Lecture 7:

Performance Optimization Part II: Locality, Communication, and Contention

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

Tunes

Beth Rowley

Nobody's Fault but Mine

(Little Dreamer)

- Beth Rowley, after starting too late on Assignment 2 and realizing it was harder than she thought.

Today: more parallel program optimization

- **Last lecture: strategies for assigning work to (threads, processors, “workers”)**
  - Goal: achieving good workload balance while also minimizing overhead
  - Tradeoffs between static and dynamic work assignment
  - Tip: keep it simple (implement, analyze, then tune/optimize if required)

- **Today: strategies for minimizing communication costs**

Terminology

Latency

The amount of time needed for an operation to complete.

Example: A memory load that misses the cache has a latency of 200 cycles.

A packet takes 20 ms to be sent from my computer to Google.

Bandwidth

The rate at which operations are performed.

Example: Memory can provide data to the processor at 25 GB/sec.

A communication link can send 10 million messages per second.

But let’s first talk about the idea of pipelining

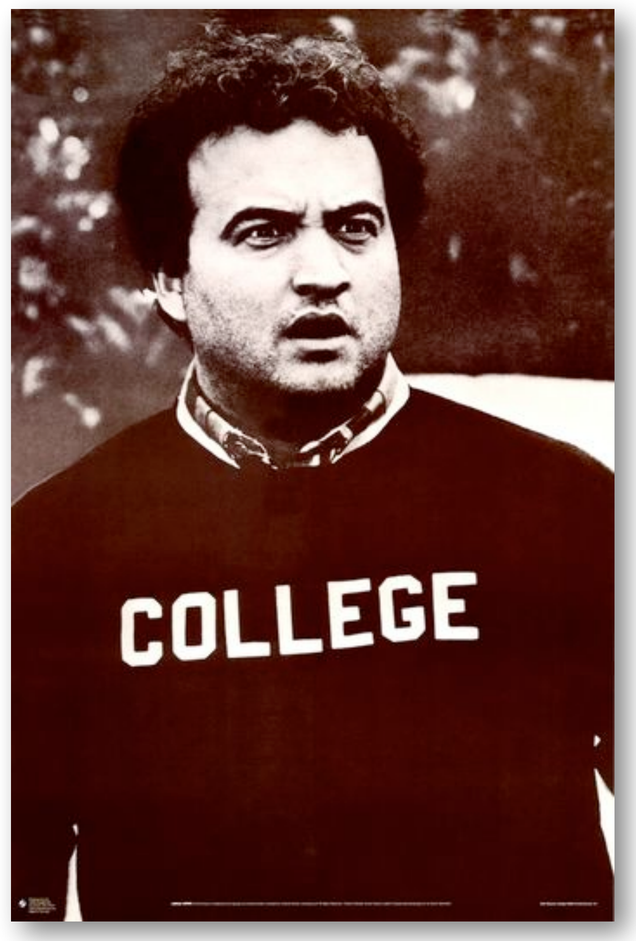
Example: doing your laundry

Operation: do your laundry

1. Wash clothes
2. Dry clothes
3. Fold clothes

Washer
45 min

Dryer
60 min



College Student
15 min

Latency of completing 1 load of laundry = 2 hours

Increasing laundry throughput

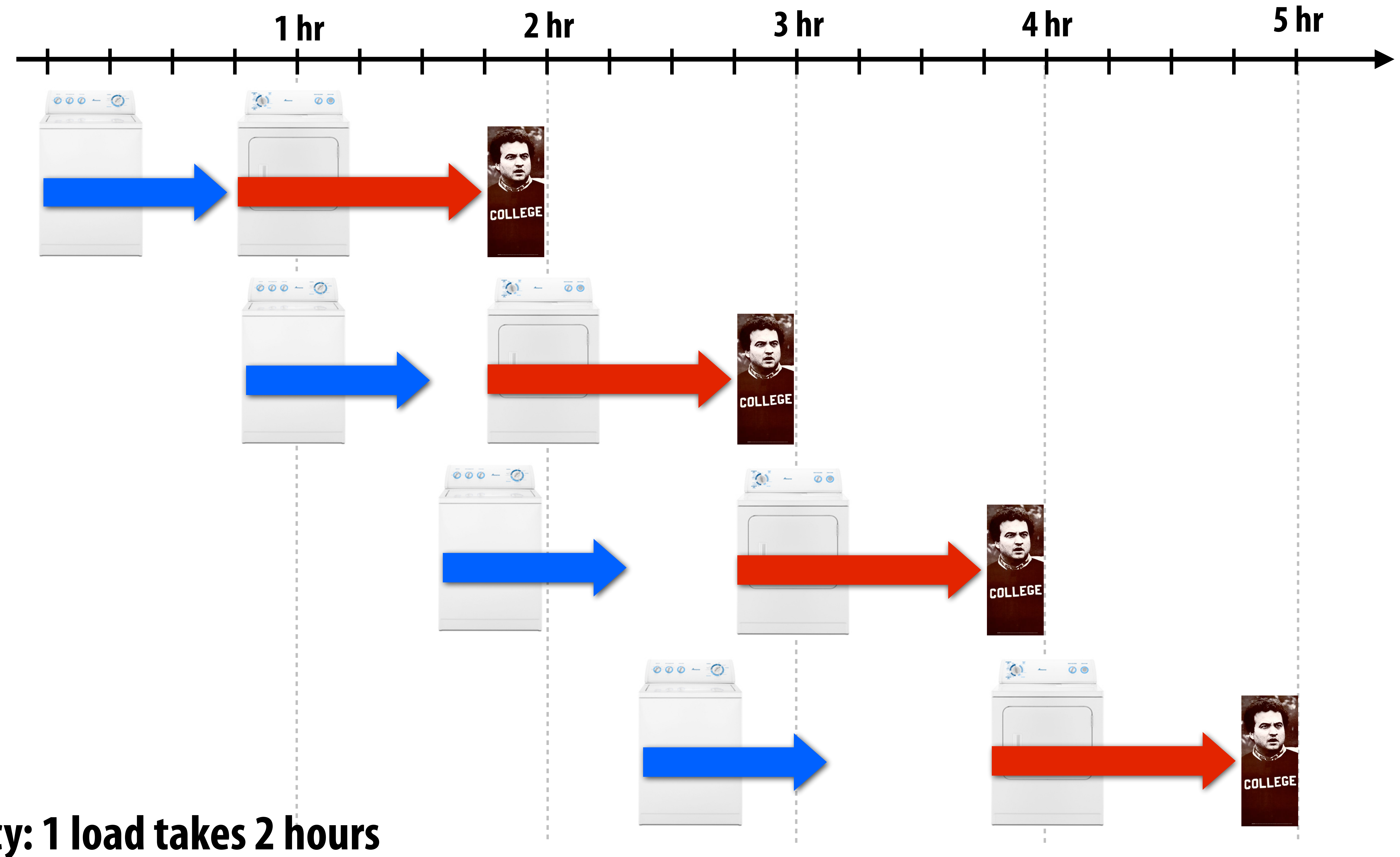
Goal: maximize throughput of many loads of laundry

On approach: duplicate execution resources:
use two washers, two dryers, and call a friend

Latency of completing 2 loads of laundry = 2 hours
Throughput increases by 2x: 1 load/hour
Number of resources increased by 2x: two washers, two dryers

Pipelining

Goal: maximize throughput of many loads of laundry

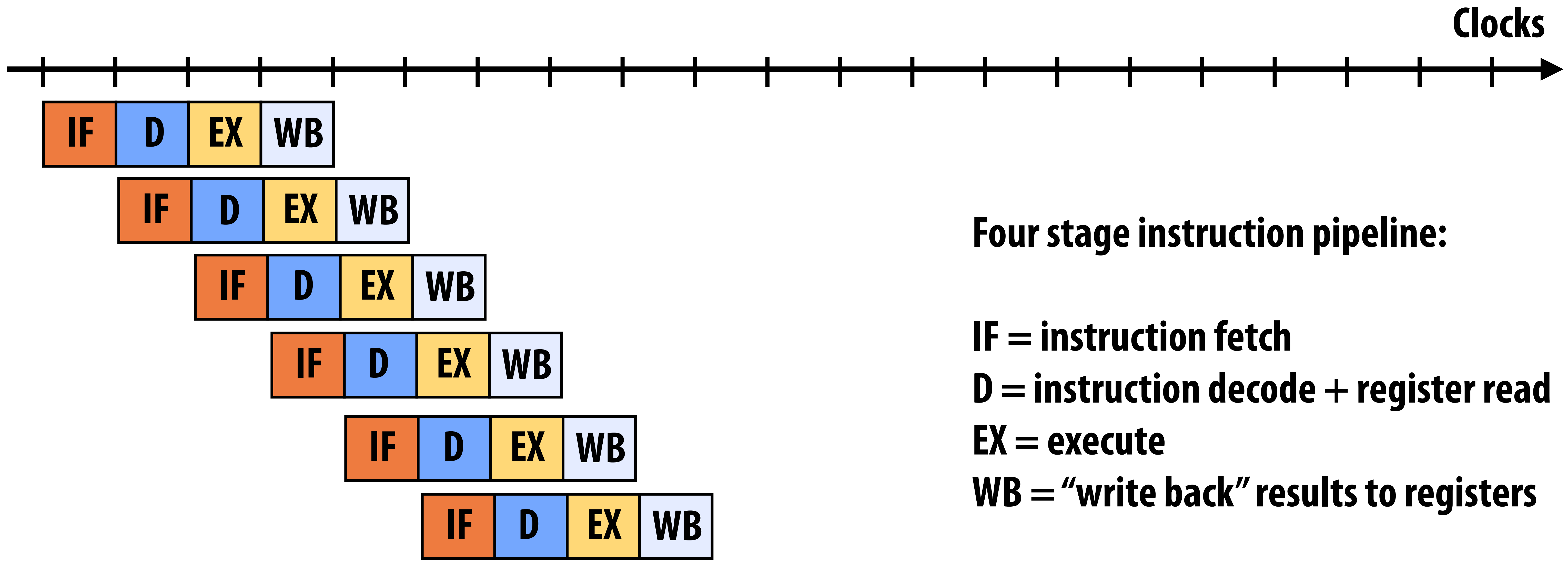


Latency: 1 load takes 2 hours
Throughput: 1 load/hour
Resources: one washer, one dryer

Another example: an instruction pipeline

Break execution of each instruction down into many steps

Enables higher clock frequency (only a simple short operation is done by each unit each clock)



Four stage instruction pipeline:

IF = instruction fetch

D = instruction decode + register read

EX = execute

WB = "write back" results to registers

Latency: 1 instruction takes 4 cycles

Throughput: 1 instruction per cycle

(Yes, care must be taken to ensure program correctness when back-to-back instructions are dependent.)

Intel Core i7 pipeline is variable length (it depends on the instruction) ~15-20 stages

A very simple model of communication

$$T(n) = T_0 + \frac{n}{B}$$

$T(n)$ = transfer time (overall latency of the operation)

T_0 = start-up latency (e.g., time until first bit arrives)

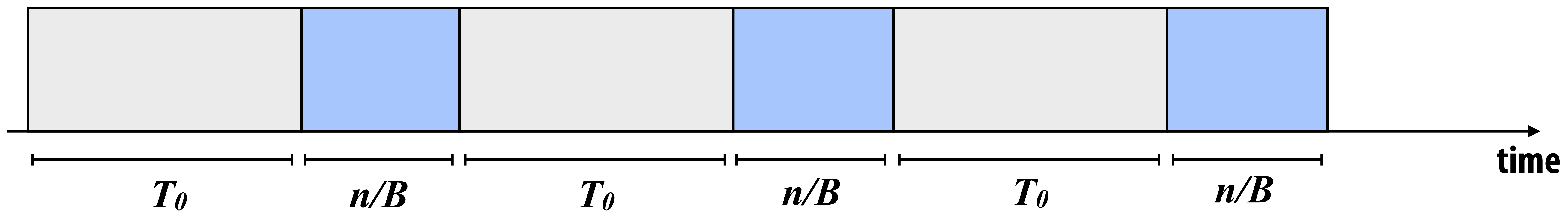
n = bytes transferred in operation

B = transfer rate (bandwidth of the link)

Assumption: processor does no other work while waiting for transfer to complete ...

Effective bandwidth = $n \,/\, T(n)$

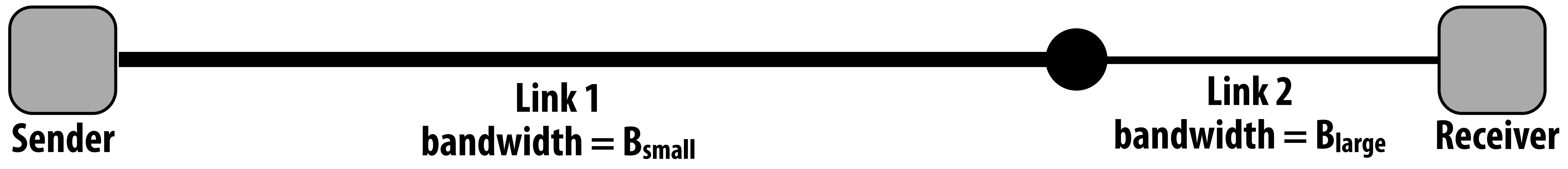
Effective bandwidth depends on transfer size

