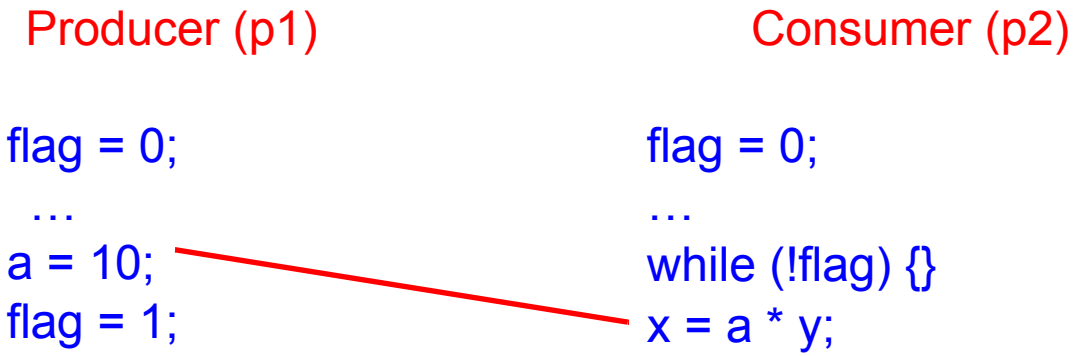
Interconnecti on

Taxonomy of Parallel Computers

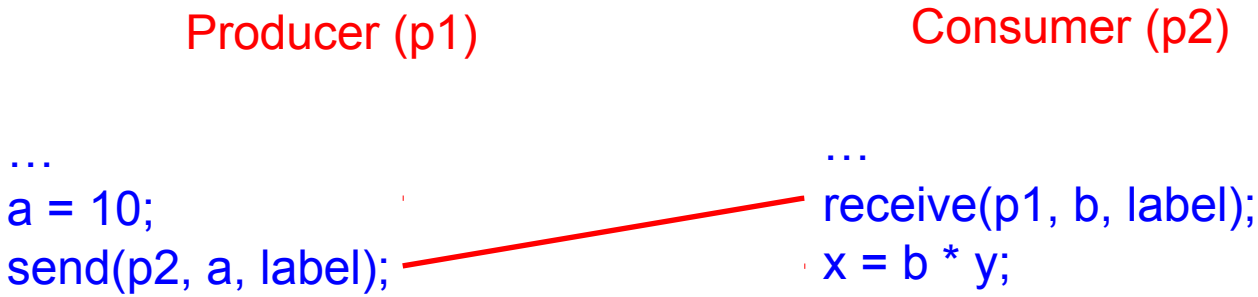
- According to memory communication model
  - Shared address or shared memory
    - Processes in different processors can use the same virtual address space
    - Any processor can directly access memory in another processor node
    - Communication is done through shared memory variables
    - Explicit synchronization with locks and critical sections
    - Arguably easier to program??
  - Distributed address or message passing
    - Processes in different processors use different virtual address spaces
    - Each processor can only directly access memory in its own node
    - Communication is done through explicit messages
    - Synchronization is implicit in the messages
    - Arguably harder to program??
    - Some standard message passing libraries (e.g., MPI)

Shared Memory vs. Message Passing

- Shared memory



- Message passing



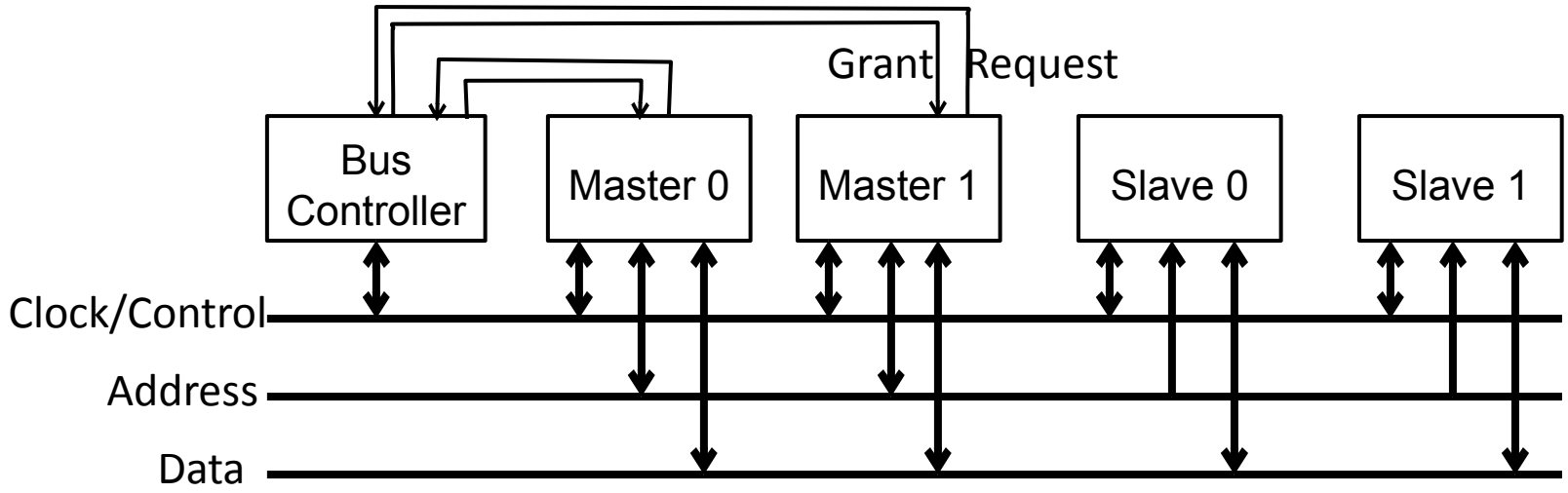
Shared Memory Multiprocessor (SMP)

- Q1 – Single address space shared by all processors/cores

- Q2 – Processors coordinate/communicate through shared variables in memory (via loads and stores)
  - Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time

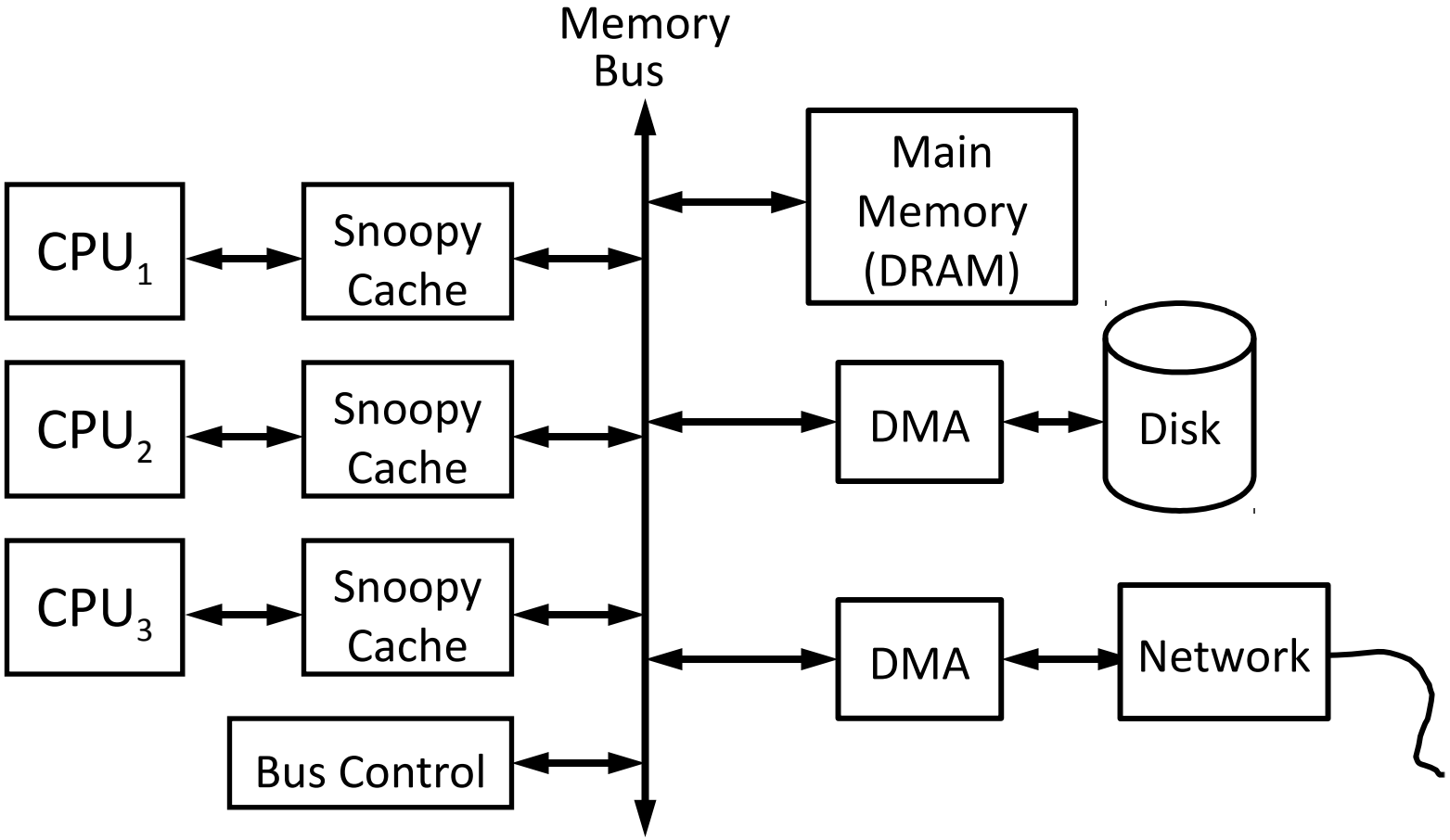
- All multicore computers today are SMP

Bus Management



- A “bus” is a collection of shared wires
  - Newer “busses” use point-point links
- Only one “master” can initiate a transaction by driving wires at any one time
- Multiple “slaves” can observe and conditionally respond to the transaction on the wires
  - slaves decode address on bus to see if they should respond (memory is most common slave)
  - some masters can also act as slaves
- Masters arbitrate for access with requests to bus “controller”
  - Some busses only allow one master (in which case, it’s also the controller)

