Lecture 13:

Memory Consistency

+ course-so-far review (if time)

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2017

Tunes

Hello

Adele

(25)

Hello, it's me
I was wondering if after all these lectures you'd like to meet
To go over everything
They say that studying's supposed to help ya
But I ain't done much studying

Hello, can you hear me
I'm in the library dreaming about what the exam is going to be
I hope its shorter, and covers ISPC
I've forgotten how it felt before I dealt with so much concurrency

"Adele, no one is going to understand your song about 418." - Producers at XL Recordings
"Okay, fine… I'll sing about nostalgia then." - Adele

Today: what you should know

- Understand the motivation for relaxed consistency models

- Understand the implications of relaxing W → R ordering

Today: who should care

- Anyone who:
  - Wants to implement a synchronization library
  - Will ever work a job in kernel (or driver) development
  - Seeks to implement lock-free data structures *
  - Does any of the above on ARM processors **

* Topic of a later lecture

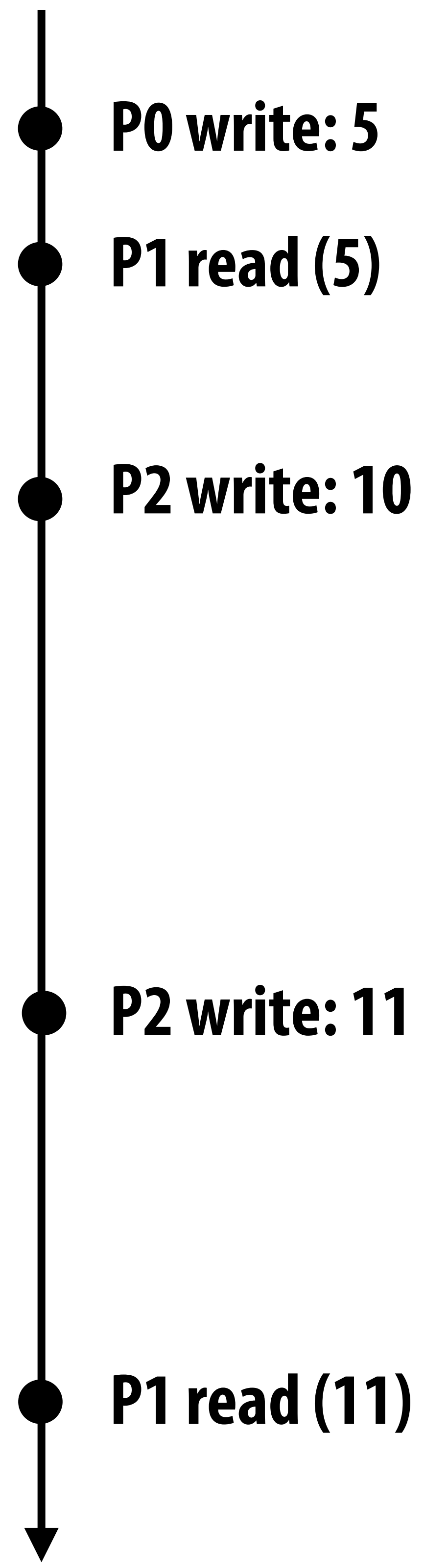
** For reasons to be described later

Memory coherence vs. memory consistency

- **Memory coherence** defines requirements for the observed behavior of reads and writes to the same memory location
  - All processors must agree on the order of reads/writes to X
  - In other words: it is possible to put all operations involving X on a timeline such that the observations of all processors are consistent with that timeline

- **Memory consistency** defines the behavior of reads and writes to different locations (as observed by other processors)
  - Coherence only guarantees that writes to address X will eventually propagate to other processors
  - Consistency deals with when writes to X propagate to other processors, relative to reads and writes to other addresses

Observed chronology of operations on address X



Coherence vs. consistency

(said again, perhaps more intuitive this time)

- **The goal of cache coherence is to ensure that the memory system in a parallel computer behaves as if the caches were not there**
  - Just like how the memory system in a uni-processor system behaves as if the cache was not there
- **A system without caches would have no need for cache coherence**
- **Memory consistency defines the allowed behavior of loads and stores to different addresses in a parallel system**
  - The allowed behavior of memory should be specified whether or not caches are present (and that's what a memory consistency model does)

Memory operation ordering

- A program defines a sequence of loads and stores (this is the "program order" of the loads and stores)

- Four types of memory operation orderings
  - W→R: write to X must commit before subsequent read from Y *
  - R→R: read from X must commit before subsequent read from Y
  - R→W: read to X must commit before subsequent write to Y
  - W→W: write to X must commit before subsequent write to Y

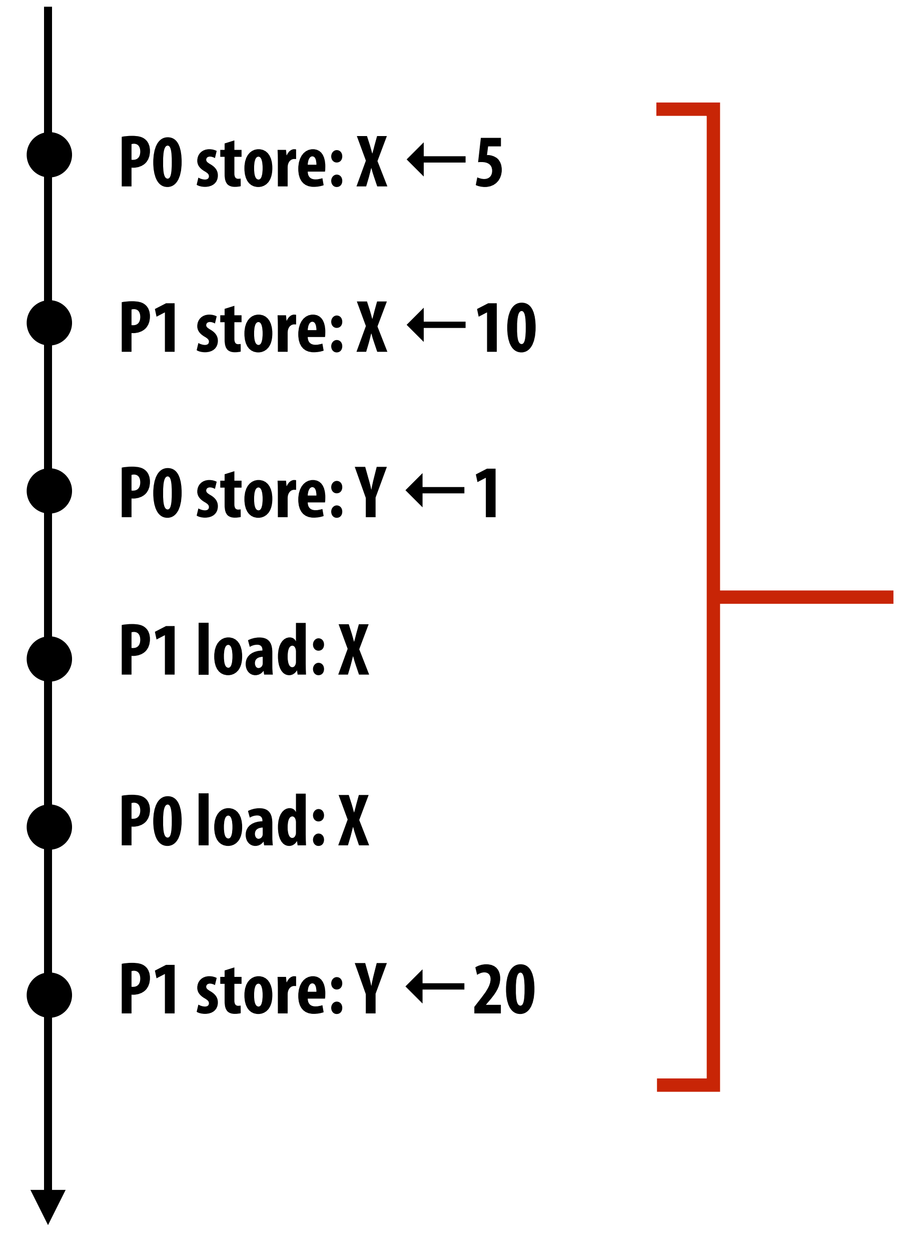
- A sequentially consistent memory system maintains all four memory operation orderings

* To clarify: "write must commit before subsequent read" means:
When a write comes before a read in program order, the write must commit (its results are visible) by the time the read occurs.

Sequential consistency

- A parallel system is sequentially consistent if the result of any parallel execution is the same as if all the memory operations were executed in some sequential order, and the memory operations of any one processor are executed in program order.

There is a chronology of all memory operations
that is consistent with observed values

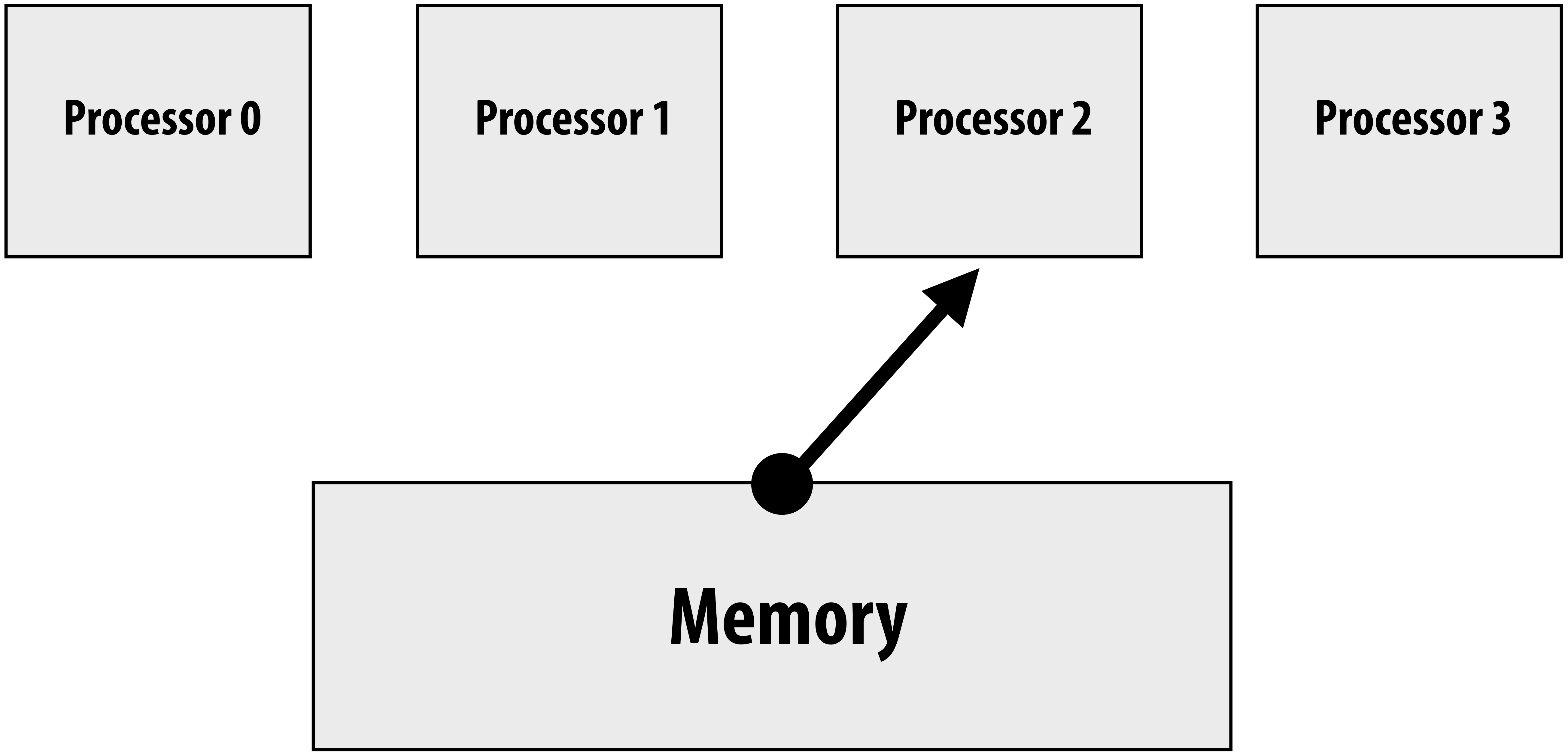


Note, now timeline lists operations to addresses X and Y

[Adve and Gharachorloo 95]

Sequential consistency (switch metaphor)

- **All processors issue loads and stores in program order**
- **Memory chooses a processor, performs a memory operation to completion, then chooses another processor, . . .**



Quick example

Thread 1 (on P1)

```
A = 1;
if (B == 0)
  print “Hello”;
```

Thread 2 (on P2)

```
B = 1;
if (A == 0)
  print “World”;
```

Assume A and B are initialized to 0.

