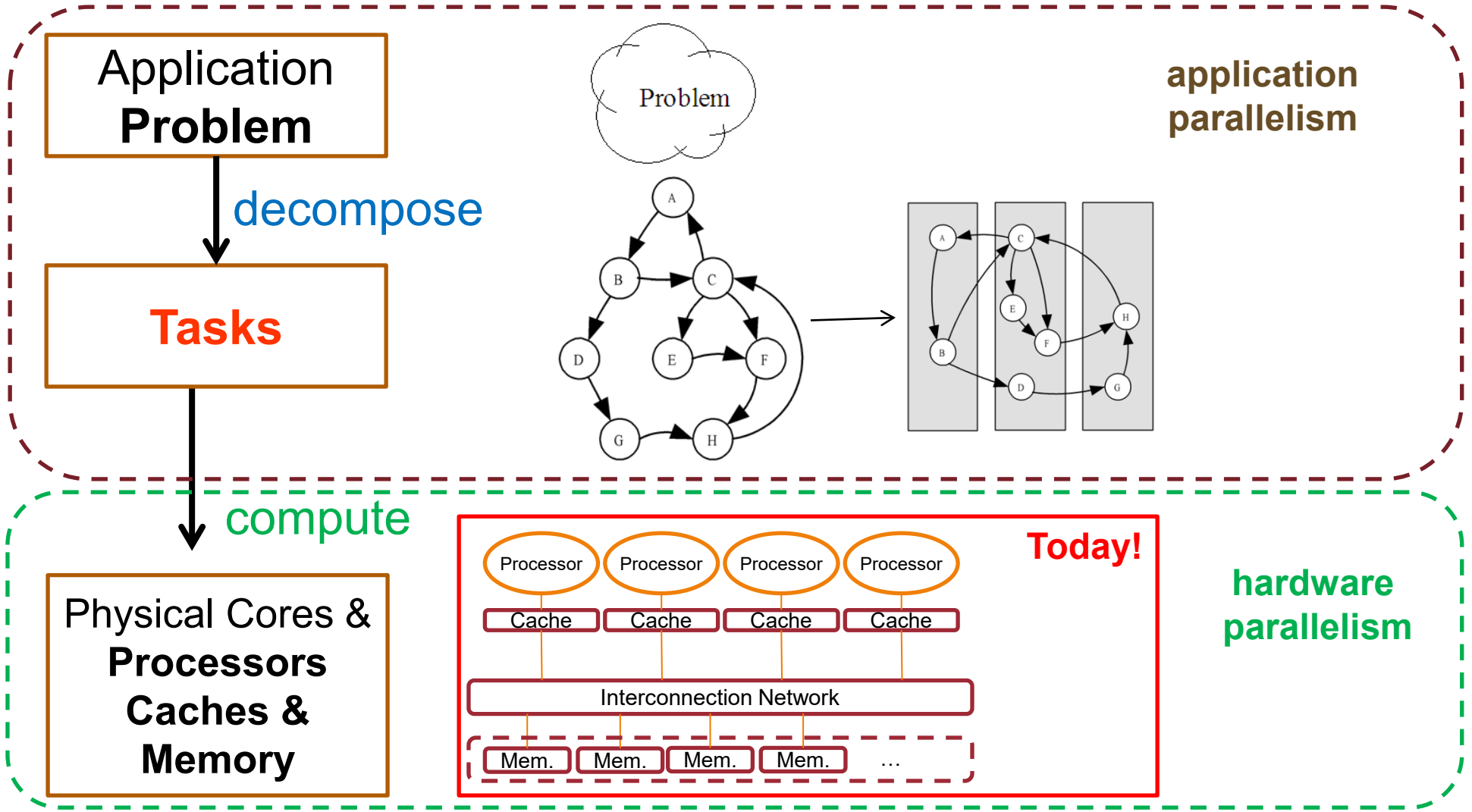
Cache Coherence
Memory Consistency

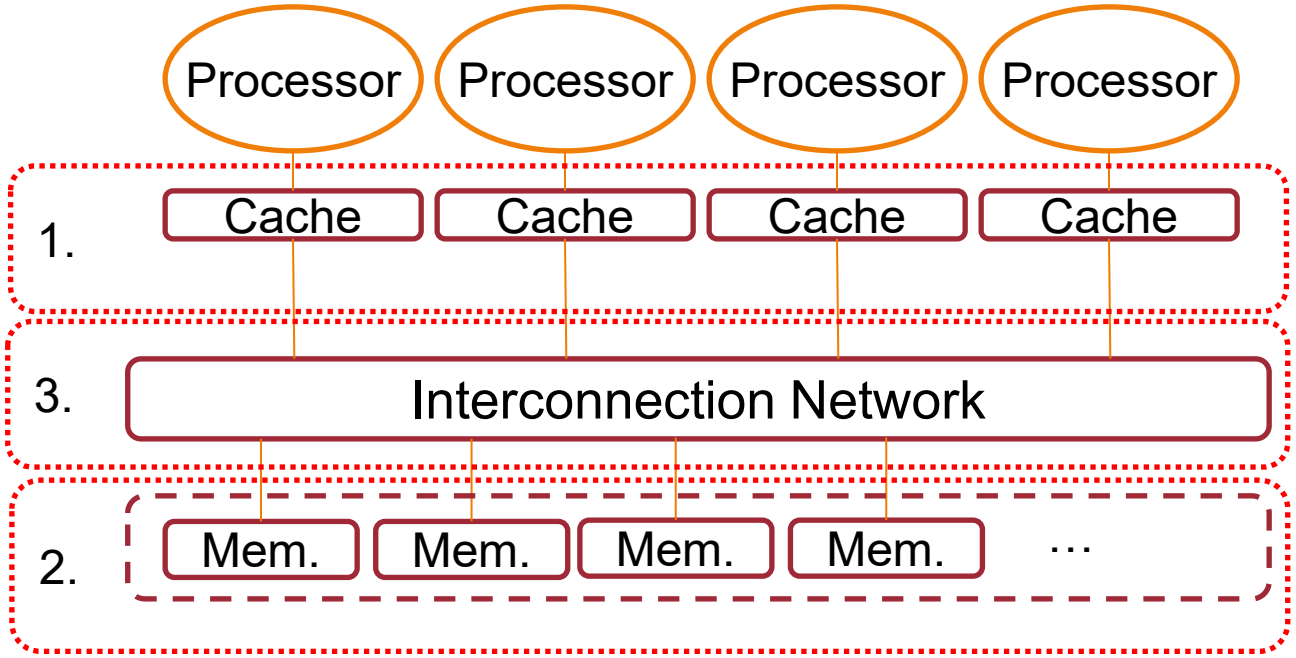
Lecture 07

Parallel Computing



Outline

1. Cache Coherence – L07
2. Memory Consistency – L07
3. Interconnection Networks
   - L10



In shared address space!

Shared Address Space Model

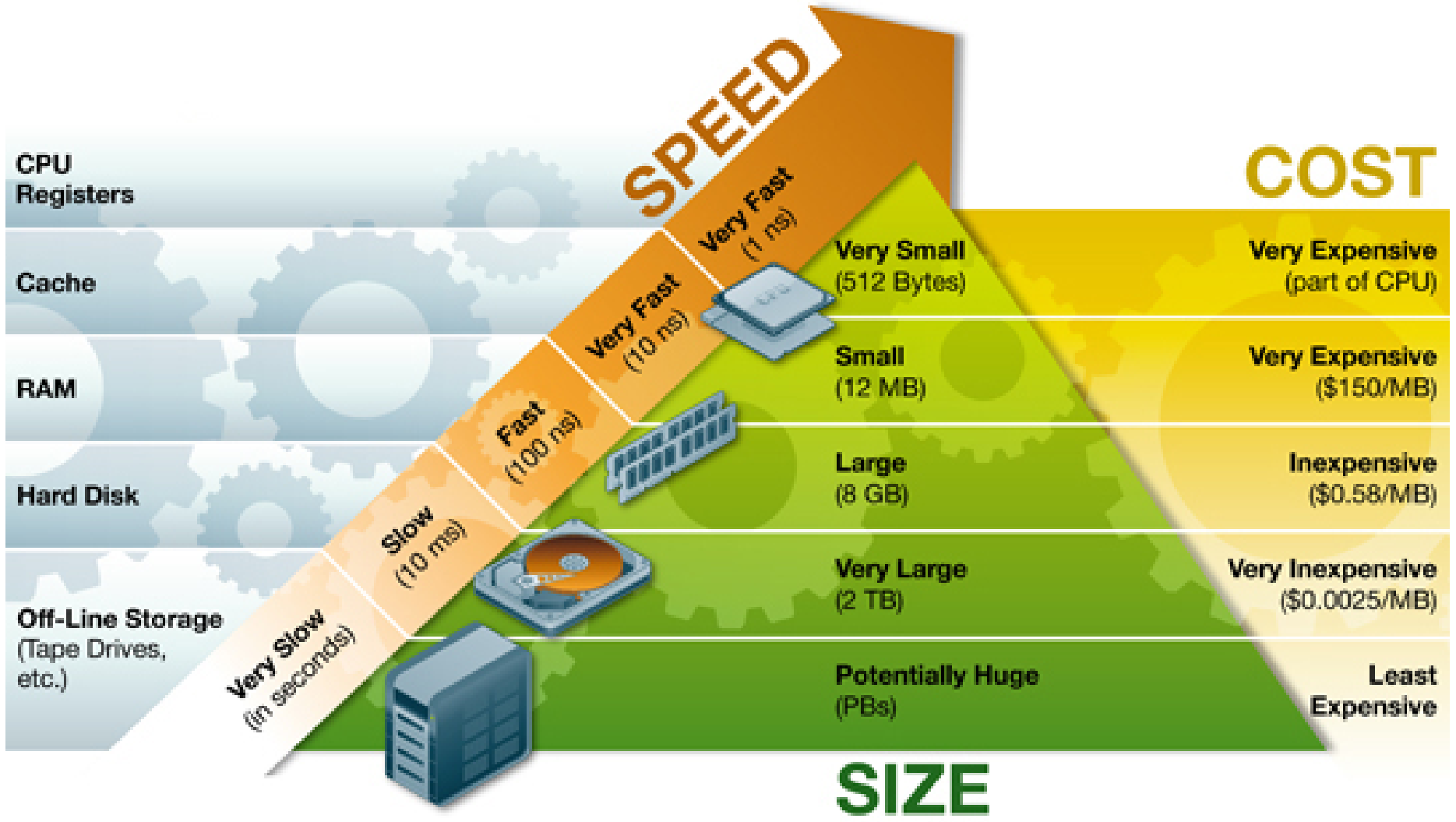
- Communication abstraction
  - Tasks communicate by reading/writing to shared variables
  - Ensure mutual exclusion via use of locks
  - Logical extension of uniprocessor programming
- Requires hardware support to implement efficiently
  - Any processor can load and store from any address
  - Even with NUMA, costly to scale
  - Matches **shared memory systems** – UMA, NUMA, etc.

There are only two hard things in Computer Science: cache invalidation and naming things.