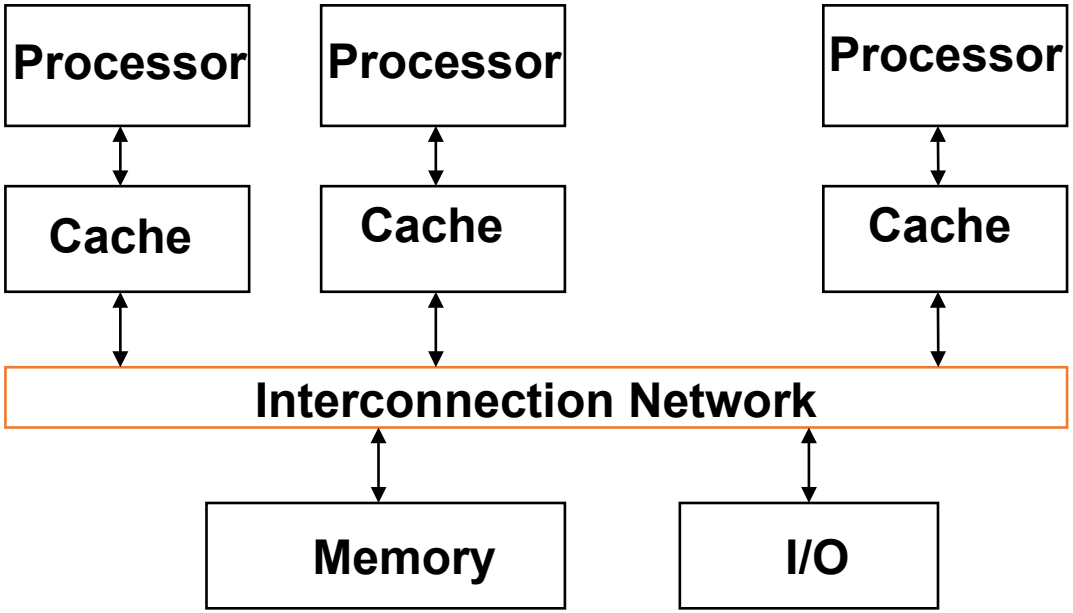
Shared-Memory Multiprocessor



Use snoopy mechanism to keep all processors' view of memory coherent

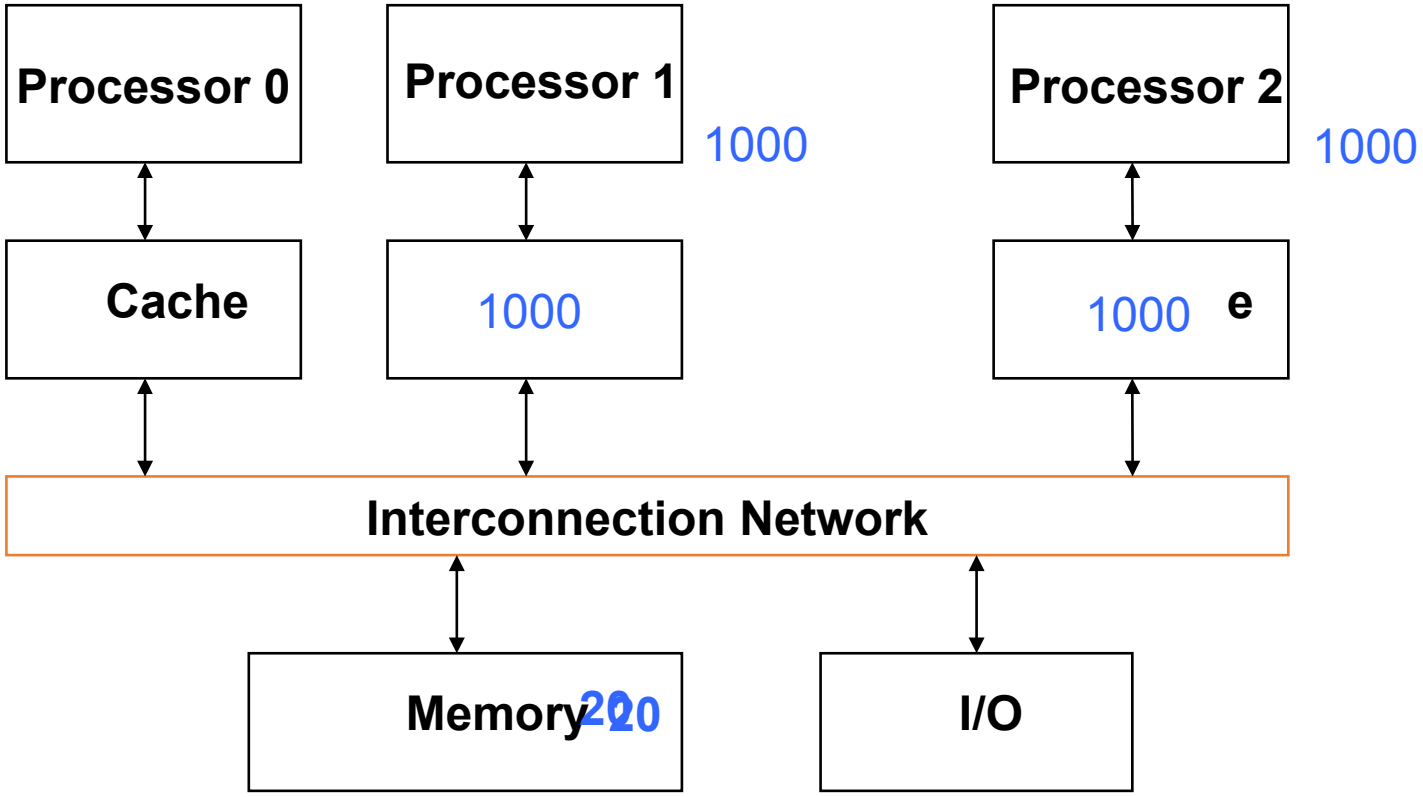
Multiprocessor Caches

- 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



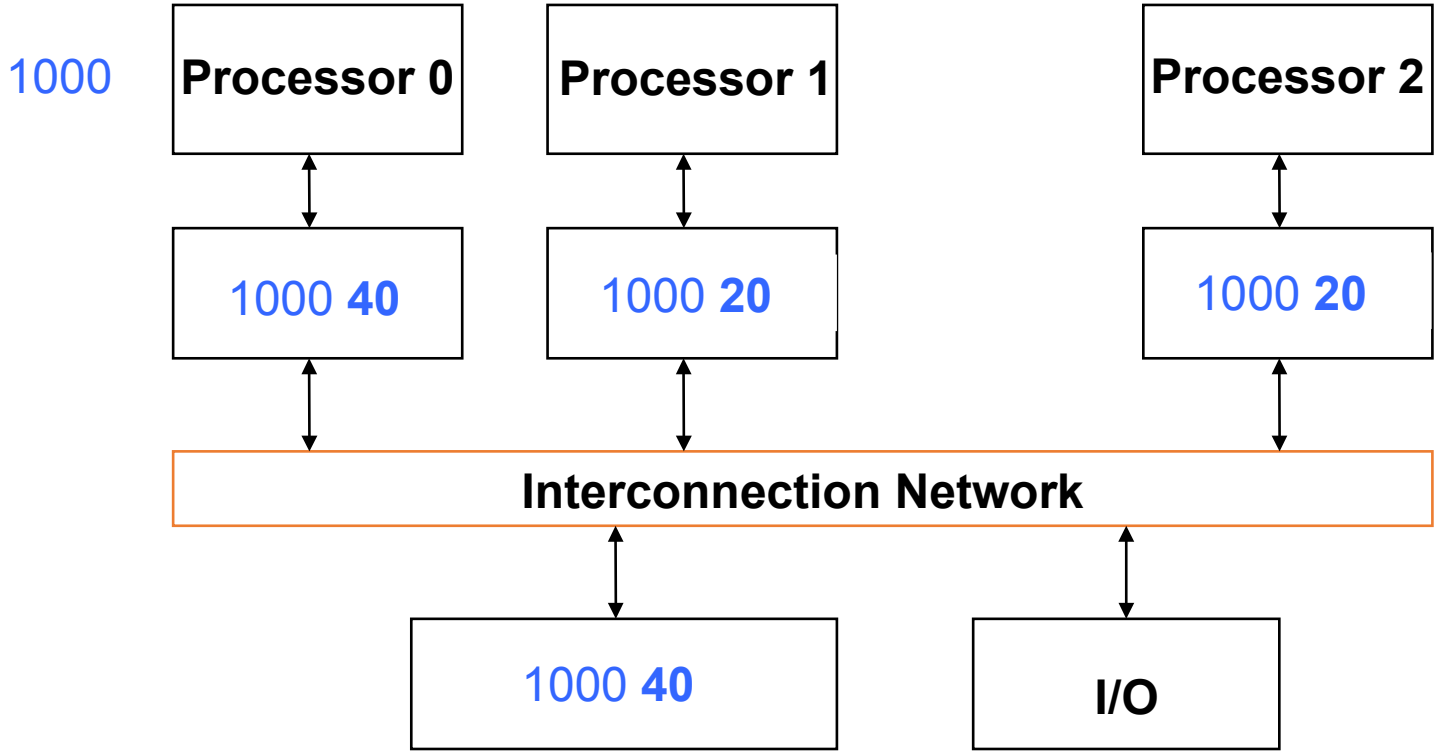
Shared Memory and Caches

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



Shared Memory and Caches

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



Problem?

