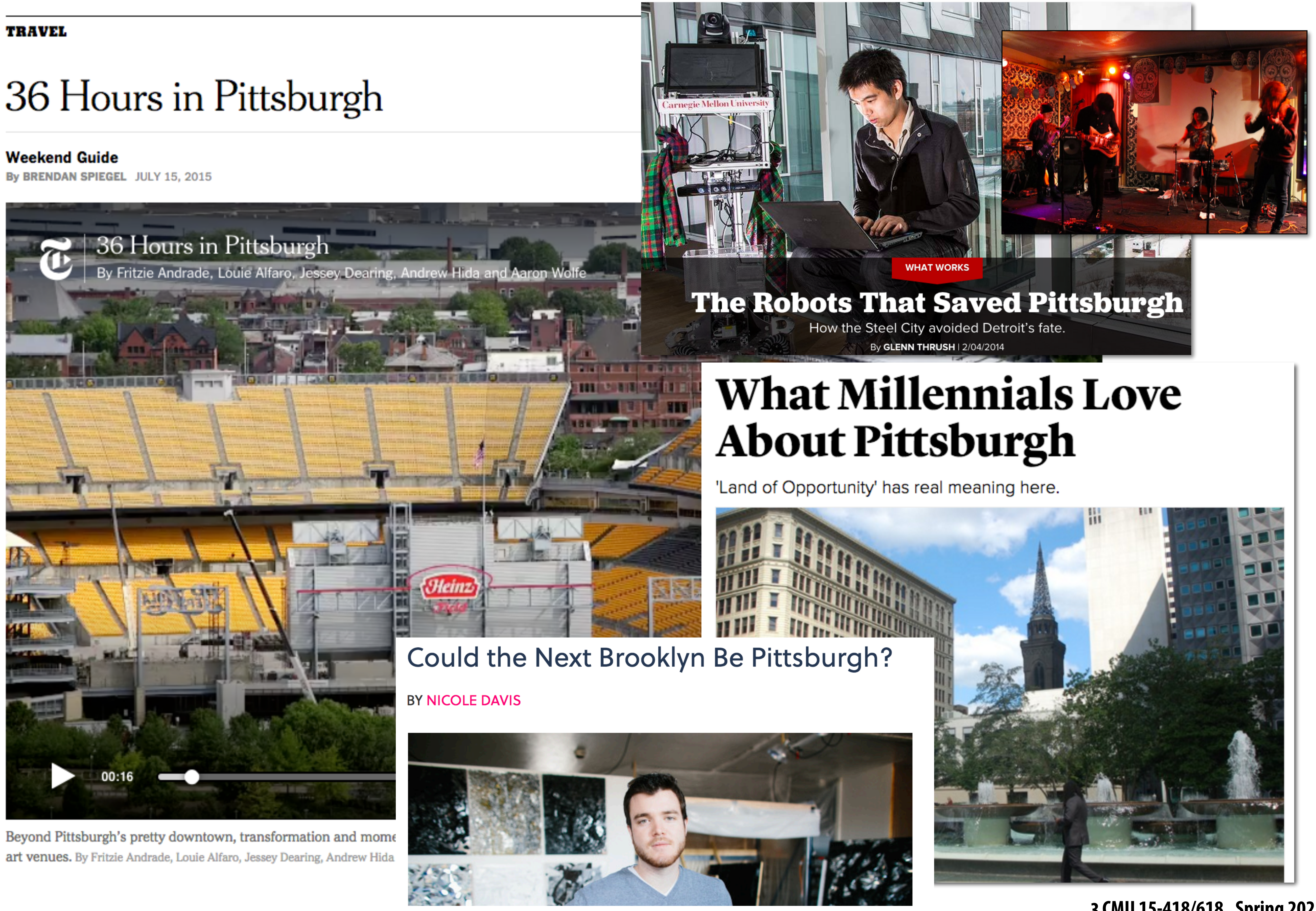
Lecture 7:

Performance Optimization Part II: Locality, Communication, and Contention

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2024

Let's talk about…
Pittsburgh

Pittsburgh is now hot stuff!



TRAVEL

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

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

WHAT WORKS

The Robots That Saved Pittsburgh

How the Steel City avoided Detroit's fate.

By **GLENN THRUSH** | 2/04/2014

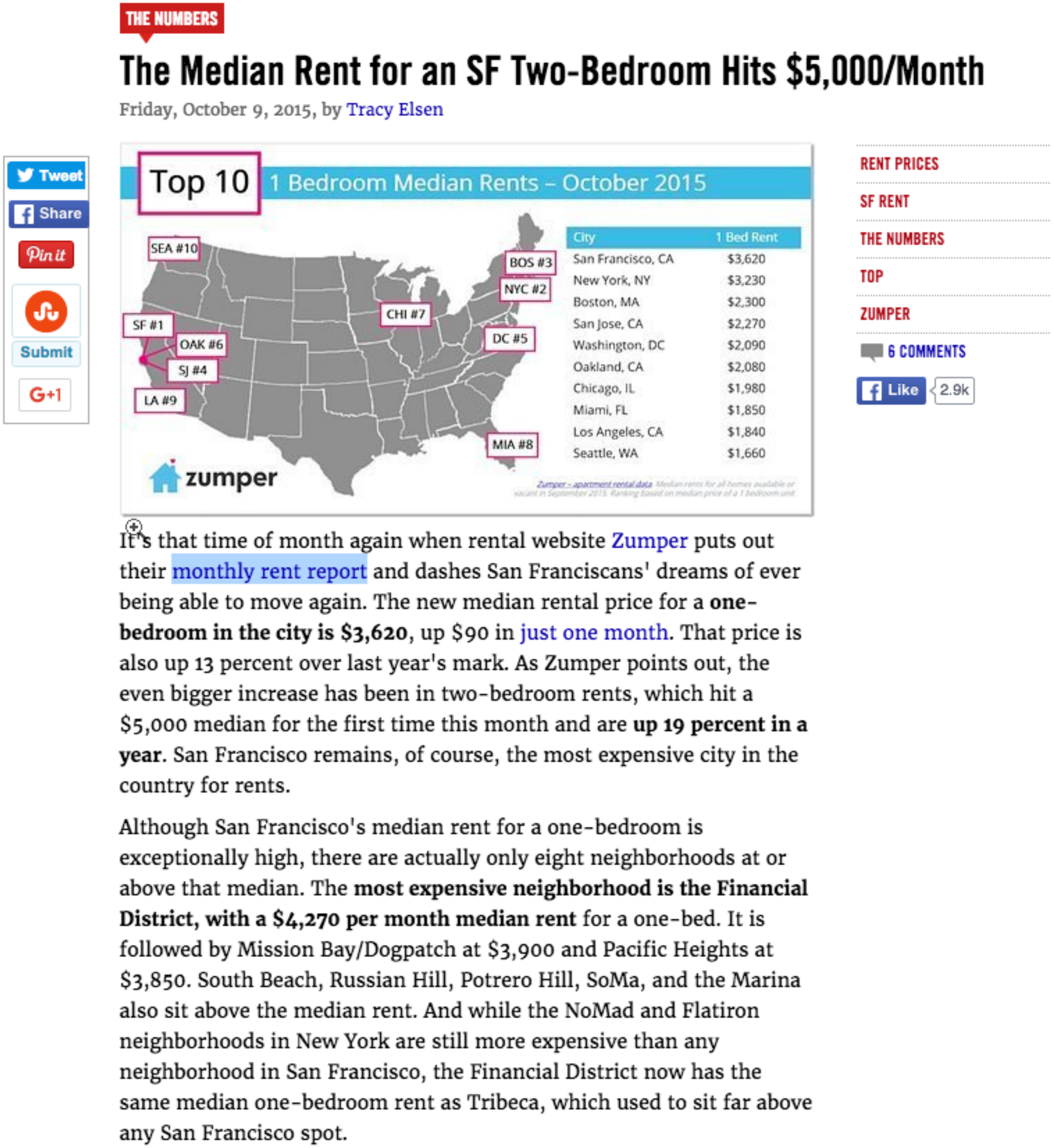
What Millennials Love About Pittsburgh

'Land of Opportunity' has real meaning here.

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

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

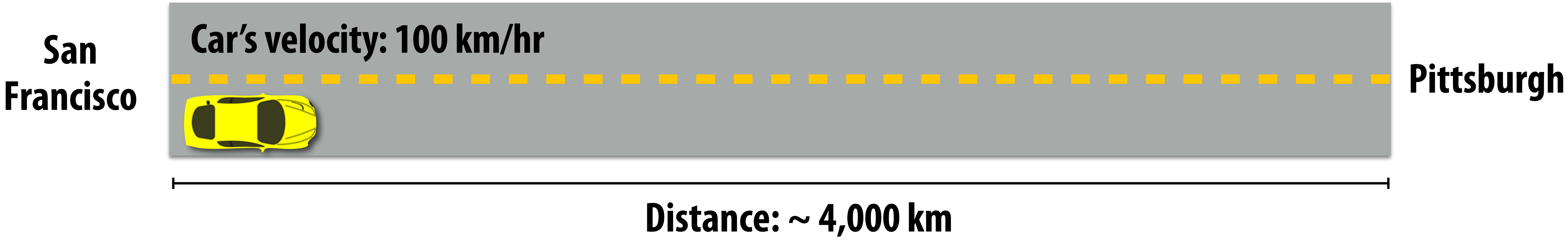
It's that time of month again when rental website Zumper puts out their monthly rent report and dashes San Franciscans' dreams of ever being able to move again. The new median rental price for a **one-bedroom in the city is $3,620**, up $90 in just one month. That price is also up 13 percent over last 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.

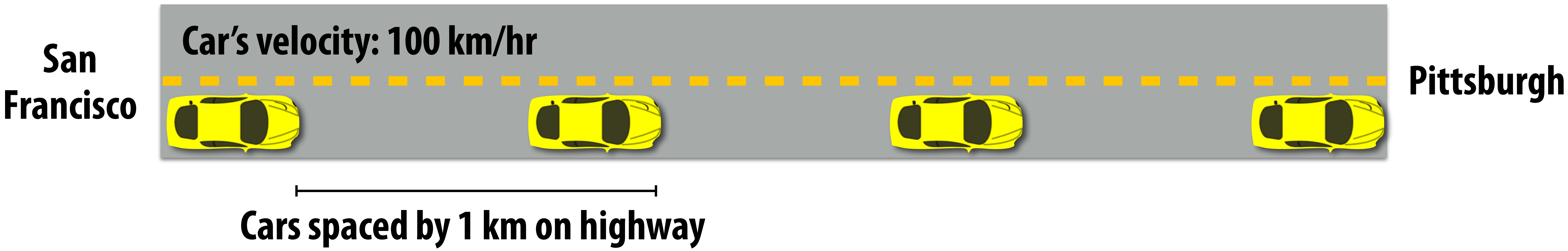
Hey, let's move to Pittsburgh!
(all the cool tech kids are doing it!)

Everyone wants to get to Pittsburgh!

(Latency vs. throughput review)

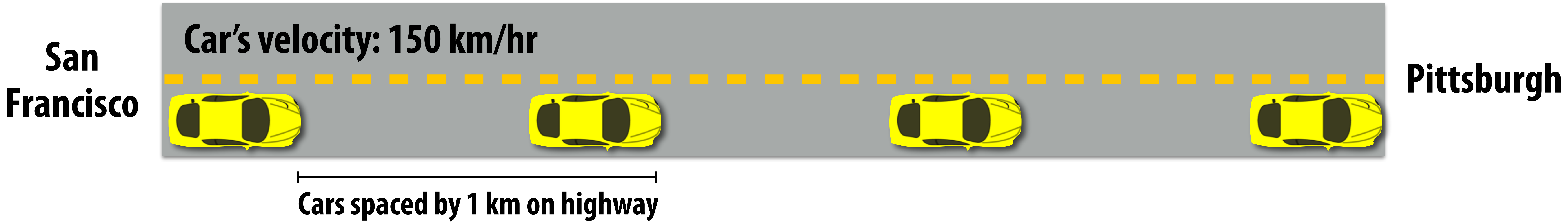


Latency of moving a person from San Francisco to Pittsburgh: 40 hours



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

Improving throughput



San Francisco

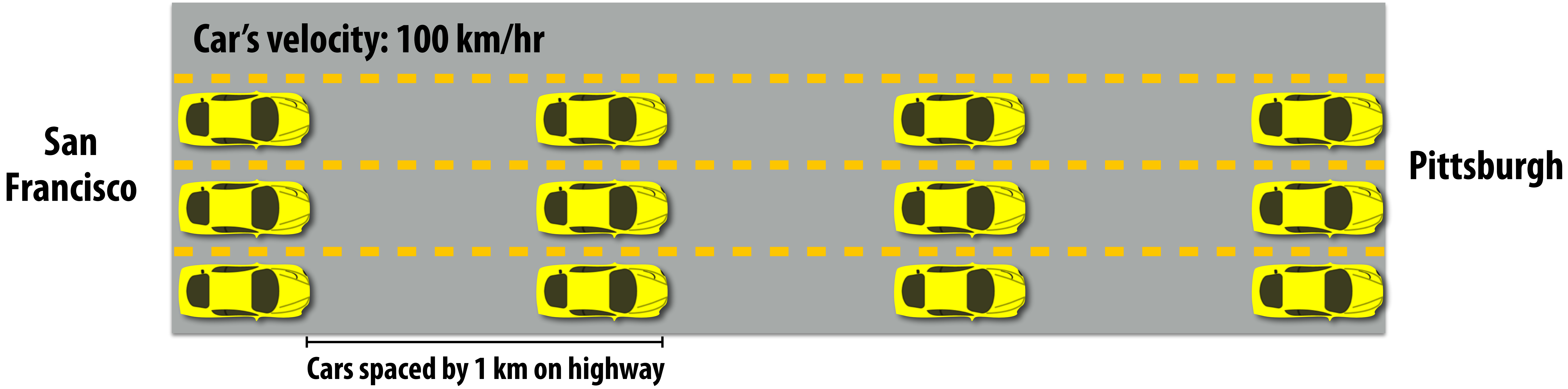
Car’s velocity: 150 km/hr

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)



San Francisco

Car’s velocity: 100 km/hr

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.

