Lecture 7:

Performance Optimization Part II: Locality, Communication, and Contention

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2020

Today: more parallel program optimization

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

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

Message-passing solver review
# (since it makes communication explicit)

Message passing model: each thread operates in its own address space

Thread 1 Address Space

Thread 2 Address Space

Thread 3 Address Space

Thread 4 Address Space
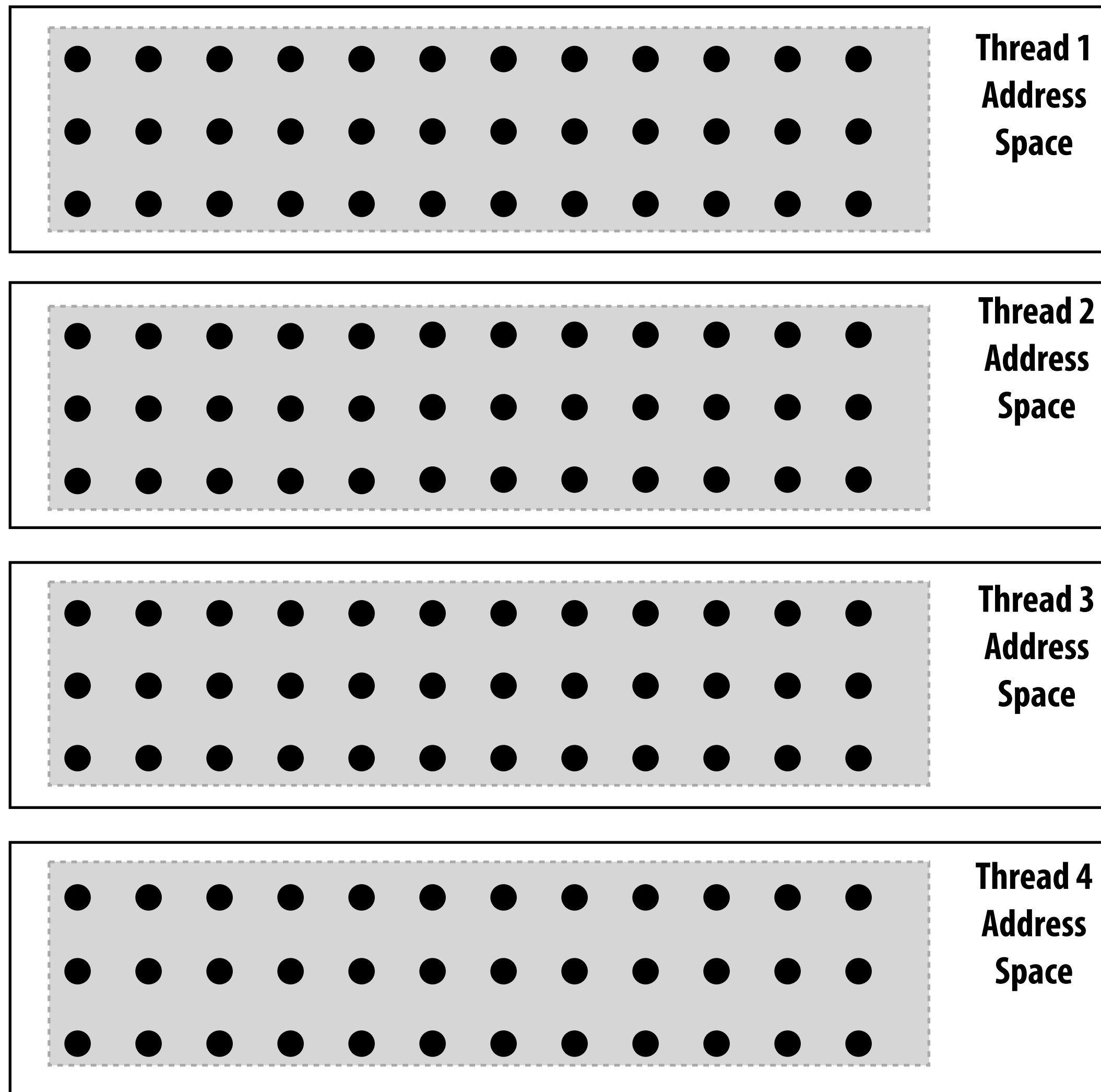
In this figure: four threads

The grid data is partitioned into four allocations, each residing in one of the four unique thread address spaces

(four per-thread private arrays)

Data replication is now required to correctly execute the program
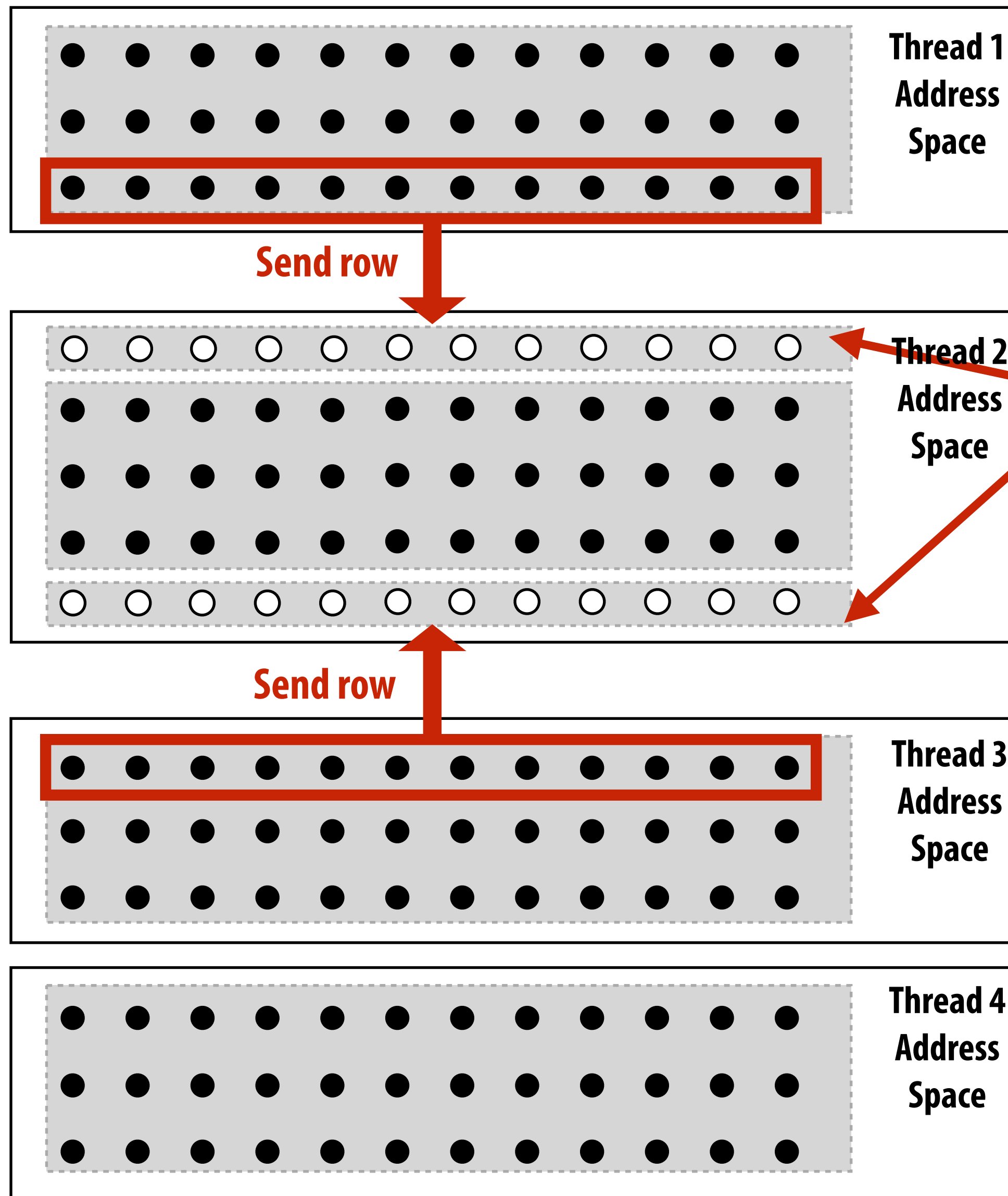
Example:

After red cell processing is complete, thread 1 and thread 3 send row of data to thread 2
(thread 2 requires up-to-date red cell information to update black cells in the next phase)

"Ghost cells" are grid cells replicated from a remote address space. It's common to say that information in ghost cells is "owned" by other threads.

Thread 2 logic:

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

int tid = get_thread_id();
int bytes = sizeof(float) * (N+2);

// receive ghost row cells (white dots)
recv(&local_data[0,0], bytes, tid-1);
recv(&local_data[rows_per_thread+1,0], bytes, tid+1);

// Thread 2 now has data necessary to perform
// future computation
```

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);
    }
  }
}
```
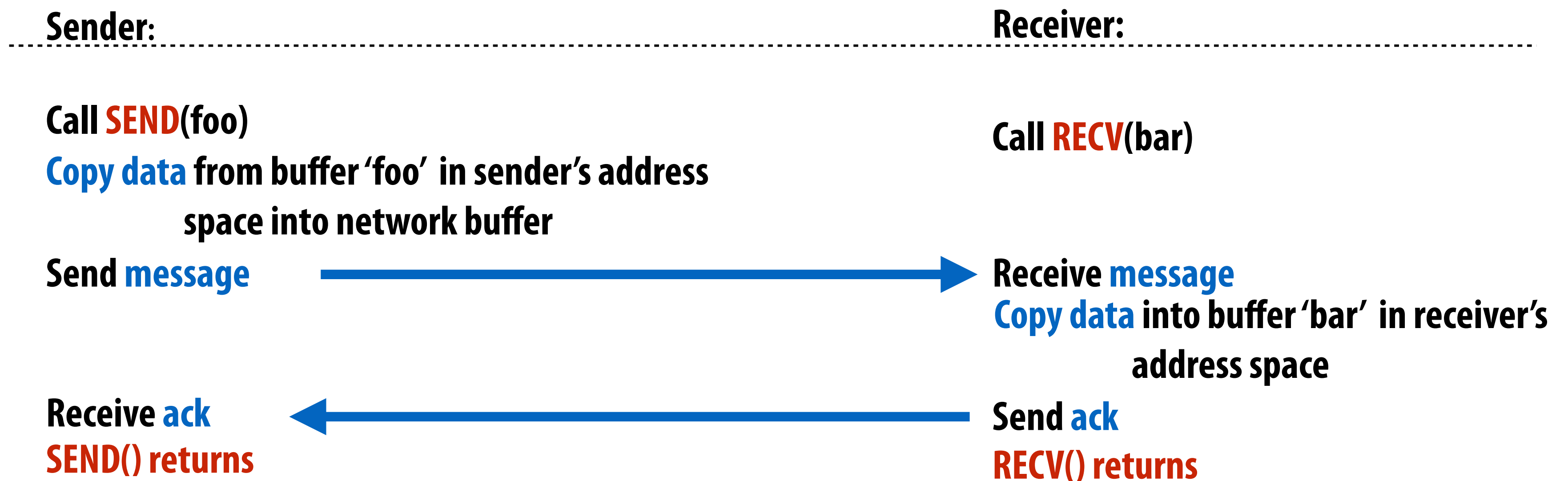
Send and receive ghost rows to "neighbor threads"

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

Synchronous (blocking) send and receive

- **send**(): call returns when sender receives acknowledgement that message data resides in address space of receiver

- **recv**(): call returns when data from received message is copied into address space of receiver and acknowledgement sent back to sender

Sender:

Receiver:

Call **SEND**(foo)
Copy data from buffer 'foo' in sender's address
      space into network buffer

Call **RECV**(bar)

Send **message** ──────────────▶ Receive **message**
Copy data into buffer 'bar' in receiver's
      address space

Receive **ack** ◀────────────── Send **ack**
SEND() returns
RECV() returns

As implemented on the prior slide, there is a big problem with our message passing solver if it uses synchronous send/recv!

Why?

How can we fix it?

(while still using synchronous send/recv)

Message passing solver (fixed to avoid deadlock)

```
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 % 2 == 0) {
      sendDown(); recvDown();
      sendUp();   recvUp();
    } else {
      recvUp();   sendUp();
      recvDown(); sendDown();
    }

    …

  }
}
```
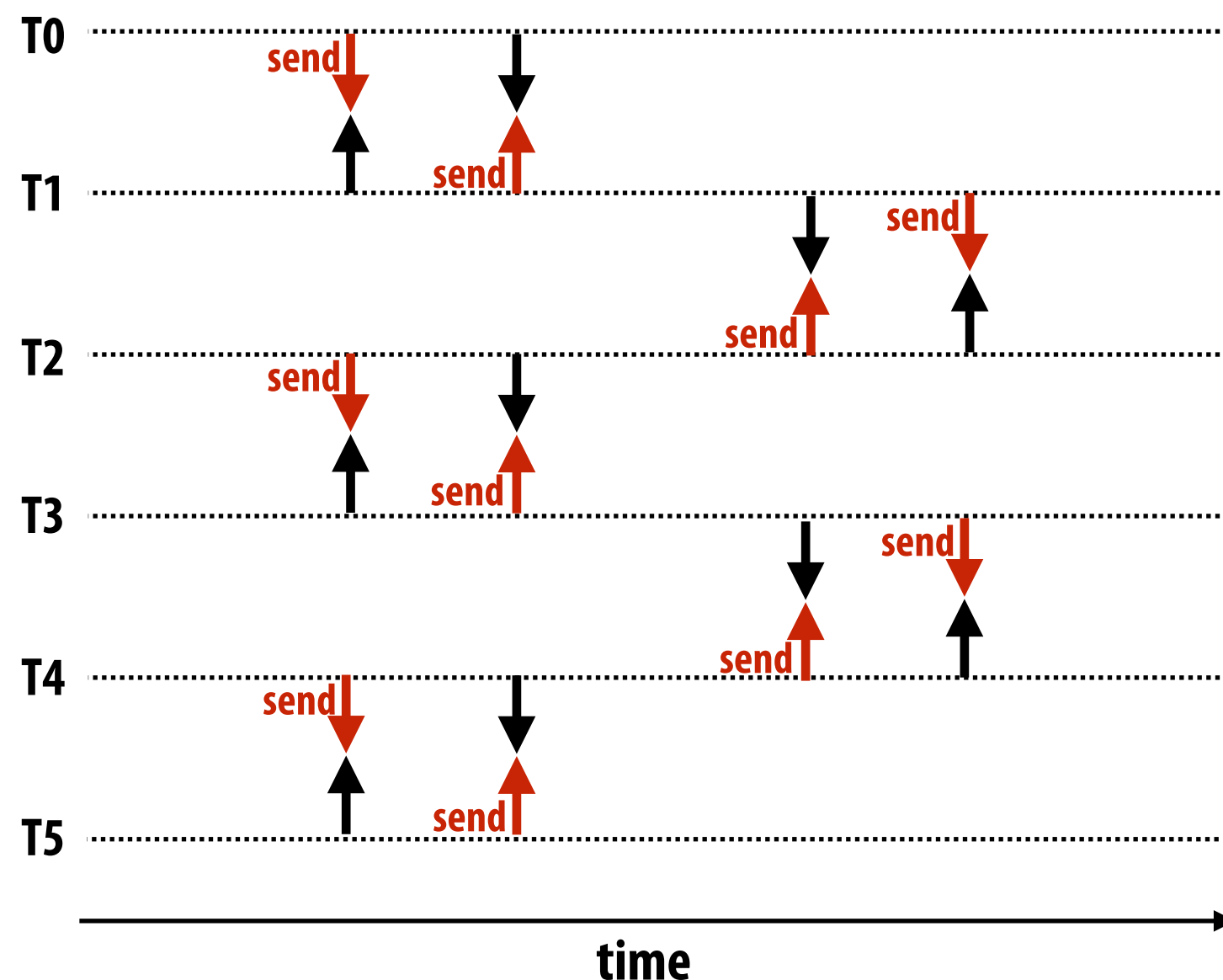
Send and receive ghost rows to "neighbor threads"
Even-numbered threads send, then receive
Odd-numbered thread recv, then send

time

Non-blocking asynchronous send/recv

- **send**(): call **returns immediately**

    - Buffer provided to send() cannot be modified by calling thread since message processing occurs concurrently with thread execution
    - Calling thread can perform other work while waiting for message to be sent

- **recv**(): posts intent to receive in the future, **returns immediately**

    - Use checksend(), checkrecv() to determine actual status of send/receipt
    - Calling thread can perform other work while waiting for message to be received

Sender:

Receiver:

Call SEND(foo)

SEND returns handle h1

Call RECV(bar)

RECV(bar) returns handle h2

Copy data from 'foo' into network buffer

Send message ———————————————————————→ Receive message

Messaging library copies data into 'bar'

Call **CHECKSEND**(h1)   // if message sent, now safe for thread to modify 'foo'

Call **CHECKRECV**(h2)

// if received, now safe for thread

// to access 'bar'

