Lecture 7:

Programming for Performance: Part II

Locality, Communication, and Contention

Parallel Computer Architecture and Programming
CMU 15-418, Spring 2013

Today: more parallel program optimization

- Recall last lecture: distributing work to processors
  - Goal: achieving good workload balance while also minimizing overhead
  - Discussed static vs. dynamic assignment
  - Tip: keep it simple (implement, analyze, then tune/optimize if required)

- Today: minimizing communication and exploiting locality

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.

"Cost"

The effect operations have on program execution time

(or some other metric, e.g., power...)

"My slow program sends most of it's time waiting on memory." (cost of latency)

"saxpy achieves low ALU utilization because it is bandwidth bound." (cost of insufficient bandwidth)

But I'm going to start with the idea of: Pipelining

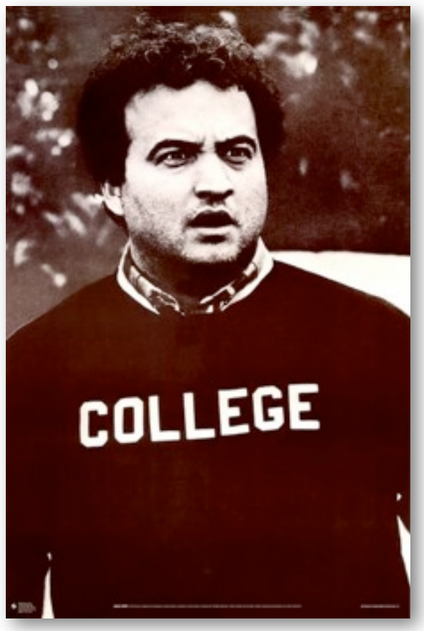
Doing your laundry

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

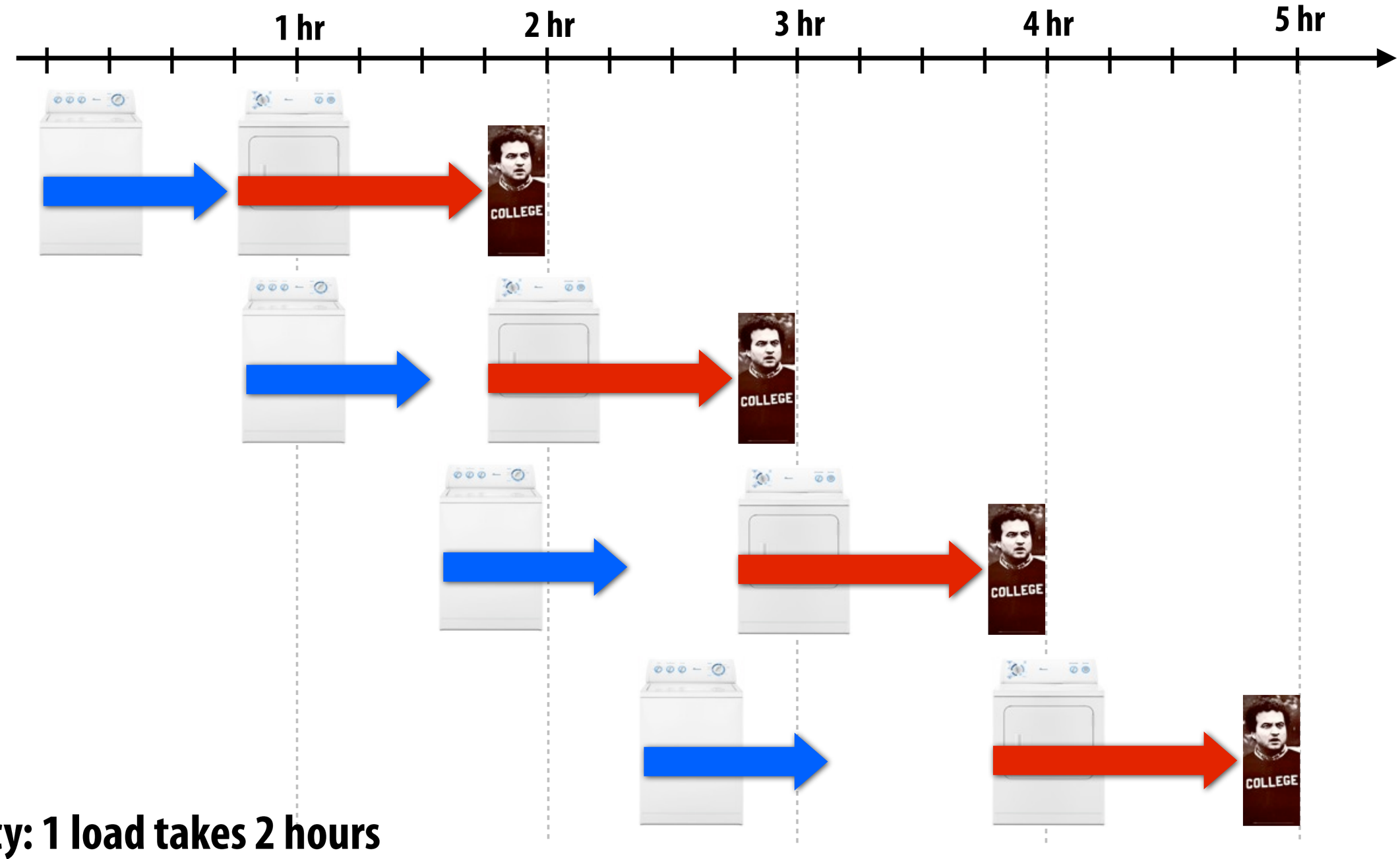
Goal: maximize throughput of many loads of laundry

