Lecture 14:
Relaxed Memory Consistency + Exam 1 Review

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

Announcements

- Exam 1 review session
  - Sunday 6:30pm
  - GHC 4405

- Will talk more about exam later in class

Today: what you should know

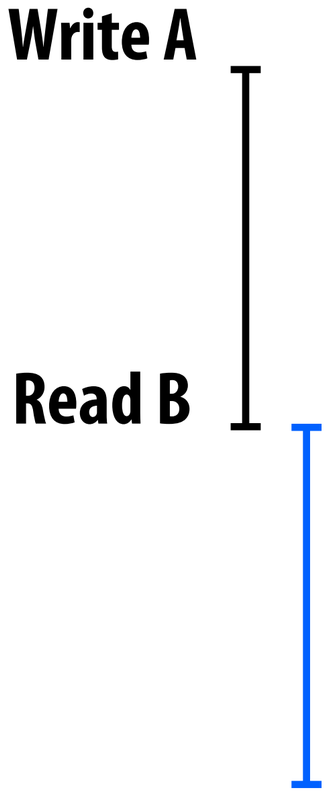
- Understand the motivation for relaxed consistency models
- Understand the implications of TSO and PC relaxed models

Relaxing memory operation ordering

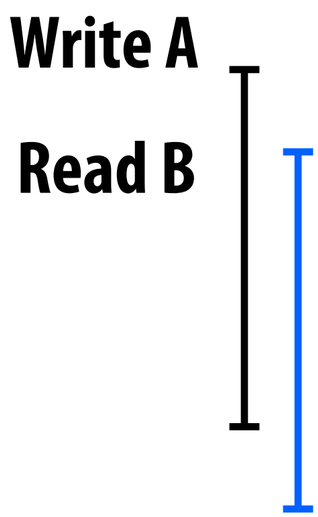
- Four types of memory operation orderings
  - W→R: write must complete before subsequent read
  - R→R: read must complete before subsequent read
  - R→W: read must complete before subsequent write
  - W→W: write must complete before subsequent write
- Sequential consistency maintains all four orderings
- Relaxed memory consistency models allow certain orderings to be violated

Motivation: hiding latency

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



vs.



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

(logical dependencies in red)

Thread 1 (on P1)

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

Thread 2 (on P2)

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

"Intuitive" notion of correct = execution produces 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
  - Processor Consistency (PC)
  - Total Store Ordering (TSO)

Allowing reads to move ahead of writes

- Total store ordering (TSO)
  - Processor P can read B before it's write to A is seen by all processors (processor can move its own reads in front of its own writes)
  - Read 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, W → W constraint still exists. Writes by the same thread are not reordered (they occur in program order)

Four example programs

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;`

Execution matches Sequential Consistency (SC)

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

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)
    - Processor supports special synchronization operations
    - Memory accesses before sync must complete before sync issues
    - Memory access after sync cannot begin until sync complete

```
reorderable reads
and writes

...

SYNC

...

reorderable reads
and writes

...

SNYC
```

Example: expressing synchronization in relaxed models

- Intel x86 ~ processor consistency (PC) model
- Provides sync instructions if software requires a specific instruction ordering not guaranteed by the consistency model
  - `lfence` ("load fence"), `sfence` ("store fence"), `mfence` ("mem fence")