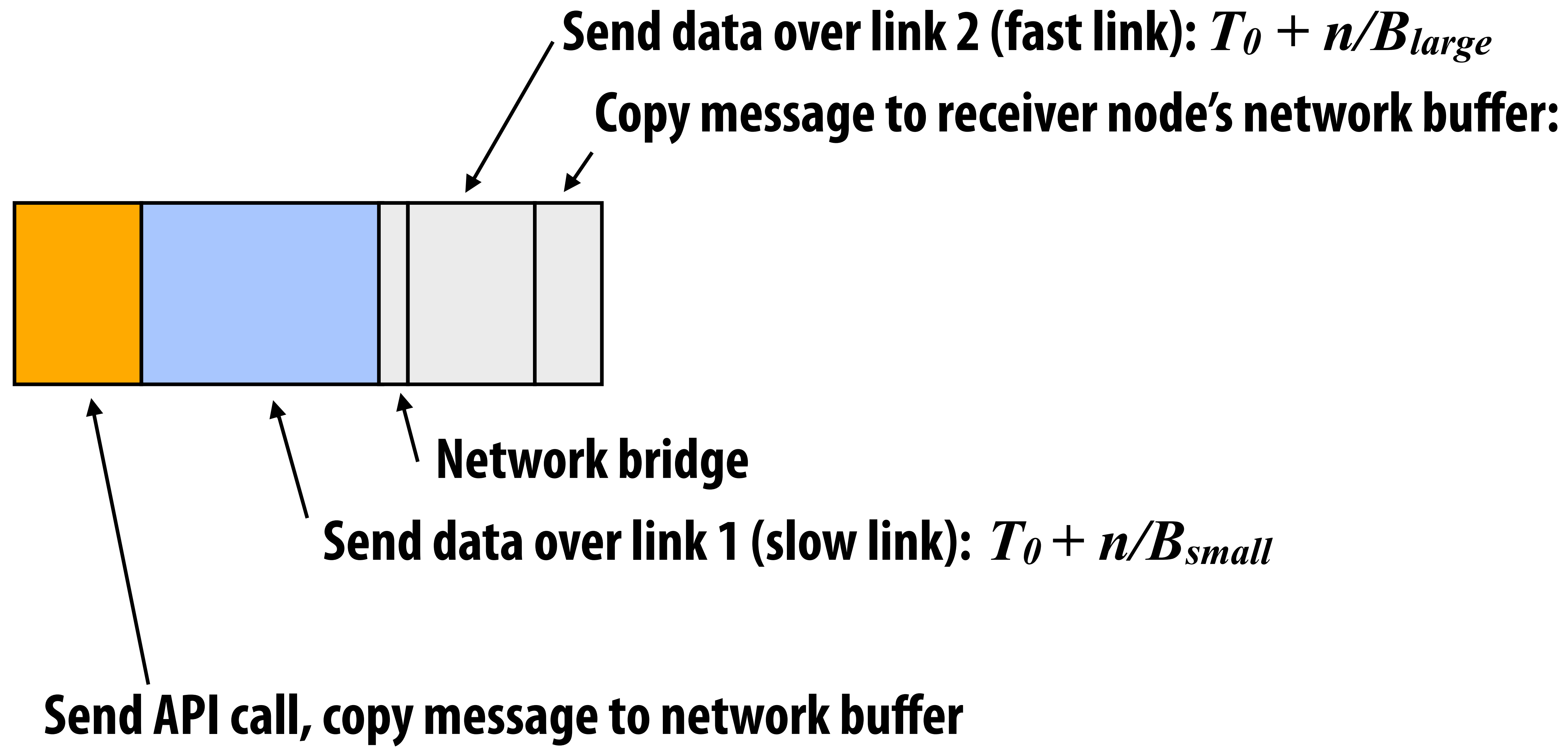


A more general model of communication

Example: sending a n-bit message

Communication time = overhead + occupancy + network delay




Send data over link 2 (fast link): $T_0 + n/B_{large}$
Copy message to receiver node's network buffer:
Network bridge
Send data over link 1 (slow link): $T_0 + n/B_{small}$
Send API call, copy message to network buffer

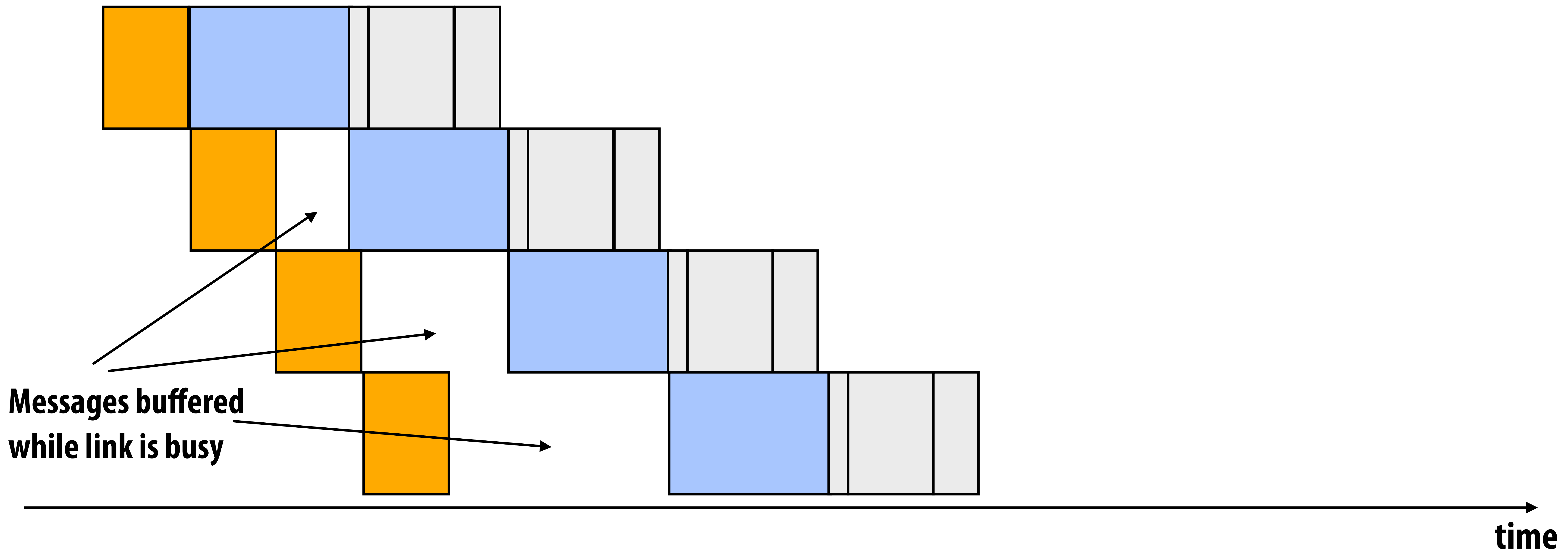
🟧 = Overhead (time spent on the communication by a processor)

🟦 = Occupancy (time for data to pass through slowest component of system)

⬜ = Network delay (everything else)

Example from: Culler, Singh, and Gupta

Pipelined communication



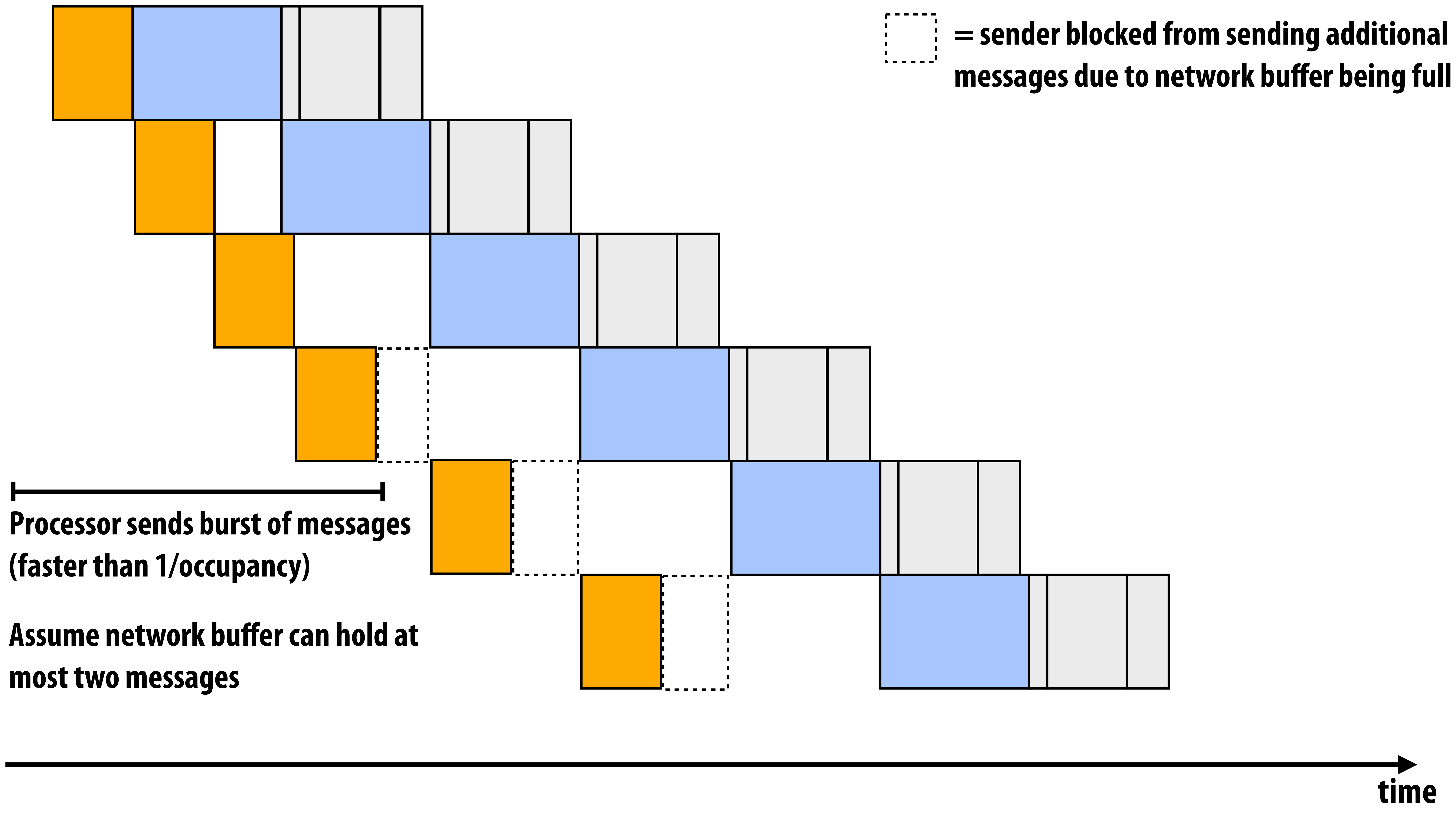
Occupancy determines communication rate (effective bandwidth)

🟧 = Overhead (time spent on the communication by a processor)

🟦 = Occupancy (time for data to pass through slowest component of system)

⬜ = Network delay (everything else)

Pipelined communication


= sender blocked from sending additional messages due to network buffer being full
Processor sends burst of messages (faster than 1/occupancy)
Assume network buffer can hold at most two messages
time

Occupancy determines communication rate

(in steady state: msg/sec = 1/occupancy)

Example from: Culler, Singh, and Gupta

Communication cost

“Cost”

The effect operations have on program execution time

(or some other metric, e.g., power…)

“That function has very high cost” (cost of having to perform the instructions)

“My slow program sends most of its time waiting on memory.” (cost of waiting on memory)

“saxpy achieves low ALU utilization because it is bandwidth bound.” (cost of waiting on memory)

Total communication cost = num_messages x (communication time - overlap)

Overlap: portion of communication performed concurrently with other work

“Other work” can be computation or other communication (as in the previous example)

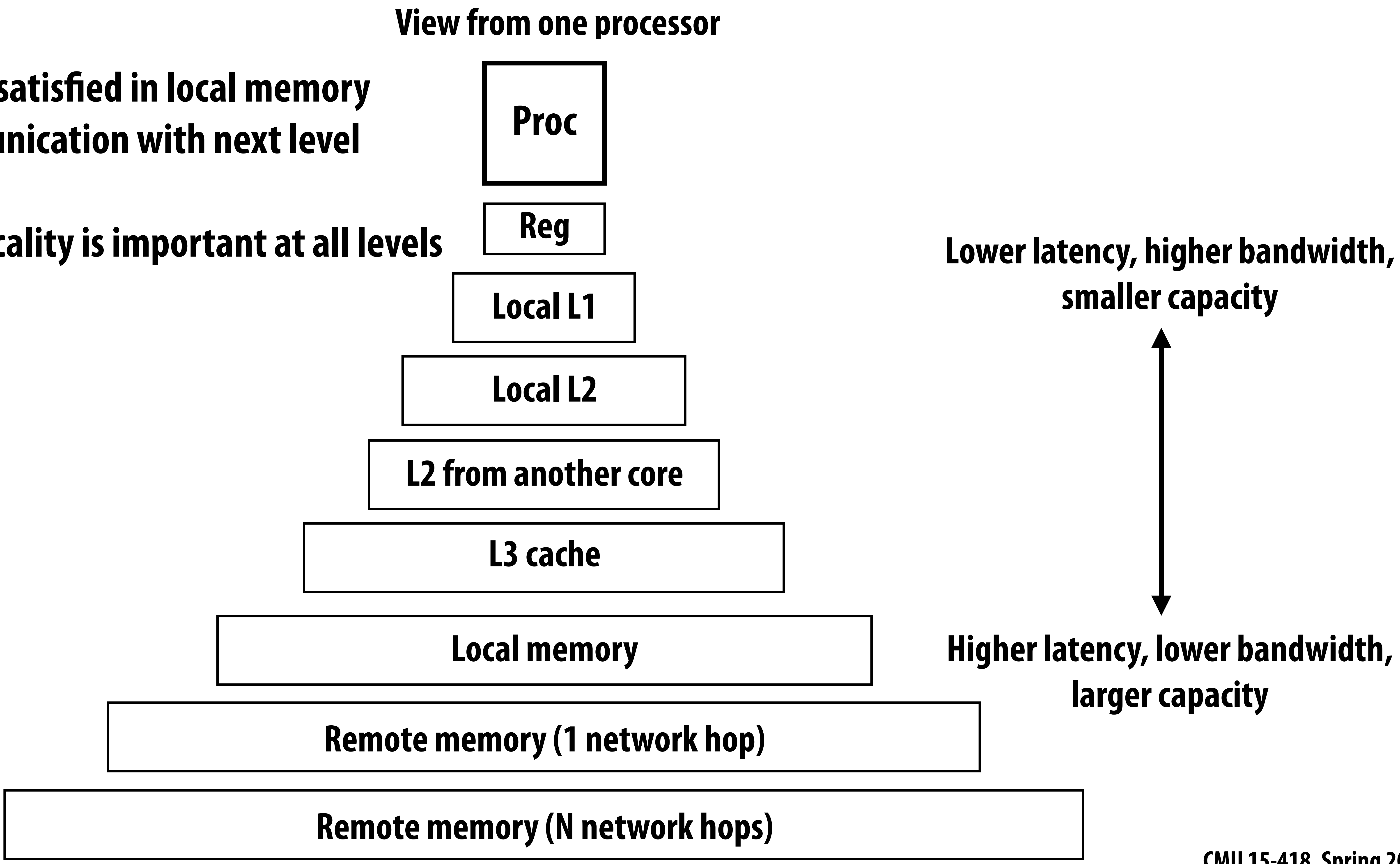
Remember, what really matters is not the absolute cost of communication, but its cost relative to the cost of the computation fundamental to the problem being solved.