Keeping Multiple Caches Coherent

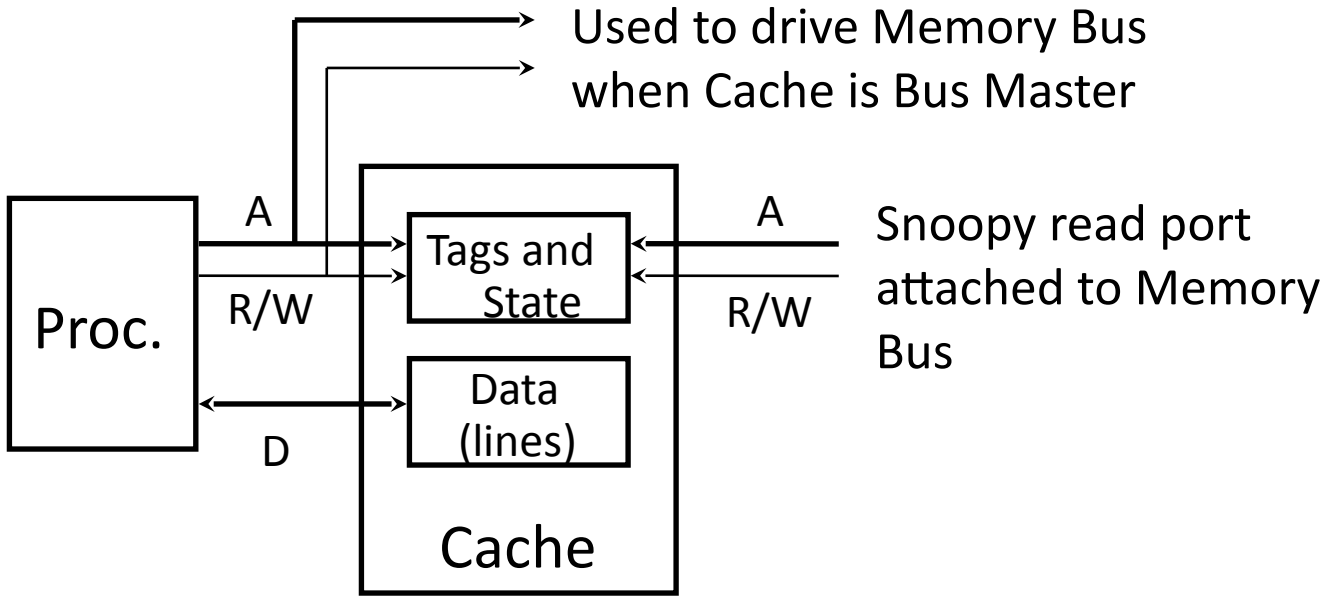
- Architect's job: shared memory
  => keep cache values ***coherent***

- Idea: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate any other copies

- Write transactions from one processor, other caches "snoop" the common interconnect checking for tags they hold
  - Invalidate any copies of same address modified in other cache

Snoopy Cache, Goodman 1983

- Idea: Have cache watch (or snoop upon) other memory transactions, and then "do the right thing"
- Snoopy cache tags are dual-ported



Snoopy Cache-Coherence Protocols

- Write miss:
  - the address is invalidated in all other caches before the write is performed
- Read miss:
  - if a dirty copy is found in some cache, a write-back is performed before the memory is read

How Does HW Keep $ Coherent?

- Each cache tracks state of each *block* in cache:

1. *Shared*: up-to-date data, other caches may have a copy
2. *Modified*: up-to-date data, changed (dirty), no other cache has a copy, OK to write, memory out-of-date
3. *Exclusive*: up-to-date data, no other cache has a copy, OK to write, memory up-to-date
    - Avoids writing to memory if block replaced
    - Supplies data on read instead of going to memory
4. Owner: up-to-date data, other caches may have a copy (they must be in Shared state)
    - Only cache that supplies data on read instead of going to memory

Two Optional Performance Optimizations of Cache Coherency via New States

Name of Common Cache Coherency Protocol: MOESI

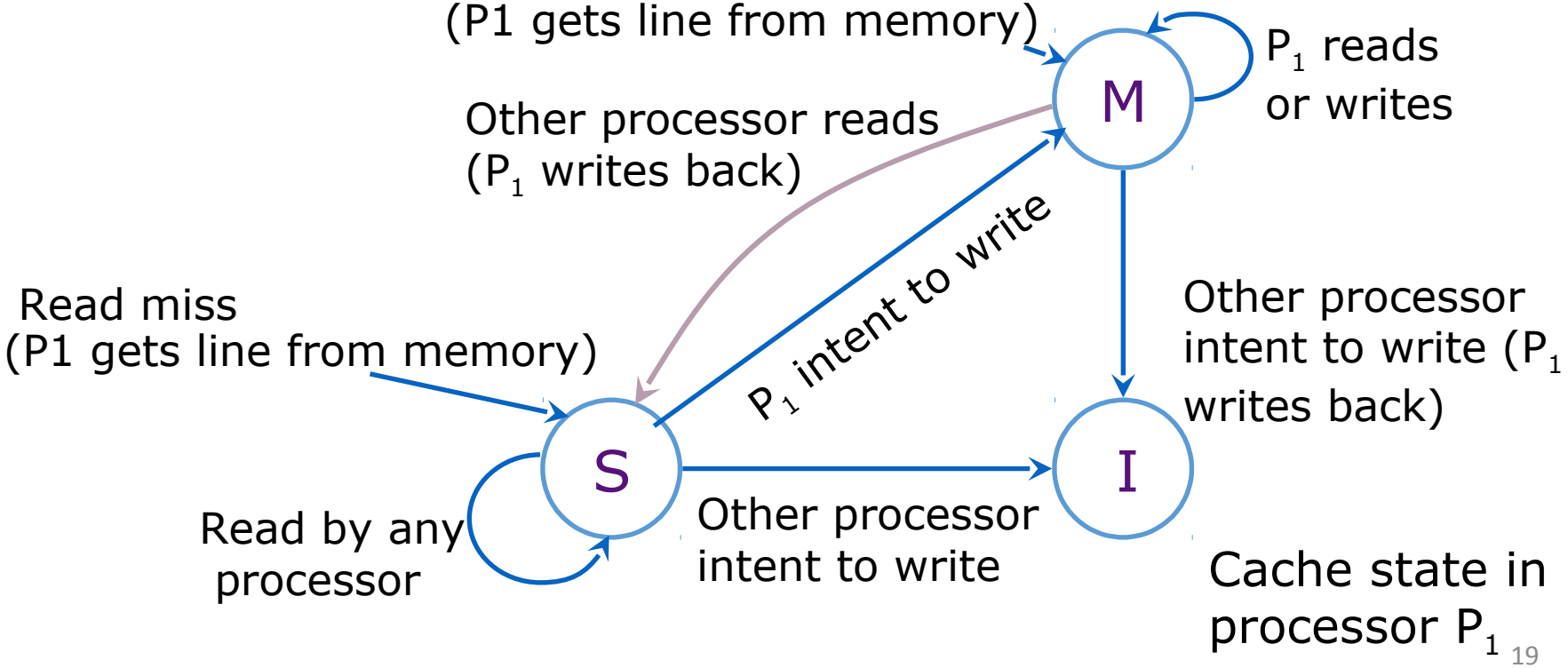
- Memory access to cache is either
  - Modified (in cache)
  - Owned (in cache)
  - Exclusive (in cache)
  - Shared (in cache)
  - Invalid (not in cache)

Snooping/Snoopy Protocols
e.g., the Berkeley Ownership Protocol
See http://en.wikipedia.org/wiki/Cache_coherence
Berkeley Protocol is a wikipedia stub!