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

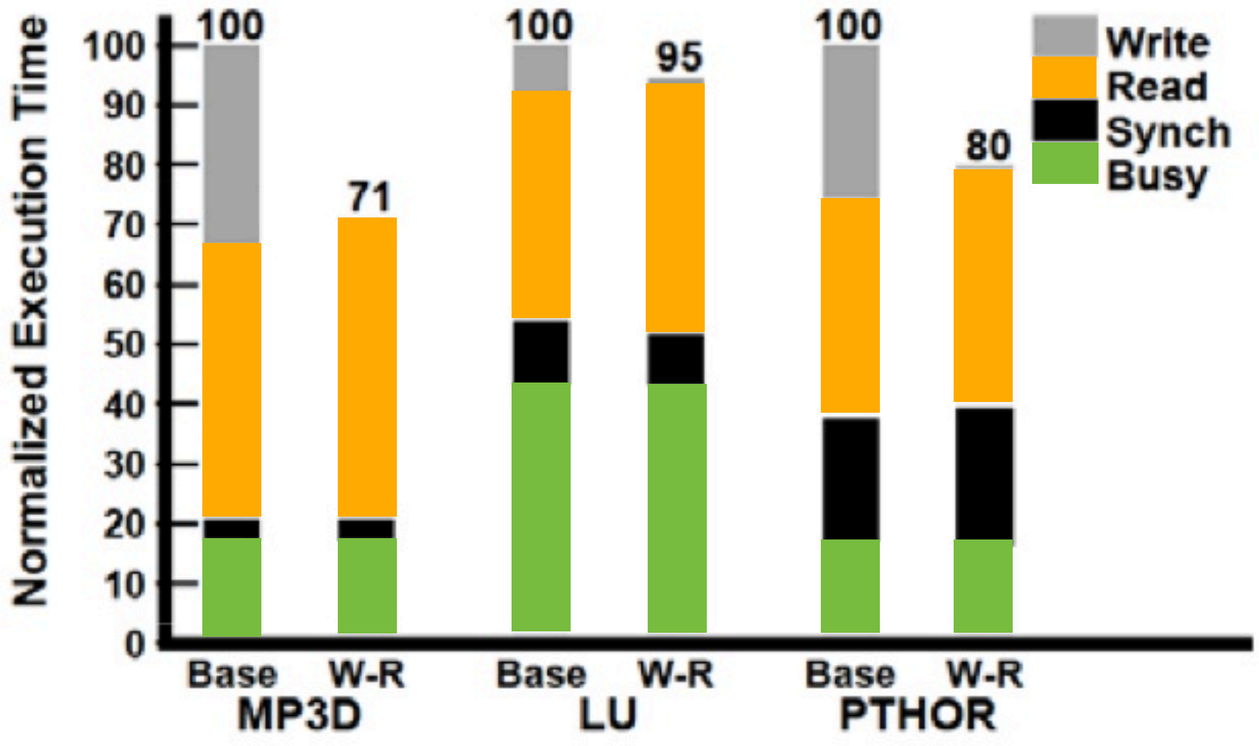
Conflicting data accesses

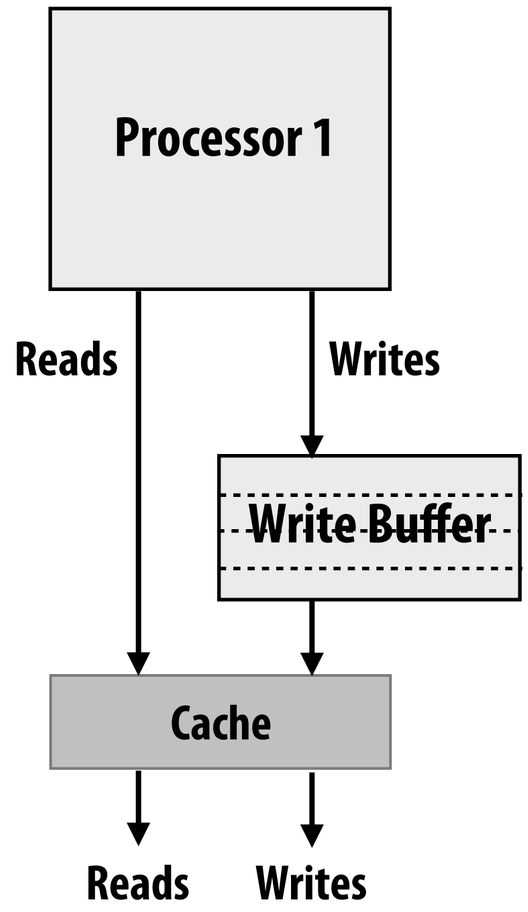
- Two memory accesses by different processors conflict if
  - they access the same memory location
  - at least one is a write

- Unsynchronized program
  - Conflicting accesses not ordered by synchronization

- Synchronized programs yield SC results on non-SC systems

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 reordering for latency hiding (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

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

Course-so-far review

Exam details

- Closed book, closed laptop
- 1 "post it" of notes (but we'll let you use both sides)
- Some resources:
  - Hennessy and Patterson. Computer Architecture: a Quantitative Approach. 5th edition
    - Chapter 4 is a great chapter on GPUs and SIMD processing
    - Chapter 5 is a good alternative discussion of cache coherence
    - 1 copy on reserve in the library
    - Scanned pdf available: /afs/cs/academic/class/15418-s12/readings
  - 1 additional copy of Culler and Singh on reserve in library

Throughput vs. latency

THROUGHPUT

The rate at which work gets done.