RED TEXT = executes concurrently with application thread
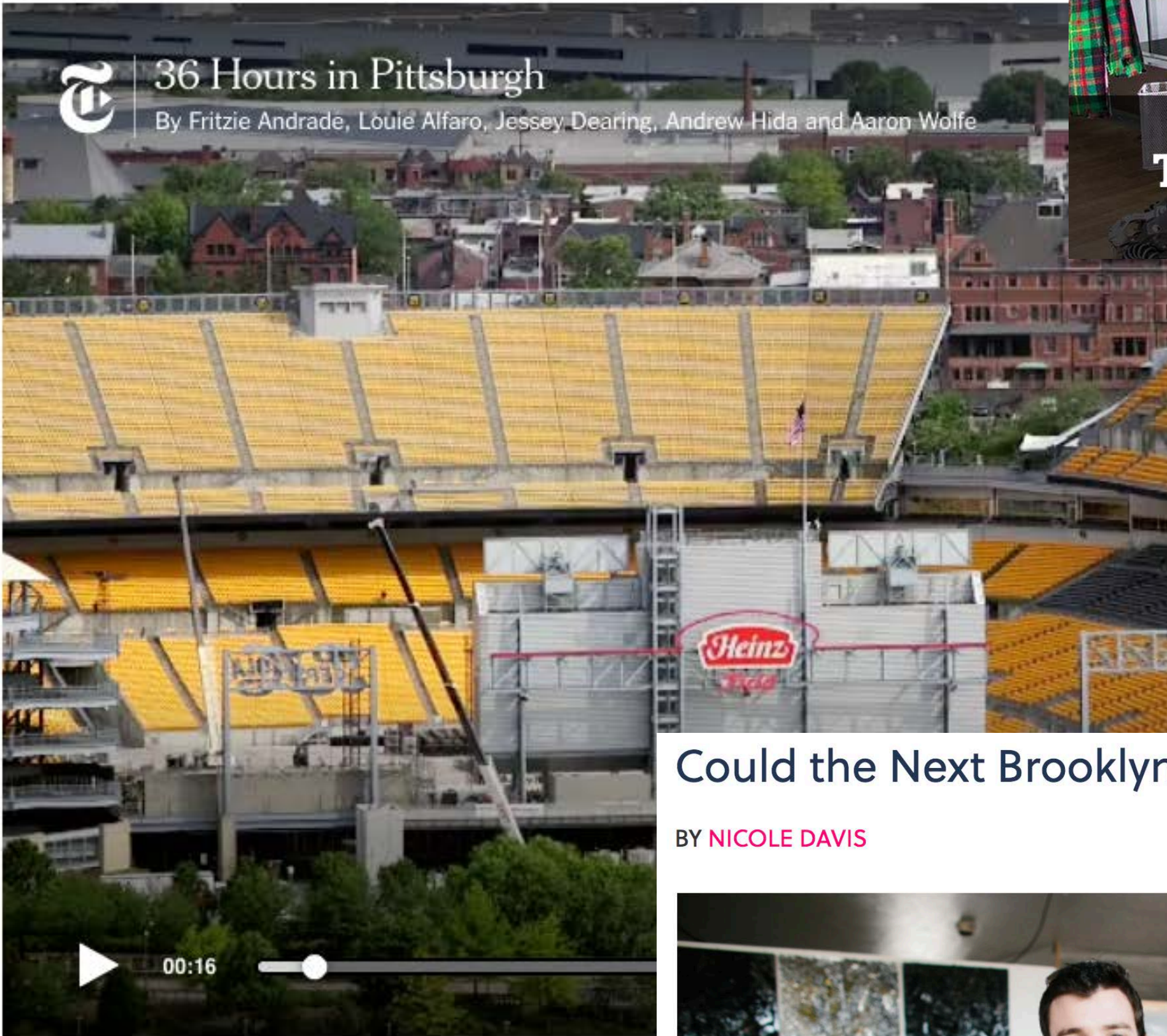
Let's talk about… Pittsburgh

Pittsburgh is now hot stuff!

36 Hours in Pittsburgh

Weekend Guide

By BRENDAN SPIEGEL JULY 15, 2015

36 Hours in Pittsburgh

By Fritzie Andrade, Louie Alfaro, Jessey Dearing, Andrew Hida and Aaron Wolfe

Carnegie Mellon University

WHAT WORKS

The Robots That Saved Pittsburgh

How the Steel City avoided Detroit's fate.

By GLENN THRUSH | 2/04/2014

What Millennials Love About Pittsburgh

'Land of Opportunity' has real meaning here.

00:16

Beyond Pittsburgh's pretty downtown, transformation and mome
art venues. By Fritzie Andrade, Louie Alfaro, Jessey Dearing, Andrew Hida

Could the Next Brooklyn Be Pittsburgh?

BY NICOLE DAVIS

And so is Bay Area rent…

THE NUMBERS

The Median Rent for an SF Two-Bedroom Hits $5,000/Month

Friday, October 9, 2015, by Tracy Elsen

Top 10 1 Bedroom Median Rents – October 2015

SEA #10 · BOS #3 · NYC #2 · CHI #7 · SF #1 · OAK #6 · SJ #4 · DC #5 · LA #9 · MIA #8

City	1 Bed Rent
San Francisco, CA	$3,620
New York, NY	$3,230
Boston, MA	$2,300
San Jose, CA	$2,270
Washington, DC	$2,090
Oakland, CA	$2,080
Chicago, IL	$1,980
Miami, FL	$1,850
Los Angeles, CA	$1,840
Seattle, WA	$1,660

zumper

Zumper – apartment rental data. Median rents for all homes available or vacant in September 2015. Ranking based on median price of a 1 bedroom unit

RENT PRICES
SF RENT
THE NUMBERS
TOP
ZUMPER
6 COMMENTS
Like 2.9k

It's that time of month again when rental website Zumper puts out their monthly rent report and dashes San Franciscans' dreams of ever being able to move again. The new median rental price for a **one-bedroom in the city is $3,620**, up $90 in just one month. That price is also up 13 percent over last year's mark. As Zumper points out, the even bigger increase has been in two-bedroom rents, which hit a $5,000 median for the first time this month and are **up 19 percent in a year**. San Francisco remains, of course, the most expensive city in the country for rents.
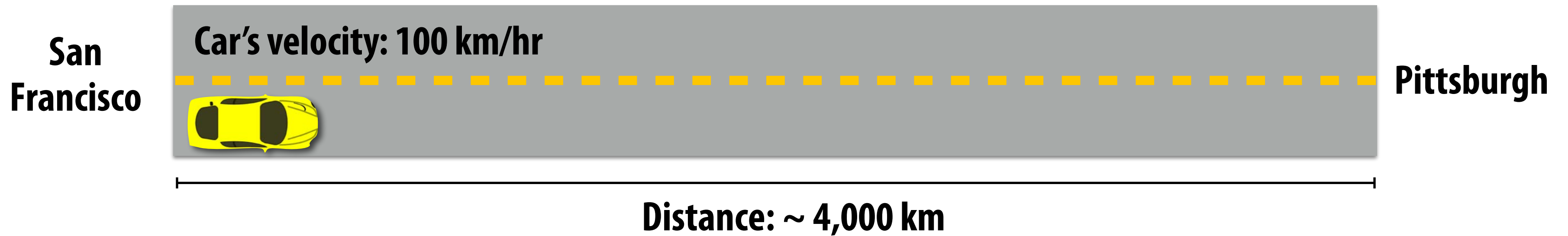
Although San Francisco's median rent for a one-bedroom is exceptionally high, there are actually only eight neighborhoods at or above that median. The **most expensive neighborhood is the Financial District, with a $4,270 per month median rent** for a one-bed. It is followed by Mission Bay/Dogpatch at $3,900 and Pacific Heights at $3,850. South Beach, Russian Hill, Potrero Hill, SoMa, and the Marina also sit above the median rent. And while the NoMad and Flatiron neighborhoods in New York are still more expensive than any neighborhood in San Francisco, the Financial District now has the same median one-bedroom rent as Tribeca, which used to sit far above any San Francisco spot.

Hey, let's move to Pittsburgh!
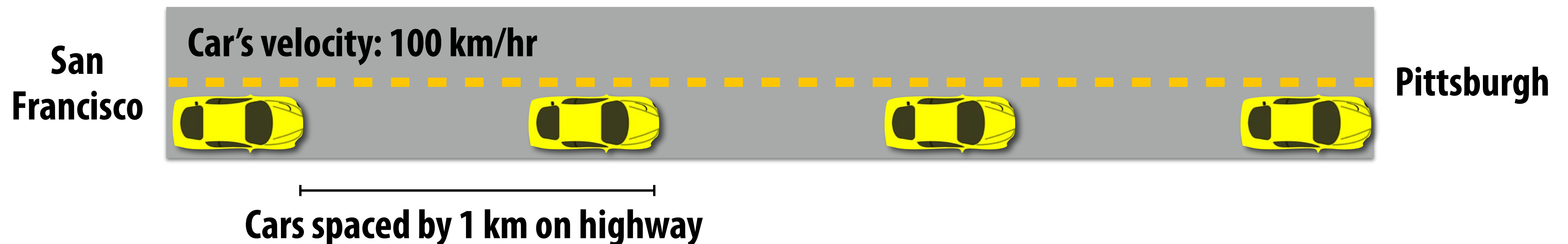(all the cool tech kids are doing it!)

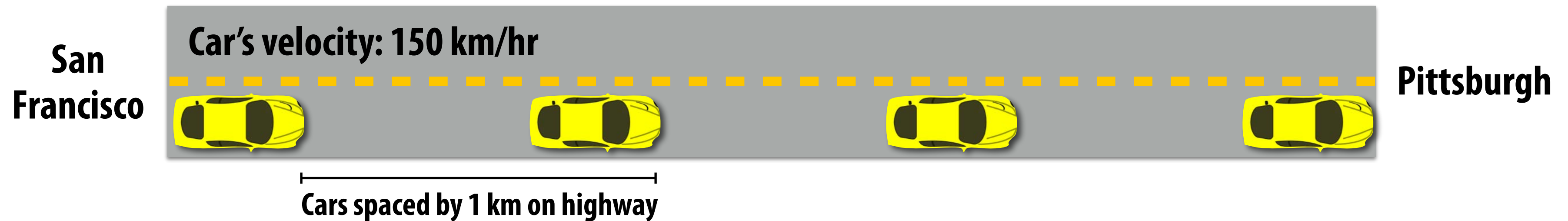# Everyone wants to get to Pittsburgh!

(Latency vs. throughput review)

San Francisco | Car's velocity: 100 km/hr | **Pittsburgh**

Distance: ~ 4,000 km

Latency of moving a person from San Francisco to Pittsburgh: 40 hours

San Francisco | Car's velocity: 100 km/hr | **Pittsburgh**

Cars spaced by 1 km on highway

Throughput: 100 people per hour (1 car every 1/100 of an hour)

Improving throughput

Car's velocity: 150 km/hr

San Francisco

Pittsburgh

Cars spaced by 1 km on highway

Approach 1: drive faster!

Throughput = **150** people per hour (1 car every 1/150 of an hour)

Car's velocity: 100 km/hr

San Francisco

Pittsburgh

Cars spaced by 1 km on highway
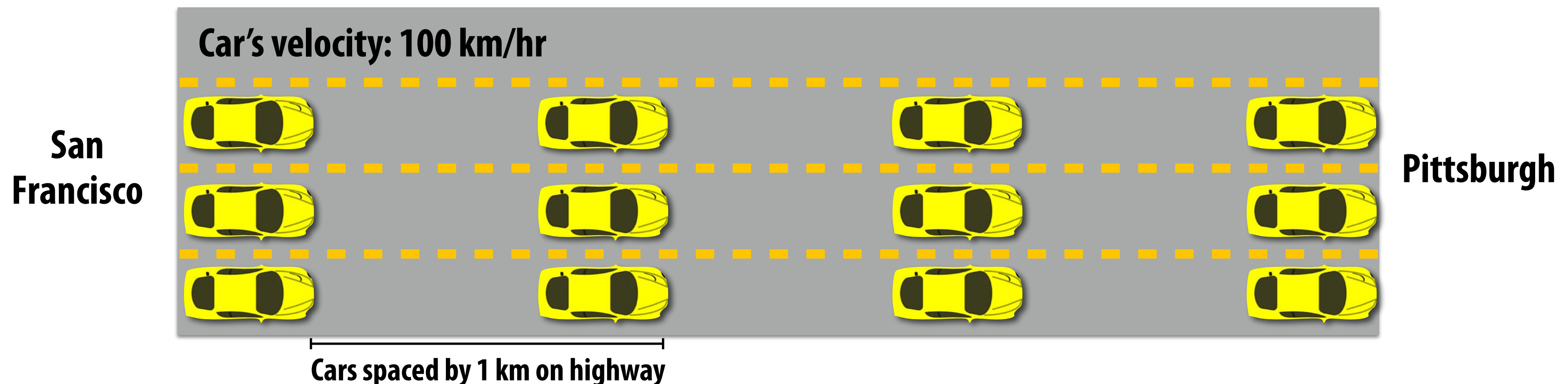
Approach 2: build more lanes!

Throughput: **300** people per hour (3 cars every 1/100 of an hour)

Review: latency vs throughput

Latency

The amount of time needed for an operation to complete.

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

Asking a question on Piazza gets response in 10 minutes

Bandwidth

The rate at which operations are performed.

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

A communication link can send 10 million messages per second

The TAs answer 50 questions per day on Piazza

What if only one car can be on the highway at a time?

When car on highway reaches Pittsburgh, the next car leaves San Francisco.

San Francisco

Car's velocity: 100 km/hr
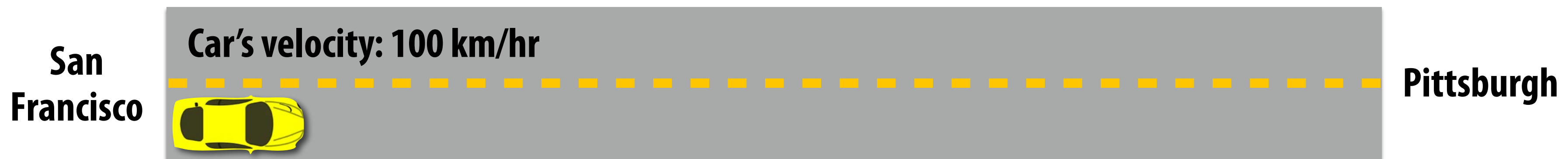
Pittsburgh

Latency of moving a person from San Francisco to Pittsburgh: 40 hours

Throughput = 1 / latency

= 1 / 40 of a person per hour (1 car every 40 hours)

Pipelining

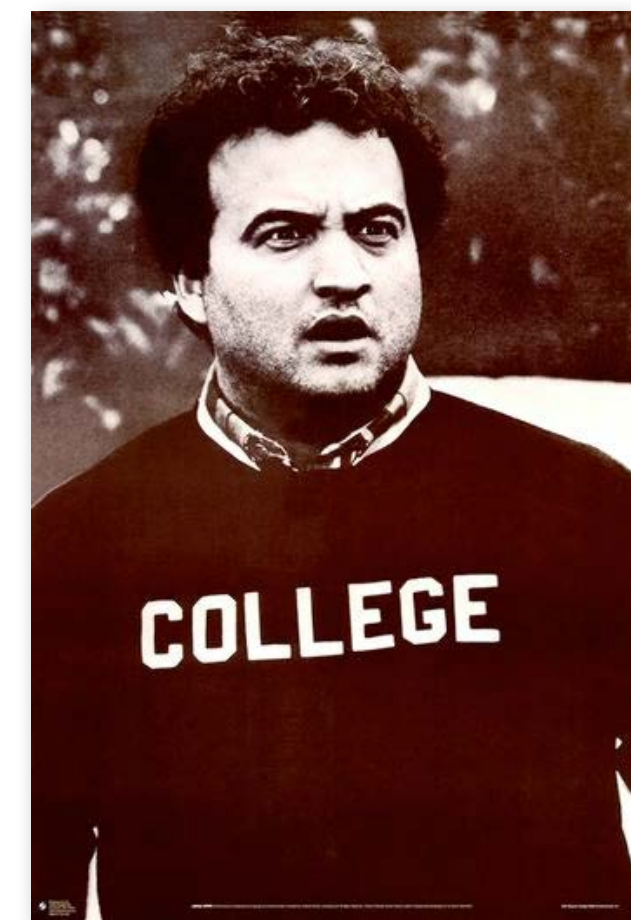# Example: doing your laundry

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

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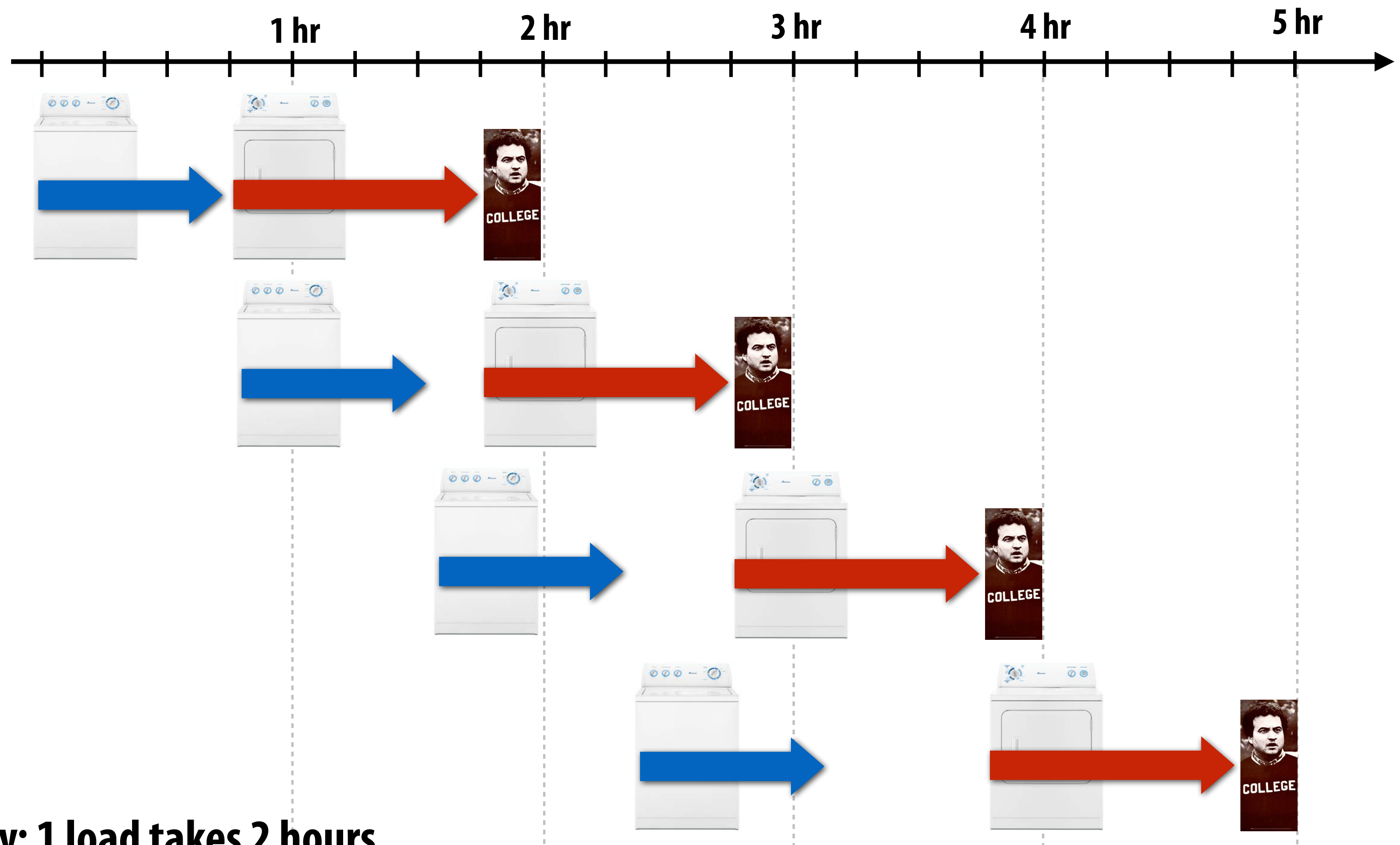use two washers, two dryers, and call a friend