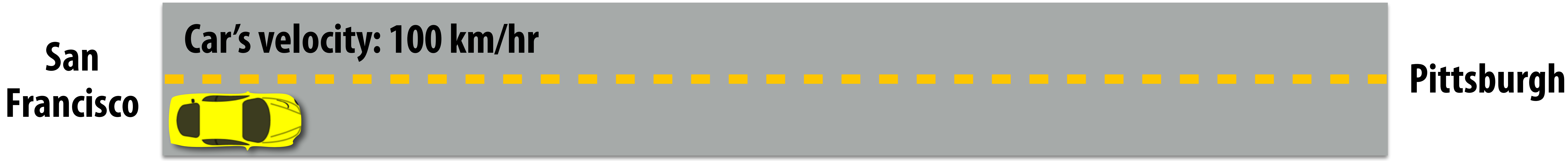
Memory can provide data to the processor at 25 GB/sec.

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.



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

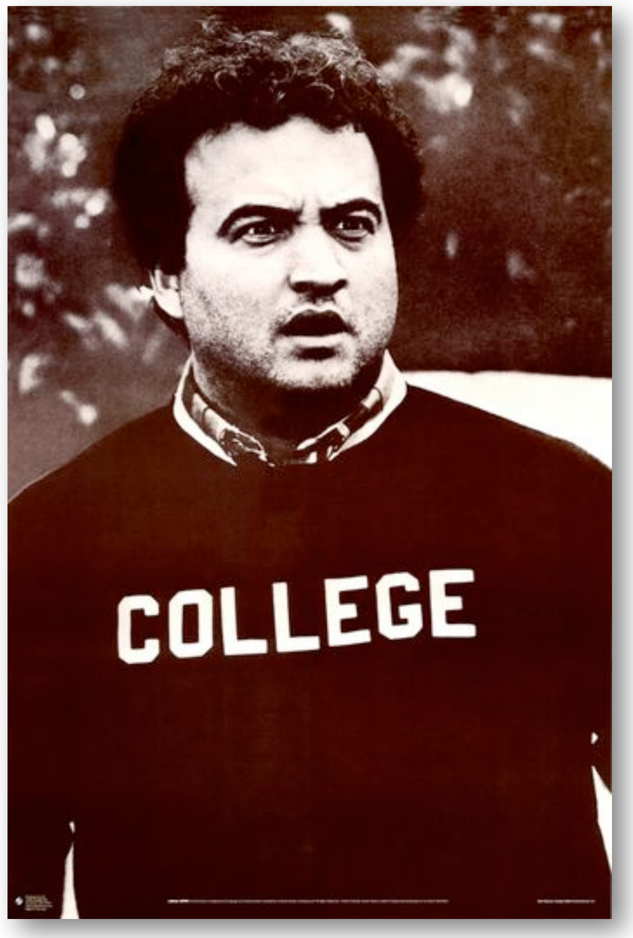
Example: doing your laundry

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

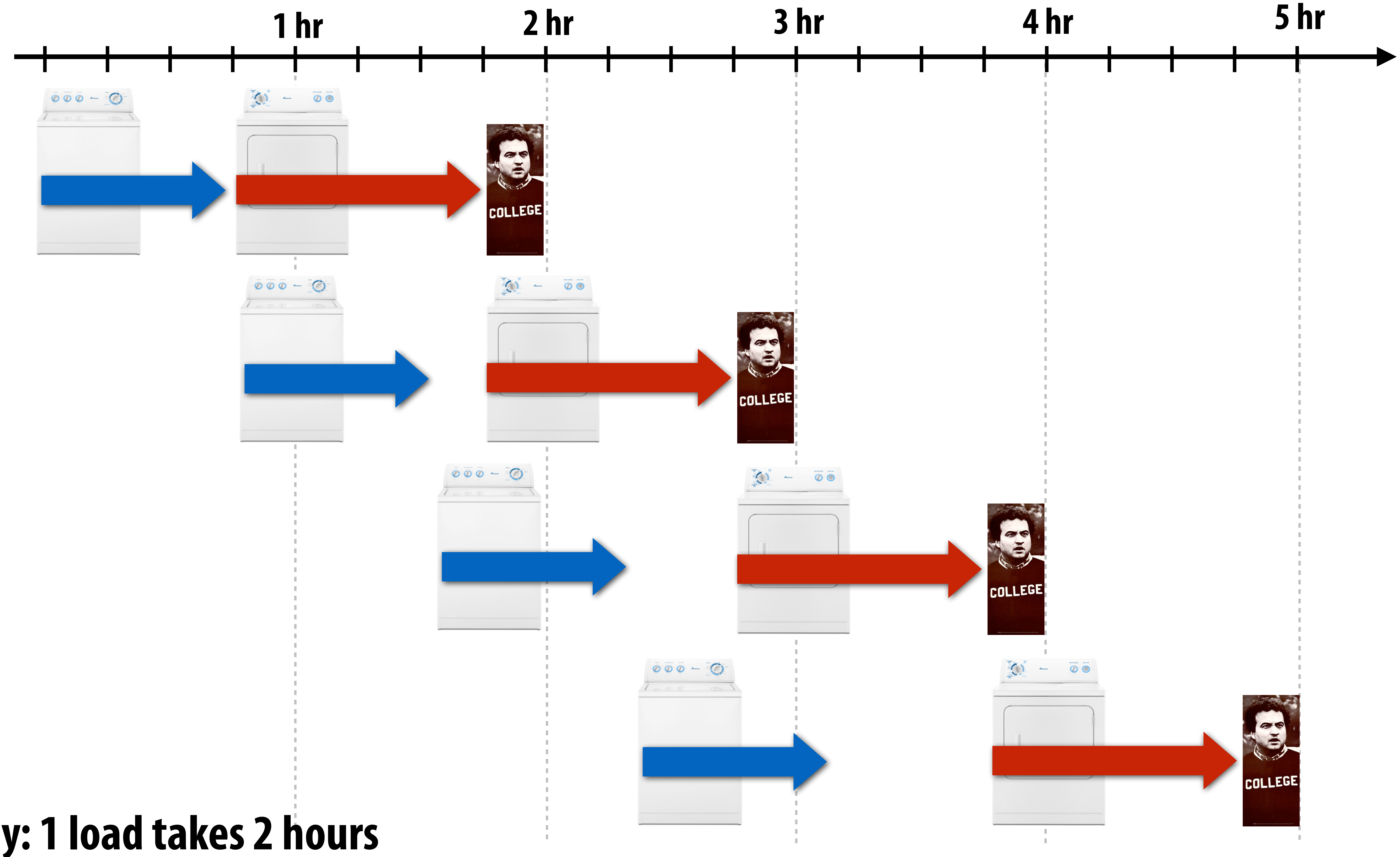
Latency of completing 2 loads of laundry = 2 hours

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

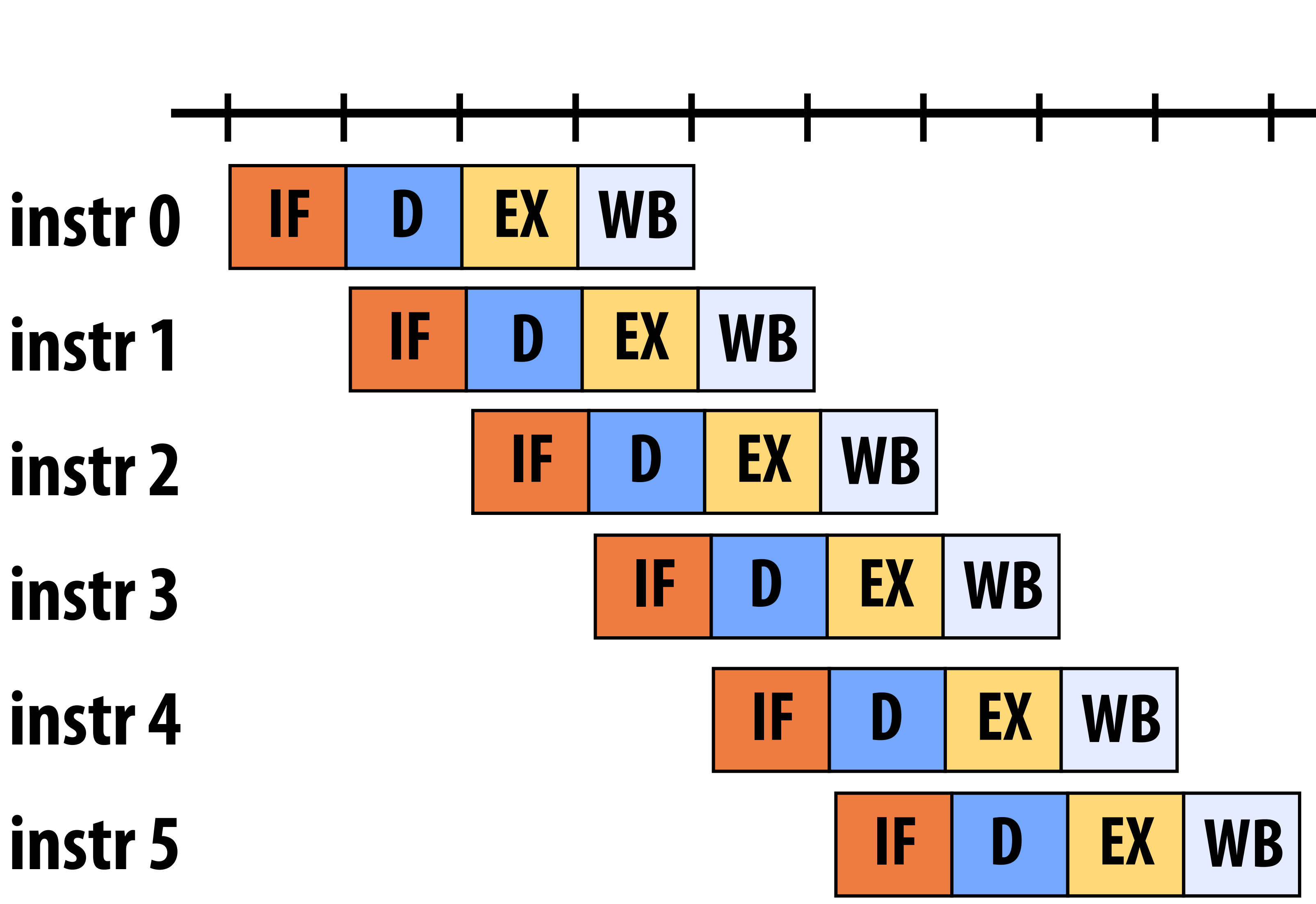
Throughput: 1 load/hour

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)



