Parallel Processors

ENCS5331: Advanced Computer Architecture

Fall 2024/2025

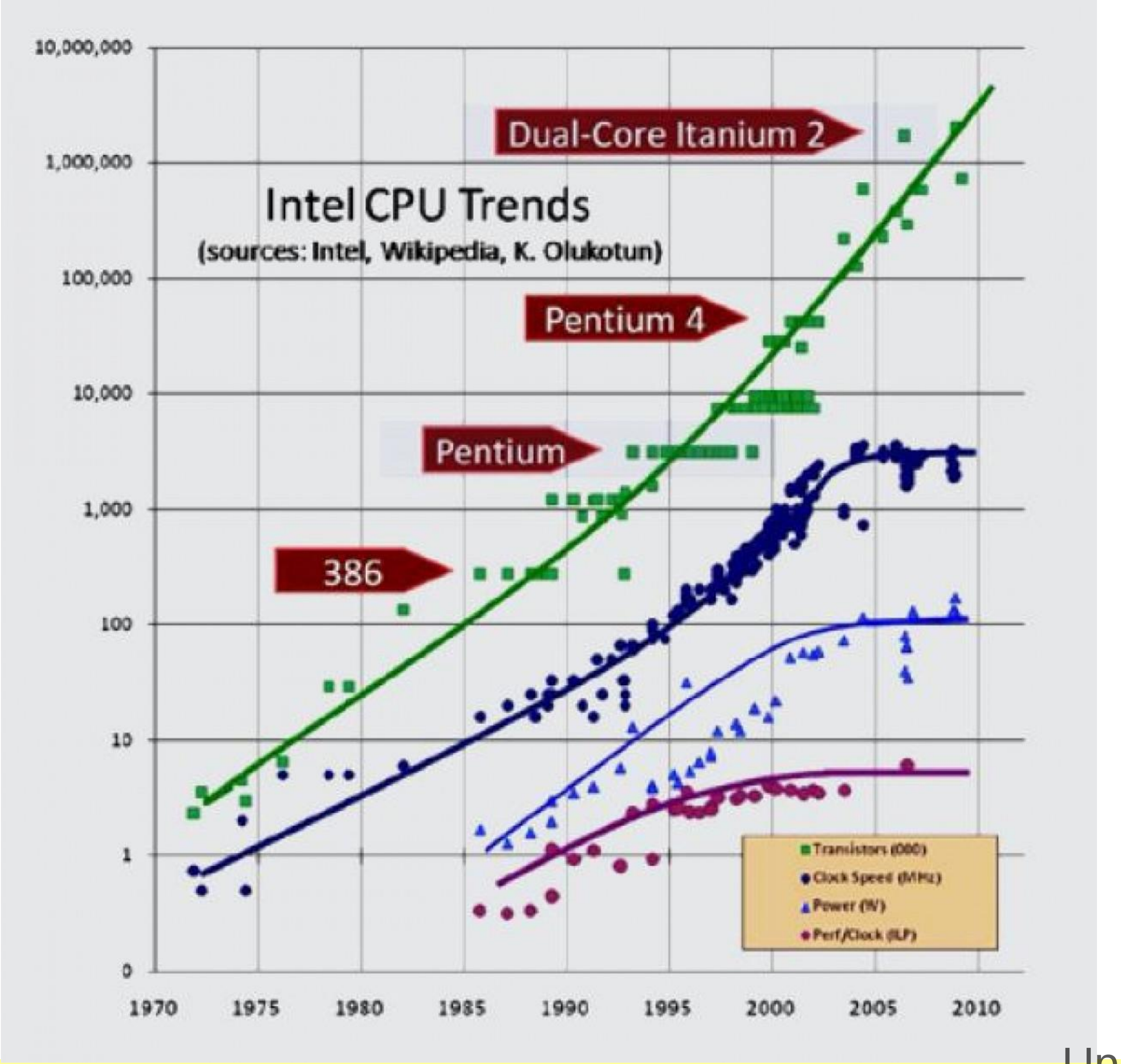
Instructor: Dr. Ayman Hroub

Special Thanks to Dr. Muhamed Mudawar (KFUPM) for most of the Slides

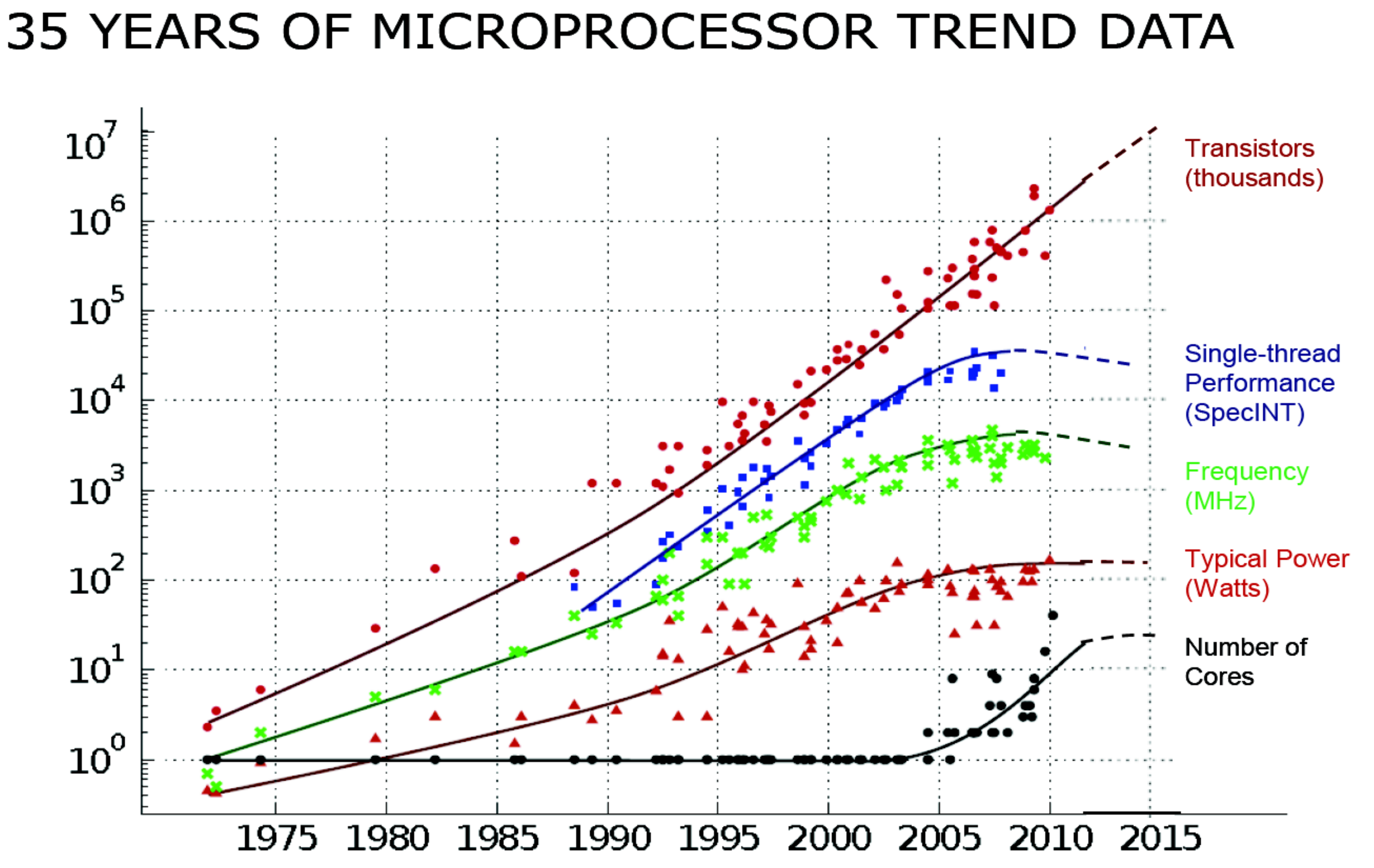
Presentation Outline

- **Introduction to Parallel Processors**
- Challenges of Parallel Programming
- Cache Coherence
- Directory Cache Coherence
- Synchronization

Motivation to Move to Multicore – Technology Constraints



Processor Trends



Original data collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond and C. Batten
Dotted line extrapolations by C. Moore

Single-Core Performance Walls

- ❖ Power Wall: cannot make the processor clock faster
- ❖ ILP (Instruction Level Parallelism) Wall
- ❖ Memory Wall

What is a Multiprocessor?

- “A parallel computer is a collection of processing elements that cooperate and communicate to solve large problems fast.”
  - Almasi and Gottlieb, Highly Parallel Computing,1989

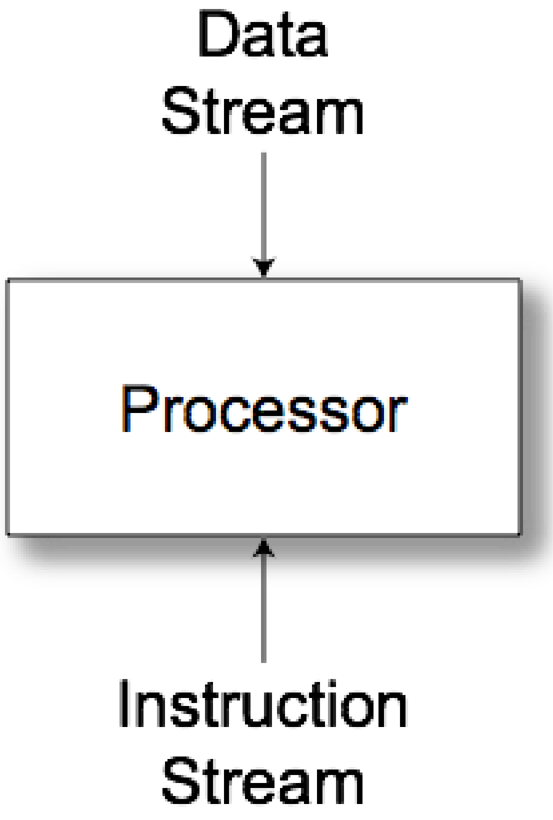
- Collection of processors, memories, and storage devices
  - That communicate and cooperate to solve large problems fast

Flynn's Taxonomy (1966)

- **SISD**: Single instruction stream, single data stream
  - Uniprocessors
- **SIMD**: Single instruction stream, multiple data streams
  - Same instruction is executed on different data
  - Exploits Data-Level Parallelism (DLP)
  - Vector processors, and Graphics Processing Units (GPUs)
- **MISD**: Multiple instruction streams, single data stream
  - No commercial implementation
- **MIMD**: Multiple instruction streams, multiple data streams
  - Most general and flexible architecture for parallel applications
  - Exploits Thread-Level Parallelism (TLP) and Data-Level Parallelism
  - Tightly-coupled versus loosely-coupled MIMD