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
Resources increased by 2x: two washers, two dryer

Pipelining

Goal: maximize throughput of many loads of laundry



Latency: 1 load takes 2 hours

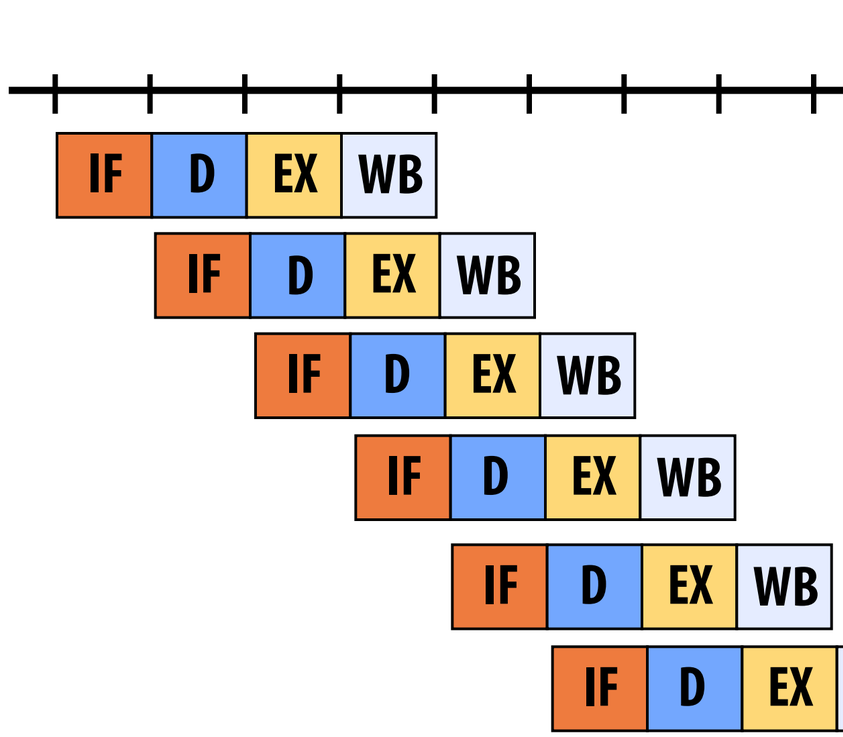
Throughput: 1 load/hour

Resources: one washer, one dryer

Another example: an instruction pipeline

Break instruction execution down into many steps

Key to scaling processor clock frequency (each clock, a simple short operation is done by each unit)



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
(Important: special care must be taken to ensure correctness in case of dependent instructions)

Modern 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)