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

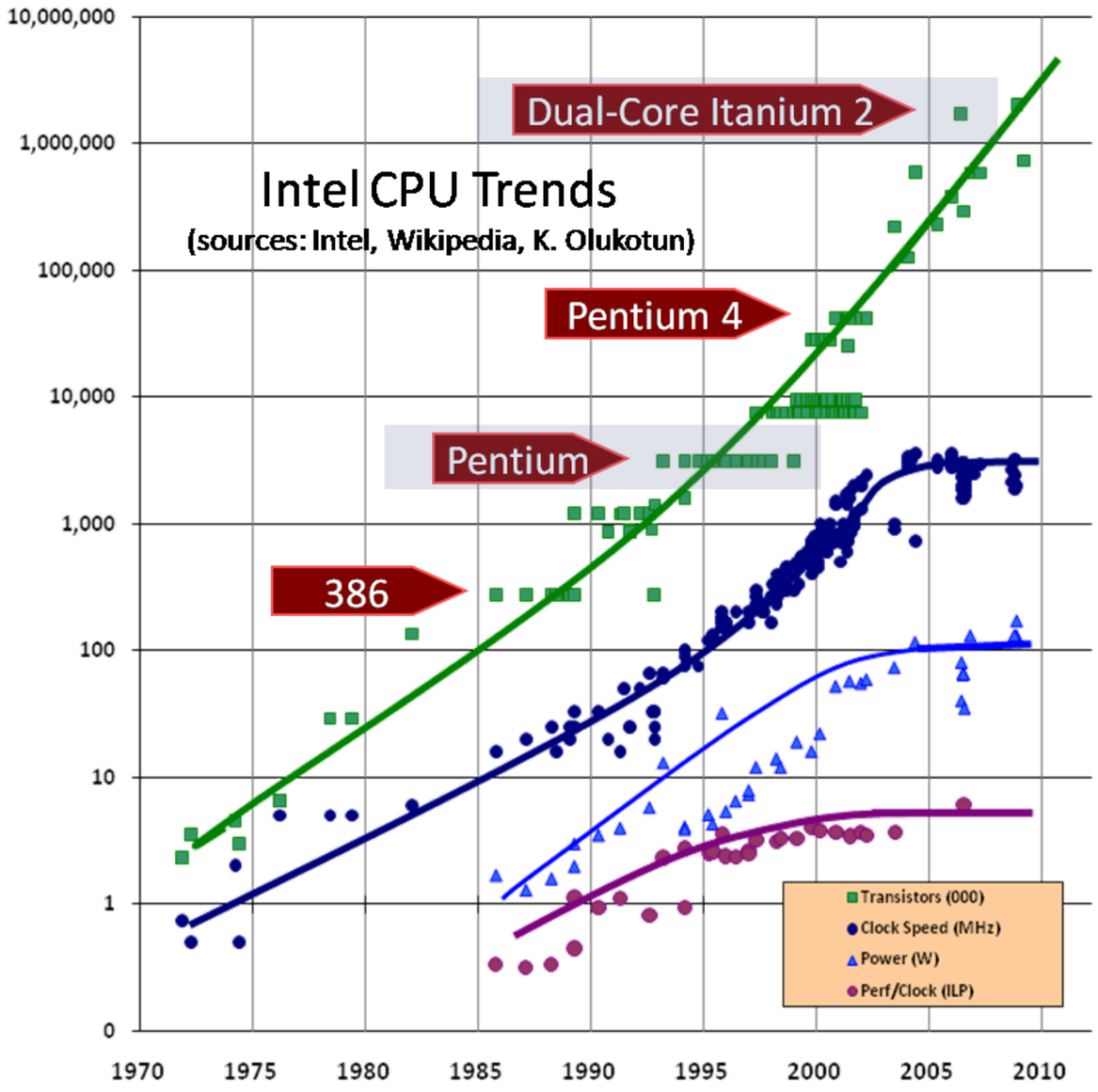
LATENCY

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?

Exploiting concurrency in modern parallel processors

	What is it? What is the benefit?
1. super-scalar 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 [largely] 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.