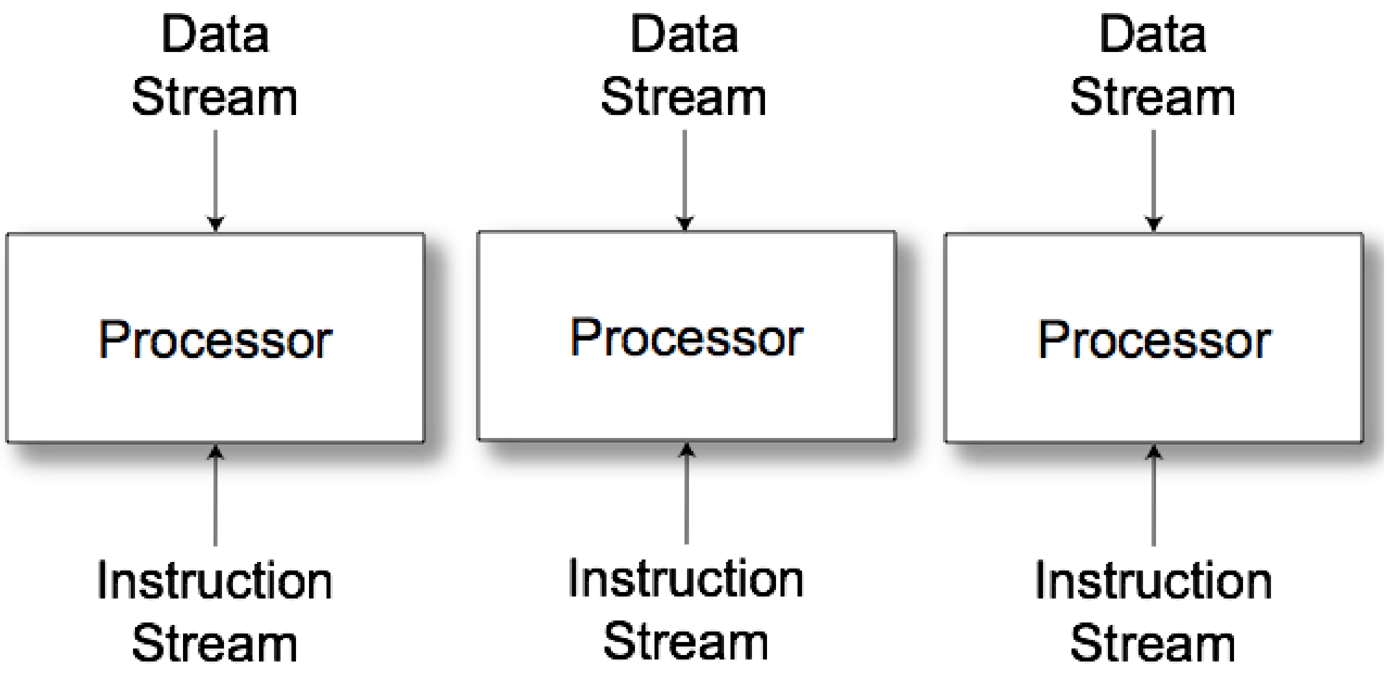
SISD

- Uniprocessor



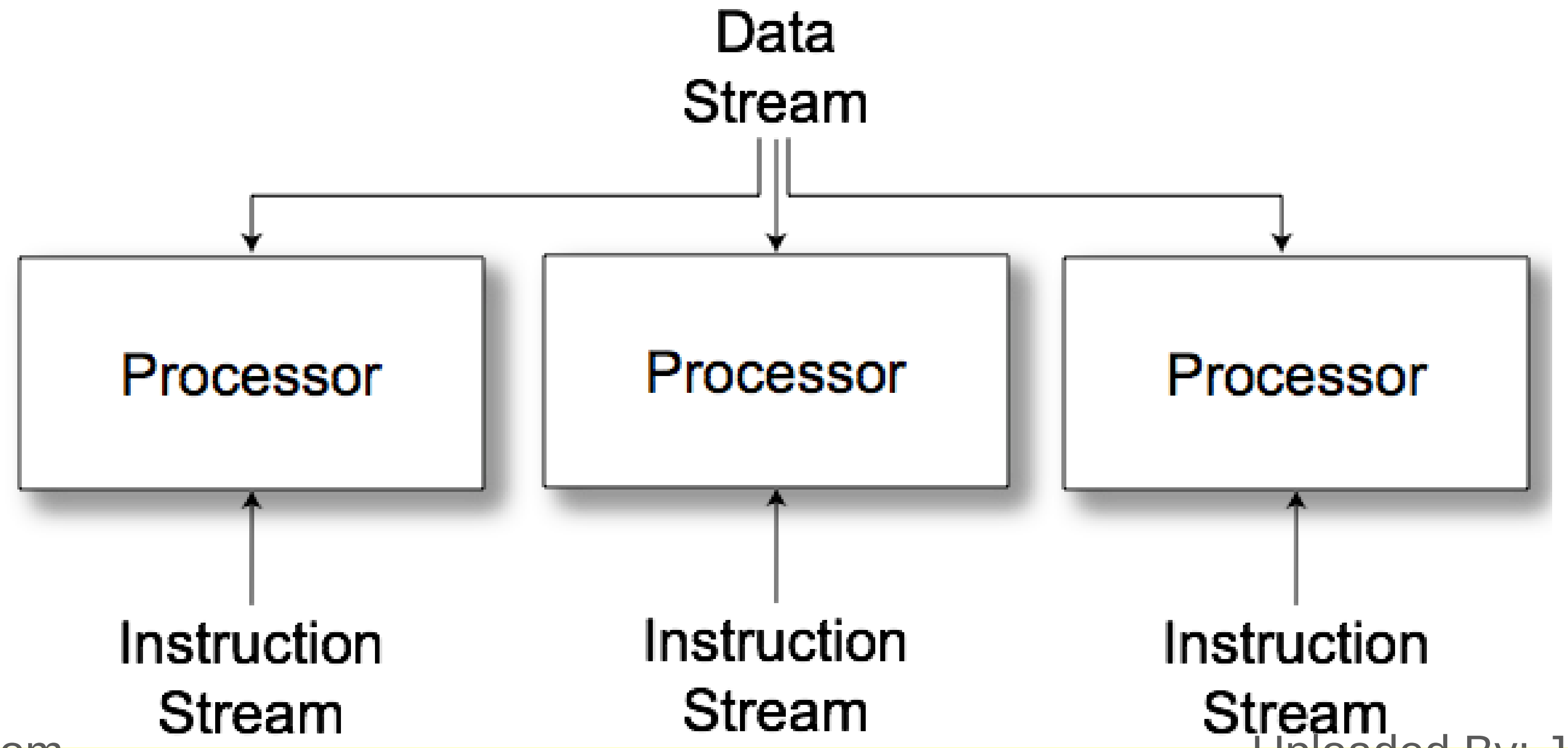
MIMD

❖ Multiple communicating processes



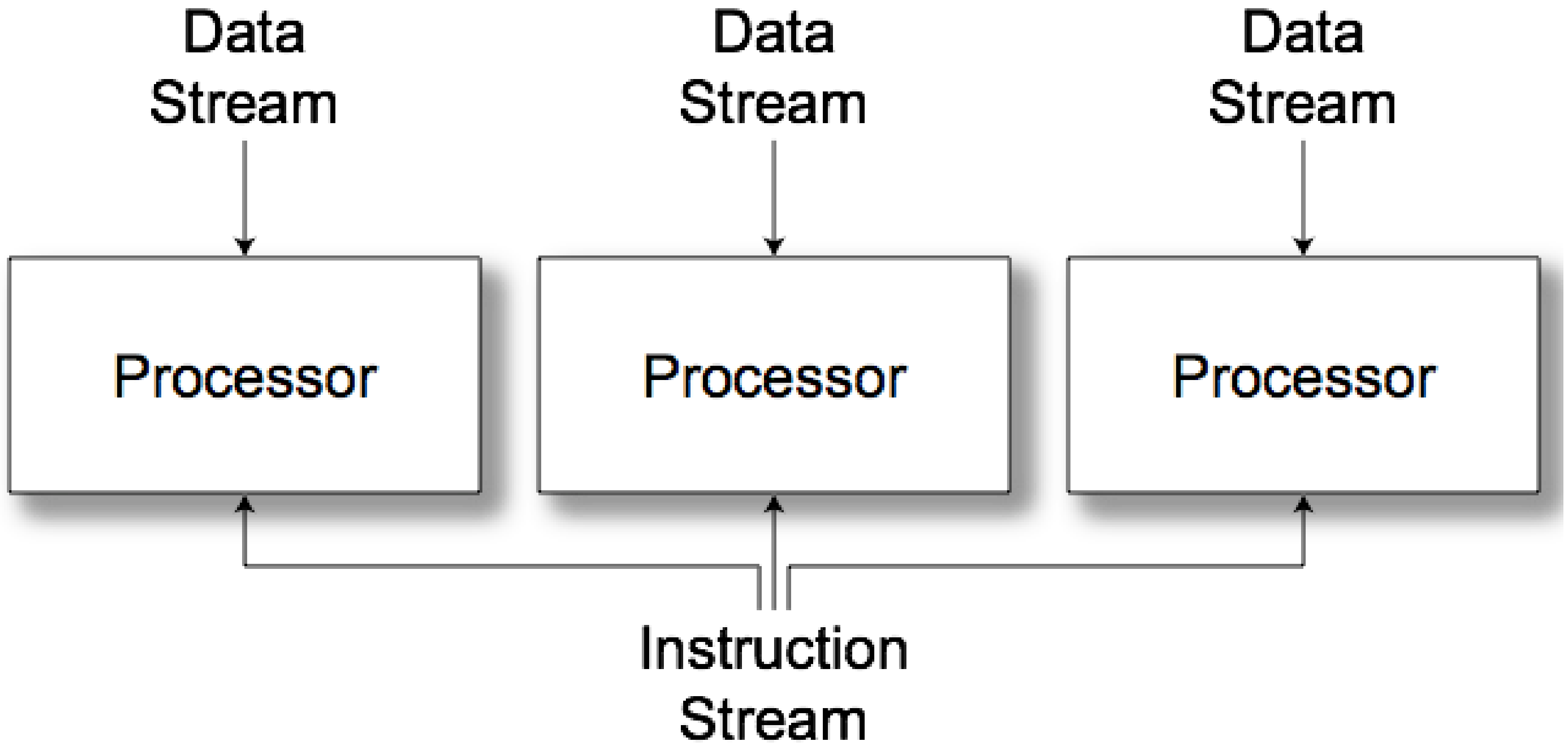
MISD

- ❖ No commercial examples
- ❖ Different operations to a single data set
  - ✧ Find primes
  - ✧ Crack passwords



SIMD

- Vector/Array computers

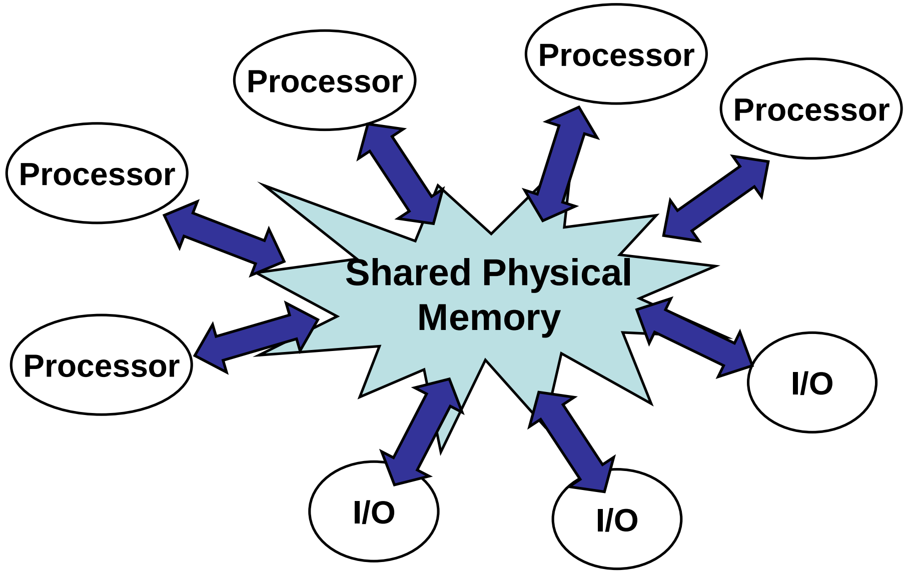


Major Multiprocessor Organizations

- **Symmetric Multiprocessors (SMP)**
  - Main memory is shared and equally accessible by all processors
  - Called also Uniform Memory Access (UMA)
  - Bus based or interconnection network based
- **Distributed Memory Multiprocessors**
  - Distributed Shared Memory (DSM) multiprocessors
    - Memory is distributed and shared and accessed by all processors
    - Non-uniform memory access (NUMA)
    - Latency varies between local and remote memory access
  - Message-Passing multiprocessors (Clusters)
    - Memory is distributed, but NOT shared
    - Each processor can access its own local memory
    - Processors communicate by sending and receiving messages