-- Phil Karlton

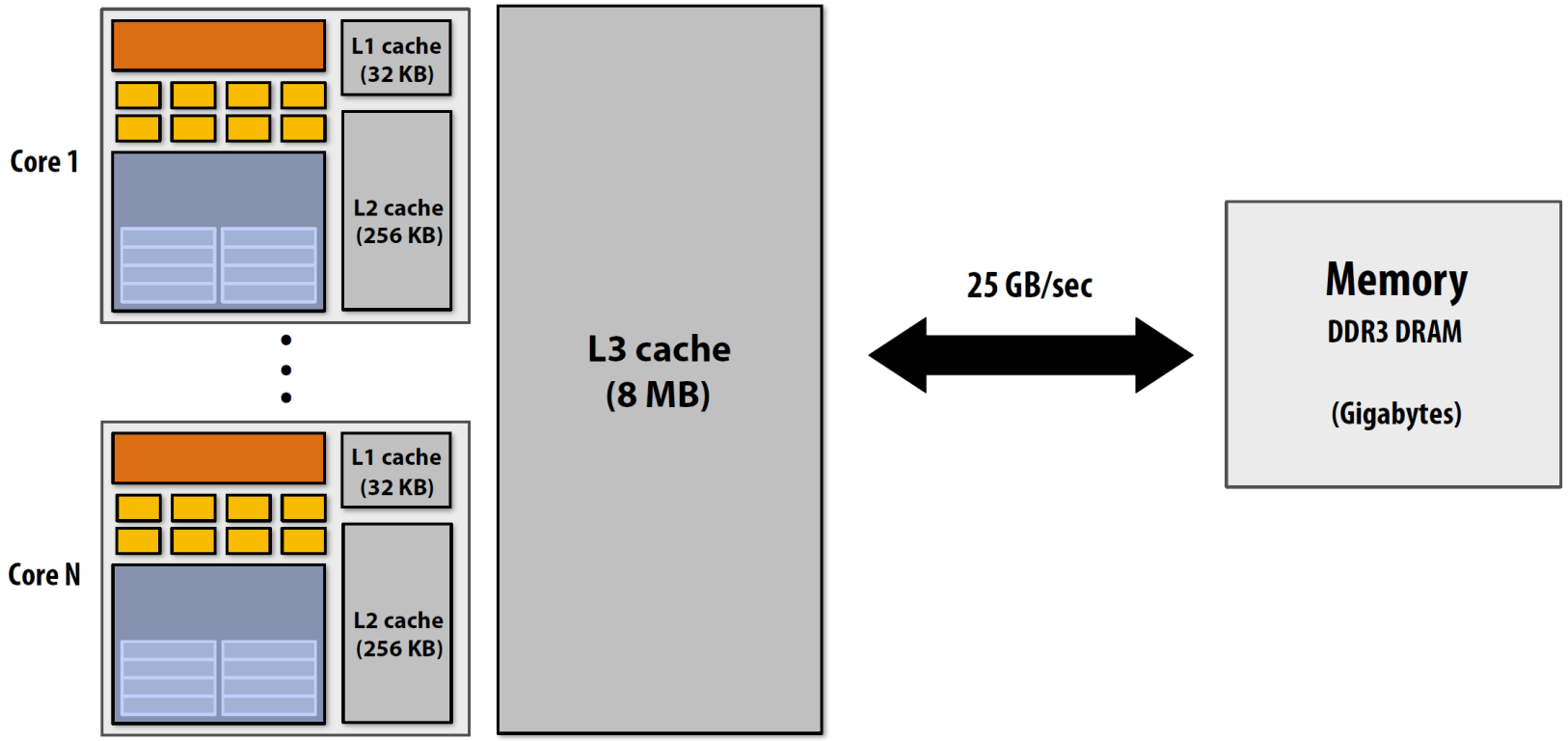
CACHE COHERENCE

The Memory Hierarchy



Recap: why do modern processors have cache?

- Processors run efficiently when data is resident in caches
  - Caches reduce memory access latency
  - Caches provide high bandwidth data transfer to CPU



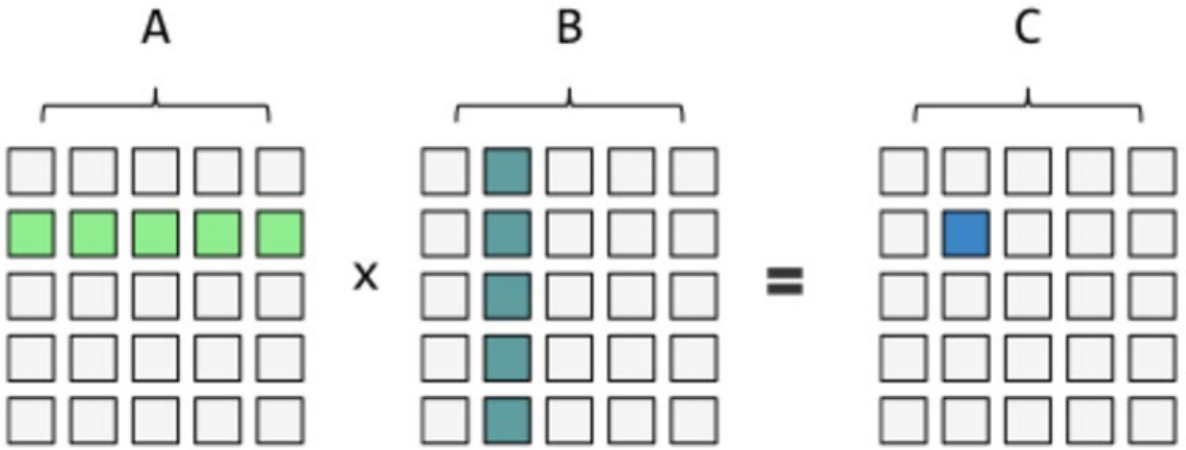
Cache Properties

- **Cache size:** larger cache increases access time (because of increased addressing complexity) but reduces cache misses

- **Block size:** data is transferred between main memory and cache in blocks of a fixed length
  - Larger blocks reduces the number of blocks but replacement lasts longer → block size should be small
  - But larger block increase the chance of spatial locality cache hit → block size should be large

