Lecture 13:
Memory Consistency

+ Course-So-Far Review

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2015

Tunes

Wanna Be On Your Mind

Valerie June

(Pushin' Against a Stone)

"Yes, I wrote songs about all the midterms I took in college."

- Valerie June

Today: what you should know

- **Understand the motivation for relaxed consistency models**

- **Understand the implications of the total store ordering (TSO) and program consistency (PC) relaxed models**

Today: who should care

- **Anyone who:**
  - **Wants to implement a synchronization library**
  - **Will ever work a job in kernel (or driver) development**
  - **Likes programming ARM processors ***

* A joke, sort of.

Memory coherence vs. memory consistency

- **Memory coherence defines requirements for the observed behavior of reads and writes to the <u>same</u> memory location (ensures all processors have consistent view of same address)**
  - All processors must agree on the order of reads/writes to X
  - In other words: it is possible to put operations on a timeline and observations of all processors are consistent with that timeline.

- **"Memory consistency" defines the behavior of reads and writes to <u>different</u> locations (as observed by other processors)**
  - Coherence only guarantees that writes to address X <u>will</u> 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

P0 write: 5

P1 read (5)

P2 write: 10

P2 write: 11

P1 read (11)
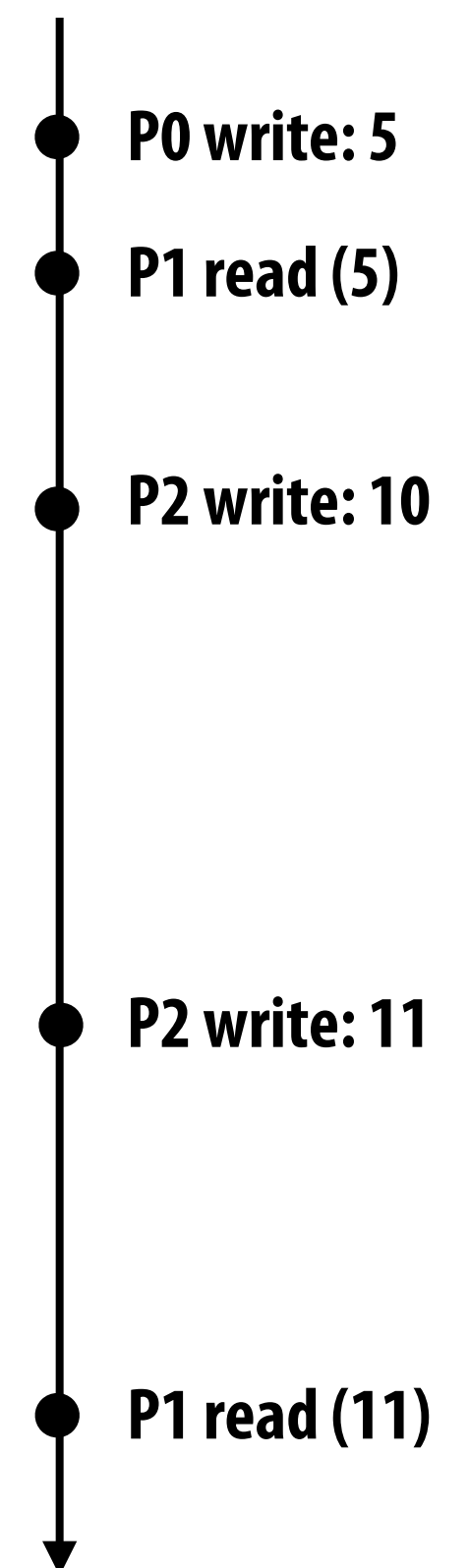
Memory operation ordering

- **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 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

- **A <u>sequentially consistent</u> memory system maintains all four memory operation orderings**
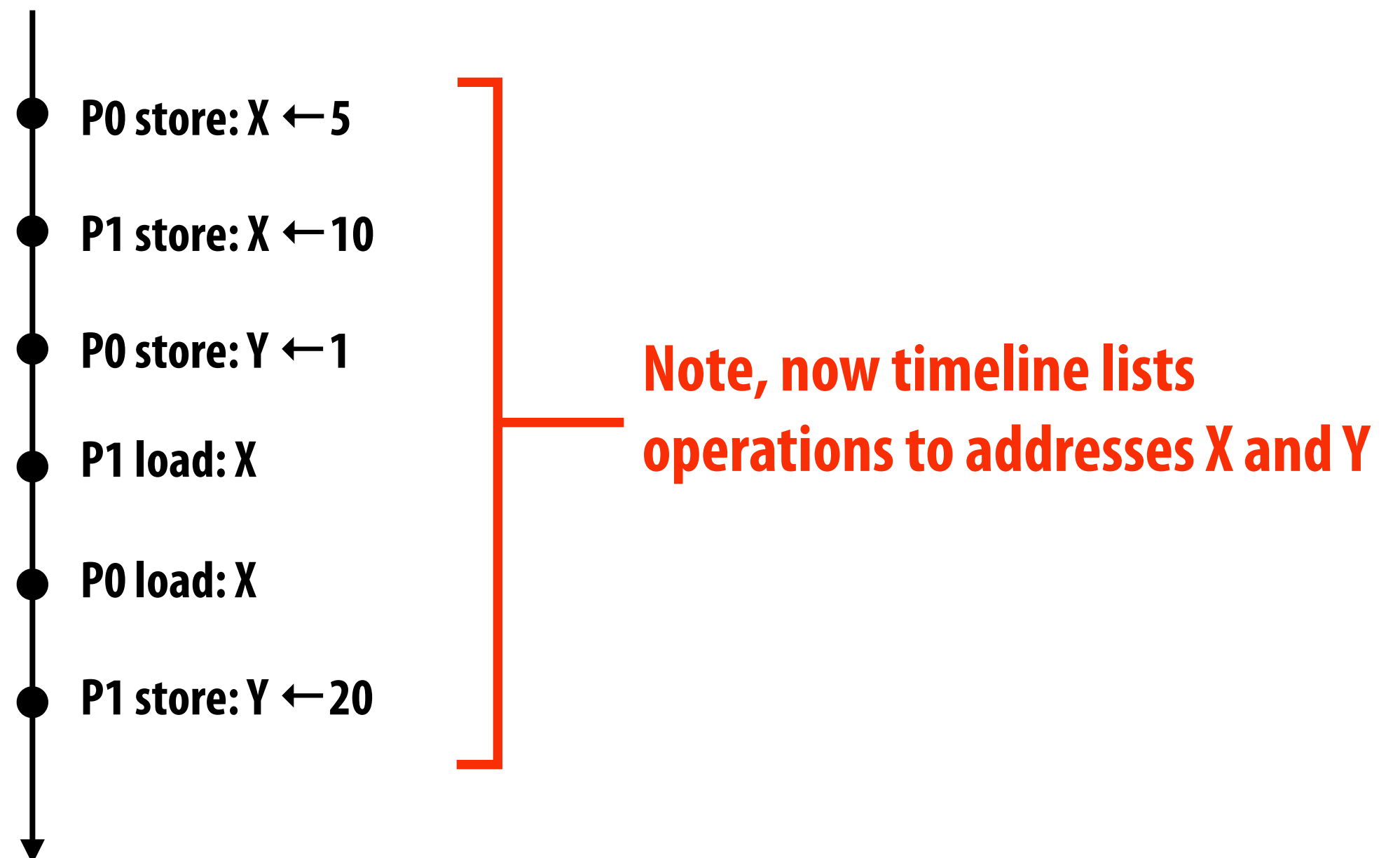
* To clarify: "write must complete before subsequent read" means:
When a write comes before a read in program order, the write must complete (its results are visible) when the read occurs.

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 <u>some</u> sequential order, and the memory operations of any one processor are executed in program order.**

**Chronology of <u>all memory operations</u> that
is consistent with observed values**

P0 store: X ← 5

P1 store: X ← 10

P0 store: Y ← 1

P1 load: X

P0 load: X

P1 store: Y ← 20

**Note, now timeline lists
operations to addresses X and Y**

[Adve and Gharachorloo 95]

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.

