Exam details

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

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);
```

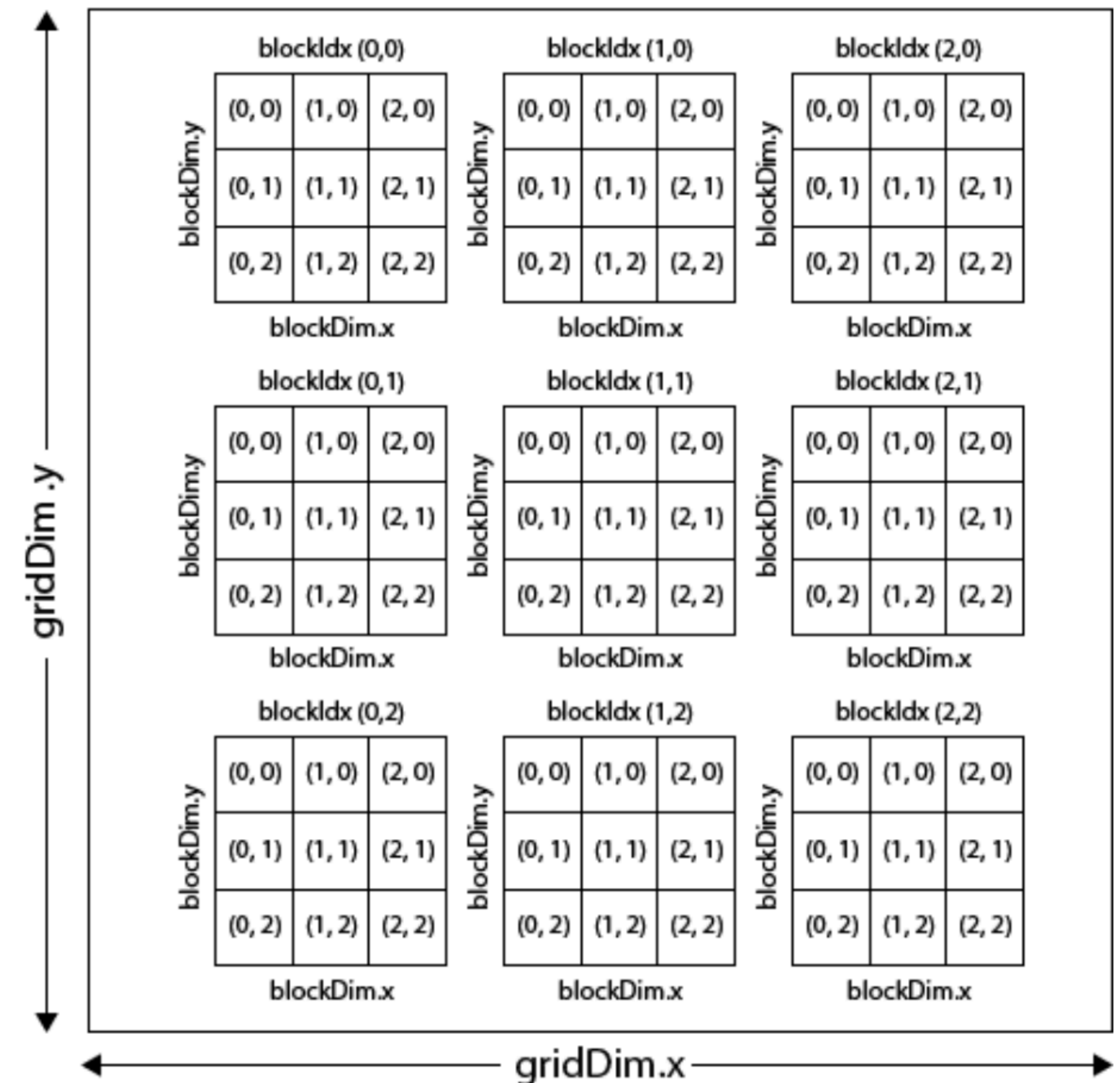
ISPC code: sinx.ispc

```
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;
   }
}
```

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**



Why not have gridIdx and threadDim?

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

Shared address space solver (SPMD execution model)

```
int     n;                    // grid size
bool    done = false;
float   diff = 0.0;
LOCK    myLock;
BARRIER myBarrier;

// allocate grid
float* A = allocate(n+2, n+2);

void solve(float* A) {
   float myDiff;
   int threadId = getThreadId();
   int myMin = 1 + (threadId * n / NUM_PROCESSORS);
   int myMax = myMin + (n / NUM_PROCESSORS)

   while (!done) {
     float myDiff = 0.f;
     diff = 0.f;
     barrier(myBarrier, NUM_PROCESSORS);
     for (j=myMin to myMax) {
        for (i = red cells in this row) {
           float prev = A[i,j];
           A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] +
                            A[i+1,j], A[i,j+1]);
           myDiff += abs(A[i,j] - prev));
     }
     lock(myLock);
     diff += myDiff;
     unlock(myLock);
     barrier(myBarrier, NUM_PROCESSORS);
     if (diff/(n*n) < TOLERANCE)            // check convergence, all threads get same answer
         done = true;
     barrier(myBarrier, NUM_PROCESSORS);
   }
}
```

Improve performance by accumulating into partial sum locally, then complete reduction globally at the end of the iteration.

compute per worker partial sum

Now only lock once per thread, not once per (i,j) loop iteration!

Grid solver example from: Culler, Singh, and Gupta

Message passing solver

Similar structure to shared address space solver, but now communication is explicit in message sends and receives

```
int N;
int tid = get_thread_id();
int rows_per_thread = N / get_num_threads();

float* localA = allocate(rows_per_thread+2, N+2);

// assume localA is initialized with starting values
// assume MSG_ID_ROW, MSG_ID_DONE, MSG_ID_DIFF are constants used as msg ids

//////////////////////////////////////

void solve() {
  bool done = false;
  while (!done) {

    float my_diff = 0.0f;

    if (tid != 0)
       send(&localA[1,0], sizeof(float)*(N+2), tid-1, MSG_ID_ROW);
    if (tid != get_num_threads()-1)
       send(&localA[rows_per_thread,0], sizeof(float)*(N+2), tid+1, MSG_ID_ROW);

    if (tid != 0)
       recv(&localA[0,0], sizeof(float)*(N+2), tid-1, MSG_ID_ROW);
    if (tid != get_num_threads()-1)
       recv(&localA[rows_per_thread+1,0], sizeof(float)*(N+2), tid+1, MSG_ID_ROW);

    for (int i=1; i<rows_per_thread+1; i++) {
       for (int j=1; j<n+1; j++) {
         float prev = localA[i,j];
         localA[i,j] = 0.2 * (localA[i-1,j] + localA[i,j] + localA[i+1,j] +
                              localA[i,j-1] + localA[i,j+1]);
         my_diff += fabs(localA[i,j] - prev);
       }
    }