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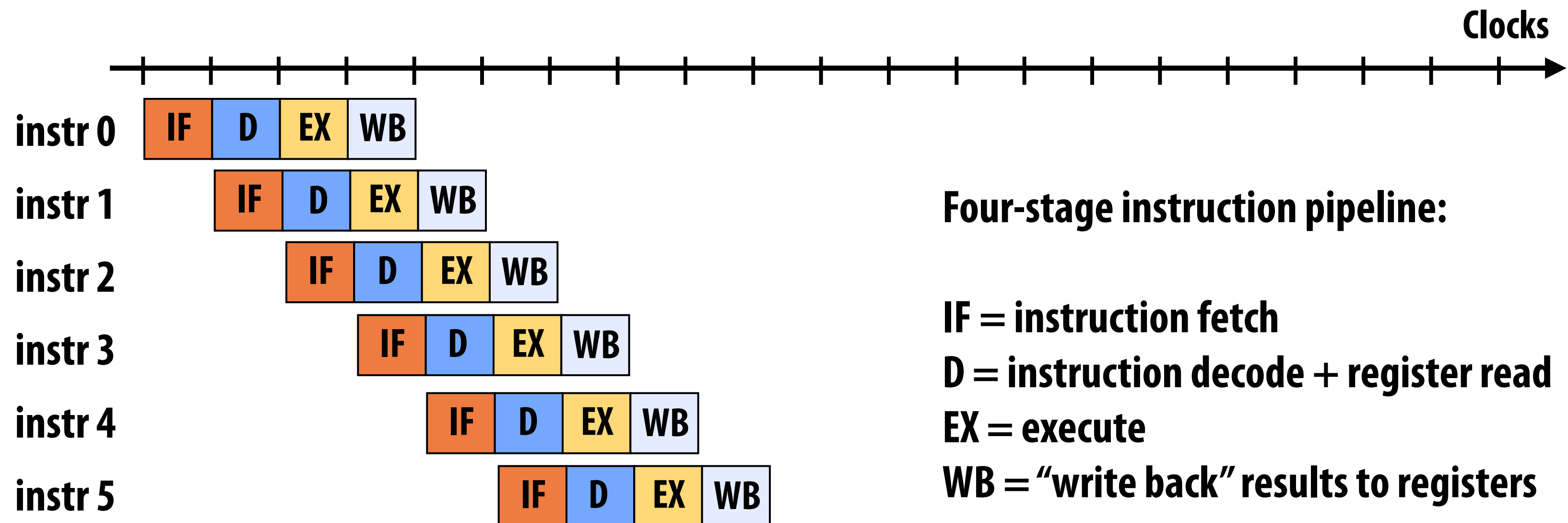
Latency: 1 load takes 2 hours
Throughput: 1 load/hour
Resources: one washer, one dryer

Another example: an instruction pipeline

Break execution of each instruction down into several smaller steps
Enables higher clock frequency (only a simple, short operation is done by each part of pipeline each clock)

Clocks

| instr 0 | IF | D | EX | WB |

| instr 1 | IF | D | EX | WB |

| instr 2 | IF | D | EX | WB |

| instr 3 | IF | D | EX | WB |

| instr 4 | IF | D | EX | WB |

| instr 5 | IF | D | EX | WB |

Four-stage instruction pipeline:

IF = instruction fetch
D = instruction decode + register read
EX = execute
WB = "write back" results to registers
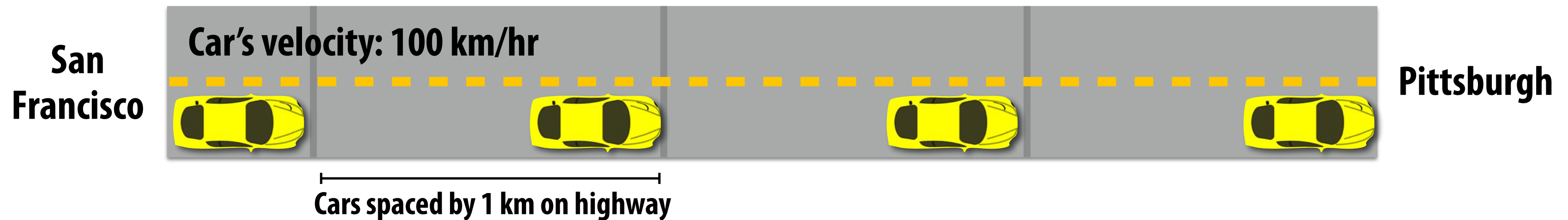
Latency: 1 instruction takes 4 cycles
Throughput: 1 instruction per cycle
(Yes, care must be taken to ensure program correctness when back-to-back instructions are dependent.)
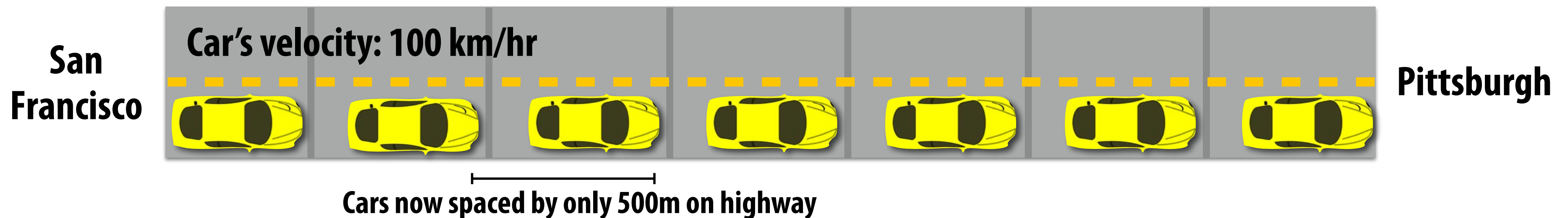
Intel Core i7 pipeline is variable length (it depends on the instruction) ~15-20 stages

Analogy to driving to Pittsburgh example

Task of driving from San Francisco to Pittsburgh is broken up into smaller subproblems that different cars can tackle in parallel
(top: subproblem = drive 1 km, bottom: subproblem = drive 500m)

San Francisco

Car's velocity: 100 km/hr

Pittsburgh

Cars spaced by 1 km on highway

Throughput = 100 people per hour (1 car every 1/100 of an hour)

San Francisco

Car's velocity: 100 km/hr

Pittsburgh

Cars now spaced by only 500m on highway

Throughput = 200 people per hour (1 car every 1/200 of an hour) *

* Equivalent throughput to maintaining 1 km spacing of cars and driving at 200 km/hr

A simple model of non-pipelined communication

Example: sending a n-bit message

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

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

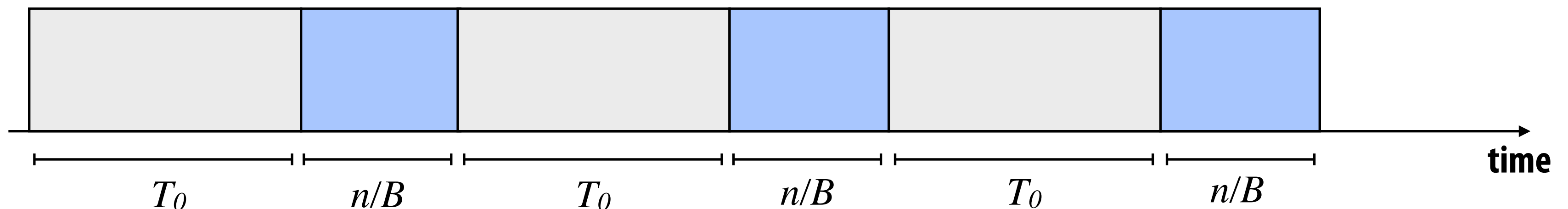T_0 = **start-up latency** (e.g., time until first bit arrives at destination)

n = **bytes transferred** in operation

B = **transfer rate** (**bandwidth** of the link)

If processor only sends next message once previous message send completes...
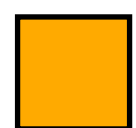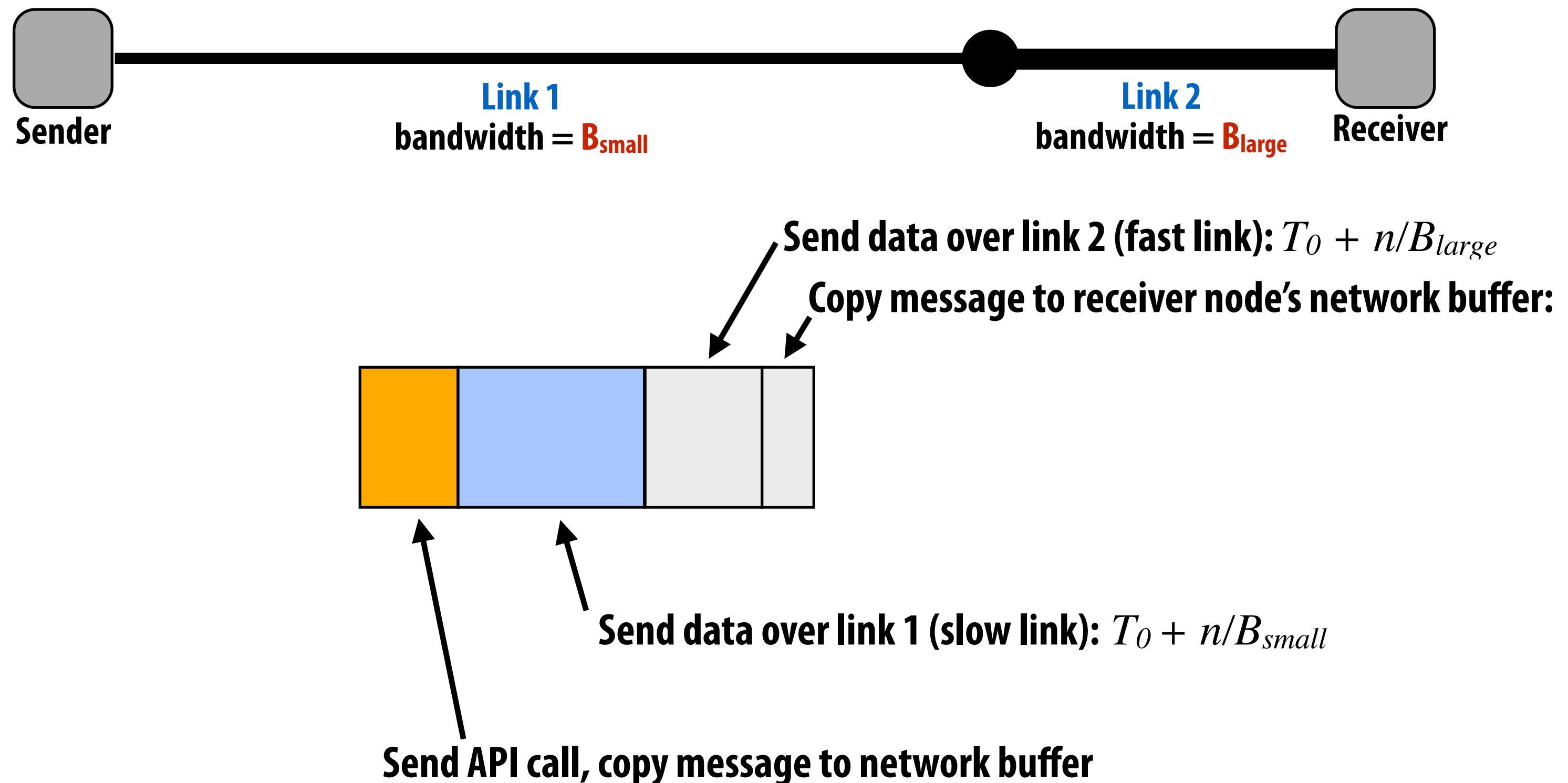
"Effective bandwidth" = $n / T(n)$

Effective bandwidth depends on transfer size (big transfers amortize startup latency)

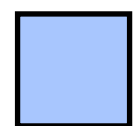| T_0 | n/B | T_0 | n/B | T_0 | n/B |

time

A more general model of communication

Example: sending a n-bit message

Total communication time = **overhead** + **occupancy** + **network delay**

Sender ——— **Link 1**
bandwidth = B_{small} ——— **Link 2**
bandwidth = B_{large} **Receiver**

Send data over link 2 (fast link): $T_0 + n/B_{large}$

Copy message to receiver node's network buffer:

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)

Pipelined communication

= sender blocked from sending additional messages due to network buffer being full

Occupancy determines communication rate! (in steady state: msg/sec = 1/occupancy)

Sending emits burst of messages (faster than 1/occupancy)

Messages are buffered while link is busy

Assume network buffer can hold at most two messages (numbers indicate number of msgs in buffer after insert)

time

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

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

= Network delay (everything else)

Cost

**The effect operations have on program execution time
(or some other metric, e.g.,energy consumed…)**

"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 time = overhead + occupancy + network delay

Total communication cost = communication time - overlap

Overlap: portion of communication performed concurrently with other work

"Other work" can be computation or other communication

Example 1: Asynchronous message send/recv allows communication to be overlapped with computation

Example 2: Pipelining allows multiple message sends to be overlapped

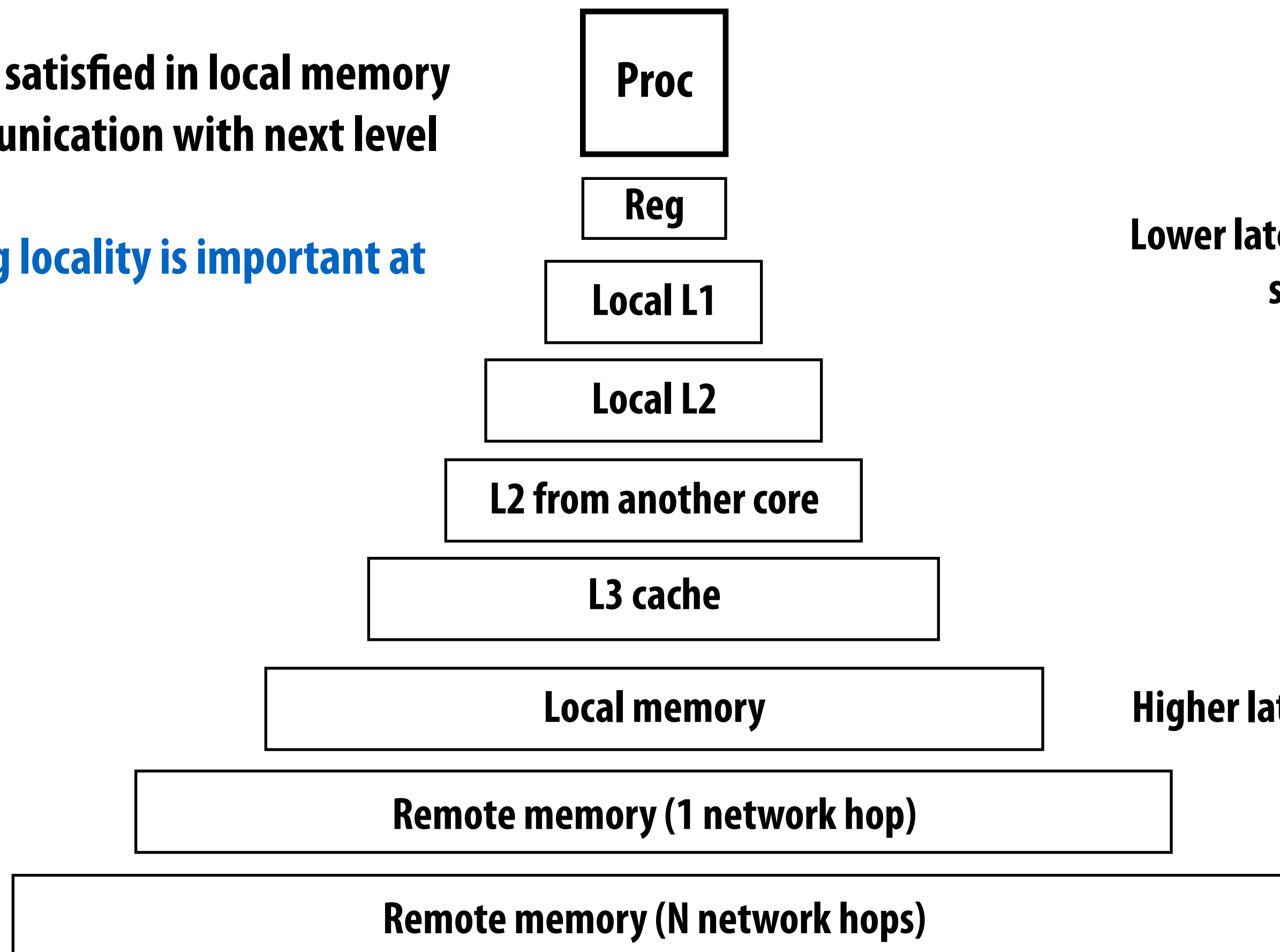# Think of a parallel system as an extended memory hierarchy