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)

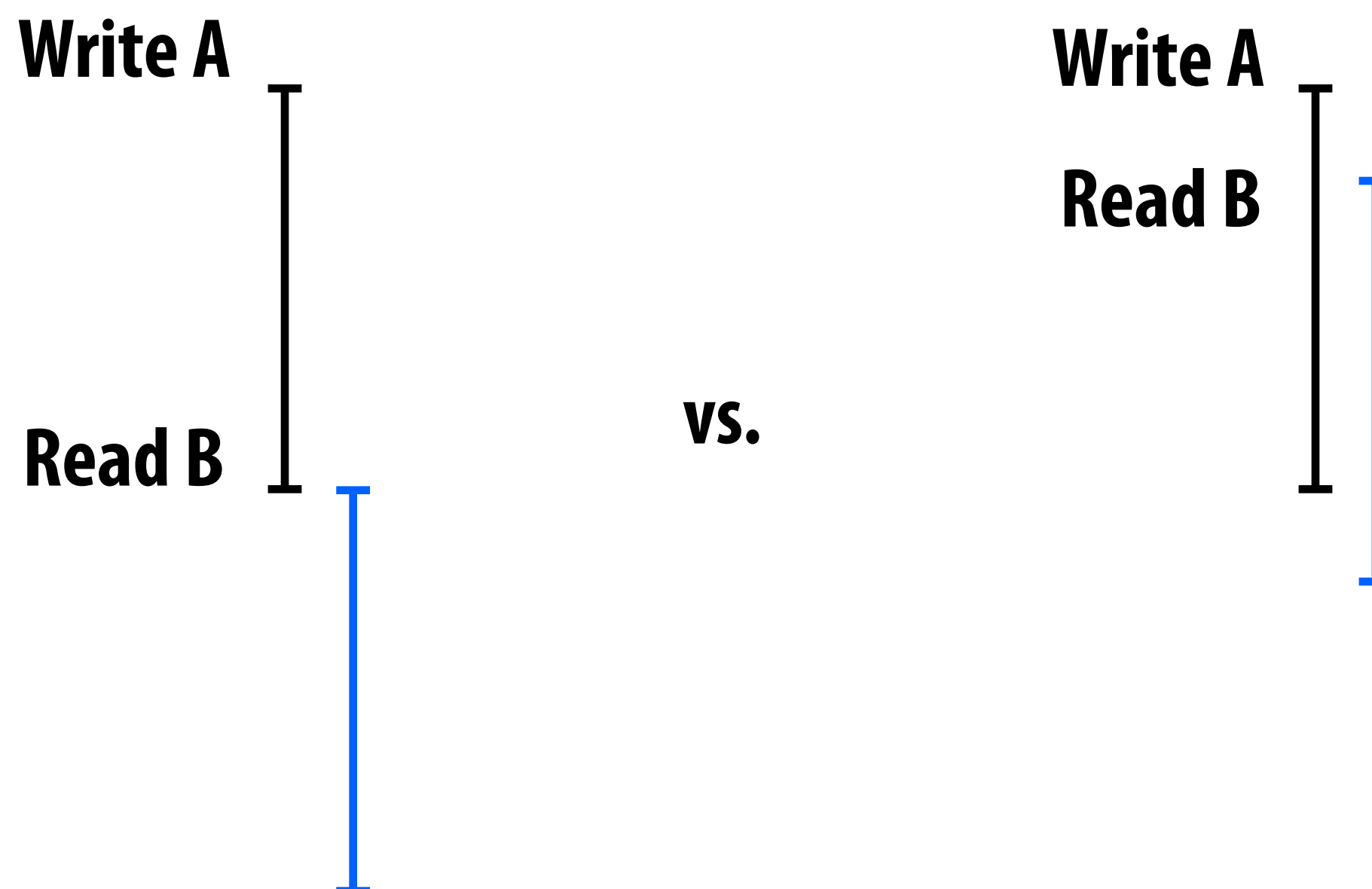**Thread 2 (on P2)**

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

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

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

Motivation for relaxed consistency: hiding latency

- ■ **Why are we interested in relaxing ordering requirements?**

  - To gain performance

  - Specifically, hiding memory latency: overlap independent memory accesses with other operations

  - Remember, memory access in a cache coherent system may entail much more work then simply reading bits from memory (finding data, sending invalidations, etc.)

Write A

Read B

vs.

Write A

Read B

Another way of thinking about relaxed ordering

Program order
(dependencies in red: required for
sequential consistency)

"Sufficient" order for correctness
(logical dependencies in red)

Thread 1 (on P1)   **Thread 2 (on P2)**

```
A = 1;

B = 1;

unlock(L);
                    lock(L);

                    x = A;

                    y = B;
```

Thread 1 (on P1)   **Thread 2 (on P2)**

```
A = 1;

B = 1;

unlock(L);
                    lock(L);

                    x = A;

                    y = B;
```

An intuitive notion of correct = execution produces the same results as a sequentially
consistent system
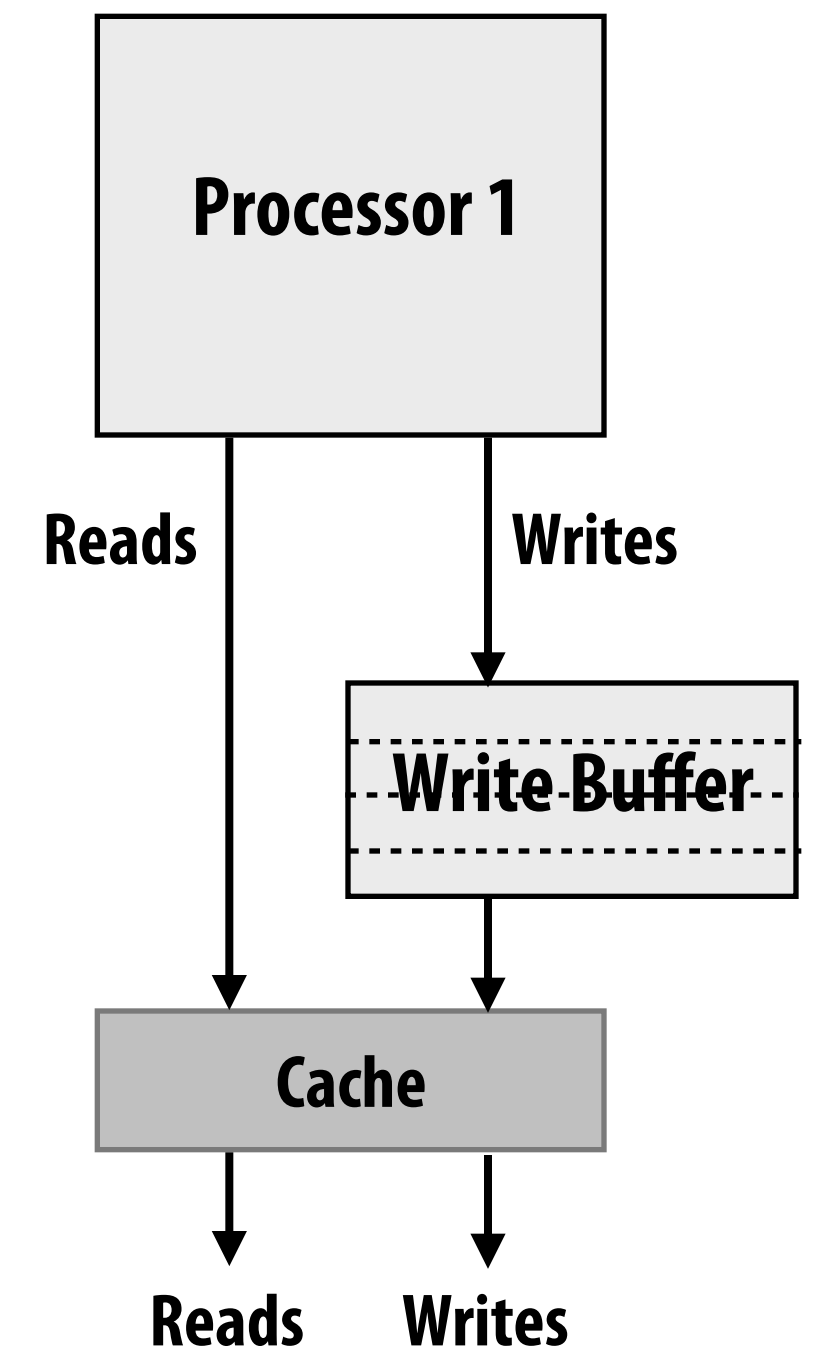
Allowing reads to move ahead of writes