Question: Imagine threads 1 and 2 are being run simultaneously on a two processor system. What will get printed?

Answer: assuming writes propagate immediately (e.g., P1 won’t continue to ‘if’ statement until P2 observes the write to A), then code will either print “hello” or “world”, but not both.

Consider various interleavings…

Thread 1 (on P1)

```
A = 1;
if (B == 0)
  print “Hello”;
```

Thread 2 (on P2)

```
B = 1;
if (A == 0)
  print “World”;
```

```
P1: Write 1 to A
P1: Read B  (it’s 0, print “Hello”)
P2: Write 1 to B
P2: Read A  (it’s 1)
```

```
P1: Write 1 to A
P2: Write 1 to B
P1: Read B  (it’s 1)
P2: Read A  (it’s 1)
```

```
P2: Write 1 to B
P2: Read A  (it’s 0, print “World”)
P1: Write 1 to A
P1: Read B  (it’s 1)
```

```
P1: Write 1 to A
P2: Write 1 to B
P2: Read A  (it’s 1)
P1: Read B  (it’s 1)
```

Answer: assuming writes propagate immediately (e.g., P1 won’t continue to ‘if’ statement until P2 observes the write to A), then code will either print “hello” or “world”, but not both.

Relaxing memory operation ordering

- A <u>sequentially consistent</u> memory system maintains all four memory operation orderings (W→R, R→R, R→W, W→W)

- Relaxed memory consistency models allow certain orderings to be violated

Back to the quick example

Thread 1 (on P1)

```
A = 1;
if (B == 0)
  print “Hello”;
```

From the processor’s perspective, these are independent instructions in each thread.

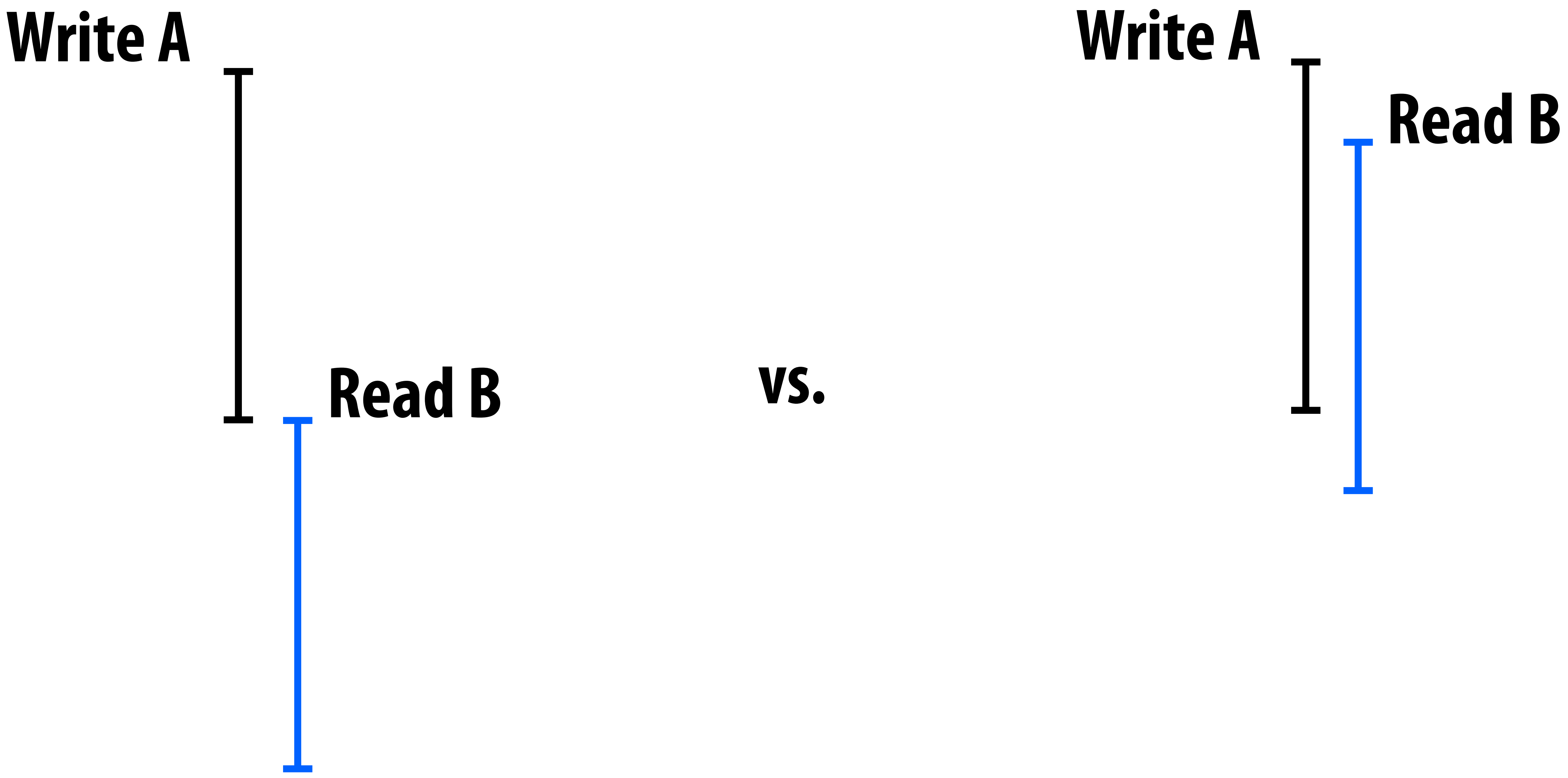
Thread 2 (on P2)

```
B = 1;
if (A == 0)
  print “World”;
```

(If this was a sequential program, it would not violate program correctness if a processor chose to reorder them… e.g., execute them concurrently)

Motivation for relaxed consistency: hiding latency

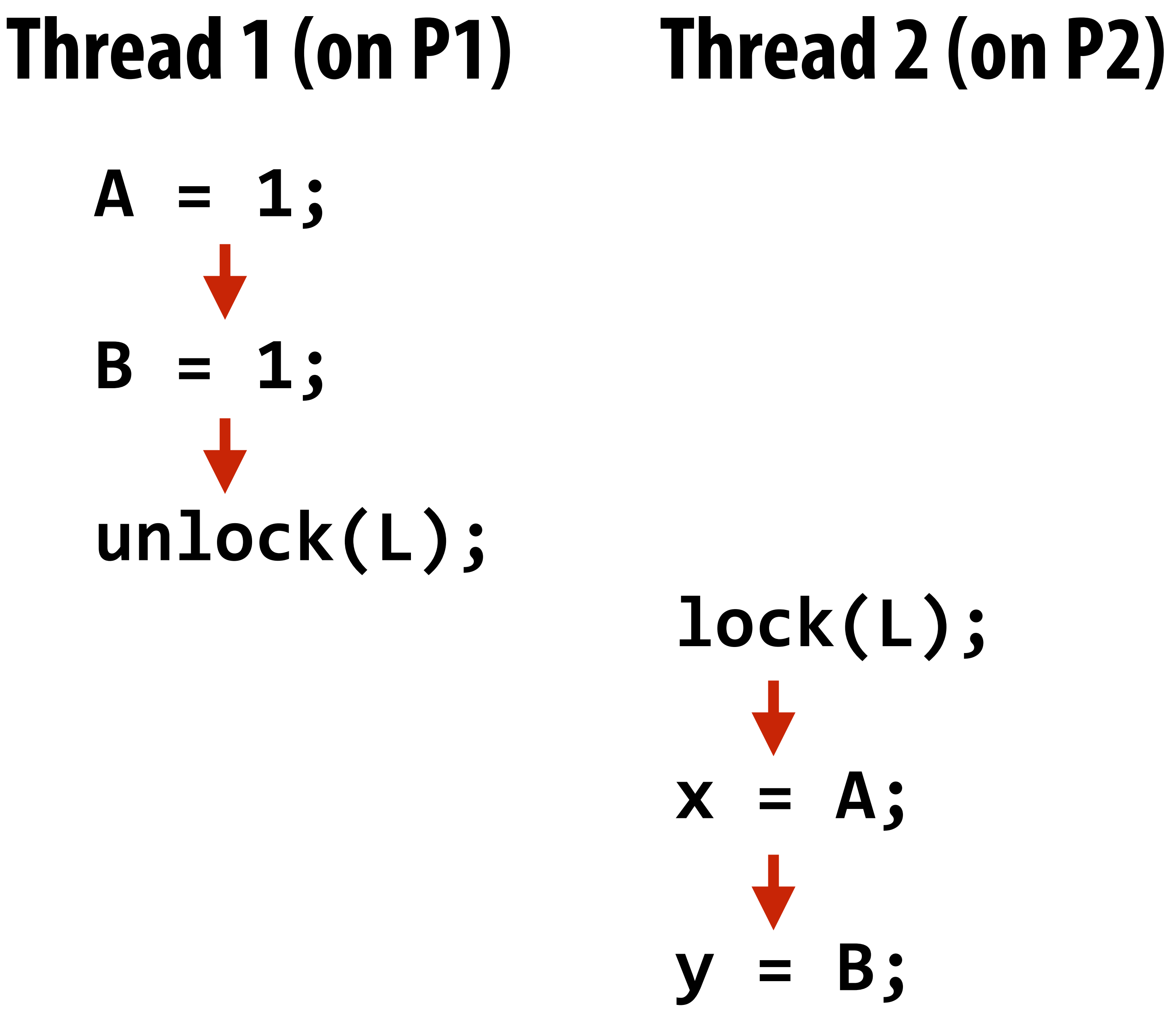
- **Why are we interested in relaxing ordering requirements?**
  - To gain performance
  - Specifically, hiding memory latency: overlap memory access operations with other operations when they are independent
  - Remember, memory access in a cache coherent system may entail much more work then simply reading bits from memory (finding data, sending invalidations, etc.)