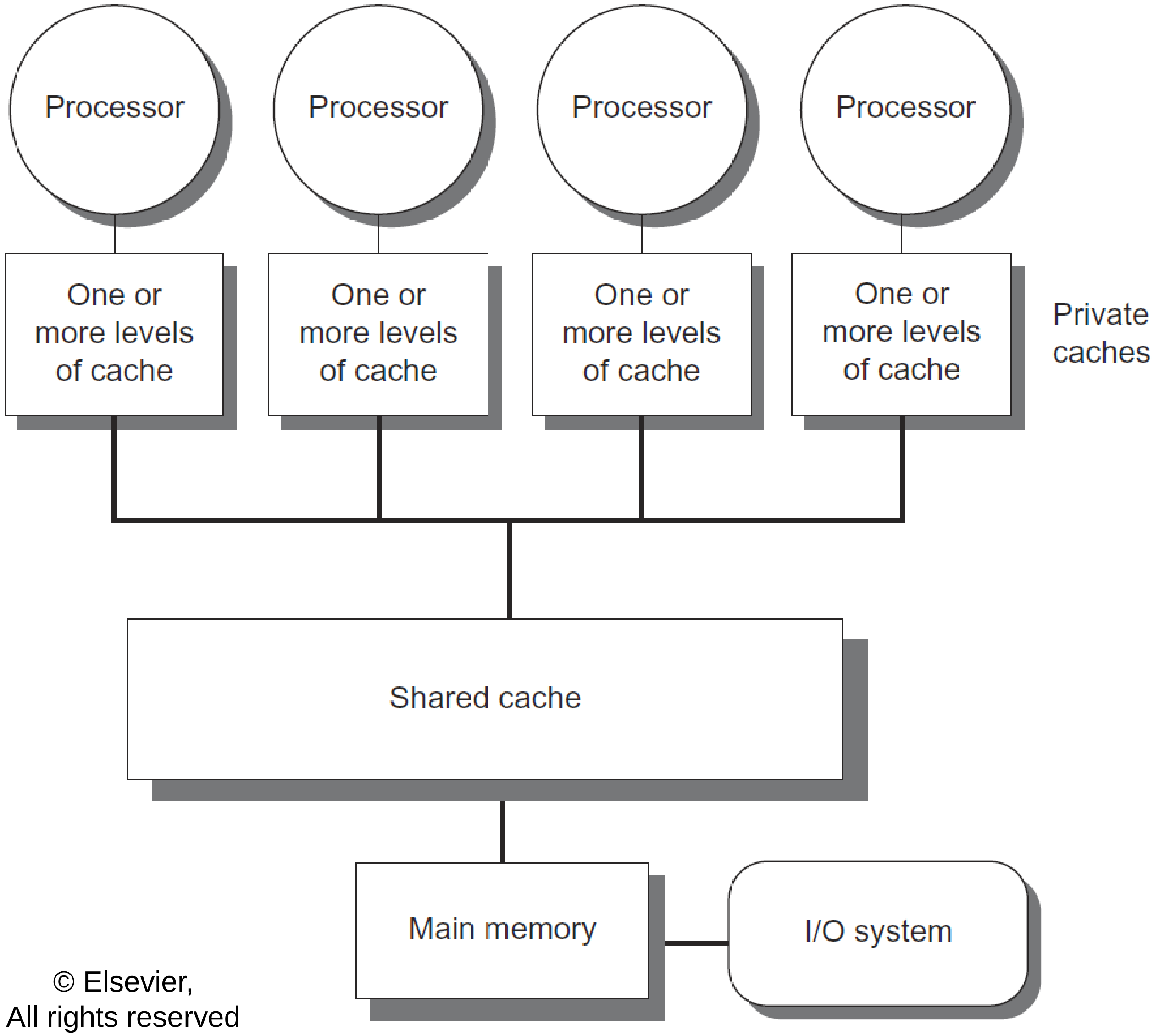
Shared Memory Architecture

- ❖ Any processor can directly reference any physical memory
- ❖ Any I/O controller to any physical memory
- ❖ Operating system can run on any processor
  - ✧ OS uses shared memory to coordinate
- ❖ Communication occurs implicitly as result of loads and stores
- ❖ Wide range of scale
  - ✧ Few to hundreds of processors
  - ✧ Memory may be physically distributed among processors
- ❖ History dates to early 1960s

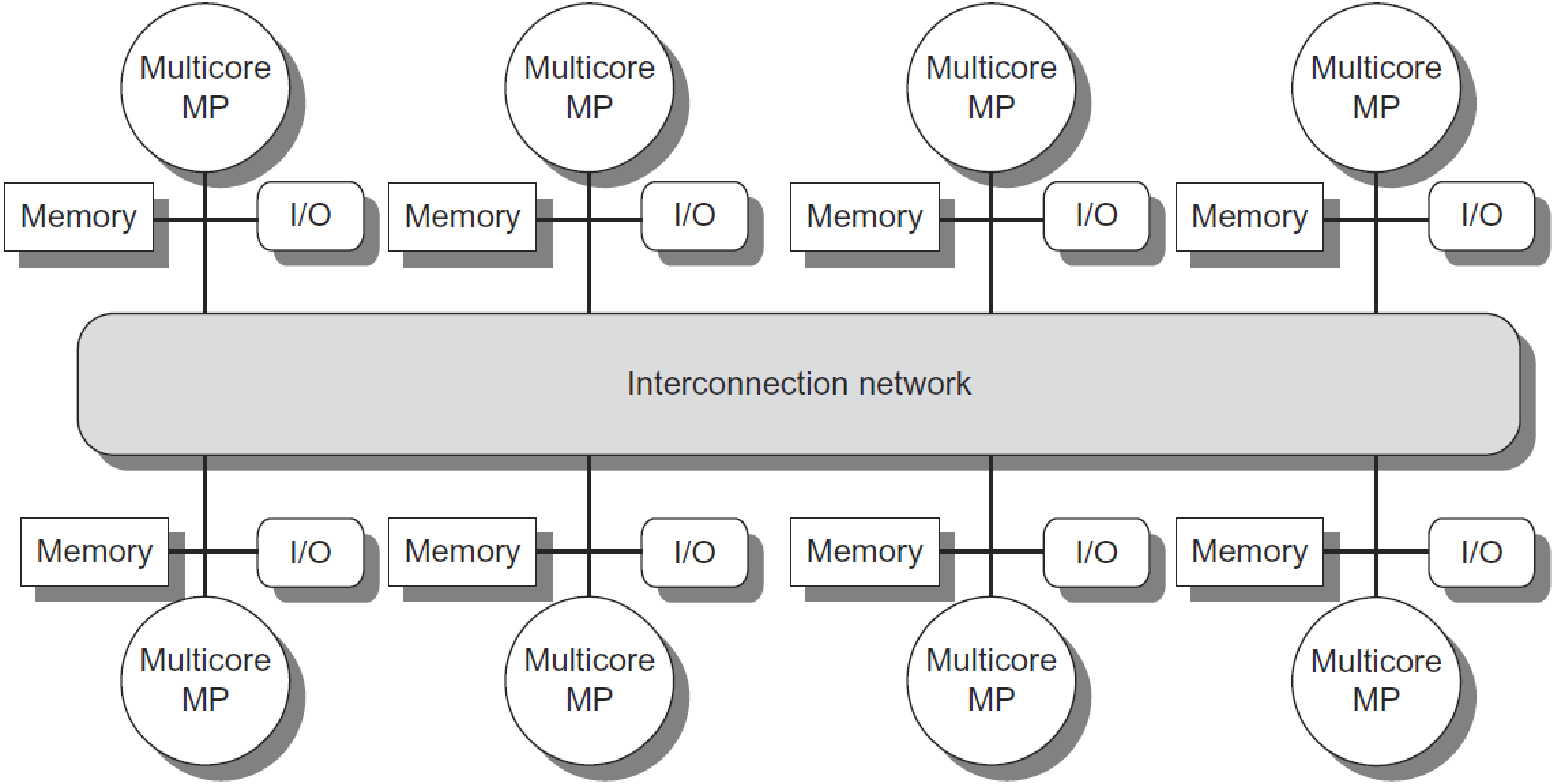


Single-Chip Multiprocessor (Multicores)

- ❖ Multiprocessor on a single chip (called multicores)
- ❖ Each core is a processor with private caches
- ❖ All processors share a common cache on chip
- ❖ Uniform access to shared cache and main memory



Distributed Memory Multiprocessors



- Memory is **distributed** among all processors
- Interconnection network connects all the (Multicore MP) nodes

Distributed Memory Architectures

- Distributed Shared Memory (tightly coupled)
  - Distributed memories are **shared** among all processors
  - Processors can access local and remote memories
  - Remote memory access over interconnection network
  - Non-uniform memory access (NUMA)
- Message Passing (loosely coupled)
  - Distributed memories are **NOT shared**
  - Processors cannot access remote memories
  - Multiple private physical address spaces
  - Easier to build and scale than distributed shared memory
  - Message passing communication over network

Multiprocessor Communication Models

- Shared Memory Architectures
  - One global physical address space
  - Distributed among all physical memories
  - Any processor can read/write any physical memory
  - Processors communicate using load and store instructions
  - Non-Uniform Memory Access (**NUMA**) for large-scale multiprocessors
- Message Passing Architectures (Clusters)
  - Separate physical address spaces for nodes
  - A compute node consists of one (or few) multicore chips and memory
  - A node cannot directly access the physical memory of another node
  - Nodes communicate via sending and receiving messages
  - Nodes are interconnected via a high-speed network

Next . . .

- Introduction to Multiprocessors
- **Challenges of Parallel Programming**
- Cache Coherence
- Directory Cache Coherence
- Synchronization

Parallel Programming Models (1)

- Parallel Task is the unit of parallel computation
- Two major parallel programming models

1. Shared Memory
   - Popular for small machines consisting of at most hundreds of cores
   - Parallel tasks are executed as **separate threads** on different cores
   - Threads communicate using **load and store** instructions to shared memory
   - Threads must **synchronize explicitly** to control their execution order

Parallel Programming Models (2)

2. Message Passing

- ✧ Popular for large machines consisting of hundreds of thousands of cores
- ✧ Parallel tasks cannot share memory, each task has its own memory
- ✧ Parallel tasks communicate explicitly using **send and receive** messages
- ✧ **Synchronization** is done **implicitly** using send and receive

Speedup Challenge: Amdahl's Law

❖ F = Fraction of the original **execution time** which is **parallelizable**

❖ P = Number of **Processors**

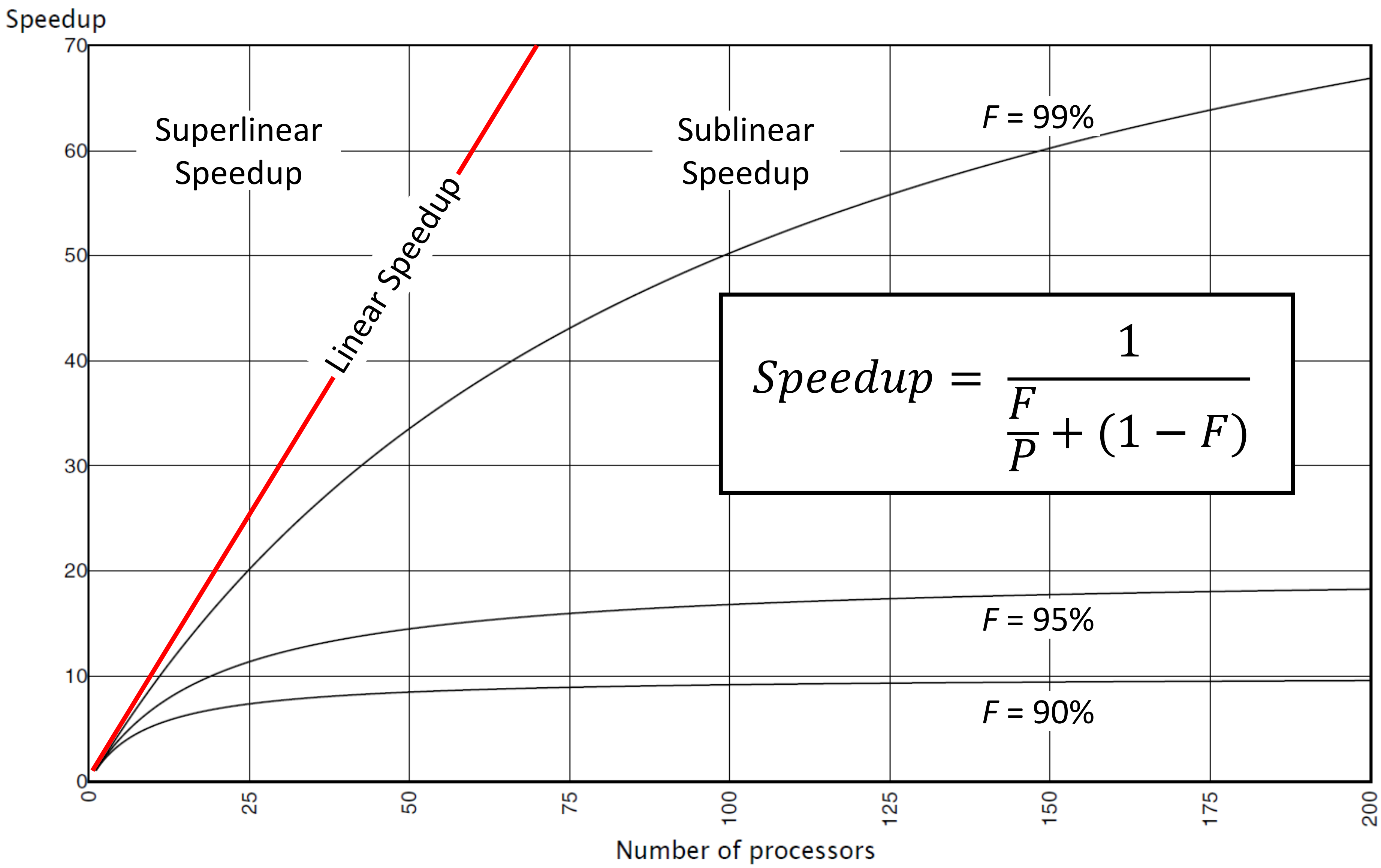
❖ $(1 - F)$ = Fraction of the execution time that cannot be parallelized

$$Speedup = \frac{1}{\frac{F}{P} + (1 - F)} \qquad \underset{P \to \infty}{Speedup} = \frac{1}{(1 - F)}$$

❖ Parallelism has an **overhead**

- ✧ Thread communication, synchronization, and load balancing
- ✧ Overhead increases with the number of processors P
- ✧ A large group of processors cannot be interconnected with short distances

Amdahl's Law Sublinear Speedup



$$Speedup = \frac{1}{\frac{F}{P} + (1 - F)}$$

Amdahl's Law: Example 1

- ❖ Want to achieve 50× speedup on 100 processors

  What fraction of the original execution time can be sequential?