Parallel system as an extended memory hierarchy

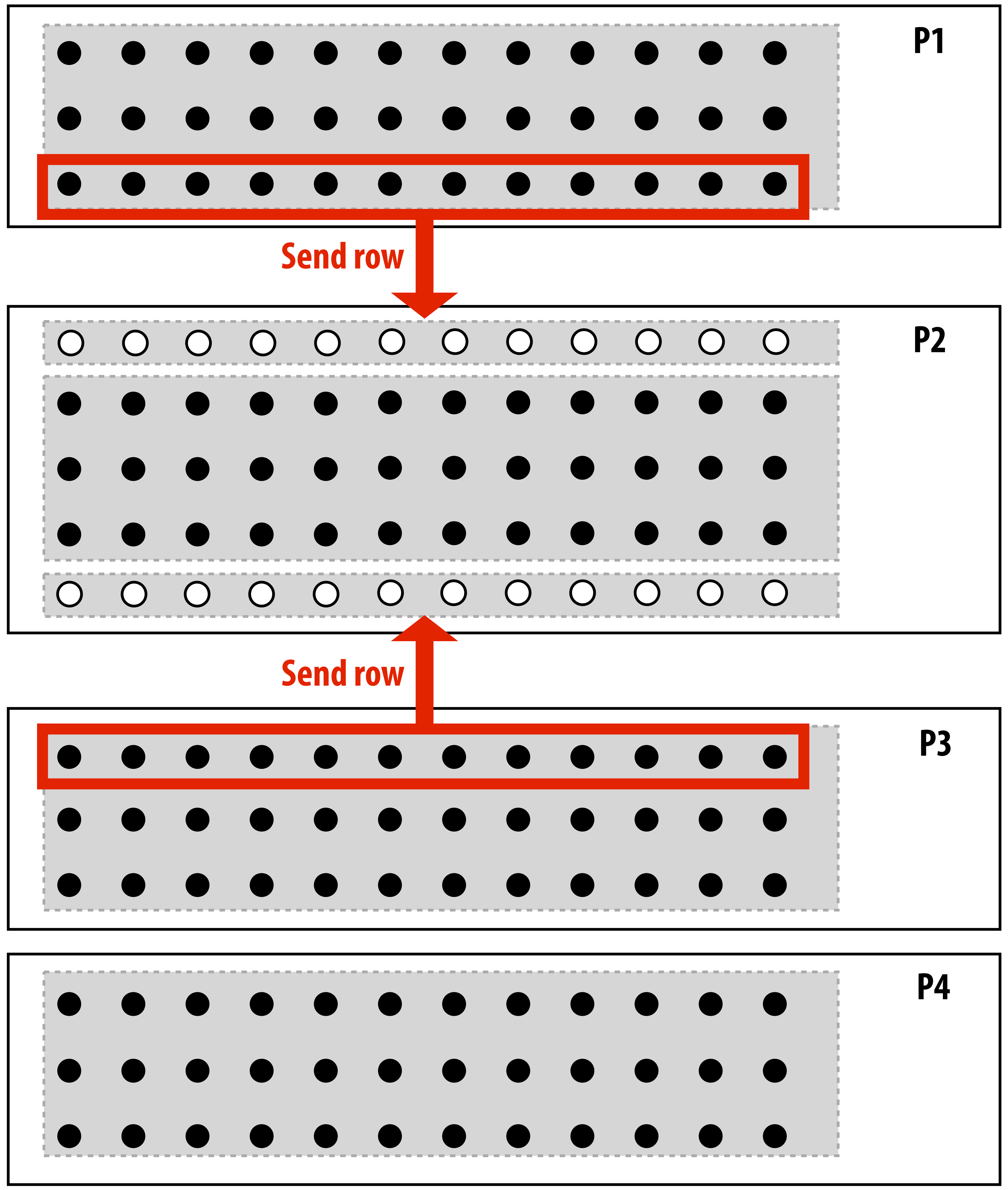
- You should think of "communication" very generally
  - Communication between a processor and its cache
  - Communication between processor and memory (e.g., memory on same machine)
  - Communication between processor and a remote memory (e.g., memory on another node in the cluster — e.g. by sending a network message)

Accesses not satisfied in local memory cause communication with next level

Managing locality is important at all levels


View from one processor
Proc
Reg
Local L1
Local L2
L2 from another core
L3 cache
Local memory
Remote memory (1 network hop)
Remote memory (N network hops)
Lower latency, higher bandwidth, smaller capacity
Higher latency, lower bandwidth, larger capacity

Inherent communication



Communication that <u>must</u> occur in a parallel algorithm. The communication is fundamental to the algorithm.

Communication-to-computation ratio

$$\frac{\text{amount of communication (e.g., bytes)}}{\text{amount of computation (e.g., instructions)}}$$

- If denominator is execution time of computation, ratio gives average bandwidth requirements

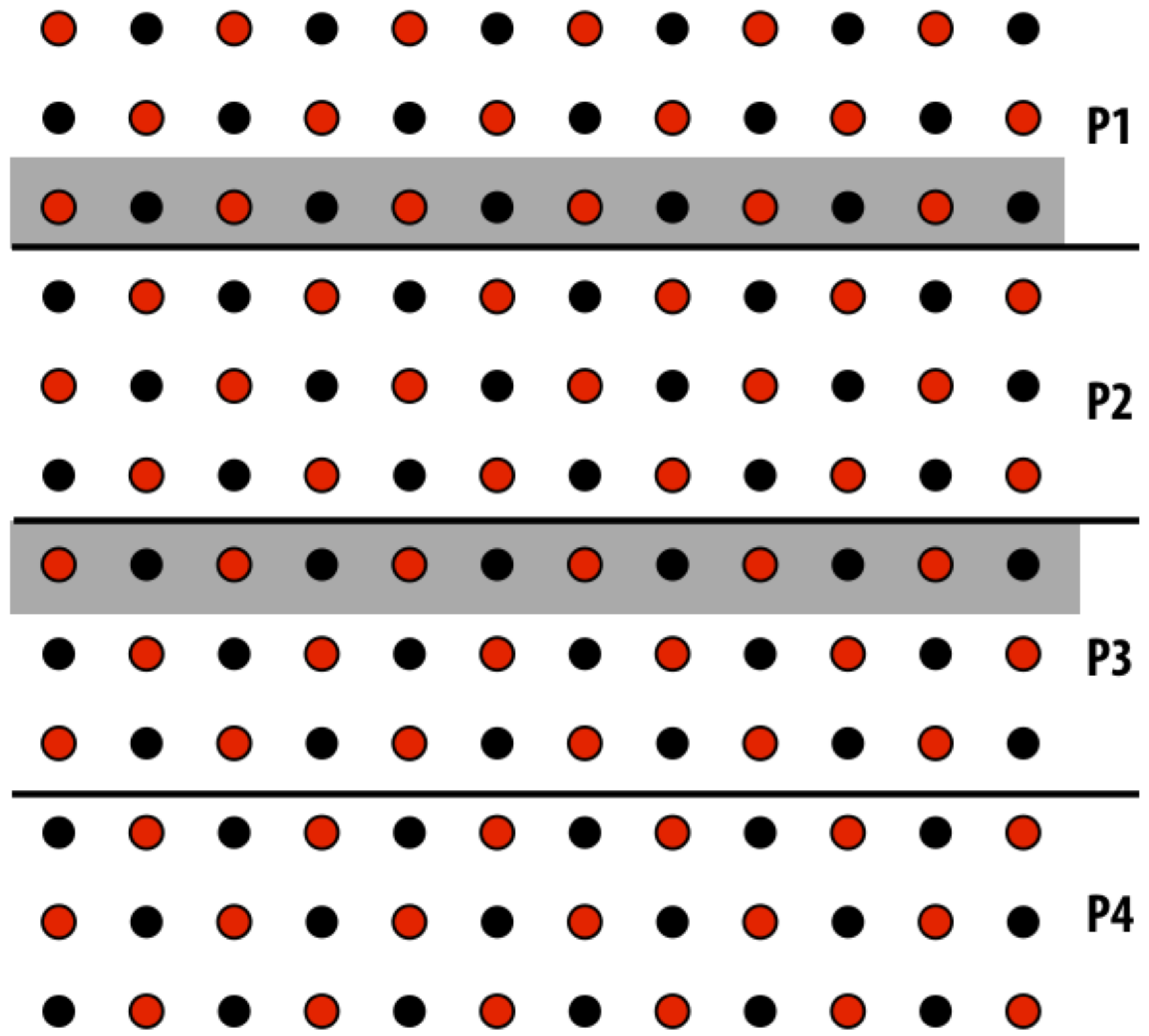
- "Arithmetic intensity" = 1 / communication-to-computation ratio
  - I personally find arithmetic intensity a more intuitive quantity, since higher is better.

- High arithmetic intensity (low communication-to-computation ratio) is required to efficiently utilize modern parallel processors since the ratio of compute capability to available bandwidth is high (recall saxpy)

Reducing inherent communication

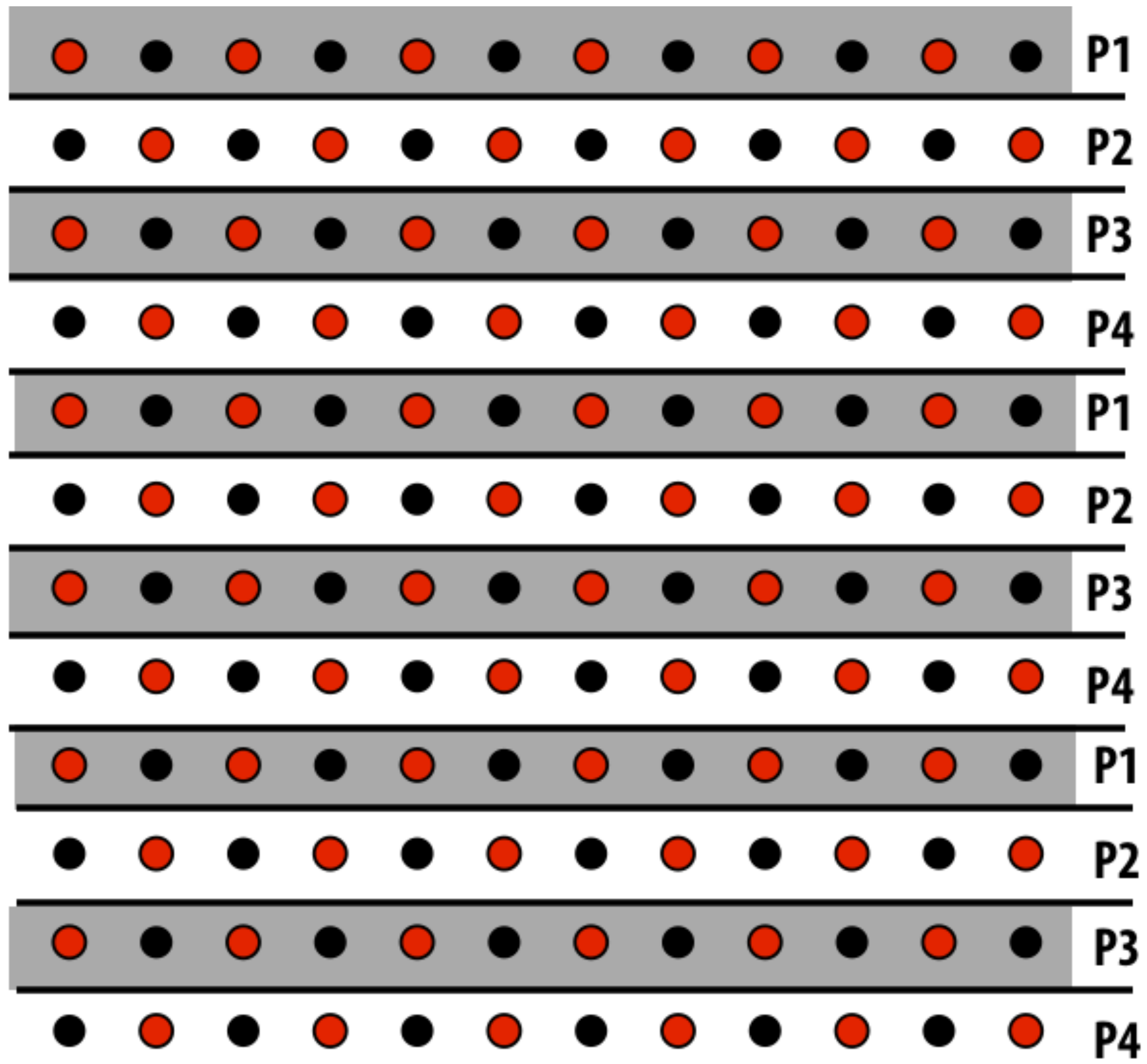
- **Good assignment can reduce inherent communication (increase arithmetic intensity)**

1D blocked assignment: N x N grid



$$\frac{\text{elements computed (per processor)} \approx N^2/P}{\text{elements communicated (per processor)} \approx 2N} \propto N / P$$