Another way of thinking about relaxed ordering

Program order

(dependencies in red: required for sequential consistency)



Thread 1 (on P1)

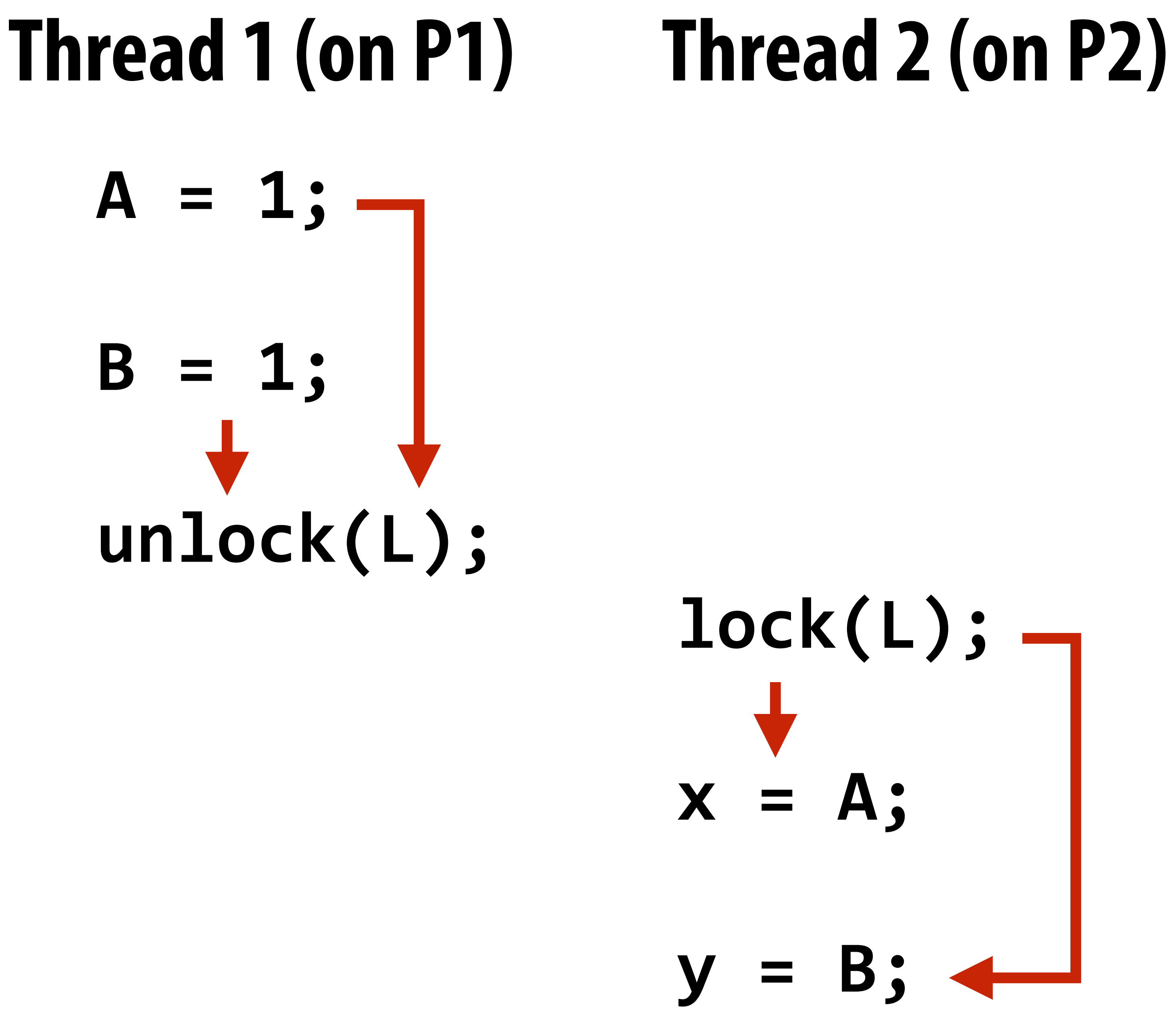
```
A = 1;
B = 1;
unlock(L);
```

Thread 2 (on P2)

```
lock(L);
x = A;
y = B;
```

"Sufficient" order for correctness

(logical dependencies in red)



Thread 1 (on P1)

```
A = 1;
B = 1;
unlock(L);
```

Thread 2 (on P2)

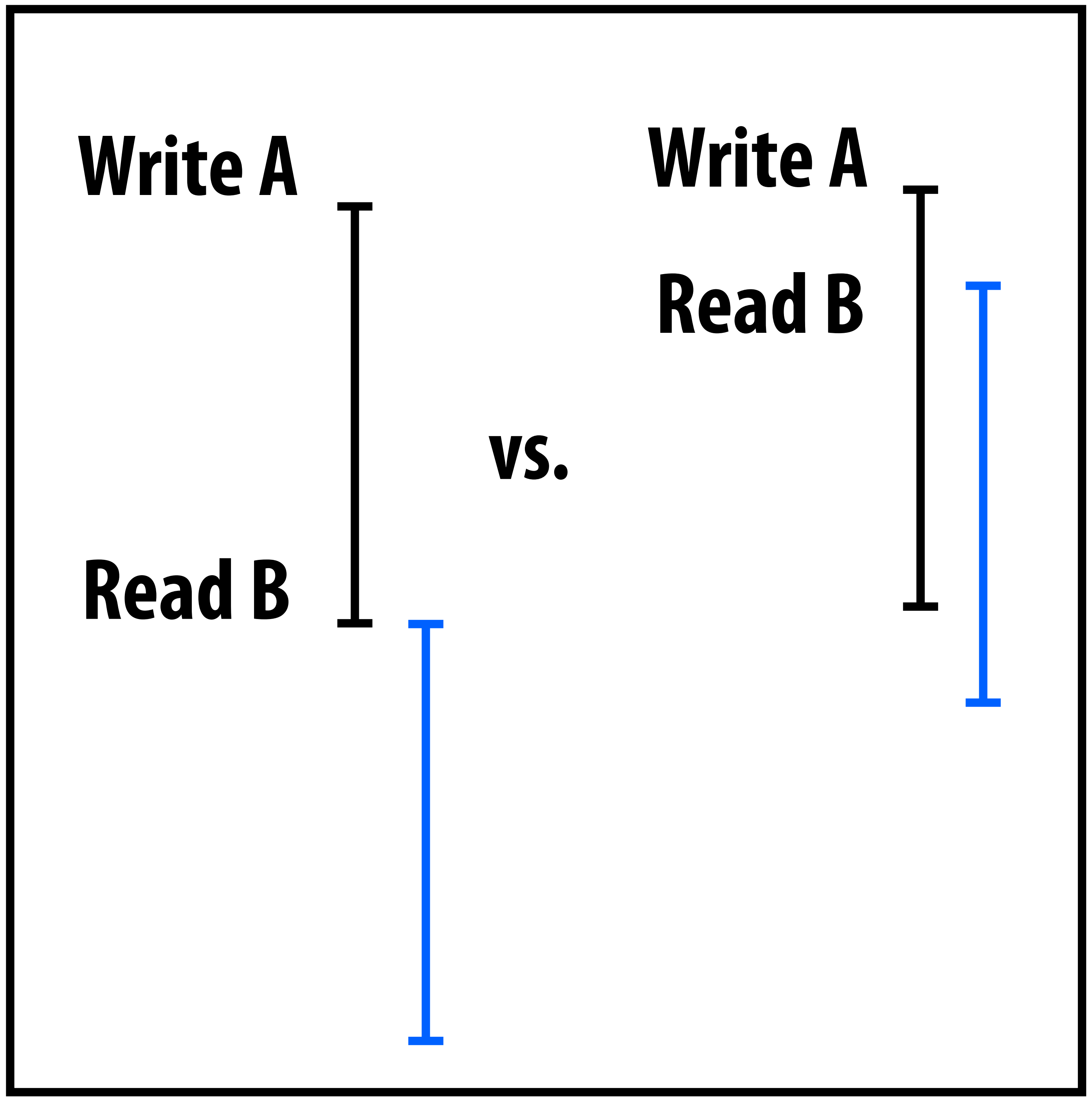
```
lock(L);
x = A;
y = B;
```

An intuitive notion of correct = execution produces the same results as a sequentially consistent system

Allowing reads to move ahead of writes

- Four types of memory operation orderings
  - ~~W→R: write must complete before subsequent read~~
  - R→R: read must complete before subsequent read
  - R→W: read must complete before subsequent write
  - W→W: write must complete before subsequent write

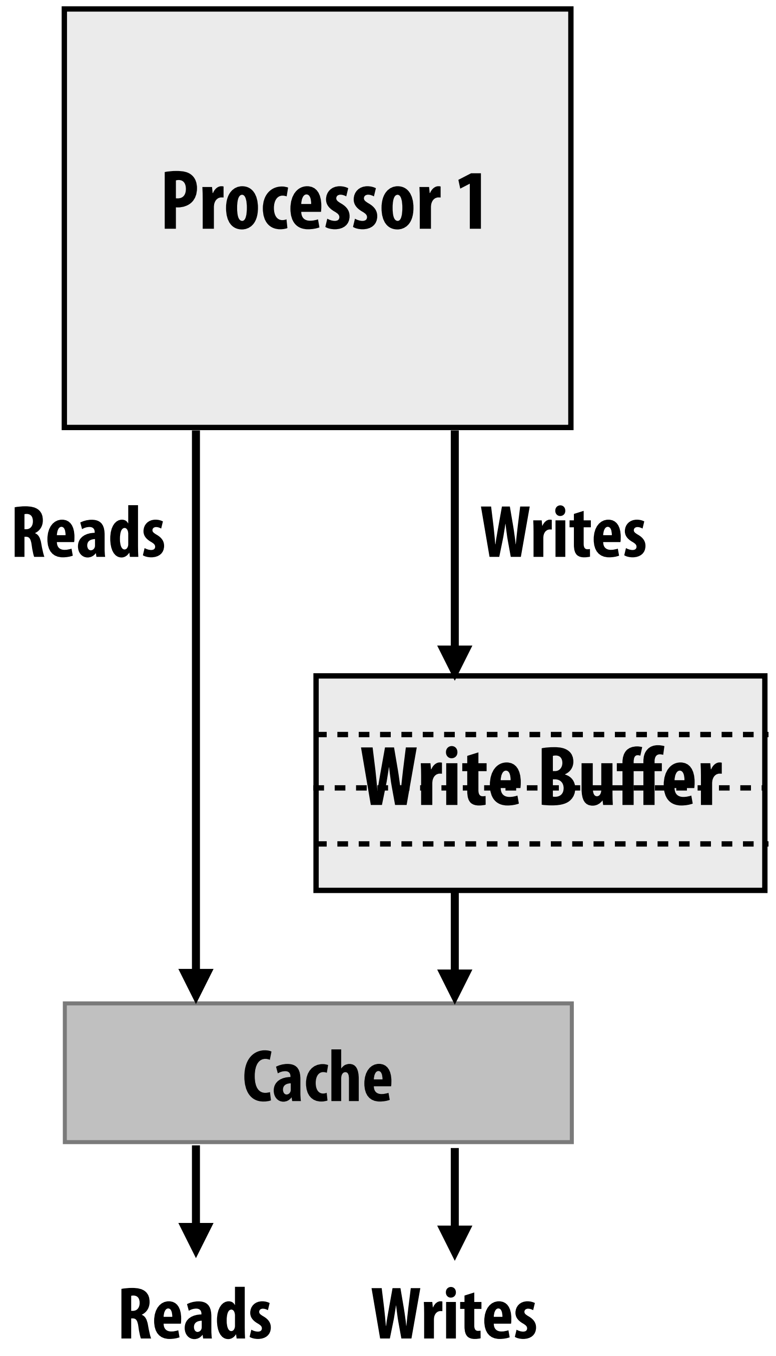
- Allow processor to hide latency of writes
  - Total Store Ordering (TSO)
  - Processor Consistency (PC)