- ❖ **Solution**

  $F_{parallel}$ = Fraction of the execution time, which is parallelizable

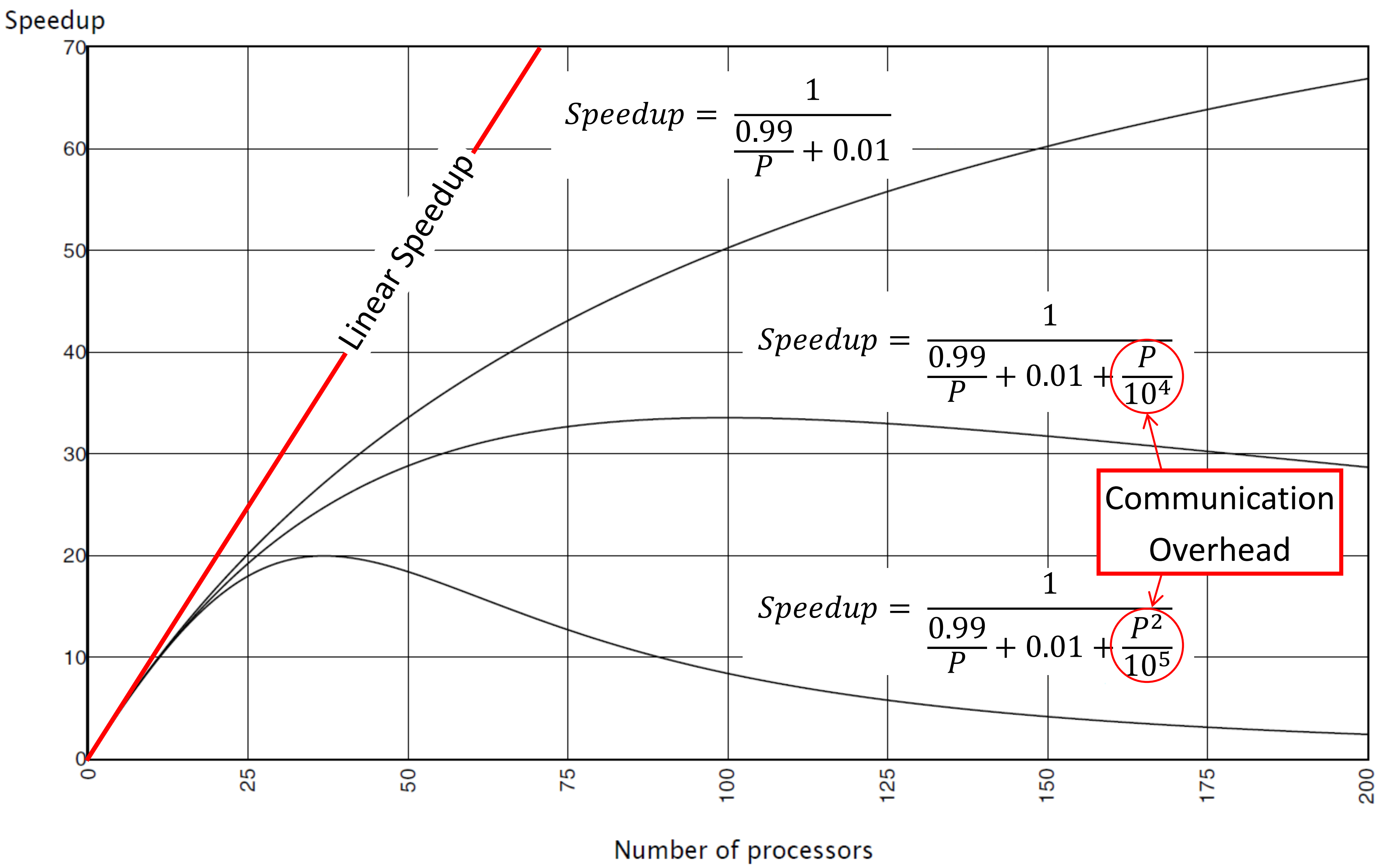
  $F_{sequential}$ = $(1 – F_{parallel})$ = Fraction of time, which is sequential

$$Speedup = \frac{1}{\frac{F_{parallel}}{100} + (1 - F_{parallel})} = 50$$

  Solving: $F_{parallel}$ = 98/99 ≅ 0.99 and $F_{sequential}$ = 0.01

- ❖ Sequential time is **at most 1%** of original execution time

Impact of Inefficient Communication on Speedup



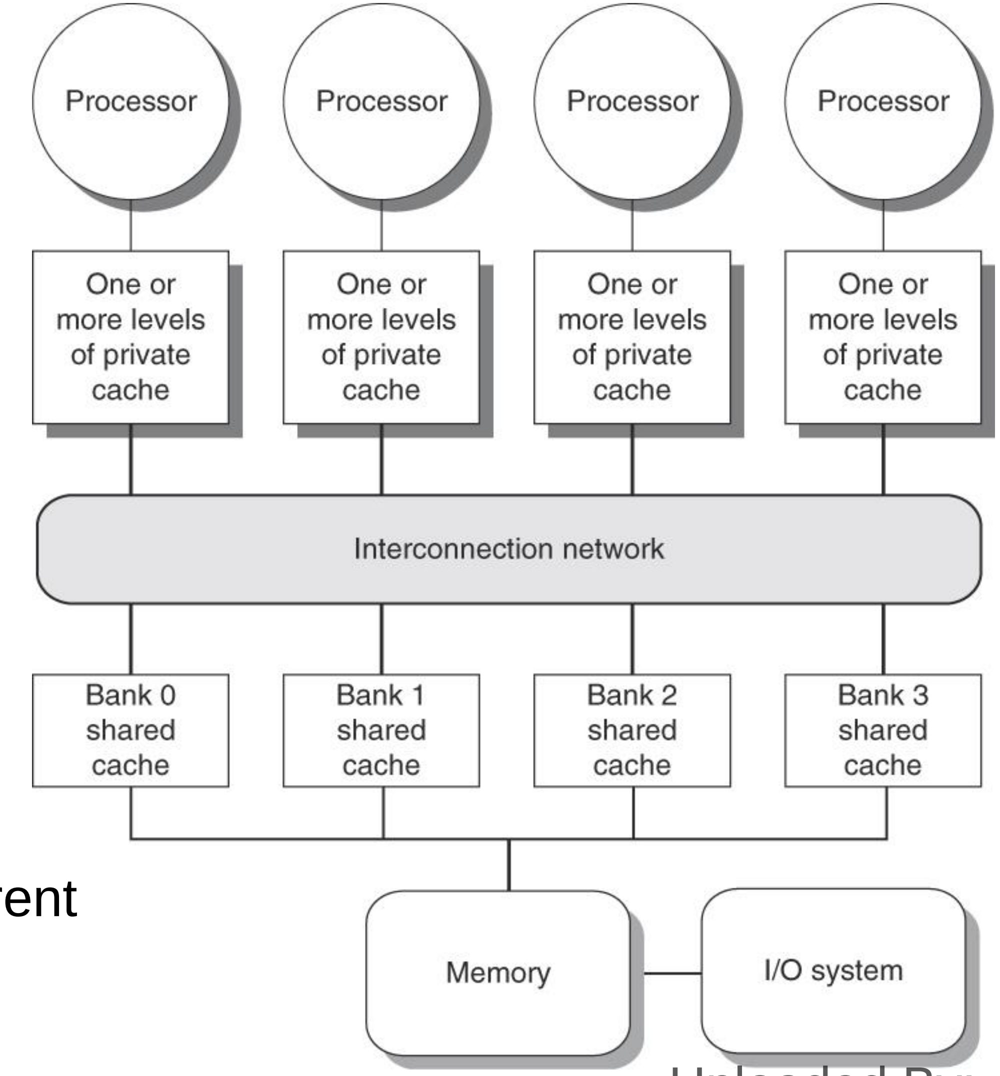
Next . . .

- Introduction to Multiprocessors
- Challenges of Parallel Programming
- **Cache Coherence**
- Directory Cache Coherence
- Synchronization

Caches in a Single-Chip Multiprocessor

- Private Caches are used inside processor cores
- Reduce average latency
  - Data is closer to processor
- Reduce traffic to memory
  - When data is cached
- Caching shared data
  - Shared data is replicated in multiple private caches
  - This requires maintaining copies of the shared data coherent

➔ **Cache coherence** problem



Cache Coherence Problem

- Private processor caches create the coherence problem
  - Copies of a shared variable can be present in multiple private data caches
- Updating copy of the shared data in one private data cache only
  - Other processors do not see the update!
  - Processors may read different data values through their caches
- Unacceptable to programming and is frequent!

Event	Memory variable X	Processor A Data Cache	Processor B Data Cache	Processor C Data Cache
Processor A reads X	4	4		
Processor B reads X	4	4	4	
Processor A stores X = 7	4	**7**	4	
Processor C reads X	4	**7**	4	4

Intuitive Coherent Memory Model

- Caches are supposed to be **transparent**
- What would happen if there were no caches?
  - All reads and writes would go to the main memory
  - Reading a location **X** should return the **last value written** to **X**
- What does **last value written** mean in a multiprocessor?
  - All operations on a **particular location X** would be **serialized**
  - All processors would **see the same access order** to location **X**

Formal Definition of Memory Coherence

A memory system is coherent if it satisfies two properties:

1. Write Propagation

   Writes by a processor become visible to other processors

   All reads by any processor to location X must return the most recently written value to location X, if the read and last write are sufficiently separated in time.

2. Write Serialization

   Writes to the same location X are serialized. Two writes to the same location X by any two processors are seen in the same order by all processors.

What to do about Cache Coherence?

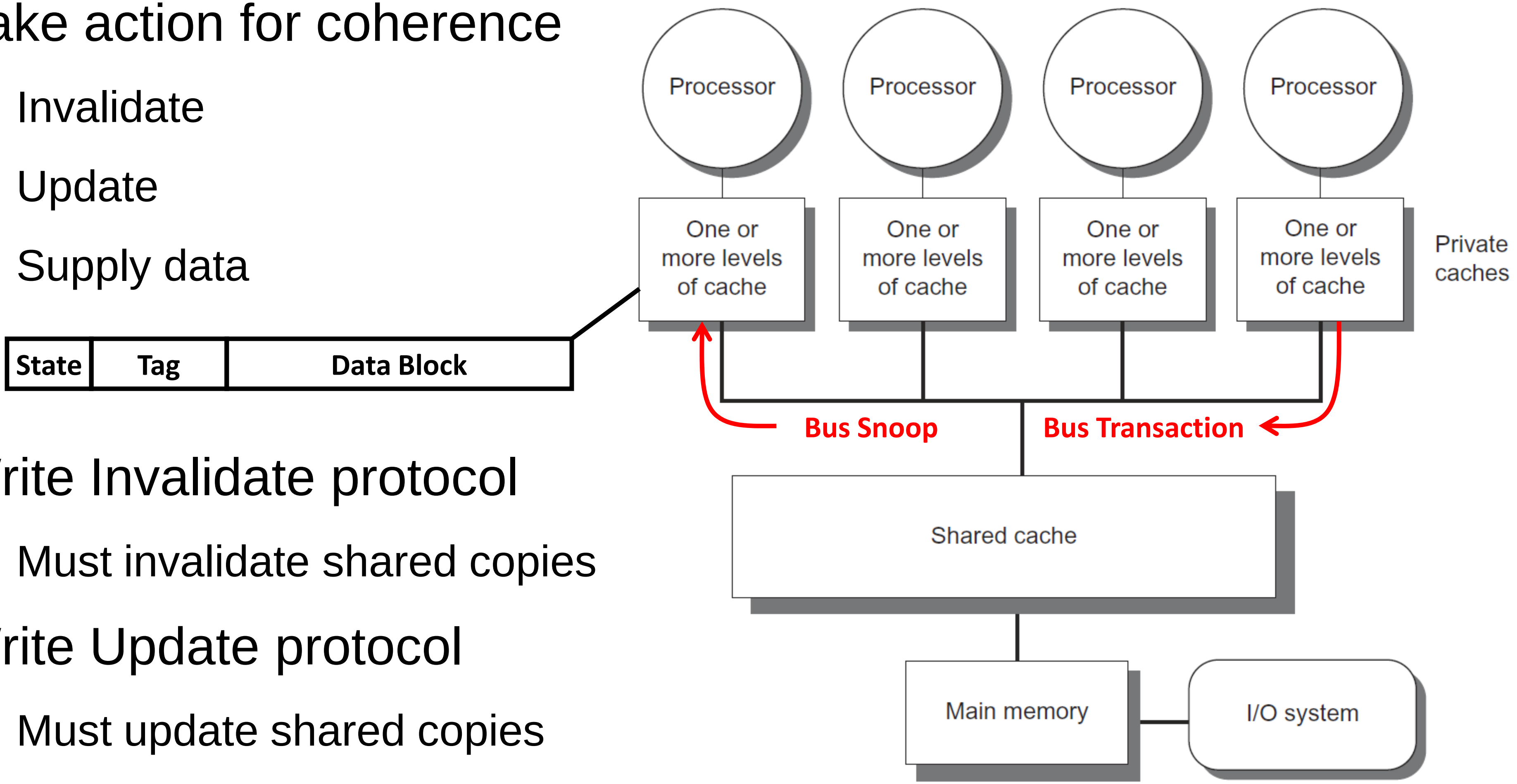
- Organize the memory hierarchy to make it go away
  - Remove private caches and use one cache shared by all processors
  - No private caches ➔ No replication of shared data
  - A switch (interconnection network) is needed to access shared cache
  - **Increases** the access **latency** of the shared cache
- Mark shared data pages as **uncacheable**
  - Shared data pages are not cached (must access memory)
  - Private data is cached only
  - We loose performance
- Detect and take actions to eliminate the problem
  - Can be addressed as a basic cache design issue