    if (tid != 0) {
       send(&mydiff, sizeof(float), 0, MSG_ID_DIFF);
       recv(&done, sizeof(bool), 0, MSG_ID_DONE);
    } else {
       float remote_diff;
       for (int i=1; i<get_num_threads()-1; i++) {
          recv(&remote_diff, sizeof(float), i, MSG_ID_DIFF);
          my_diff += remote_diff;
       }
       if (my_diff/(N*N) < TOLERANCE)
         done = true;
       for (int i=1; i<get_num_threads()-1; i++)
         send(&done, sizeof(bool), i, MSD_ID_DONE);
    }
  }
}
```

Send and receive ghost rows to “neighbor threads”

Perform computation
(just like in shared address space version of solver)

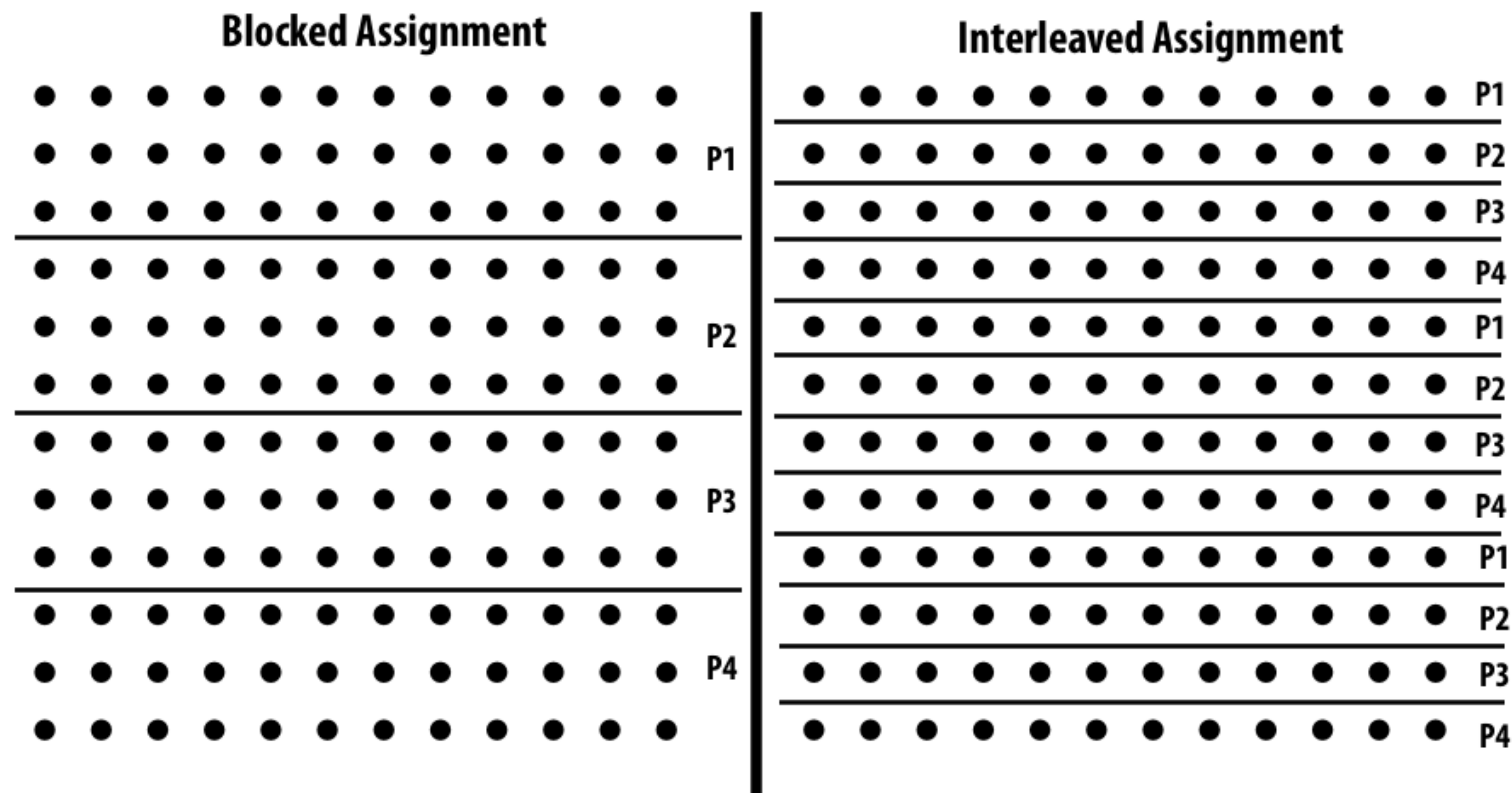
All threads send local my_diff to thread 0

Thread 0 computes global diff, evaluates termination predicate and sends result back to all other threads

Example pseudocode from: Culler, Singh, and Gupta

Static assignment

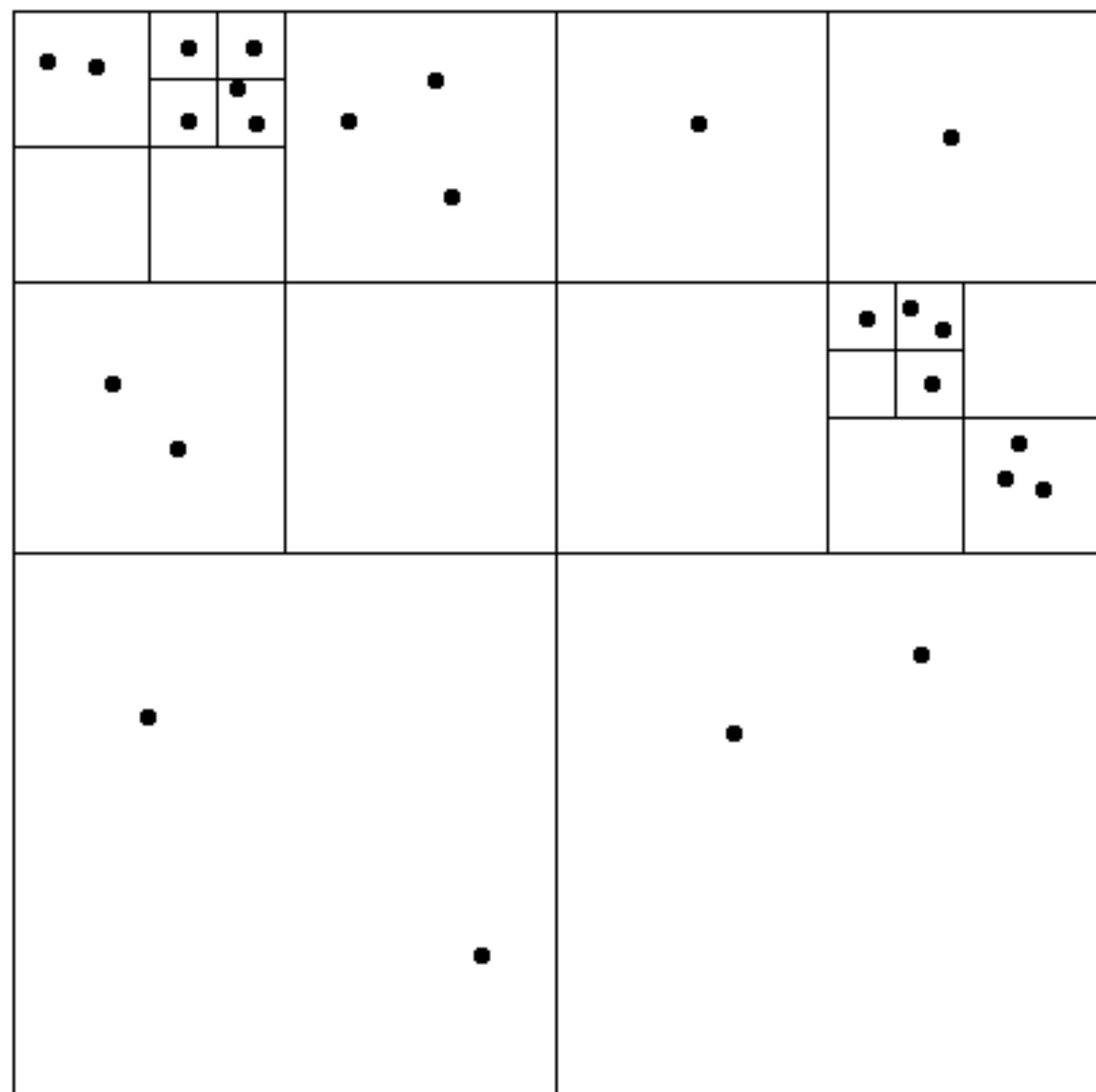
- Assignment of work to threads is pre-determined
  - Not necessarily determined at compile-time (assignment algorithm may depend on runtime parameters such as input data size, number of threads, etc.)
- Recall solver example: assign equal number of grid cells (work) to each thread (worker)
