Course so far review

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

Exam details

- Closed book, closed notes
- A4 paper
- Covers all lecture material through Lecture on Directory-based cache coherence
- Must use either blue or black pen (no pencils or other pen colors)
- Typical question formats:
  - Short answer
  - Multiple choice with explanations

Throughput vs. latency

THROUGHPUT

The rate at which work gets done.

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

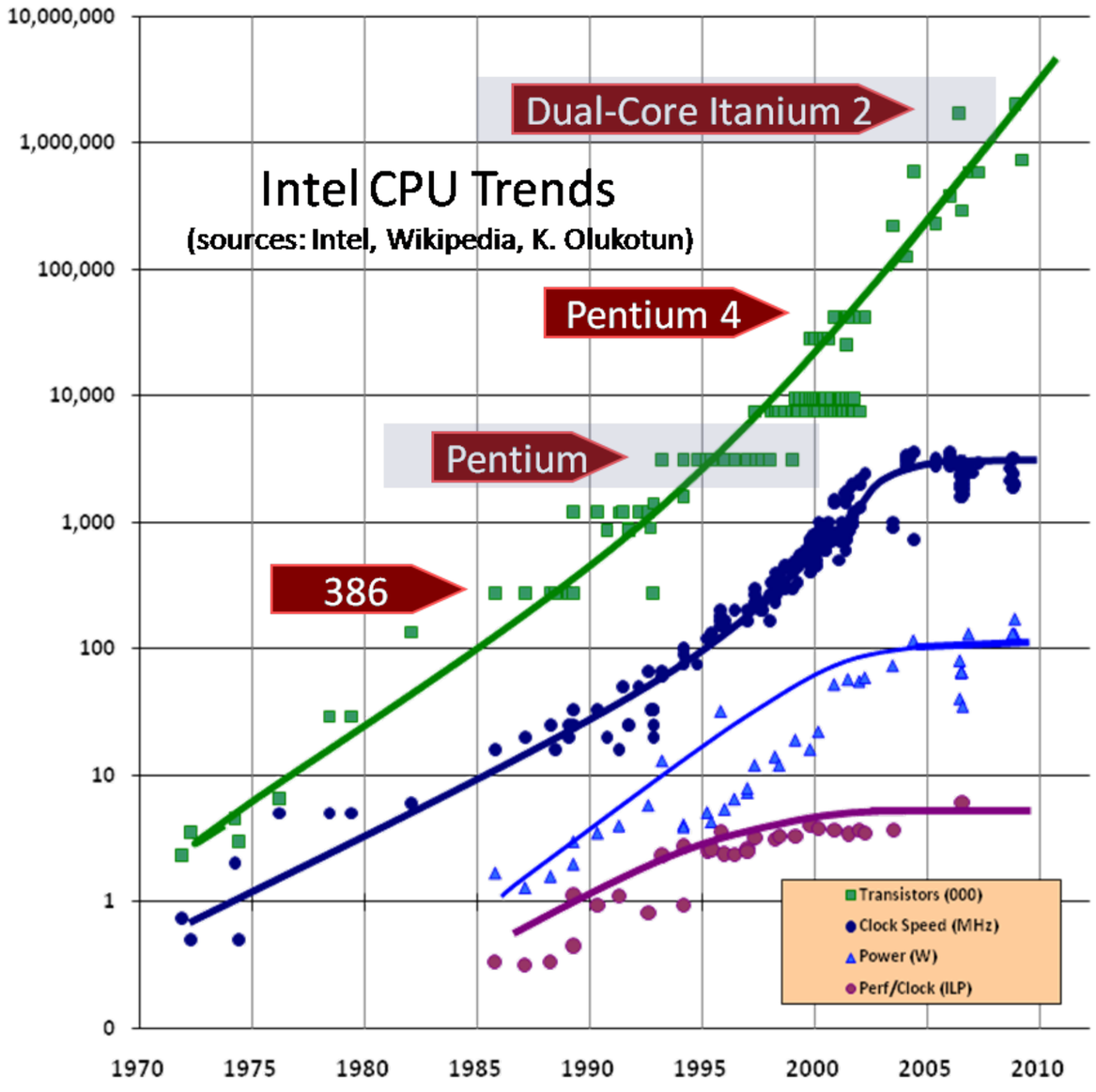
LATENCY

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.**

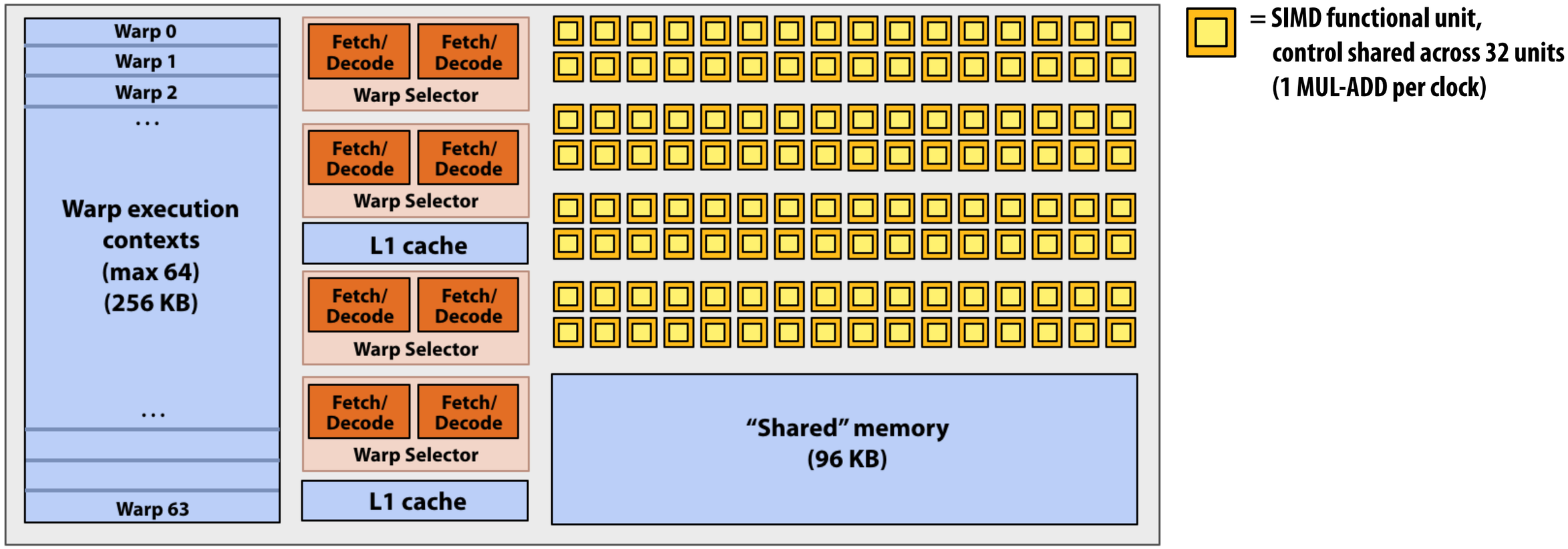
Frequently discussed processor examples

- **Intel Core i9 CPU**
  - 8 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 980 GPU**
  - 16 "cores" (called SMM 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

- **Intel Xeon Phi**
  - 61 cores
  - Each core: supports 4 threads, issues 16-wide SIMD instructions

Multi-threaded, SIMD execution on GPU



- Describe how CUDA threads are mapped to the execution resources on this GTX 980 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);
```

sin(x) in ISPC

"Interleaved" assignment of array elements to program instances

C++ code: `main.cpp`

```
#include "sinx_ispc.h"

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```

ISPC code: `sinx.ispc`

```
export void sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    // assumes N % programCount = 0
    for (uniform int i=0; i<N; i+=programCount)
    {
        int idx = i + programIndex;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6;  // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

ISPC Keywords:

`programCount`: number of simultaneously executing instances in the gang (uniform value)

`programIndex`: id of the current instance in the gang. (a non-uniform value: "varying")

`uniform`: A type modifier. All instances have the same value for this variable. Its use is purely an optimization. Not needed for correctness.

sin(x) in ISPC: version 2

"Blocked" assignment of elements to instances

C++ code: `main.cpp`