Cache State-Transition Diagram
The MSI protocol

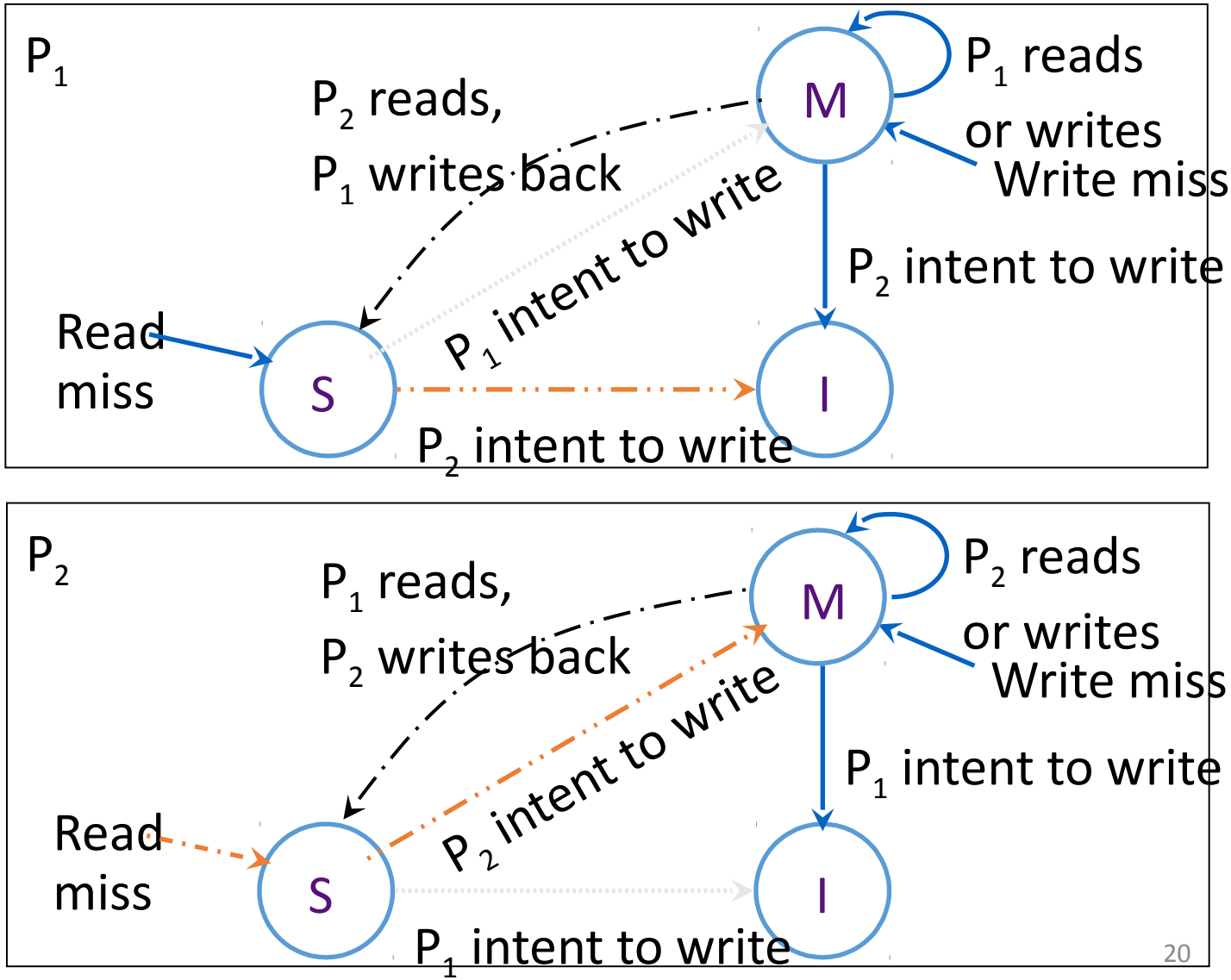
Each cache line has state bits

M: Modified
S: Shared
I: Invalid

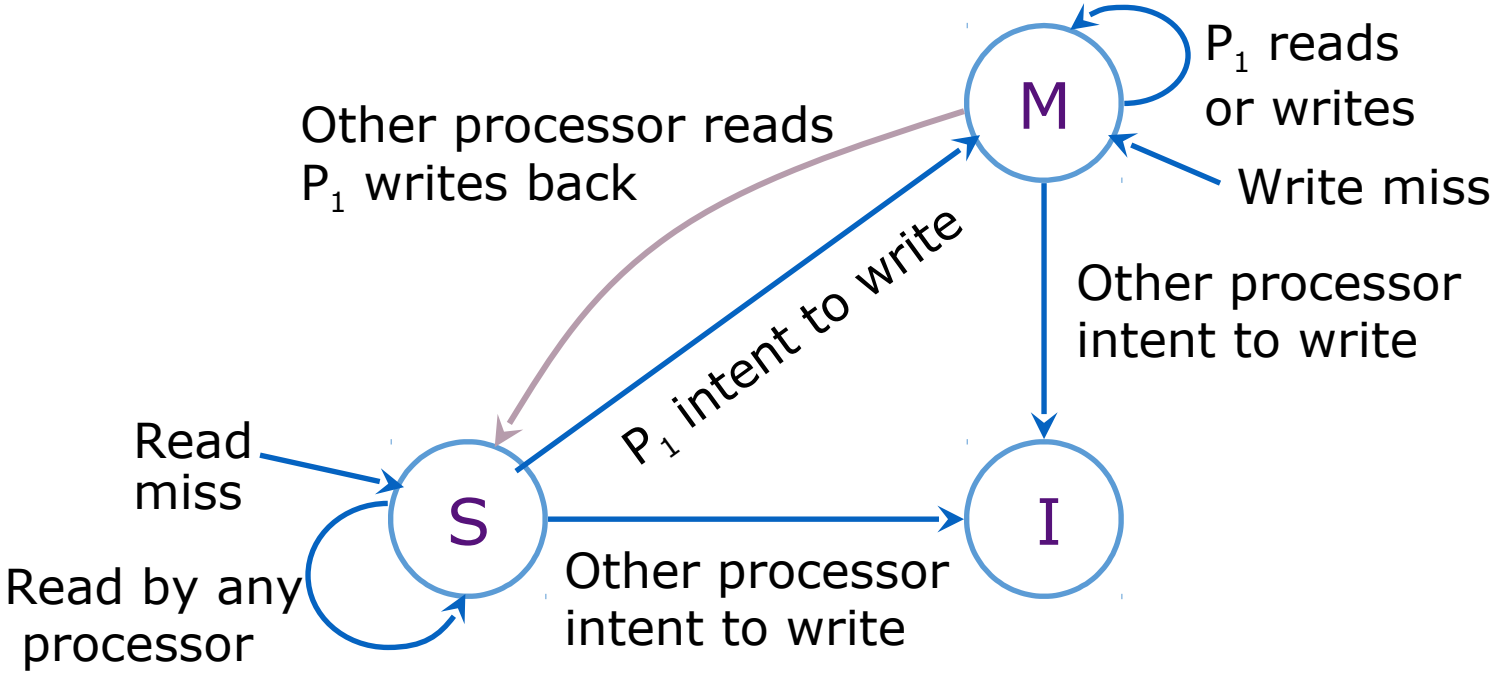


Two-Processor Example (Reading and writing the same cache line)

P_1 reads
P_1 writes
P_2 reads
P_2 writes
P_1 reads
P_1 writes
P_2 writes
P_1 writes



Observation

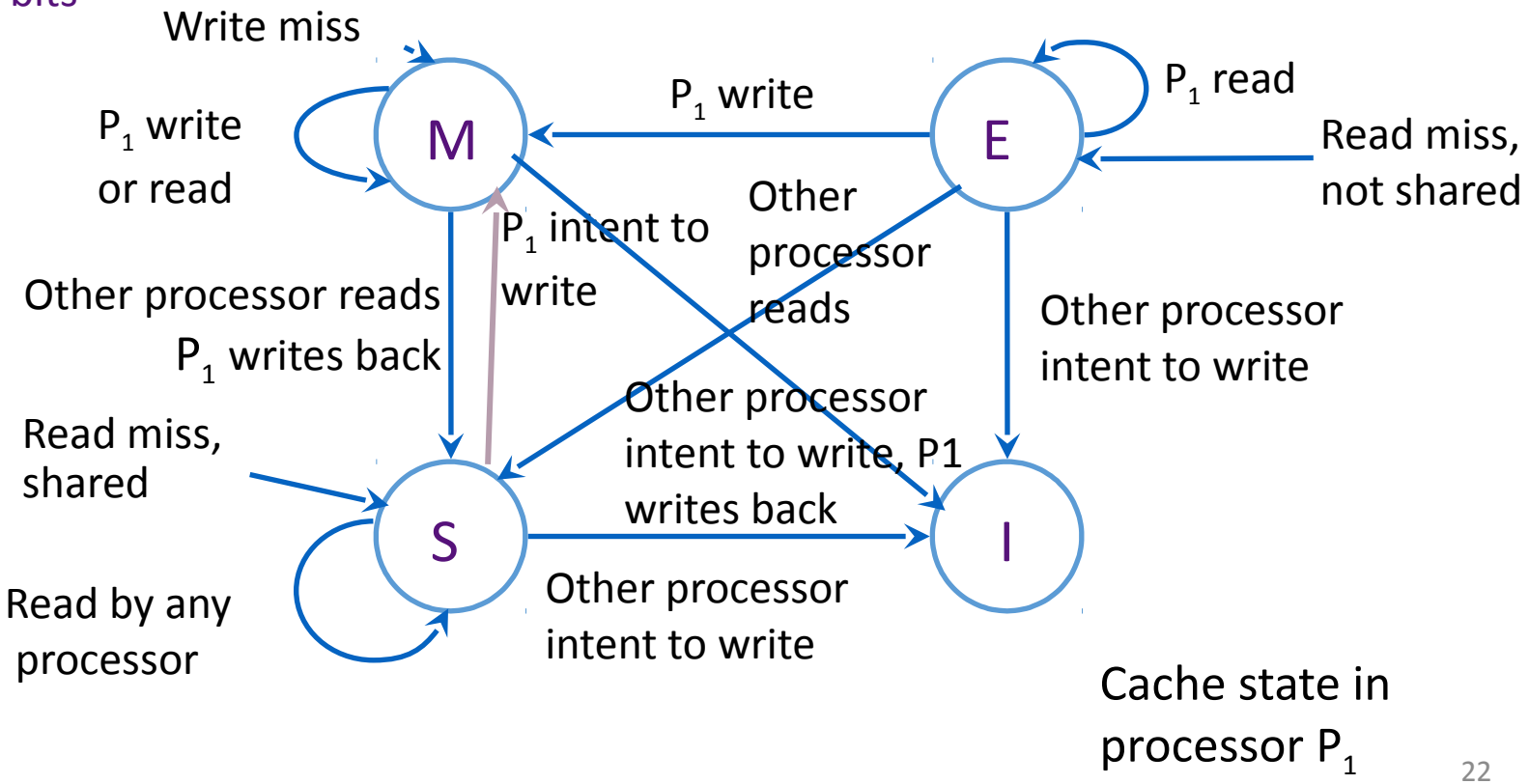


- If a line is in the M state then no other cache can have a copy of the line!
- Memory stays coherent, multiple differing copies cannot exist