Hardware Cache Coherence Solutions

- Coherent Caches should provide
  - **Migration:** movement of shared data between processors
  - **Replication:** multiple copies of shared data (simultaneous reading)
- Cache Coherence Protocol
  - Tracking the sharing (replication) of data blocks in private caches
- Snooping Cache
  - Works well with small bus-based multiprocessors
  - Each cache monitors bus to track sharing status of each block
- Directory Based Schemes
  - Keep track of what is being shared in one place, called directory
  - Centralized memory ➔ Centralized directory
  - Distributed shared memory ➔ Distributed directory

Snooping Cache Coherence Protocols

- Cache controller **snoops** all transactions on the shared bus
- Transaction is **relevant** if address matches tag in the cache
- Take action for coherence
  - Invalidate
  - Update
  - Supply data



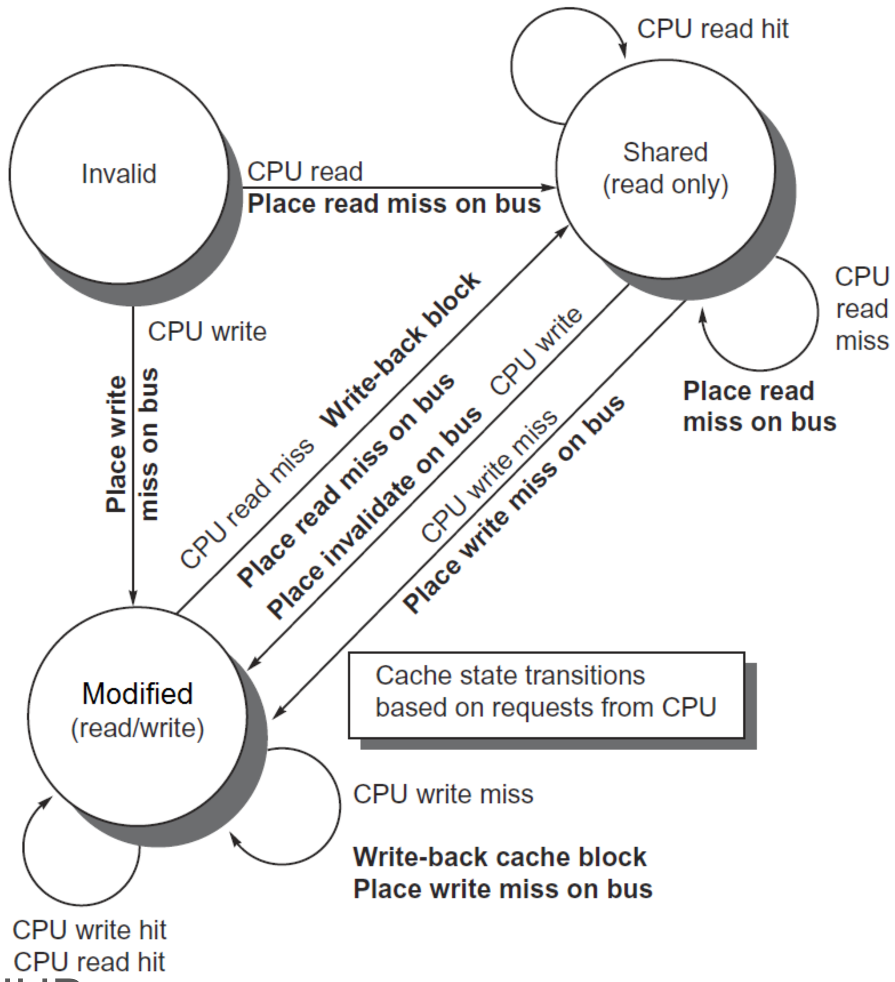
- Write Invalidate protocol
  - Must invalidate shared copies
- Write Update protocol
  - Must update shared copies

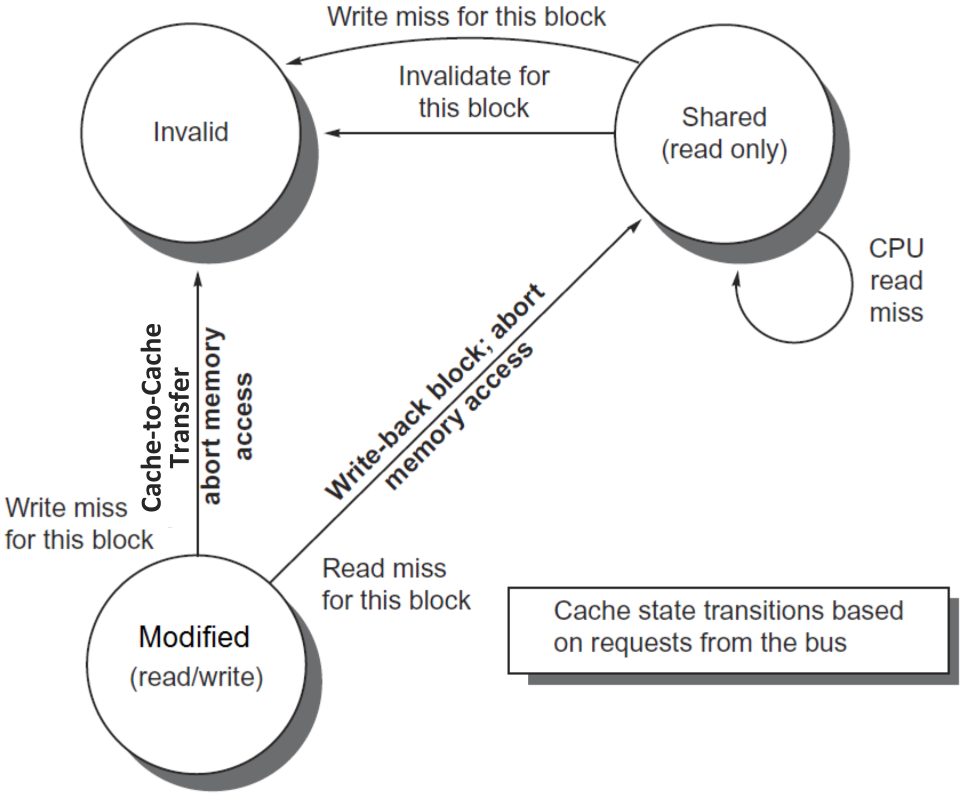
MSI Snoopy Cache Coherence Protocol

- Three States for write-back data cache
  1. **Modified**: only this cache has a **modified copy** of this block
  2. **Shared**: block is **read-only** and can be replicated in other caches
  3. **Invalid**: block is invalid
- Four Bus Transactions
  1. **Read Miss**: Service a read miss on bus
  2. **Write Miss**: Service a write miss on bus (obtain exclusive copy)
  3. **Invalidate**: Invalidate copies of this block in other caches
  4. **Write Back**: Write back a modified block on replacement
- On Write, invalidate all other copies
  - Write cannot complete until invalidate transaction appears on bus
  - Write serialization: transactions are serialized on bus

MSI Snoopy Cache Coherence Protocol

- Three Cache States: M (Modified), S (Shared), I (Invalid)
  - Only one cache can have block in Modified state





MSI Snoopy Cache Coherence Protocol

Request	Source	State	Action and Explanation
Read Hit	Processor	Shared or Modified	Normal Hit: Read data in local data cache (no transaction)
Read Miss	Processor	Invalid	Normal Miss: **Read miss on bus**, Wait for data, then change state to **Shared**
Read Miss	Processor	Shared	Replace block: Place **Read miss on bus**, Wait for data block, keep **Shared** state
Read Miss	Processor	Modified	Replace block: Place **Write-Back** block, Place **Read miss on bus**, Wait for data block, then change state to **Shared**
Write Hit	Processor	Modified	Normal Hit: Write data in local data cache (no transaction)
Write Hit	Processor	Shared	Coherence: Place **Invalidate on bus** (no data), then change state to **Modified**
Write Miss	Processor	Invalid	Normal Miss: Place **Write miss on bus**, wait for data, change state to **Modified**
Write Miss	Processor	Shared	Replace block: Place **Write miss on bus**, wait for data, change state **Modified**
Write Miss	Processor	Modified	Replace block: **Write-Back** block, Place **Write miss on bus**, wait for data block
Read Miss	Bus	Shared	No action: **Serve read miss** from shared cache or memory
Read Miss	Bus	Modified	Coherence: **Write-Back & Serve read miss**, then change state to **Shared**
Invalidate	Bus	Shared	Coherence: **Invalidate shared block** in other caches (change state to **Invalid**)
Write Miss	Bus	Shared	Coherence: **Invalidate shared block** in other caches, **Serve write miss**
Write Miss	Bus	Modified	Coherence: **Serve write miss** (cache-to-cache transfer) and **Invalidate block**

Example on MSI Cache Coherence

Request	Processor P1			Processor P2			Bus			Shared Cache		
	State	Addr	Value	State	Addr	Value	Proc	Addr	Action	State	Addr	Value
P1: Read A1							P1	A1	Rd Miss	S	A1	15
	S	A1	15									
P2: Read A1							P2	A1	Rd Miss	S	A1	15
	S	A1	15	S	A1	15						
P2: Write 10 to A1							P2	A1	Invalidate			
	I	A1	15	M	A1	10				I	A1	15
P1: Read A1							P1	A1	Rd Miss			
	S	A1	10	S	A1	10	P2	A1	Wr Back	M	A1	10
P1: Write 20 to A1							P1	A1	Invalidate			
	M	A1	20	I	A1	10				I	A1	10
P2: Write 35 to A1							P2	A1	Wr Miss			
	I	A1	20	M	A1	35	P1	A1	Transfer	I	A1	10

Alternative Cache Coherence Protocols

- **MESI** Cache Coherence Protocol: Four States
  1. **Modified**: only this cache has a **modified copy** of this block
  2. **Exclusive**: only this cache has a **clean copy** of this block
  3. **Shared**: block may be replicated in more than one cache (**read-only**)
  4. **Invalid**: block is invalid

  Exclusive State: prevents invalidate on a write hit (no bus transaction)

- **MOESI** Cache Coherence Protocol: Five States
  1. **Modified**: only this cache has a **modified copy** of this block
  2. **Owned**: this cache is owner, other caches can share block (**read-only**)
  3. **Exclusive**: only this cache has a **clean copy** of this block
  4. **Shared**: block may be replicated in more than one cache (**read-only**)
  5. **Invalid**: block is invalid

  Owner must supply data to others on a miss: **cache-to-cache** transfer

Next . . .