```
#include "sinx_ispc.h"

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```

ISPC code: `sinx.ispc`

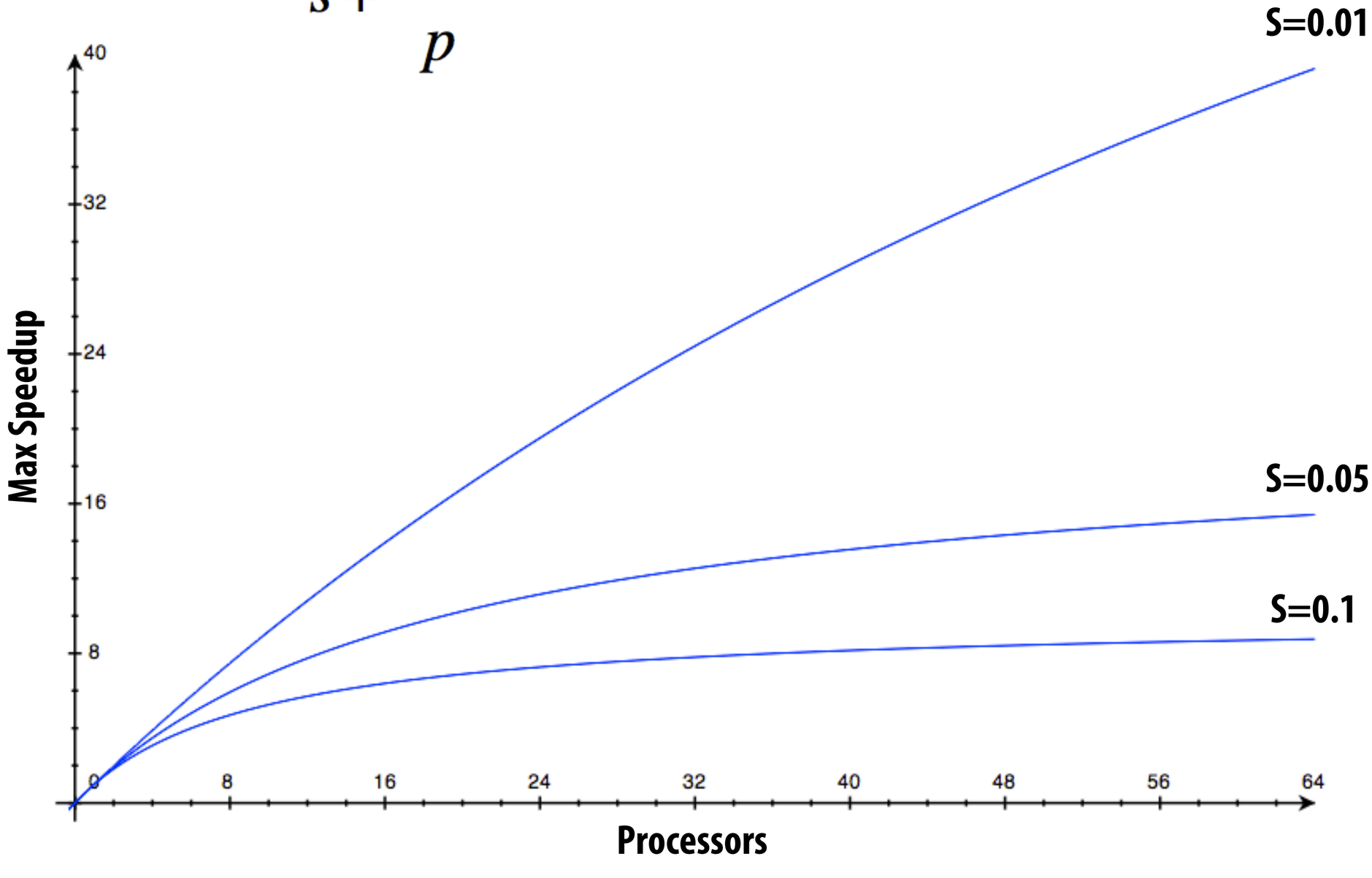
```
export void sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    // assume N % programCount = 0
    uniform int count = N / programCount;
    int start = programIndex * count;
    for (uniform int i=0; i<count; i++)
    {
        int idx = start + i;
        float value = x[idx];
        float numer = x[idx] * x[idx] * x[idx];