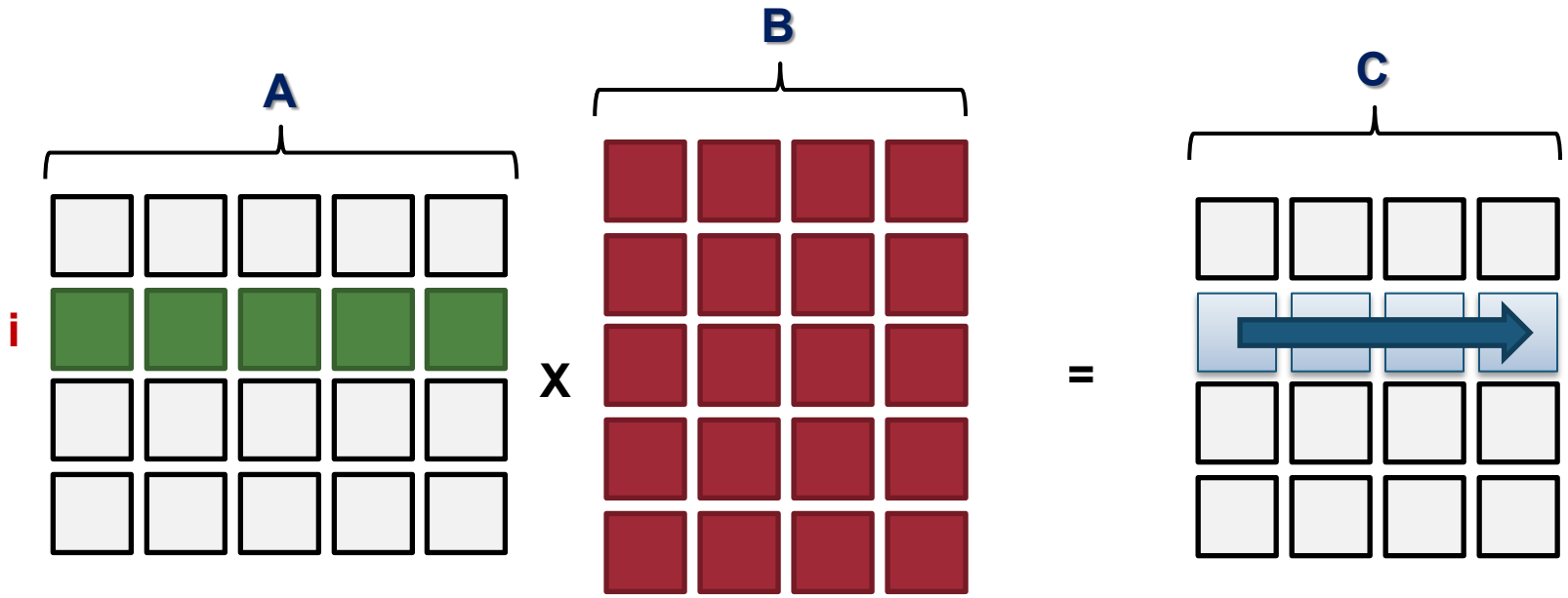
- Typical sizes for L1 cache blocks are 4 or 8 memory words

Case Study: Matrix Multiplication




```
for i ← 0 to n-1
    for j ← 0 to n-1
        c[i, j] ← 0
        for k ← 0 to n-1
            c[i, j] ← c[i, j] + a[i, k] x b[k, j]
```

One Iteration



- Read:
  - Row i^{th} of matrix **A**
  - Entire matrix **B**
- Write:
  - Row i^{th} of matrix **C**

What are the potential cache related performance problems??

Matrix Multiplication and Cache

- Matrix Row-Column Ordering
  - Row-major vs Column-major

- Size of matrix:
  - Motivating question: How large is the square matrix that can be stored entirely in a 256KB cache?

  - You can assume double-precision floating point numbers (i.e. 8 bytes).

Write Policy

- Write-through - write access is immediately transferred to main memory
  - Advantage: always get the newest value of a memory block
  - Disadvantages: slow down due to many memory accesses
    - Use a write buffer
- Write-back - write operation is performed only in the cache
  - Write is performed to the main memory when the cache block is replaced
  - Uses a dirty bit
  - Advantages: less write operations
  - Disadvantages: memory may contain invalid entries

Write-back Cache

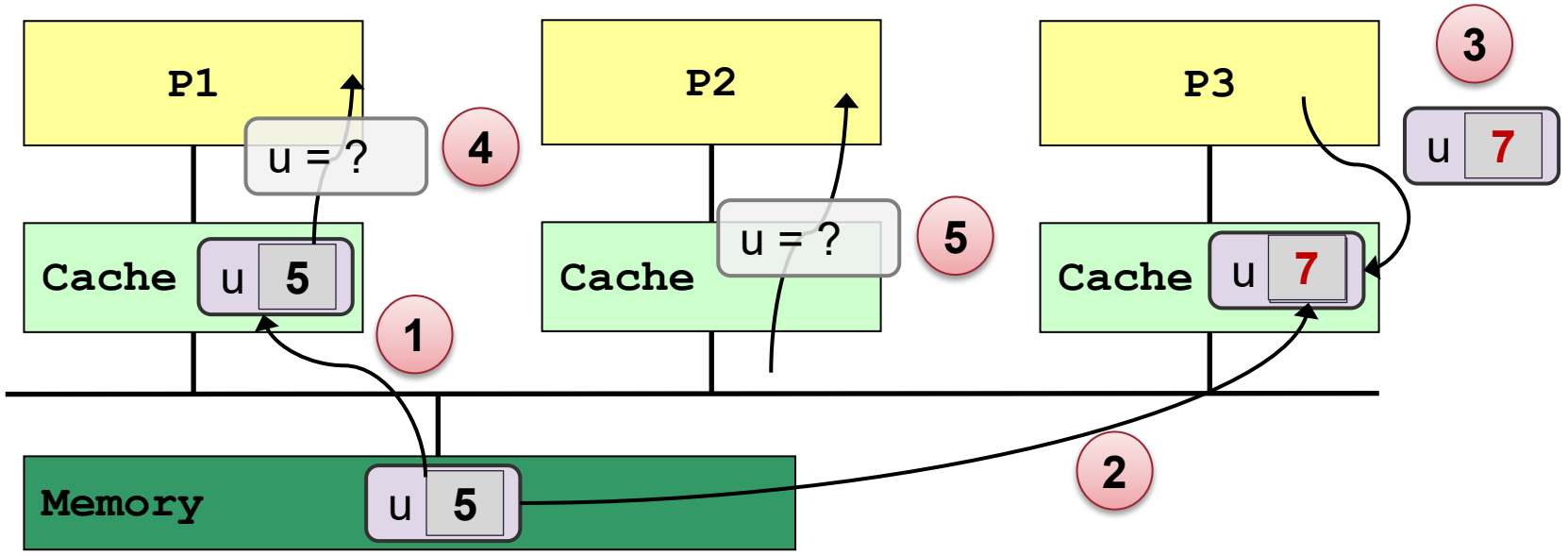
- Warm up: uniprocessor
- Example: processor executes `int x = 1;`