  - We discussed two static assignments of work to workers (blocked and interleaved)



- Good properties of static assignment: simple, essentially zero runtime overhead
(in this example: extra work to implement assignment is a little bit of indexing math)

"Semi-static" assignment

- Cost of work is predictable for near-term future
  - Idea: recent past good predictor of near future
- Application periodically profiles itself and re-adjusts assignment
  - Assignment is "static" for the interval between re-adjustments



Particle simulation:

Redistribute particles as they move over course of simulation (if motion is slow, redistribution need not occur often)

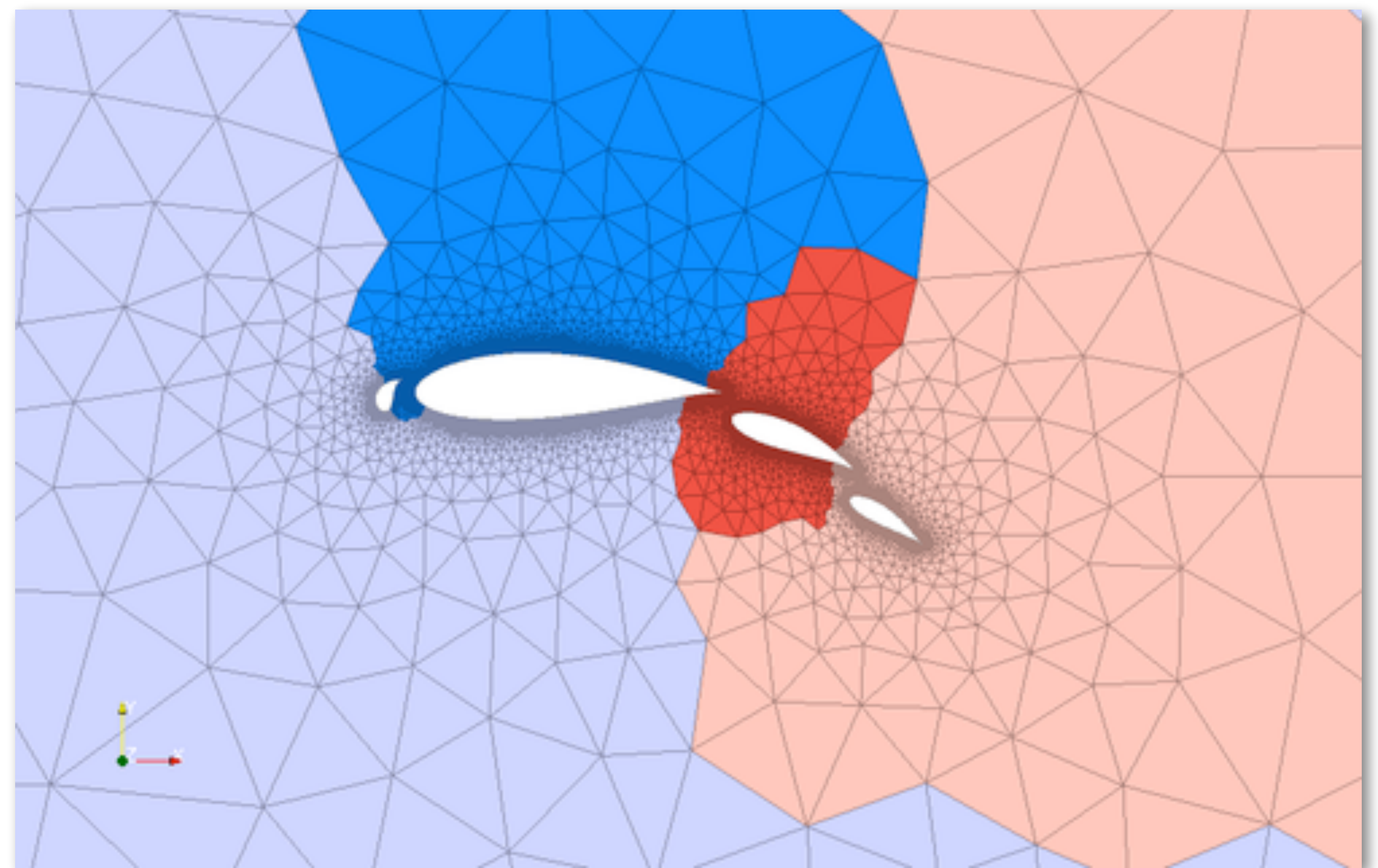


Image credit: http://typhon.sourceforge.net/spip/spip.php?article22

Adaptive mesh:

Mesh is changed as object moves or flow over object changes, but changes occur slowly (color indicates assignment of parts of mesh to processors)

Dynamic assignment

Program determines assignment dynamically at runtime to ensure a well distributed load. (The execution time of tasks, or the total number of tasks, is unpredictable.)

Sequential program
(independent loop iterations)