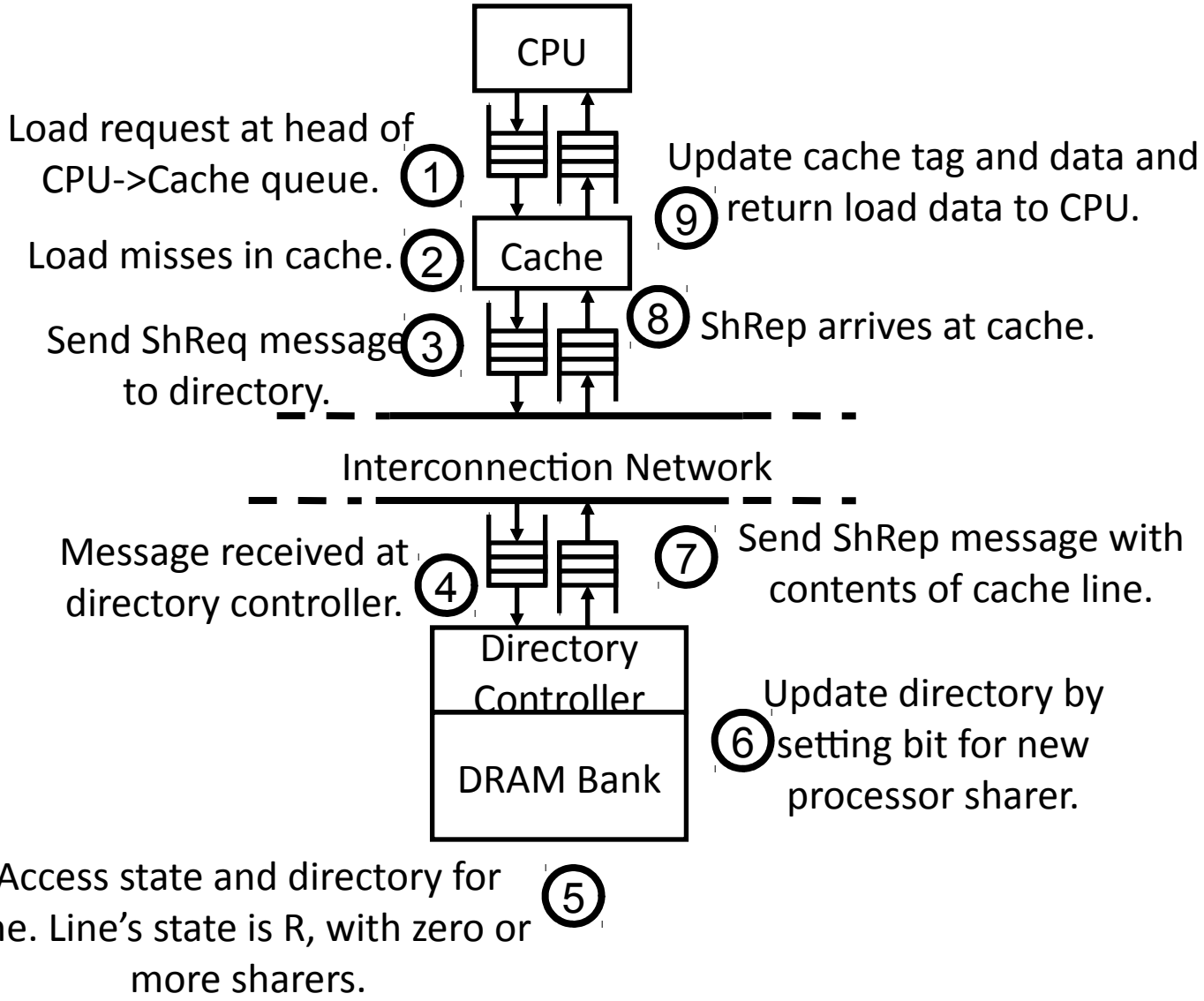
MESI: An Enhanced MSI protocol
increased performance for private data

Each cache line has a tag

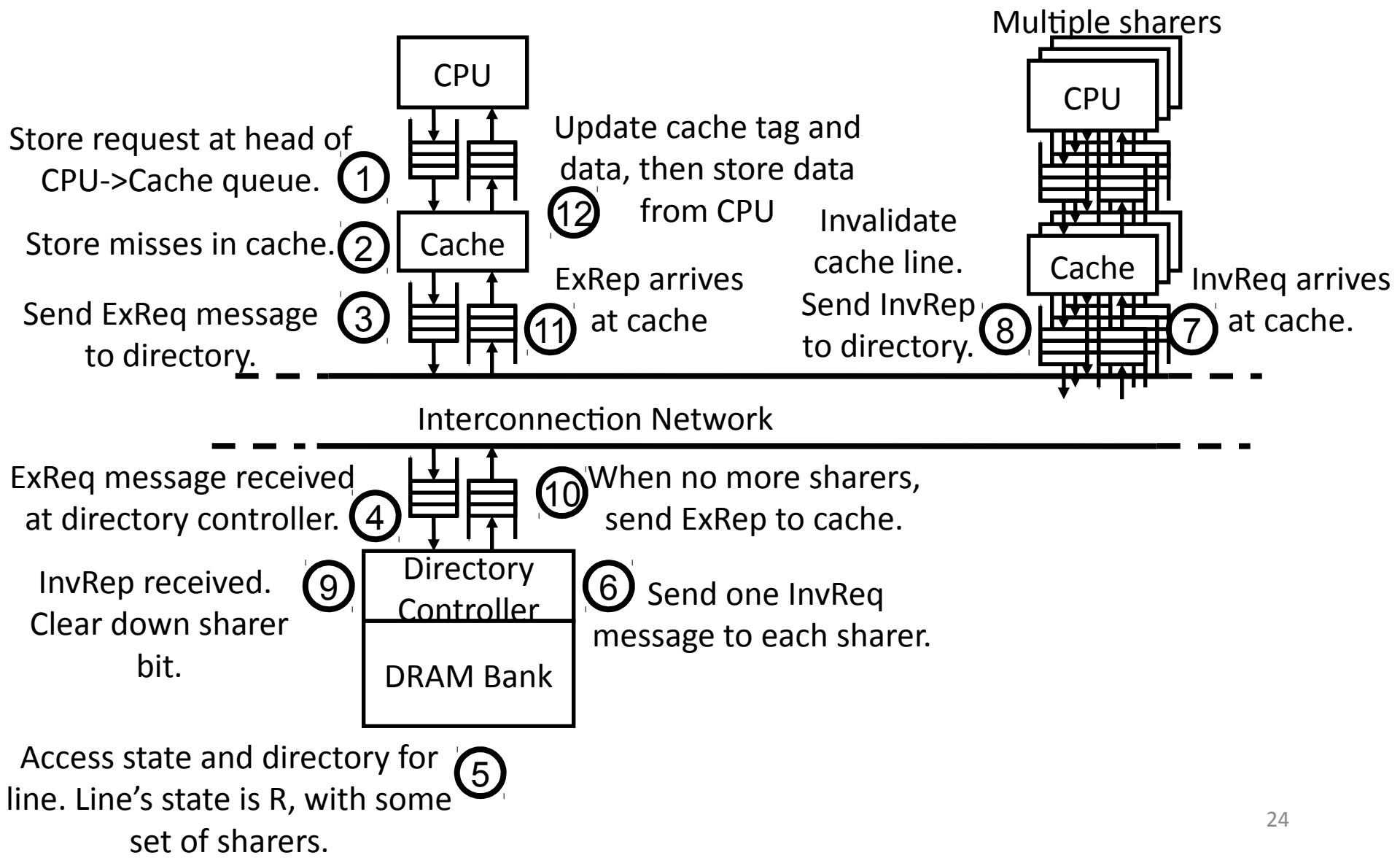
M: Modified Exclusive
E: Exclusive but unmodified
S: Shared
I: Invalid