Write buffering example

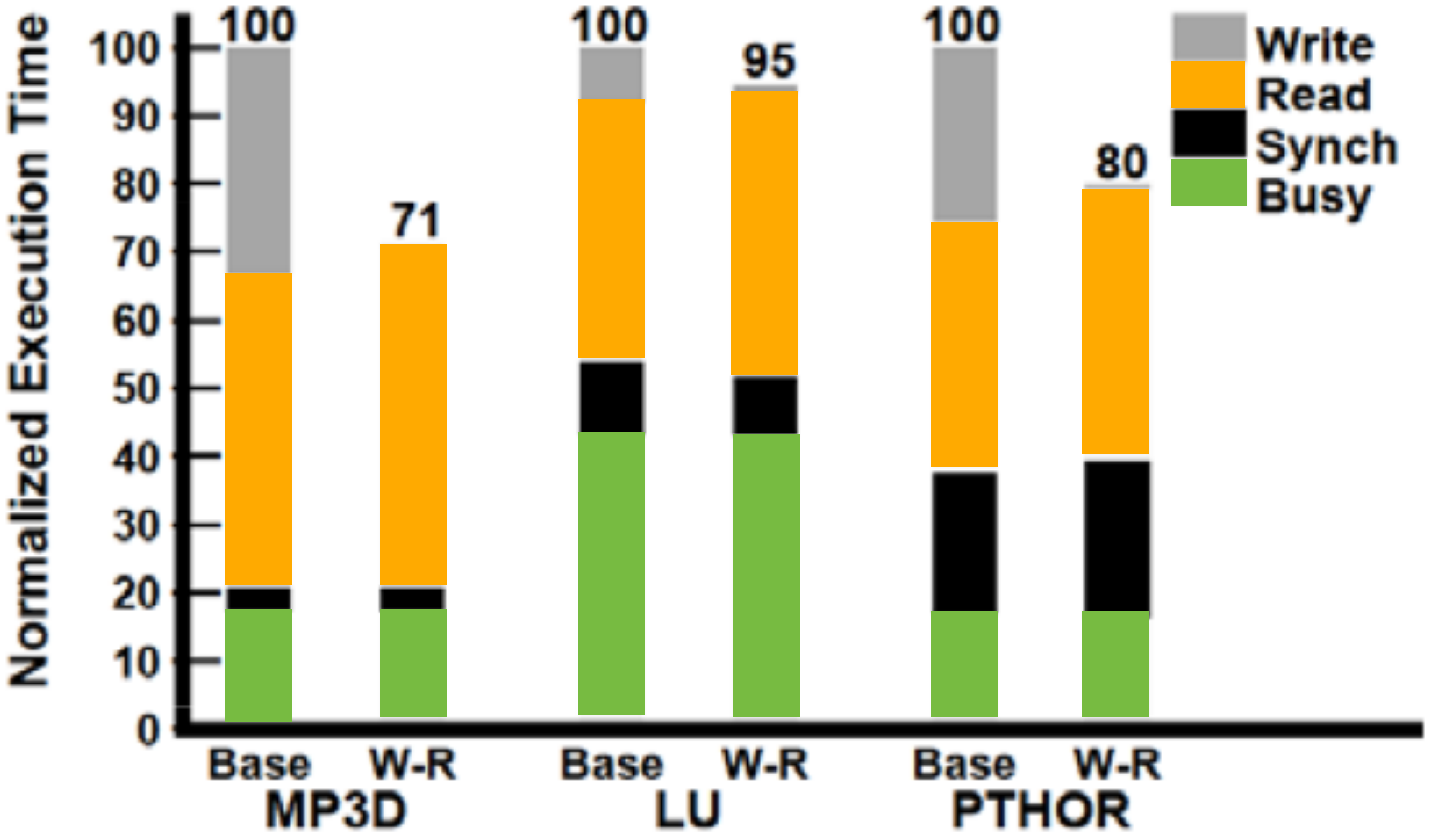
- Write buffering is a common processor optimization that allows reads to proceed in front of prior writes
  - When store is issued, processor buffers store in write buffer (assume store is to address X)
  - Processor immediately begins executing subsequent loads, provided they are not accessing address X (exploits ILP in program)
  - Further writes can also be added to write buffer (write buffer is processed in order, there is no W→W reordering)

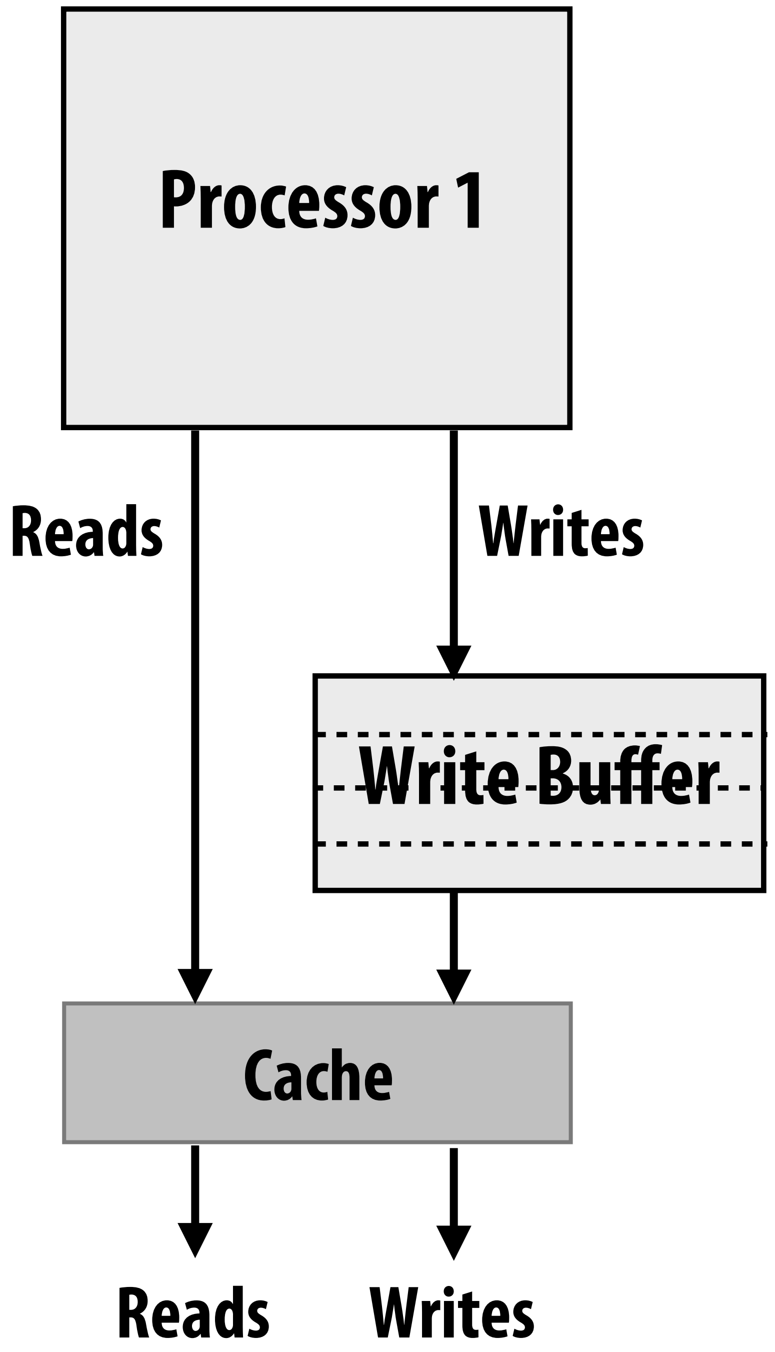
- Write buffering relaxes W→R ordering



* Do not confuse a write buffer (shown here) with a cache's write-back buffer (discussed last lecture). Both buffers exist to hide the latency of memory operations. However, the write buffer holds writes that have been issued by the processor, but not yet committed in the system. The write-back buffer holds dirty cache lines that must be flushed to memory so memory stays up to date. The lines are dirty because there was some write to them completed by the processor a long time ago. (This is a good distinction to discuss in comments.)

Relaxed consistency performance





Base: Sequentially consistent execution. Processor issues one memory operation at a time, stalls until completion

W-R: relaxed W → R ordering constraint (write latency almost fully hidden)

Allowing reads to move ahead of writes

- **Total store ordering (TSO)**
  - **Processor P can read B before its write to A is seen by all processors (processor can move its own reads in front of its own writes)**
  - **Reads by other processors cannot return new value of A until the write to A is observed by all processors**

- **Processor consistency (PC)**
  - **Any processor can read new value of A before the write is observed by all processors**

- **In TSO and PC, only W → R order is relaxed. The W → W constraint still exists. Writes by the same thread are not reordered (they occur in program order)**

Four example programs

Assume A and B are initialized to 0
Assume prints are loads

❶

Thread 1 (on P1)

```
A = 1;
flag = 1;
```

Thread 2 (on P2)

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

❷

Thread 1 (on P1)

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

Thread 2 (on P2)

```
print B;
print A;
```

❸

Thread 1 (on P1)

```
A = 1;
```

Thread 2 (on P2)

```
while (A == 0);
B = 1;
```

Thread 3 (on P3)

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

❹

Thread 1 (on P1)

```
A = 1;
print B;
```

Thread 2 (on P2)

```
B = 1;
print A;
```

Do results of execution match that of sequential consistency (SC)

	1	2	3	4
Total Store Ordering (TSO)	✓	✓	✓	✗
Processor Consistency (PC)	✓	✓	✗	✗

Clarification

- **The cache coherency problem exists because of the optimization of duplicating data in multiple processor caches. The copies of the data must be kept coherent.**

- **Relaxed memory consistency issues arise from the optimization of reordering memory operations. (Consistency is unrelated to whether or not caches exist in the system.)**

Allowing writes to be reordered

- **Four types of memory operation orderings**
  - ~~W→R: write must complete before subsequent read~~
  - R→R: read must complete before subsequent read
  - R→W: read must complete before subsequent write
  - ~~W→W: write must complete before subsequent write~~

- **Partial Store Ordering (PSO)**
  - Execution may not match sequential consistency on program 1 (P2 may observe change to `flag` before change to `A`)

Thread 1 (on P1)	Thread 2 (on P2)
`A = 1;`	`while (flag == 0);`
`flag = 1;`	`print A;`

Why might it be useful to allow more aggressive memory operation reorderings?

- W → W: processor might reorder write operations in a write buffer (e.g., one is a cache miss while the other is a hit)

- R → W, R → R: processor might reorder independent instructions in an instruction stream (out-of-order execution)

- Keep in mind these are all valid optimizations if a program consists of a single instruction stream

Allowing all reorderings

- Four types of memory operation orderings
  - ~~W→R: write must complete before subsequent read~~
  - ~~R→R: read must complete before subsequent read~~
  - ~~R→W: read must complete before subsequent write~~
  - ~~W→W: write must complete before subsequent write~~

- Examples:
  - Weak ordering (WO)
  - Release Consistency (RC)
    - Processors support special synchronization operations
    - Memory accesses before memory `fence` instruction must complete before the `fence` issues
    - Memory accesses after fence cannot begin until fence instruction is complete

```
reorderable reads
and writes here

...

MEMORY FENCE

...

reorderable reads
and writes here

...

MEMORY FENCE
```

Example: expressing synchronization in relaxed models

- **Intel x86/x64 ~ total store ordering**
  - **Provides sync instructions if software requires a specific instruction ordering not guaranteed by the consistency model**
    - **`mm_lfence` ("load fence": wait for all loads to complete)**
    - **`mm_sfence` ("store fence": wait for all stores to complete)**
    - **`mm_mfence` ("mem fence": wait for all me operations to complete)**

- **ARM processors: very relaxed consistency model**

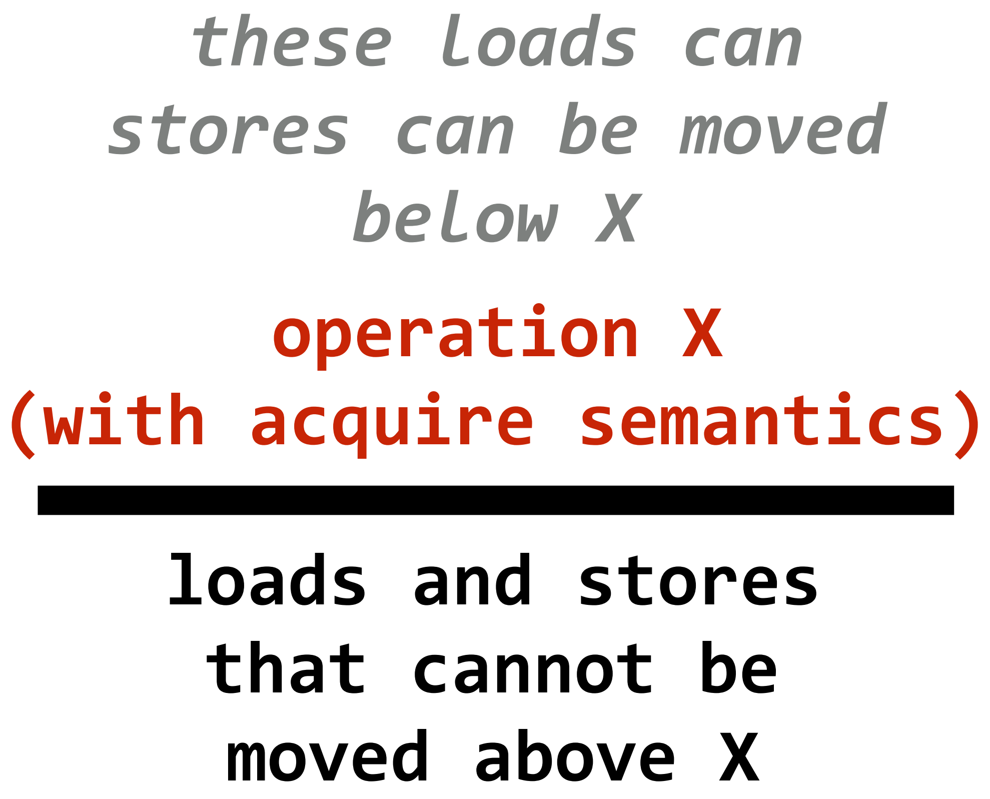
A cool post on the role of memory fences in x86:
http://bartoszmilewski.com/2008/11/05/who-ordered-memory-fences-on-an-x86/