```
int N = 1024;
int* x = new int[N];
bool* prime = new bool[N];

// initialize elements of x here

for (int i=0; i<N; i++)
{
    // unknown execution time
    is_prime[i] = test_primality(x[i]);
}
```

Parallel program
(SPMD execution by multiple threads, shared address space model)

```
int N = 1024;
// assume allocations are only executed by 1 thread
int* x = new int[N];
bool* is_prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;    // shared variable

while (1) {
  int i;
  lock(counter_lock);
  i = counter++;
  unlock(counter_lock);
  if (i >= N)
     break;
  is_prime[i] = test_primality(x[i]);
}
```

atomic_incr(counter);

Increasing task granularity

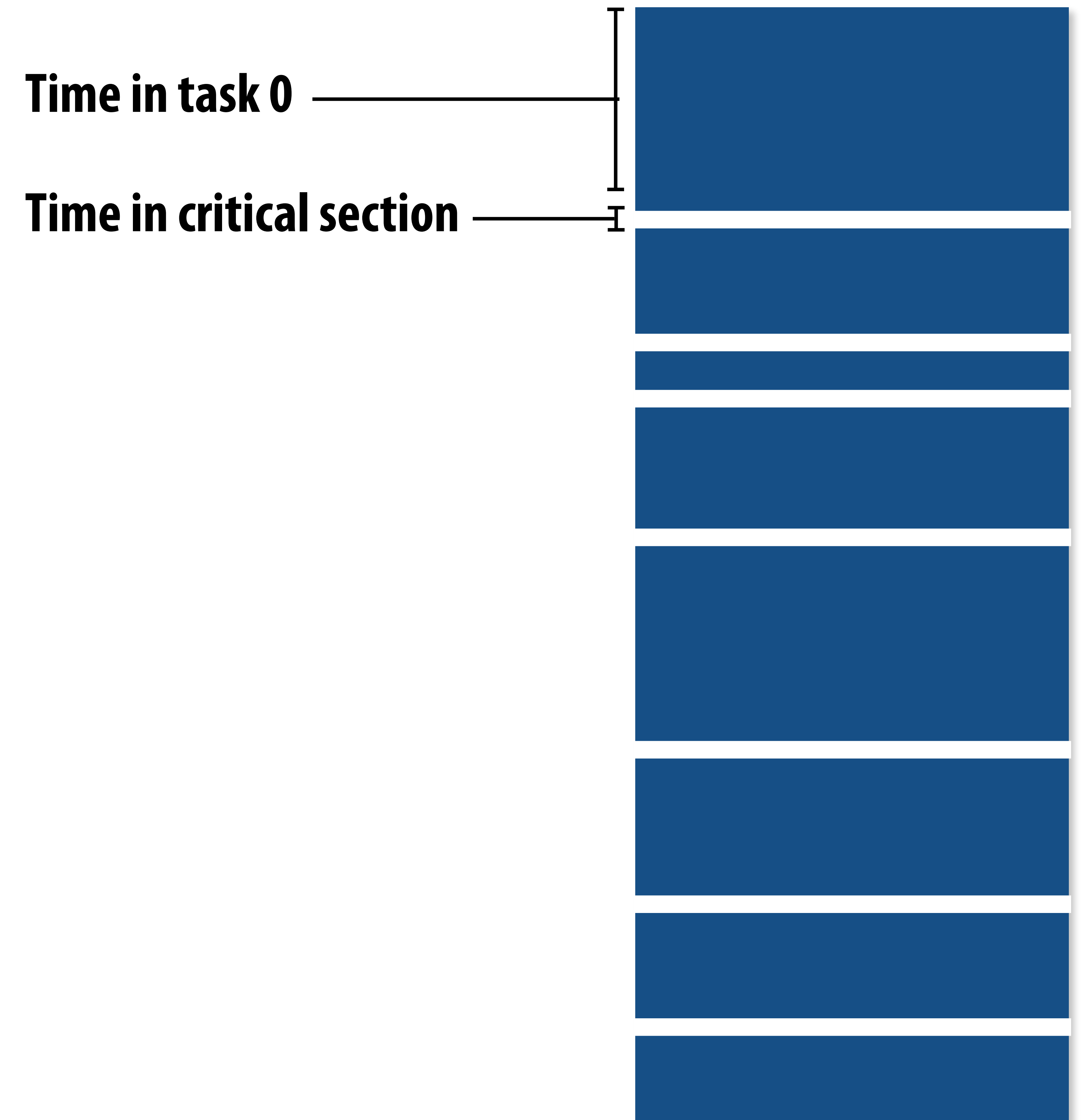
```
const int N = 1024;
const int GRANULARITY = 10;
// assume allocations are only executed by 1 thread

float* x = new float[N];
bool* prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;