- **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 A   Write A

Read B

Read B   vs.
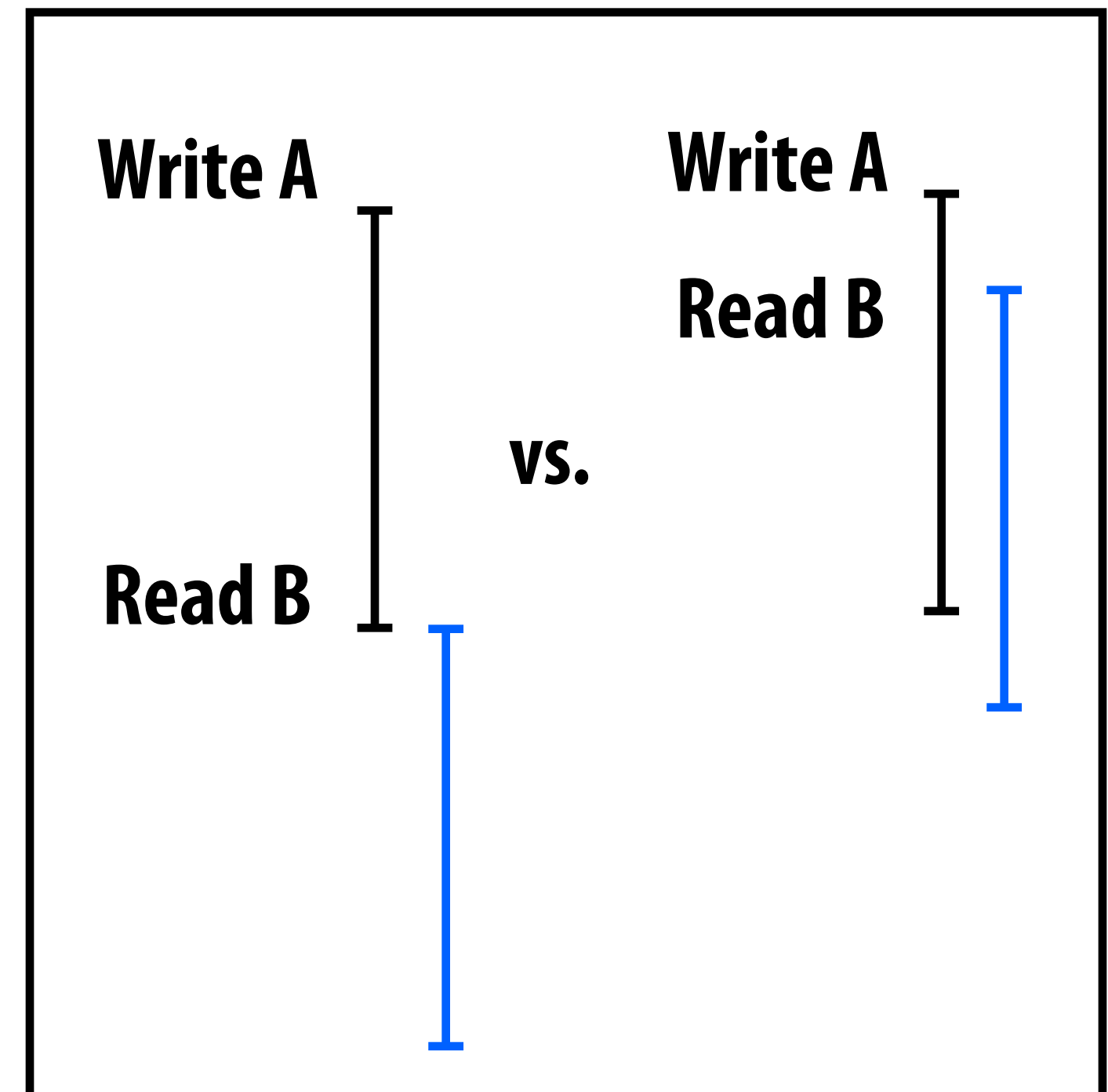
Write buffering example

- **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 (this is 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**

Processor 1

Reads   Writes

Write Buffer

Cache

Reads   Writes

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 <u>all processors</u>

- **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

①

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

②

Thread 1 (on P1)	Thread 2 (on P2)
`A = 1;`	`print B;`
`B = 1;`	`print A;`

③

Thread 1 (on P1)	Thread 2 (on P2)	Thread 3 (on P3)
`A = 1;`	`while (A == 0);`	`while (B == 0);`
	`B = 1;`	`print A;`

④

Thread 1 (on P1)	Thread 2 (on P2)
`A = 1;`	`B = 1;`
`print B;`	`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. (This is unrelated to whether there are caches 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;`

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 access 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

Conflicting data accesses

- **Two memory accesses by different processors <u>conflict</u> 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 <u>data-races</u>: 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 <u>data-race-free</u>