Exploiting concurrency in modern parallel processors

	Who is responsible for mapping?
1. super-scalar execution	Usually not a programmer responsibility: ILP automatically detected by processor hardware or by compiler (or both)
2. SIMD execution	In very 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 pragmas
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 GPU**
  - 4 cores
  - Each core:
    - Supports 2 threads (hyperthreading)
    - Can issue 8-wide SIMD instructions (AVX instructions)
    - Can perform multiple instructions per clock

- **NVIDIA GTX 480 GPU**
  - 15 cores
  - Each core:
    - Supports up to 48 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, but we haven't talked about it

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

Decomposition: asst 1, program 2

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

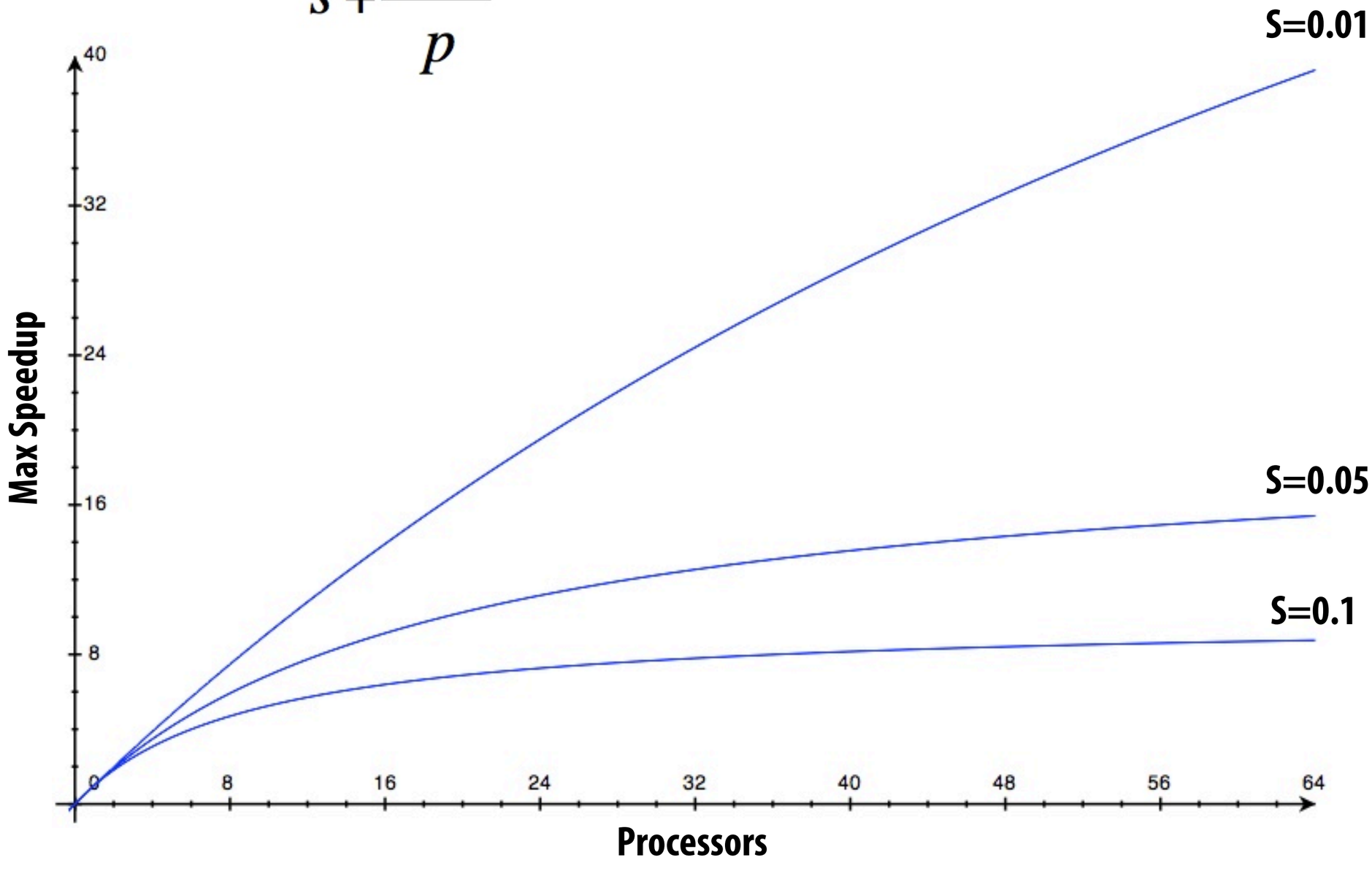
```
uniform int rowsPerTask = height / 2;

// create a bunch of tasks
launch[2] < mandelbrot_ispc_task(
                x0, y0, x1, y1,
                width, height,
                rowsPerTask,
                maxIterations,
                output) >;
```