1. Processor performs write to address that "misses" in cache
2. Cache selects location to place line in cache, if there is a dirty line currently in this location, the dirty line is written out to memory
3. Cache loads line from memory ("allocates line in cache")
4. Whole cache line is fetched
5. Cache line is marked as dirty

Line state	Tag	Data (64 bytes)

Cache Coherence Problem

- Multiple copies of the same data exists on different caches
- Local update by processor → Other processors may still see the unchanged data



Memory Coherence for Shared Address Space

- Intuitively, reading value at an address should return the last value written at that address by any processor
- Uniprocessor: typically writes come from one source, the processor
- Multiprocessor: a single share address space leads to a memory coherence problem because there is
  - Global storage space (main memory)
  - Per-processor local storage (per-processor caches)

Memory Coherence: Definition

- Three properties of a *coherent* memory system
  - Can be used to evaluate cache coherence

- Property One:
  - Given the sequence
    1. **`P`** write to **`X`**
    2. No write to **`X`**
    3. **`P`** read from **`X`**
  - **`P`** should get the value written in (1)
  - Known as **Program Order** property

Memory Coherence: Definition (2)

- Property Two:
  - Given the sequence
    1. P_1 write to **X**
    2. No further write to **X**
    3. P_2 read from **X**
  - P_2 should read value written by P_1

  ➔ Writes become visible to other processors

  - Known as **Write Propagation** property

Memory Coherence: Definition (3)

- Property Three:
  - Given the sequence
    1. Write v_1 to **x** (by any processor)
    2. Write v_2 to **x** (by any processor)
  - Processors can **never** read **x** as v_2, then **later** as v_1

  ➔ All writes to a location (from the same or different processors) are seen in the same order by all processors

  - Known as **Write Serialization** property

Maintaining Memory Coherence

- Cache coherence can be maintained by:
  - Software based solution
    - OS + Compiler + Hardware aided solution
    - E.g. OS uses page-fault mechanism to propagate writes
  - Hardware based solution
    - Most common on multiprocessor system
    - Known as *cache coherence protocols*

- Major tasks for hardware CC protocols:
  - Track the sharing status of a cache line
  - Handle the update to a shared cache line
    - i.e. maintain the coherence

Tracking Cache Line Sharing Status

- Two major categories:
  - **Snooping Based**
    - No centralized directory
    - Each cache keeps track of the sharing status
    - Cache *monitors* or *snoop* on the bus
      - to update the status of cache line
      - takes appropriate action
    - Most common protocol used in architectures with a bus
  - **Directory Based**
    - Sharing status is kept in a centralized location
    - Commonly used with NUMA architectures

Snooping Cache Coherence

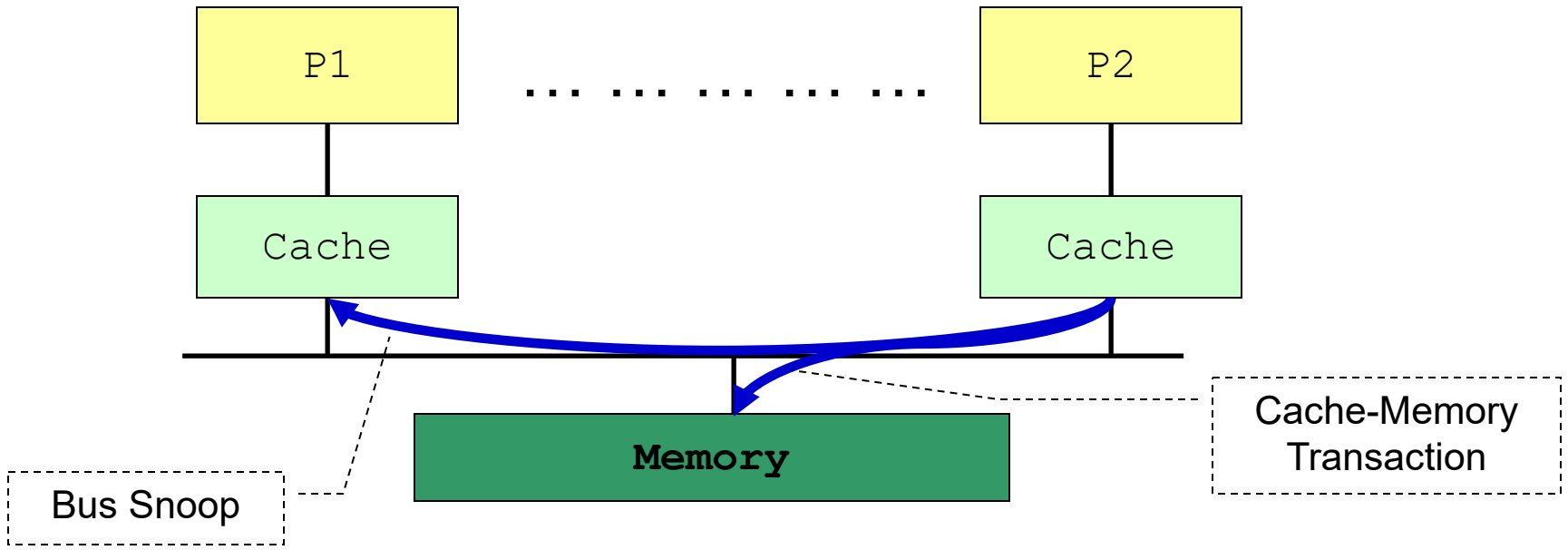
- Bus-based cache coherence
  - All the processors on the bus can observe every bus transactions ➔ (Write Propagation)
  - Bus transactions are visible to the processors in the same order ➔ (Write Serialization)