I want you to 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, accessed by sending a network message)

View from one processor

Accesses not satisfied in local memory cause communication with next level

So **managing locality is important at all levels**

| 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

Two reasons for communication: inherent vs. artifactual communication

Inherent communication
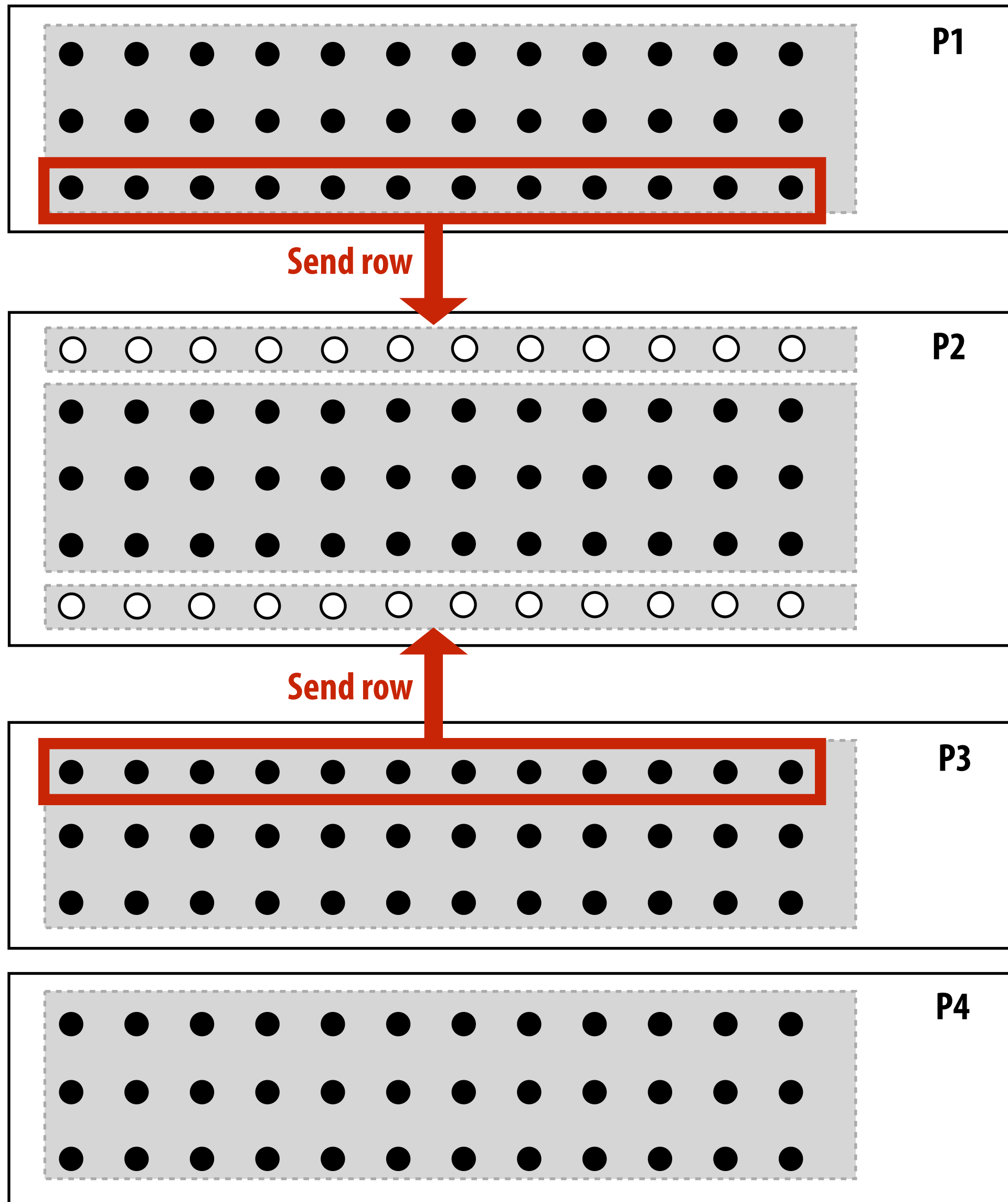
Communication that **must** occur in a parallel algorithm. The communication is fundamental to the algorithm.

In our messaging passing example at the start of class, sending ghost rows was inherent communication

Communication-to-computation ratio

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

- If denominator is the execution time of computation, ratio gives average bandwidth requirement of code

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

- 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 element-wise vector multiple from lecture 2)
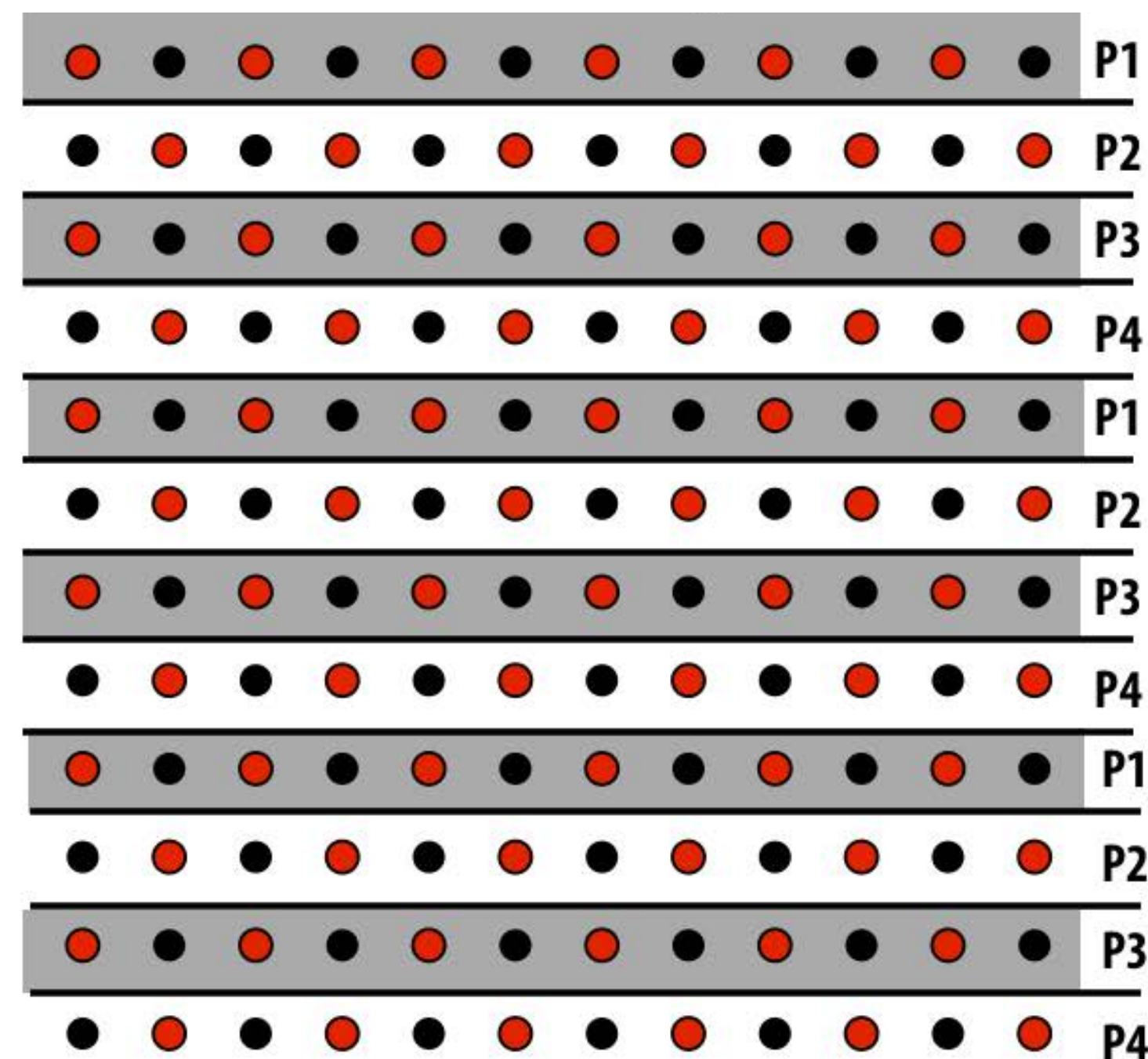
Reducing inherent communication

Good assignment decisions can reduce inherent communication (increase arithmetic intensity)

1D blocked assignment: N x N grid

1D interleaved assignment: N x N grid
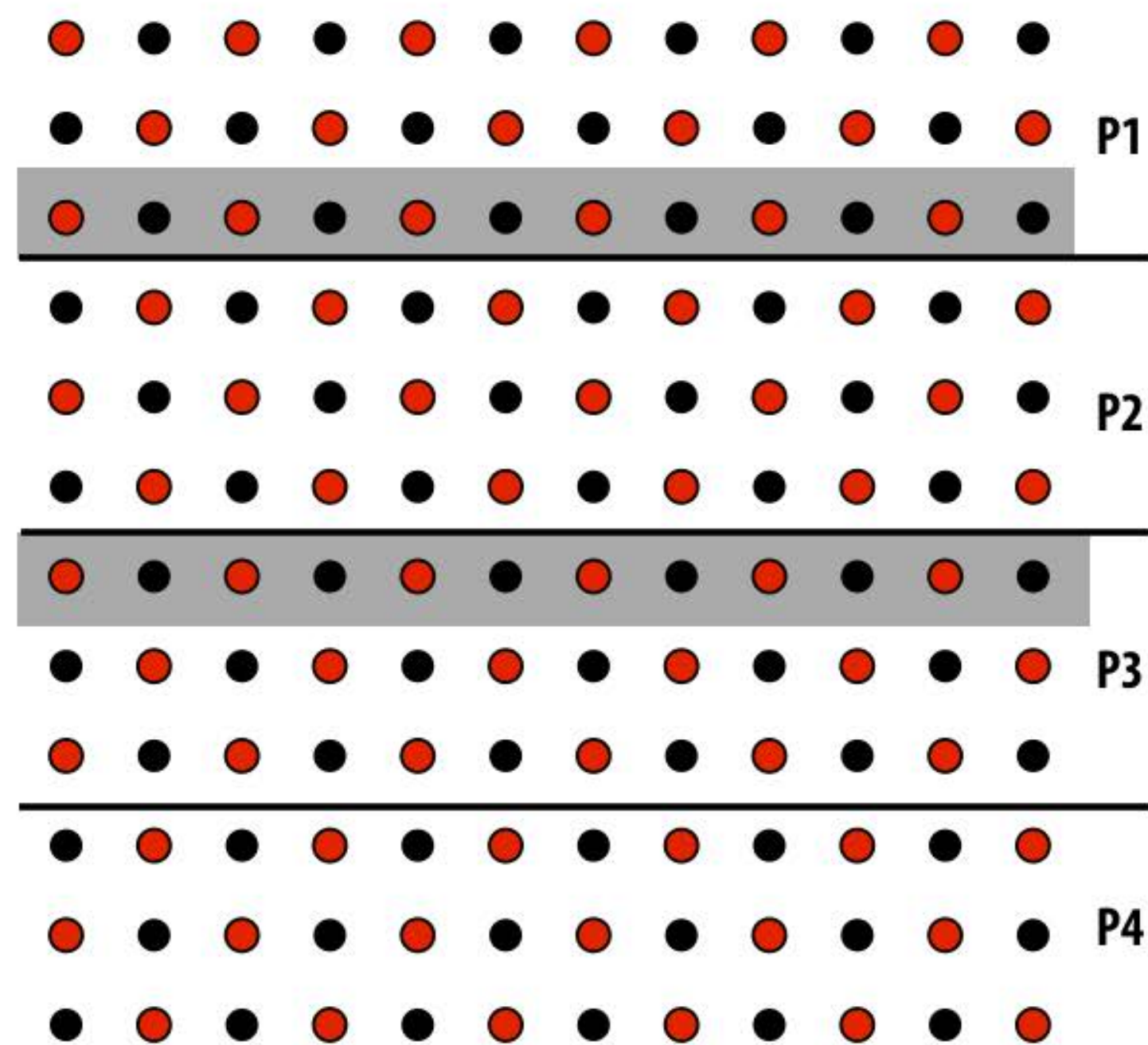
$$\frac{\text{elements computed (per processor)} \approx N^2/P}{\text{elements communicated (per processor)} \approx 2N} \propto N / P$$
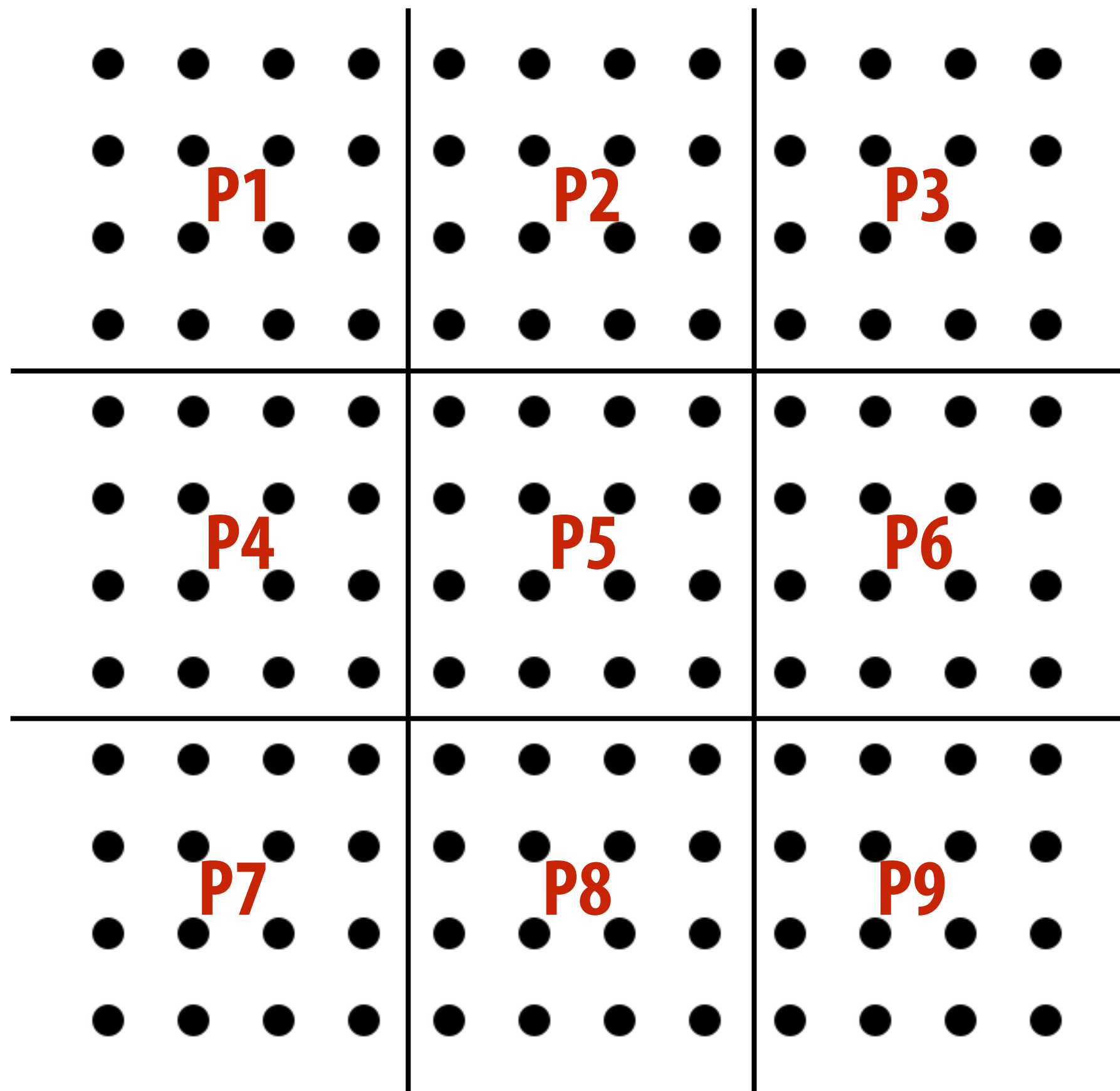
$$\frac{\text{elements computed}}{\text{elements communicated}} = 1/2$$