while (1) {
  int i;
  lock(counter_lock);
  i = counter;
  counter += GRANULARITY;
  unlock(counter_lock);
  if (i >= N)
     break;
  int end = min(i + GRANULARITY, N);
  for (int j=i; j<end; j++)
     is_prime[j] = test_primality(x[j]);
}
```



Coarse granularity partitioning: 1 "task" = 10 elements

Decreased synchronization cost

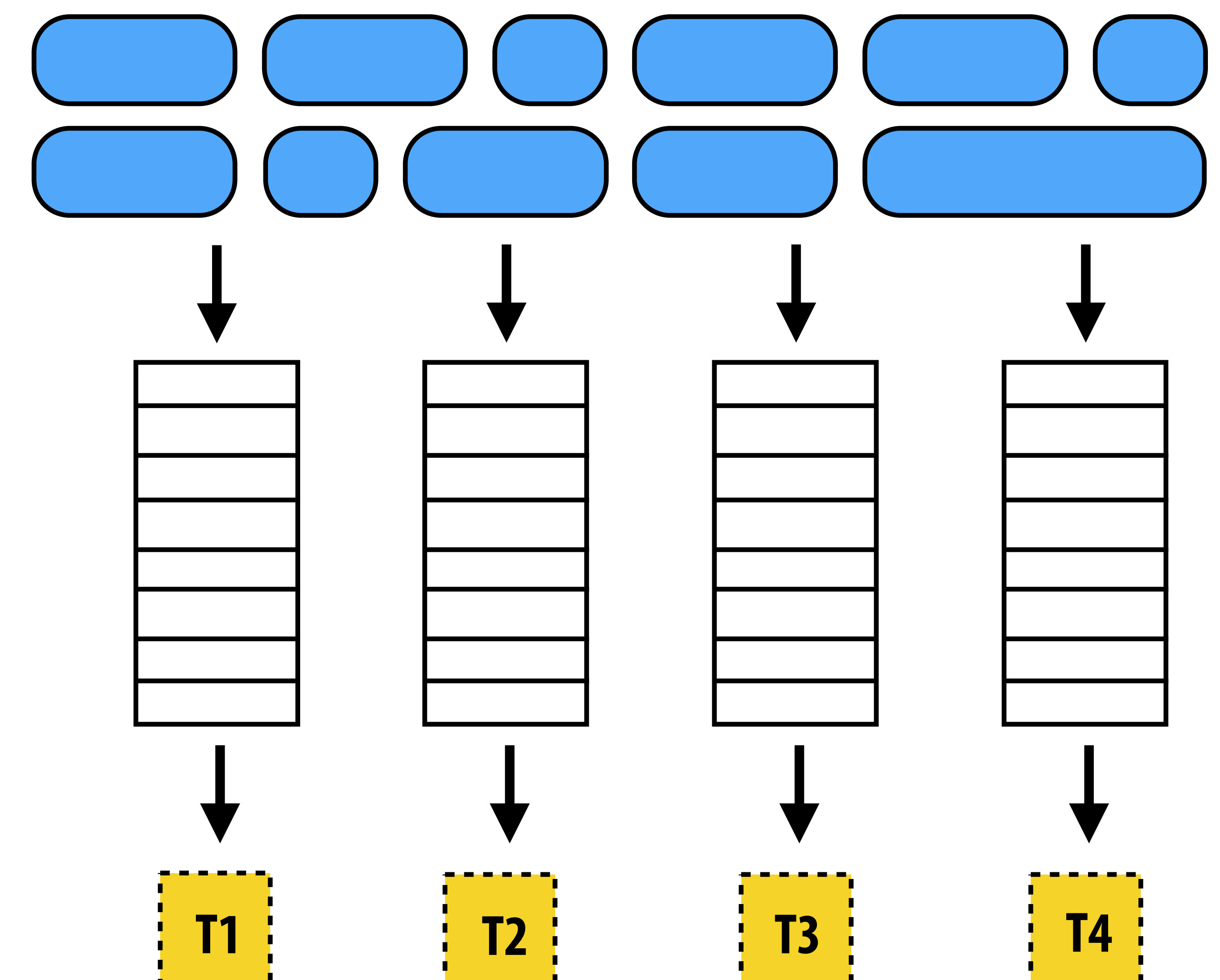
(Critical section entered 10 times less)

Distributed work queues

- Costly synchronization/communication occurs during stealing
  - But not every time a thread takes on new work
  - Stealing occurs only when necessary to ensure good load balance

- Leads to increased locality
  - Common case: threads work on tasks they create (producer-consumer locality)

- Implementation challenges
  - Who to steal from?
  - How much to steal?
  - How to detect program termination?
  - Ensuring local queue access is fast (while preserving mutual exclusion)



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