- The cache controllers:
  - "snoop" on the bus and monitor the transactions
  - Takes action for relevant bus transaction
    - i.e., if it has the memory block in cache

- Granularity of cache coherence is cache block

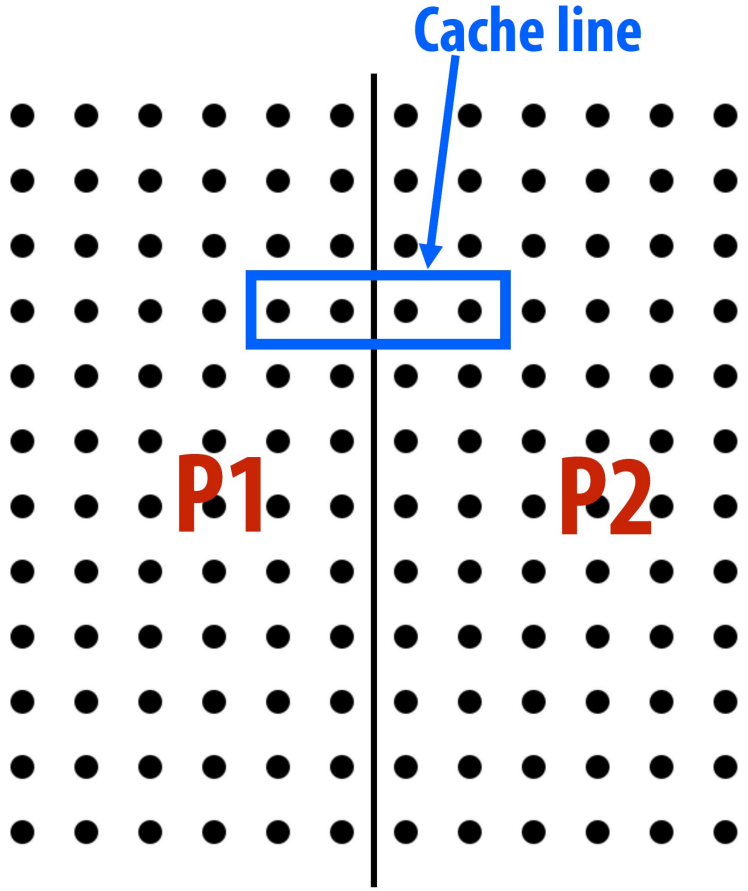
Snooping Based Protocol : Illustration

- Note:
  - Bus is a shared medium
  - Transaction on the bus can be monitored by all caches connected



Cache Coherence Implications

- Overhead in shared address space
  - CC appears as increased memory latency in multiprocessor
  - CC lowers the hit rate in cache
- False sharing
  - 2 processors write to different addresses
  - The addresses map to the same cache line



Read, Write, Write, Read, Ok?

MEMORY CONSISTENCY MODELS

Memory Consistency: Definition

- **Coherence** ensures that each processor has consistent view of memory through its local cache
  - All writes to **SAME memory location** (address) should be seen by all processors in the same order
  - Writes to an address by one processor will eventually be observed by other processors – but does not specify **when**

- Memory **consistency** ensures
  - Constraints on the order in which memory operations can appear to execute – **when** the operations are seen by other processors?
  - For **DIFFERENT memory locations**

- The model is then used:
  - By programmers to reason about correctness and program behavior
  - By system designers to decide the reordering possible by hardware and compiler

Warm up: Uniprocessor

- In uniprocessor, we assume:
  - Memory operations are performed in program order
  - Memory operations are atomic