        uniform int denom = 6;  // 3!
        uniform int sign = -1;

        for (uniform int j=1; j<=terms; j++)
        {
            value += sign * numer / denom
            numer *= x[idx] * x[idx];
            denom *= (j+3) * (j+4);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

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

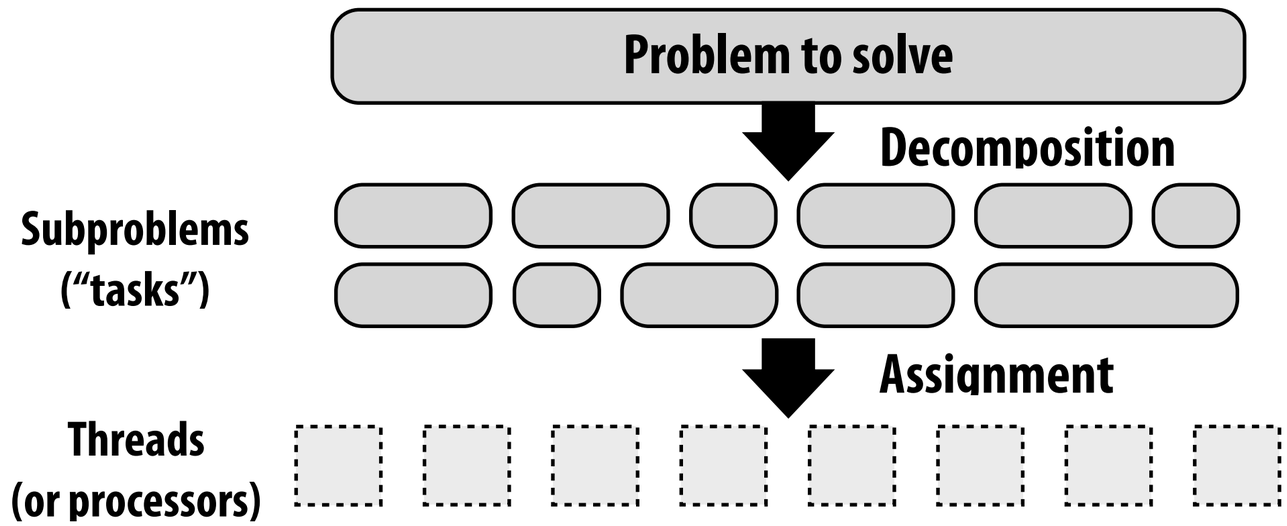
- Your boss gives your team a piece of code for which 25% of the operations are inherently serial and instructs you to