- **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

Relaxed consistency performance
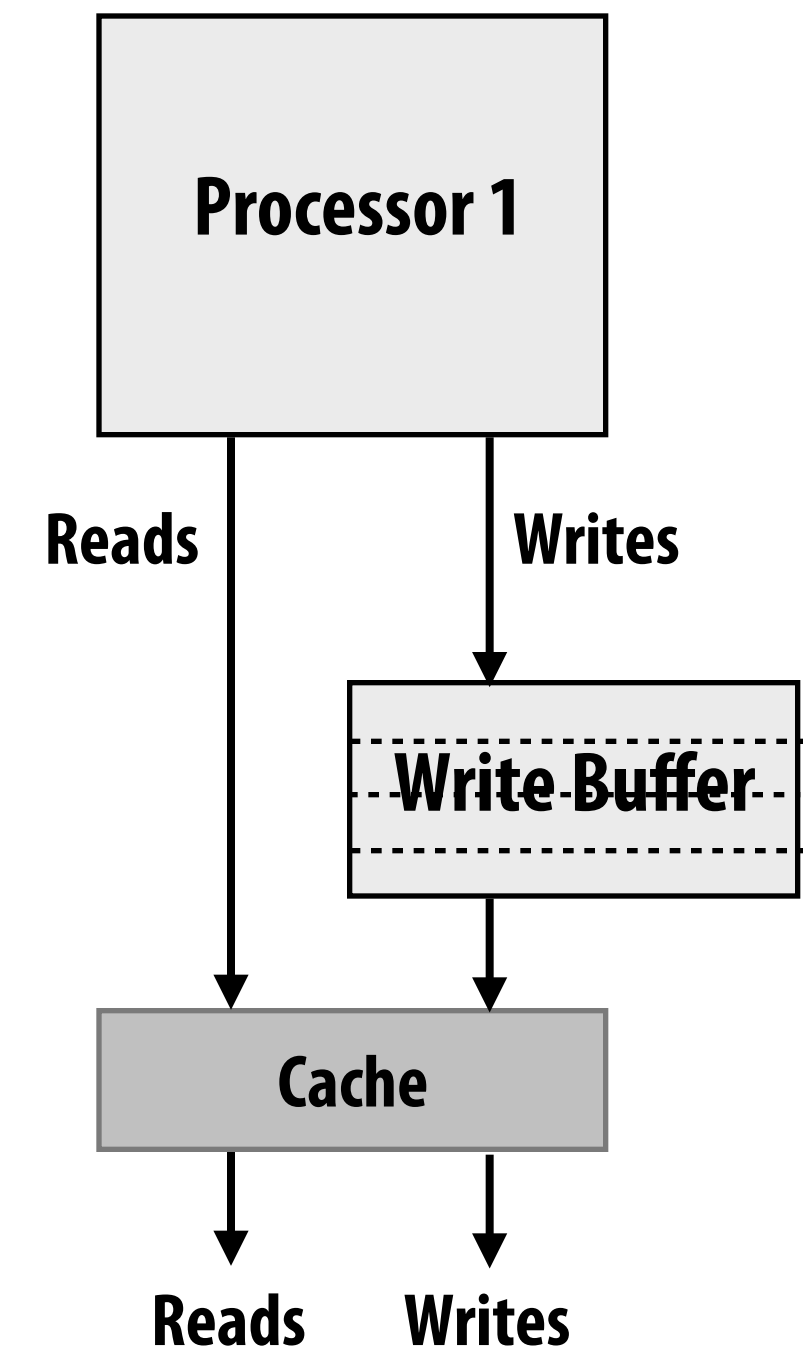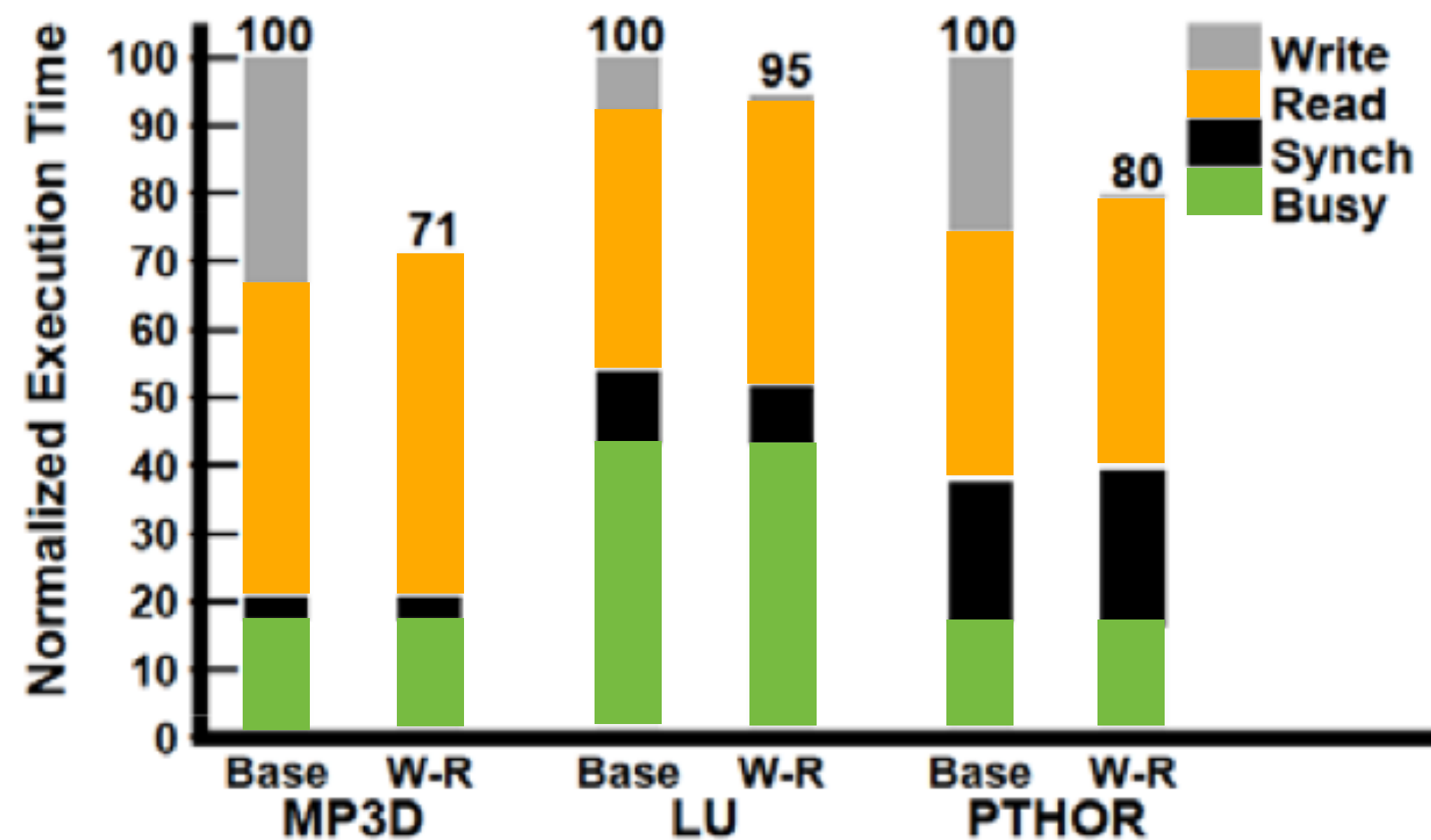
Base: Sequentially consistent execution. Processor issues one memory operation at a time, stalls until completion
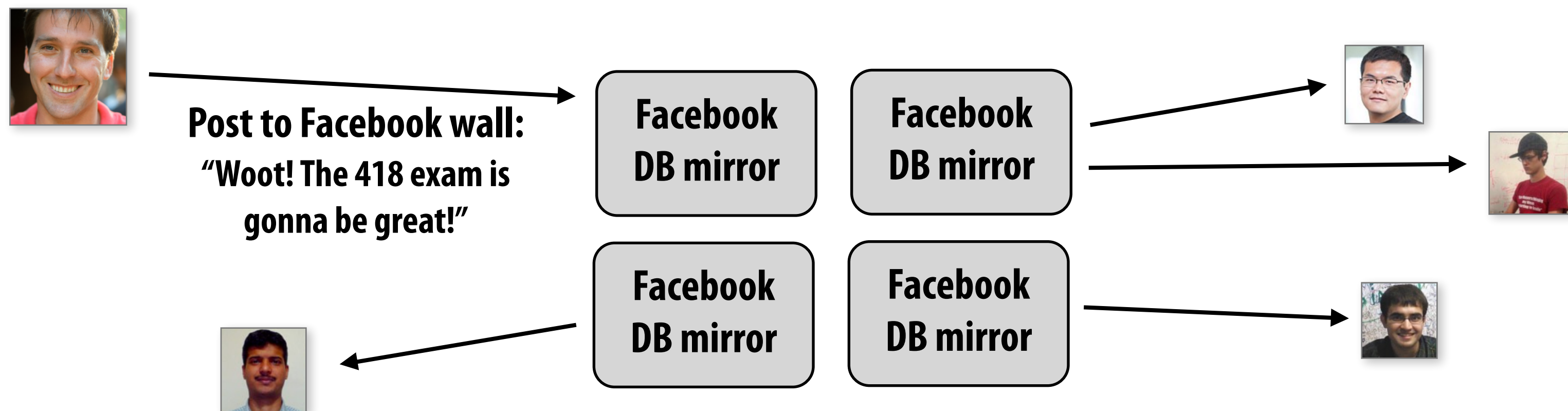
W-R: relaxed W→R ordering constraint
(write latency almost fully hidden)