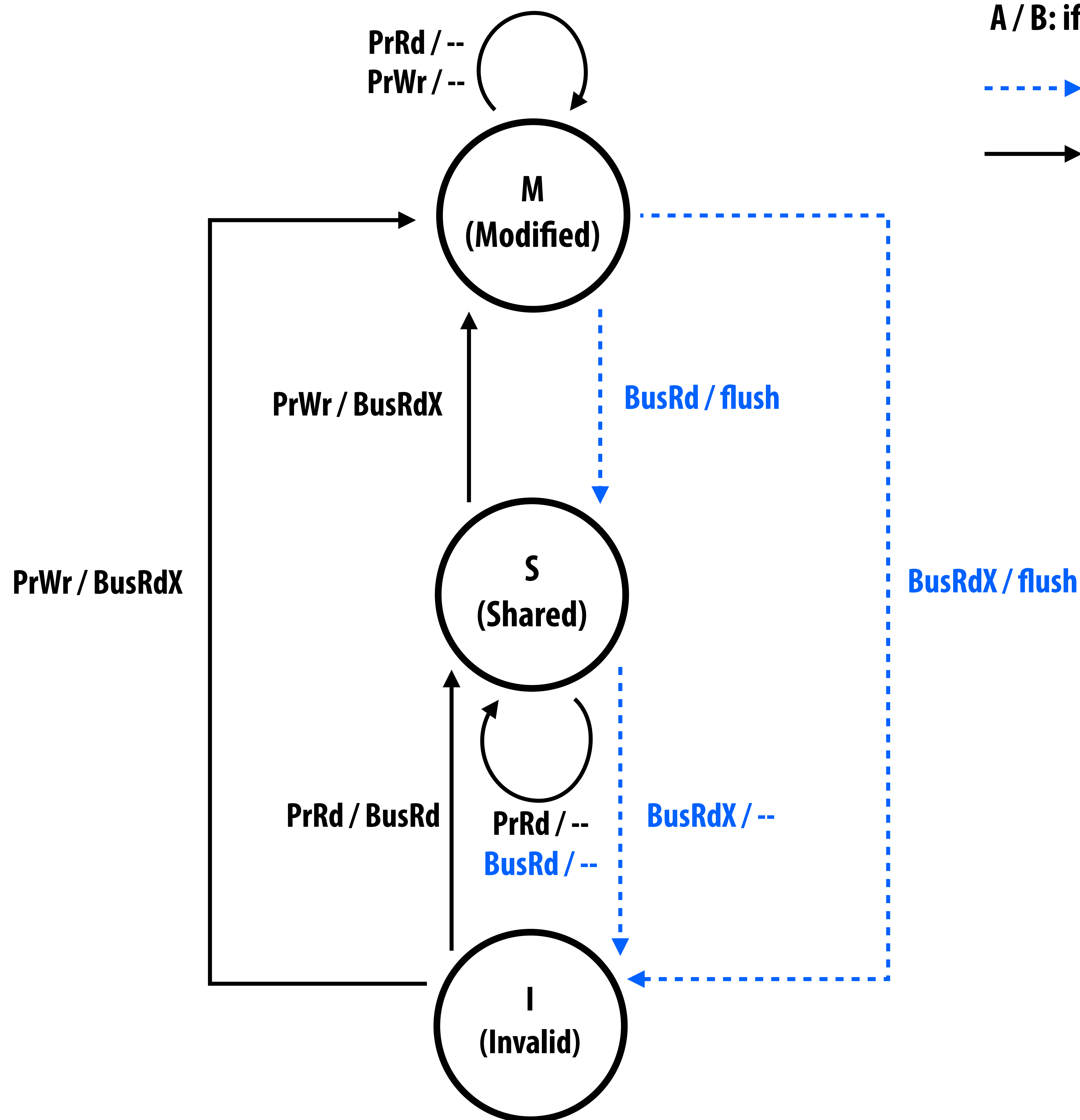
SNOOPING

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

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

Assignment 2, Problem 3

Action	P0 X	P0 Y	P1 X	P1 Y
Initial	I	I	I	I
P0: LD X	S/0			
P1: LD X	S/0		S/0	
P0: ST X ← 1	M/1		I	
P0: ST X ← 2	M/2			
P1: ST X ← 3	I		M/3	
P1: LD X			M/3	
P0: LD X	S/3		S/3	
P0: ST X ← 4	M/4		I	
P1: LD X	S/4		S/4	
P0: LD Y		S/0		
P0: ST Y ← 1	S/4	M/1	S/4	
P1: ST Y ← 2		I		M/2

X and Y have value 0 at start of execution.

Inclusion property of caches

All lines in closer [to processor] cache are also in farther [from processor] cache

- e.g., contents of L1 are a subset of contents of L2
- Thus, all transactions relevant to L1 are also relevant to L2, so it is sufficient for only the L2 to snoop the interconnect

If line is in owned state (M in MSI/MESI) in L1, it must also be in owned state in L2

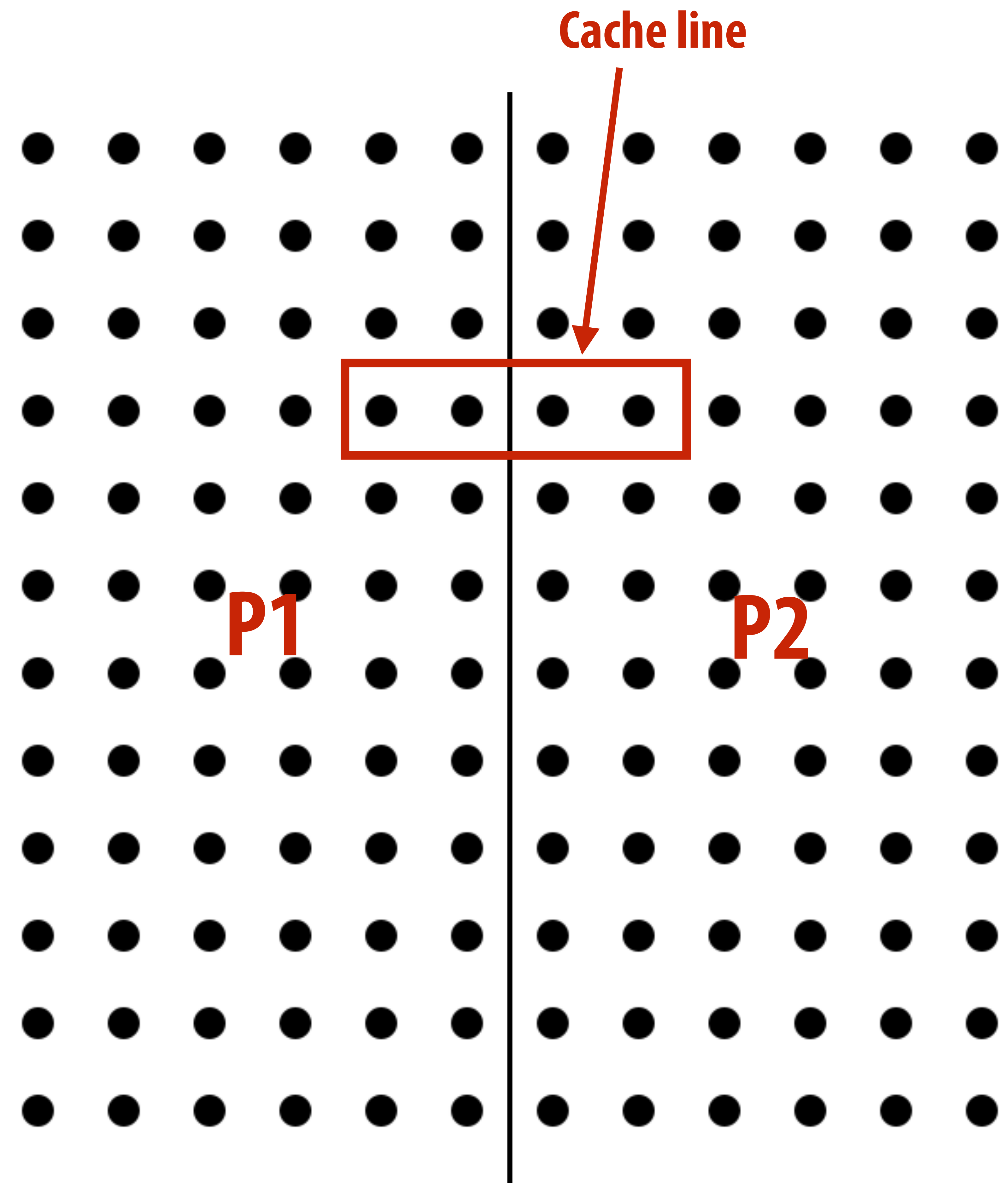