parallelize the application on a six-core machines in GHC 3000. He expects you to achieve 5x speedup on this application.

- 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 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

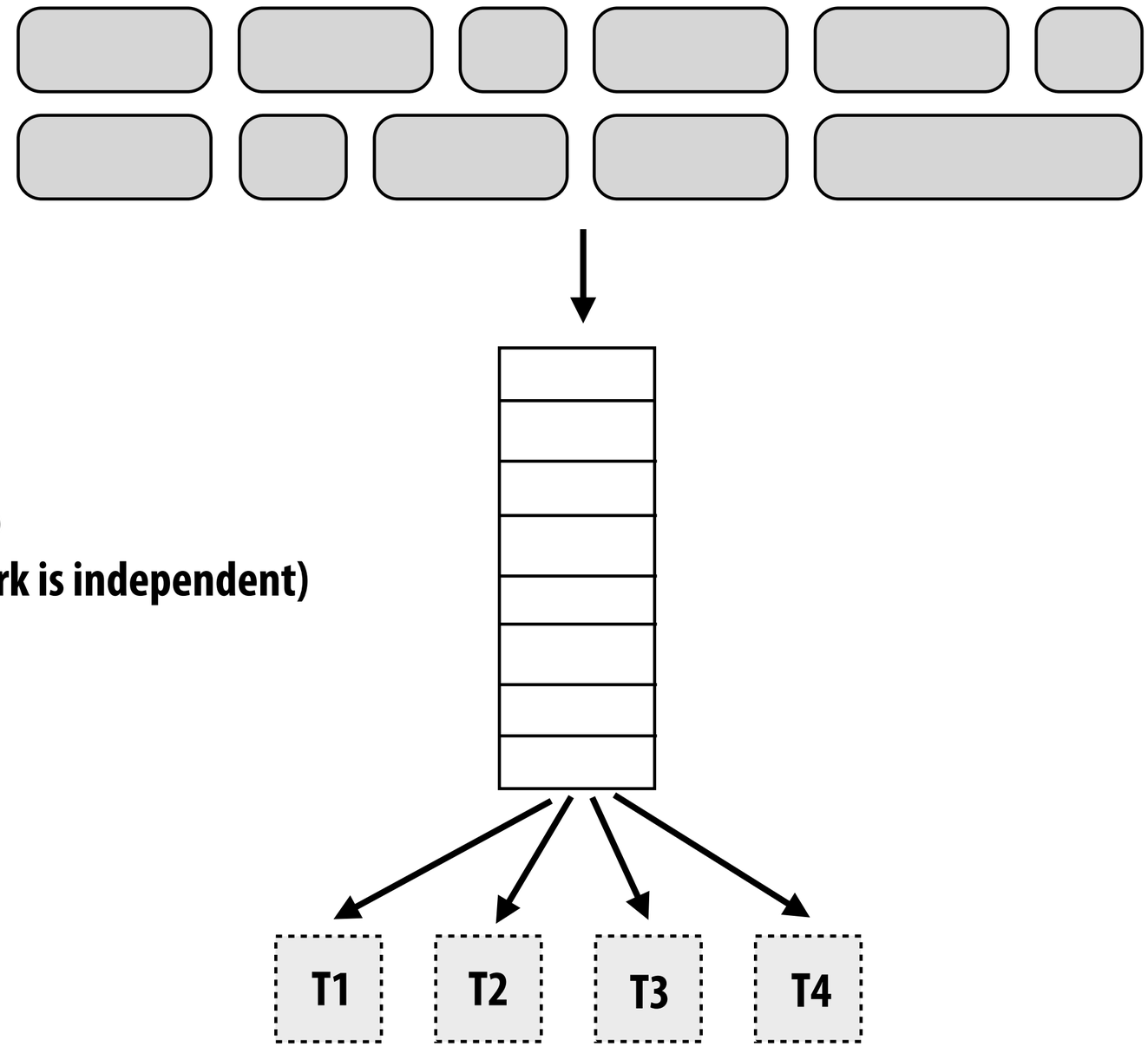
Balancing the workload

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)

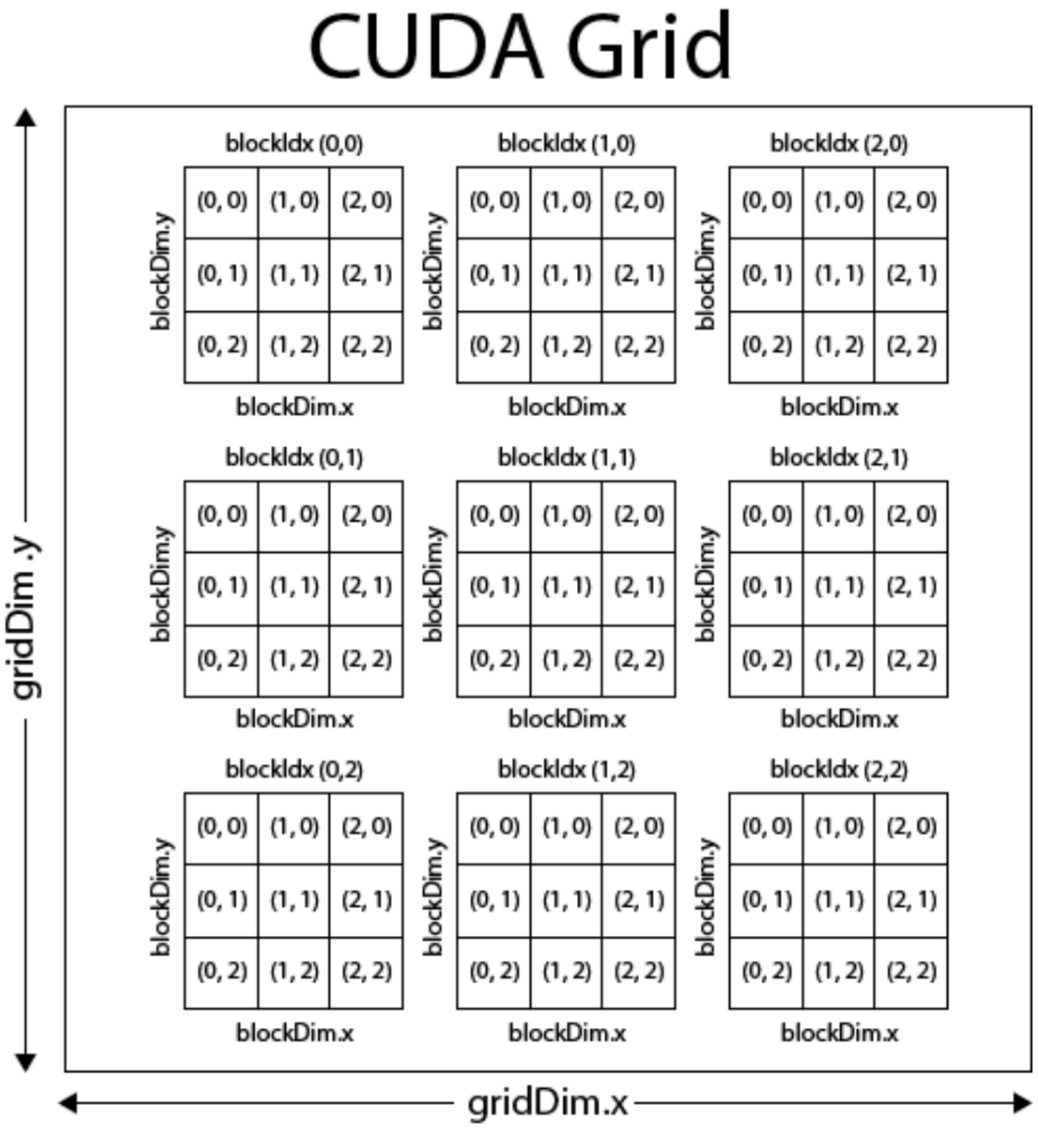
Worker threads:
Pull data from work queue
Push new work to queue as it's created

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"

Grid, Block, and Thread

- gridDim: The dimensions of the grid
- blockIdx: The block index within the grid
- blockDim: The dimensions of the block
- threadIdx: The thread index within the block



Basic CUDA syntax

Regular application thread running on CPU (the "host")

"Host" code : serial execution
Running as part of normal C/C++ application on CPU

Bulk launch of many CUDA threads
"launch a grid of CUDA thread blocks"
Call returns when all threads have terminated