Read miss, to uncached or shared line

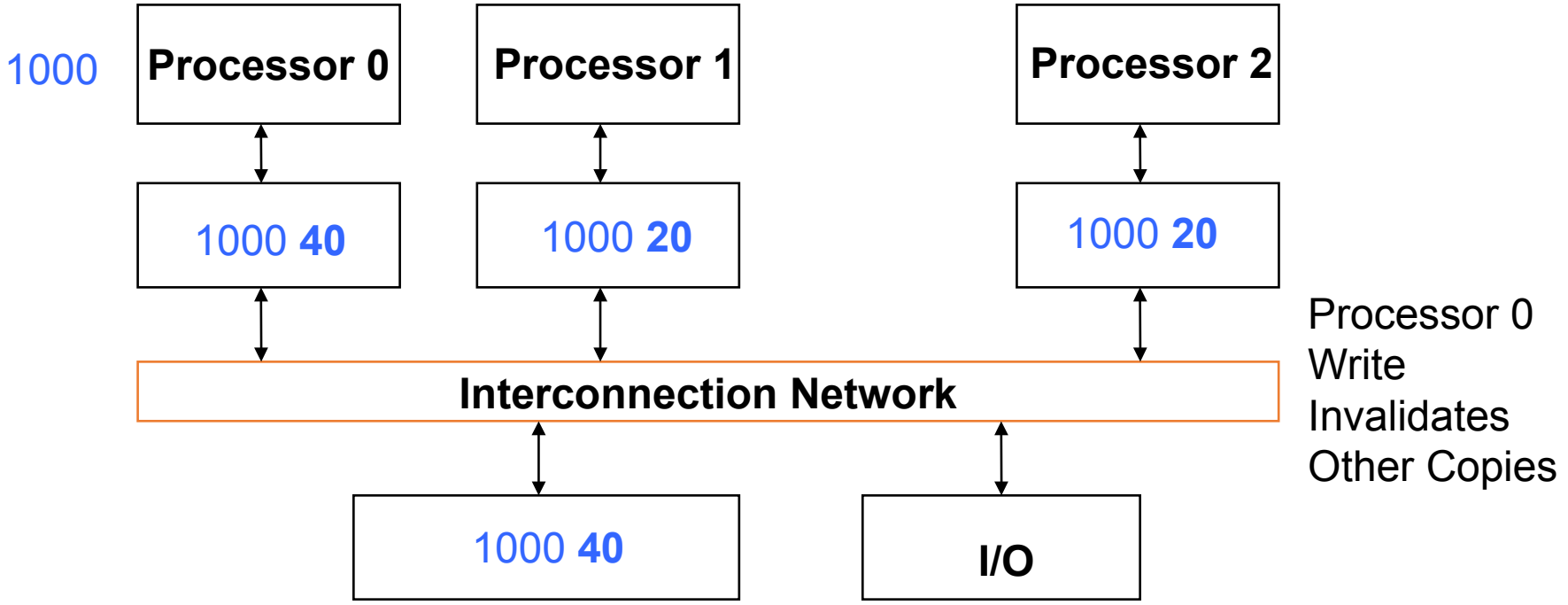


Write miss, to read shared line



Shared Memory and Caches

- Example, now with cache coherence
  - Processors 1 and 2 read Memory[1000]
  - Processor 0 writes Memory[1000] with 40



Peer Instruction: Which Statement is True?

RED: Using write-through caches removes the need for cache coherence

GREEN: Every processor store instruction must check contents of other caches

ORANGE: Most processor load and store accesses only need to check in local private cache

YELLOW: Only one processor can cache any memory location at one time

Review MOESI Cache Coherency

1. *Shared*: up-to-date data, other caches may have copy
2. *Modified*: up-to-date data, changed (dirty), no other cache has copy, OK to write, memory out-of-date
3. *Exclusive*: up-to-date data, no other cache has copy, OK to write, memory up-to-date
4. *Owner*: up-to-date data, other caches may have a copy (they must be in Shared state)

I. If in Exclusive state, processor can write without notifying other caches

II. Owner state is variation of Shared state to let caches supply data instead of going to memory on read miss

III. Exclusive state is variation of Modified state to let caches avoid writing to memory on a miss

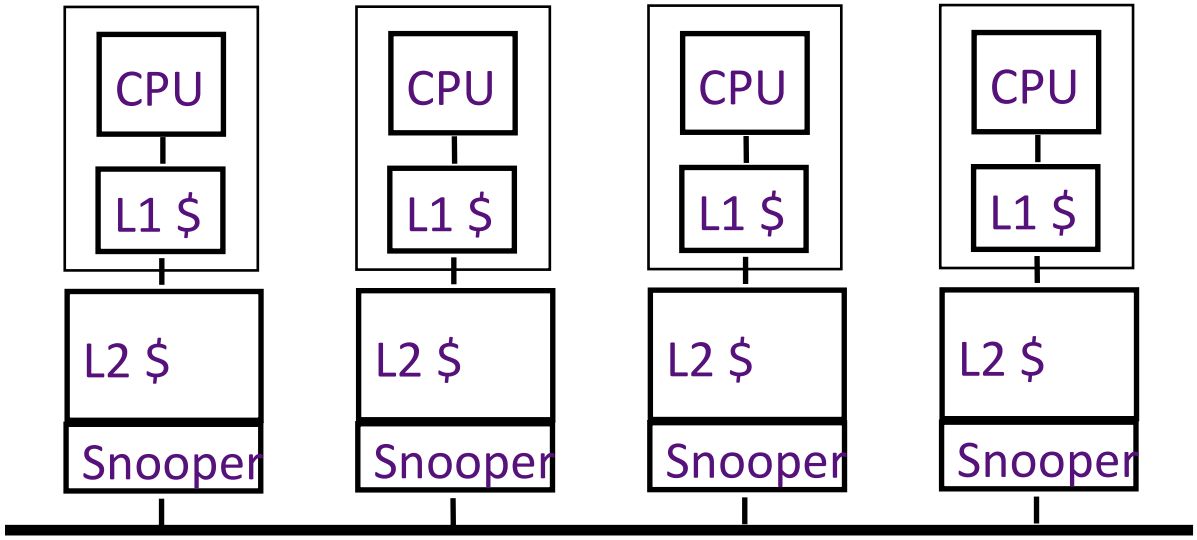
RED I only

ORANGE I and II

GREEN II only

YELLOW I, II and III

Optimized Snoop with Level-2 Caches



- Processors often have two-level caches
  - small L1, large L2 (usually both on chip now)
- Inclusion property: entries in L1 must be in L2
  - Miss in L2 ⇒ Not present in L1
  - Only if invalidation hits in L2 ⇒ probe and invalidate in L1
- Snooping on L2 does not affect CPU-L1 bandwidth

Performance of Symmetric Multiprocessors (SMPs)

Cache performance is combination of:

- Uniprocessor cache miss traffic
- Traffic caused by communication
  - Results in invalidations and subsequent cache misses
- Coherence misses
  - Sometimes called a Communication miss
  - 4th C of cache misses along with Compulsory, Capacity, & Conflict.

Scaling Snoopy/Broadcast Coherence

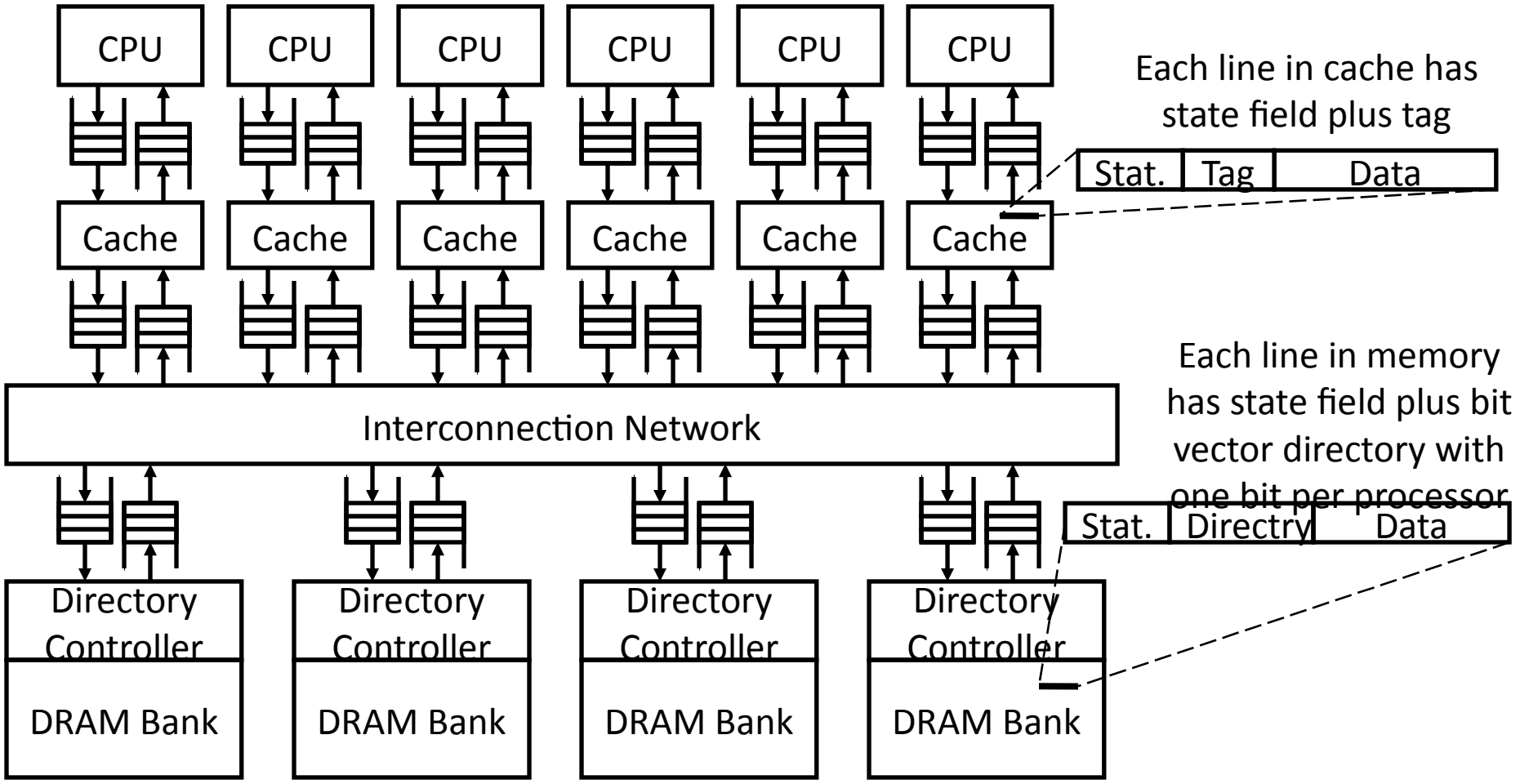
- When any processor gets a miss, must probe every other cache
- Scaling up to more processors limited by:
  - Communication bandwidth over bus
  - Snoop bandwidth into tags

- Can improve bandwidth by using multiple interleaved buses with interleaved tag banks
  - E.g, two bits of address pick which of four buses and four tag banks to use – (e.g., bits 7:6 of address pick bus/tag bank, bits 5:0 pick byte in 64-byte line)
- Buses don't scale to large number of connections, so can use point-to-point network for larger number of nodes, but then limited by tag bandwidth when broadcasting snoop requests.
- Insight: Most snoops fail to find a match!