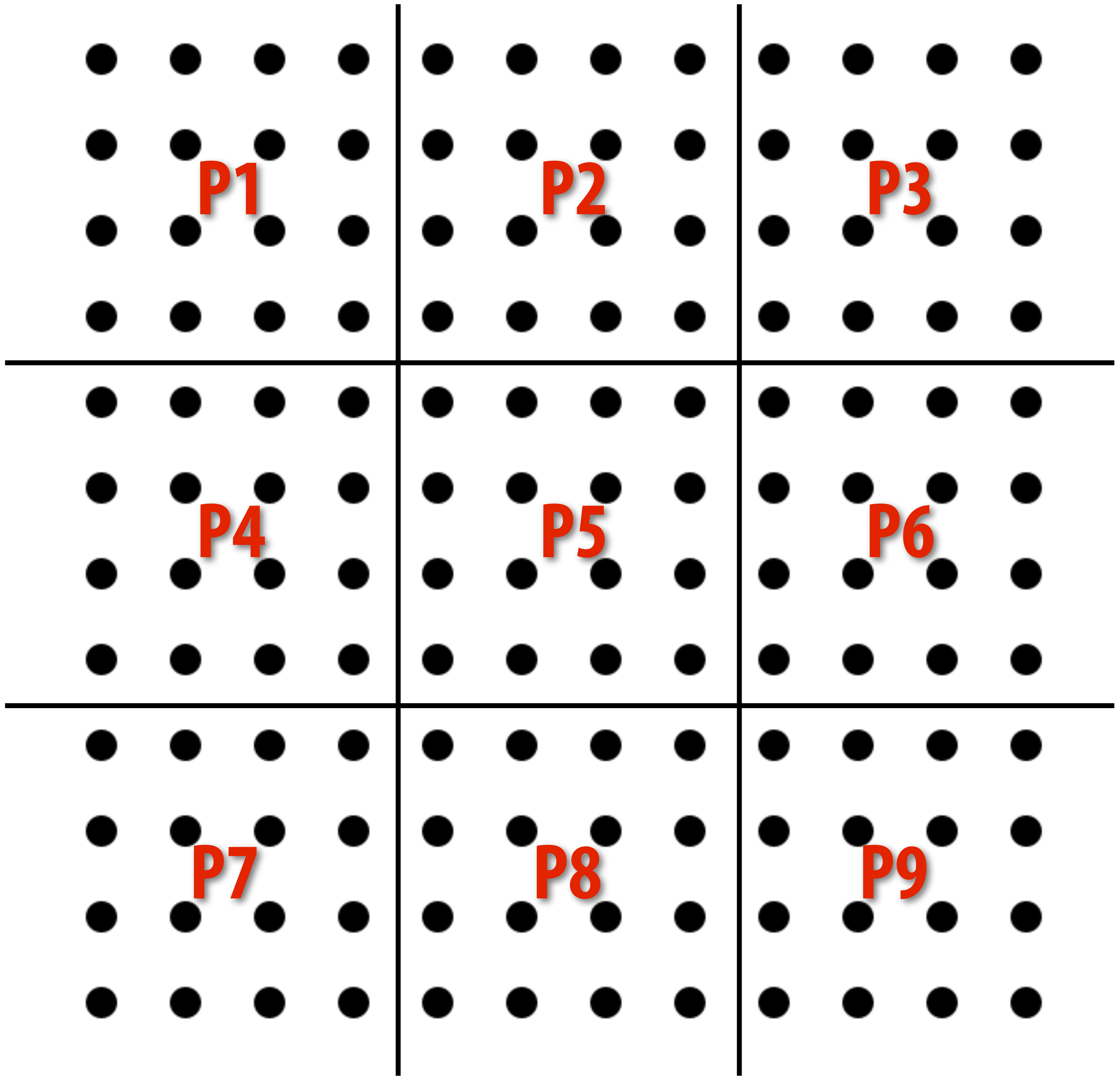
1D interleaved assignment: N x N grid



$$\frac{\text{elements computed}}{\text{elements communicated}} = 1/2$$

Reducing inherent communication

2D blocked assignment: N x N grid



N^2 elements

P processors

elements computed: (per processor) $\frac{N^2}{P}$

elements communicated: (per processor) $\propto \frac{N}{\sqrt{P}}$

arithmetic intensity: $\frac{N}{\sqrt{P}}$

Asymptotically better communication scaling than 1D blocked assignment

Communication costs increase sub-linearly with P

Assignment captures 2D locality of algorithm

Artifactual communication

- **Inherent communication: information that fundamentally must be moved between processors to carry out the algorithm given the specified assignment (assumes unlimited capacity caches, minimum granularity transfers, etc.)**

- **Artifactual communication: all other communication (artifactual communication results from practical details of system implementation)**

Artifactual communication examples

- System might have a minimum granularity of transfer (result: system must communicate more data than what is needed)
    - Program loads one 4-byte float value but entire 64-byte cache line must be transferred from memory (16x more communication than necessary)

- System might have rules of operation that result in unnecessary communication:
    - Program stores 16 consecutive 4-byte float values, so entire 64-byte cache line is loaded from memory, and then subsequently stored to memory (2x overhead)

- Poor allocation of data in distributed memories (data doesn't reside near processor that accesses it most)

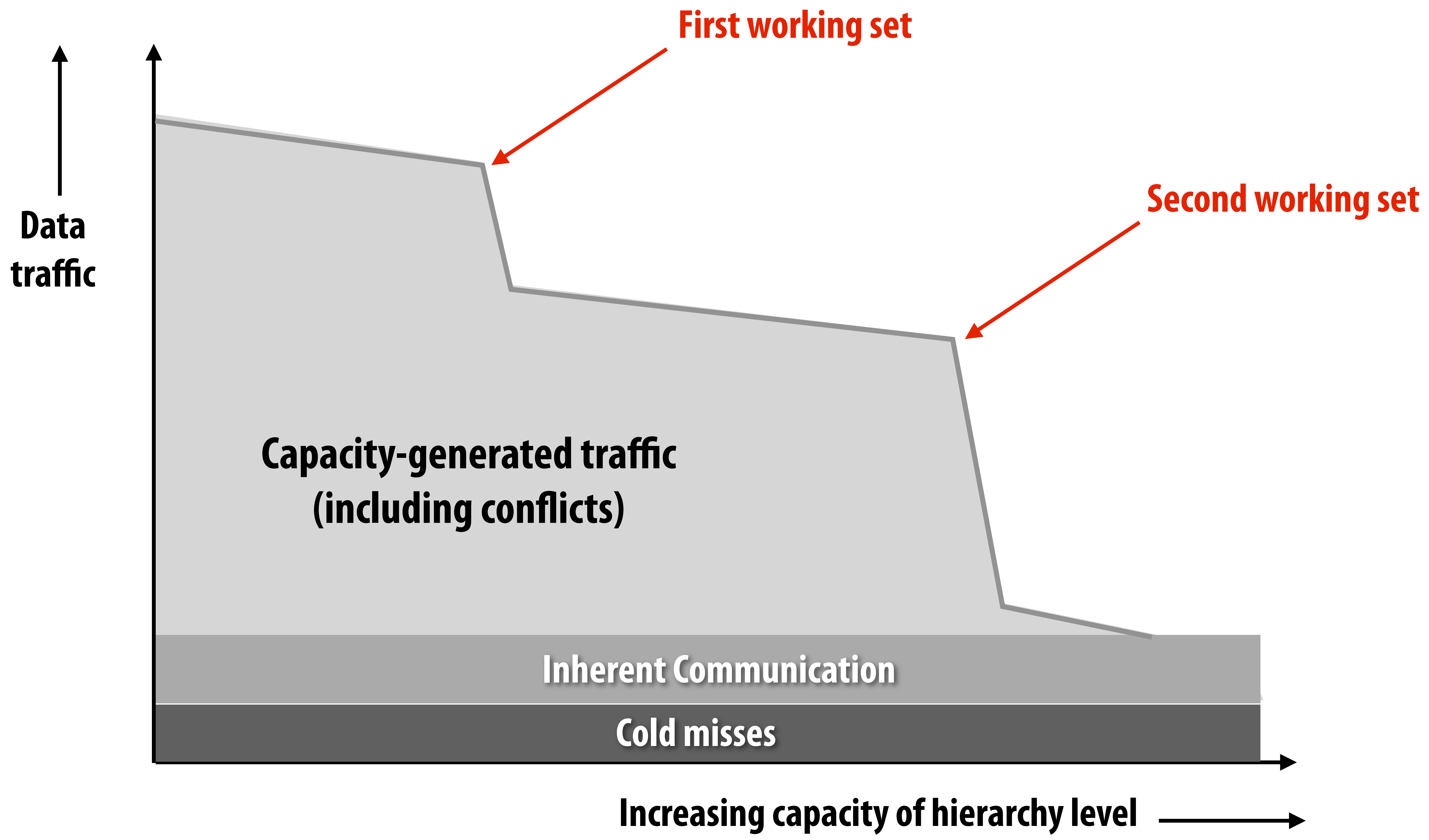
- Finite replication capacity (same data communicated to processor multiple times because cache is too small to retain it between accesses)

Review of the three (now four) Cs

You are expected to know this from 15-213!

- **Cold miss**
  First time data touched. Unavoidable in a sequential program.
- **Capacity miss**
  Working set is larger than cache. Can be avoided/reduced by increasing cache size.
- **Conflict miss**
  Miss induced by cache management policy. Can be avoided/reduced by changing cache associativity, or data access pattern in application.
- **Communication miss (new)**
  Due to inherent or artifactual communication in parallel system.

Communication: working set perspective

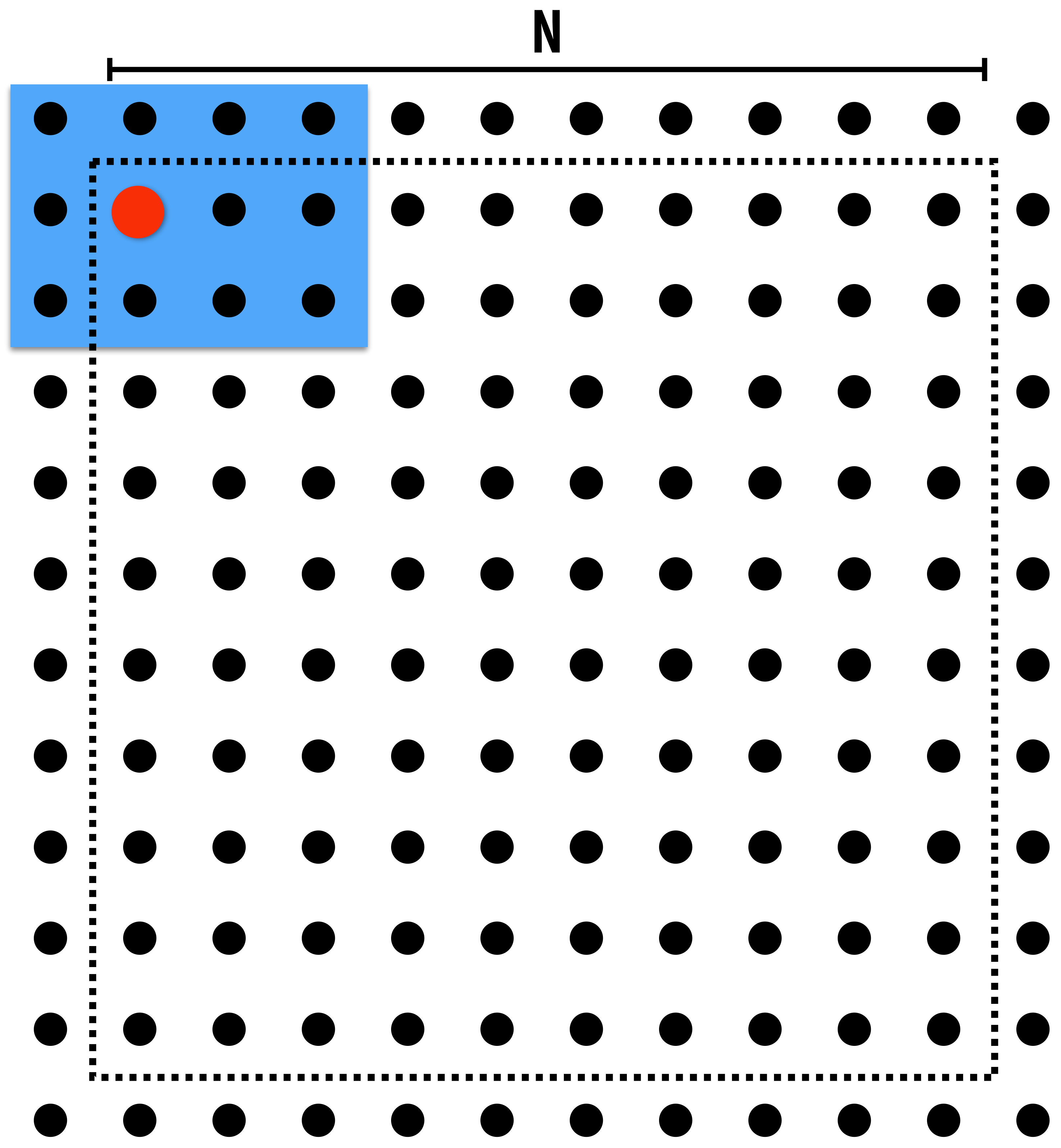


This diagram holds true at any level of the memory hierarchy in a parallel system

Question: how much capacity should an architect build for this workload?

More techniques for reducing communication

Data access in grid solver: row-major traversal



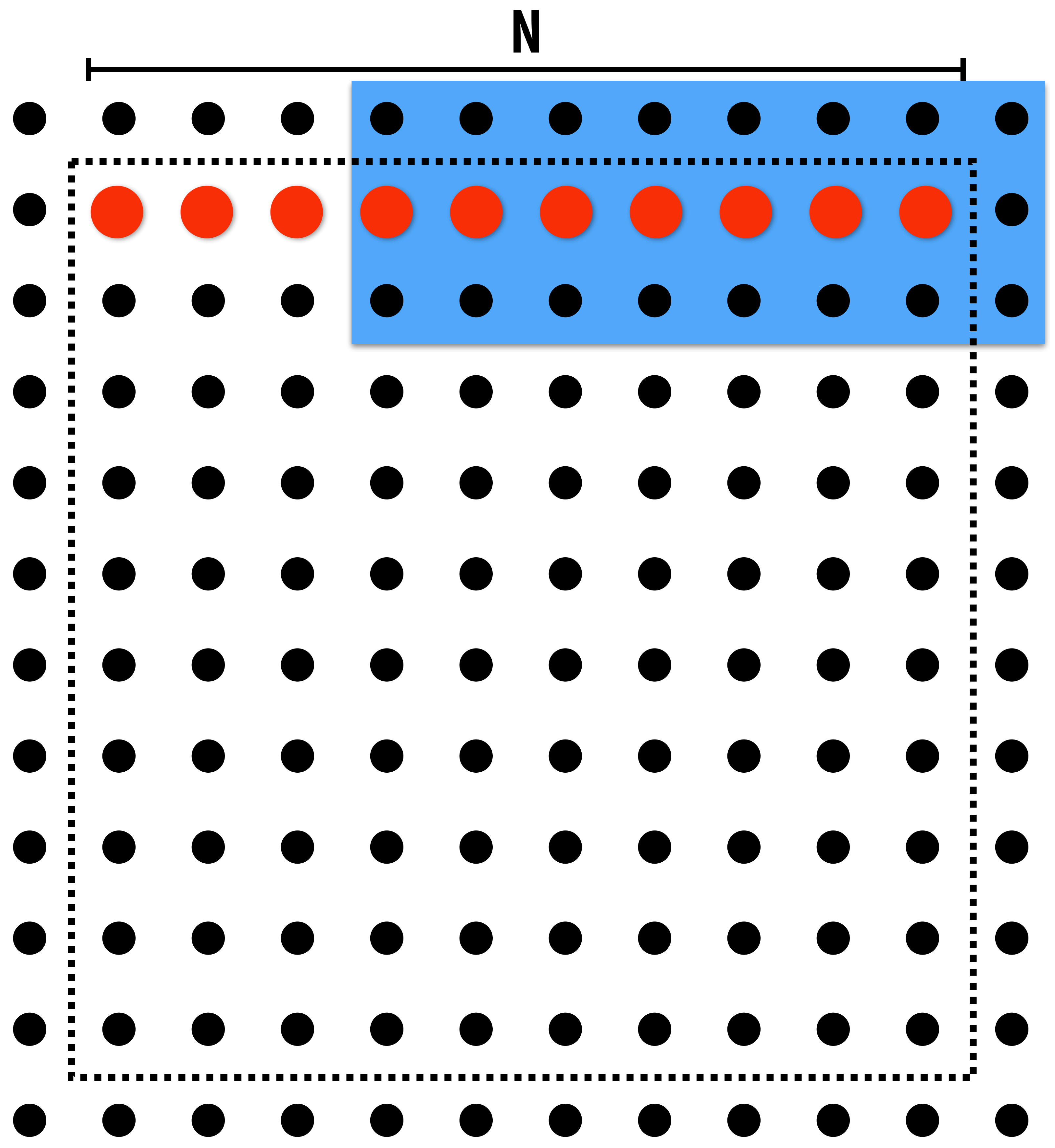
Assume row-major grid layout.

Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

Recall grid solver application.
Blue elements show data in cache after update to red element.

Data access in grid solver: row-major traversal



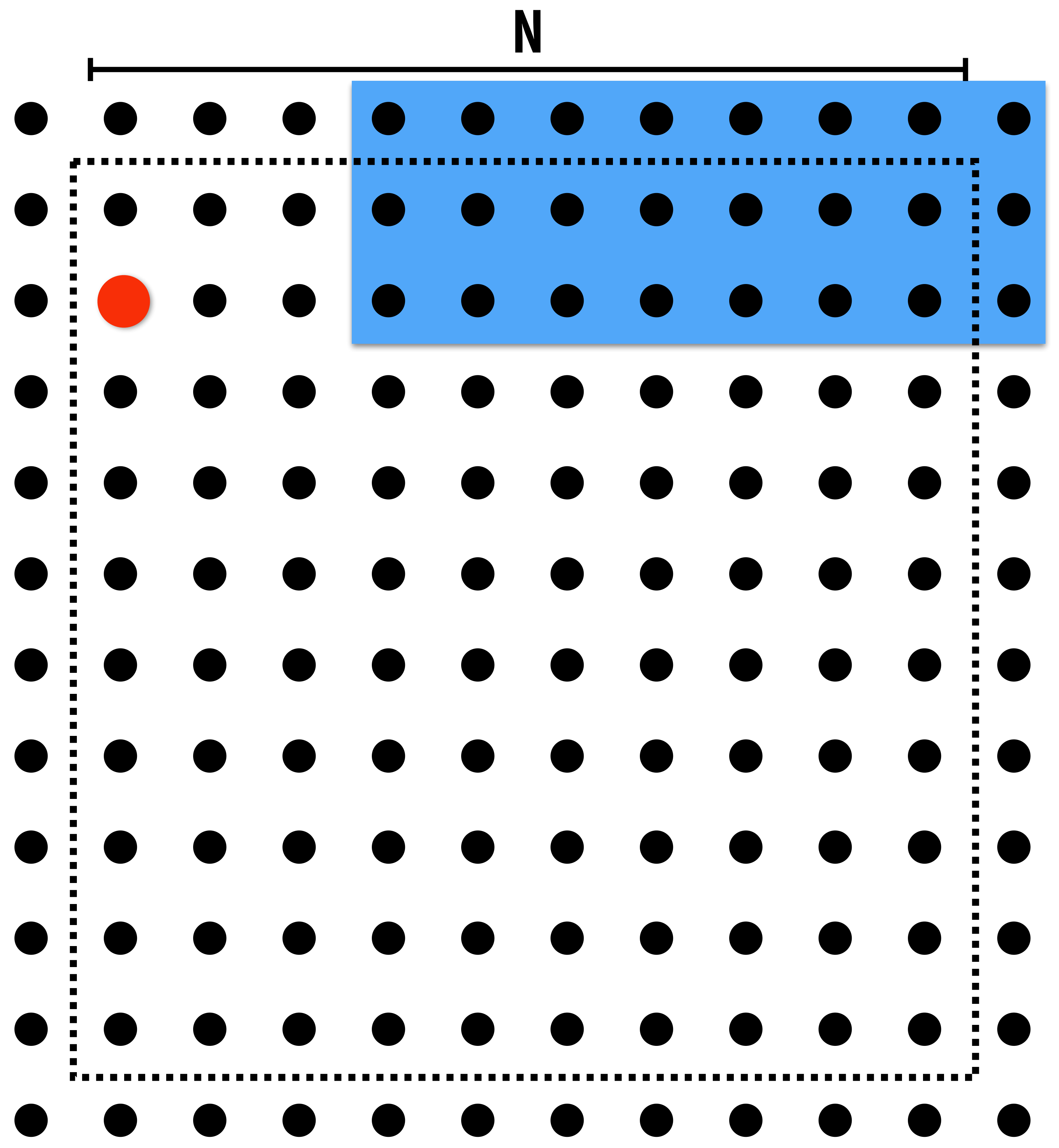
Assume row-major grid layout.

Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

Blue elements show data in cache at end of processing first row.

Problem with row-major traversal: long time between accesses to same data



Assume row-major grid layout.

Assume cache line is 4 grid elements.

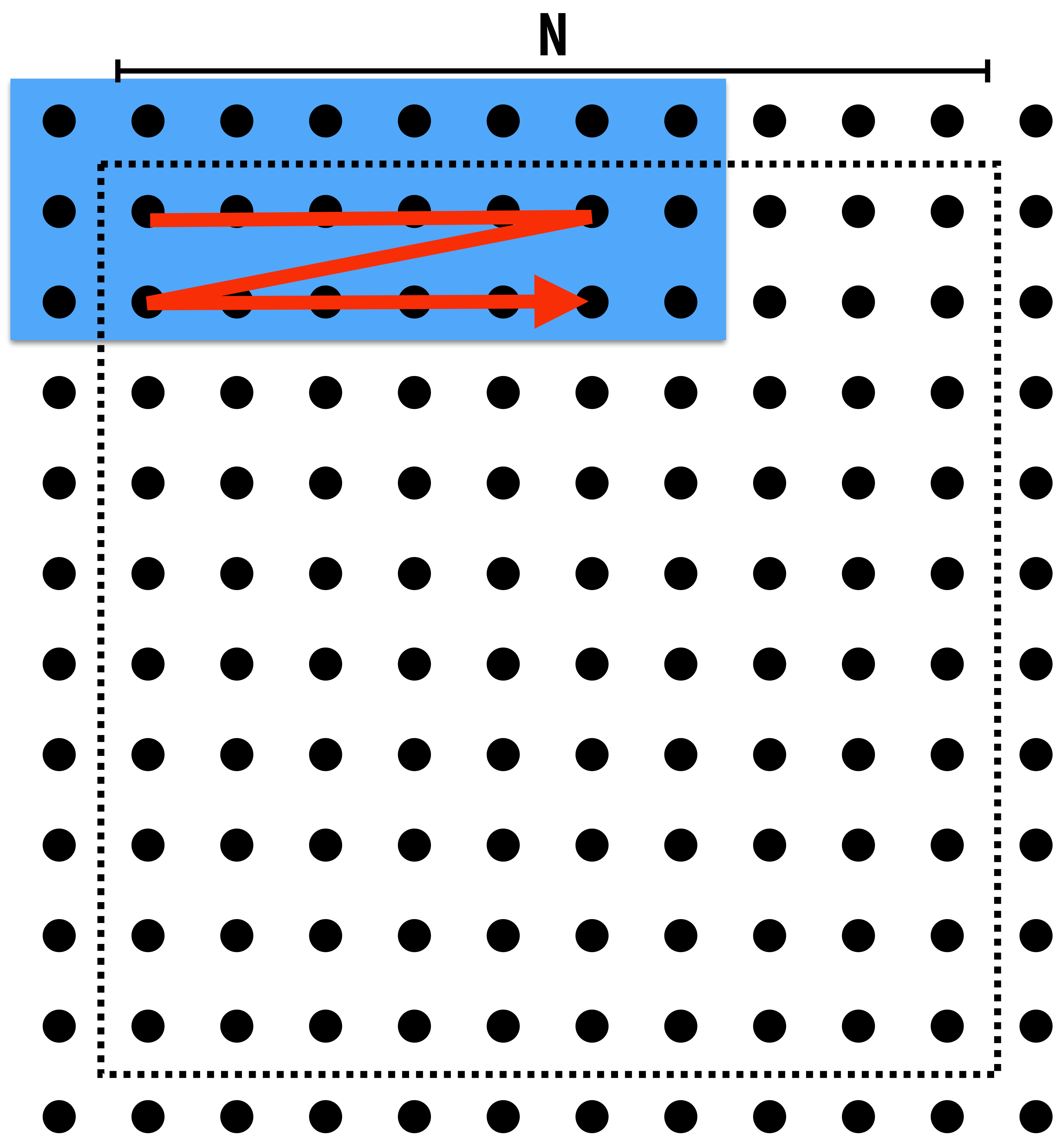
Cache capacity is 24 grid elements (6 lines)

Although elements (0,2) and (1,1) had been accessed previously, at start go processing row 2, they are no longer present in cache.

(What type of miss is this?)

On average, program loads one cache line per element processed.

Improve temporal locality by changing grid traversal order



Assume row-major grid layout.

Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

"Blocked" iteration order.
(recall cache lab in 15-213)

Improving temporal locality by fusing loops

```
void add(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] + B[i];
}

void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
       C[i] = A[i] * B[i];
}


float* A, *B, *C, *D, *tmp;

// assume arrays are allocated here

// compute D = (A + B) * C
add(n, A, B, tmp);
mult(n, tmp, C, D);
```

Two loads, one store per math op (arithmetic intensity = 1/3)

Two loads, one store per math op (arithmetic intensity = 1/3)

Overall arithmetic intensity = 1/3

```
void fused_muladd(int n, float* A, float* B, float* C, float* D) {
    for (int i=0; i<n; i++)
       D[i] = (A[i] + B[i]) * C[i];
}

// compute D = (A + B) * C
fused_muladd(n, A, B, C, D);
```

Three loads, one store per 2 math ops (arithmetic intensity = 1/2)

Code on top is more modular (e.g, array based math library)
Code on bottom performs better. Why?

Improve arithmetic intensity through sharing

- **Exploit sharing: co-locate tasks that operate on the same data**
  - Schedule threads working on the same data structure at the same time on the same processor
  - Reduces inherent communication

- **Example: CUDA thread block**
  - Abstraction used to localize related processing in a CUDA program
  - Threads in block often cooperate to perform an operation (leverage fast access to / synchronization via CUDA shared memory)
  - So GPU implementations always schedule threads from the same block on the same GPU core

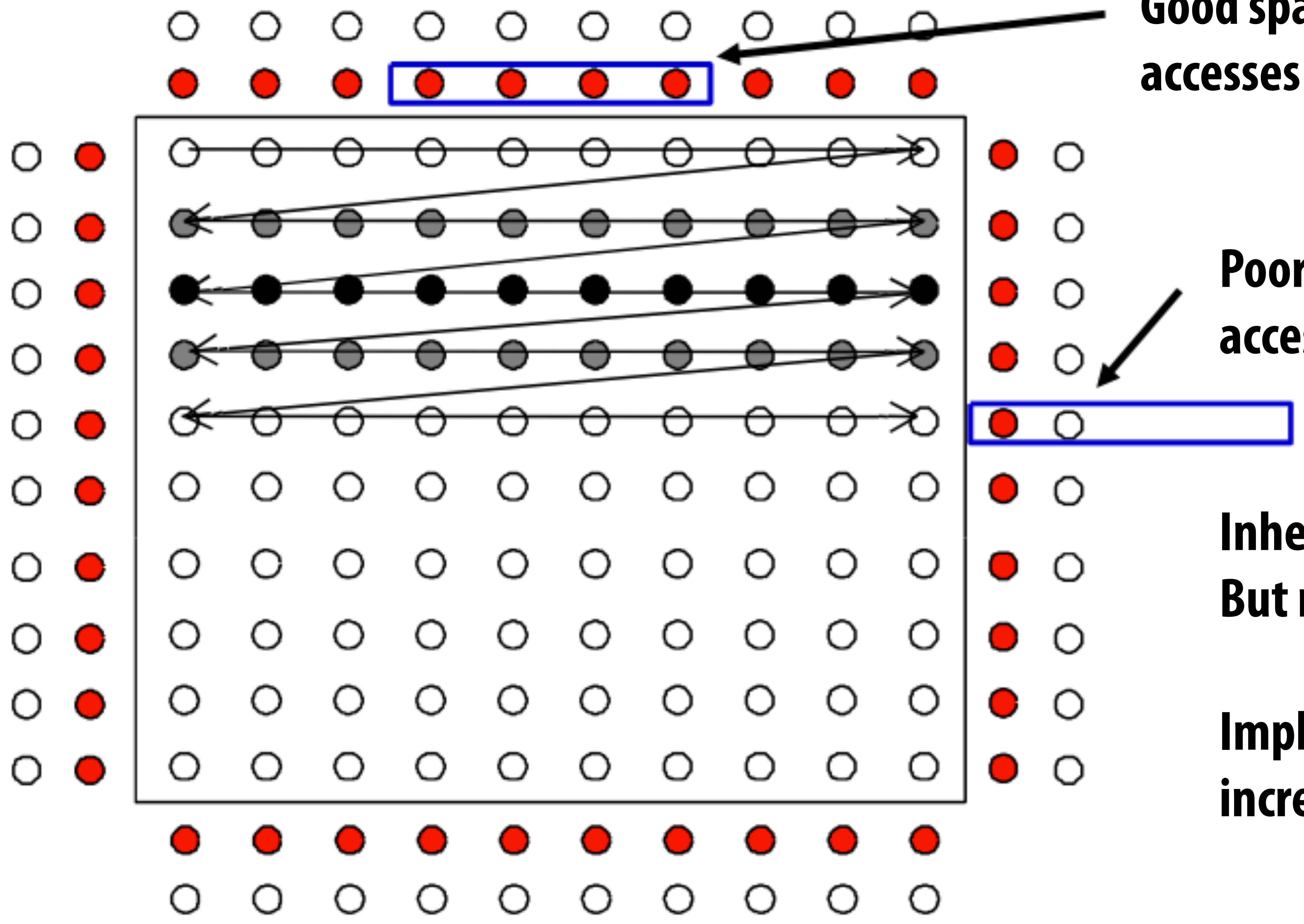
Exploiting spatial locality

- Granularity of communication can be very important
  - Granularity of communication / data transfer
  - Granularity of coherence (future lecture)

Artifactual communication due to comm. granularity

2D blocked assignment of data to processors as described previously.