ARM has some great examples in their programmer's reference:
http://infocenter.arm.com/help/topic/com.arm.doc.genc007826/Barrier_Litmus_Tests_and_Cookbook_A08.pdf

A great list:
http://www.cl.cam.ac.uk/~pes20/weakmemory/

Acquire/release semantics

- Operation X with acquire semantics: prevent reordering of X with any load/store after X in program order
  - Other processors see X's effect before the effect of all subsequent operations
  - Example: taking a lock must have acquire semantics

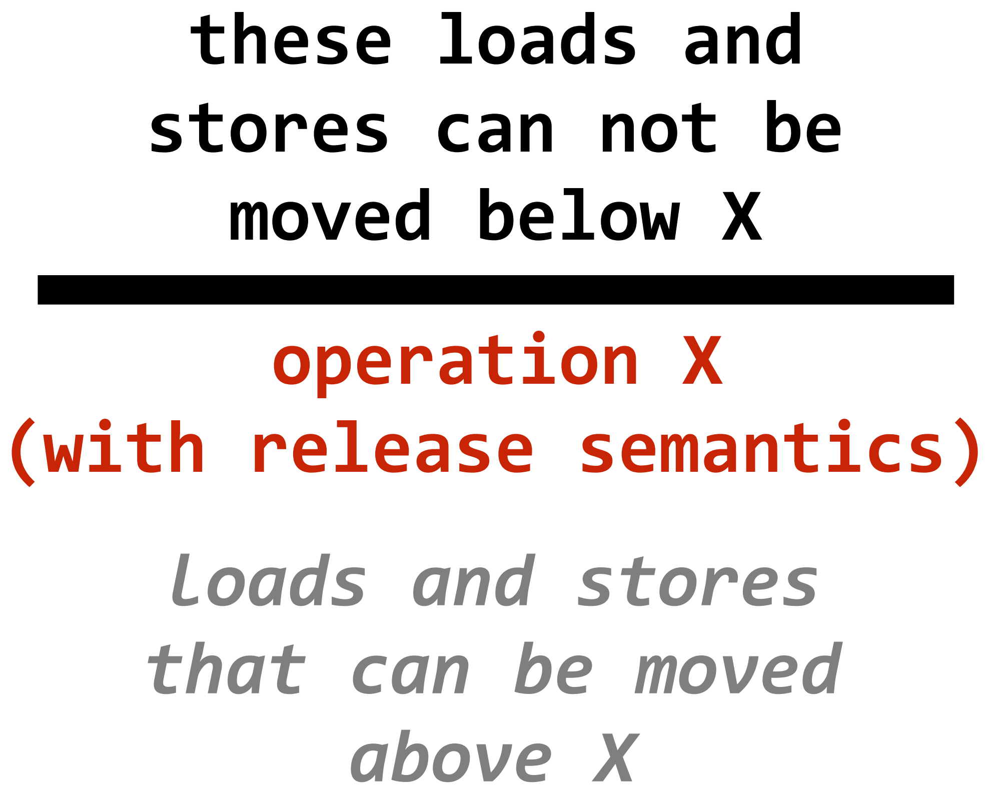


these loads can stores can be moved below X

operation X (with acquire semantics)

loads and stores that cannot be moved above X

- Operation X with release semantics: prevent reordering of X with any load/store before X in program order
  - Other processors see effects of all prior operations before seeing effect of X.
  - Example: releasing a lock must have release semantics



these loads and stores can not be moved below X

operation X (with release semantics)

loads and stores that can be moved above X

C++ 11 atomic<T>

- Provides atomic read, write, read-modify-write of entire objects
  - Atomicity may be implemented by mutex or efficiently by processor-supported atomic instructions (if T is a basic type)
  - More on this after spring break

- Provides memory ordering semantics for operations before and after atomic operations
  - By default: sequential consistency
  - See std::memory_order or more detail

Thread 1 (on P1)

```
atomic<int> ready;
int foo;

foo = 1;
ready.store(1);
```

Thread 2 (on P2)

```
// other code…
while (ready.load()==0);
// use foo here…
```

C++ atomic ensures sequentially consistent behavior by default, so compiler must emit appropriate fences on x86

C++ 11 atomic<T>

- Provides atomic read, write, read-modify-write of entire objects
  - Atomicity may be implemented by mutex or efficiently by processor-supported atomic instructions (if T is a basic type)
  - More on this after spring break

- Provides memory ordering semantics for operations before and after atomic operations
  - By default: sequential consistency
  - See std::memory_order or more detail

Thread 1 (on P1)

```
atomic<int> ready;
int foo;

foo = 1;
ready.store(1, memory_order_release);
```

Thread 2 (on P2)

```
// other code…
while (ready.load(memory_order_acquire)==0);
// use foo here…
```

No fence required on x86

Conflicting data accesses

- Two memory accesses by different processors conflict if…
  - They access the same memory location
  - At least one is a write

- Unsynchronized program
  - Conflicting accesses not ordered by synchronization (e.g., a fence, operation with release/acquire semantics, barrier, etc.)
  - Unsynchronized programs contain data races: the output of the program depends on relative speed of processors (non-deterministic program results)

Synchronized programs

- **Synchronized programs yield SC results on non-SC systems**
  - Synchronized programs are data-race-free

- **In practice, most programs you encounter will be synchronized (via locks, barriers, etc. implemented in synchronization libraries)**
  - Rather than via ad-hoc reads/writes to shared variables like in the earlier "four example programs" slide

Summary: relaxed consistency

- **Motivation: obtain higher performance by allowing recording of memory operations (reordering is not allowed by sequential consistency)**

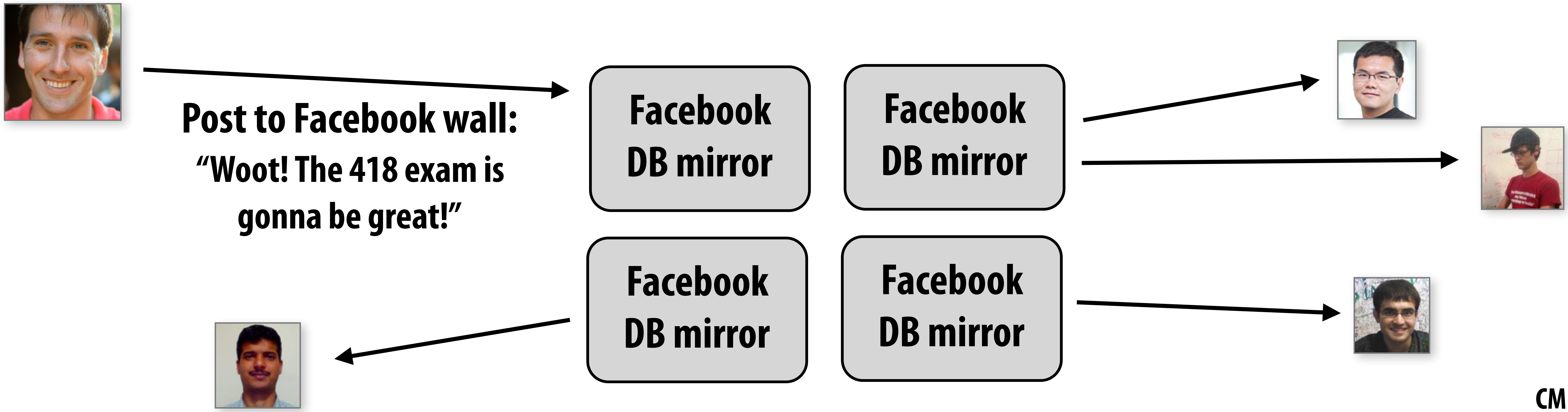
- **One cost is software complexity: programmer or compiler must correctly insert synchronization to ensure certain specific operation orderings when needed**
  - **But in practice complexities encapsulated in libraries that provide intuitive primitives like lock/unlock, barrier (or lower level primitives like fence)**
  - **Optimize for the common case: most memory accesses are not conflicting, so don't design a system that pays the cost as if they are**

- **Relaxed consistency models differ in which memory ordering constraints they ignore**

Eventual consistency in distributed systems

- For many of you, relaxed memory consistency will be a key factor in writing web-scale programs in distributed environments

- "Eventual consistency"
  - Say machine A writes to an object X in a shared distributed database
  - Many copies of database exist for performance scaling and redundancy
  - Eventual consistency guarantees that if there are no other updates to X, A's update will eventually be observed by all other nodes in the system (note: no guarantees on when, so updates to objects X and Y might propagate to different clients differently)



Post to Facebook wall:
"Woot! The 418 exam is gonna be great!"

Facebook DB mirror

Facebook DB mirror

Facebook DB mirror

Facebook DB mirror

If time: course so far review

(a more-or-less randomly selected collection of topics from previous lectures)

Exam details

- Closed book, closed laptop
- One "post it" note (but we'll let you use both sides)
- Covers all lecture material so far in course, including today's discussion of memory consistency
- The TAs will lead a review session on Monday at 6:30 in Rashid
    - Please come with questions

Throughput vs. latency

THROUGHPUT

The rate at which work gets done.