Scalable Approach: Directories

- Every memory line has associated directory information
  - keeps track of copies of cached lines and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information

Directory Cache Protocol

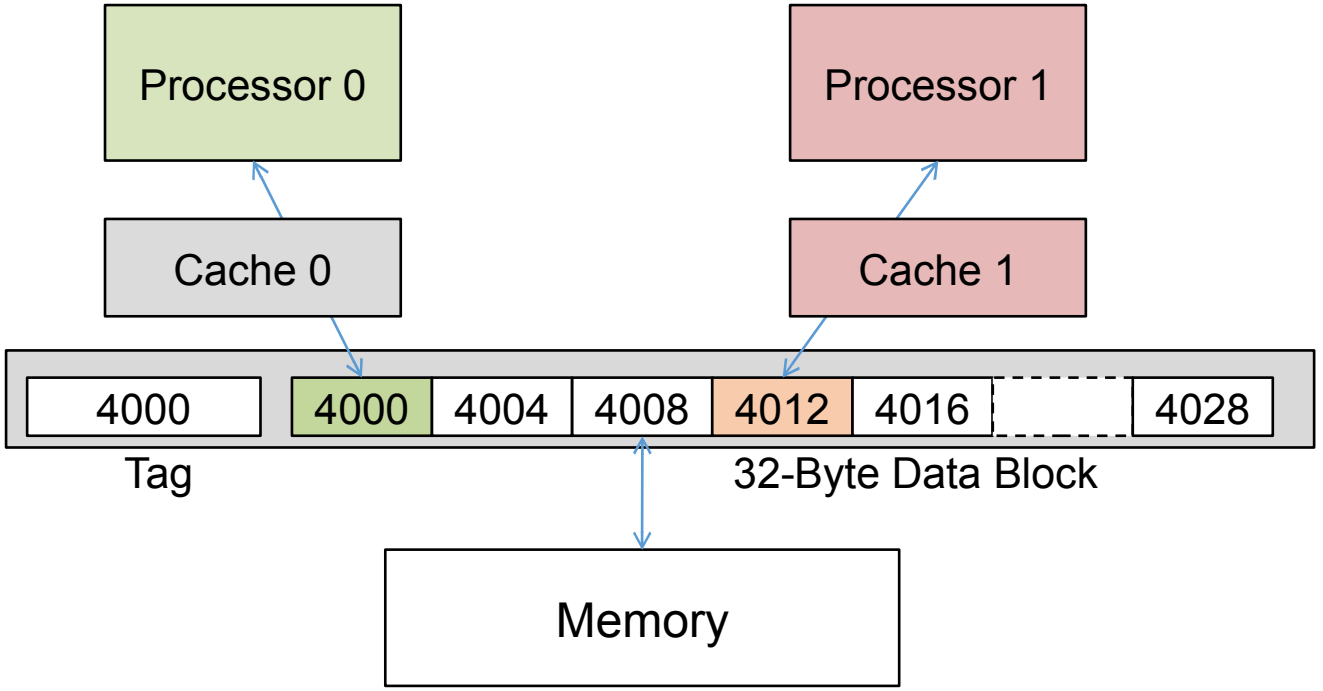


- Assumptions: Reliable network, FIFO message delivery between any given source-destination pair

Concurrency Management

- Protocol would be easy to design if only one transaction in flight across entire system
- But, want greater throughput and don't want to have to coordinate across entire system
- Great complexity in managing multiple outstanding concurrent transactions to cache lines
  - Can have multiple requests in flight to same cache line!

Cache Coherency Tracked by Block



- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose in X location 4000, Y in 4012
- What will happen?

Coherency Tracked by Cache Block

- Block ping-pongs between two caches even though processors are accessing disjoint variables

- Effect called *false sharing*

- How can you prevent it?

False Sharing

state line addr data0 data1 ... dataN

- A cache line contains more than one word
- Cache-coherence is done at the line-level and not word-level
- Suppose M1 writes wordi and M2 writes wordk and i ≠ k but both words have the same line address.
- What can happen?

Remember The 3Cs?

- Compulsory (cold start or process migration, 1st reference):
  - First access to block, impossible to avoid; small effect for long-running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)

- Capacity (not compulsory and…)
  - Cache cannot contain all blocks accessed by the program ***even with perfect replacement policy in fully associative cache***
  - Solution: increase cache size (may increase access time)

- Conflict (not compulsory or capacity and…):
  - Multiple memory locations map to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (may increase access time)
  - Solution 3: improve replacement policy, e.g.. LRU

Fourth “C” of Cache Misses!
Coherence Misses

- Misses caused by coherence traffic with other processor

- Also known as *communication* misses because represents data moving between processors working together on a parallel program

- For some parallel programs, coherence misses can dominate total misses

Coherency Misses

- True sharing misses arise from the communication of data through the cache coherence mechanism
  - Invalidates due to 1st write to shared line
  - Reads by another CPU of modified line in different cache
  - Miss would still occur if line size were 1 word

- False sharing misses when a line is invalidated because some word in the line, other than the one being read, is written into
  - Invalidation does not cause a new value to be communicated, but only causes an extra cache miss
  - Line is shared, but no word in line is actually shared
    → miss would not occur if line size were 1 word

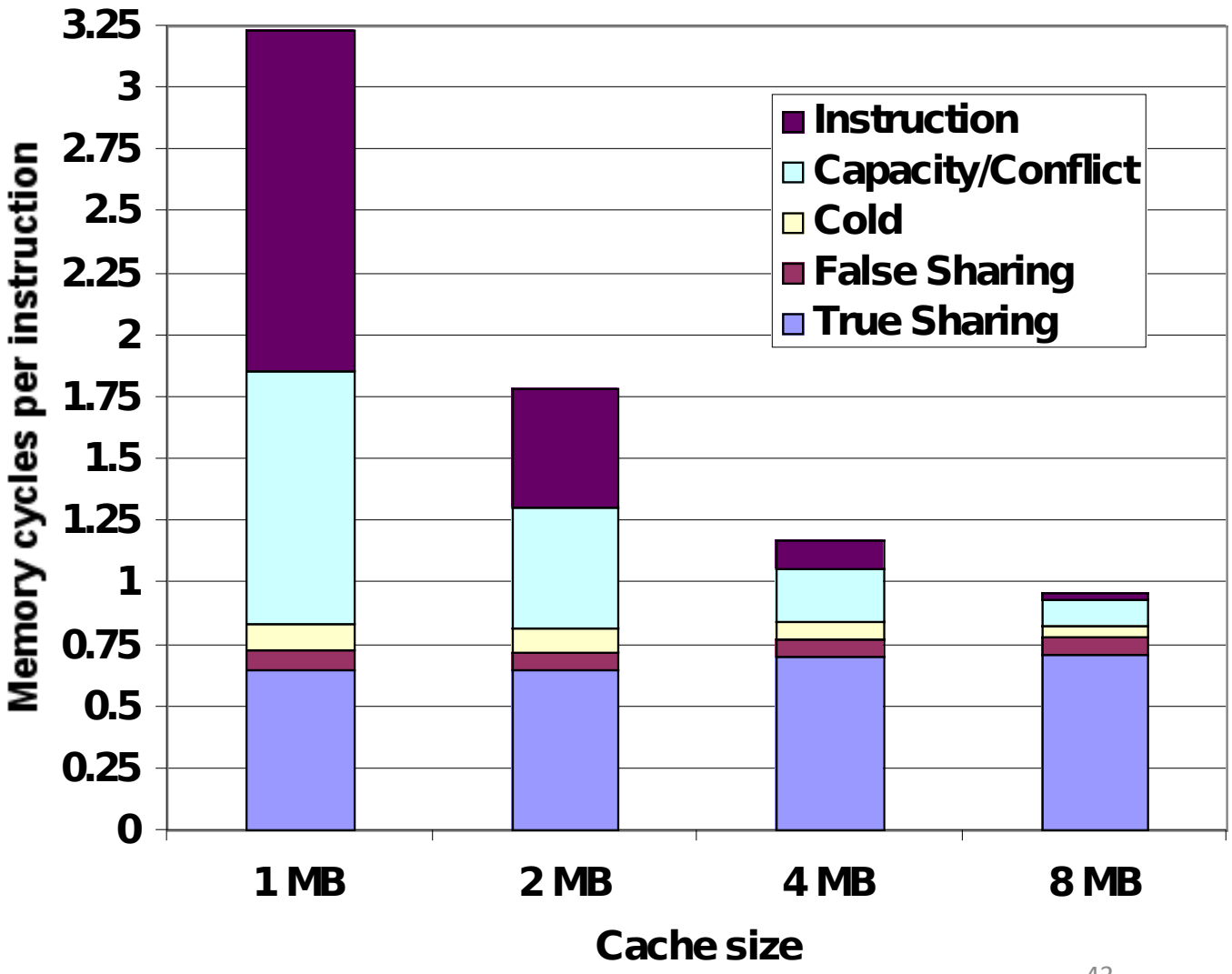
Example: True v. False Sharing v. Hit?