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: $\propto \dfrac{N}{\sqrt{P}}$
(per processor)

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

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 placement of data** in distributed memories (data doesn't reside near processor that accesses it the 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.
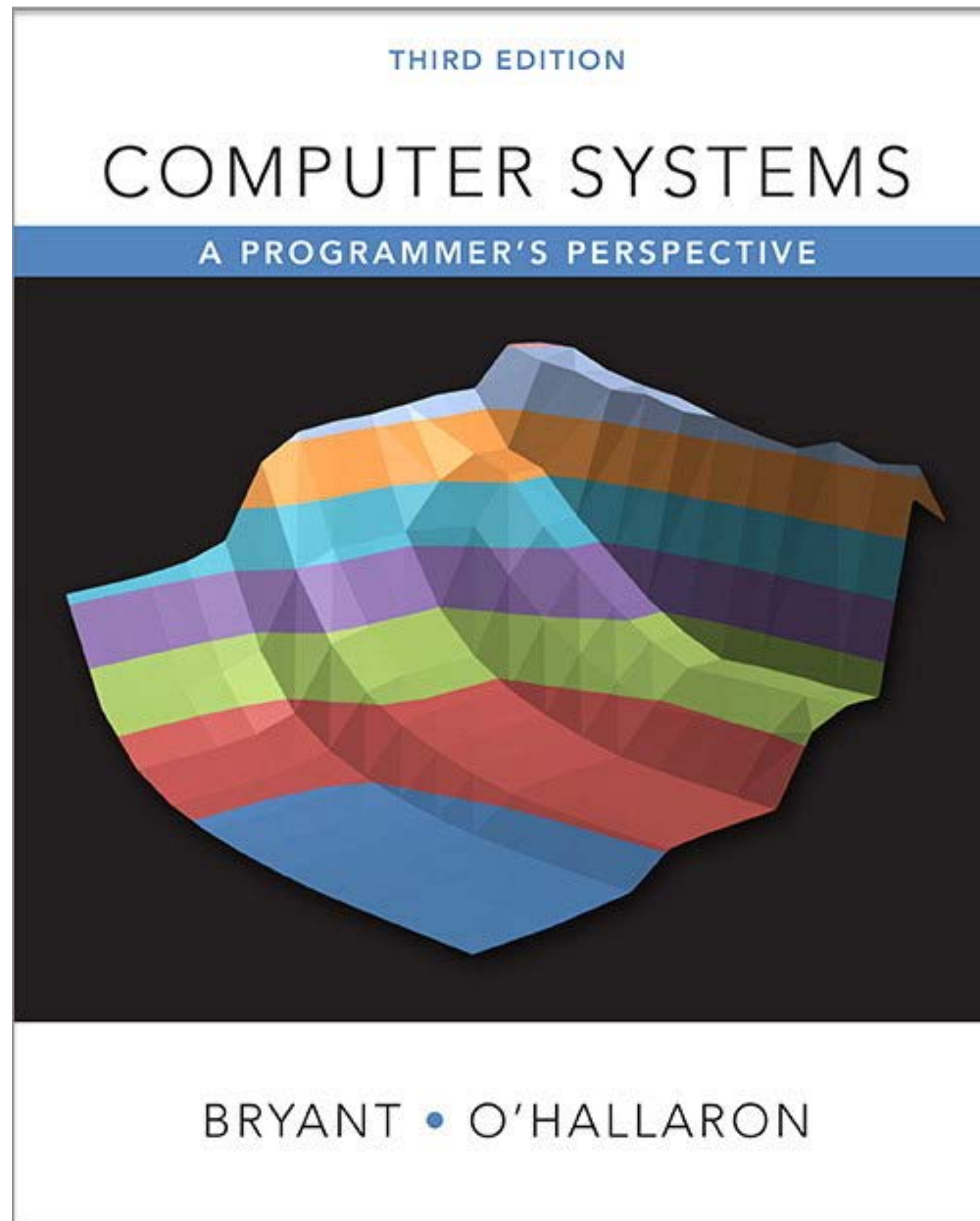
- **Communication miss (new)**

  Due to inherent or artifactual communication in parallel system
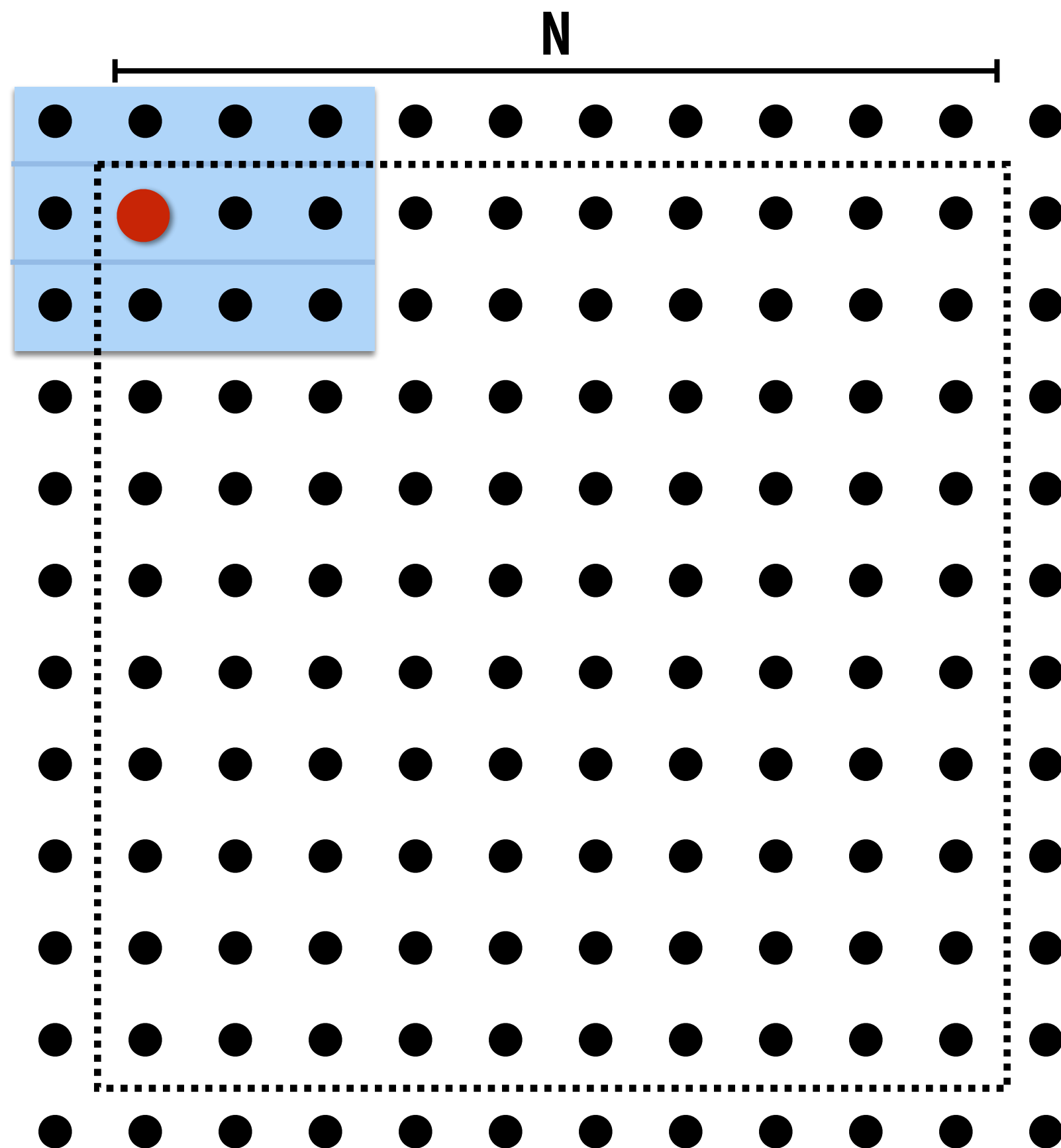
Communication: working set perspective
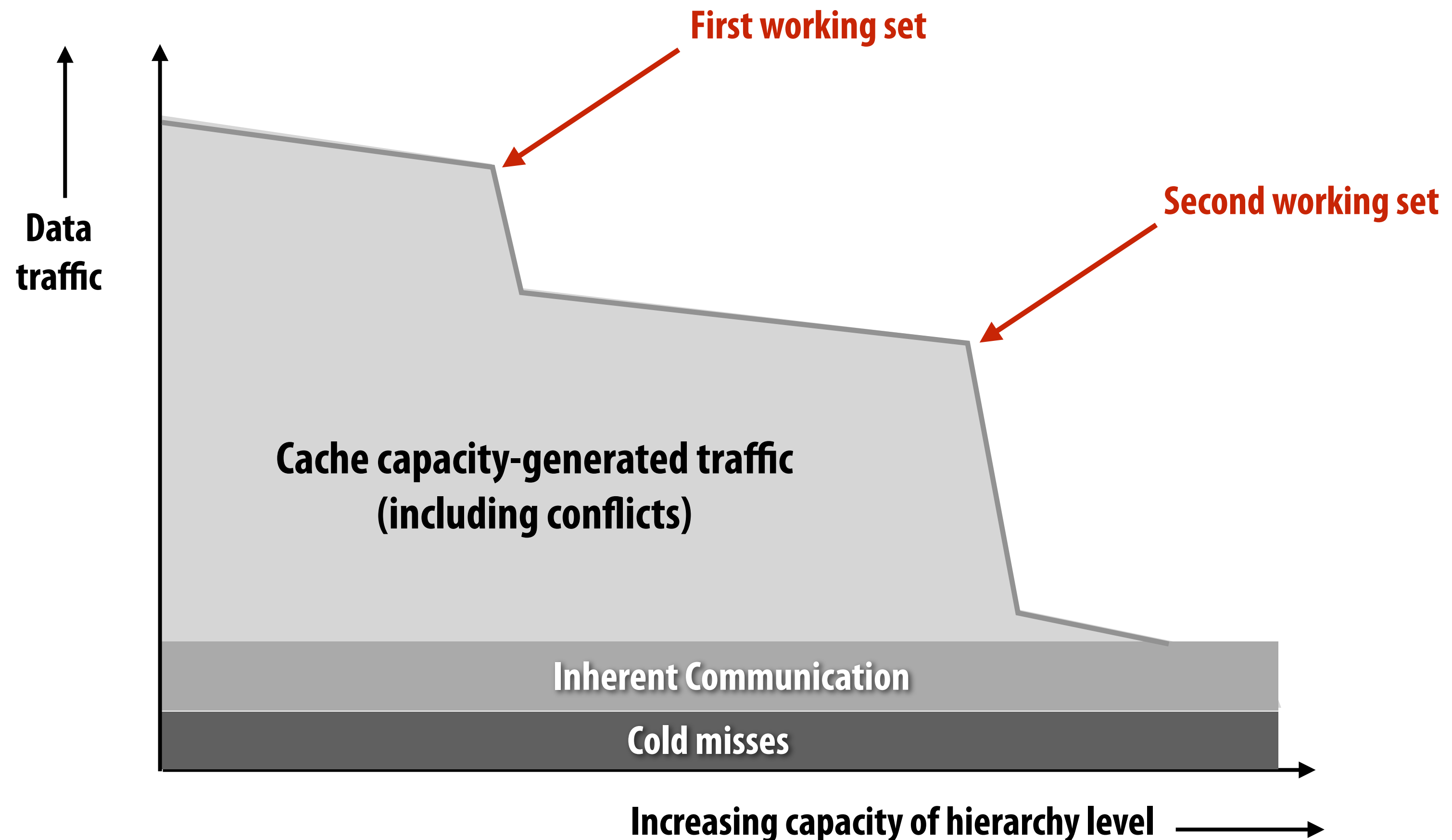
This diagram holds true at any level of the memory hierarchy in a parallel system
Question: how much capacity should an architect build for this workload?

Does the graph on the previous slide look familiar?

Techniques for reducing communication

Data access in grid solver: row-major traversal

N
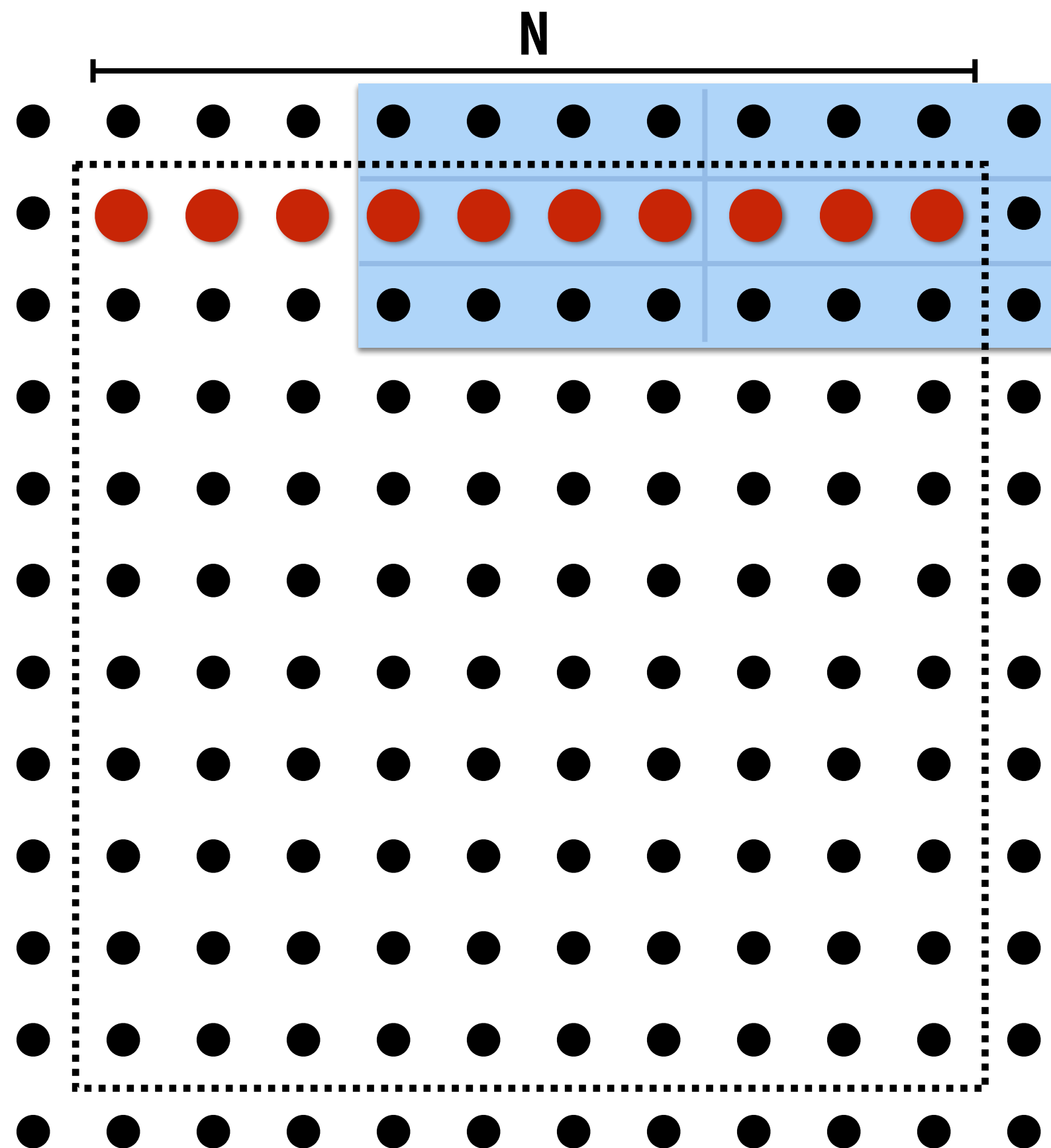
Assume row-major grid layout.

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.