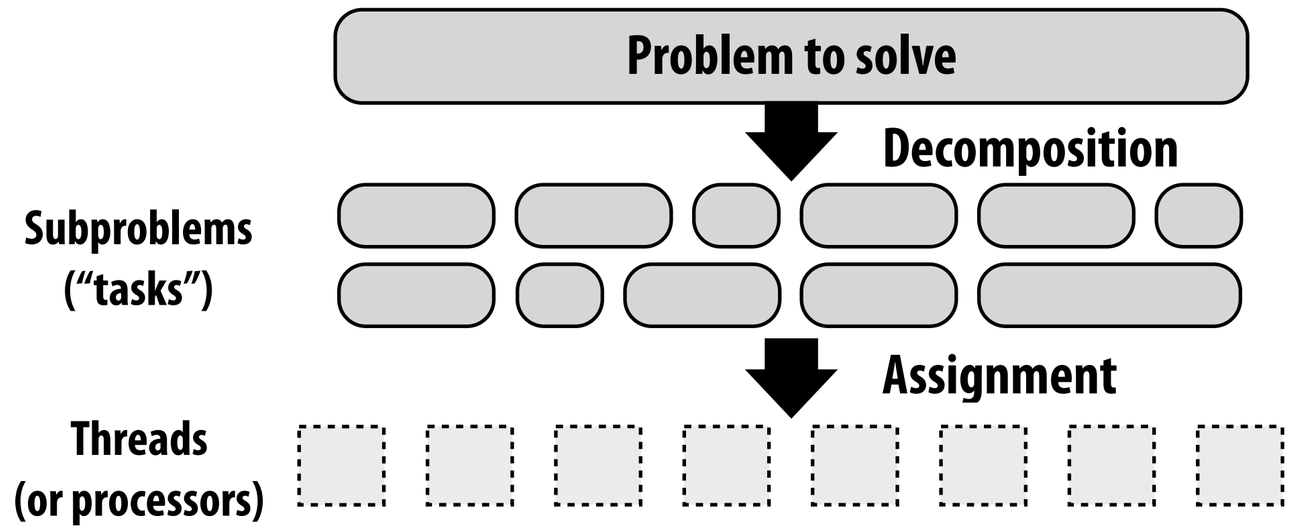
Amdahl's law

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

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



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

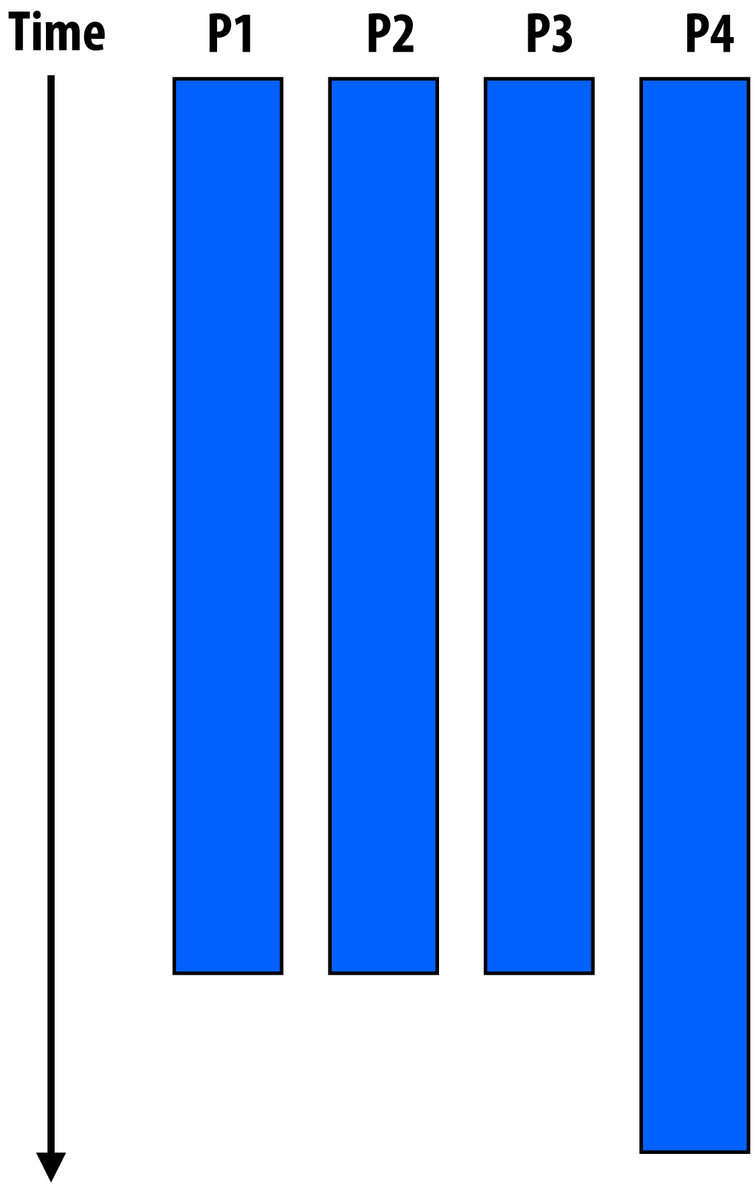
Assignment of subproblems to processors is determined as the program runs.