- Assume X and Y in same cache line.
  P1 and P2 both read X and Y before.

Request	P1 Cache State	P2 Cache State	Explanation
P1 Write X	Shared (X,Y)	Shared(X,Y)	
	Modified(X,Y)	Invalid(X,Y)	P1 Invalidates block(X,Y) in P2
P2 Read X	Modified(X,Y)	Invalid(X,Y)	TRUE Sharing Miss
	Shared (X,Y)	Shared(X,Y)	Write-back & Copy block from P1 to P2
P1 Write X	Shared (X,Y)	Shared(X,Y)	
	Modified(X,Y)	Invalid(X,Y)	P1 Invalidates block(X,Y) in P2
P2 Read Y	Modified(X,Y)	Invalid(X,Y)	False Sharing Miss
	Shared (X,Y)	Shared(X,Y)	Write-back & Copy block from P1 to P2
P1 Write X	Shared (X,Y)	Shared(X,Y)	
	Modified(X,Y)	Invalid(X,Y)	P1 Invalidates block(X,Y) in P2
P2 Write Y	Modified(X,Y)	Invalid(X,Y)	False Sharing Miss
	Invalid(X,Y)	Modified(X,Y)	Write-back & Copy block from P1 to P2
P1 Read Y	Invalid(X,Y)	Modified(X,Y)	TRUE Sharing Miss
	Shared (X,Y)	Shared(X,Y)	Write-back & Copy block from P2 to P1

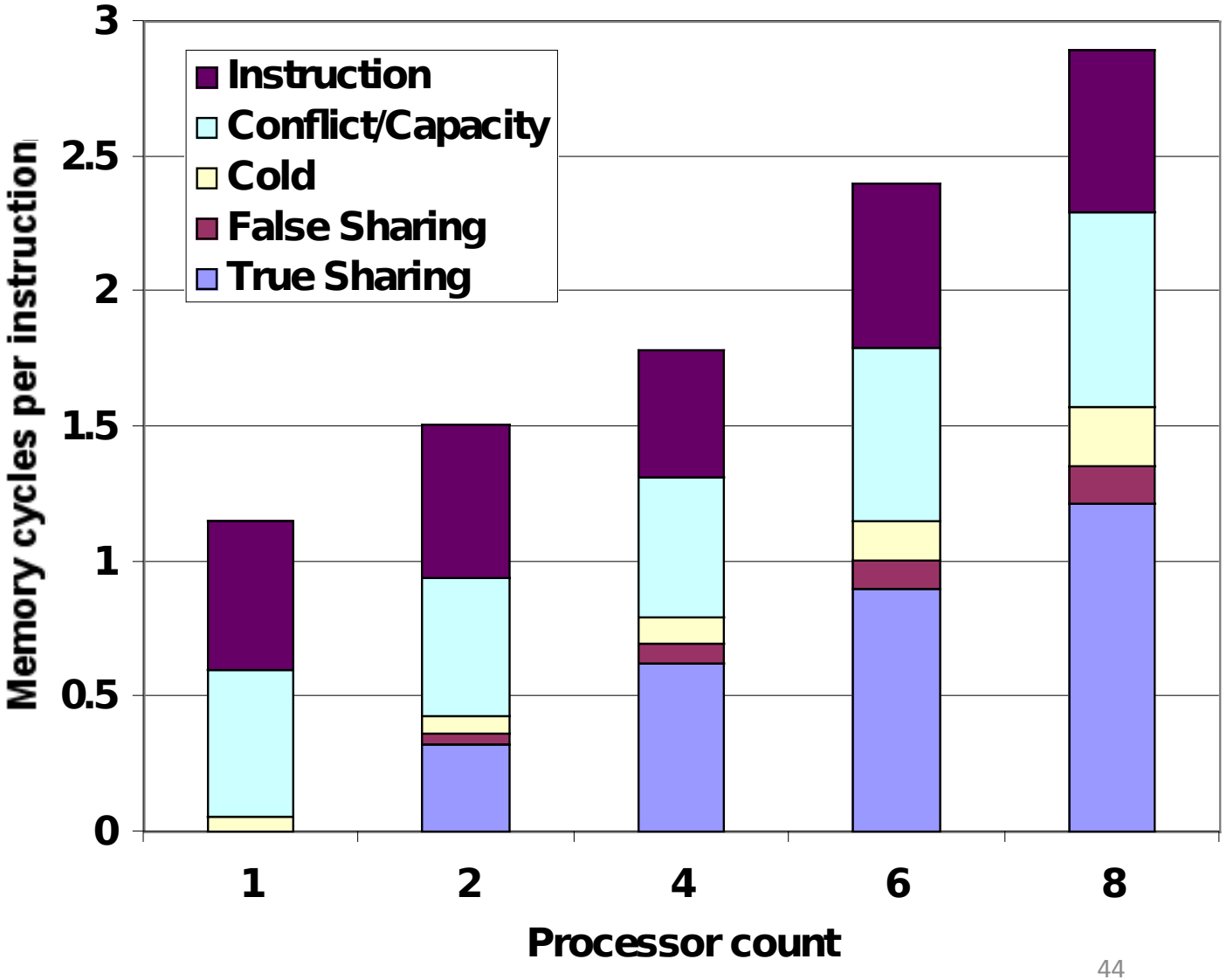
MP Performance 4-Processor Commercial Workload: OLTP, Decision Support (Database), Search Engine

- Uniprocessor cache misses improve with cache size increase (Instruction, Capacity/Conflict, Compulsory)
- True sharing and false sharing unchanged going from 1 MiB to 8 MiB (L3 cache)



MP Performance 2MiB Cache Commercial Workload: OLTP, Decision Support (Database), Search Engine

- True sharing, false sharing increase going from 1 to 8 CPUs



False Sharing in OpenMP

```
int i;
double  x, pi, sum[NUM_THREADS];
#pragma omp parallel private (i, x)
{  int id = omp_get_thread_num();
   for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREAD)
   {
     x = (i+0.5)*step;
     sum[id] += 4.0/(1.0+x*x);
   }
}
```

- What is the problem?
- Sum[0] is 8 bytes in memory, Sum[1] is adjacent 8 bytes in memory => false sharing if block size > 8 bytes

Peer Instruction: Avoid False Sharing

```
{  int i;       double  x, pi, sum[10000];
#pragma omp parallel private (i, x)
{  int id = omp_get_thread_num(),  fix = __________;
   for (i=id, sum[id]=0.0; i< num_steps; i=i+NUM_THREADS)
   {
     x = (i+0.5)*step;
     sum[id*fix] += 4.0/(1.0+x*x);
   }
```

- What is best value to set **`fix`** to prevent false sharing?

RED	**`omp_get_num_threads();`**
GREEN	Constant for number of blocks in cache
ORANGE	Constant for size of blocks in bytes
YELLOW	Constant for size of blocks in doubles

Types of Speedups and Scaling

- Scalability: adding x times more resources to the machine yields close to x times better "performance"
  - Usually resources are processors (but can also be memory size or interconnect bandwidth)
  - Usually means that with x times more processors we can get ~x times speedup for the same problem
  - In other words: How does efficiency (see Lecture 1) hold as the number of processors increases?

- In reality we have different scalability models:
  - Problem constrained
  - Time constrained

- Most appropriate scalability model depends on the user interests

Types of Speedups and Scaling

- Problem constrained (PC) scaling:
  - Problem size is kept fixed
  - Wall-clock execution time reduction is the goal
  - Number of processors and memory size are increased
  - "Speedup" is then defined as:

$$S_{PC} = \frac{Time(1\ processor)}{Time(p\ processors)}$$

  - Example: Weather simulation that does not complete in reasonable time

Example: Weather simulation that does not complete in reasonable time

Types of Speedups and Scaling

- Time constrained (TC) scaling:
  - Maximum allowable execution time is kept fixed
  - Problem size increase is the goal
  - Number of processors and memory size are increased
  - "Speedup" is then defined as:

$$S_{TC} = \frac{Work(p\ processors)}{Work(1\ processor)}$$

  - Example: weather simulation with refined grid
  - Example: weather simulation with refined grid

And, in Conclusion, …

- OpenMP as simple parallel extension to C
  - Threads level programming with **`parallel for`** pragma, **`private`** variables, **`reductions`**, …
  - ≈ C: small so easy to learn, but not very high level and it's easy to get into trouble

- ILP vs. TLP
  - CMP (Chip Multiprocessor aka Symmetric Multiprocessor) vs. SMT (Simultaneous Multithreading)
  - Cache coherency implements shared memory even with multiple copies in multiple caches
  - False sharing a concern; watch block size!