Assume: cache line communication granularity (line contains four elements)



Good spatial locality for non-local accesses to top-bottom rows

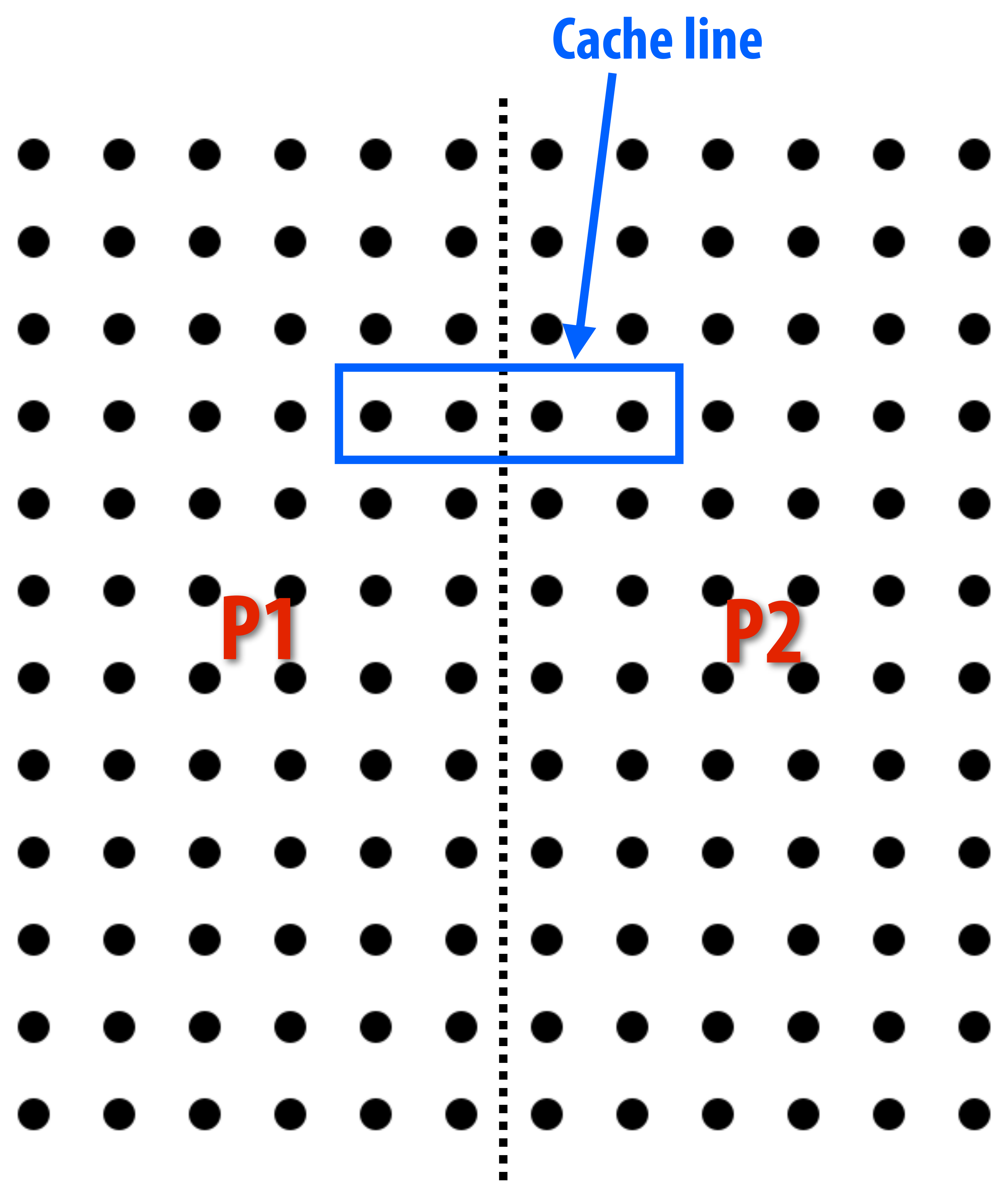
Poor spatial locality for non-local accesses to left-right columns

Inherently need one element from line. But must communicate four.

Implication: artifactual communication increases with cache line size!

● = required elements assigned to other processors

Artifactual communication due to cache line communication granularity



Data partitioned in half to column. Partitions assigned to threads running on P1 and P2

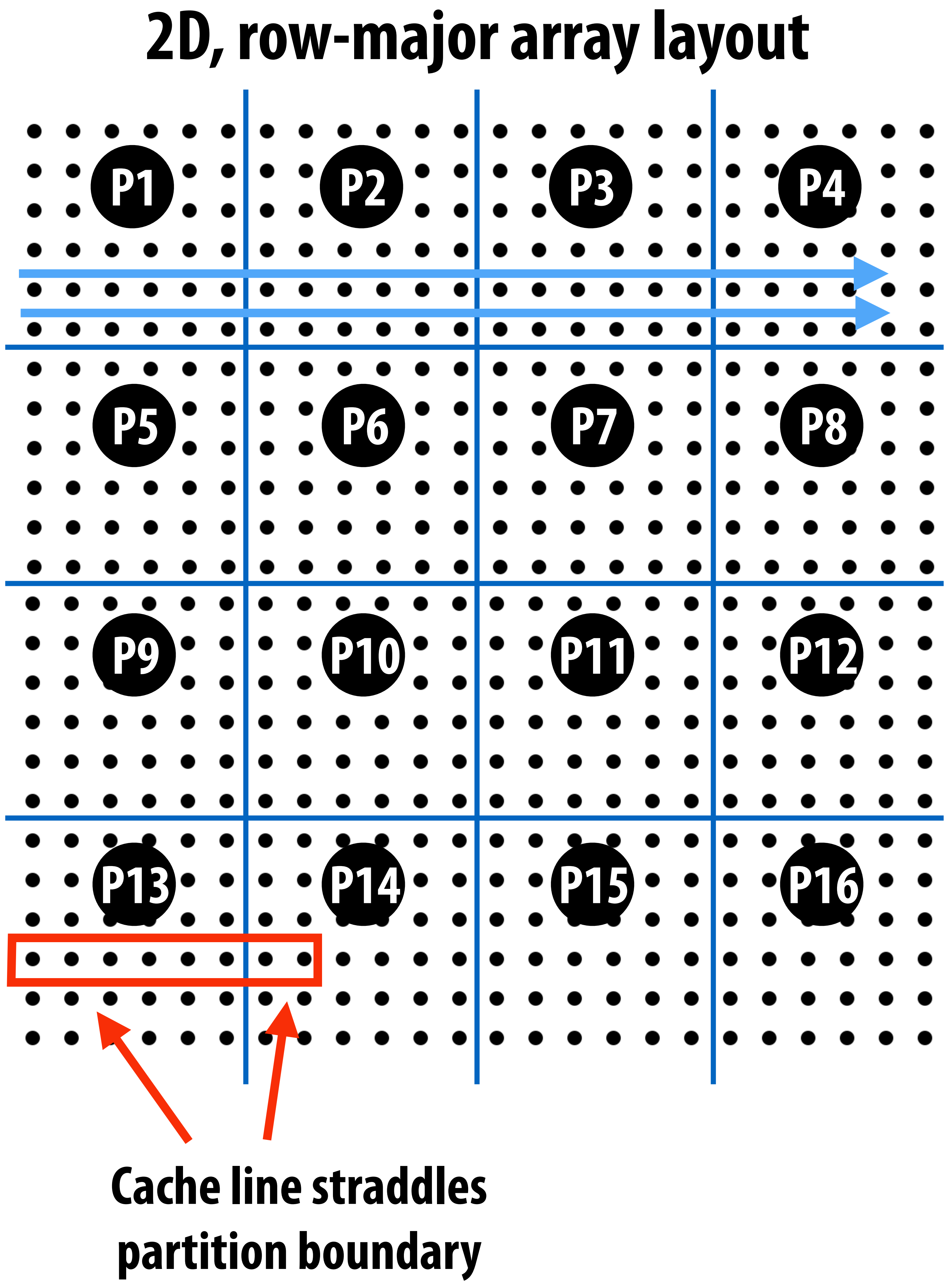
Threads access their assigned elements (no inherent communication exists)

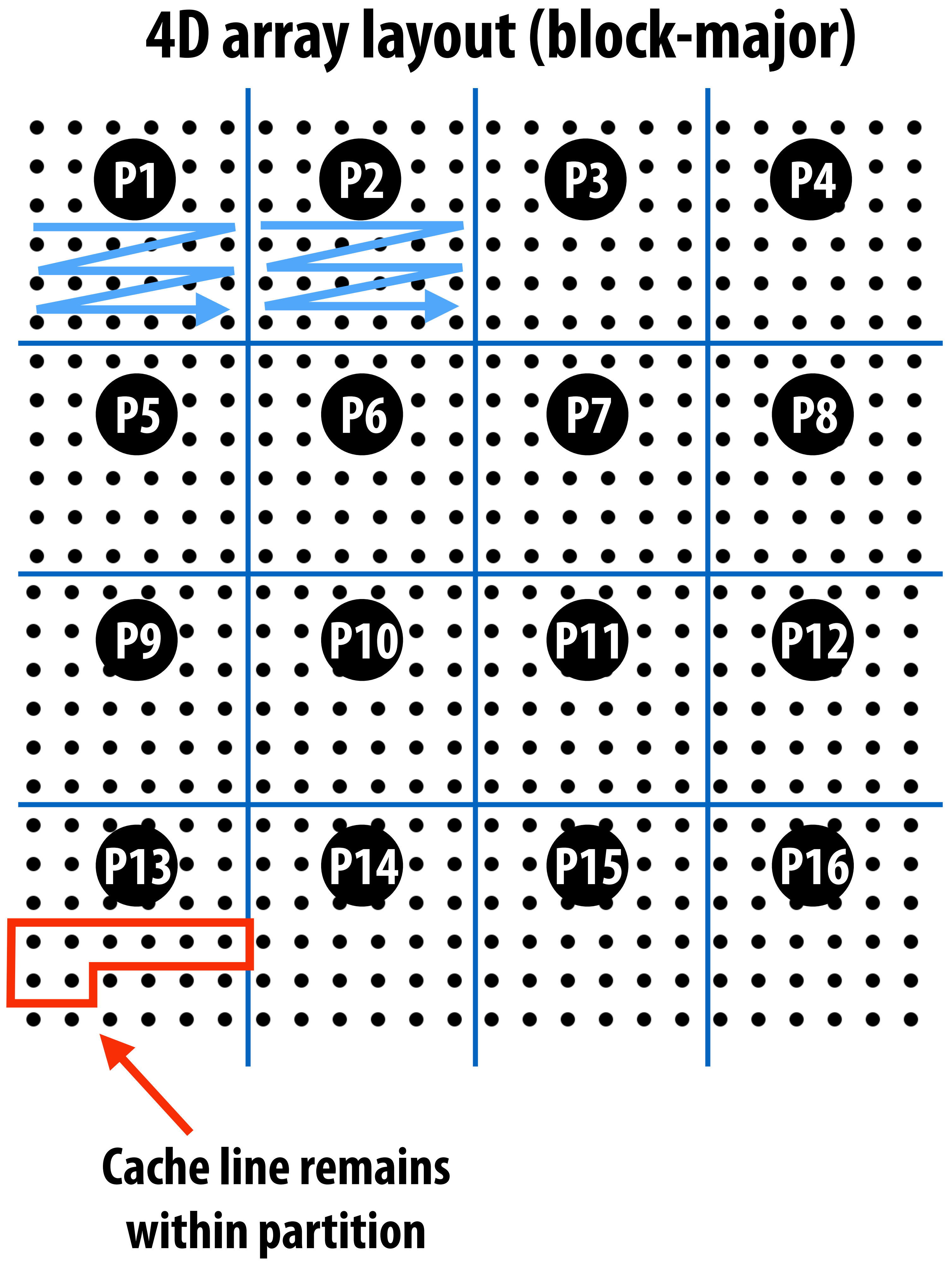
But data access on real machine triggers (artifactual) communication due to the cache line being written to by both processors *

* further detail in the upcoming cache coherence lectures

Reducing artifactual comm: blocked data layout

(Blue lines indicate consecutive memory addresses)


2D, row-major array layout
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
P12
P13
P14
P15
P16
Cache line straddles partition boundary


4D array layout (block-major)
P1
P2
P3
P4
P5
P6
P7
P8
P9
P10
P11
P12
P13
P14
P15
P16
Cache line remains within partition

Note: make sure you are clear about the difference between blocked assignment of work (true in both cases above), and blocked day layout (only at right)

Structuring communication to reduce cost

Total communication cost = num_messages x (message communication time - overlap)

Total communication cost = num_messages x (overhead + occupancy + network delay - overlap)

Total communication cost = num_messages x (overhead + (T_0 + n/B + contention) + network delay - overlap)

Occupancy consists of the time it takes to transfer a message (n bytes) over slowest link + delays due to contention for link