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

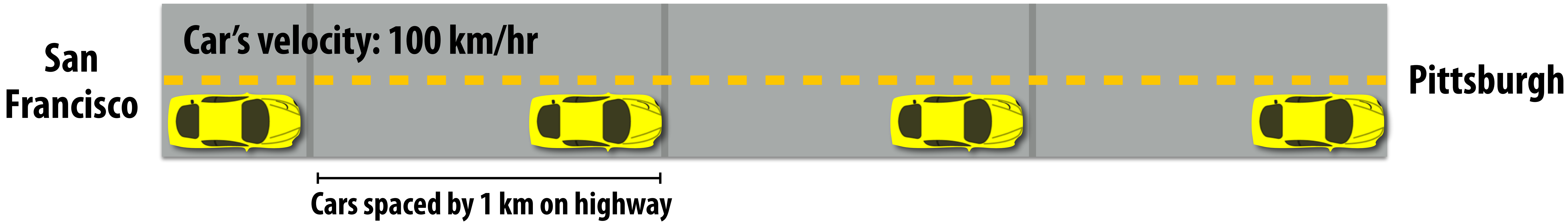
Throughput: 1 instruction per cycle

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

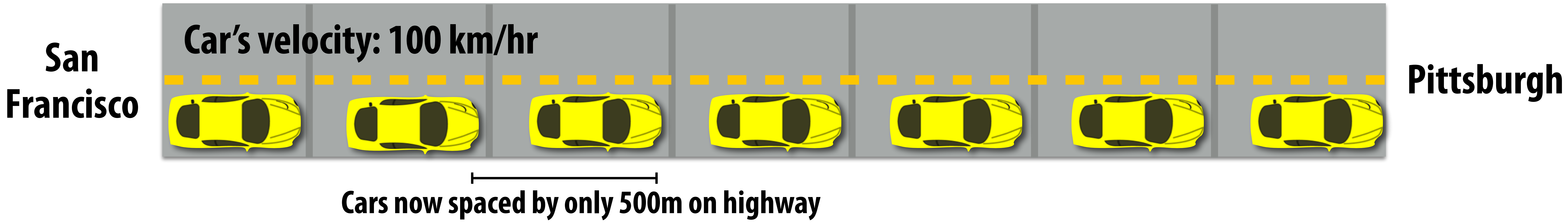
Intel Core i7 pipeline is variable length (it depends on the instruction) ~15-20 stages

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)



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 an 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)

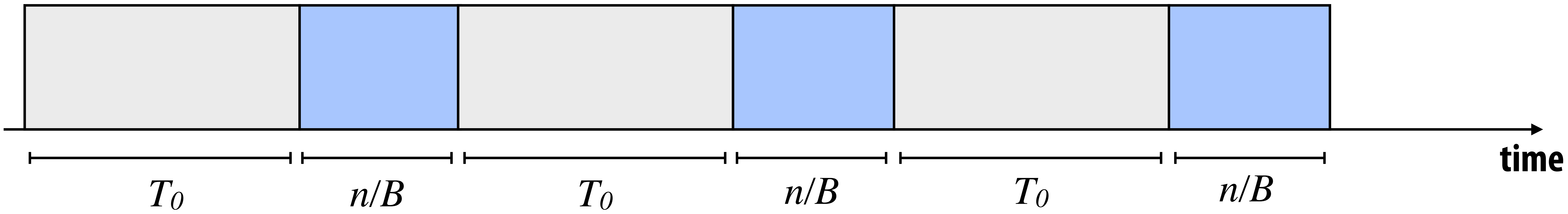
n = bytes transferred in operation

B = transfer rate (bandwidth of the link)

If processor only sends next message once previous message send completes…

"Effective bandwidth" = $n / T(n)$

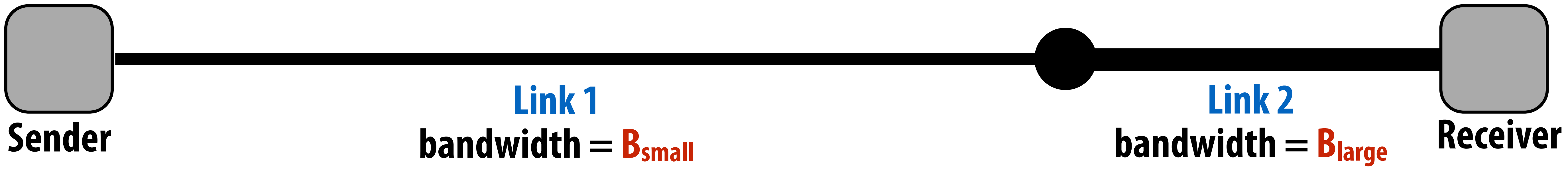
Effective bandwidth depends on transfer size (big transfers amortize startup latency)

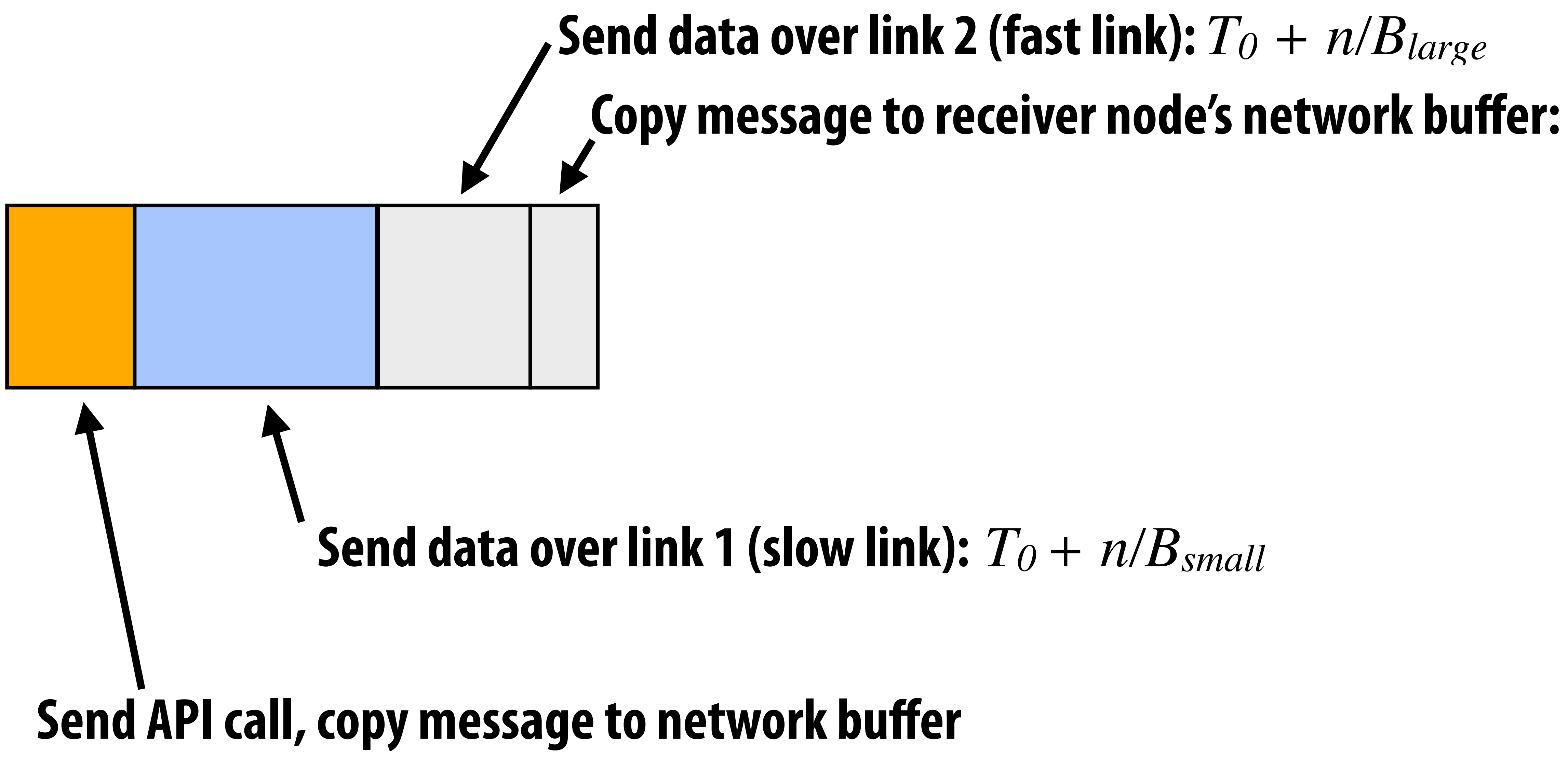


A more general model of communication

Example: sending a n-bit message

Total 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:

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)

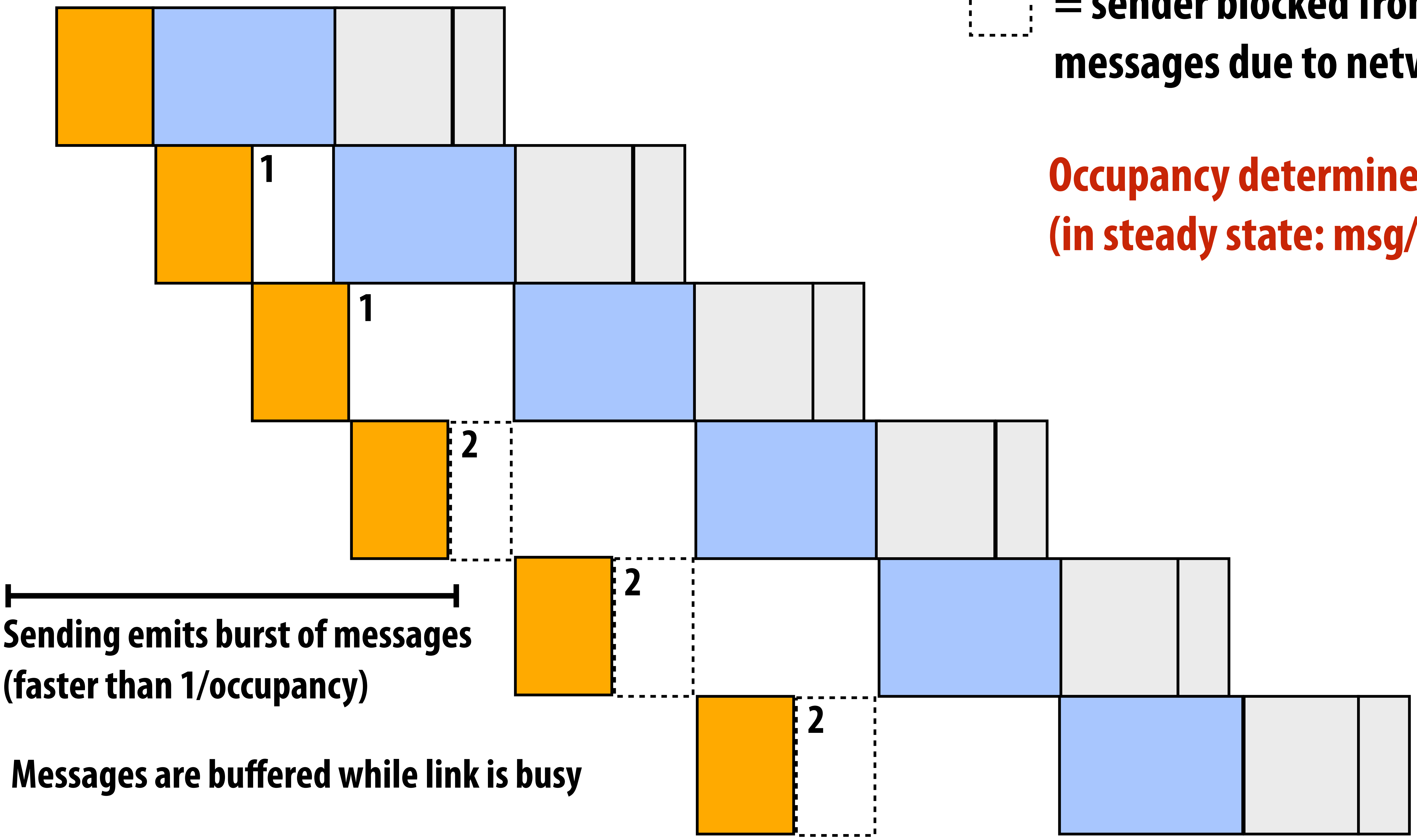
⬜ = Network delay (everything else)

Example from: Culler, Singh, and Gupta

Pipelined communication

⬚ = 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)