Summary: relaxed consistency

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

- **One cost is software complexity: programmer or compiler must correctly insert synchronization to ensure certain specific ordering**
  - **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 pay 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 <u>eventually</u> be observed by <u>all</u> 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" of notes (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 Saturday at 3pm in Rashid Aud**
  - **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
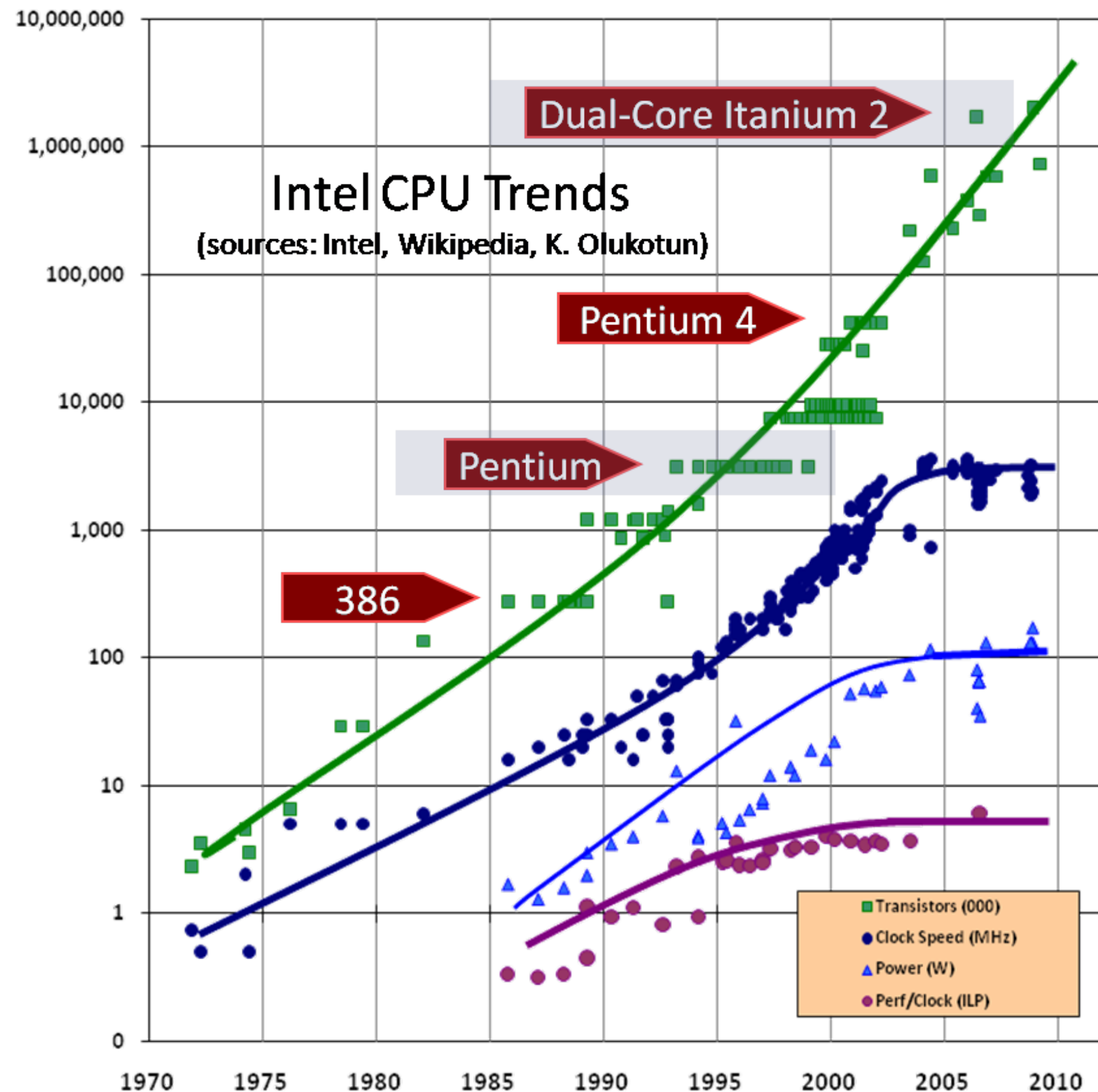
Is the new performance focus on throughput, or latency?

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 680 GPU**
  - 6 "cores" (called SMX core by NVIDIA)
  - 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

- **Blacklight Supercomputer**
  - 512 CPUs
    - Each CPU: 8 cores
      - Each core: supports 2 threads, issues 4-wide SIMD instructions (SSE instructions)

Multi-threaded, SIMD execution on GPU
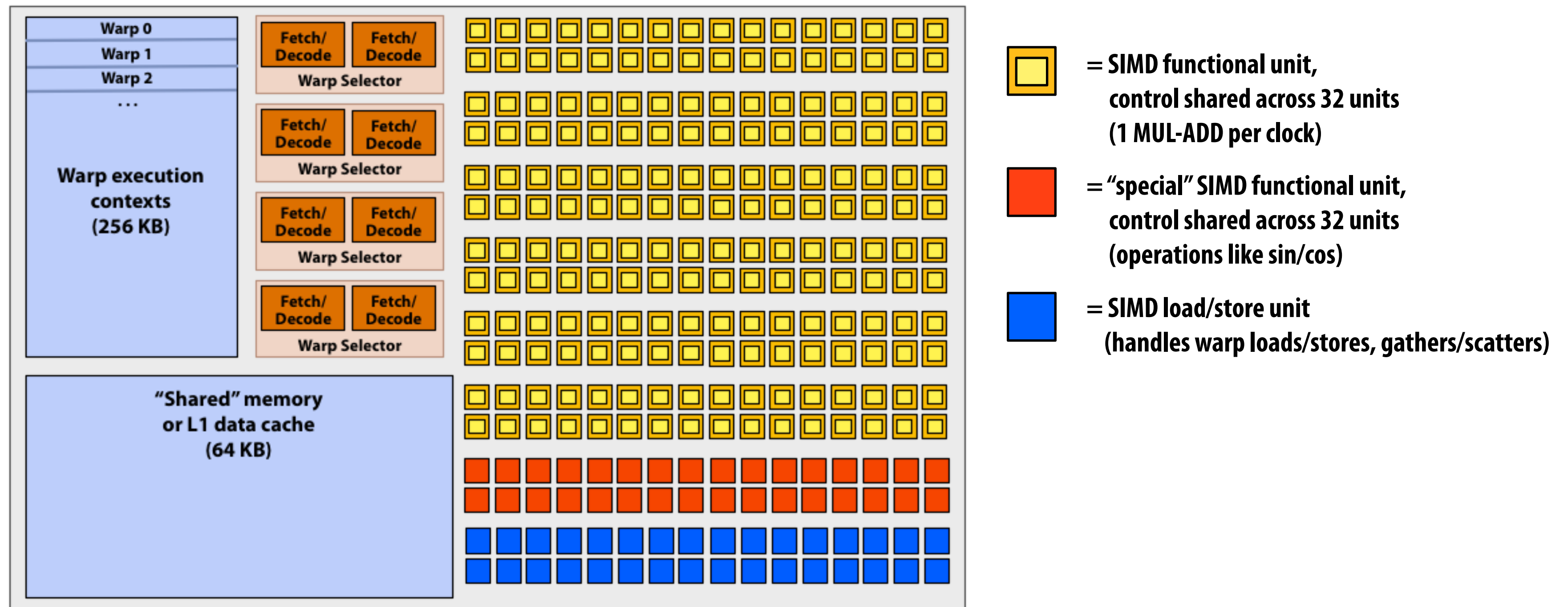
- **Describe how CUDA threads are mapped to the execution resources on this GTX 680 GPU?**
  - e.g., describe how the processor executes instructions each clock