```
const int Nx = 12;
const int Ny = 6;

dim3 threadsPerBlock(4, 3, 1);
dim3 numBlocks(Nx/threadsPerBlock.x,
               Ny/threadsPerBlock.y, 1);

// assume A, B, C are allocated Nx x Ny float arrays

// this call will trigger execution of 72 CUDA threads:
// 6 thread blocks of 12 threads each
matrixAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

SPMD execution of device kernel function:

CUDA kernel definition

"CUDA device" code: kernel function (`__global__` denotes a CUDA kernel function) runs on GPU

Each thread computes its overall grid thread id from its position in its block (`threadIdx`) and its block's position in the grid (`blockIdx`)

```
// kernel definition
__global__ void matrixAdd(float A[Ny][Nx],
                          float B[Ny][Nx],
                          float C[Ny][Nx])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    C[j][i] = A[j][i] + B[j][i];
}
```

CUDA synchronization constructs

- `__syncthreads()`
  - Barrier: wait for all threads in the block to arrive at this point

- Atomic operations
  - e.g., `float atomicAdd(float* addr, float amount)`
  - Atomic operations on both global memory and shared memory variables

- Host/device synchronization
  - Implicit barrier across all threads at return of kernel

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

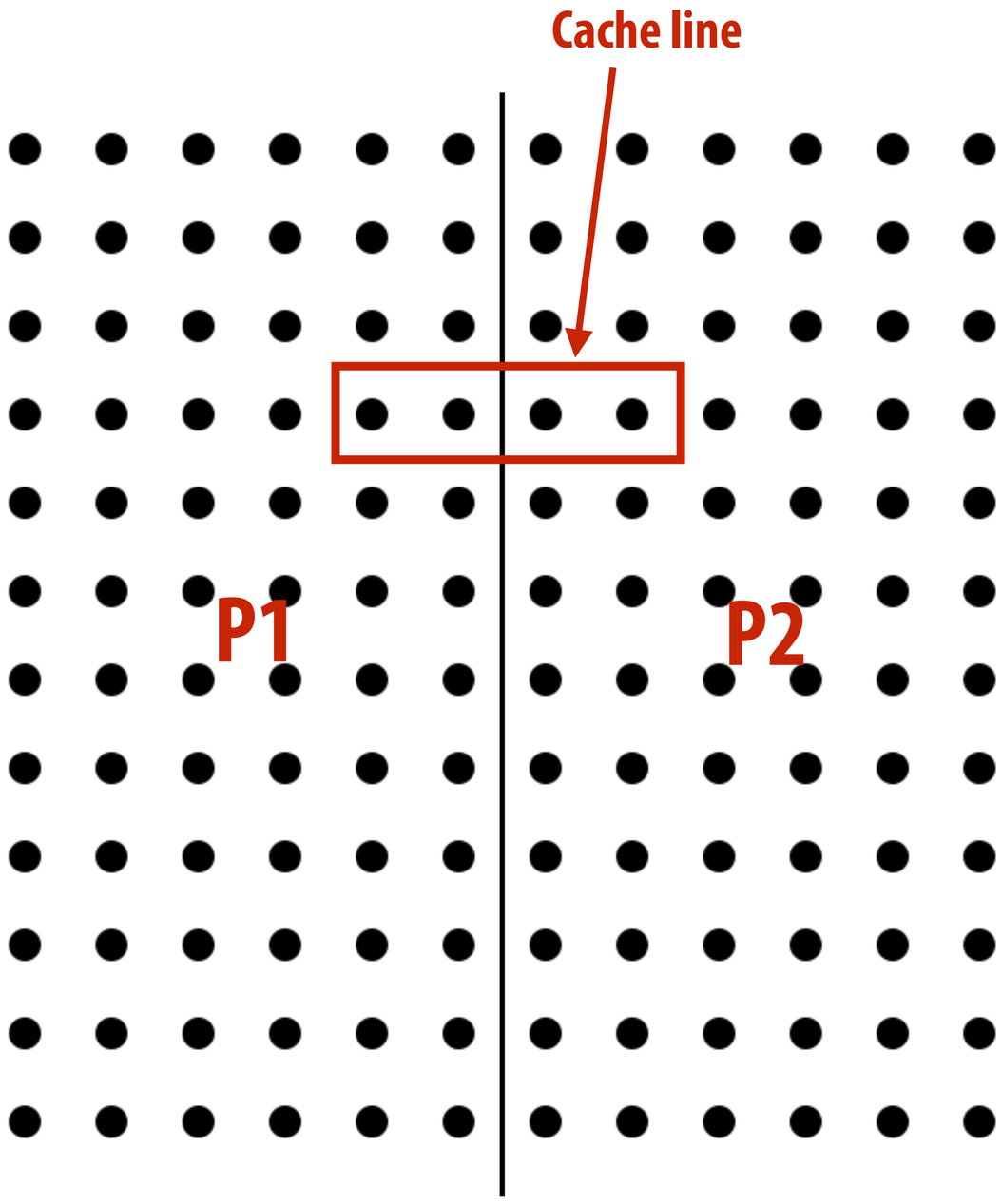
Bad: higher complexity, overhead of directory storage, additional latency due to longer critical path