Example from: Culler, Singh, and Gupta

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

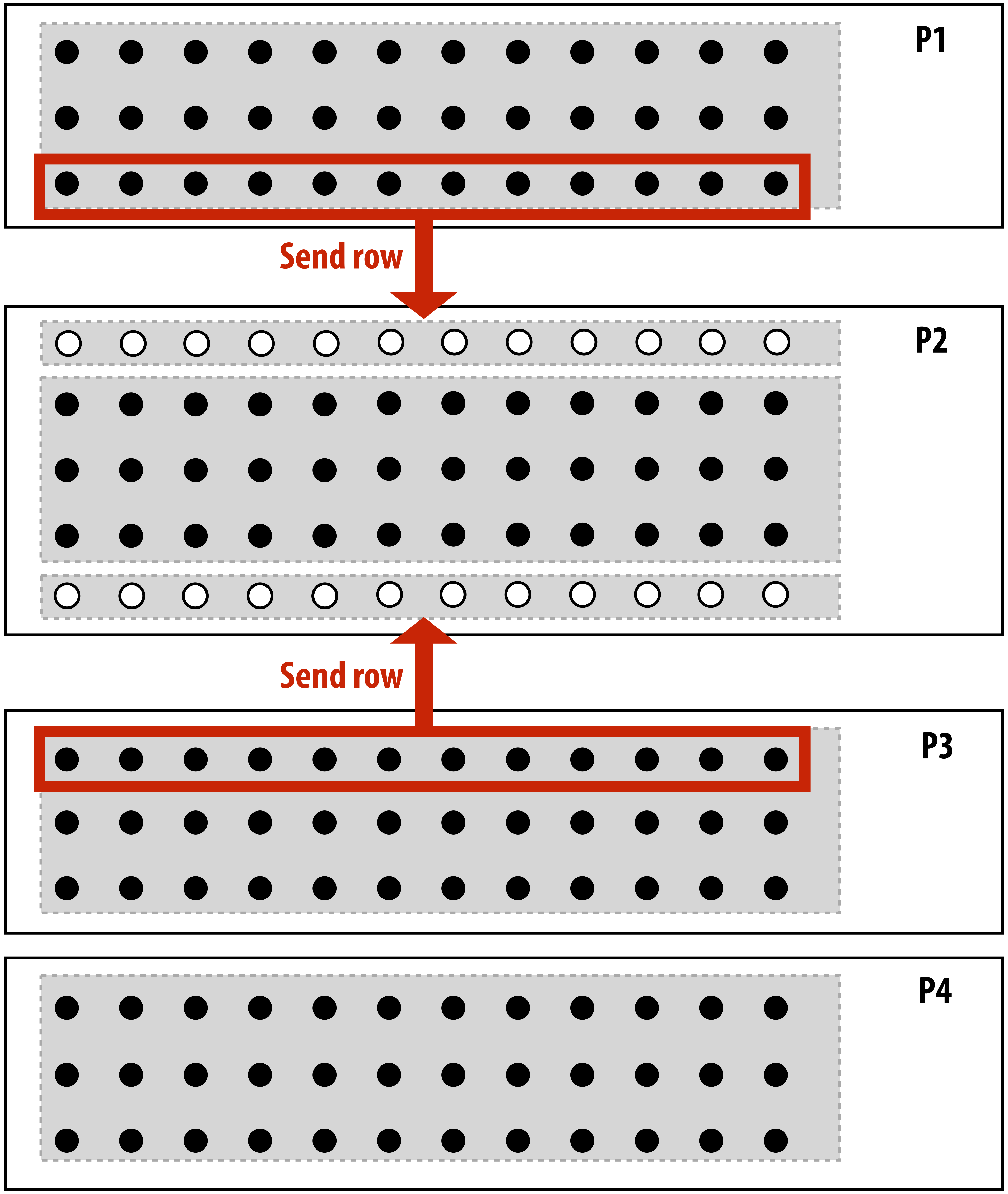
"Other work" can be computation or other communication

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

Example 2: Pipelining allows multiple message sends to be overlapped

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

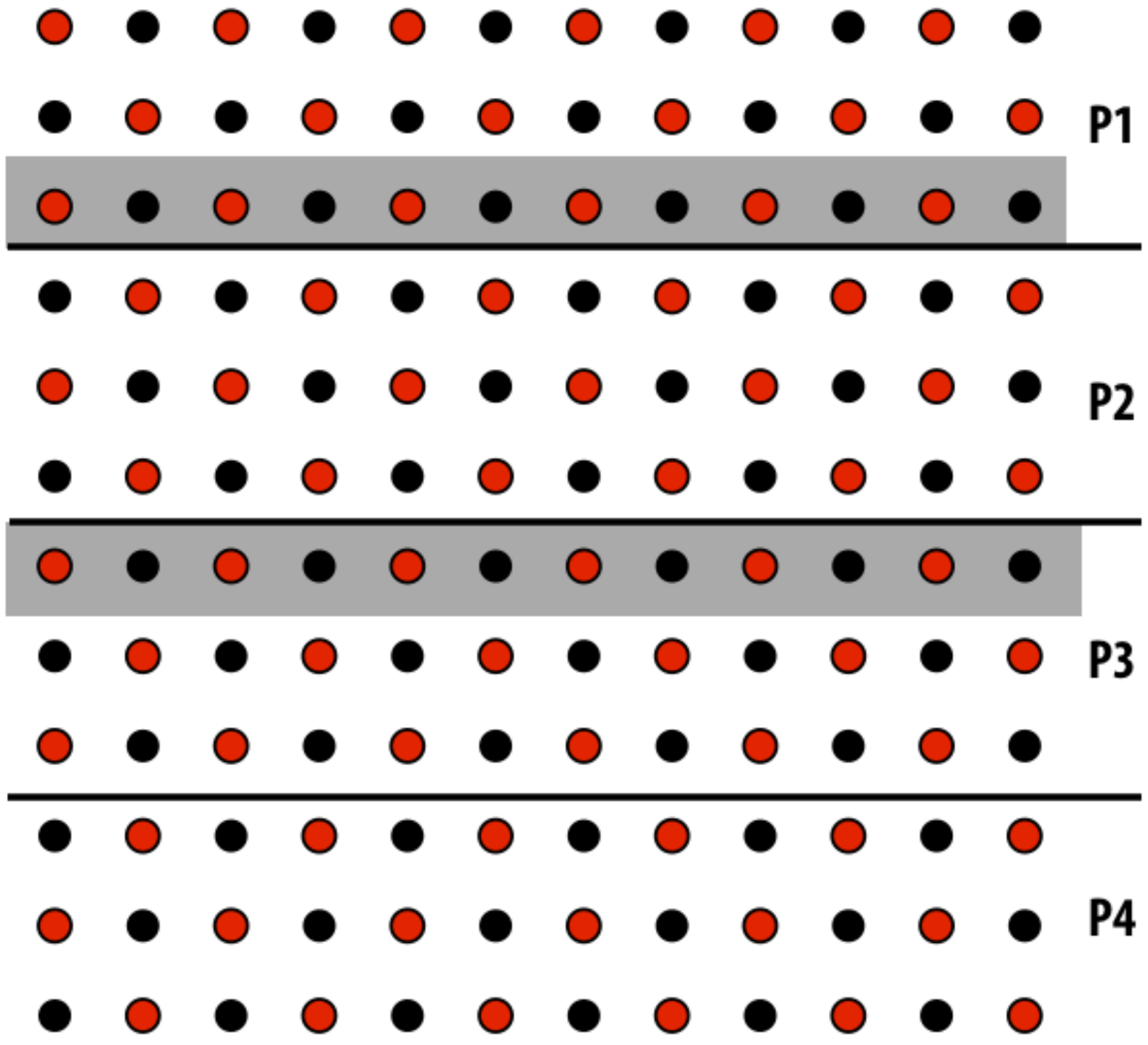
- "Arithmetic intensity" = 1 / communication-to-computation ratio
  - I find arithmetic intensity a more intuitive quantity, since higher is better.
  - It also sounds cooler

- 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

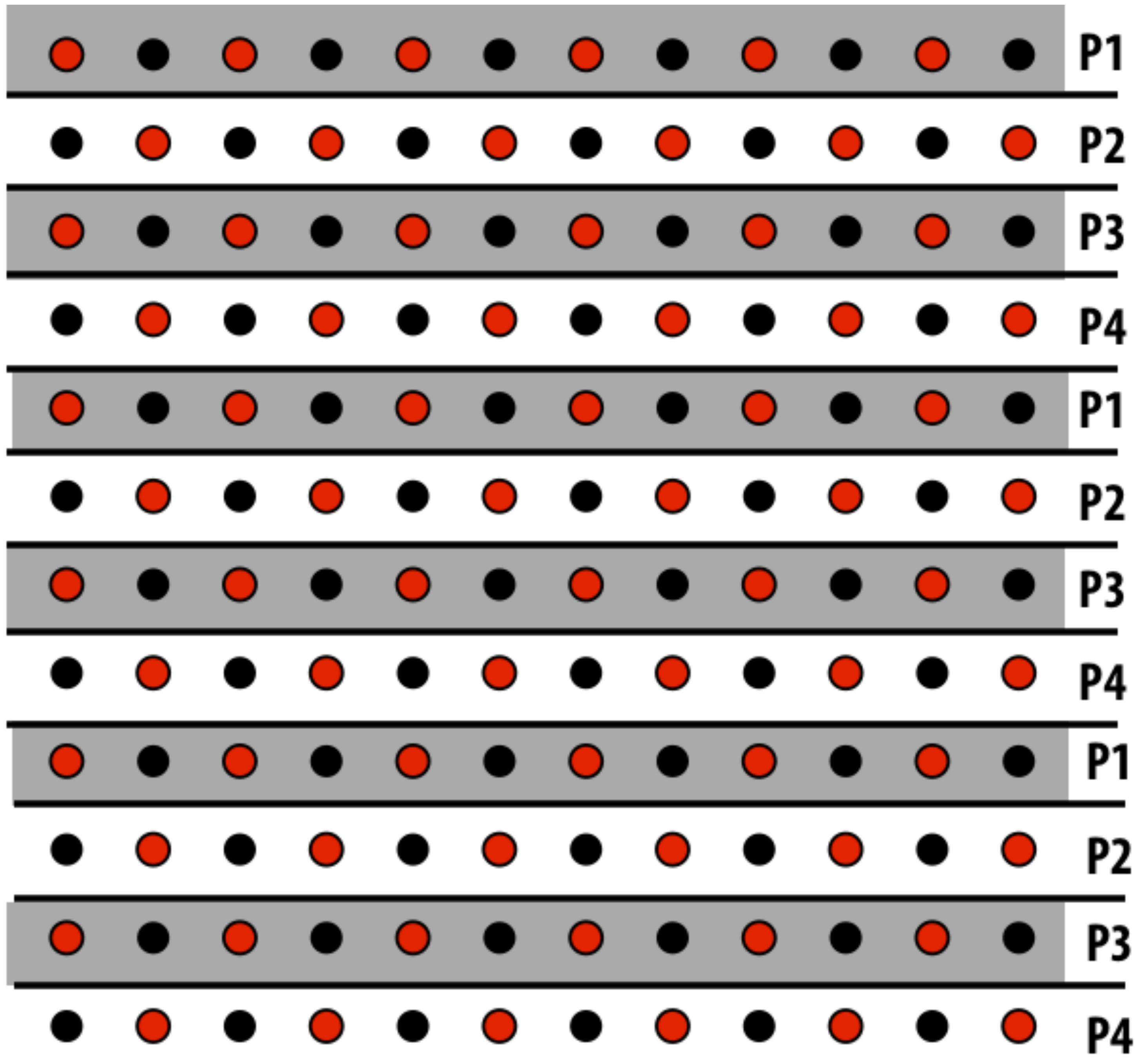
Good assignment decisions can reduce inherent communication (increase arithmetic intensity)