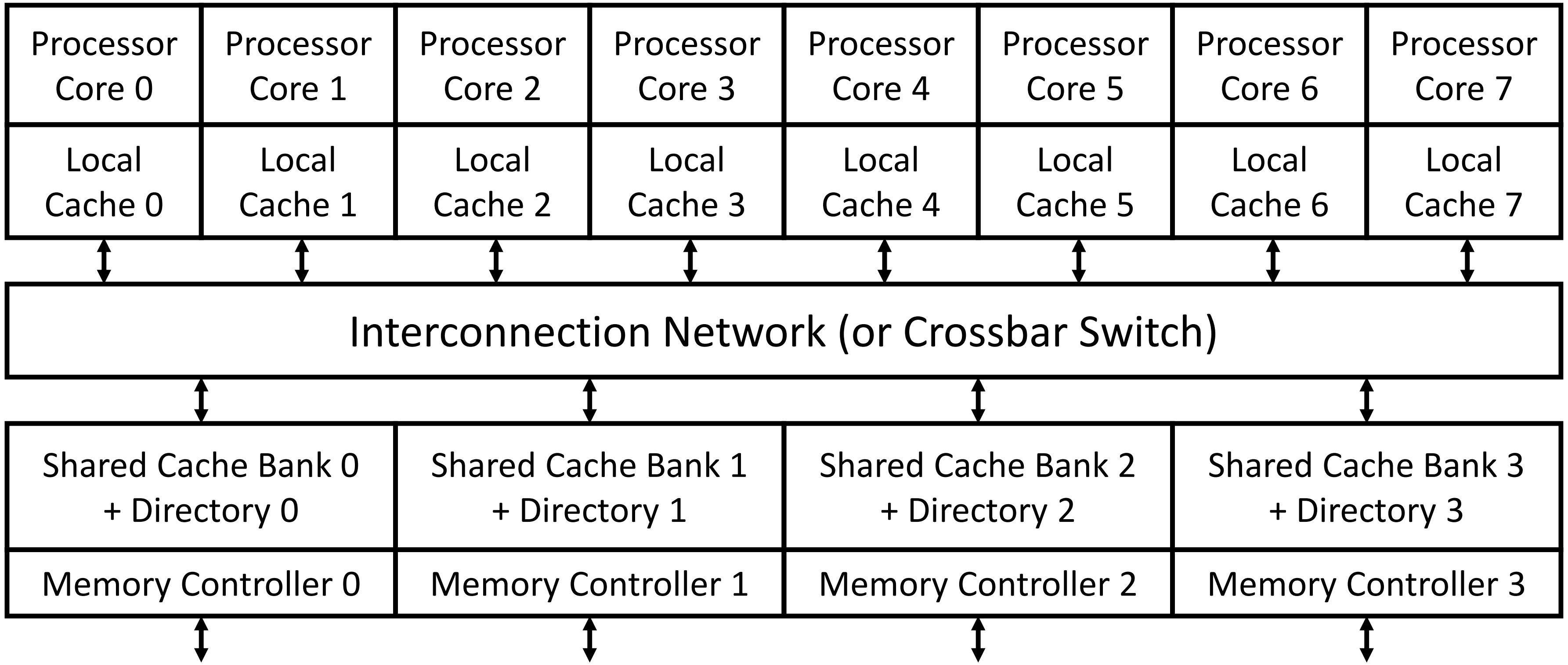
- ❖ Introduction to Multiprocessors
- ❖ Challenges of Parallel Programming
- ❖ Cache Coherence
- ❖ **Directory Cache Coherence**
- ❖ Synchronization

Limitations of Snooping Protocols

- Buses have limitations for scalability
  - Limited number of processor cores that can be attached to a bus
  - Contention on the use of the shared bus
  - Snooping bandwidth is a bottleneck for large number of processor cores
- On-Chip interconnection network ➔ Parallel communication
  - Multiple processor cores can access shared cache banks at the same time
  - Allows chip multiprocessor to scale beyond few processor cores
- Snooping is difficult on network other than bus or ring
  - Must broadcast coherence traffic to all processors, which is inefficient
- How to enforce cache coherence without broadcast?
  - Have a directory that records the state of each cached block
  - Directory entry specifies which private caches have copies of the block

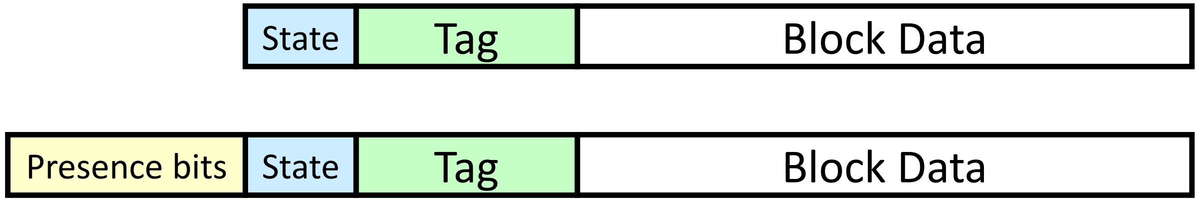
Directory in a Chip Multiprocessor

- Directory in outermost cache (shared by all processor cores)
  - Directory keeps track of copies of each block in local caches
- Outermost cache is split into multiple banks (parallel access)
  - Number of cache banks can vary (not related to number of cores)



Directory in the Shared Cache

- Shared Cache is **inclusive** with respect to all local caches
  - Shared cache contains a superset of the blocks in local caches
  - Example: Intel Core i7
- Directory is implemented in the shared cache
  - Each block in the shared cache is augmented with presence bits
  - If *k* processors then ***k* presence bits + state** per block in shared cache
  - Presence bits indicate which cores have a copy of the cache block
  - Each block has **state** information in private and shared cache
  - State = M (Modified), S (Shared), or I (Invalid) in local cache



Block in a Local Cache

Block in a Shared Cache

Terminology

- ❖ **Local (or Private) Cache (local node or requesting node)**
  - ✧ Where a processor request originates
- ❖ **Home Directory (home node)**
  - ✧ Where information about a cache block is kept
  - ✧ Directory uses presence bits and state to track cache blocks
- ❖ **Remote Cache (remote node)**
  - ✧ Has a copy of the cache block
- ❖ Cache Coherence ensures **Single-Writer, Multiple-Readers**
  - ✧ If a block is **modified** in a local cache then **one valid copy** can exist
    - ▪ Shared Cache and memory are not updated
- ❖ No bus and don't want to broadcast to all processor cores
  - ✧ All messages have explicit responses

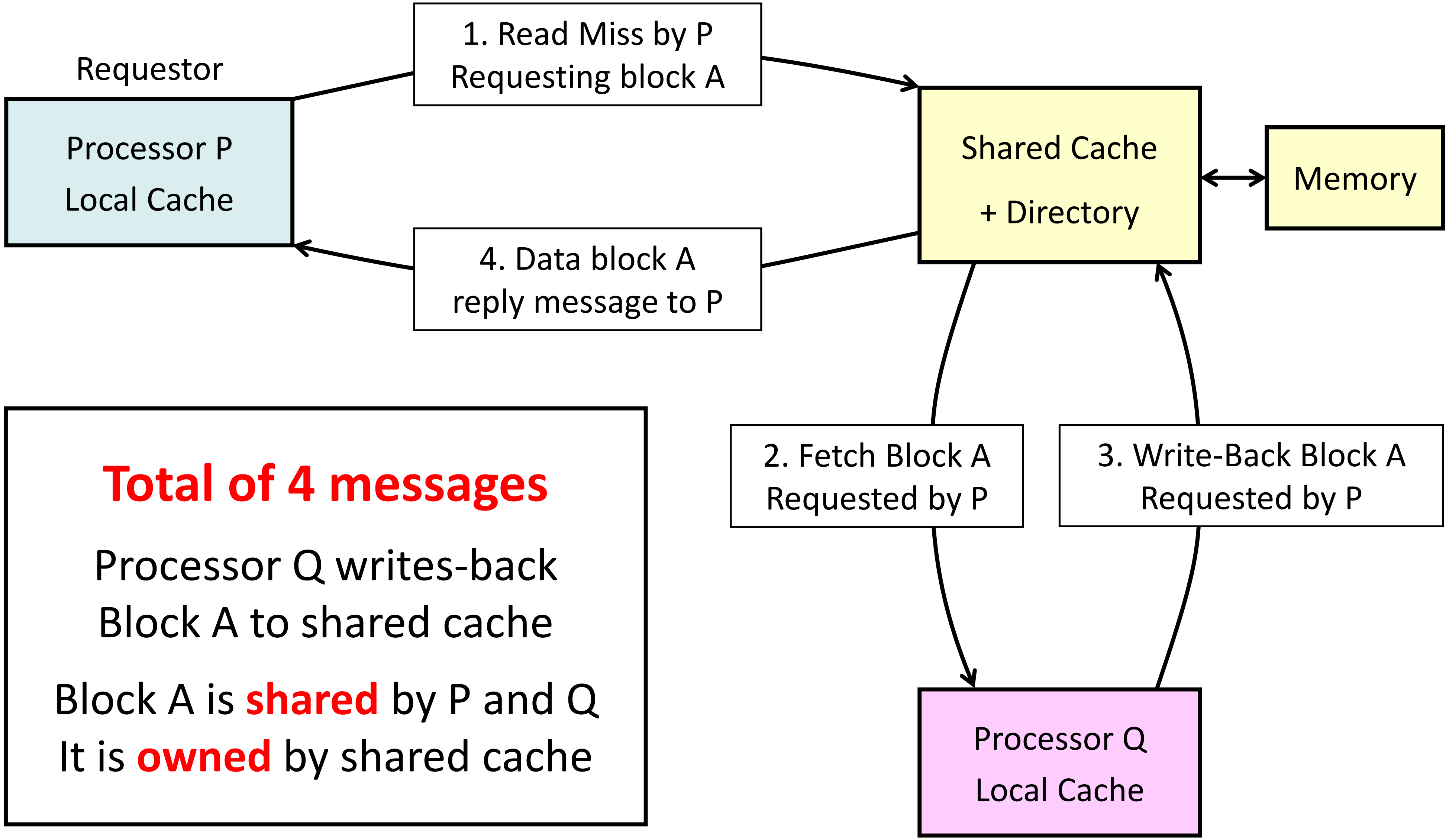
States for Local and Shared Cache

- Three states for a **local** (private) cache block:
  1. **Modified**: only this cache has a **modified copy** of this block
  2. **Shared**: block may be replicated in more than one cache (**read-only**)
  3. **Invalid**: block is invalid, not present in this local cache
- Four states for a **shared** cache block (directory):
  1. **Modified**: only one local cache is the owner of this block
     - One local cache (one presence bit) has a **modified copy** of this block
  2. **Owned**: shared cache is the owner of the modified block
     - Modified block was written-back to shared cache, but not to memory
     - A block in the owned state can be shared by multiple local caches
  3. **Shared**: block may be replicated in more than one cache (**read-only**)
  4. **Invalid**: block is invalid in the shared cache and in any local cache

Read Miss by Processor P

- Processor **P** sends **Read Miss** message to **home directory**
- Home Directory: block is **Shared** or **Owned**
  - Directory sends **data reply** message to **P**, and sets presence bit of **P**
  - Local cache of processor **P** changes state of received block to **shared**
- Home Directory: block is **Modified**
  - Directory sends **Fetch** message to remote cache that modified block
  - Remote cache sends **Write-Back** message to directory (shared cache)
  - Remote cache changes state of block to **shared**
  - Directory changes state of shared block to **owned**
  - Directory sends **data reply** message to **P**, and sets presence bit of **P**
  - Local cache of processor **P** changes state of received block to **shared**
- Home Directory: block is **Invalid** ➔ get block from memory

Read Miss to a Block in Modified State



Total of 4 messages

Processor Q writes-back Block A to shared cache

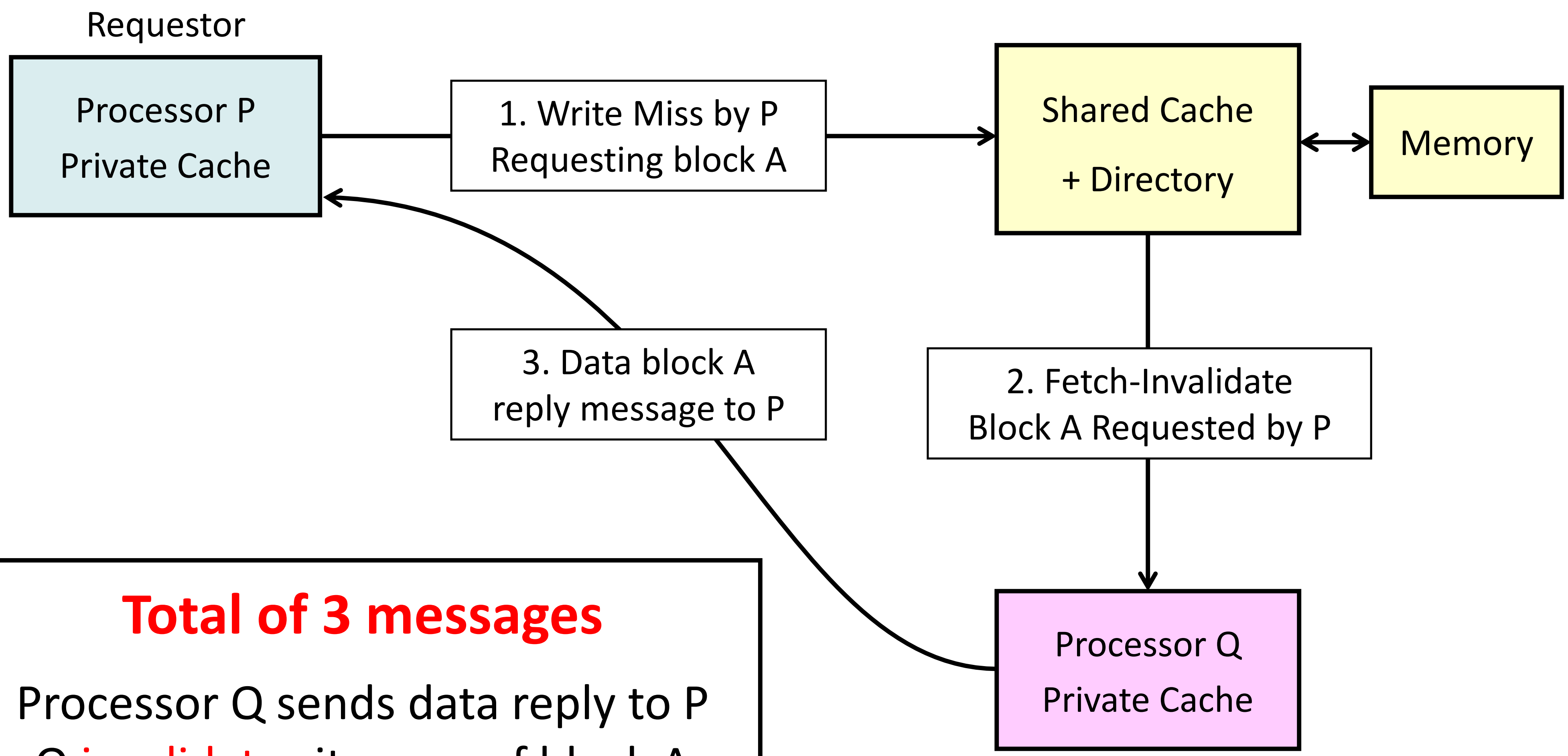
Block A is **shared** by P and Q

It is **owned** by shared cache

Write Miss Message by P to Directory

- Home Directory: block is **Modified**
  - Directory sends **Fetch-Invalidate** message to remote cache of **Q**
  - Remote cache of processor **Q** sends **data reply** message directly to **P**
  - Remote cache changes state of block to **invalid**
  - Local cache of **P** changes the state of received block to **modified**
  - Directory clears presence bit of **Q** and sets presence bit of **P**
- Home Directory: block is **Shared** or **Owned**
  - Directory sends **invalidate** messages to all sharers (presence bits)
  - Directory receives **acknowledge** message and clears presence bits
  - Directory sends **data reply** message to **P**, sets presence bit of **P**
  - Local cache of **P** and directory change state of the block to **modified**
- Home Directory: **Invalid** ➔ get block from memory