- Operations per second
- Bytes per second (bandwidth)
- Tasks per hour

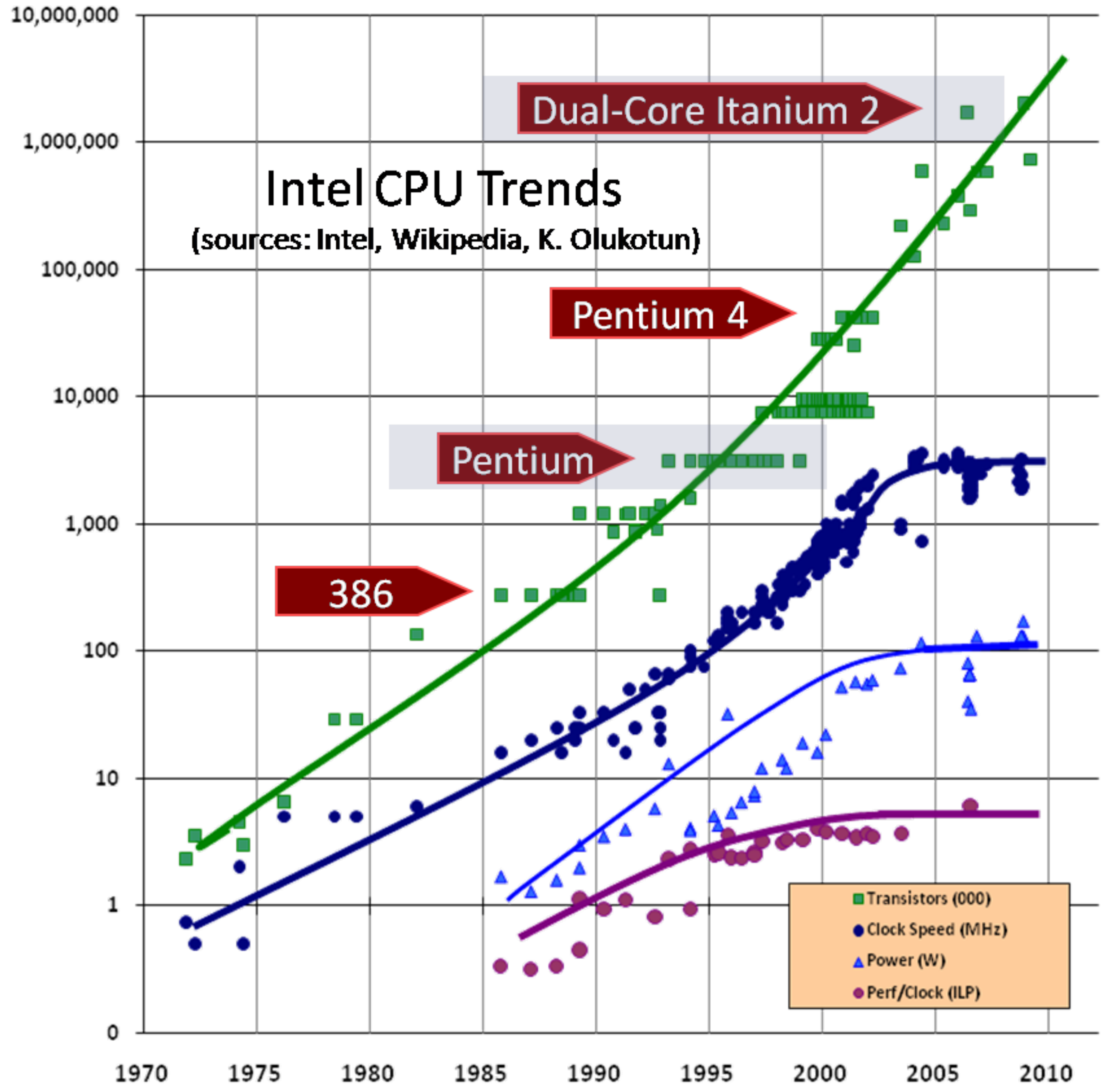
LATENCY

The amount of time for an operation to complete

- An instruction takes 4 clocks
- A cache miss takes 200 clocks to complete
- It takes 20 seconds for a program to complete

Ubiquitous parallelism

- What motivated the shift toward multi-core parallelism in modern processor design?
  - Inability to scale clock frequency due to power limits
  - Diminishing returns when trying to further exploit ILP



Is the new performance focus on throughput, or latency?

Techniques for exploiting independent operations in applications

What is it? What is the benefit?

1. **superscalar execution**

   Processor executes multiple instructions per clock. Super-scalar execution exploits instruction level parallelism (ILP). When instructions in the same thread of control are independent they can be executed in parallel on a super-scalar processor.

2. **SIMD execution**

   Processor executes the same instruction on multiple pieces of data at once (e.g., one operation on vector registers). The cost of fetching and decoding the instruction is amortized over many arithmetic operations.

3. **multi-core execution**

   A chip contains multiple (mainly) independent processing cores, each capable of executing independent instruction streams.

4. **multi-threaded execution**

   Processor maintains execution contexts (state: e.g, a PC, registers, virtual memory mappings) for multiple threads. Execution of thread instructions is interleaved on the core over time. Multi-threading reduces processor stalls by automatically switching to execute other threads when one thread is blocked waiting for a long-latency operation to complete.

Techniques for exploiting independent operations in applications

	Who is responsible for mapping?
1. superscalar execution	Usually not a programmer responsibility: ILP automatically detected by processor hardware or by compiler (or both) (But manual loop unrolling by a programmer can help)
2. SIMD execution	In simple cases, data parallelism is automatically detected by the compiler, (e.g., assignment 1 saxpy). In practice, programmer explicitly describes SIMD execution using vector instructions or by specifying independent execution in a high-level language (e.g., ISPC gangs, CUDA)
3. multi-core execution	Programmer defines independent threads of control. e.g., pthreads, ISPC tasks, openMP #pragma
4. multi-threaded execution	Programmer defines independent threads of control. But programmer must create more threads than processing cores.

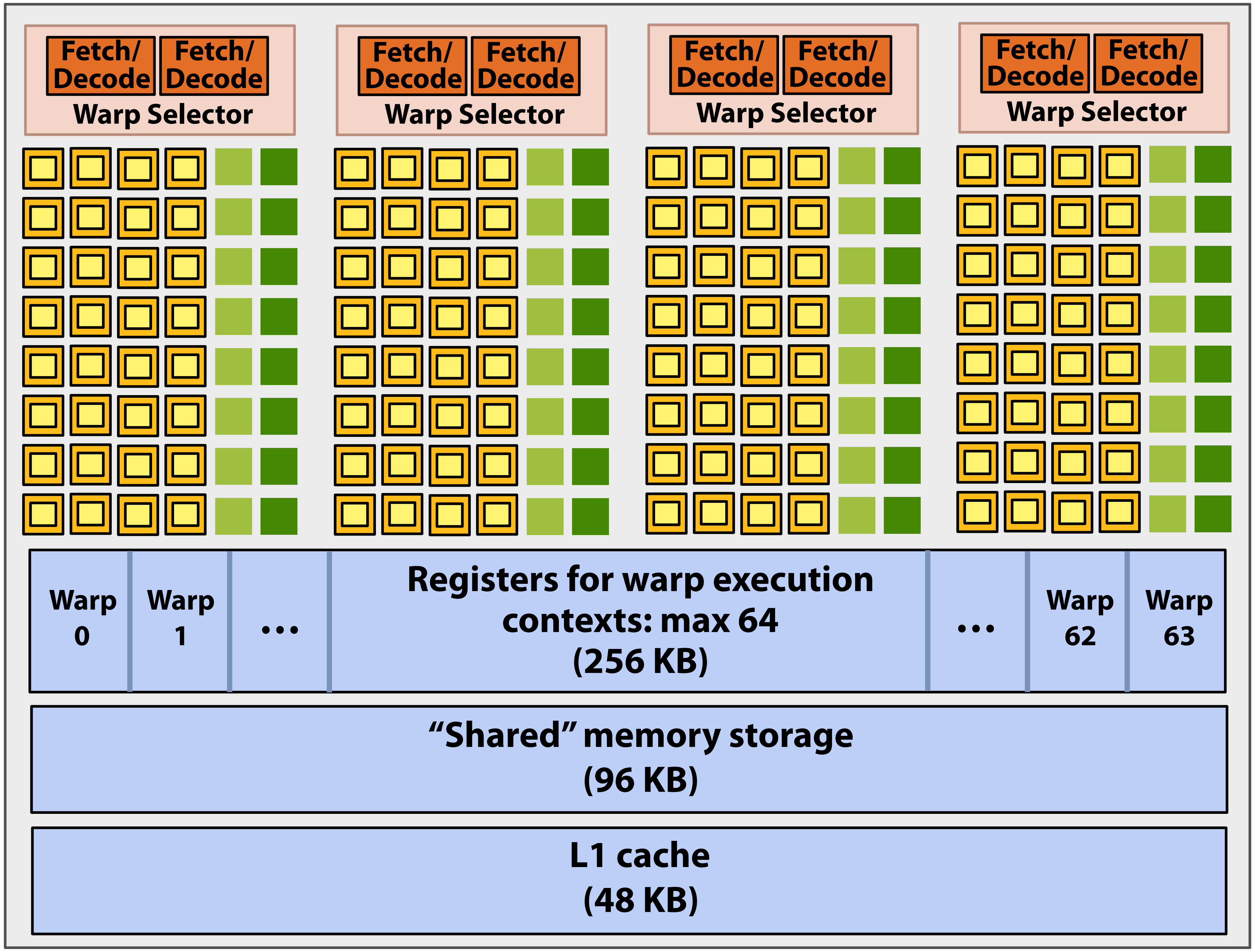
Frequently discussed processor examples

- **Intel Core i7 CPU**
  - 4 cores
  - Each core:
    - Supports 2 threads (“Hyper-Threading”)
    - Can issue 8-wide SIMD instructions (AVX instructions) or 4-wide SIMD instructions (SSE)
    - Can execute multiple instructions per clock (superscalar)

- **NVIDIA GTX 1080 GPU**
  - 20 “SM cores”
  - Each core:
    - Supports up to 64 warps (warp is a group of 32 “CUDA threads”)
    - Issues 32-wide SIMD instructions (same instruction for all 32 “CUDA threads” in a warp)
    - Also capable of issuing multiple instructions per clock

- **Intel Xeon Phi (KNL)**
  - 72 cores
  - Each core: supports 4 threads, 16-wide SIMD instructions
  - Also capable of multiple instructions per clock (up to 2 SIMD instructions)

Multi-threaded, SIMD execution on GPU



🟨 = SIMD functional unit, control shared across 32 units (1 MUL-ADD per clock)

🟩 = SIMD special function unit (sin, cos, etc.)

■ = load/store

SM resource limits:

- Max warp execution contexts: 64 (2,048 total CUDA threads)
- 96 KB of shared memory

- Describe how CUDA threads are mapped to the execution resources on this GTX 1080 GPU SM?
  - e.g., describe how the processor executes instructions each clock

Decomposition: assignment 1, program 3

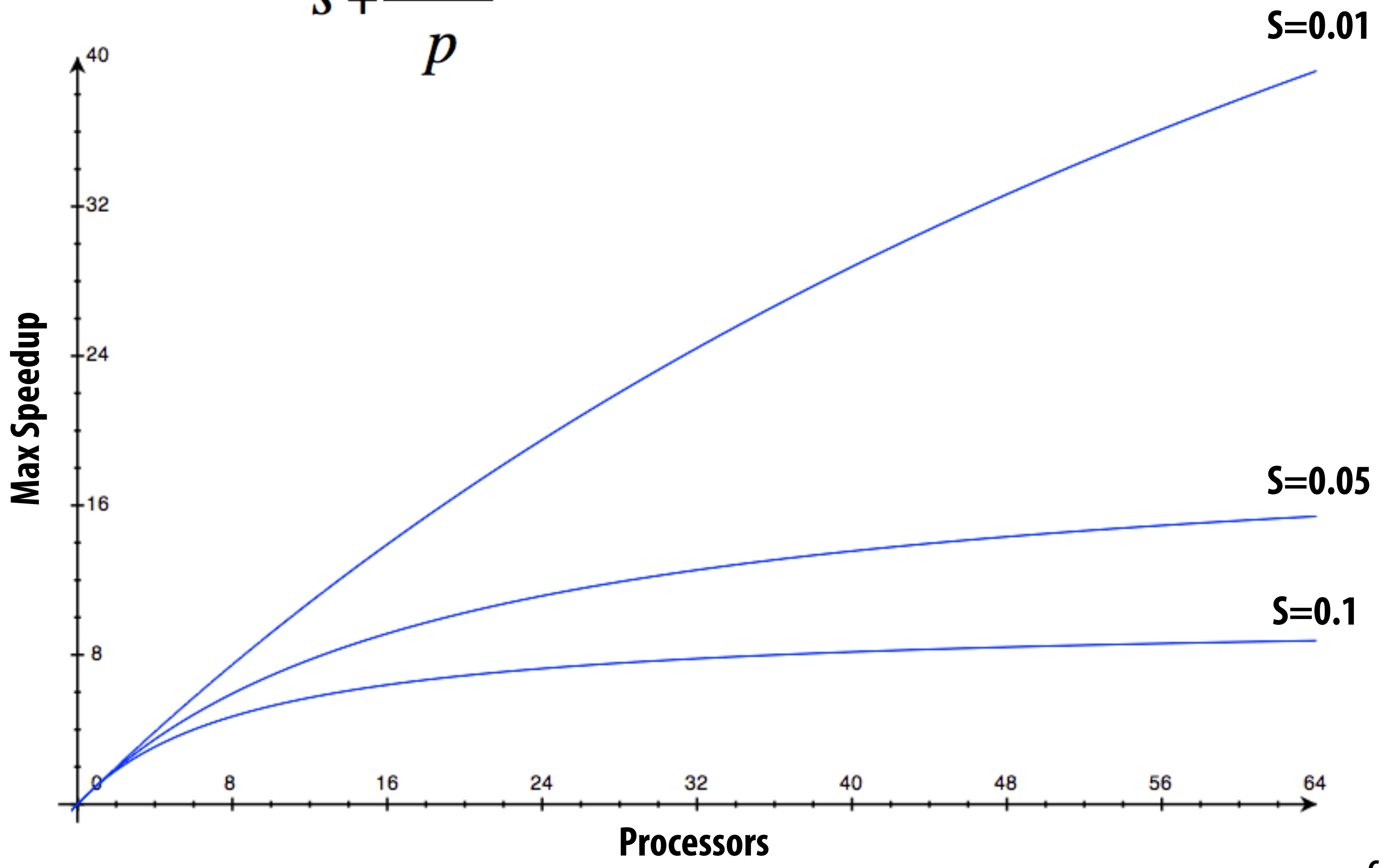
- You used ISPC to parallelize the Mandelbrot generation
- You created a bunch of tasks. How many? Why?

```
uniform int rowsPerTask = height / 2;

// create a bunch of tasks

launch[2] mandelbrot_ispc_task(
               x0, y0, x1, y1,
               width, height,
               rowsPerTask,
               maxIterations,
               output);
```