1D blocked assignment: N x N grid



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

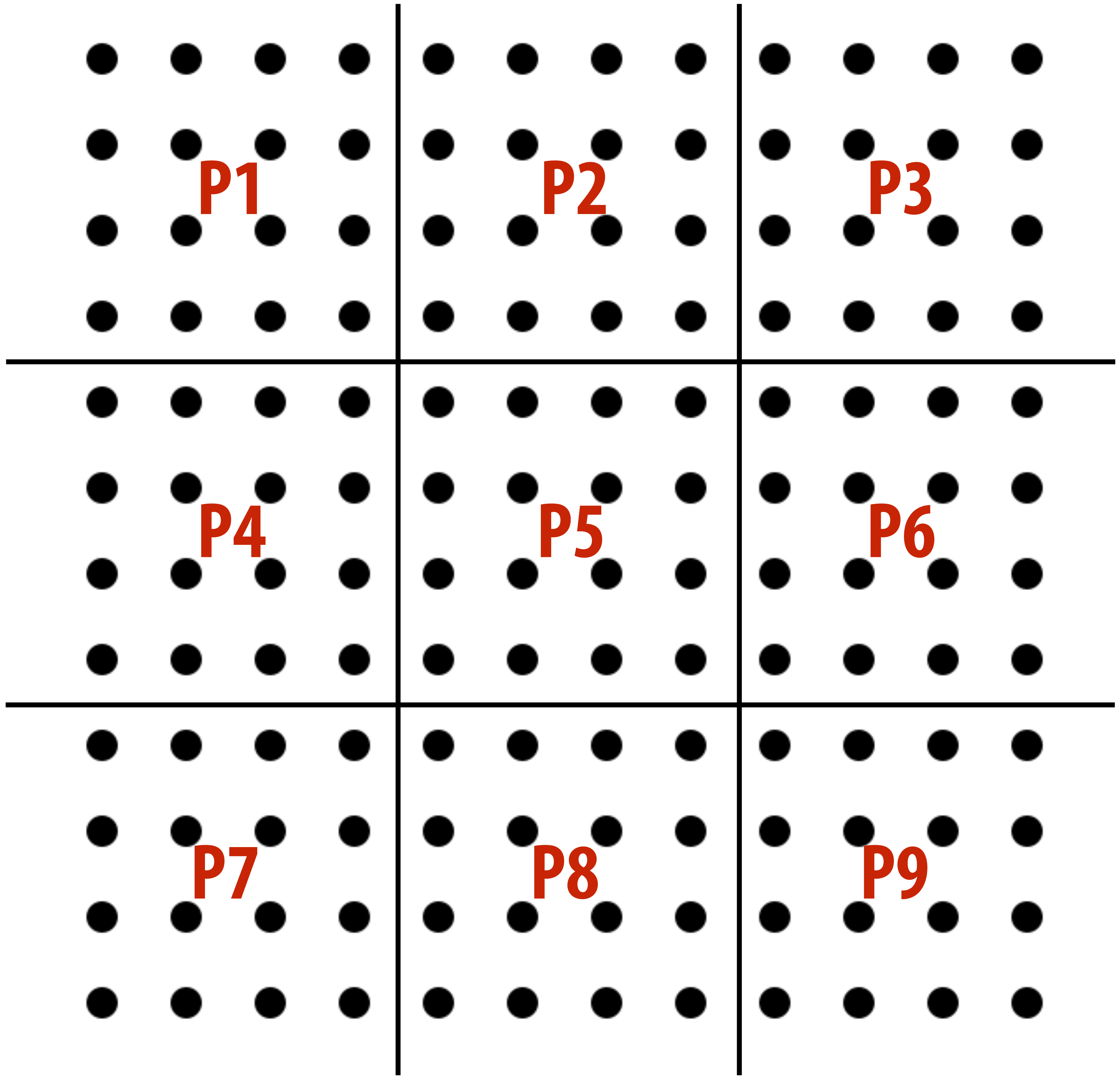
1D interleaved assignment: N x N grid



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

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)

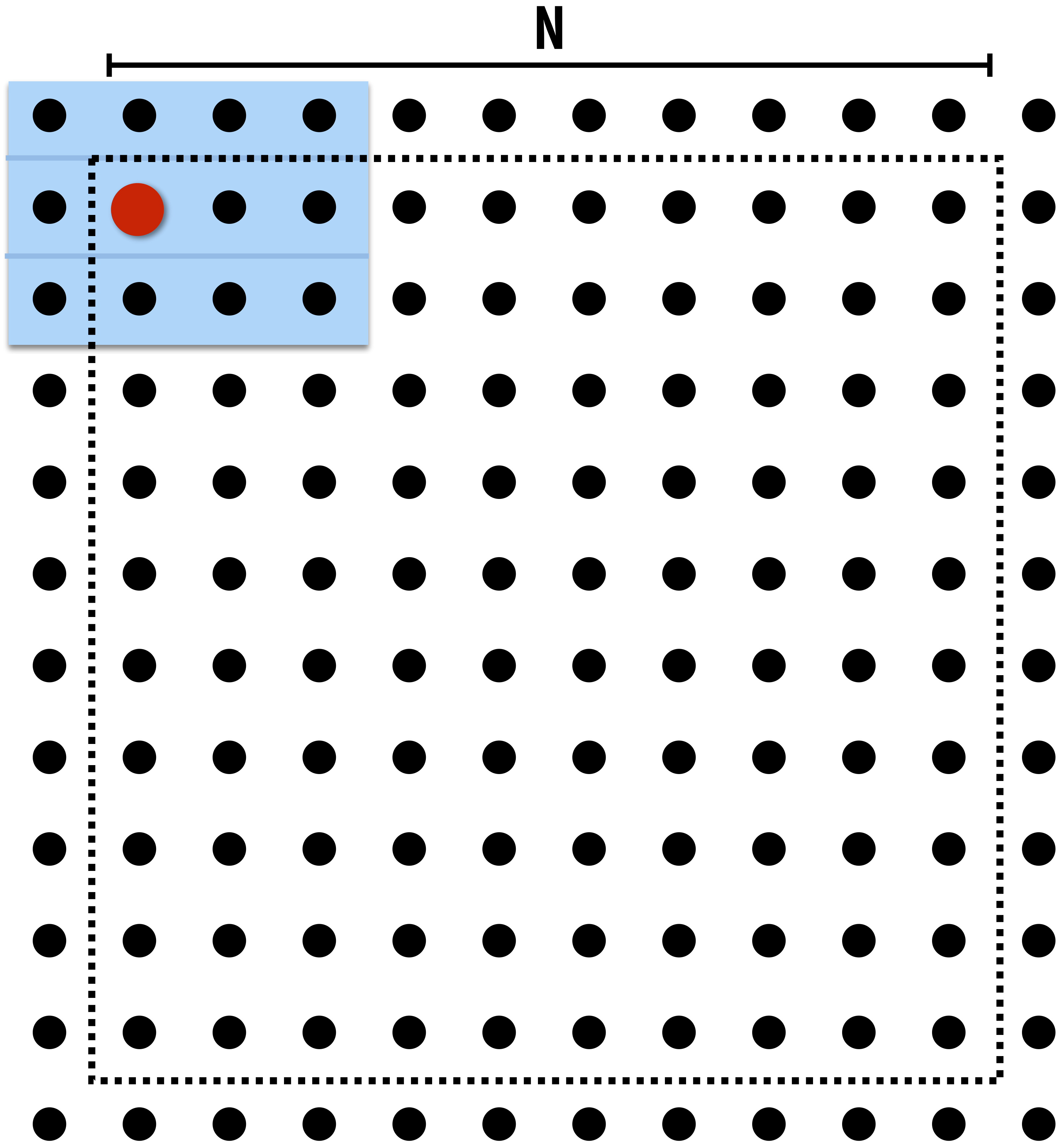
- System might have rules of operation that result in unnecessary communication:
  - Program stores 16 consecutive 4-byte float values, so entire 64-byte cache line is loaded from memory, and then subsequently stored to memory (2x overhead)

- Poor 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)

Techniques for reducing communication

Data access in grid solver: row-major traversal



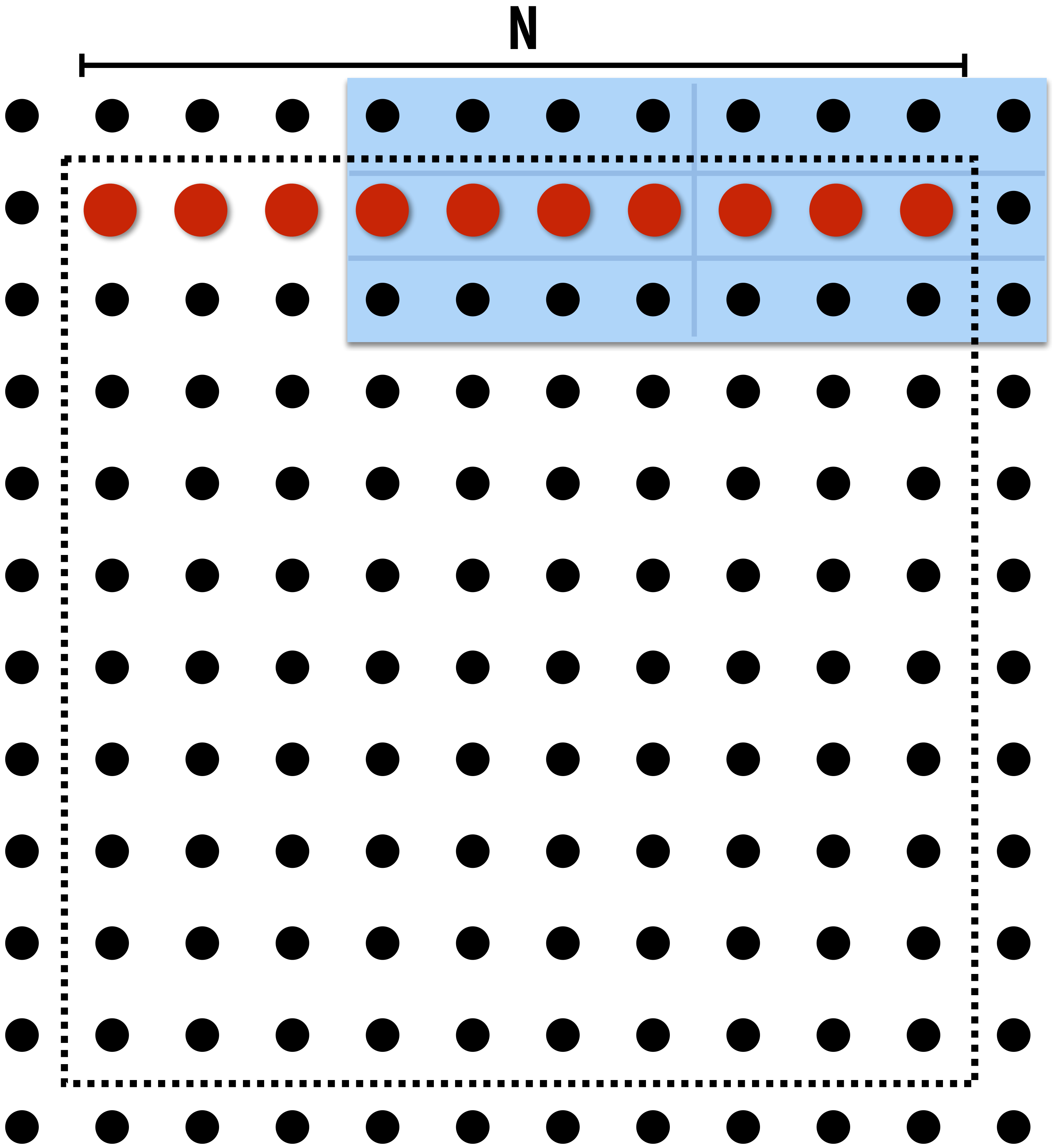
Assume row-major grid layout.

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



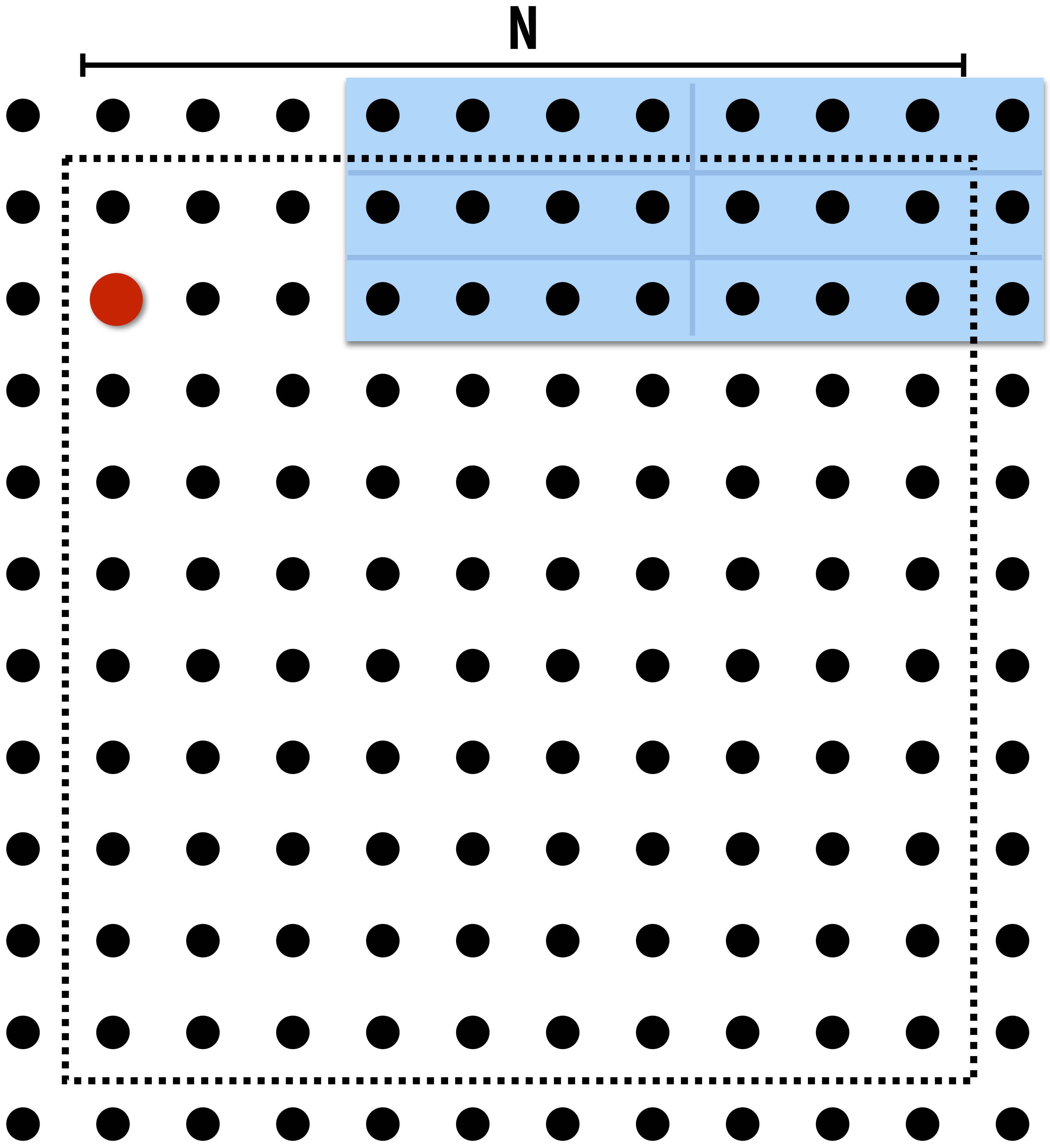
Assume row-major grid layout.