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)



Recall Amdahl's Law:
Only small amount of load imbalance can significantly bound maximum speedup

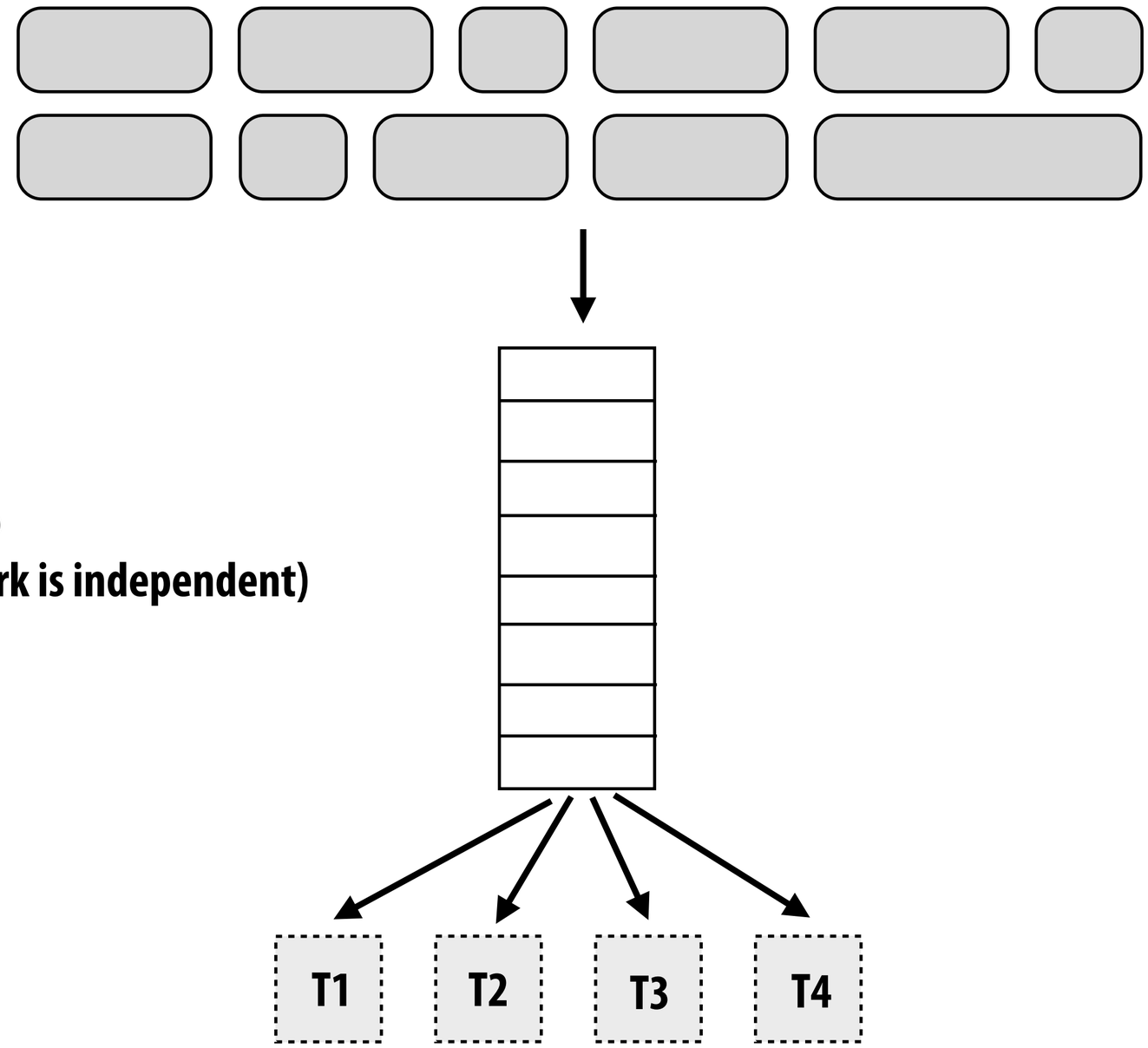
P4 does 20% more work → P4 takes 20% longer to complete