Amdahl's law

- Let S = the fraction of sequential execution that is inherently sequential
- Max speedup on P processors given by:

$$\text{speedup} \leq \frac{1}{s + \frac{1-s}{p}}$$



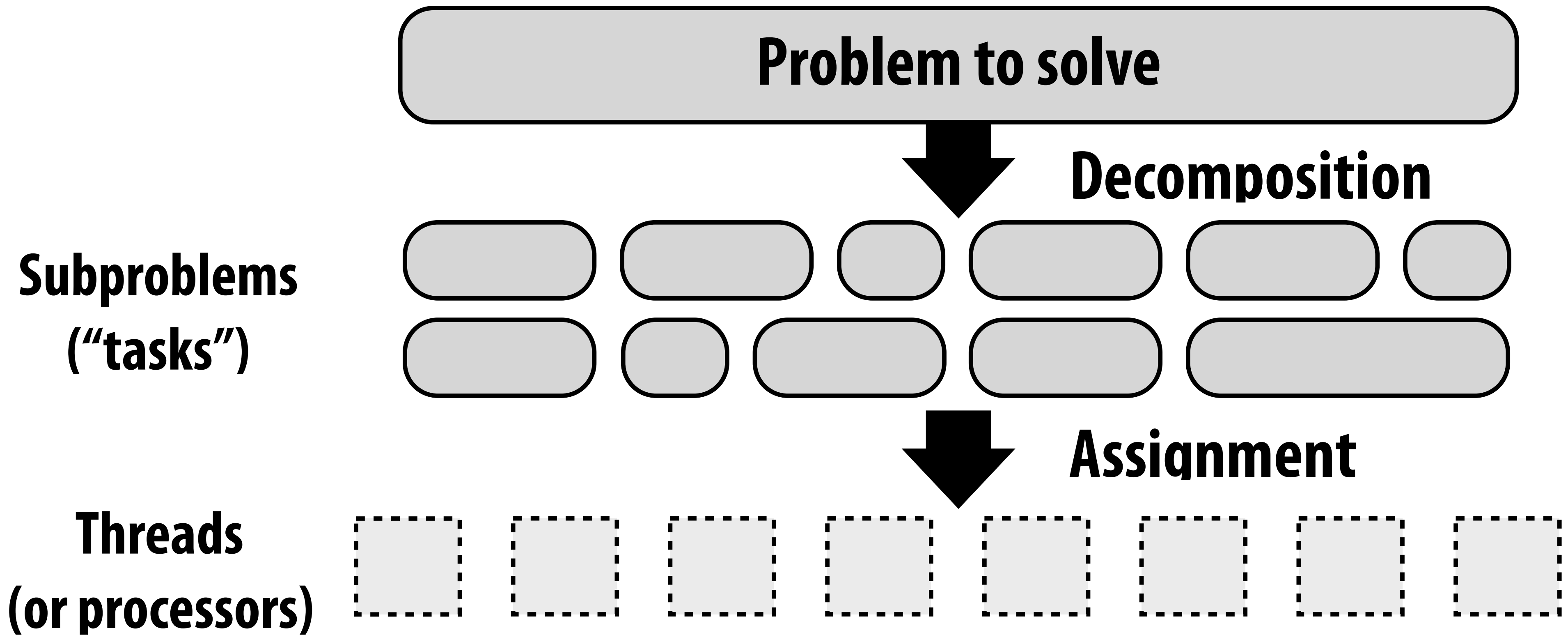
Thought experiment

- Your boss gives your team a piece of code for which 25% of the operations are inherently serial and instructs you to parallelize the application on a six-core machine. He expects you to achieve 5x speedup on this application.

- Your friend shouts at your boss, “that is %#*$(%*!@ impossible”!
- Your boss shouts back, “I want employees with a can-do attitude! You haven’t thought hard enough.”

- Who is right?

Work assignment



STATIC ASSIGNMENT

Assignment of subproblems to processors is determined before (or right at the start) of execution. Assignment does not dependent on execution behavior.

Good: very low (almost none) run-time overhead
Bad: execution time of subproblems must be predictable (so programmer can statically balance load)

Examples: solver kernel, OCEAN, mandlebrot in asst 1, problem 1, ISPC foreach

DYNAMIC ASSIGNMENT

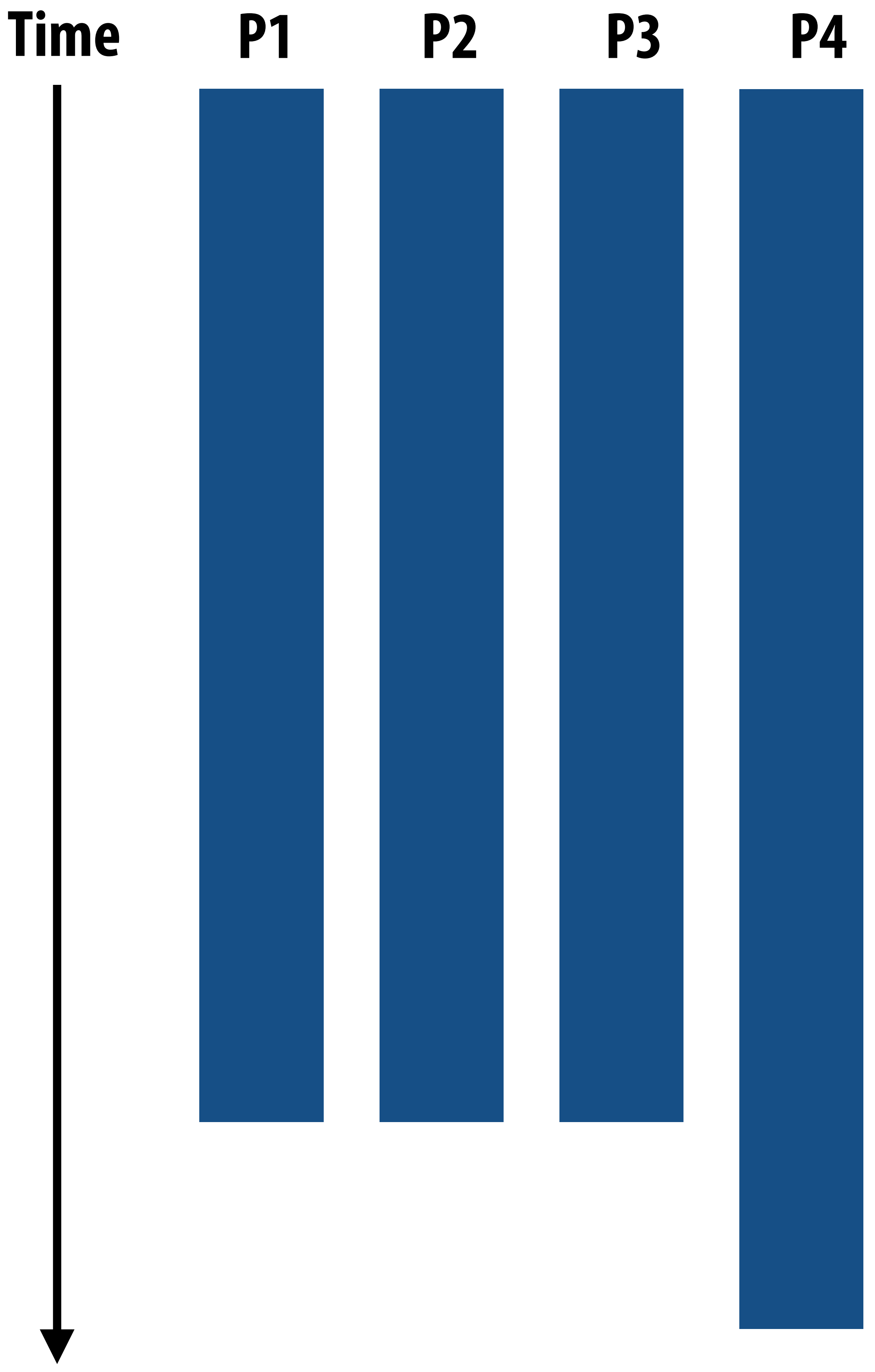
Assignment of subproblems to processors is determined as the program runs.

Good: can achieve balance load under unpredictable conditions
Bad: incurs runtime overhead to determine assignment

Examples: ISPC tasks, executing grid of CUDA thread blocks on GPU, assignment 3, shared work queue

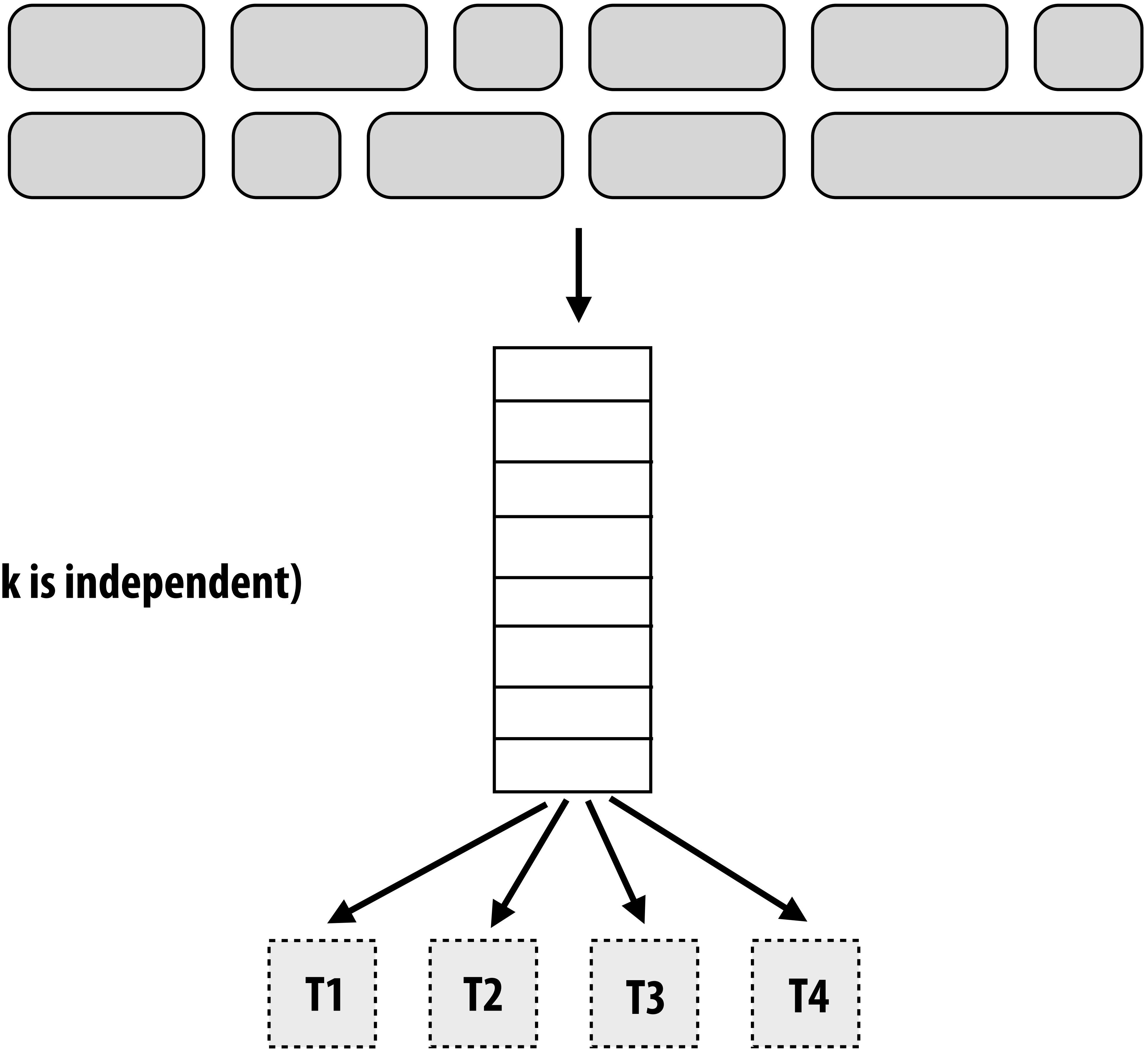
Balancing the workload

Ideally all processors are computing all the time during program execution
(they are computing simultaneously, and they finish their portion of the work at the same time)