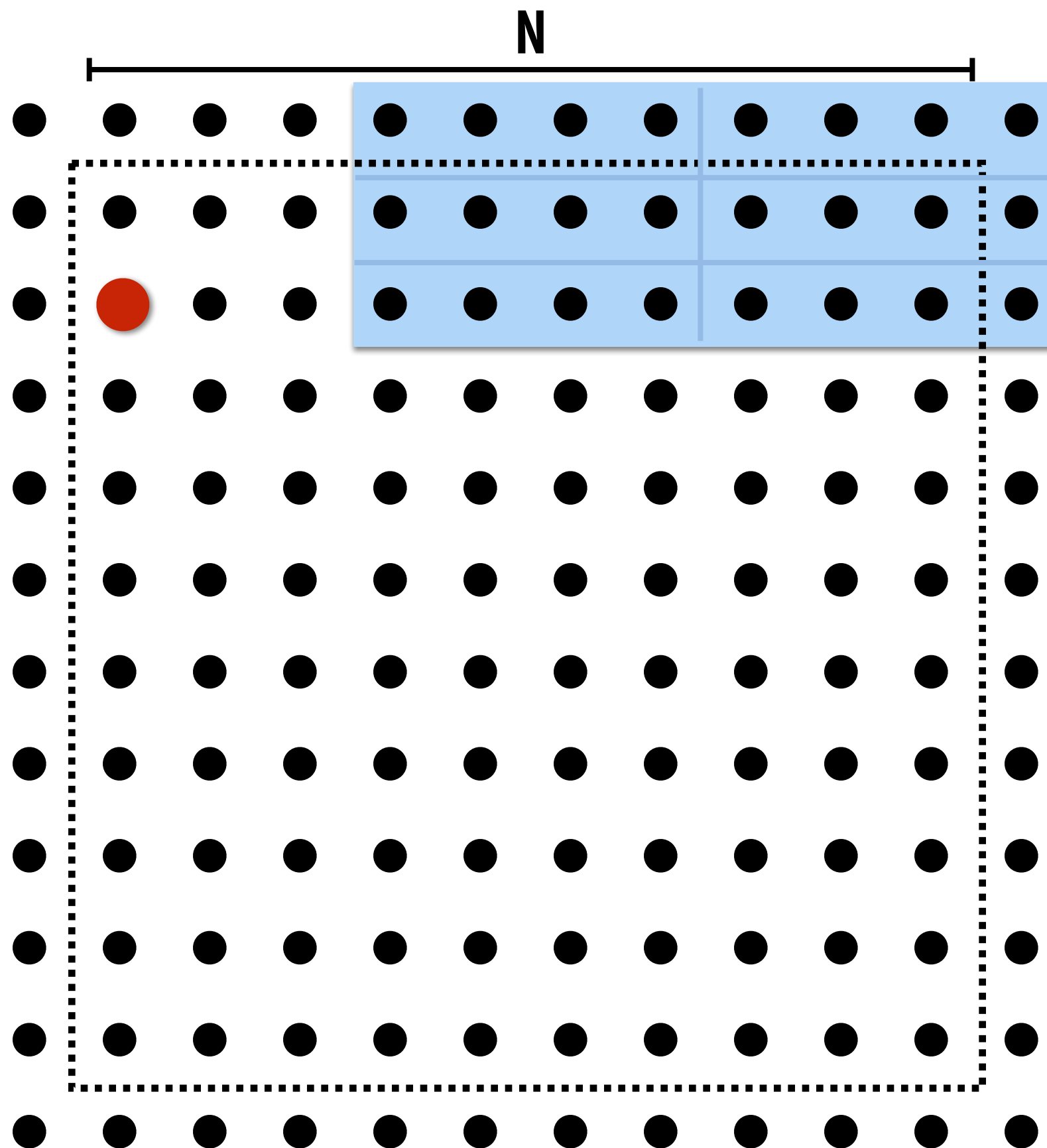# Data access in grid solver: row-major traversal

N

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.

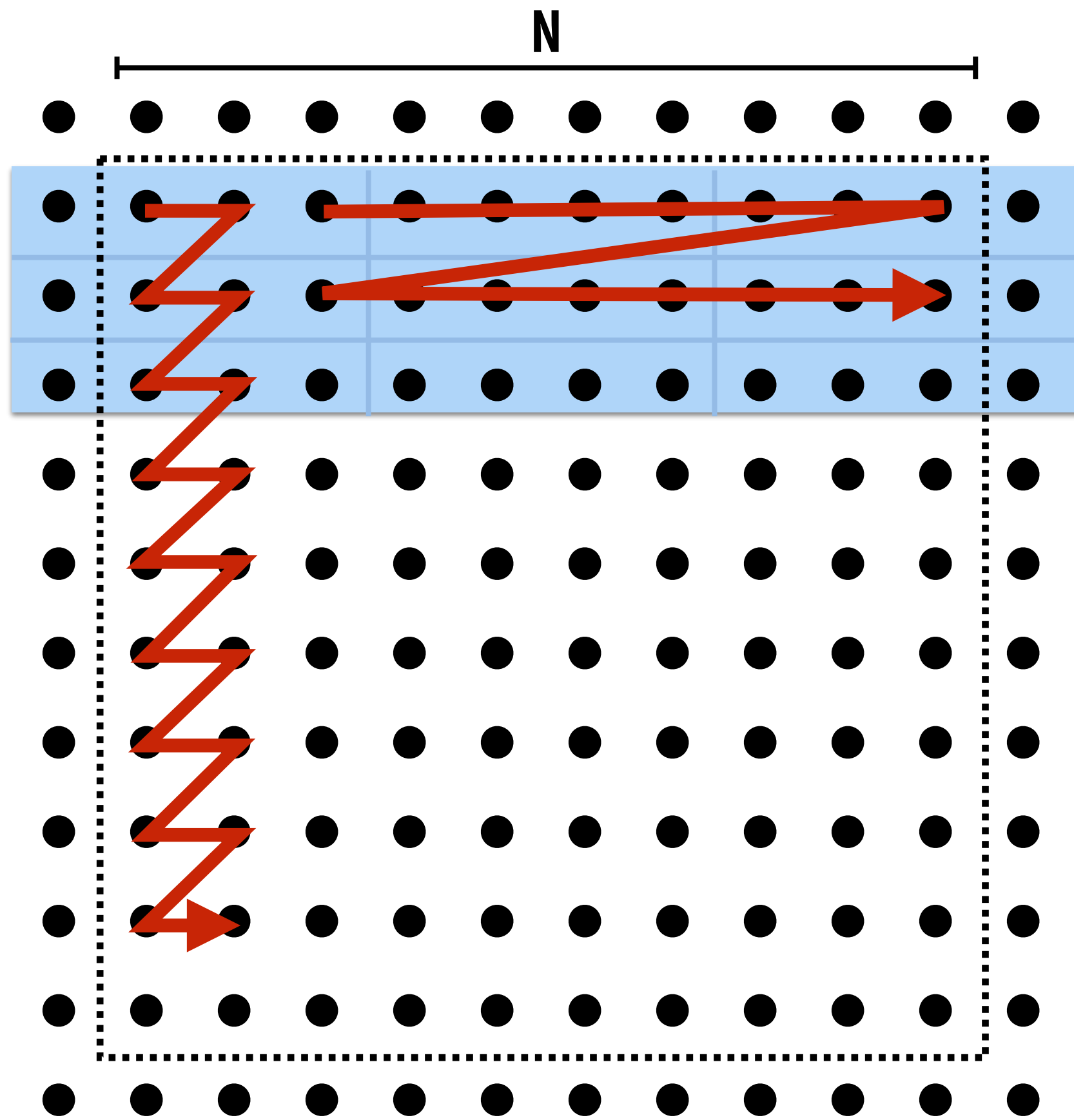Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

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

(What type of miss is this?)

This program loads three lines for every four elements.

Improving 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)

Now three lines for every eight elements.

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

```
void mul(int n, float* A, float* B, float* C) {
    for (int i=0; i<n; i++)
        C[i] = A[i] * B[i];
}
```
Two loads, one store per math op (arithmetic intensity = 1/3)

```
float* A, *B, *C, *D, *E, *tmp1, *tmp2;

// assume arrays are allocated here

// compute E = D + ((A + B) * C)
add(n, A, B, tmp1);
mul(n, tmp1, C, tmp2);
add(n, tmp2, D, E);
```
Overall arithmetic intensity = 1/3