Decomposition: assignment 1, program 3

- **You used ISPC to parallelize the Mandelbrot generation**
- **You created a bunch of tasks. How many? Why?**
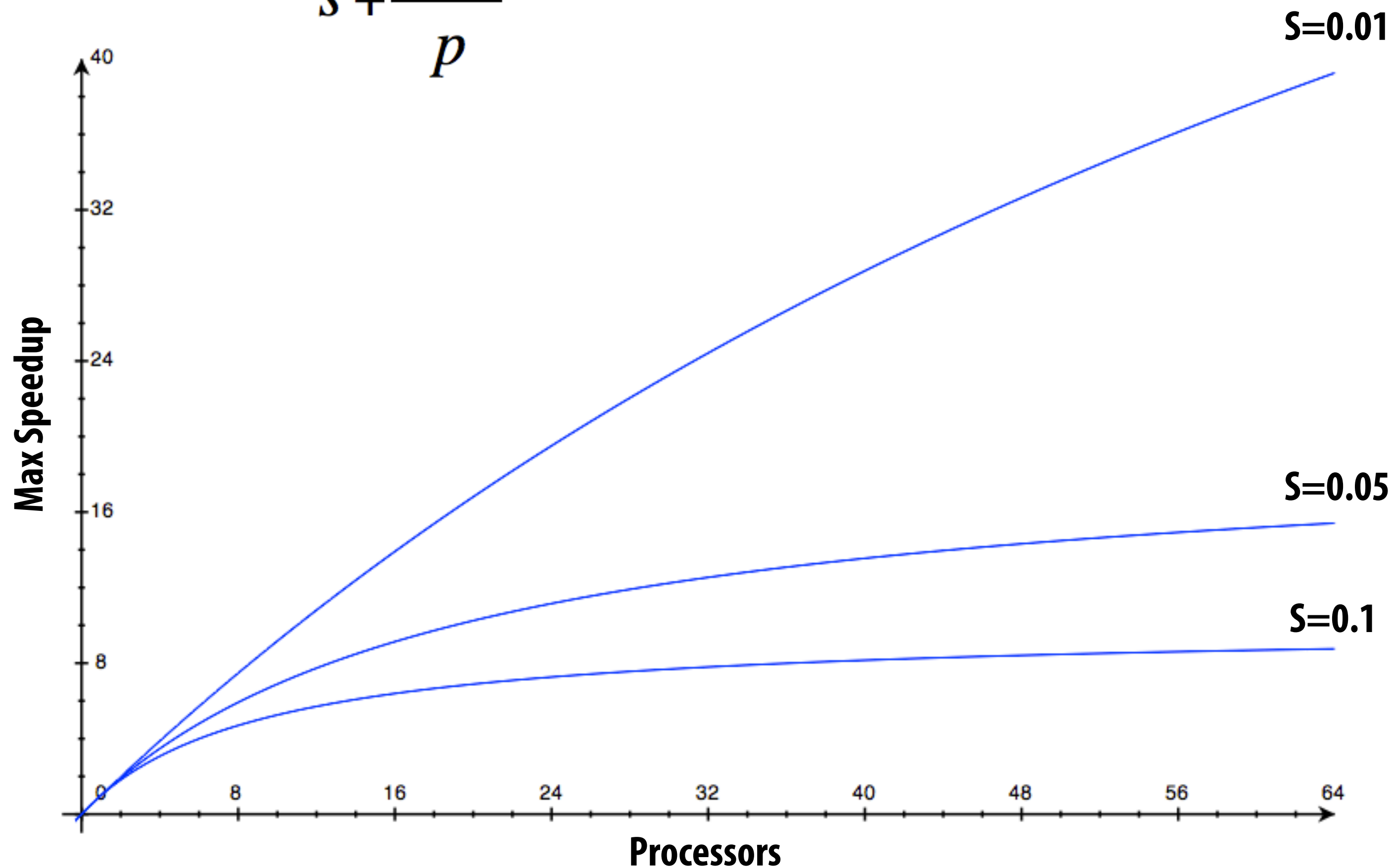
```
uniform int rowsPerTask = height / 2;

// 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
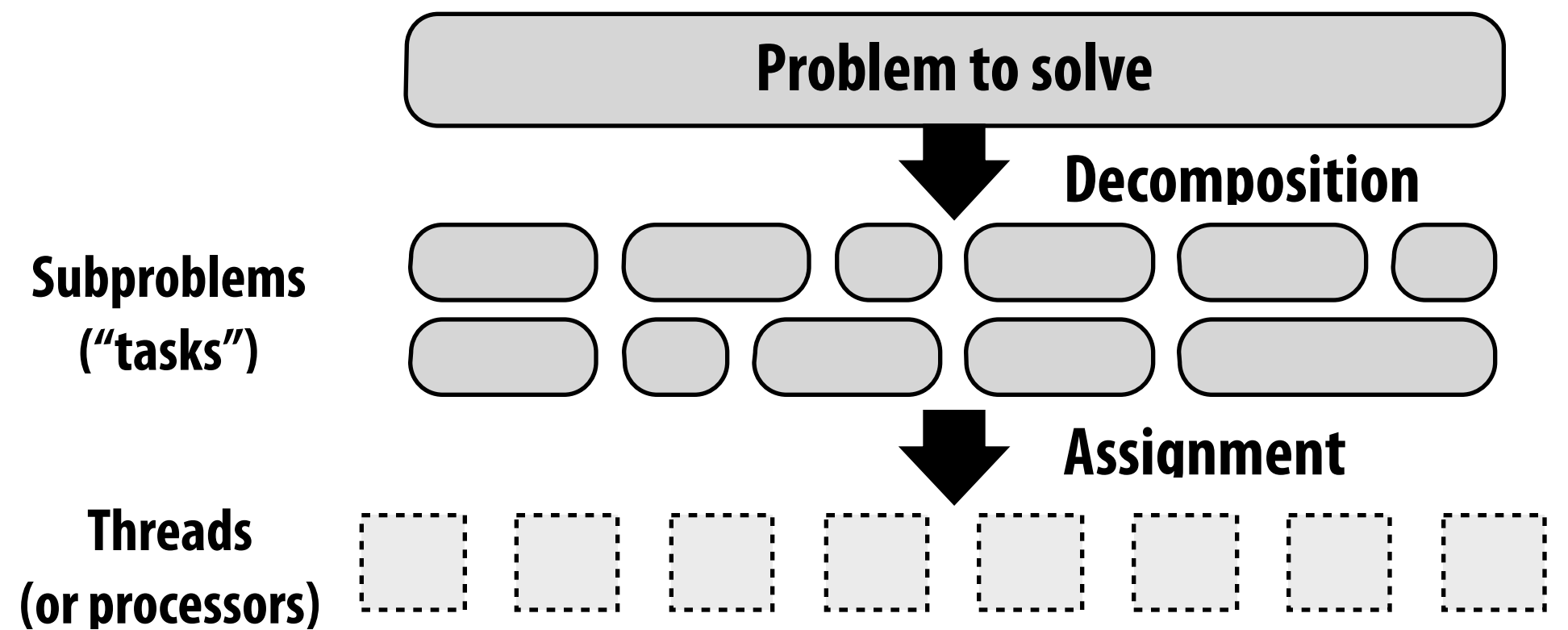
■ Max speedup on P processors given by:

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

S=0.01

S=0.05

S=0.1

Max Speedup

Processors

40

32

24

16

8

0   8   16   24   32   40   48   56   64

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 machines in GHC 3000. 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 in the right?**

Work assignment