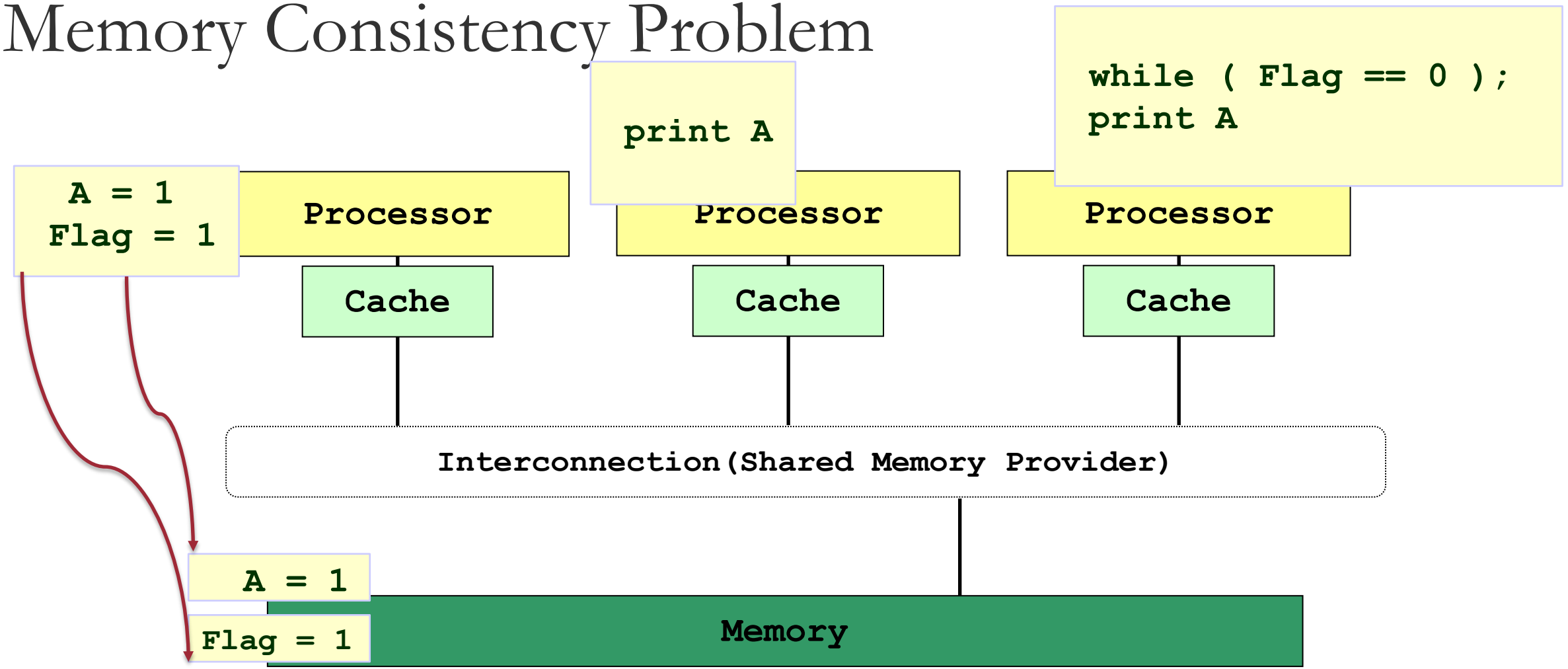
- In actual execution:
  - Read may be out of order w.r.t. write (for different memory locations)
  - Write may not be reflected in main memory

Memory Operations on Multiprocessors

- Four types of memory operation orderings
  - W→R: write to X must complete before subsequent read from Y *
  - R→R: read from X must complete before subsequent read from Y
  - R→W: read to X must complete before subsequent write to Y
  - W→W: write to X must complete before subsequent write to Y
- Reordered in an unintuitive way to hide write latencies
  - Application programmers rarely see this behavior
  - Systems (OS and compiler) developers see it all the time

*must complete ~ the results are visible

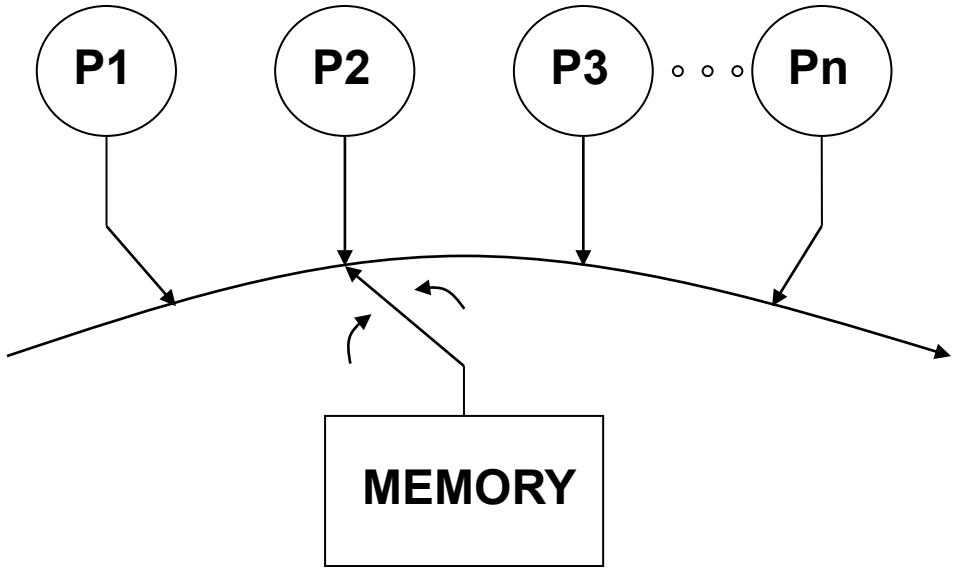
Memory Consistency Problem



Sequential Consistency Model (SC) – Lamport 1976

- A **sequentially consistent** memory system:
  - Every processor issues its memory operations in program order
    - **Global result of all memory accesses of all processors appears to all processors in the same sequential order** irrespective of the arbitrary interleaving of the memory accesses by different processors
    - Effect of each memory operation must be visible to all processors before the next memory operation on any processor
- Extension of uniprocessor memory model:
  - Intuitive, but can result in loss of performance

Sequential Consistency: Illustration



- **As if**:
  - One memory space
  - Only one memory operation at any point in time

Example: Sequential Consistent

processor	P_1	P_2	P_3
program	(1) $x_1 = 1$;	(3) $x_2 = 1$;	(5) $x_3 = 1$;
	(2) print x_2, x_3;	(4) print x_1, x_3;	(6) print x_1, x_2;

- **Possible outputs**
  - (1)-(3)-(5)-**(2)-(4)-(6)**
    → **111111**
  - (1)-**(2)**-(3)-(4)-(5)-(6)
    → **001011**
- Memory accesses from different processors are arbitrarily interleaved!