→ 20% of parallel program runtime is essentially serial execution

(clarification: work in serialized section here is about 5% of a sequential program's execution time: S=.05 in Amdahl's law eqn)

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"

Programming model abstractions

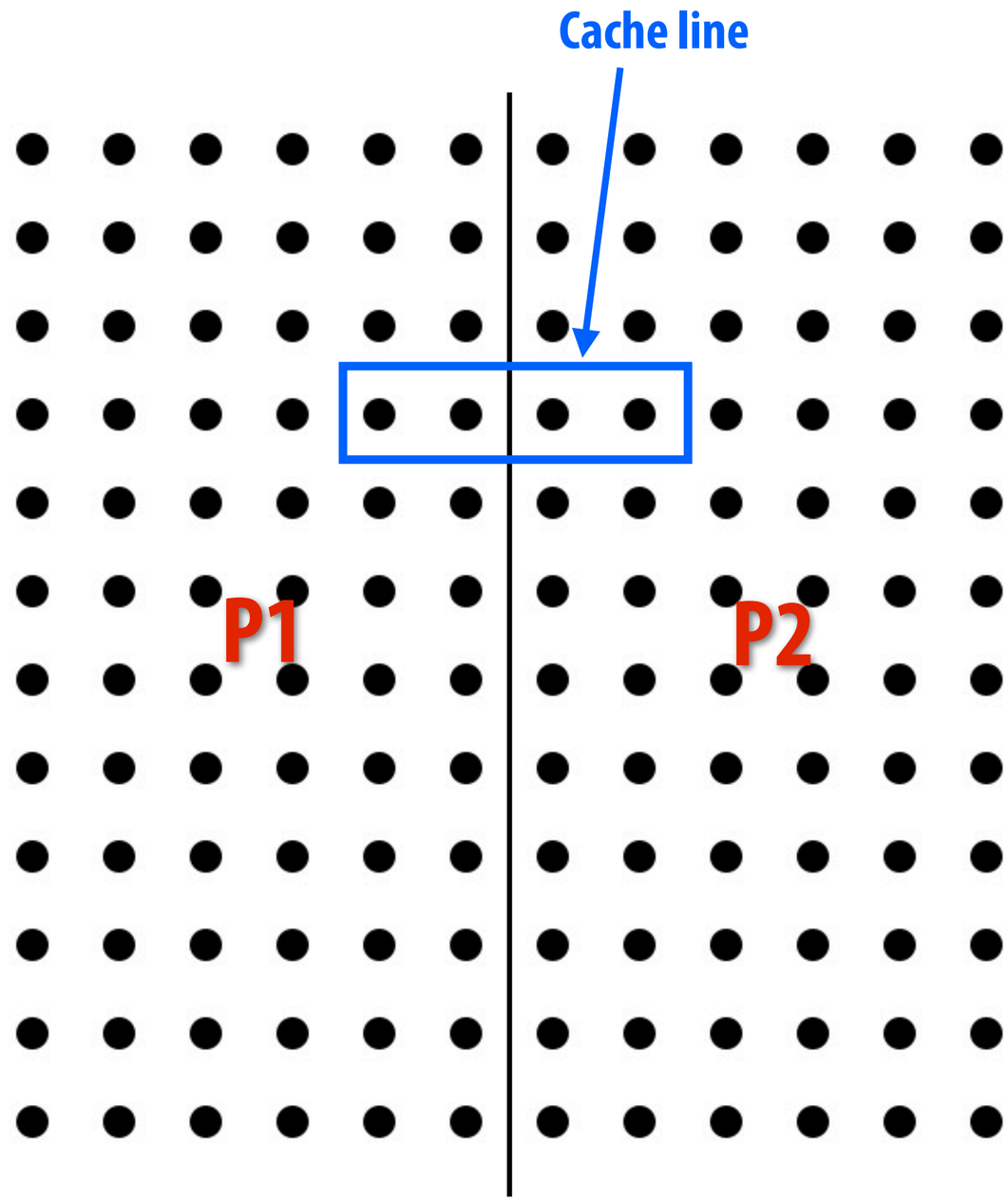
	Structure?	Communication?	Sync?
1. shared address space	Multiple processors sharing an address space.	Implicit: loads and stores to shared variables	Synchronization primitives such as locks and barriers
2. message passing	Multiple processors, each with own memory address space.	Explicit: send and receive 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.