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.

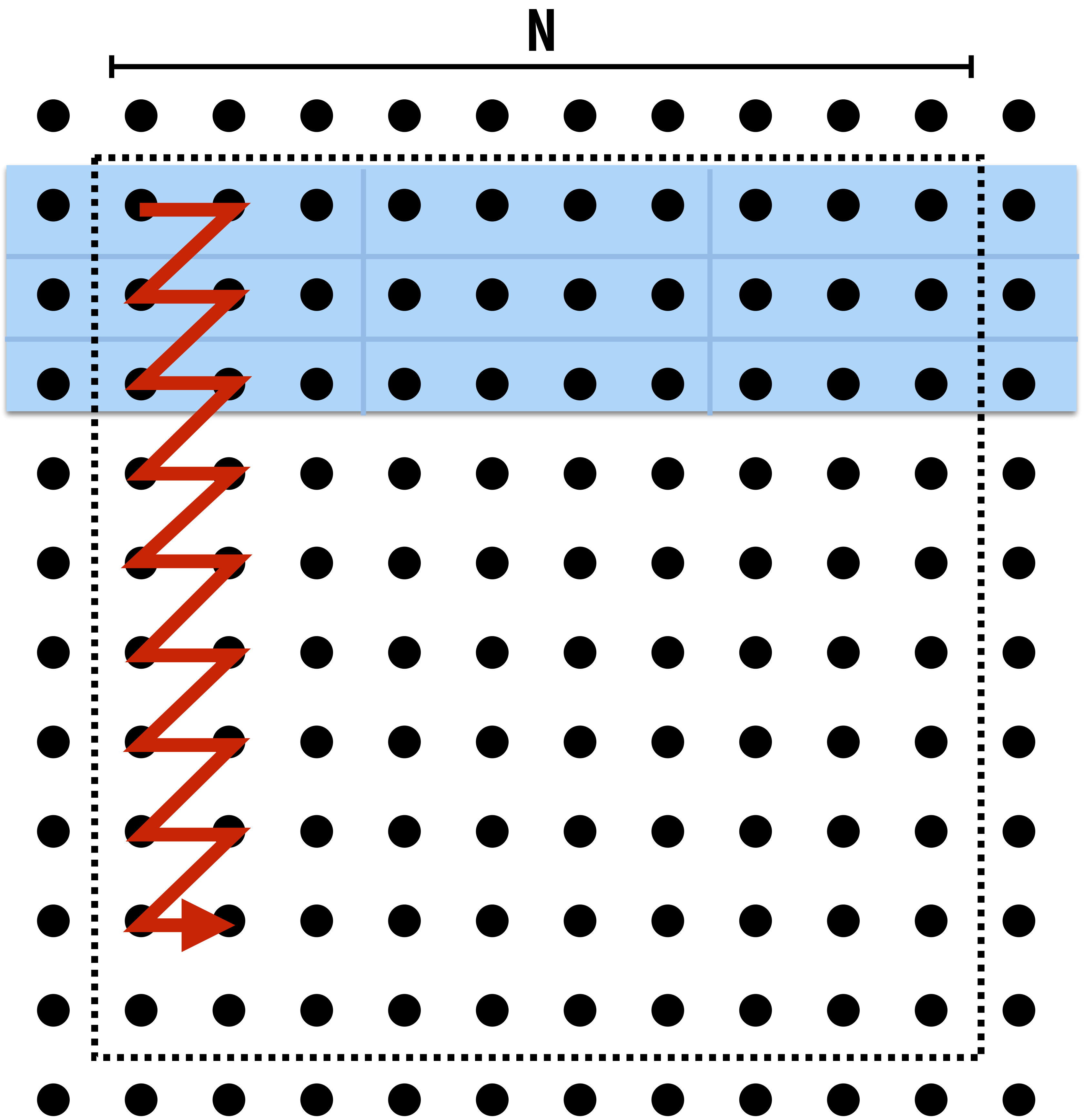
Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

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

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)