- **Impossible outputs**
  - **011001**

Motivation for Relaxed Consistency

- Hide latencies!
  - To gain performance
  - Specifically, hiding memory latency: overlap memory access operations with other operations when they are independent
- How?
  - Overlap memory access operations with other operations when they are independent

Relaxed Consistency

- Relaxed Consistency – relax the ordering of memory operations if data dependencies allow
  - Dependencies: if two operations access the **same memory location**
    - R → W: anti-dependence (WAR)
    - W → W: output dependence (WAW)
    - W → R: flow dependence (RAW)
    - Must be preserved!
- Relaxed memory consistency models allow certain orderings to be violated

Relaxed Consistency: Idea

Processor	P_1	P_2	P_3
Program	(1) $x_1 = 1$; (2) print (x_2)	(3) $x_2 = 1$; (4) print (x_1)	(5) $x_3 = 1$; (6) print (x_3)

- SC:
  - (1) → (2) and (3)→(4) and (5) →(6)
  - e.g. (1)-(2)-(5)-(6)-(3)-(4)
  - When there is no data dependency!
- Example of **RC**:
  - relax W → R
- Possible with RC:
  - (3)-(4) or (4)-(3)
  - (1)-(2) or (2)-(1)
  - (5)-(6)
  - Not: (6)-(5)

Relaxed Memory Consistency Models

- Memory models resulted from relaxation of SC's requirements
- Program order relaxation:
  - Write → Read
  - Write → Write
  - Read → Read or Write
- All models provide overriding mechanism to allow a programmer to intervene

Write-to-Read Program Order

- Key Idea:
  - Allow a read on processor **P** to be reordered w.r.t. to the previous write of the same processor
    - Hide the write latency
  - Different timing of the return of the read defines different models
  - **Data dependencies must be preserved!**

TSO and PC:

~~W -> R~~

R -> R

R -> W

W -> W

- Example Models:
  - **T**otal **S**tore **O**rdering (**TSO**) :
    - Return the value written by P earlier without waiting for it to be serialized
  - **P**rocessor **C**onsistency (**PC**) :
    - Return the value of any write (even from another processor) before the write is propagated or serialized

Example 1

P1

```
A = 1
Flag = 1
```

P2

```
while ( Flag == 0 );
print A
```

SC:
W -> R
R -> R
R -> W
W -> W

- A = Flag = 0 initially
- Can A = 0 be printed under the models?
  - SC: A = 0 ✗
  - TSO/PC: A = 0 ✗

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

Example 2

P1

```
A = 1
B = 1
```

P2

```
print B
print A
```

SC:
W -> R
R -> R
R -> W
W -> W

- A = B = 0 initially
- Can A = 0; B = 1 be printed under the models?
  - SC: A = 0; B = 1
  - TSO/PC: A = 0; B = 1

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

Example 3

P1	P2	P3
`A = 1`	`while ( A == 0 );` `B = 1`	`while ( B == 0 );` `print A`

SC:
W -> R
R -> R
R -> W
W -> W

- A = B = 0 initially
- Can A = 0 be printed under the models?
  - SC: A = 0
  - TSO: A = 0
  - PC: A = 0

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

Example 4

P1

```
A = 1
print B
```

P2

```
B = 1
print A
```

SC:
W -> R
R -> R
R -> W
W -> W

- A = B = 0 initially
- Can A = 0; B = 0 be printed under the models?
  - SC: A = 0 ; B = 0
  - TSO: A = 0 ; B = 0
  - PC: A = 0 ; B = 0

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

Write-to-Write Program Order

- Key Idea:
  - Writes can bypass earlier writes (to different locations) in write buffer
  - Allow write miss to overlap and hide latency

- Example Model:
  - **P**artial **S**tore **O**rdering (**PSO**)
    - Relax W → R order
      - Similar to TSO
    - Relax W → W order

PSO:
~~W -> R~~
R -> R
R -> W
~~W -> W~~

Example 5

P1

```
A = 1
Flag = 1
```

P2

```
while ( Flag == 0 );
print A
```

- A = Flag = 0 initially
- Can A = 0 be printed under the models?
  - SC: A = 0
  - TSO/PC: A = 0
  - PSO: A = 0

SC:
W -> R
R -> R
R -> W
W -> W

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

PSO:
~~W -> R~~
R -> R
R -> W
~~W -> W~~

Example 6

P1

```
A = 1
B = 1
```

P2

```
print B
print A
```

- A = B = 0 initially
- Can A = 0; B = 1 be printed under the models?
  - SC: A = 0; B = 1
  - TSO/PC: A = 0; B = 1
  - PSO: A = 0; B = 1

SC:
W -> R
R -> R
R -> W
W -> W

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

PSO:
~~W -> R~~
R -> R
R -> W
~~W -> W~~

Example 7

P1	P2	P3
`A = 1`	`while ( A == 0 );` `B = 1`	`while ( B == 0 );` `print A`

- A = B = 0 initially
- Can A = 0 be printed under the models?
  - SC: A = 0
  - TSO: A = 0
  - PC: A = 0
  - PSO: A = 0

SC:
W -> R
R -> R
R -> W
W -> W

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

PSO:
~~W -> R~~
R -> R
R -> W
~~W -> W~~

Example 8

P1

```
A = 1
print B
```

P2

```
B = 1
print A
```

- A = B = 0 initially
- Can A = 0; B = 0 be printed under the models?
  - SC: A = 0 ; B = 0
  - TSO: A = 0 ; B = 0
  - PC: A = 0 ; B = 0
  - PSO: A = 0 ; B = 0

SC:
W -> R
R -> R
R -> W
W -> W

TSO and PC:
~~W -> R~~
R -> R
R -> W
W -> W

PSO:
~~W -> R~~
R -> R
R -> W
~~W -> W~~

For Your Exploration

- Weak ordering models
  - No completion order of the memory operations is guaranteed
  - i.e. relax R → R, R → W -> out-of-order execution
  - Provide additional synchronization operations
    - Lock/unlock
    - Memory fence (barrier)

Summary

- Cache coherence
  - Ensures that each processor has consistent view of memory through its local cache
- Memory consistency
  - Order of memory accesses → opportunity for reducing program execution time

Midterm

- 5 Oct, 2pm at the lecture
- 15% of the grade
- Covers:
  - Lectures 1 - 6
  - Tutorials 1 - 2
  - Labs 1 – 3
- Open book