Write Miss to a Block in Modified State



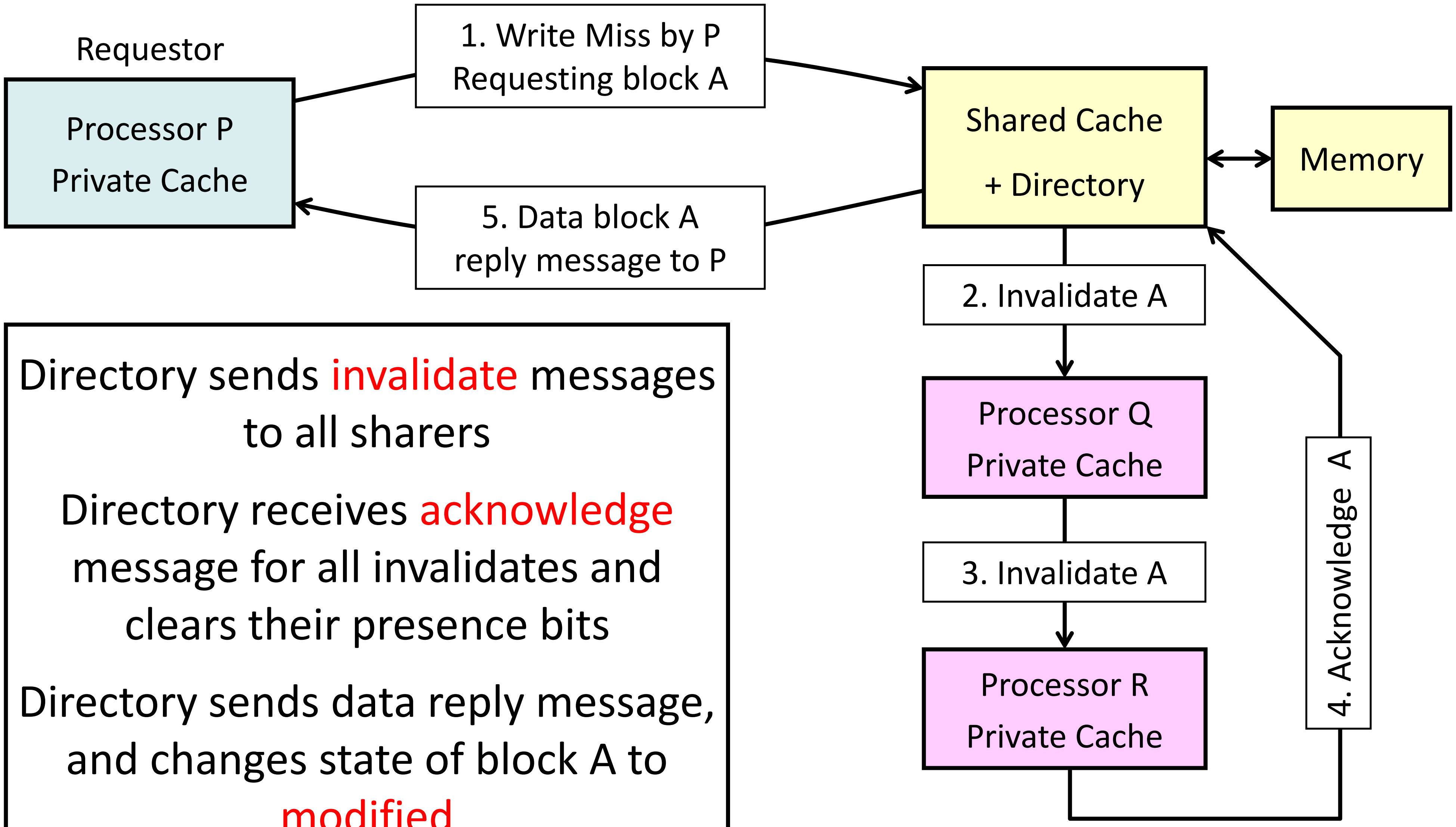
Total of 3 messages

Processor Q sends data reply to P

Q invalidates its copy of block A

Block A is then modified by P

Write Miss to a Block with Sharers

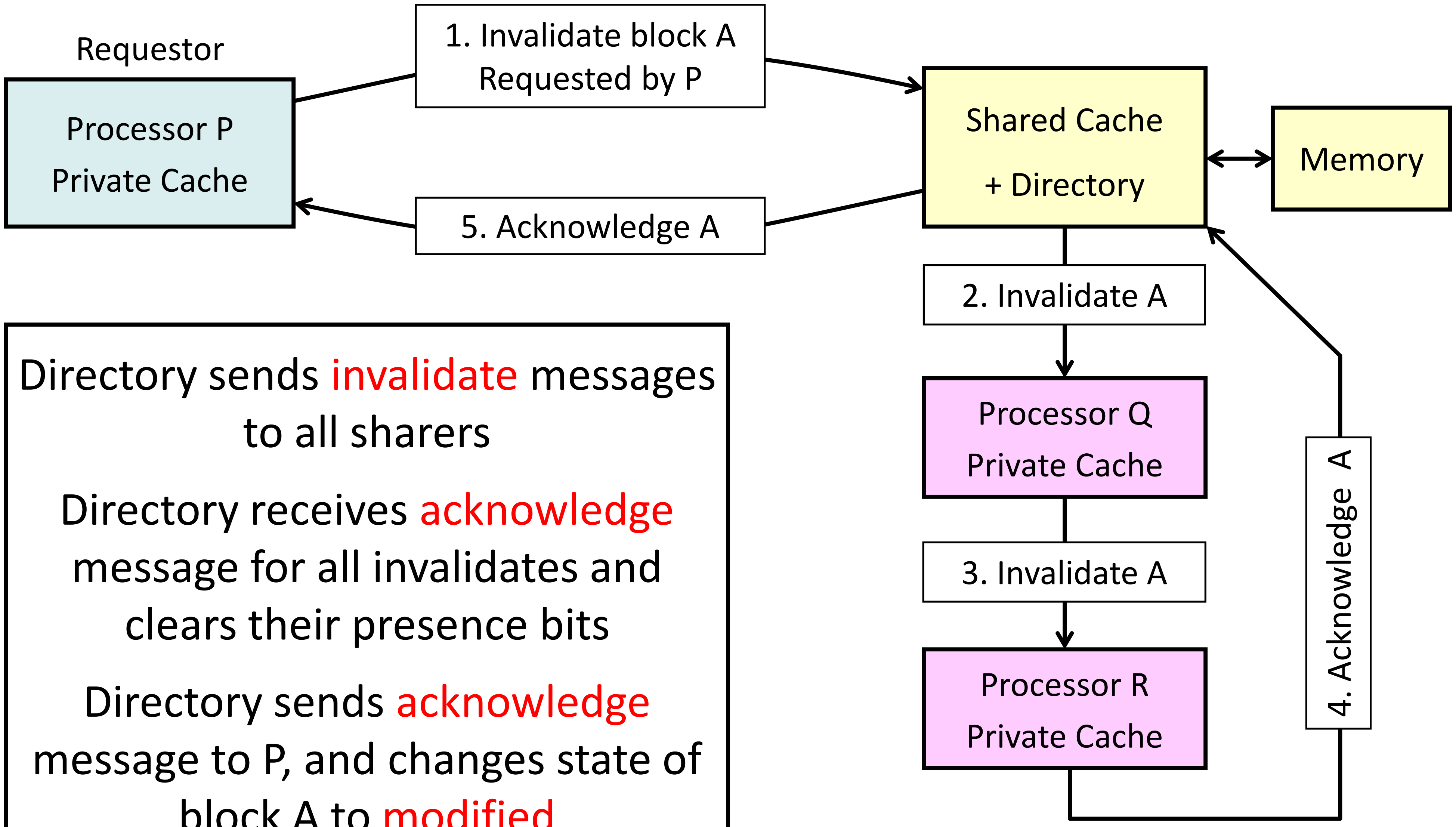


Directory sends invalidate messages to all sharers

Directory receives acknowledge message for all invalidates and clears their presence bits

Directory sends data reply message, and changes state of block A to modified

Invalidating a Block with Sharers



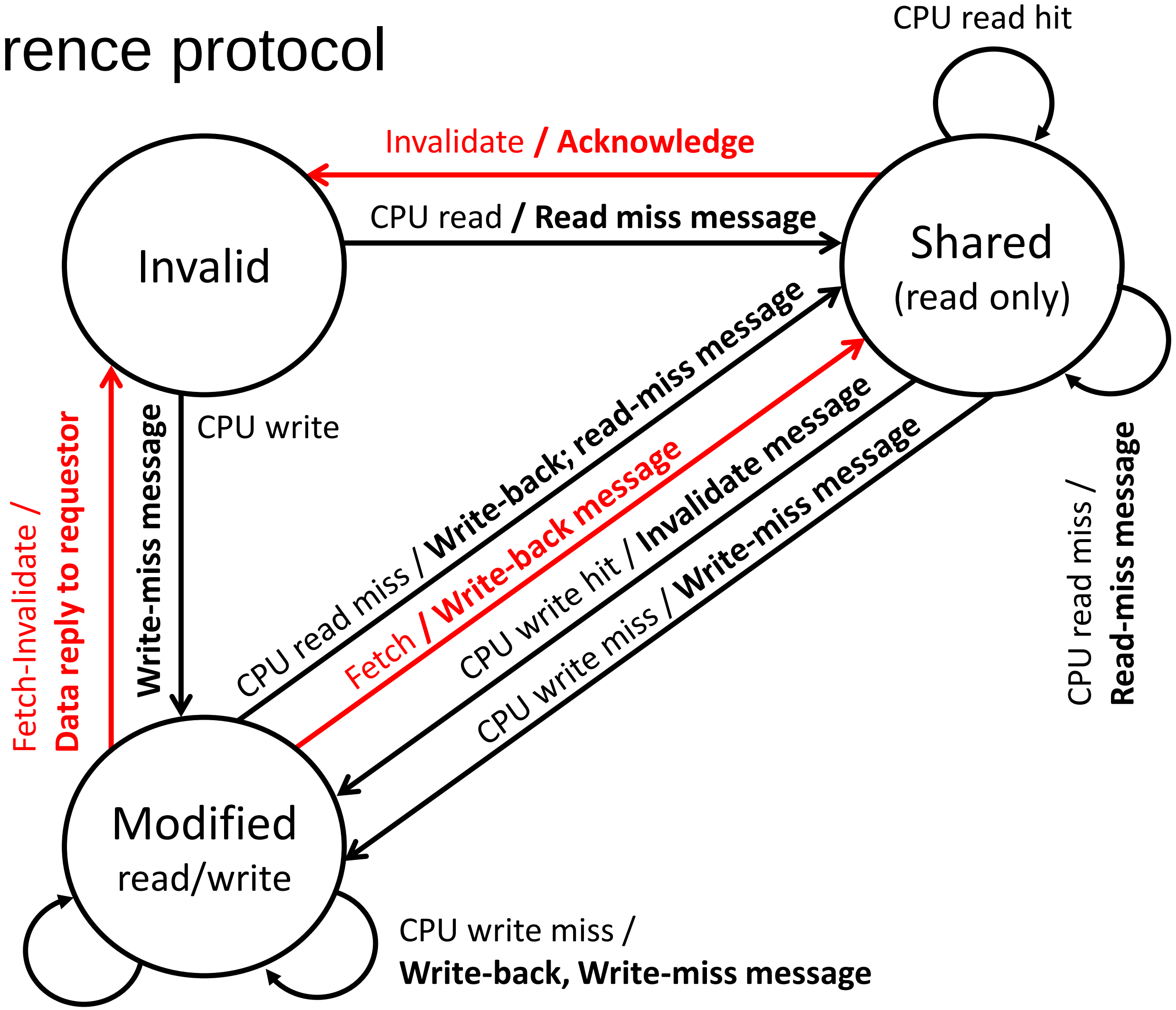
Directory sends invalidate messages to all sharers