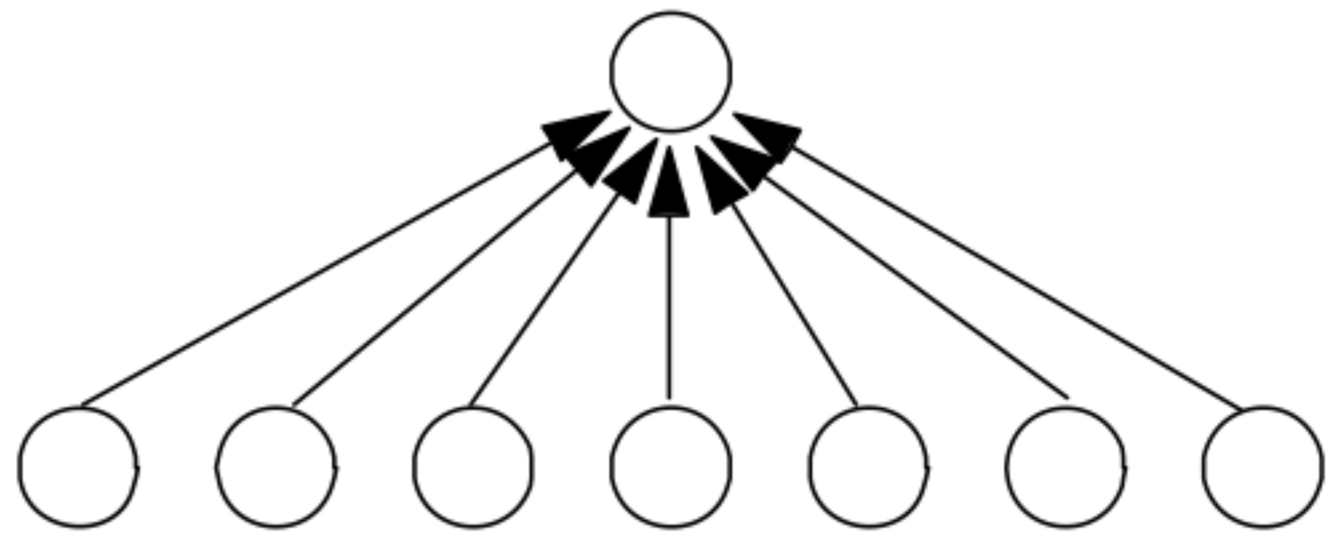
Demo: contention

Contention

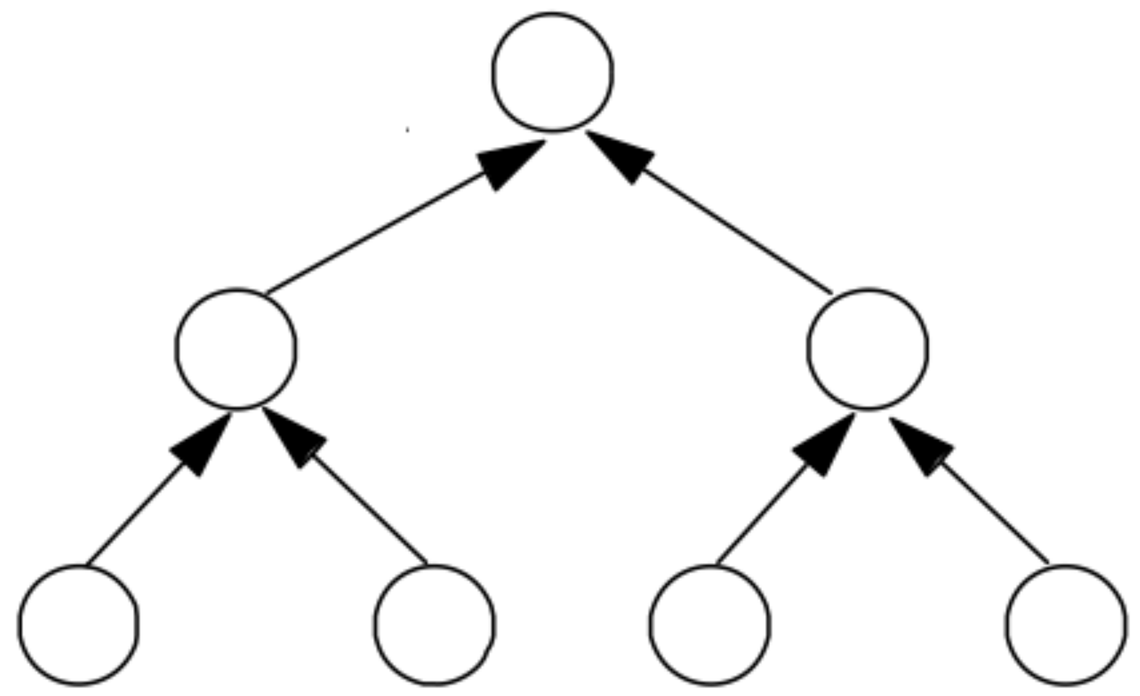
- A resource can perform operations at a given throughput (number of transactions per unit time)
  - Memory, communication links, servers, etc.

- Contention occurs when many requests to a resource are made within a small window of time (the resource is a “hot spot”)

Example: updating a shared variable



Flat communication:
potential for high contention
(but low latency if no contention)

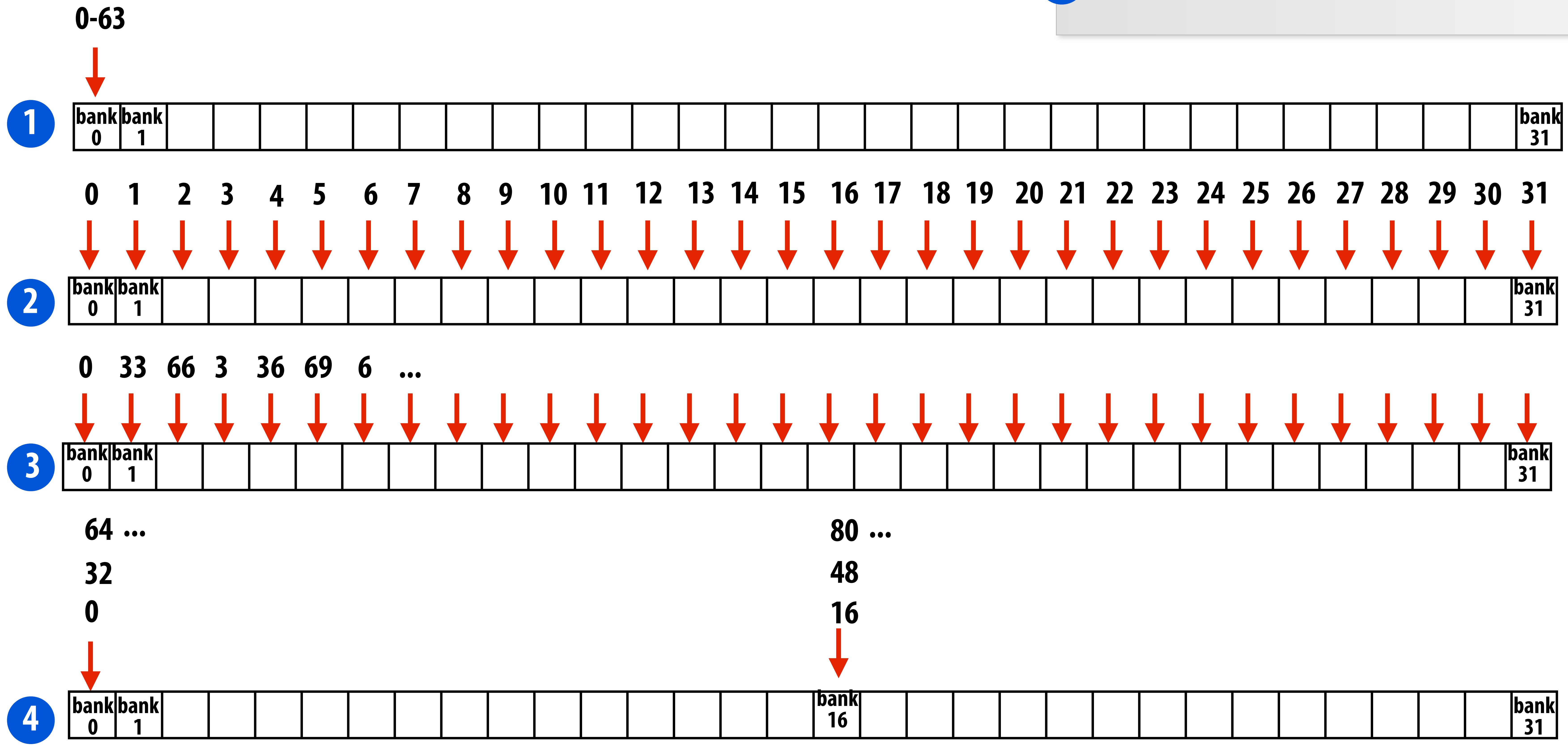


Tree structured communication:
reduces contention
(but higher latency under no contention)

Contention example: NVIDIA GTX 480 shared memory

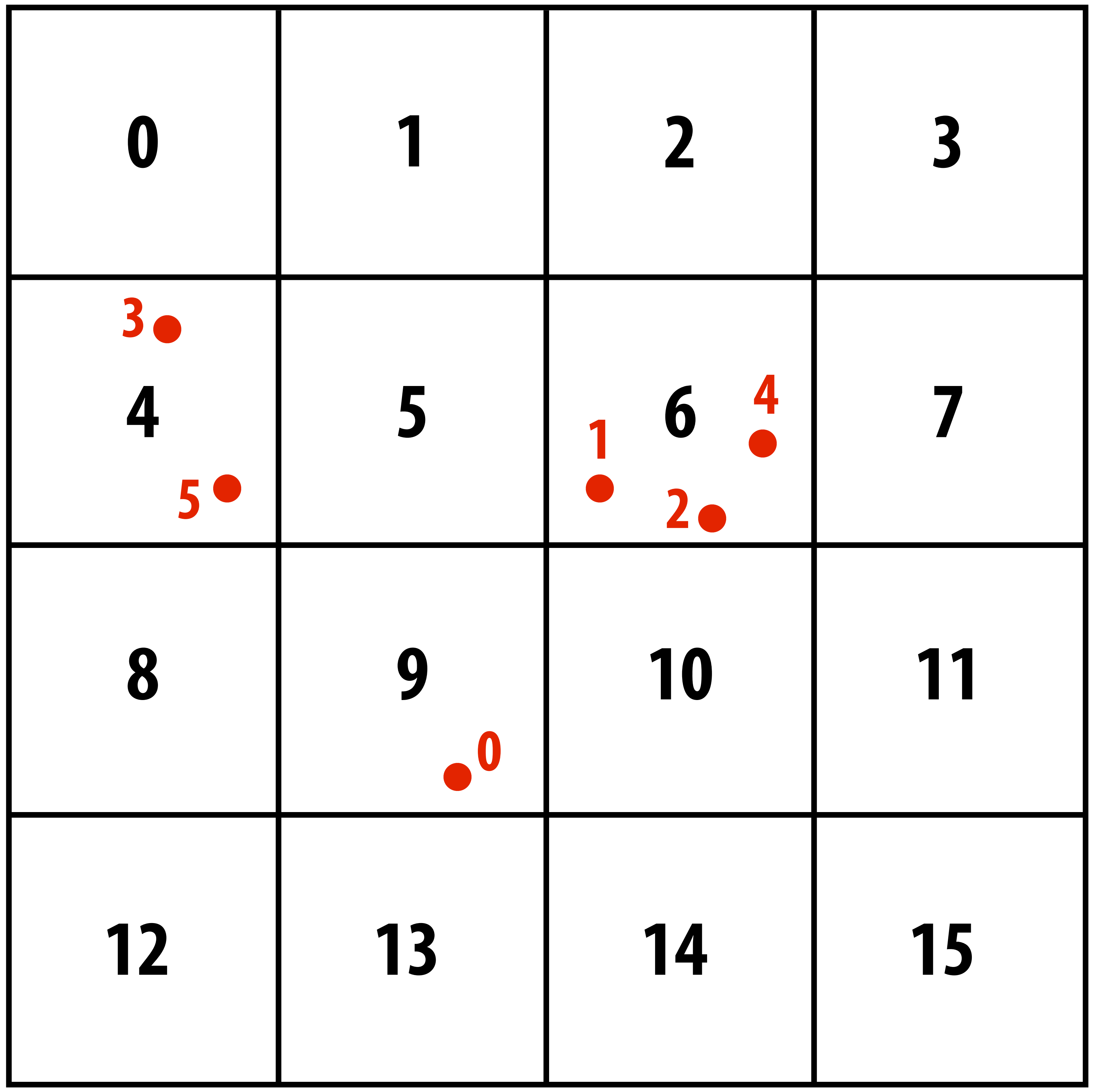
- Shared memory implementation
  - On-chip indexable storage, physically partitioned into 32 banks
  - Address X is stored in bank B, where B = X % 32
  - Each bank can be accessed in parallel (one word per bank per clock)
  - Special broadcast mode if all addresses are the same (example 1)
- Figure shows addresses requested from each bank as a result of shared memory load instruction from 32 threads in a WARP

```
__shared__ float A[512];

int index = threadIdx.x;

1  float x1 = A[0];           // single cycle

2  float x2 = A[index];       // single cycle

3  float x3 = A[3*index];     // single cycle

4  float x4 = A[16 * index]; // 16 cycles
```