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

// compute E = D + (A + B) * C
fused(n, A, B, C, D, E);
```
Four loads, one store per 3 math ops (arithmetic intensity = 3/5)

Code on top is more modular (e.g, array-based math library like numarray in Python)
Code on bottom performs much better. Why?

Improve arithmetic intensity by sharing data

- **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)

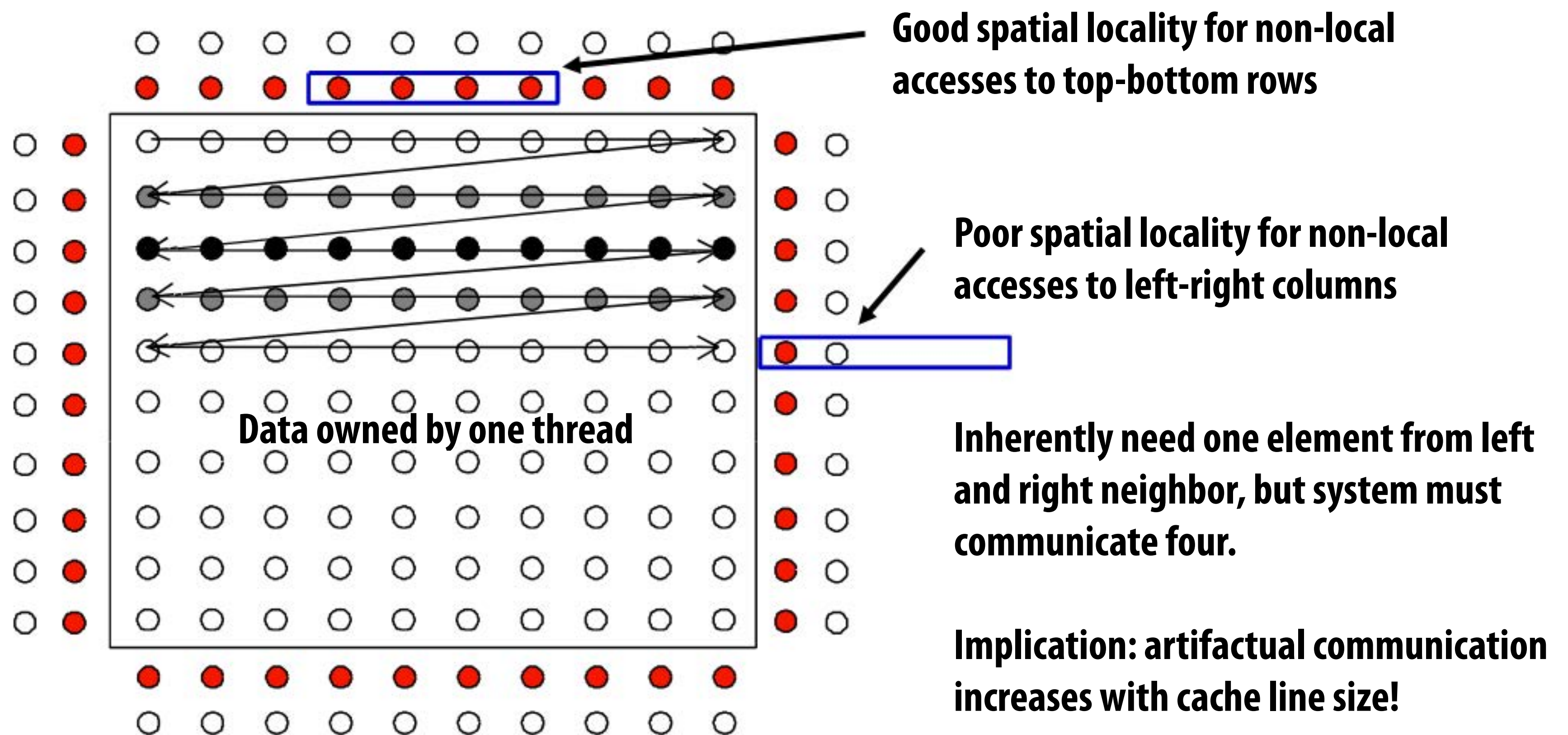 - So GPU implementations always schedule threads from the same block on the same GPU core

Exploiting spatial locality

- **Granularity of communication can be important because it may introduce artifactual communication**

  - **Granularity of communication / data transfer**

 - **Granularity of cache coherence (will discuss in future lecture)**

Artifactual communication due to comm. granularity

2D blocked assignment of data to processors as described previously. Assume: communication granularity is a cache line, and a cache 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

Data owned by one thread

Inherently need one element from left and right neighbor, but system 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

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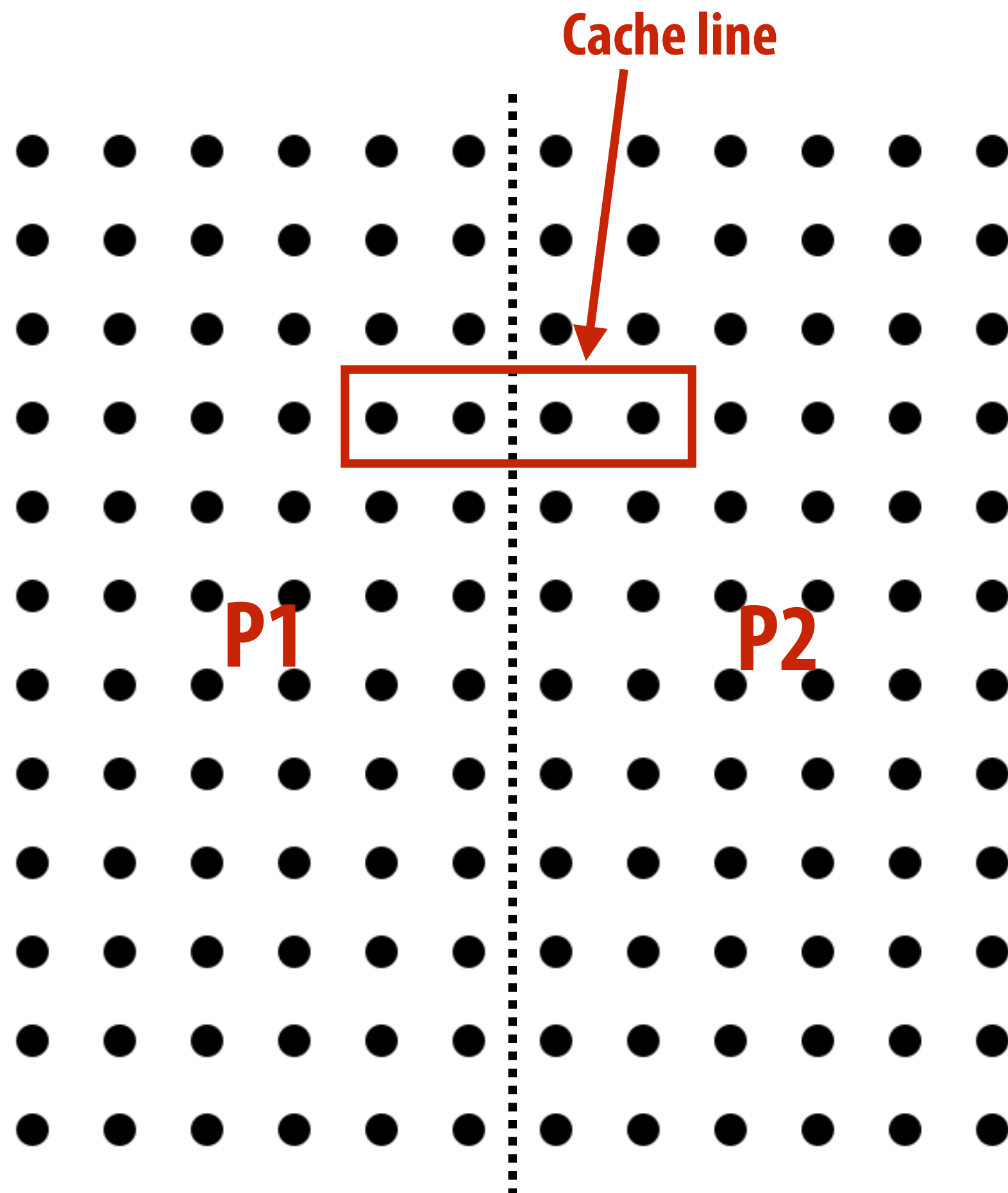
Data partitioned in half by column. Partitions assigned to threads running on P1 and P2

P1   **P2**

Threads access their assigned elements
(no <u>inherent</u> 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
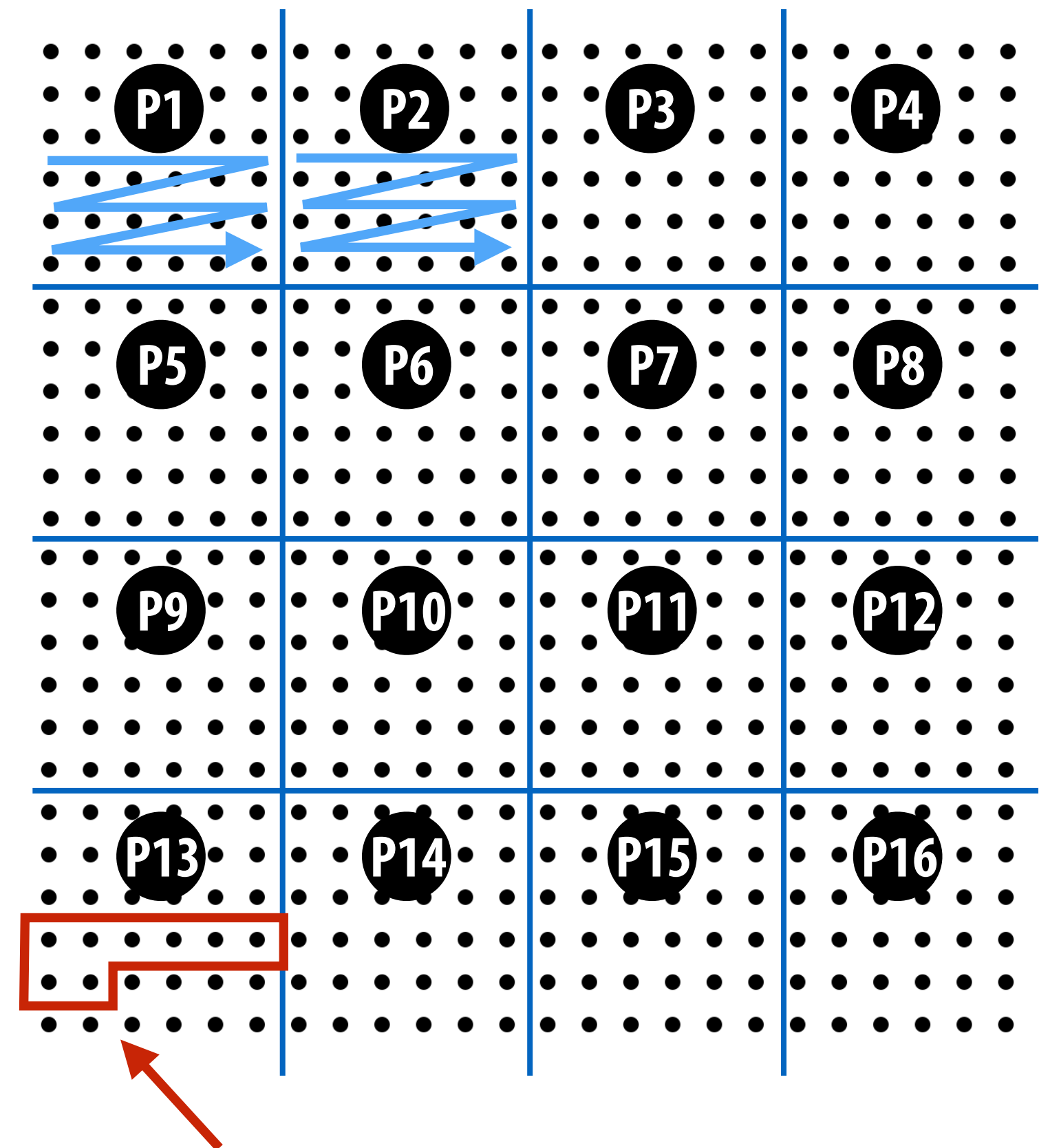
Reducing artifactual comm: blocked data layout

(Blue lines indicate consecutive memory addresses)

2D, row-major array layout

4D array layout (block-major)
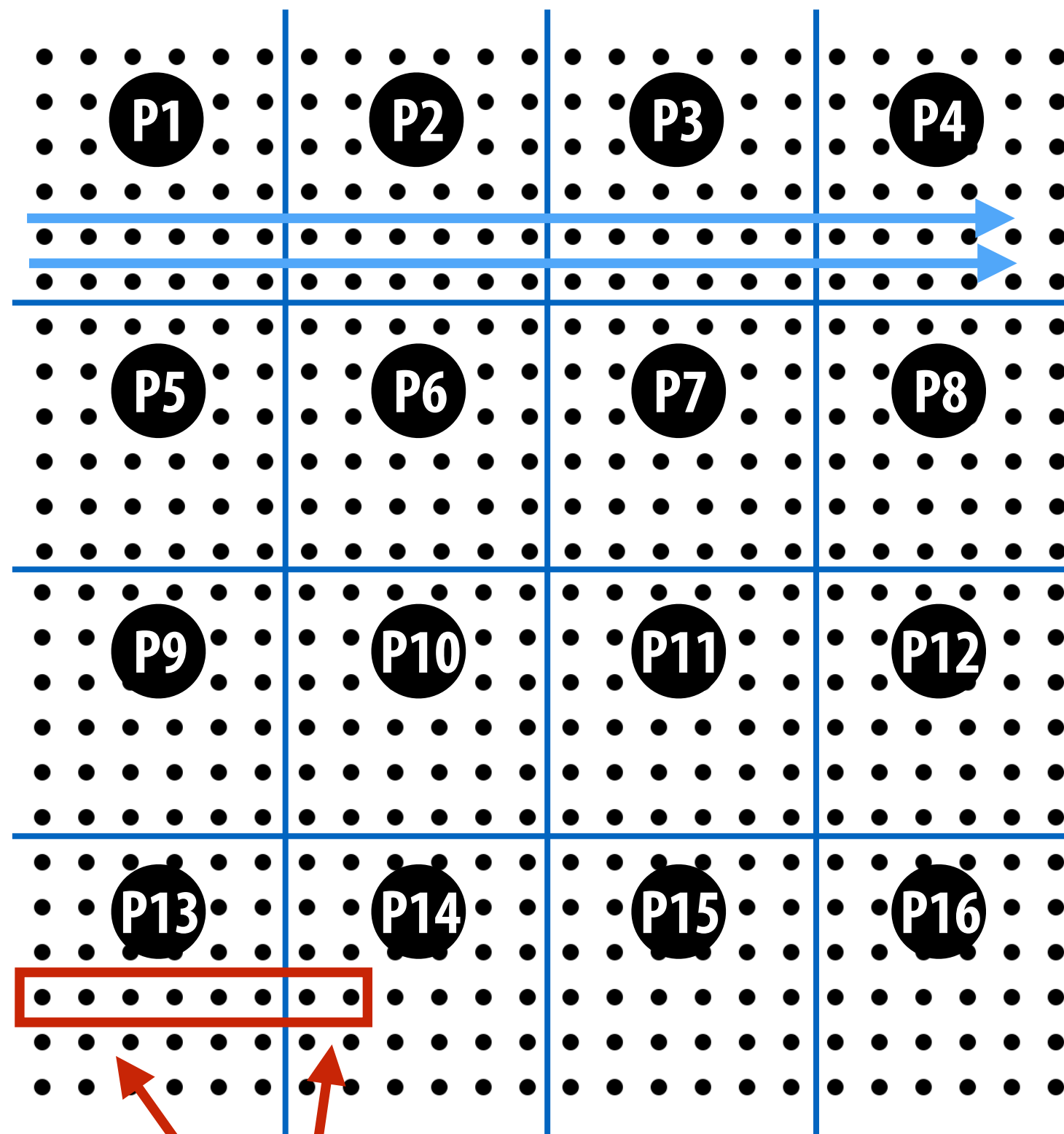
Consecutive addresses
straddle partition boundary

Consecutive addresses remain
within single partition

**Note: don't confuse blocked assignment of work to threads (true in both cases above)
with blocked data layout in the address space (only at right)**
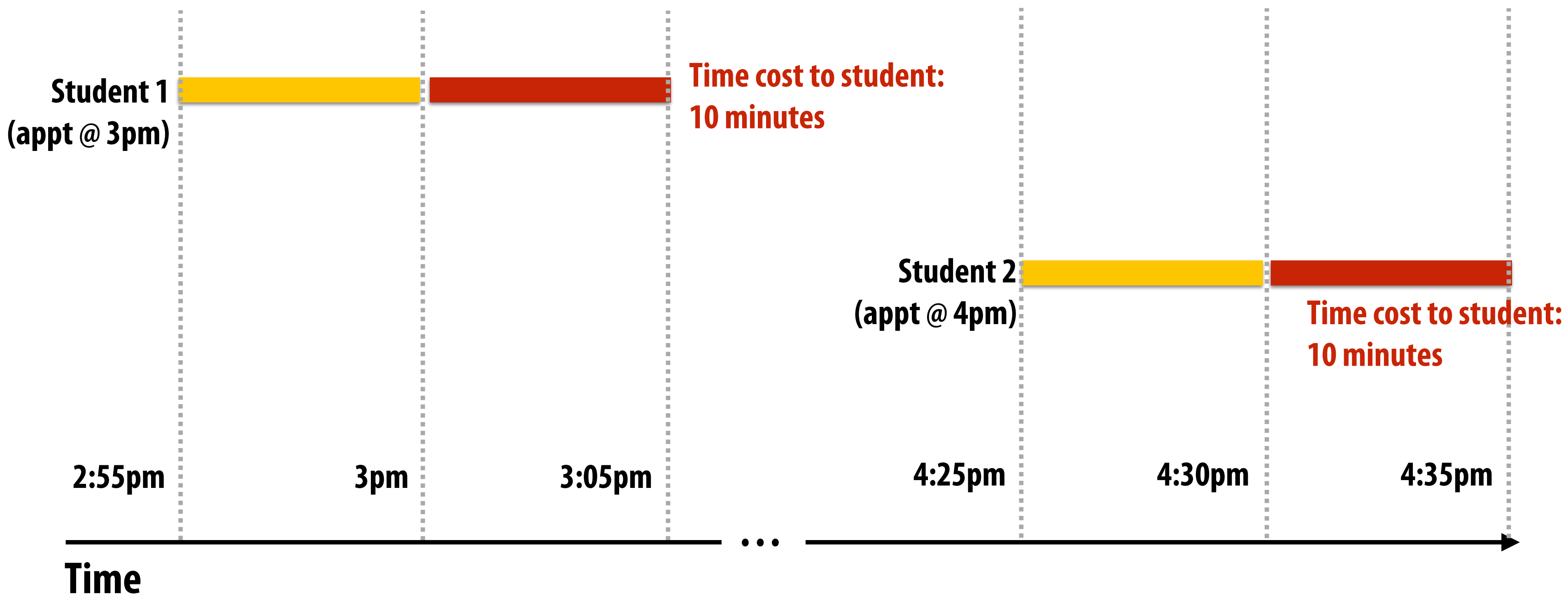
Contention

Example: two students make appointments to talk to me about course material (at 3pm and at 4:30pm)

- Operation to perform: Professor Mowry helps a student with a question
- Execution resource: Professor Mowry
- Steps in operation:
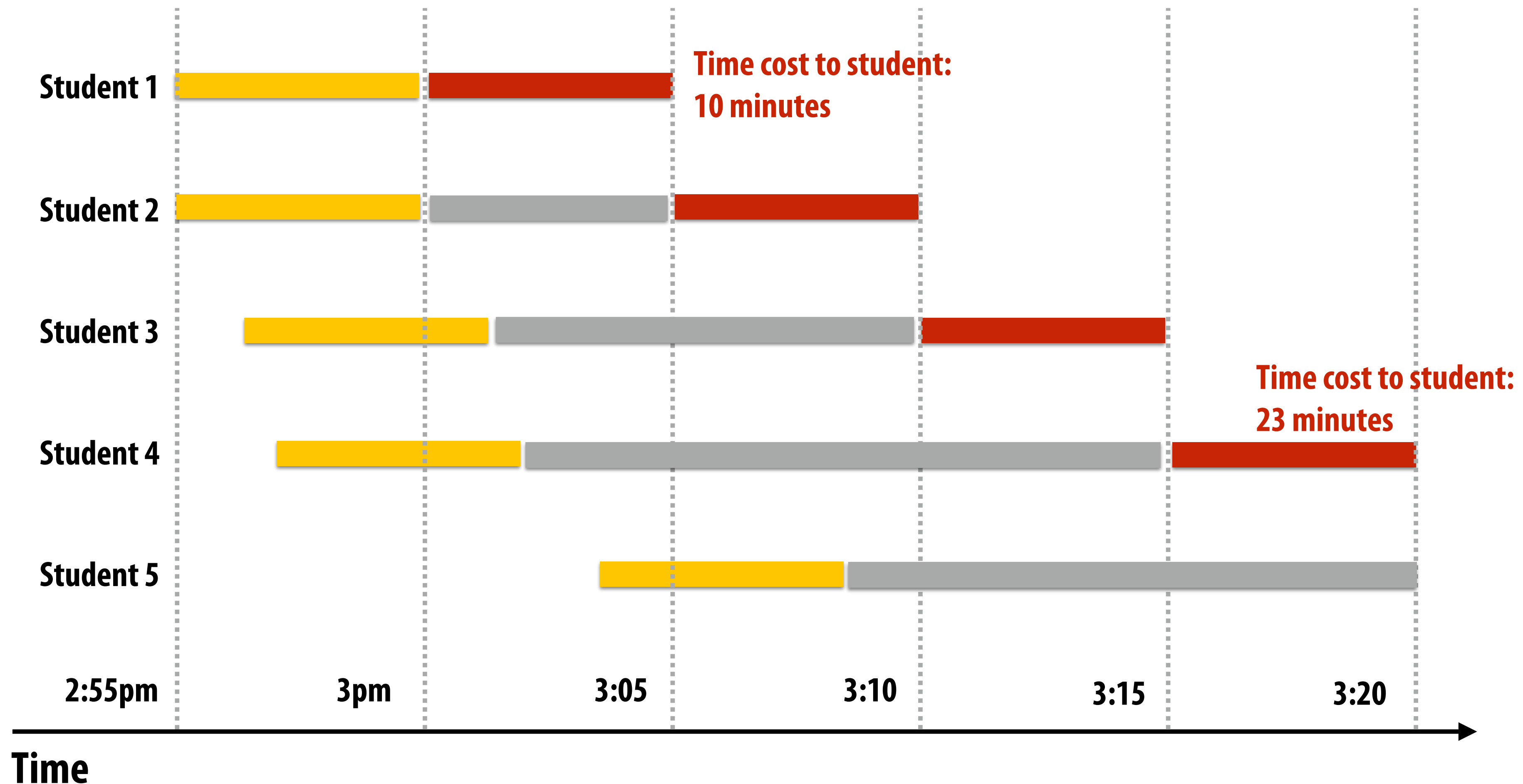 1. Student walks from Gates Cafe to Prof. Mowry's office (5 minutes) = 🟨
  2. Student waits in line (??) = ⬜
  3. Student gets question answered with insightful answer (5 minutes) = 🟥

Student 1
(appt @ 3pm)

Time cost to student:
10 minutes

Student 2
(appt @ 4pm)

Time cost to student:
10 minutes

| 2:55pm | 3pm | 3:05pm | 4:25pm | 4:30pm | 4:35pm |

. . .

Time

Office hours from 3-3:20pm (no appointments)

Student 1

Time cost to student:
10 minutes

Student 2

Student 3

Time cost to student:
23 minutes

Student 4

Student 5

2:55pm   3pm   3:05   3:10   3:15   3:20

Time

■ = Walk to Prof. Mowry's office (5 minutes)   ■ = Wait in line   ■ = Get question answered