- Allows L2 to determine if a bus transaction is requesting a modified cache line in L1 without requiring information from L1

Artifactual vs. inherent communication

INHERENT COMMUNICATION

ARTIFACTUAL COMMUNICATION

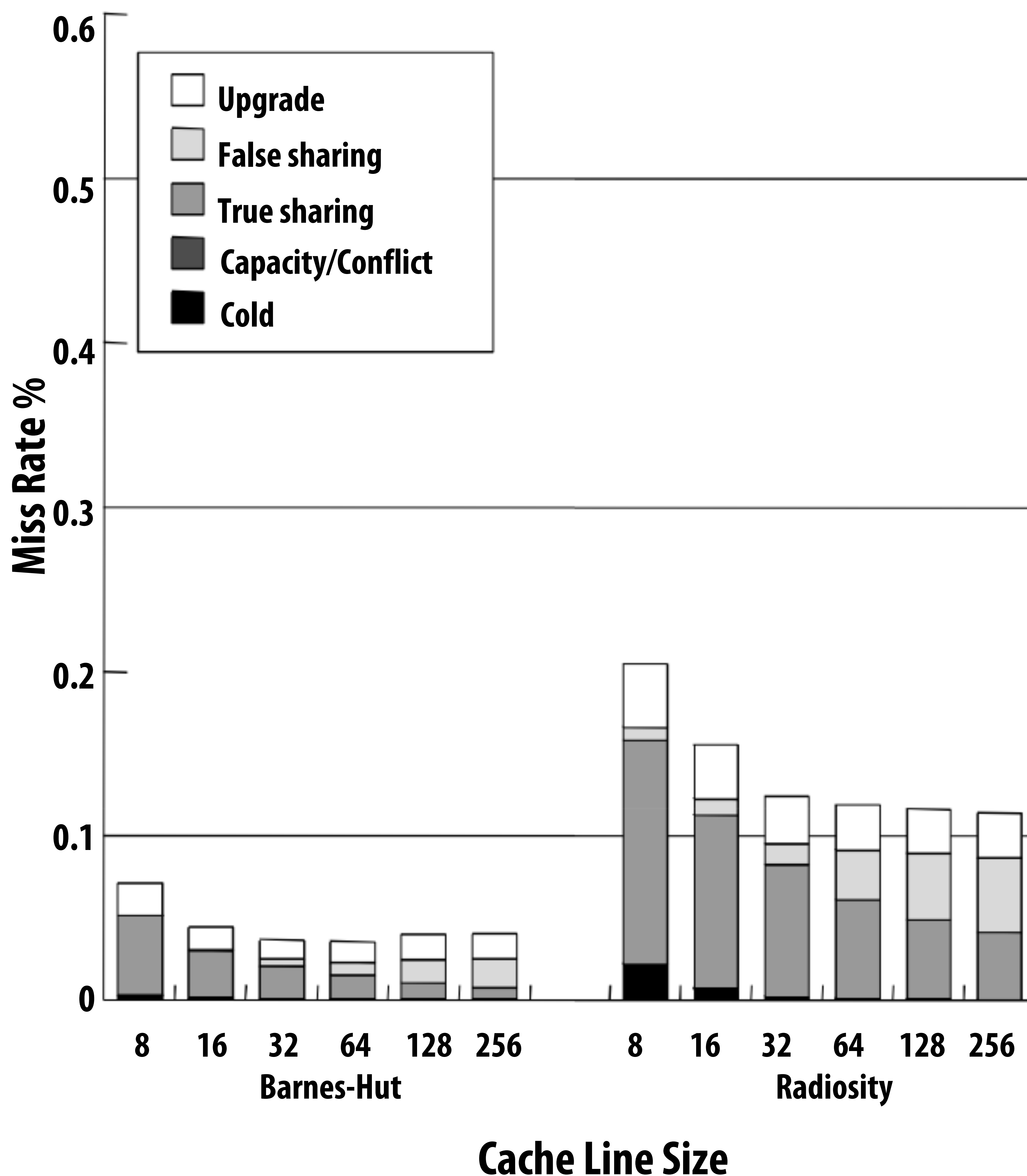
FALSE SHARING

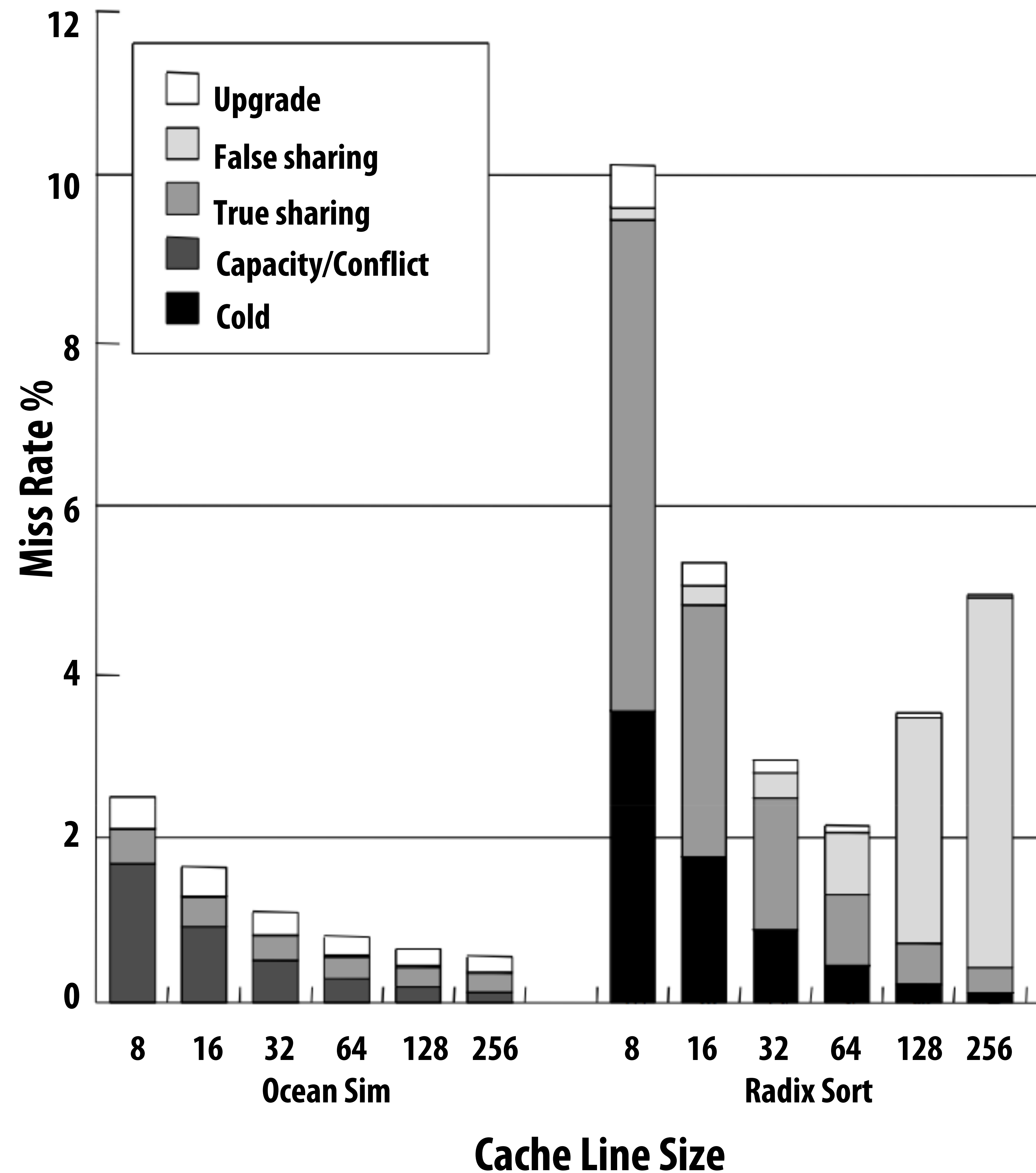


Problem assignment as shown. Each processor reads/writes only from its local data.

Impact of cache line size on miss rate

Results from simulation of a 1 MB cache (four example applications)





*** Note: I separated the results into two graphs because of different Y-axis scales**

Figure credit: Culler, Singh, and Gupta

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.

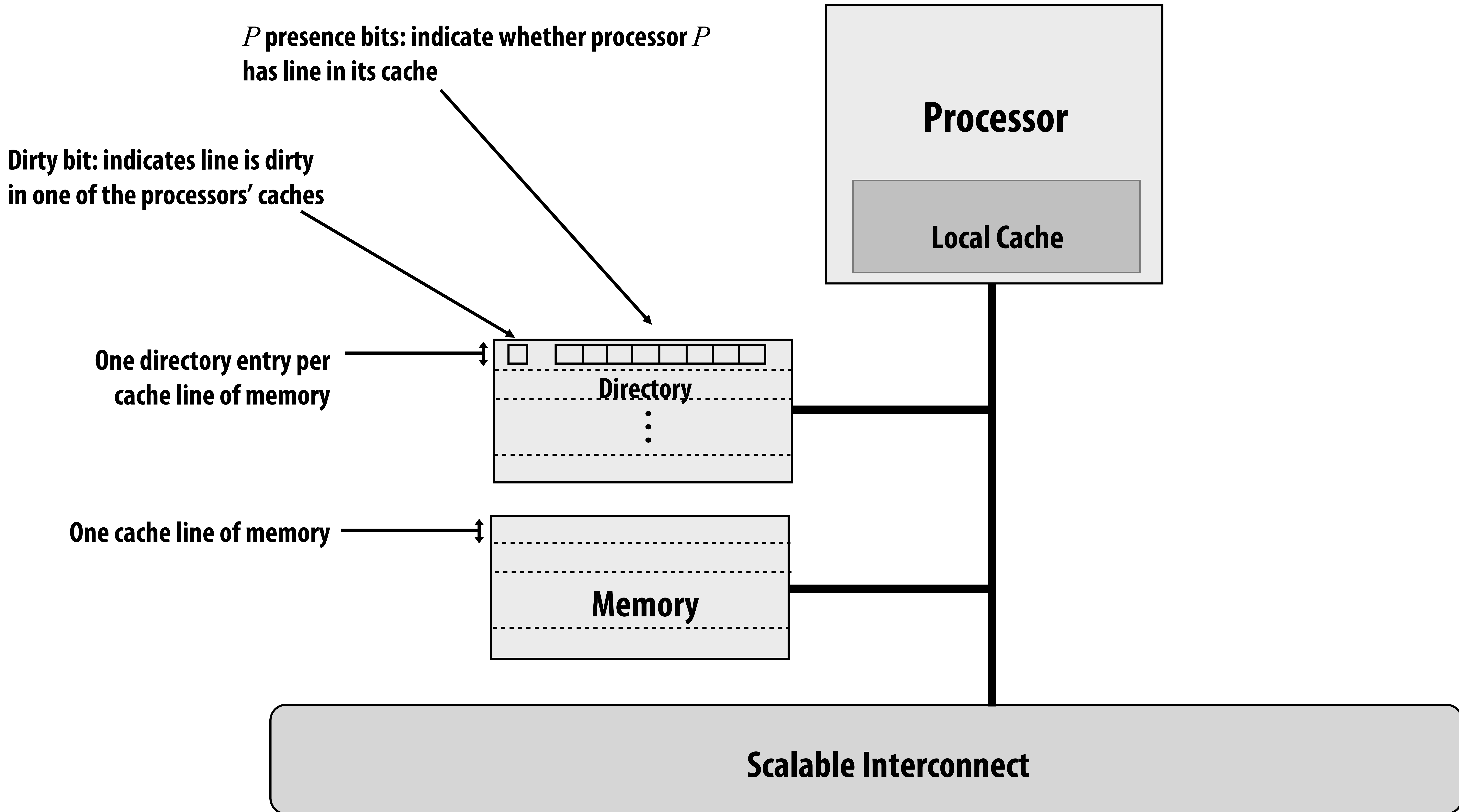
Good: simple, low latency
Bad: broadcast traffic limits scalability

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. Different implementation of directories.

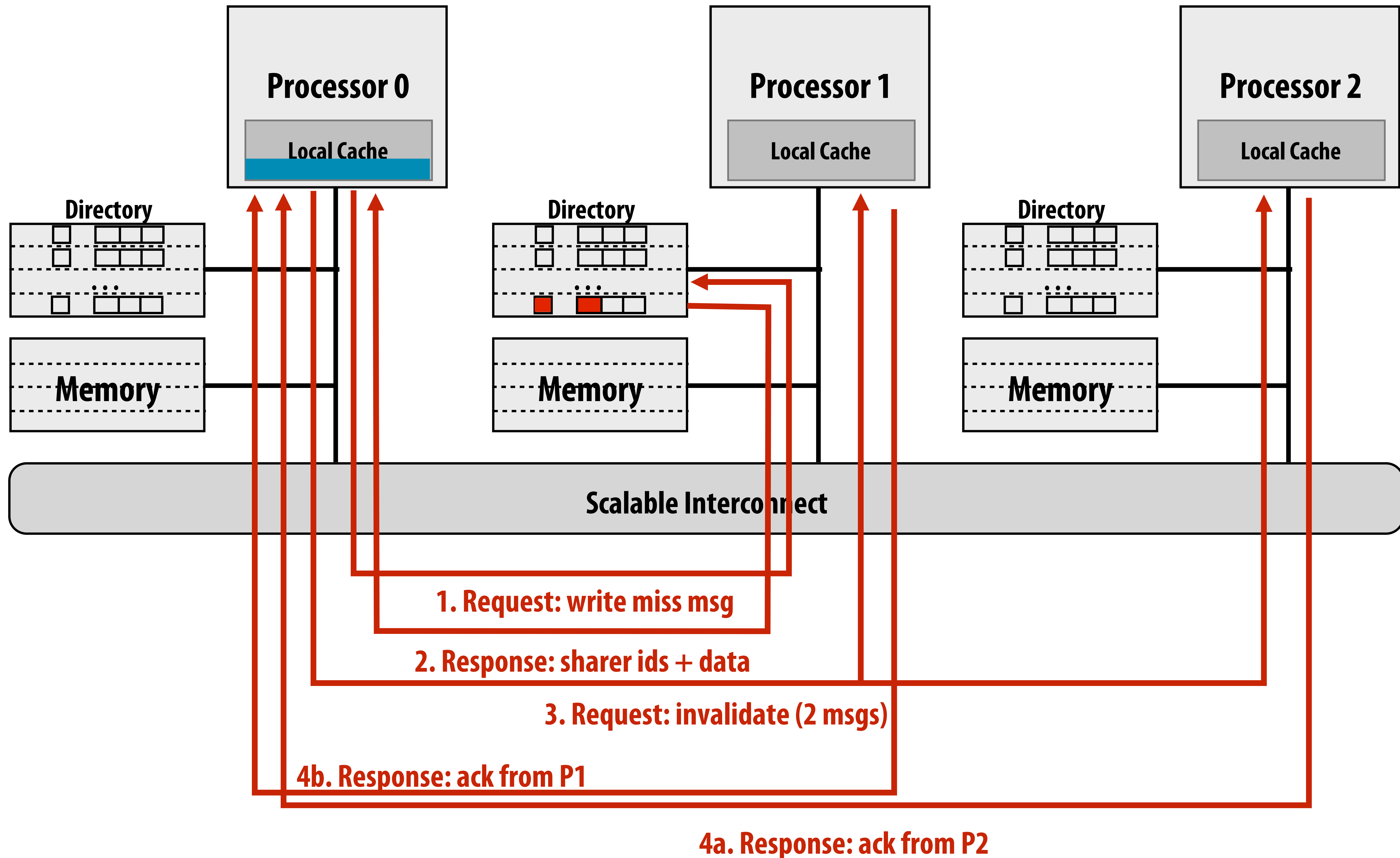
Good: coherence traffic scales with number of sharers, and number of sharers is usually low
Bad: higher complexity, overhead of directory storage, additional latency due to longer critical path

A very simple directory



Example 3: write miss

Write to memory by processor 0: line is clean, but resident in P1's and P2's caches



After receiving both invalidation acks, P0 can perform write

Reducing storage overhead of directory

Optimizations on full-bit vector scheme

- Increase cache line size (reduce M term)
  - What are possible problems with this approach? (consider graphs from last lecture)
- Group multiple processors into a single directory “node” (reduce P term)
  - Need only one directory bit per node, not one bit per processor
  - Hierarchical: could use snooping protocol to maintain coherence among processors in a node, directory across nodes

We will now discuss two alternative schemes

- Limited pointer schemes (reduce P)
- Sparse directories