Artifactual vs. inherent communication

INHERENT COMMUNICATION

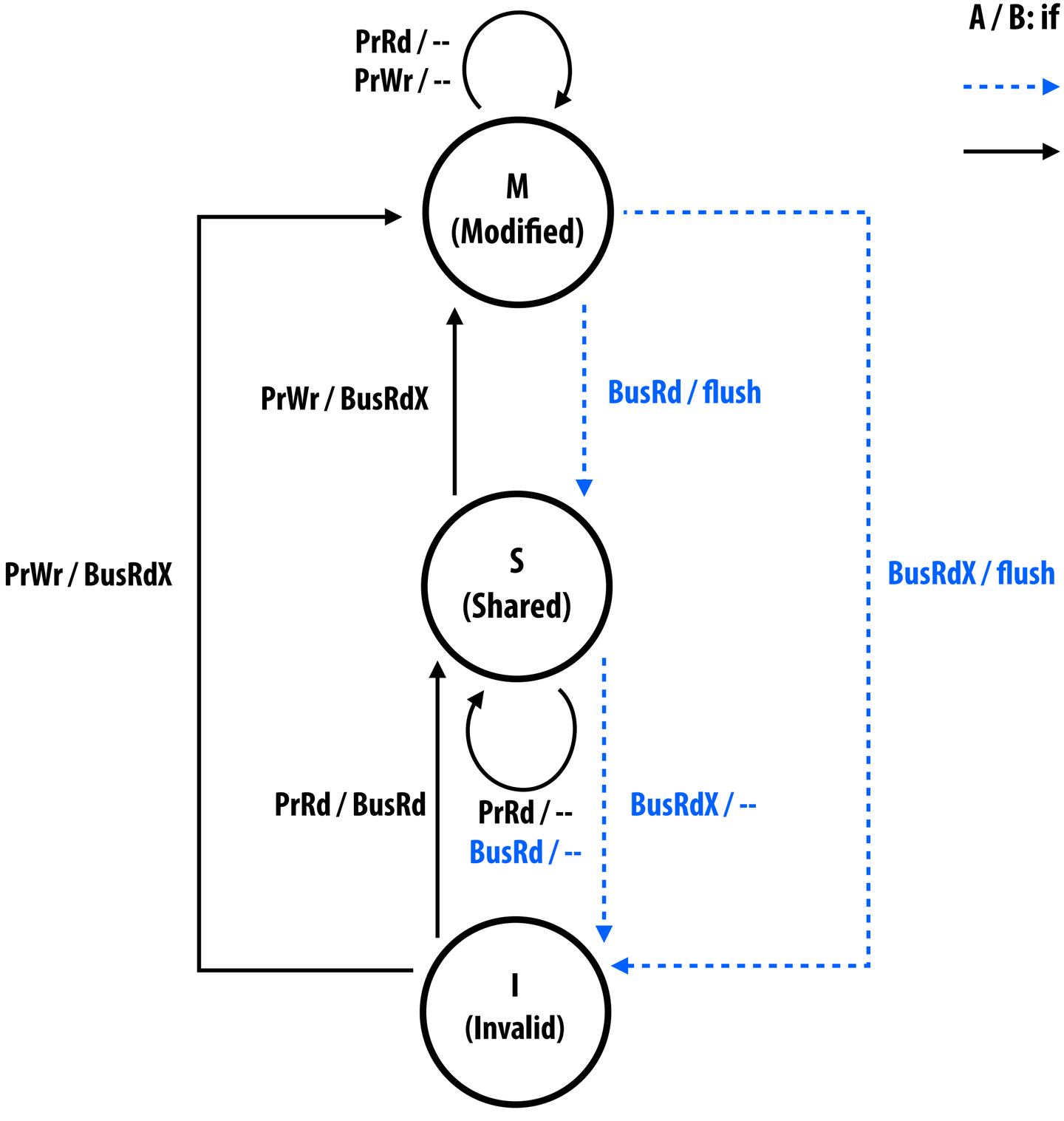
ARTIFACTUAL COMMUNICATION

FALSE SHARING



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

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