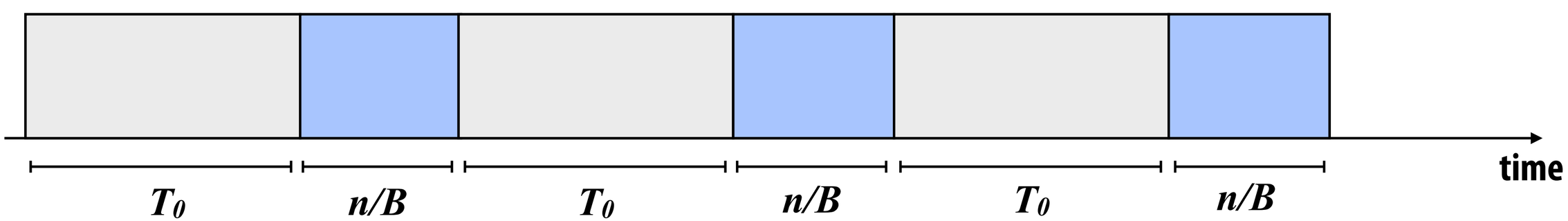
n = bytes transferred

B = transfer rate (bandwidth of the link)

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

Effective bandwidth = $n / T(n)$

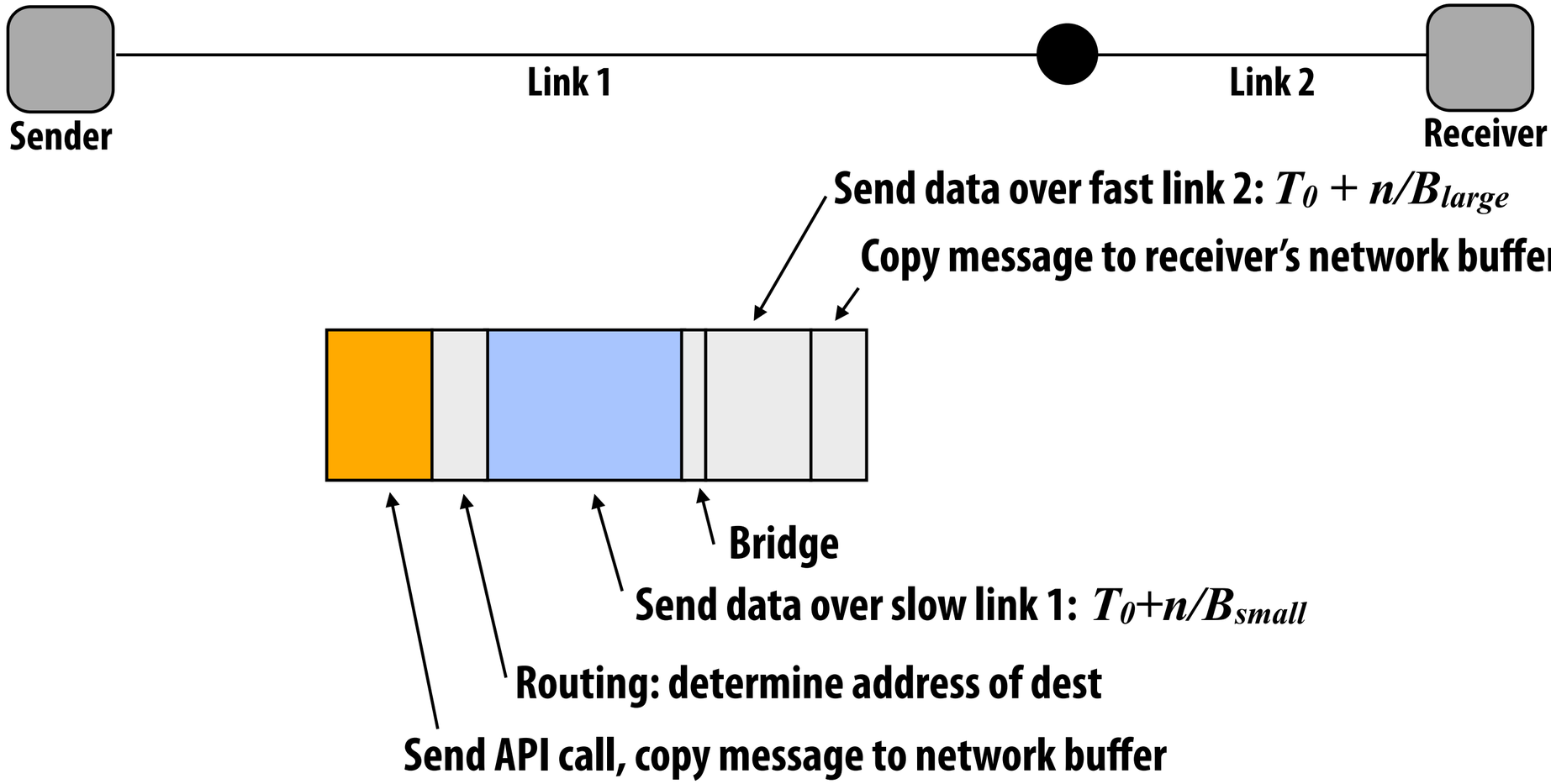
Effective bandwidth depends on transfer size