Problem: contention for shared resource results in longer overall operation times (and likely higher cost to students)
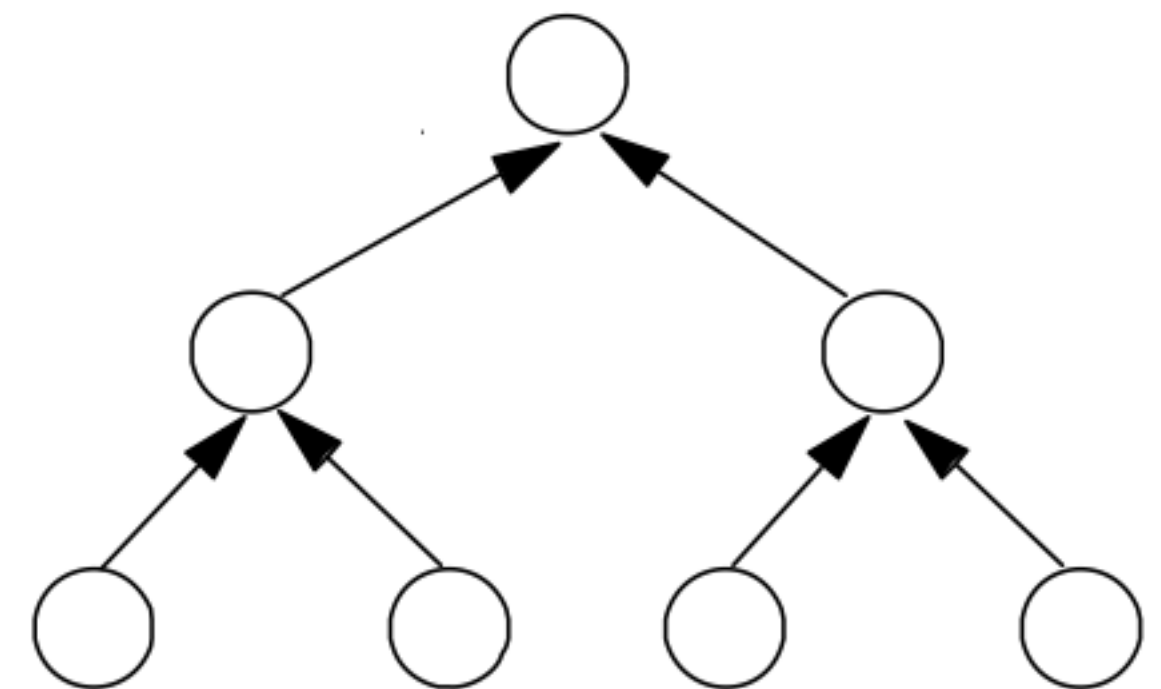
Contention

- **A resource can perform operations at a given throughput (number of transactions per unit time)**
  - **Memory, communication links, servers, TA's at office hours, 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
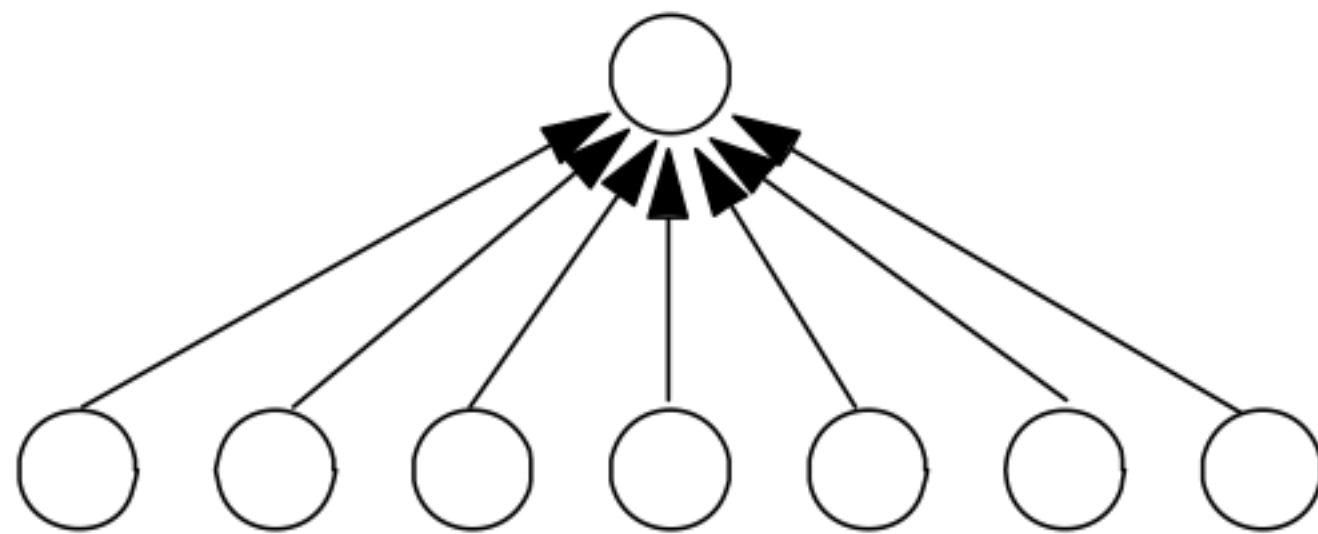
**Flat communication:
potential for high contention
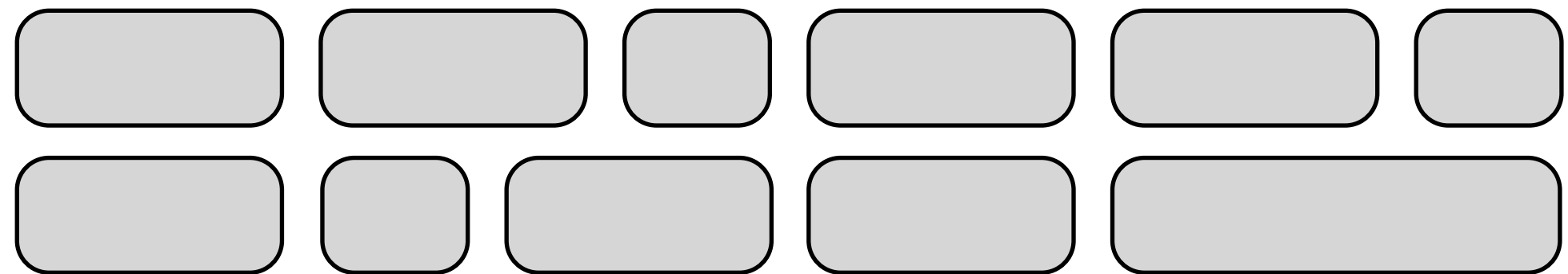(but low latency if no contention)**

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

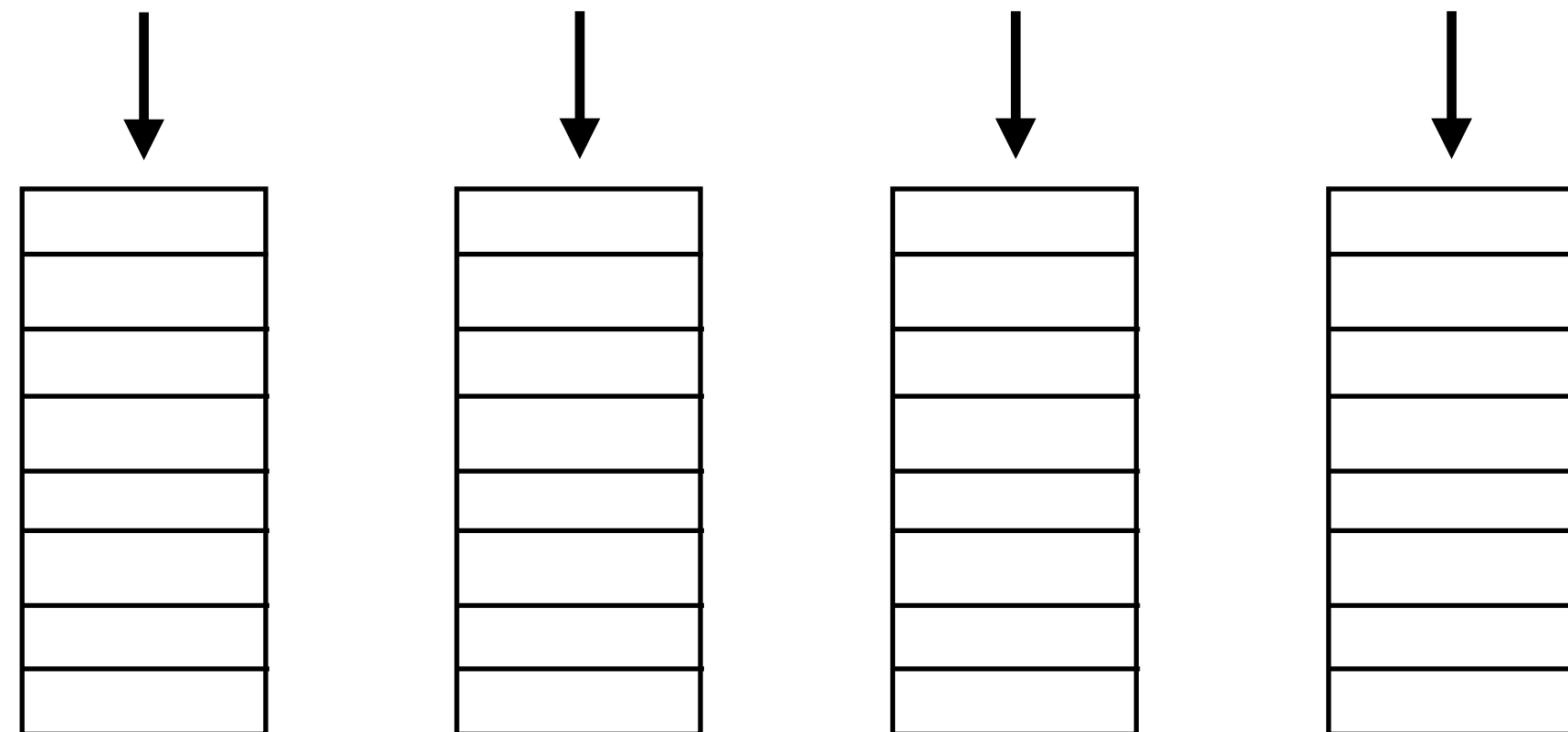# Example: distributed work queues serve to reduce contention (contention in access to single shared work queue)

Subproblems
(a.k.a. "tasks", "work to do")

Set of work queues
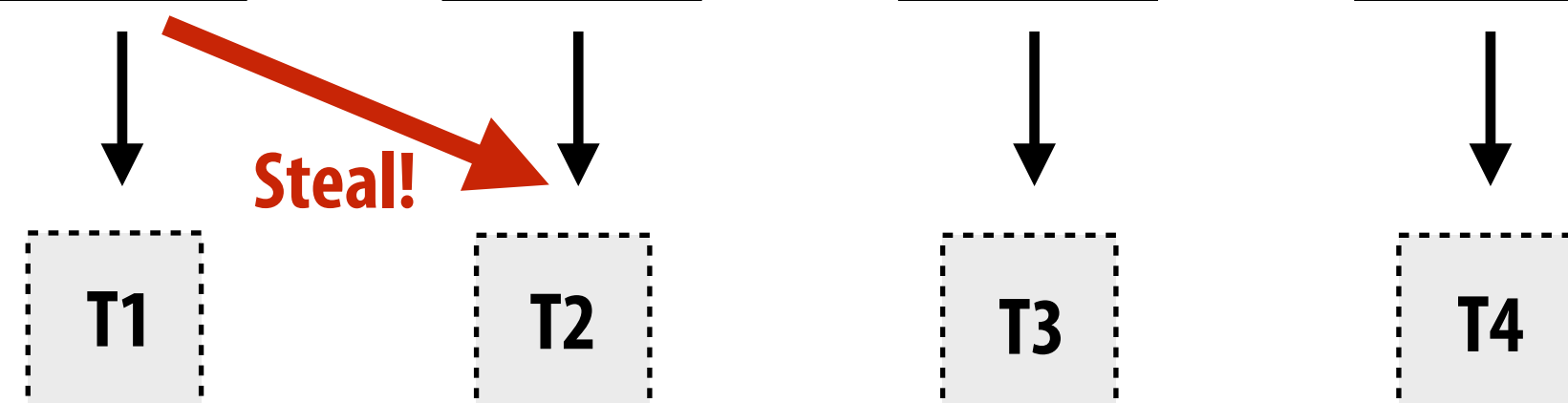
(In general, one per worker thread)

Worker threads:

Pull data from OWN work queue

Push new work to OWN work queue

When local work queue is empty...

STEAL work from another work queue

Steal!

T1

T2

T3

T4

Example: create grid of particles data structure on large parallel machine (e.g, a GPU)

- **Problem: place 1M point particles in a 16-cell uniform grid based on 2D position**
  - **Parallel data structure manipulation problem: build a 2D array of lists**

- **GTX 980 GPU: Up to 2048 CUDA threads per SMM core, and 16 SMM cores on the GPU**

0	**1**	**2**	**3**
4	**5**	**6**	**7**
8	**9**	**10**	**11**
12	**13**	**14**	**15**

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**
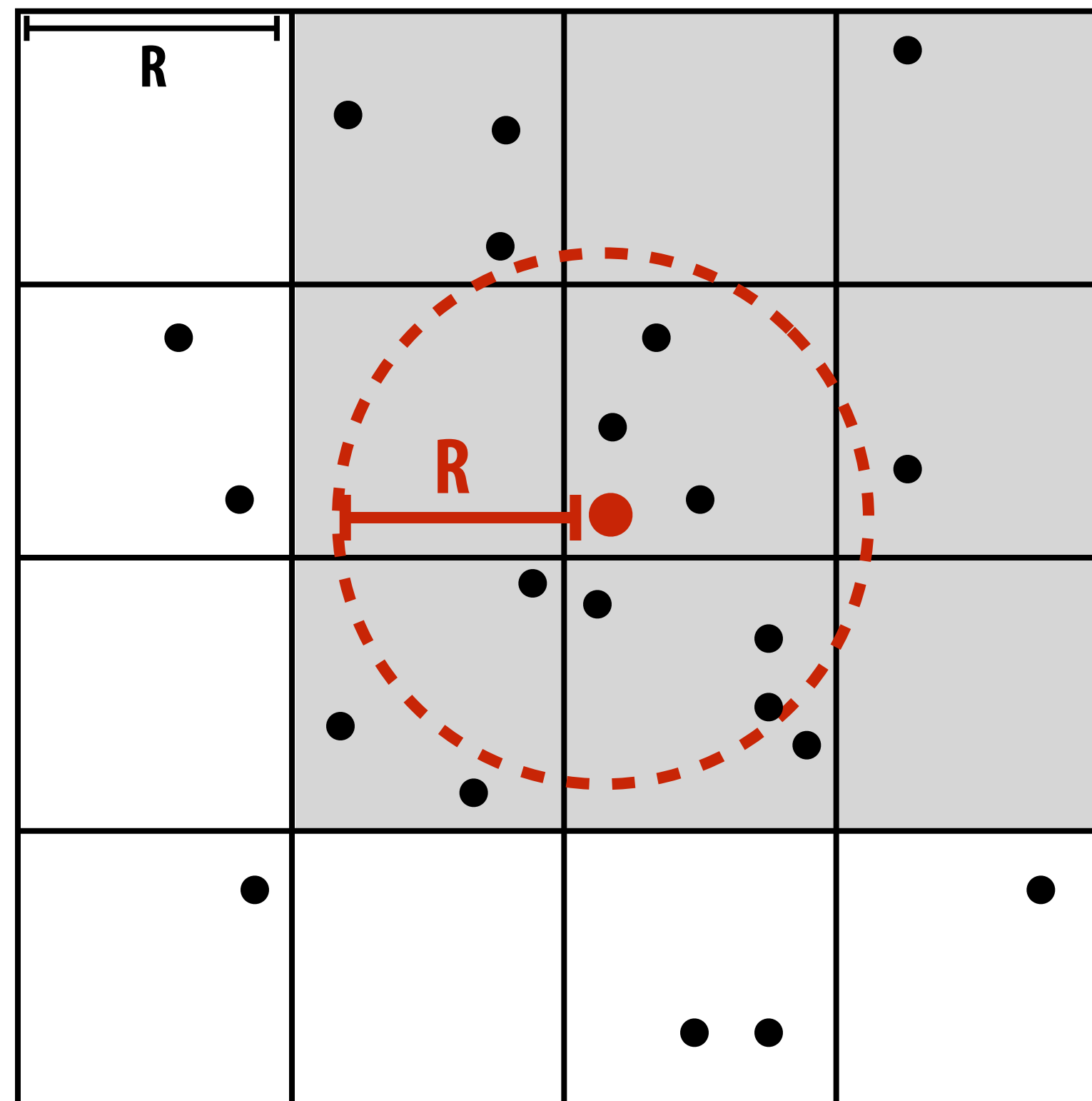
Solution 1: parallelize over cells

- **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 to efficiently utilize 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
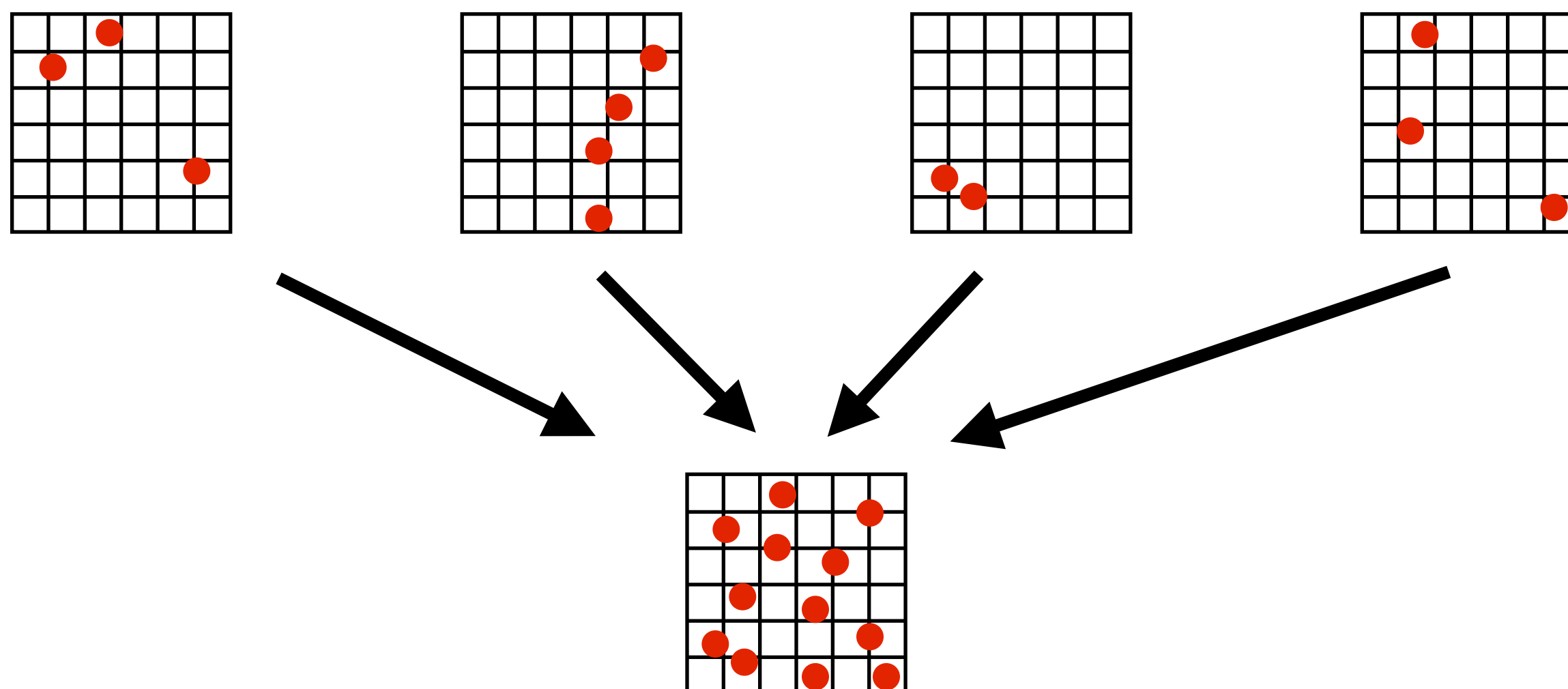
Solution 5: data-parallel approach

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Step 1: compute cell containing each particle (parallel over input 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 maintains a large amount of parallelism and 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 over particle_index elements)

```
cell = grid_index[index]
if (index == 0)
    cell_starts[cell] = index;
else if (cell != grid_index[index-1]) {
    cell_starts[cell] = index;
    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 particle_index array (each innovation has a unique valid of 'index')

cell_starts

				0		2			5	· · ·

cell_ends (not inclusive)

				2		5			6	· · ·

0   1   2   3   4   5   6   7   8   9   10

Reducing communication costs

- **Reduce overhead** of communication 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 communication/computation 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 given how the problem is decomposed and how work is assigned

  - Artifactual communication depends on machine implementation details (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)