Artifactual vs. inherent communication

INHERENT COMMUNICATION

ARTIFACTUAL COMMUNICATION

FALSE SHARING



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

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

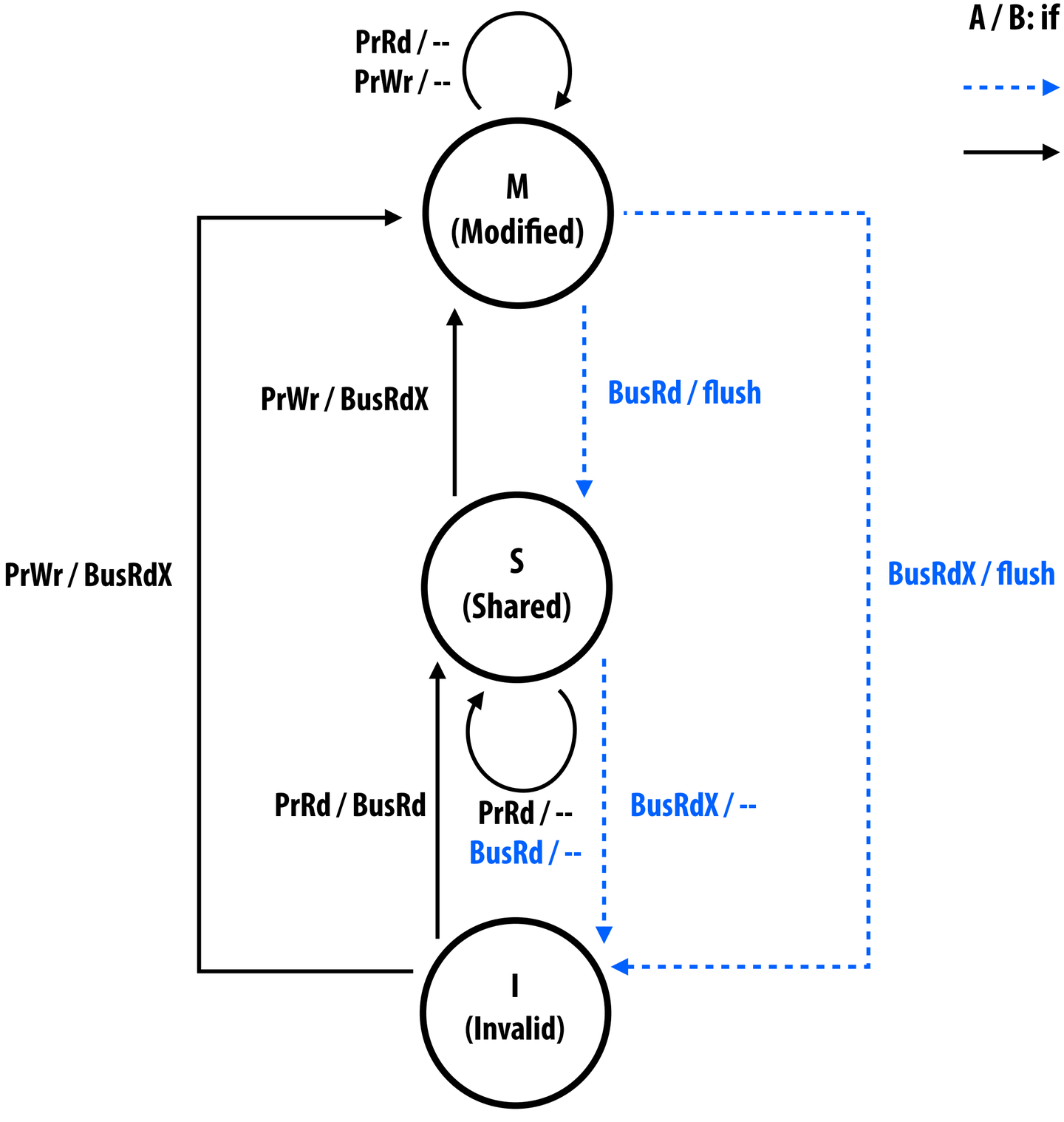
DIRECTORIES

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

CPU vs. GPU

- What would you say is the most notable architectural difference you observe when programming the CPU vs. the GPU?