Problem to solve

↓ **Decomposition**

Subproblems ("tasks")

↓ **Assignment**

Threads (or processors)

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
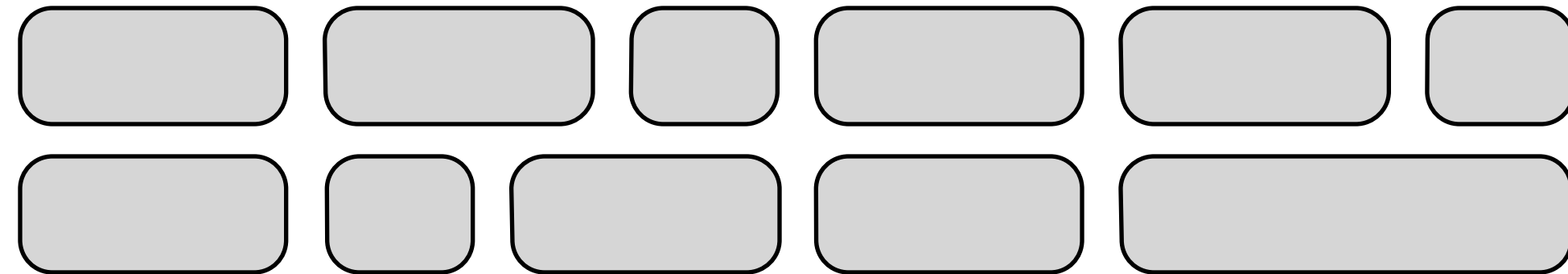
Balancing the workload

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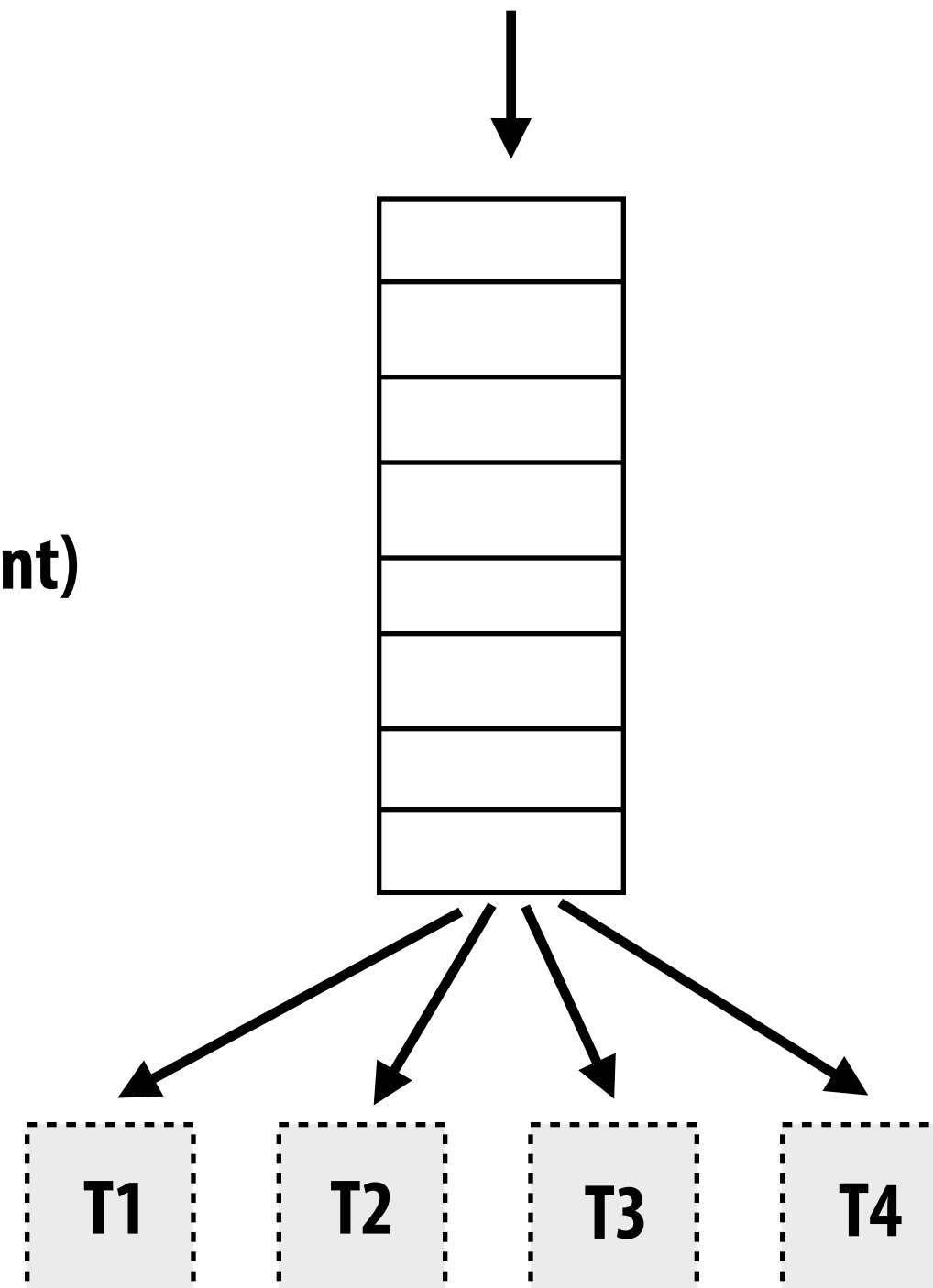**(they are computing simultaneously, and they finish their portion of the work at the same time)**

Time   P1   P2   P3   P4
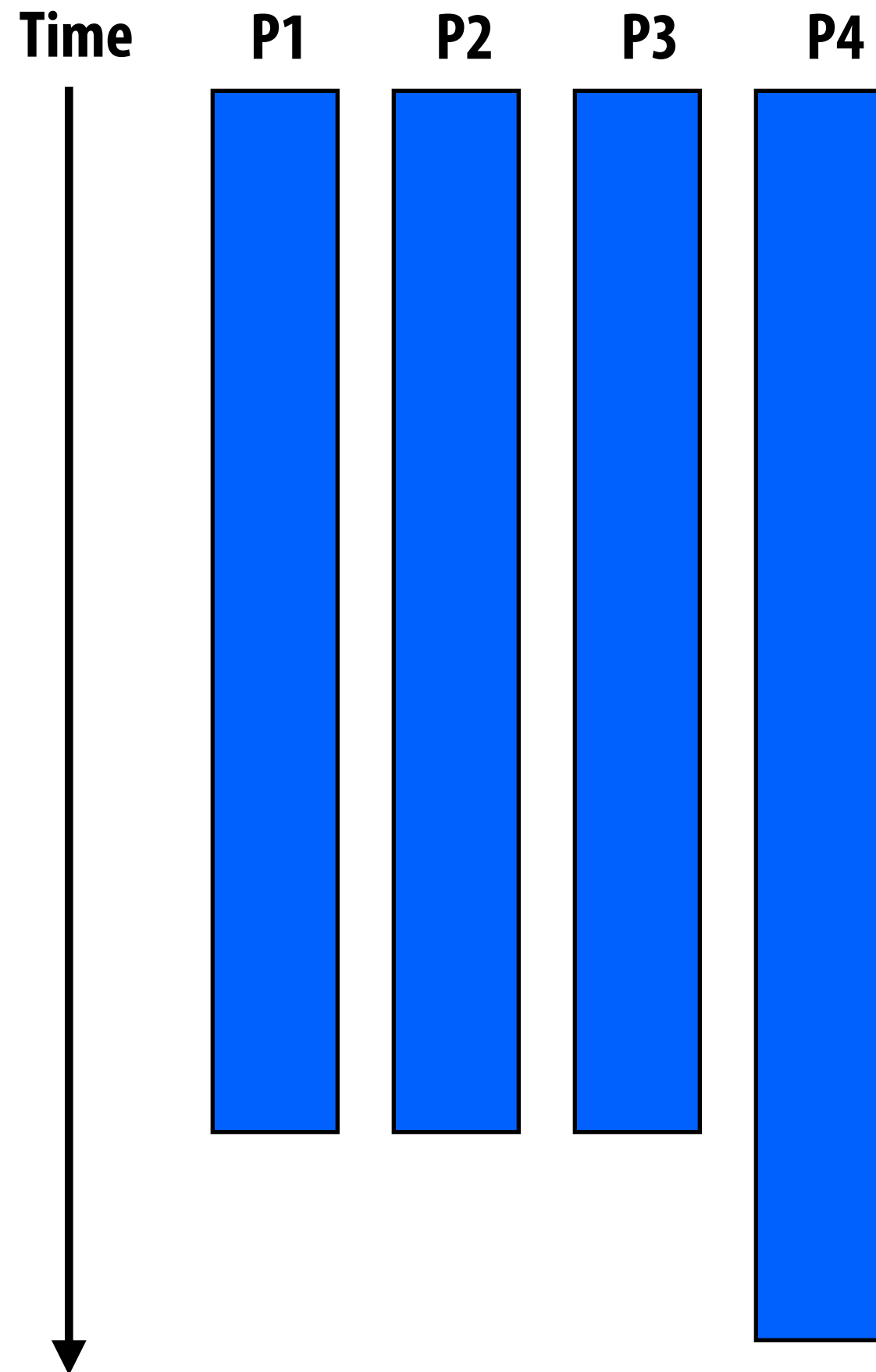
Load imbalance can significantly reduce overall speedup

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)

T1 T2 T3 T4

Worker threads:
Pull data from work queue
Push new work to queue as it's created