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, *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);
```

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(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

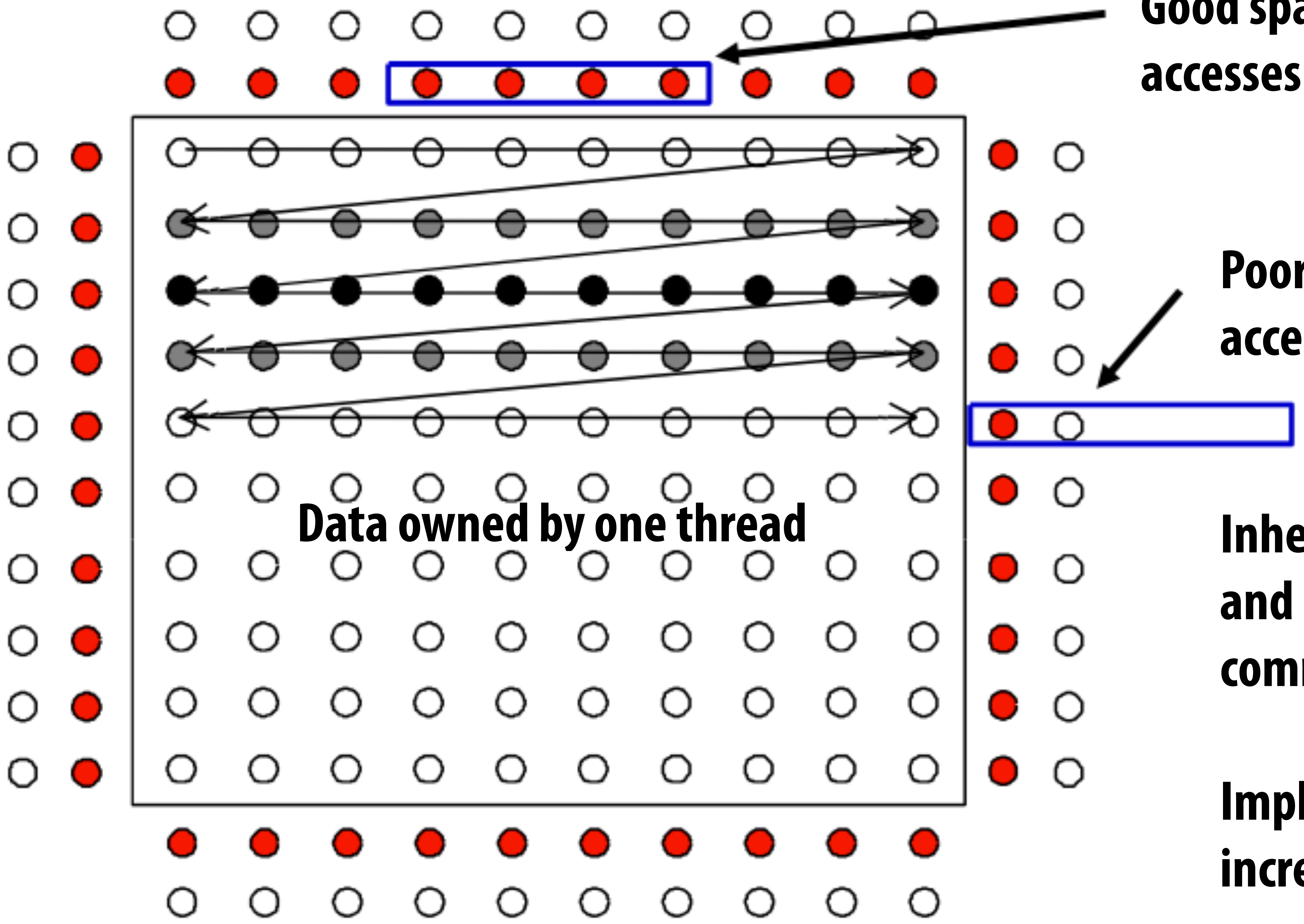
Exploiting spatial locality

- 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

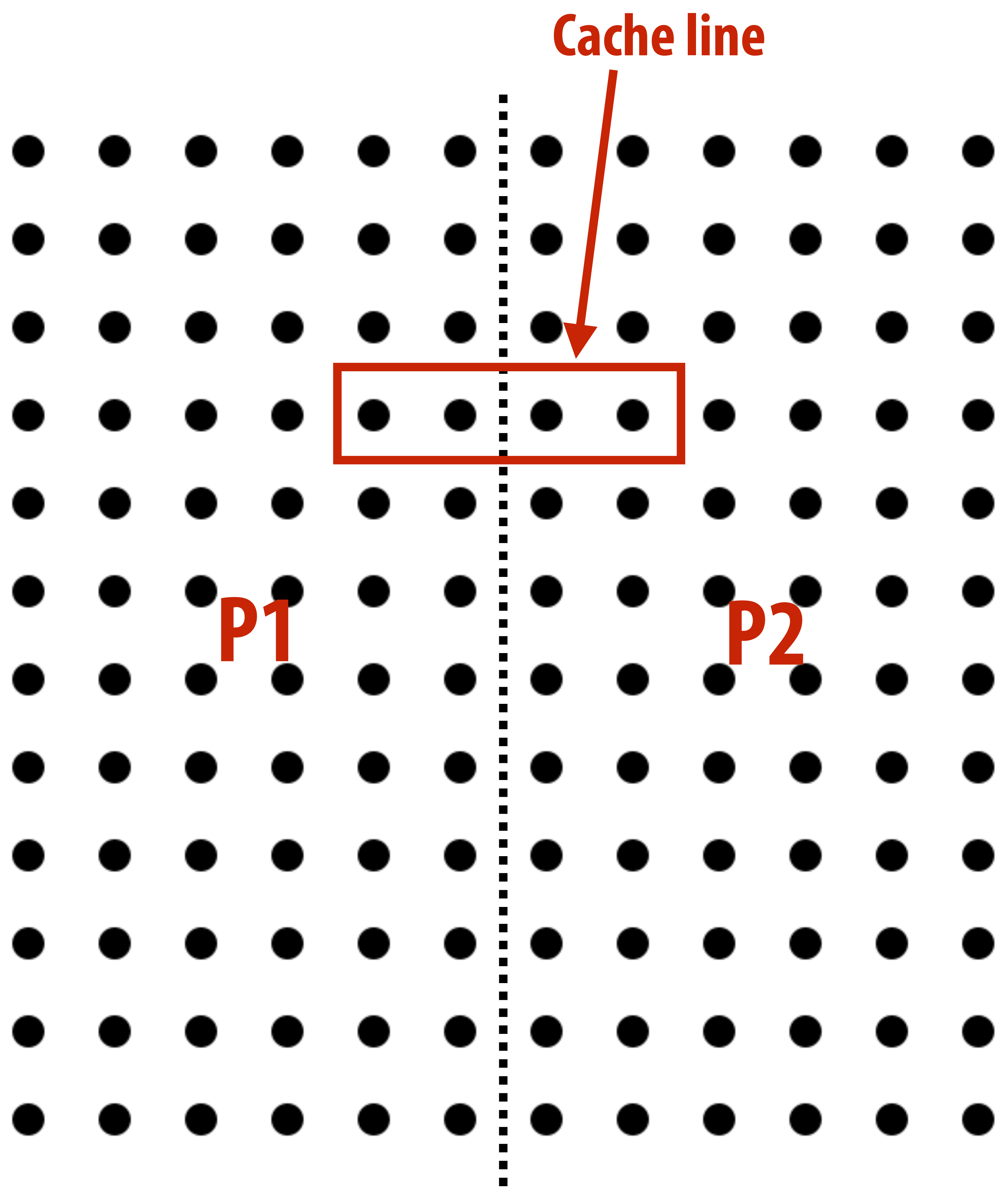
Poor spatial locality for non-local accesses to left-right columns

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



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

Threads access their assigned elements (no <u>inherent</u> communication exists)

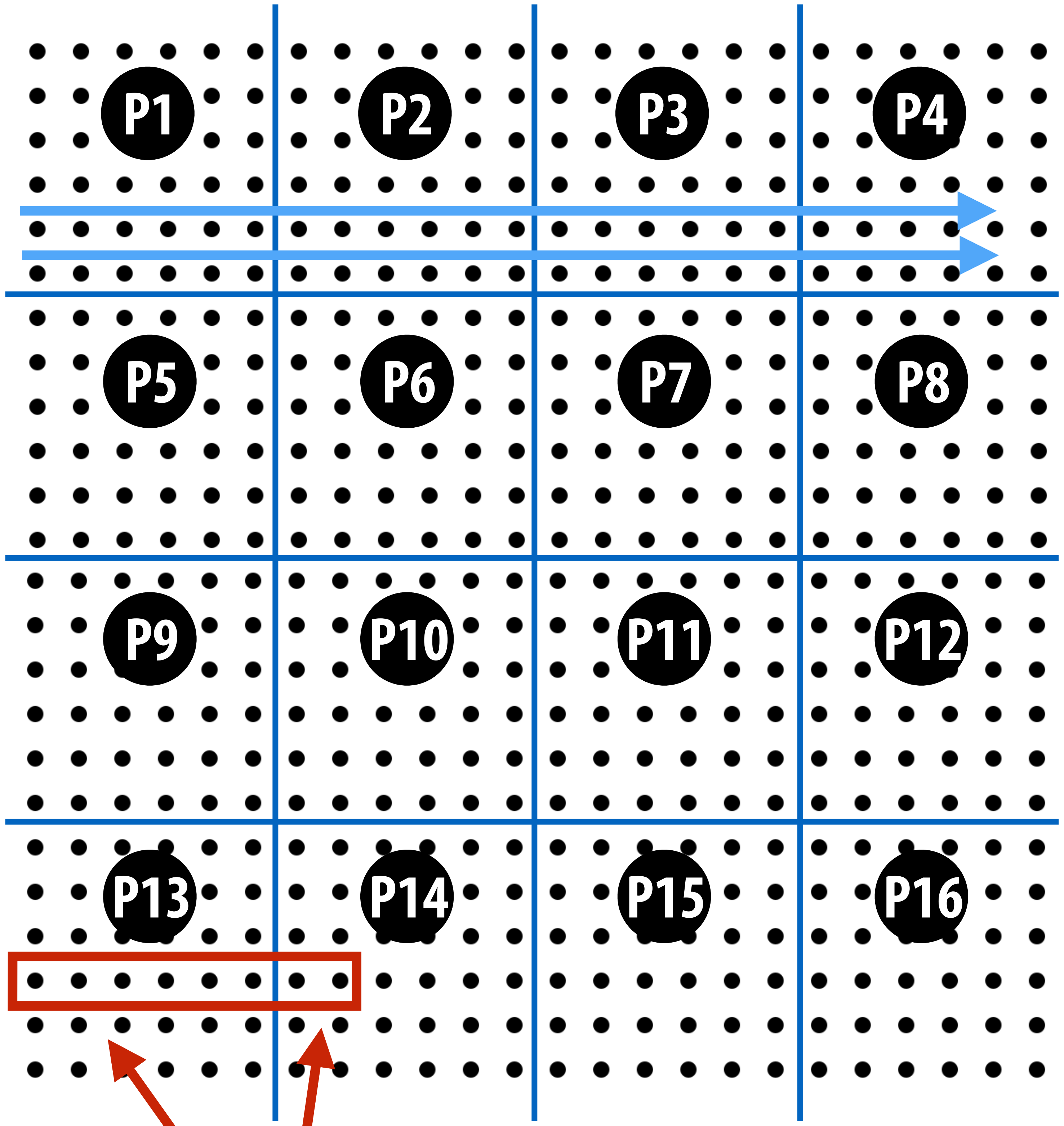
But data access on real machine triggers (artifactual) communication due to the cache line being written to by both processors *

* further detail in the upcoming cache coherence lectures

Reducing artifactual comm: blocked data layout

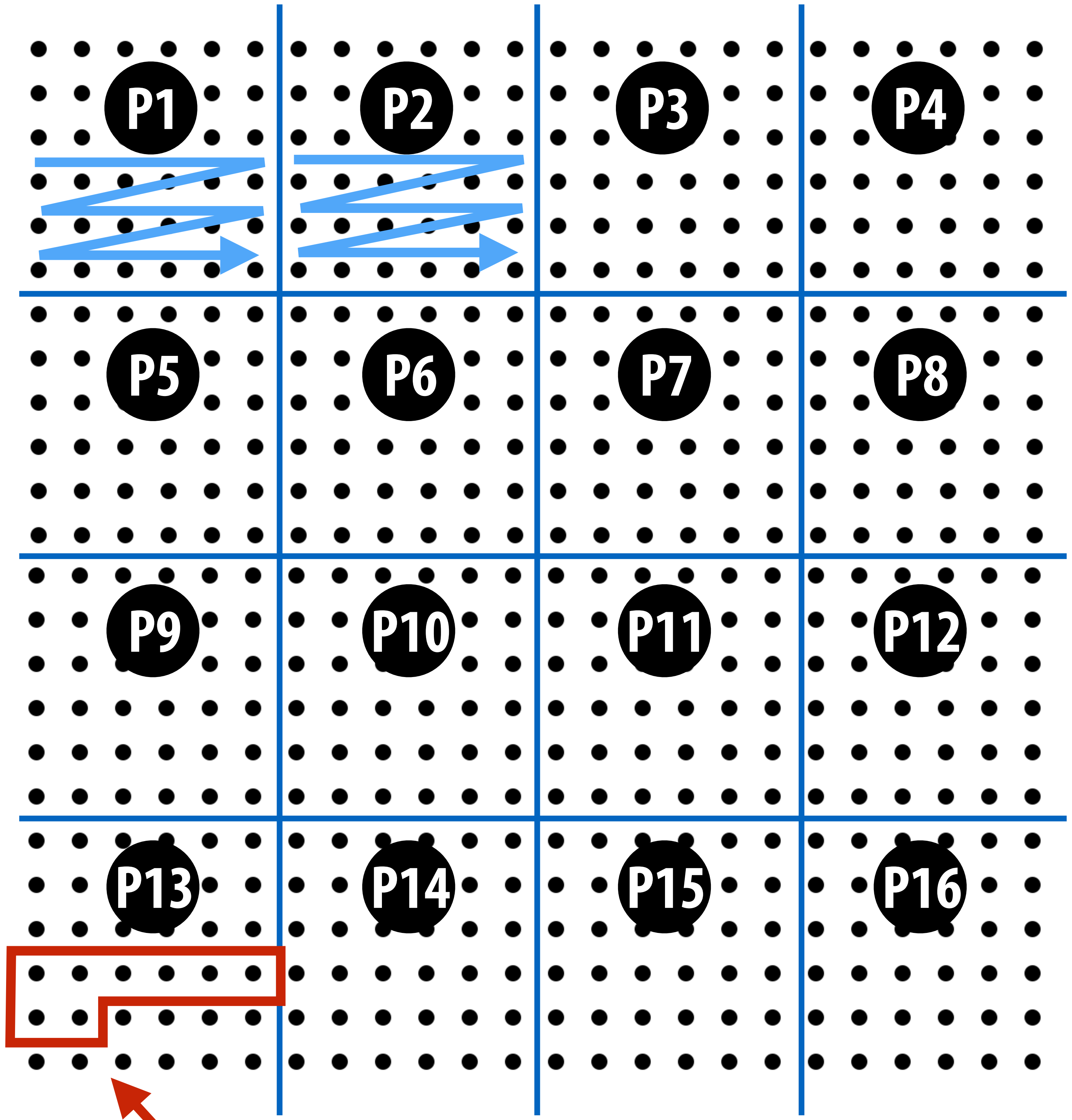
(Blue lines indicate consecutive memory addresses)

2D, row-major array layout



Consecutive addresses straddle partition boundary

4D array layout (block-major)



Consecutive addresses remain within single partition