Contention example: particle data structure on GPU

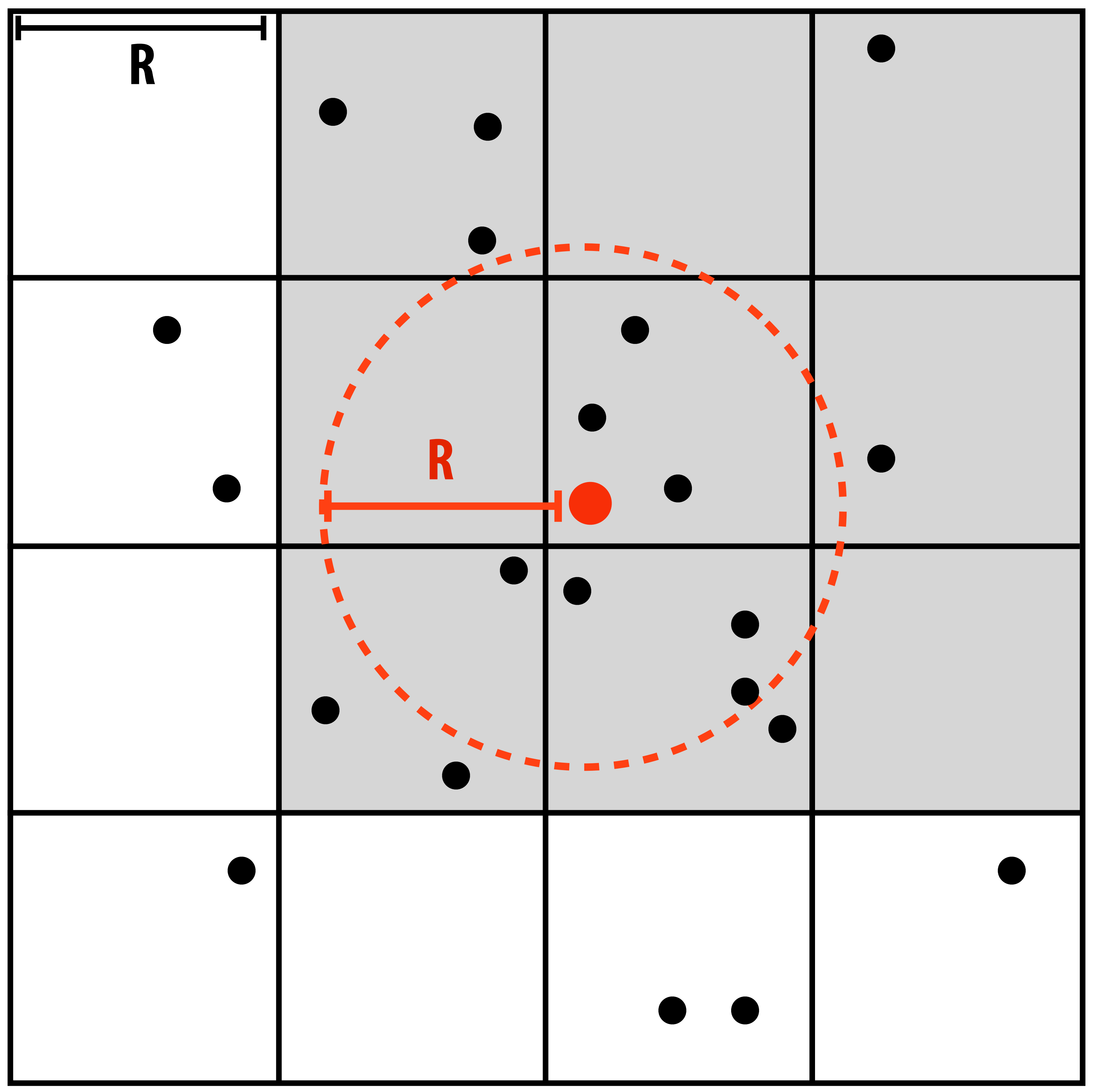
- Recall: Up to 2048 CUDA threads per SMX core on a GTX 680 GPU (8 SMX cores)
- Problem: place 100K point particles in a 16-cell uniform grid based on 2D position
  - Parallel data structure manipulation problem: build a 2D array of lists



Cell id	Count	Particle id
0	0	
1	0	
2	0	
3	0	
4	2	3, 5
5	0	
6	3	1, 2, 4
7	0	
8	0	
9	1	0
10	0	
11	0	
12	0	
13	0	
14	0	
15	0	

Common use of this structure: N-body problems

- A common operation is to compute interactions with neighboring particles
- Example: given particle, find all particles within radius R
  - Create grid with cells of size R
  - Only need to inspect particles in surrounding grid cells



Solution 1: parallelize over cells

- **One possible answer is to decompose work by cells: for each cell, independently compute particles within it (eliminates contention because no synchronization is required)**
    - **Insufficient parallelism: only 16 parallel tasks, but need thousands of independent tasks for GPU)**
    - **Work inefficient: performs 16 times more particle-in-cell computations than sequential algorithm**

```
list cell_lists[16];      // 2D array of lists

for each cell c           // in parallel
   for each particle p    // sequentially
       if (p is within c)
          append p to cell_lists[c]
```

Solution 2: parallelize over particles

- **Another answer: assign one particle to each CUDA thread. Thread computes cell containing particle, then atomically updates list.**
  - Massive contention: thousands of threads contending for access to update single shared data structure

```
list cell_list[16];    // 2D array of lists
lock cell_list_lock;

for each particle p          // in parallel
   c = compute cell containing p
   lock(cell_list_lock)
   append p to cell_list[c]
   unlock(cell_list_lock)
```

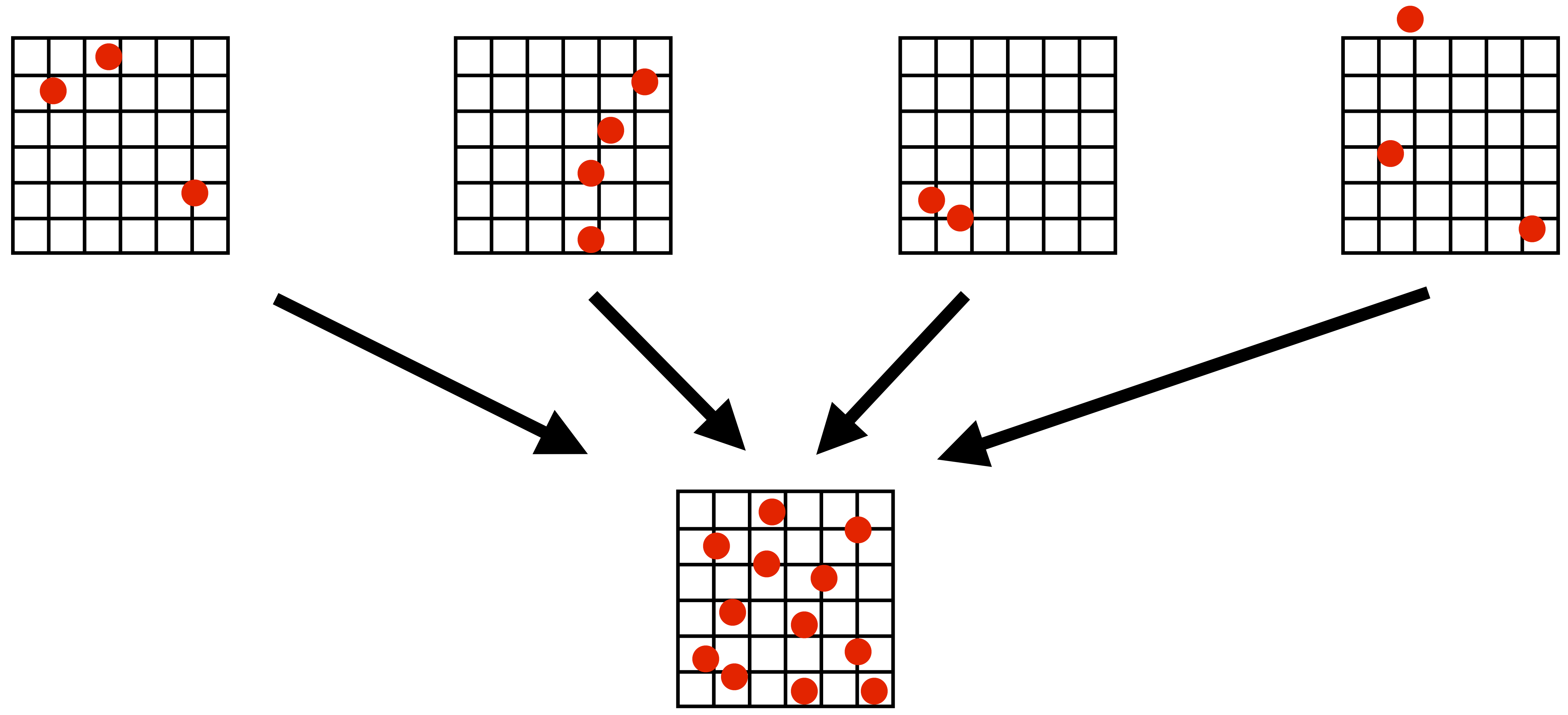
Solution 3: use finer-granularity locks

- Alleviate contention for single global lock by using per-cell locks
  - Assuming uniform distribution of particles in 2D space... ~16x less contention than solution 2

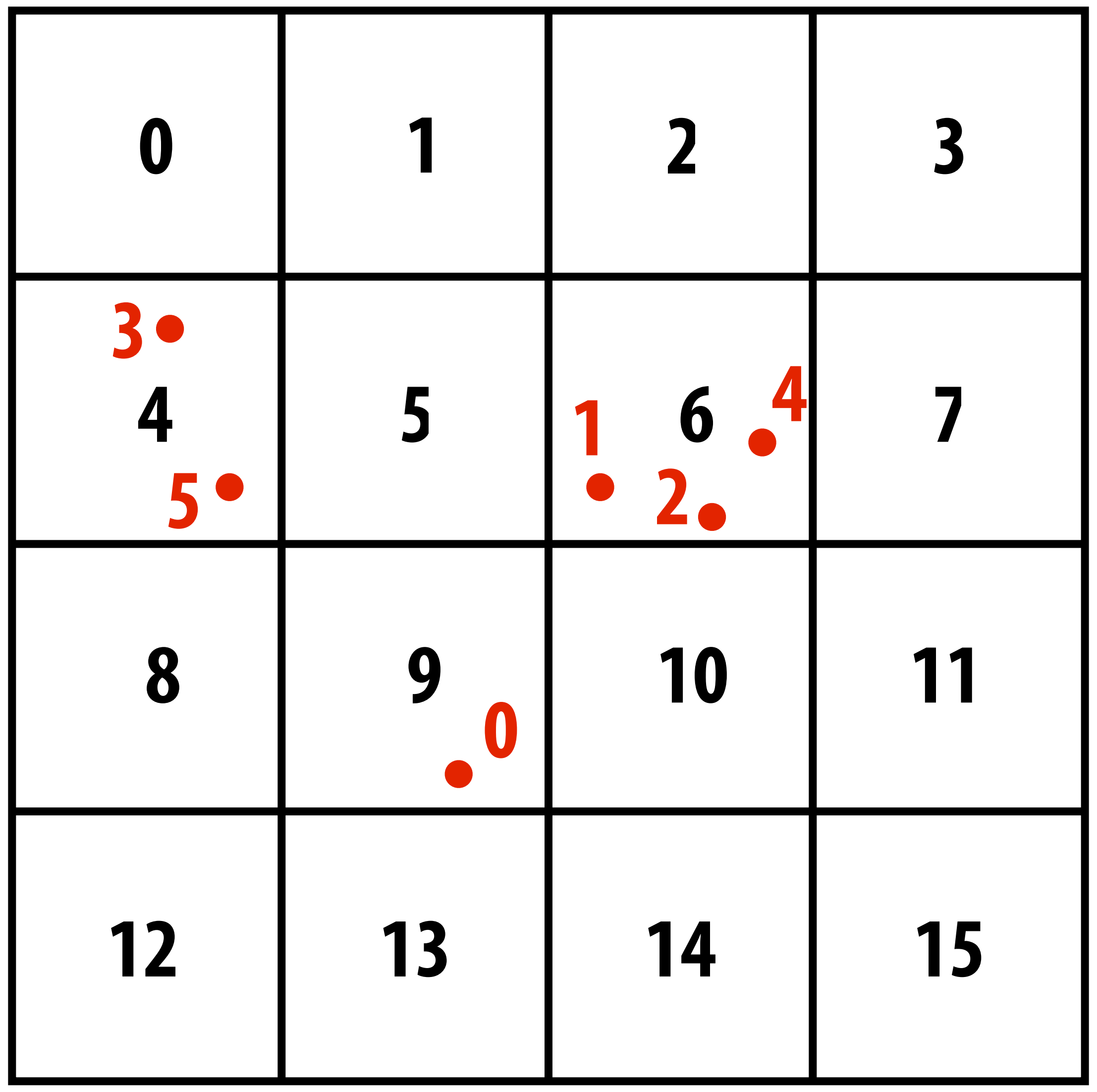
```
list cell_list[16];    // 2D array of lists
lock cell_list_lock[16];

for each particle p           // in parallel
   c = compute cell containing p
   lock(cell_list_lock[c])
   append p to cell_list[c]
   unlock(cell_list_lock[c])
```

Solution 4: compute partial results + merge

- Yet another answer: generate N "partial" grids in parallel, then combine
  - Example: create N thread blocks (at least as many thread blocks as SMX cores)
  - All threads in thread block update same grid
    - Enables faster synchronization: contention reduced by factor of N and also cost of synchronization is lower because it is performed on block-local variables (in CUDA shared memory)
  - Requires extra work: merging the N grids at the end of the computation
  - Requires extra memory footprint: Store N grids of lists, rather than 1



Solution 5: data-parallel approach



Step 1: compute cell containing each particle (parallel across particles)

Particle Index:	0	1	2	3	4	5
Grid_index:	9	6	6	4	6	4

Step 2: sort results by cell (particle index array permuted based on sort)

Particle Index:	3	5	1	2	4	0
Grid_index:	4	4	6	6	6	9

This solution removes the need for fine-grained synchronization... at cost of a sort and extra passes over the data (extra BW)

Step 3: find start/end of each cell (parallel across array elements)

```
cell = grid_index[index]
if (index == 0 || cell != grid_index[index-1]) {
    cell_starts[cell] = index;
    if (index > 0)              // if not first, set cell_ends
       cell_ends[grid_index[index-1]] = index;
}
if (index == numParticles-1) // special case for last cell
    cell_ends[cell] = index+1;
```

This code is run for each element of array

	0	1	2	3	4	5	6	7	8	9	10	
cell_starts	0xff	0xff	0xff	0xff	0	0xff	2	0xff	0xff	5	0xff	• • •
cell_ends	0xff	0xff	0xff	0xff	2	0xff	5	0xff	0xff	6	0xff	• • •

Reducing communication costs

- **Reduce overhead to sender/receiver**
  - Send fewer messages, make messages larger (amortize overhead)
  - Coalesce many small messages into large ones
- **Reduce delay**
  - Application writer: restructure code to exploit locality
  - HW implementor: improve communication architecture
- **Reduce contention**
  - Replicate contended resources (e.g., local copies, fine-grained locks)
  - Stagger access to contended resources
- **Increase overlap**
  - Application writer: use asynchronous communication (e.g., async messages)
  - HW implementor: pipelining, multi-threading, pre-fetching, out-of-order exec
  - Requires additional concurrency in application (more concurrency than number of execution units)

Summary: optimizing communication

- Inherent vs. artifactual communication
  - Inherent communication is fundamental give how the problem is decomposed and how work is assigned
  - Artifactual communication depends on the machine (often as important to performance as inherent communication)

- Improving program performance:
  - Identify and exploit locality: communicate less (increase arithmetic intensity)
  - Reduce overhead (fewer, large messages)
  - Reduce contention
  - Maximize overlap of communication and processing (hide latency so as to not incur cost)