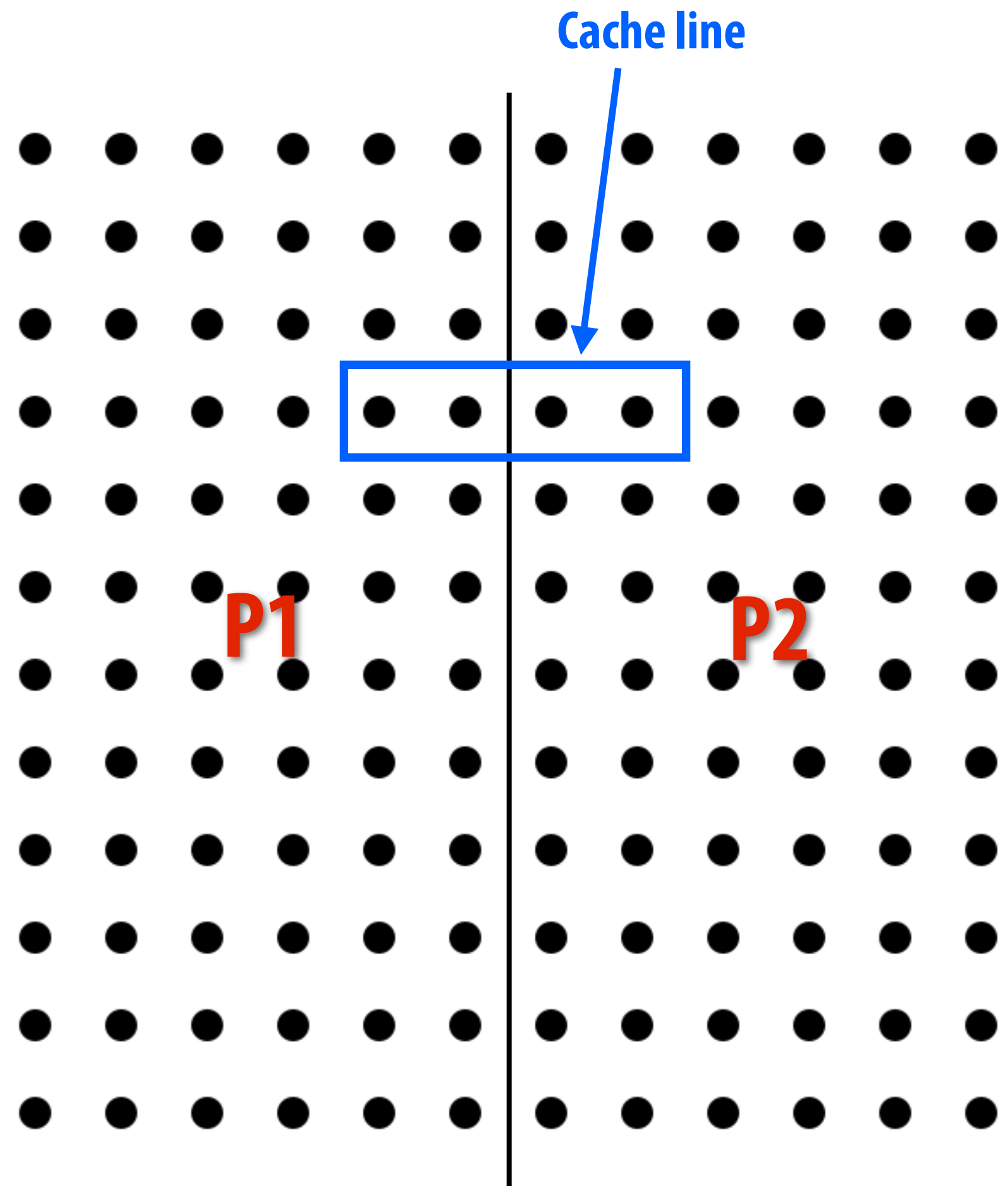
Decomposition in assignment 2

- **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

Cache line

P1   P2

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 <u>broadcasts</u> 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 <u>point-to-point</u> 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
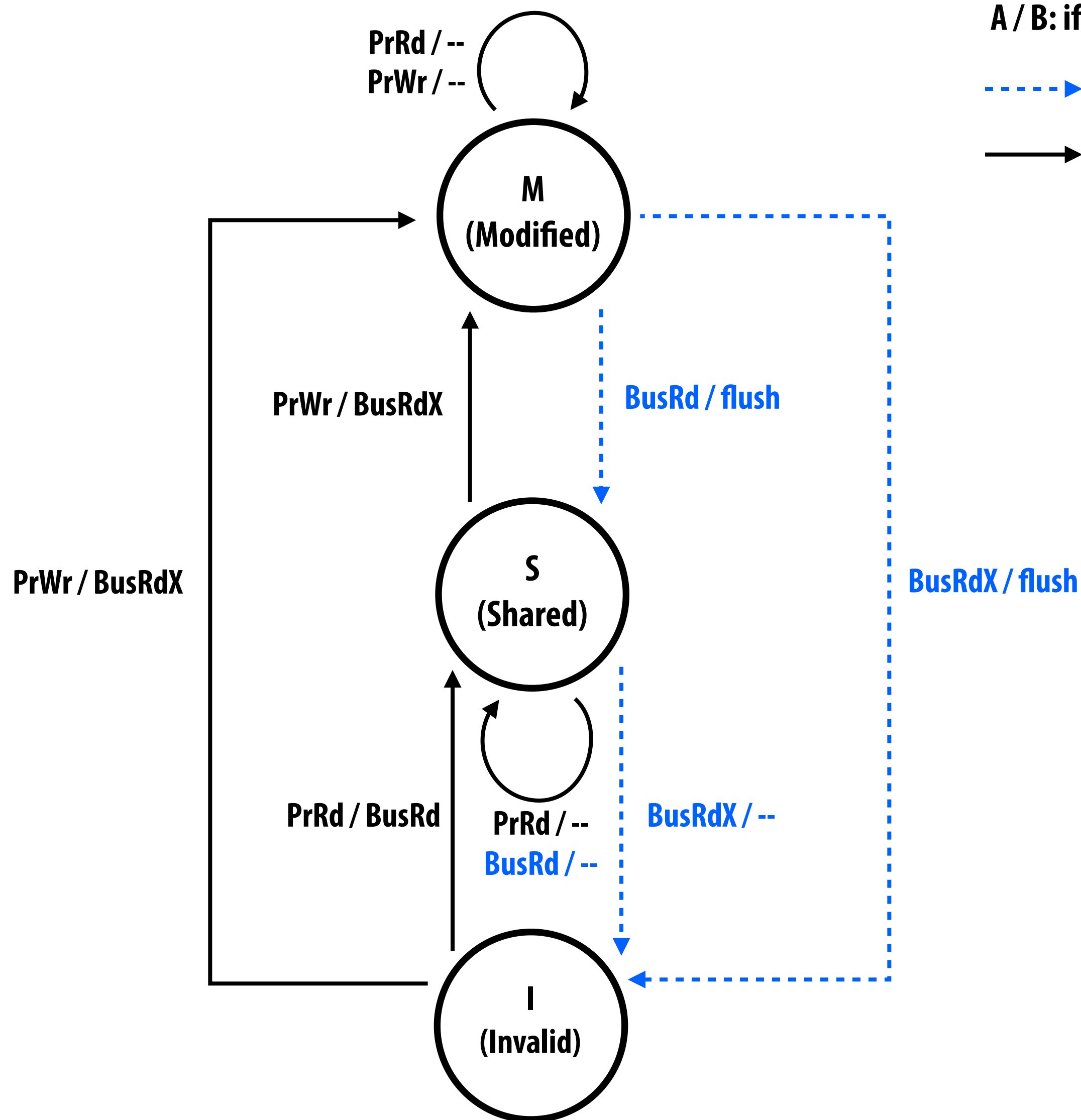
A / B: if action A is observed by cache controller, action B is taken

- - - ▶ Broadcast (bus) initiated transaction

———▶ Processor initiated transaction

PrRd / --
PrWr / --

M
(Modified)

PrWr / BusRdX

BusRd / flush

BusRdX / flush

PrWr / BusRdX

S
(Shared)

PrRd / BusRd

PrRd / --
BusRd / --

BusRdX / --

I
(Invalid)