Directory receives acknowledge message for all invalidates and clears their presence bits

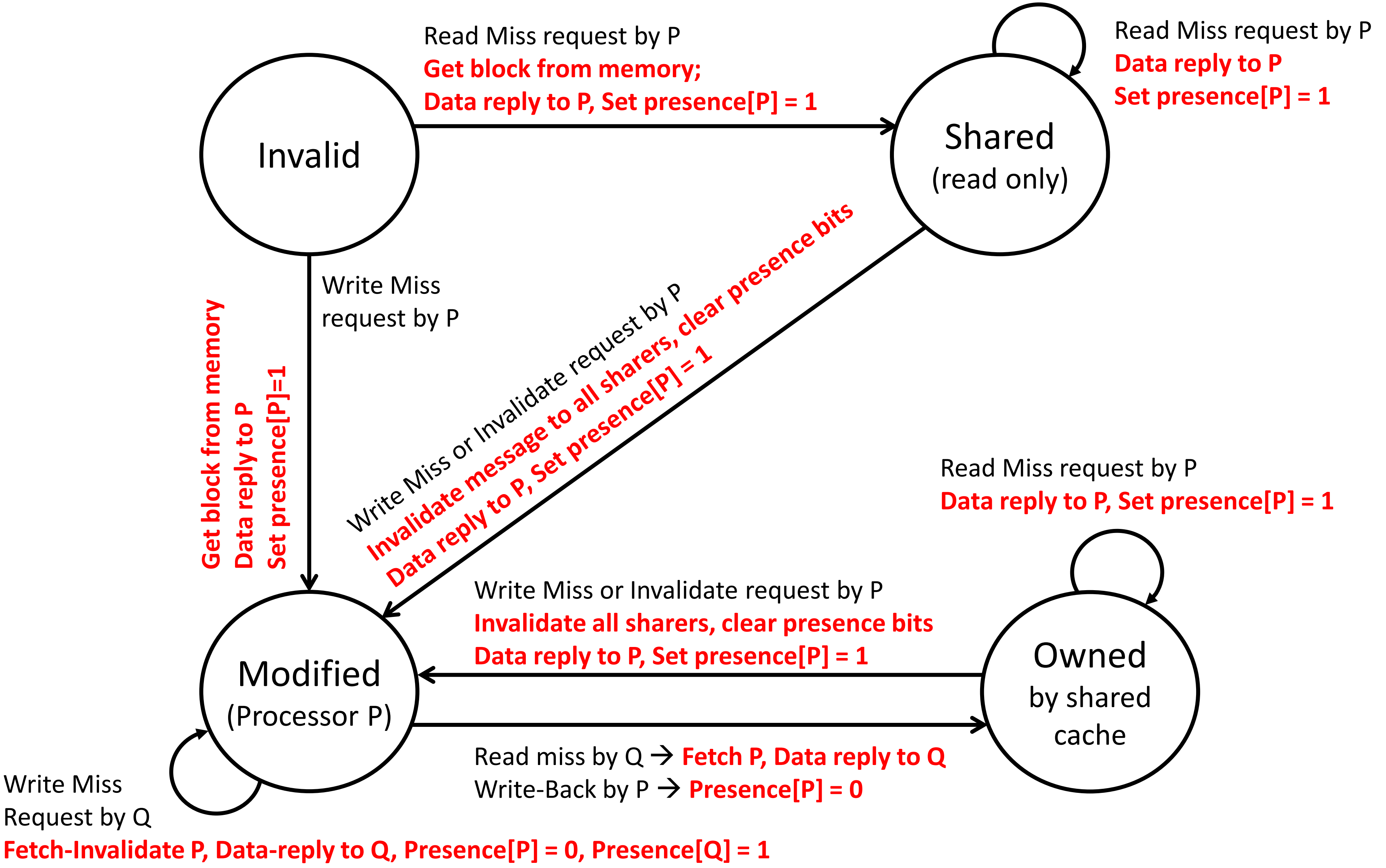
Directory sends acknowledge message to P, and changes state of block A to modified

MSI State Diagram for a Local Cache

- Three states for a cache block in a local (private) cache
- Similar to snooping coherence protocol
- Requests **by processor**
  - **Black arrows**
- Requests **by directory**
  - **Red arrows**



MOSI State Diagram for Directory



Next . . .

- Introduction to Multiprocessors
- Challenges of Parallel Programming
- Cache Coherence
- Directory Cache Coherence
- **Synchronization**

Synchronization

- Cache coherence allows parallel threads to **communicate**
- However, coherence **does not synchronize** parallel threads
- Synchronization is required to control the execution of threads
- Three types of synchronization are widely used:

1. **Lock** synchronization
2. **Barrier** synchronization

- Synchronization **ensures the correctness** of parallel execution
- However, synchronization can be a **performance bottleneck**
  - Reduces the speedup and performance of parallel threads

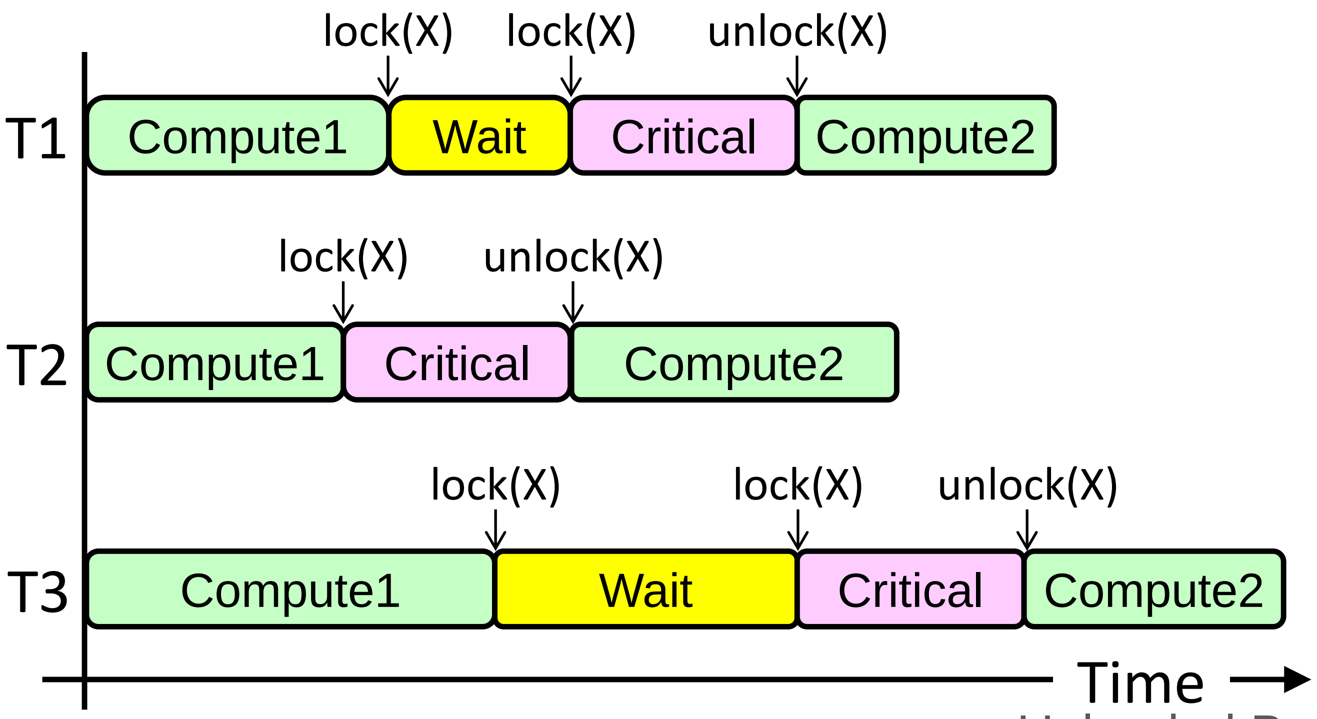
Lock Synchronization

- Protects a **critical section** accessed by parallel threads
- A critical section is a sequence of instructions that
  - **Read – Modify – Write** shared data in memory
  - Only **one thread** can be in the critical section at a time
- Two synchronization operations are defined:
  1. **Lock(X)**, just before entering critical section
  2. **Unlock(X)**, just before leaving critical section
- Only **one thread** is allowed to lock variable **X** at a time
- Other threads **must wait** until the variable **X** is unlocked
- Access to the critical section is **serialized**

Critical Section

- Critical sections with lock/unlock make threads wait
- Using one lock variable X to lock an array increases contention
- Using many lock variables (fine-grain locking) reduces contention
- Atomic (read-modify-write) instructions can help reducing locks

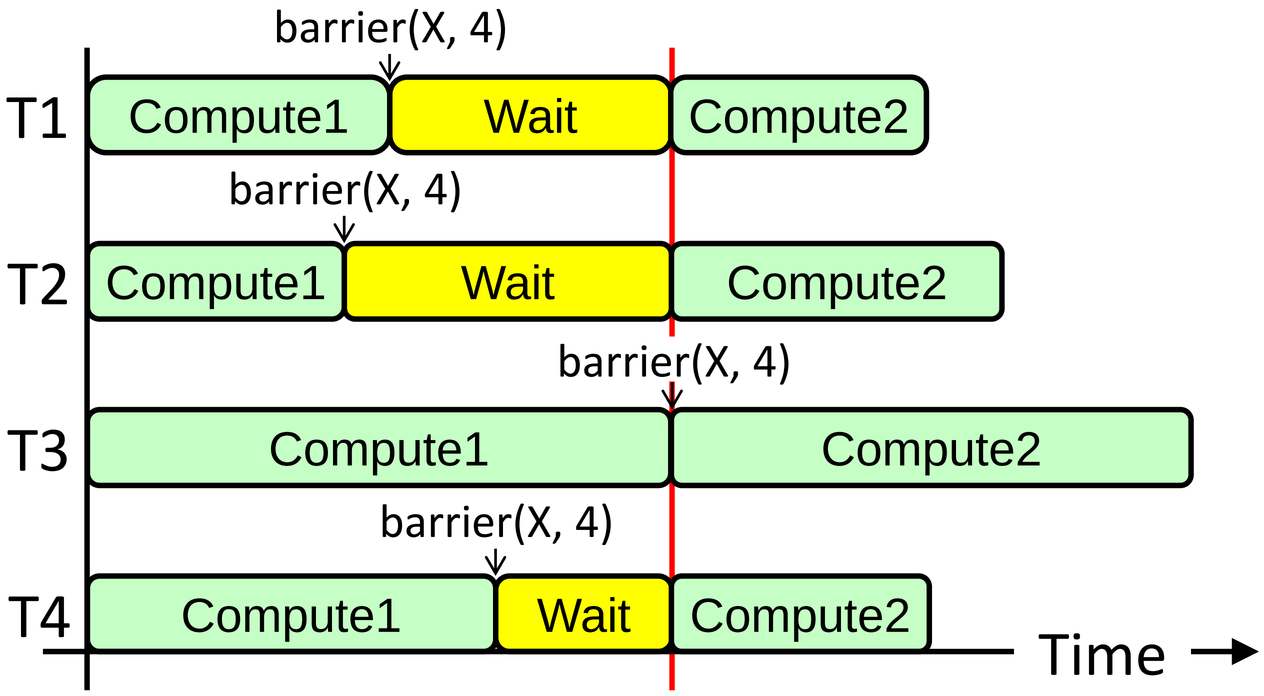
```
Each thread:
Compute1
lock(X)
   Critical
   Section
unlock(X)
Compute2
```



Barrier Synchronization

- ❖ Threads **must wait** until **All threads** reach the barrier
- ❖ **Last thread** arriving the barrier **releases all waiting** threads
- ❖ Total execution time depends on the **slowest thread**
- ❖ Threads must be **balanced** to avoid loosing performance

```
Each thread:
loop {
   Compute1
}
Barrier(X,4)
loop {
   Compute2
}
```



Concluding Remarks

- Goal: higher performance using multiple processors
- Difficulties
  - Developing parallel software
  - Devising appropriate architectures
- Many reasons for optimism
  - Changing software and application environment
  - Chip-level multiprocessors
    - Lower latency, higher bandwidth interconnect
- An ongoing challenge for computer architects