Load imbalance can significantly reduce overall speedup

Dynamic assignment using work queues



Sub-problems
(aka "tasks", "work")

Shared work queue: a list of work to do
(for now, let's assume each piece of work is independent)

Worker threads:
Pull data from work queue
Push new work to queue as it's created

Decomposition in assignment 2

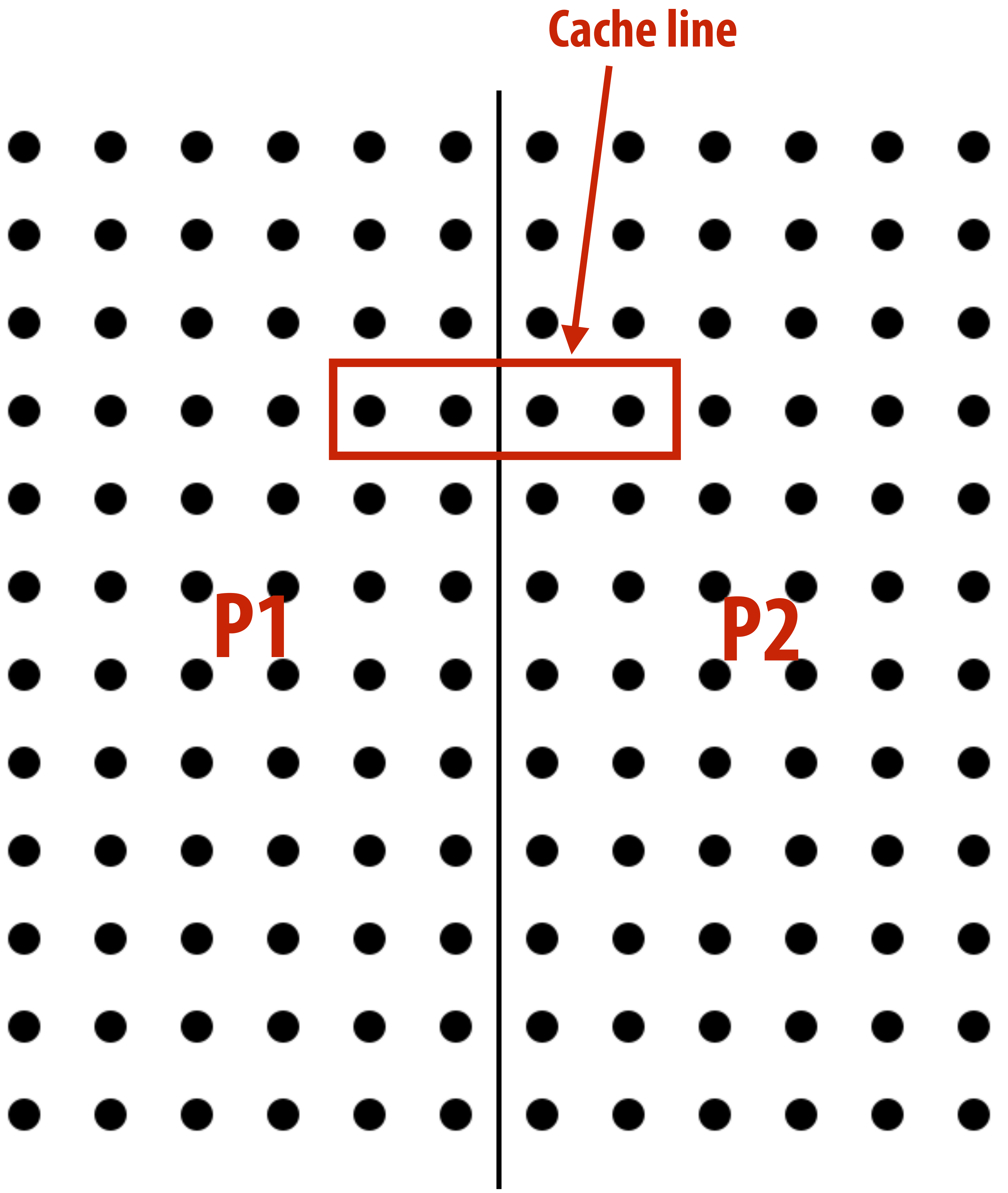
- Most solutions decomposed the problem in several ways
    - Decomposed screen into tiles (“task” per tile)
        - Decomposed tile into per circle “tasks”
        - Decomposed tile into per pixel “tasks”

Artifactual vs. inherent communication

INHERENT COMMUNICATION

ARTIFACTUAL COMMUNICATION

FALSE SHARING



Problem assignment as shown. Each processor reads/writes only from its local data.

Programming model abstractions

	Structure?	Communication?	Sync?
1. shared address space	Multiple processors sharing an address space.	Implicit: loads and stores to shared variables	Synchronization primitives such as locks and barriers
2. message passing	Multiple processors, each with own memory address space.	Explicit: send and receive messages	Build synchronization out of messages.
3. data-parallel	Rigid program structure: single logical thread containing `map(f, collection)` where "iterations" of the map can be executed concurrently	Typically not allowed within map except through special built-in primitives (like "reduce"). Comm implicit through loads and stores to address space	Implicit barrier at the beginning and end of the map.

Cache coherence

Why cache coherence?

Hand-wavy answer: would like shared memory to behave "intuitively" when two processors read and write to a shared variable. Reading a value after another processor writes to it should return the new value. (despite replication due to caches)

Requirements of a coherent address space

1. **A read by processor P to address X that follows a write by P to address X, should return the value of the write by P** ***(assuming no other processor wrote to X in between)***

2. **A read by a processor to address X that follows a write by another processor to X returns the written value... if the read and write are sufficiently separated in time** ***(assuming no other write to X occurs in between)***

3. **Writes to the same location are serialized; two writes to the same location by any two processors are seen in the same order by all processors.**
   (Example: if values 1 and then 2 are written to address X, no processor observes 2 before 1)

Condition 1: program order (as expected of a uniprocessor system)

Condition 2: write propagation: The news of the write has to eventually get to the other processors. Note that precisely <u>when</u> it is propagated is not defined by definition of coherence.

Condition 3: write serialization

Implementing cache coherence

Main idea of invalidation-based protocols: before writing to a cache line, obtain exclusive access to it

SNOOPING

Each cache broadcasts its cache misses to all other caches. Waits for other caches to react before continuing.

Good: simple, low latency

Bad: broadcast traffic limits scalability

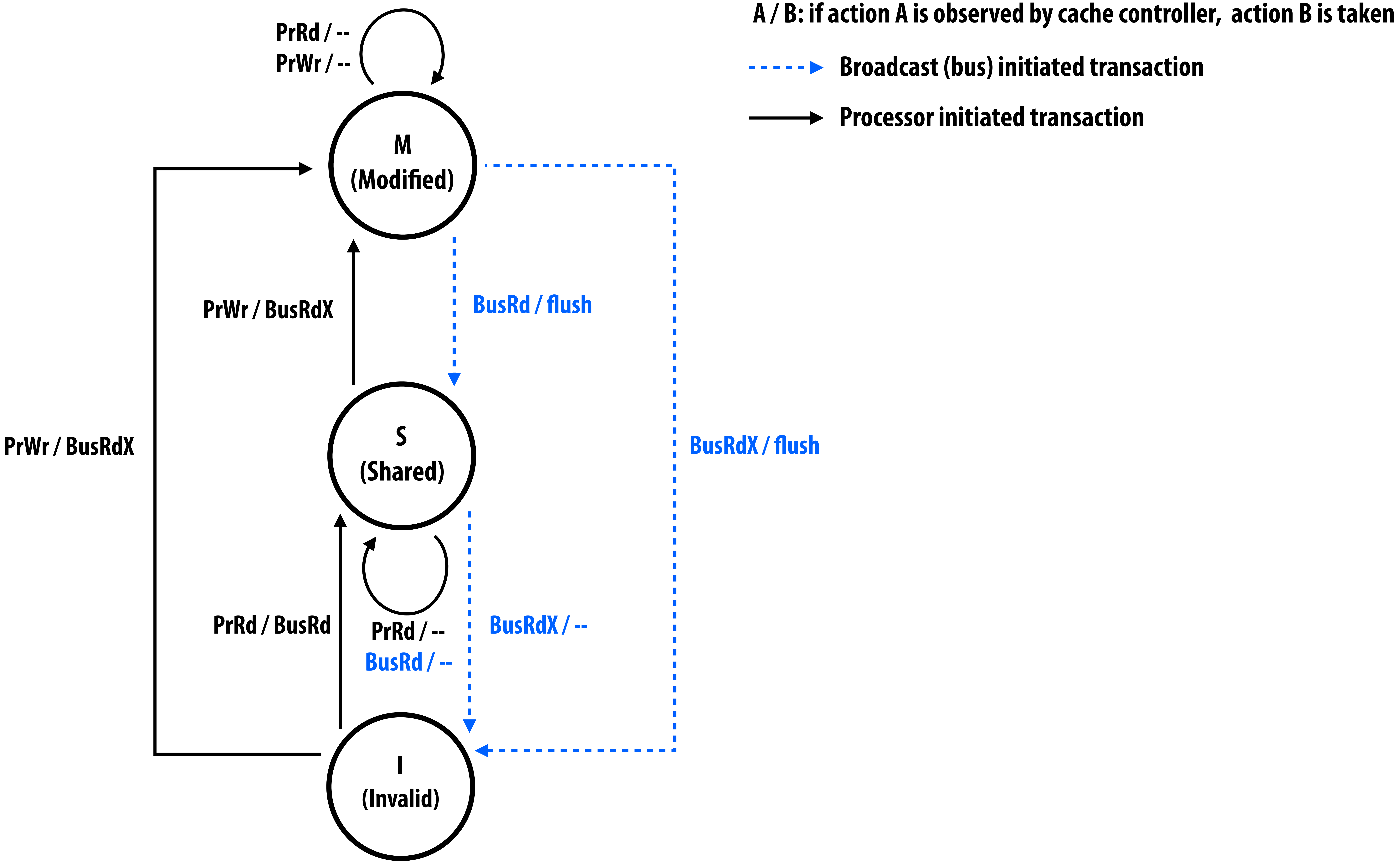
DIRECTORIES

Information about location of cache line and number of shares is stored in a centralized location. On a miss, requesting cache queries the directory to find sharers and communicates with these nodes using point-to-point messages.

Good: coherence traffic scales with number of sharers, and number of sharers is usually low

Bad: higher complexity, overhead of directory storage, additional latency due to longer critical path

MSI state transition diagram



A / B: if action A is observed by cache controller, action B is taken
Broadcast (bus) initiated transaction
Processor initiated transaction
PrRd / --
PrWr / --
M
(Modified)
PrWr / BusRdX
BusRd / flush
PrWr / BusRdX
S
(Shared)
BusRdX / flush
PrRd / BusRd
PrRd / --
BusRd / --
BusRdX / --
I
(Invalid)