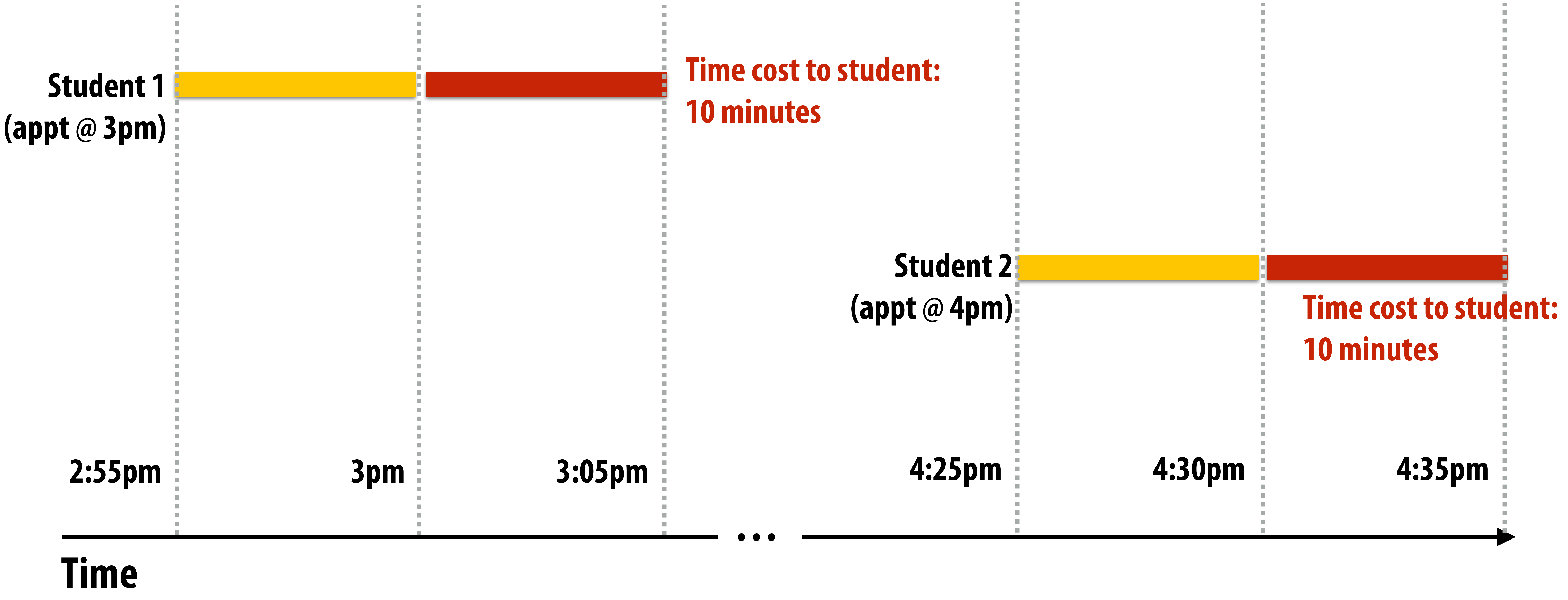
Better temporal locality

Less cache sharing between processors

Contention

Example: two students make appointments to talk to Prof. Mowry (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) = 🟥



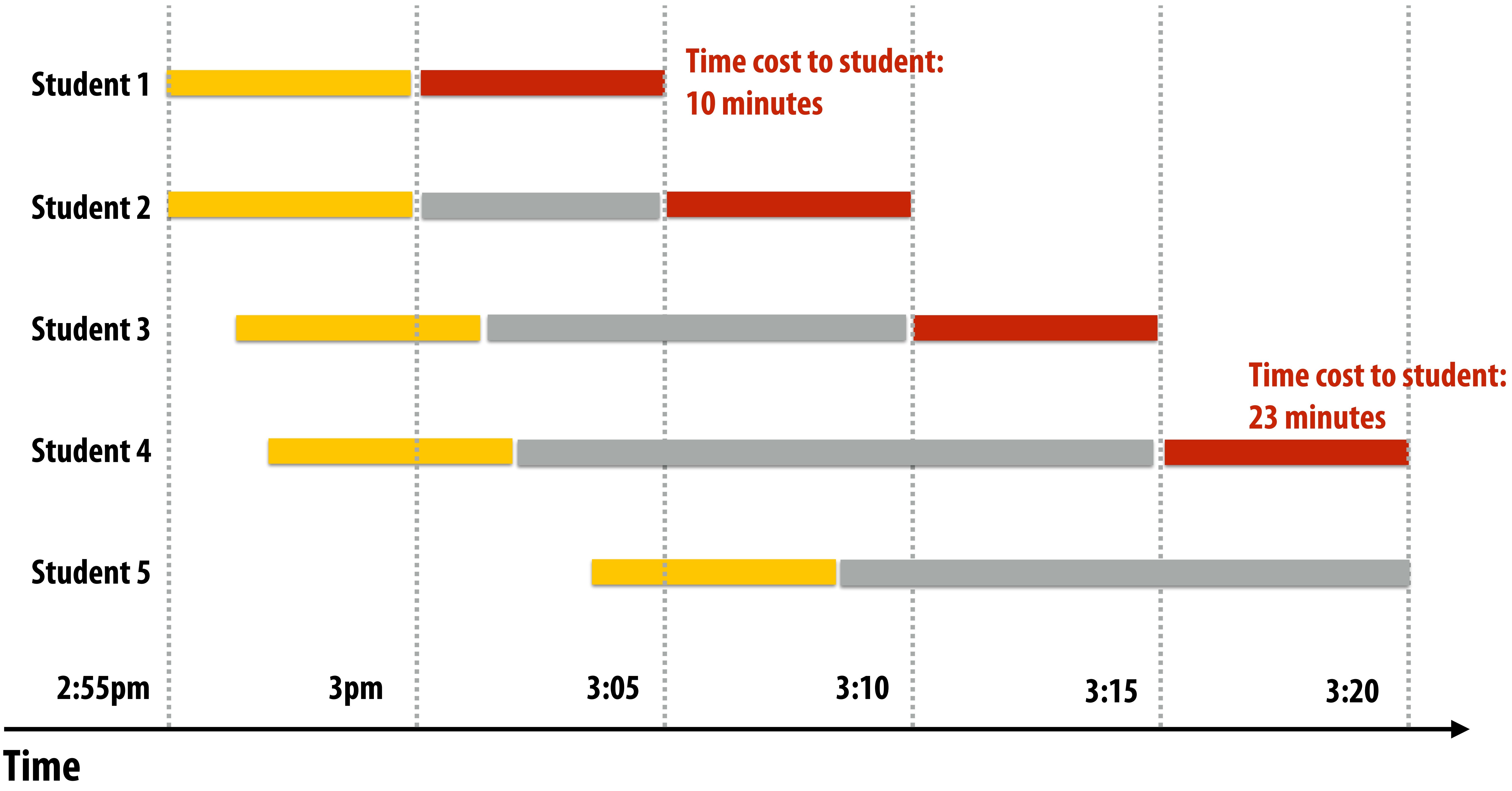
Student 1 (appt @ 3pm)

Time cost to student: 10 minutes

Student 2 (appt @ 4pm)

Time cost to student: 10 minutes

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



🟨 = Walk to Prof. Mowry's office (5 minutes)

⬜ = Wait in line

🟥 = Get question answered

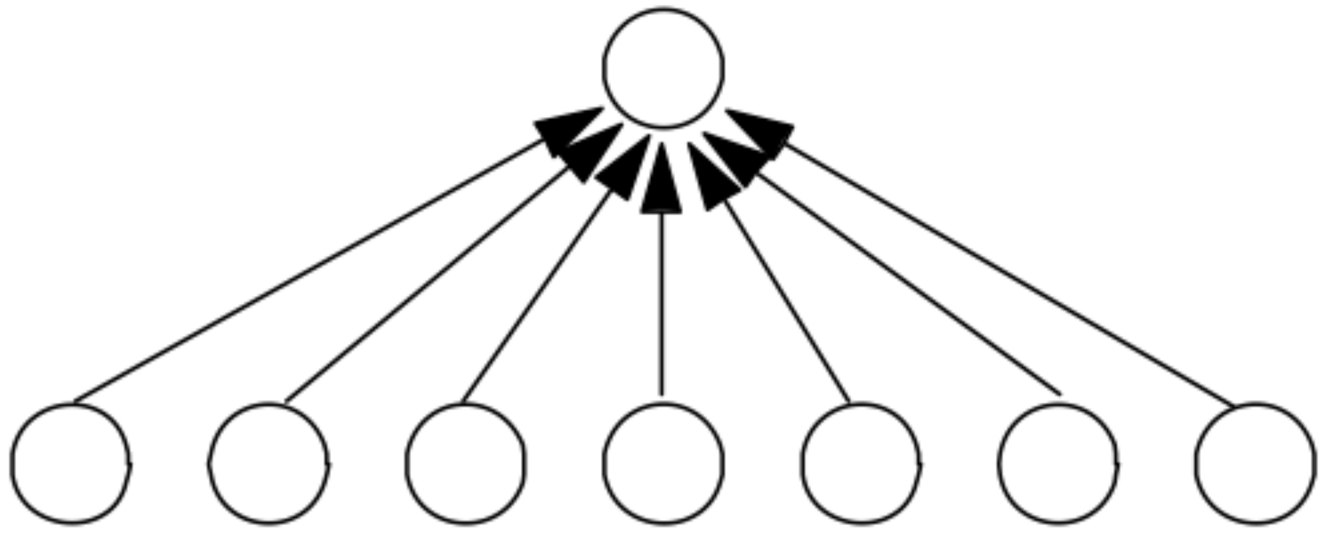
Problem: contention for shared resource results in longer overall operation times (and likely higher cost to students)

Contention

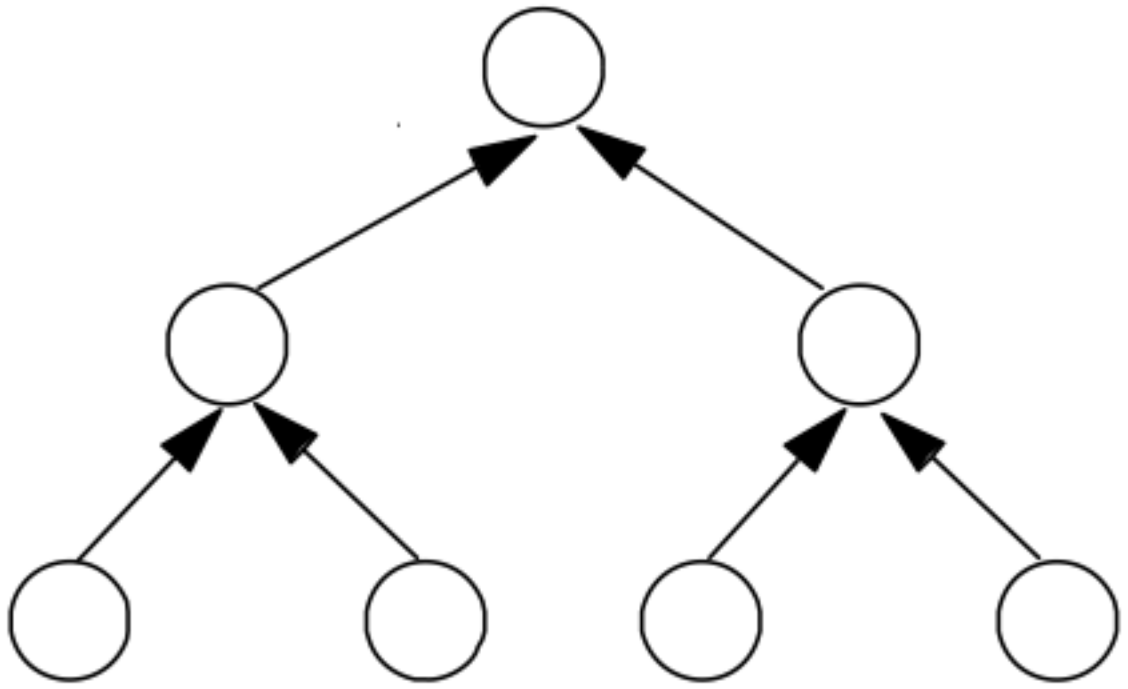
- A resource can perform operations at a given throughput (number of transactions per unit time)
  - Memory, communication links, servers, TA's at office hours, etc.

- 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)



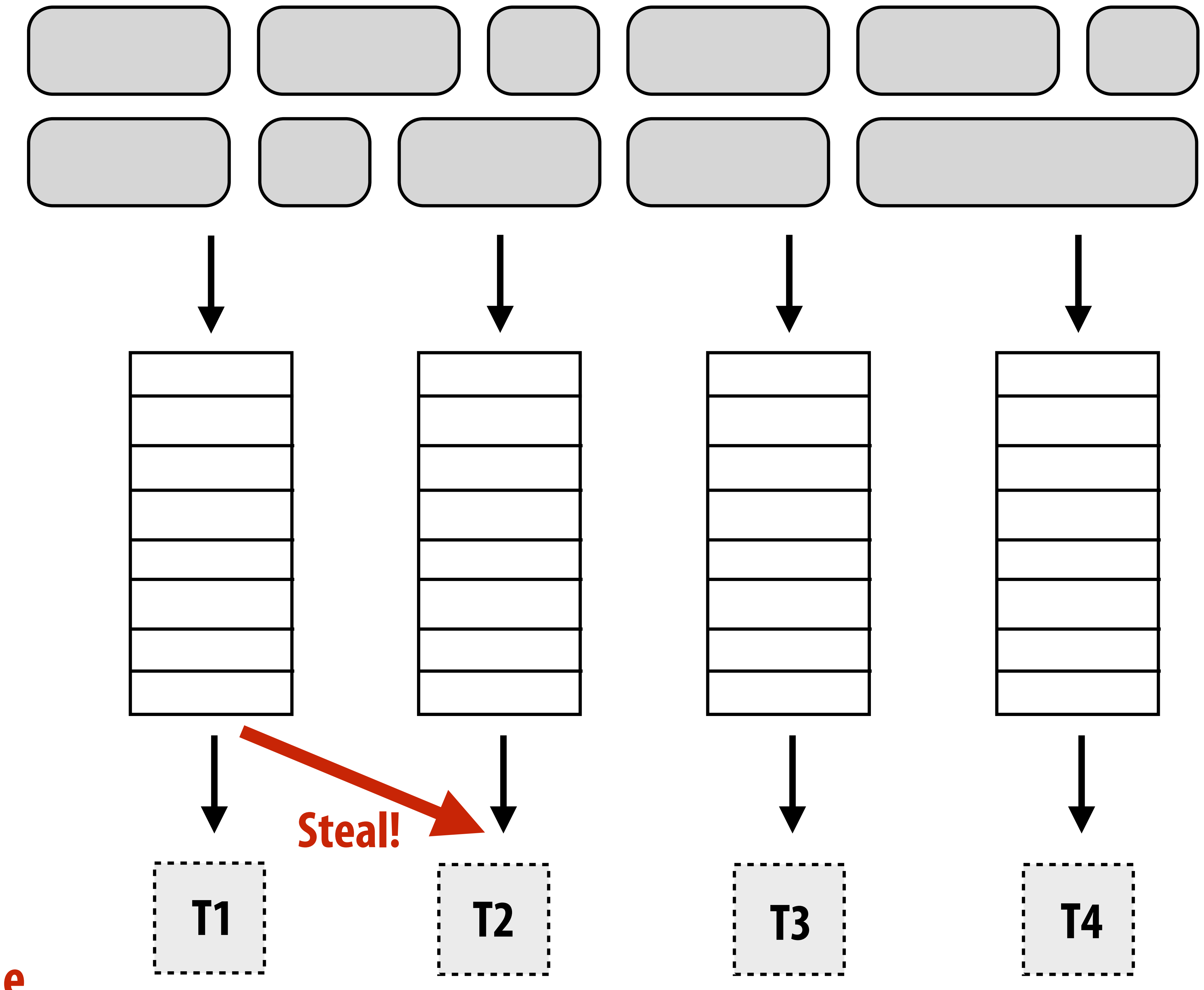
Tree structured communication:
reduces contention
(but higher latency under no contention)

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



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)