A more general model of communication

Communication time = overhead + occupancy + network delay

Example: sending a message

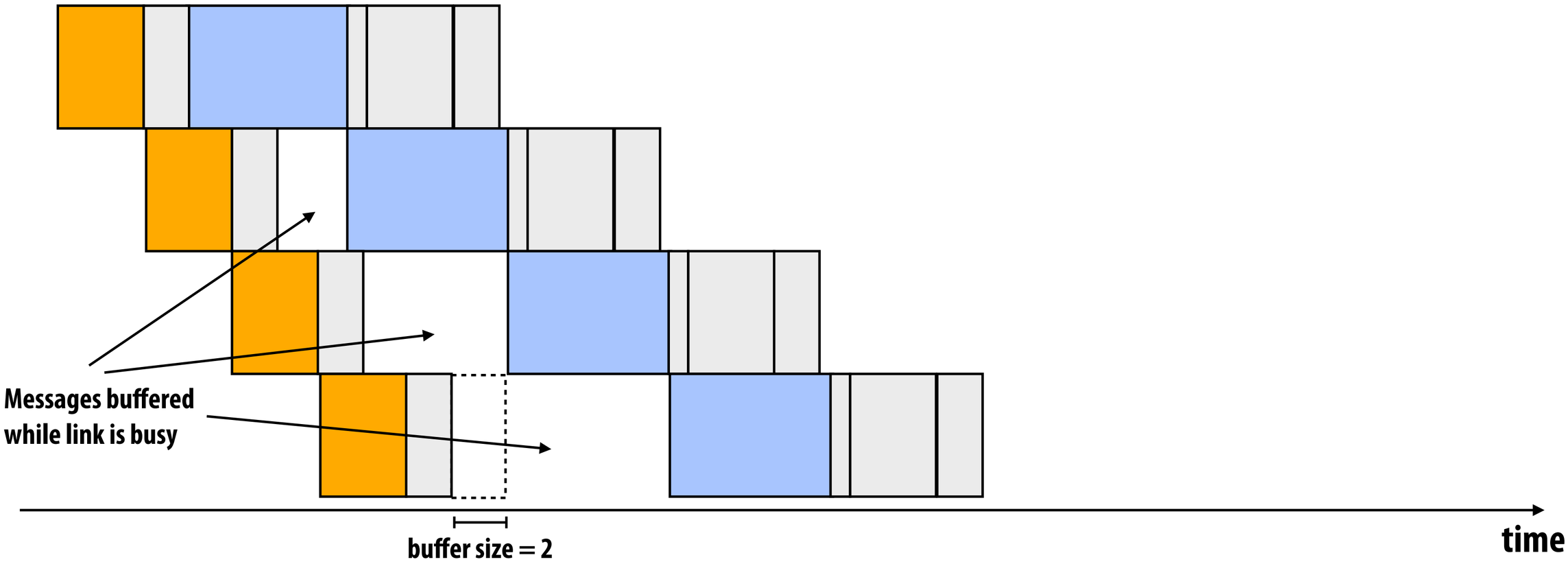


= Overhead (time spent on the communication by a processor)

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

= Network delay (everything else)

Pipelined communication



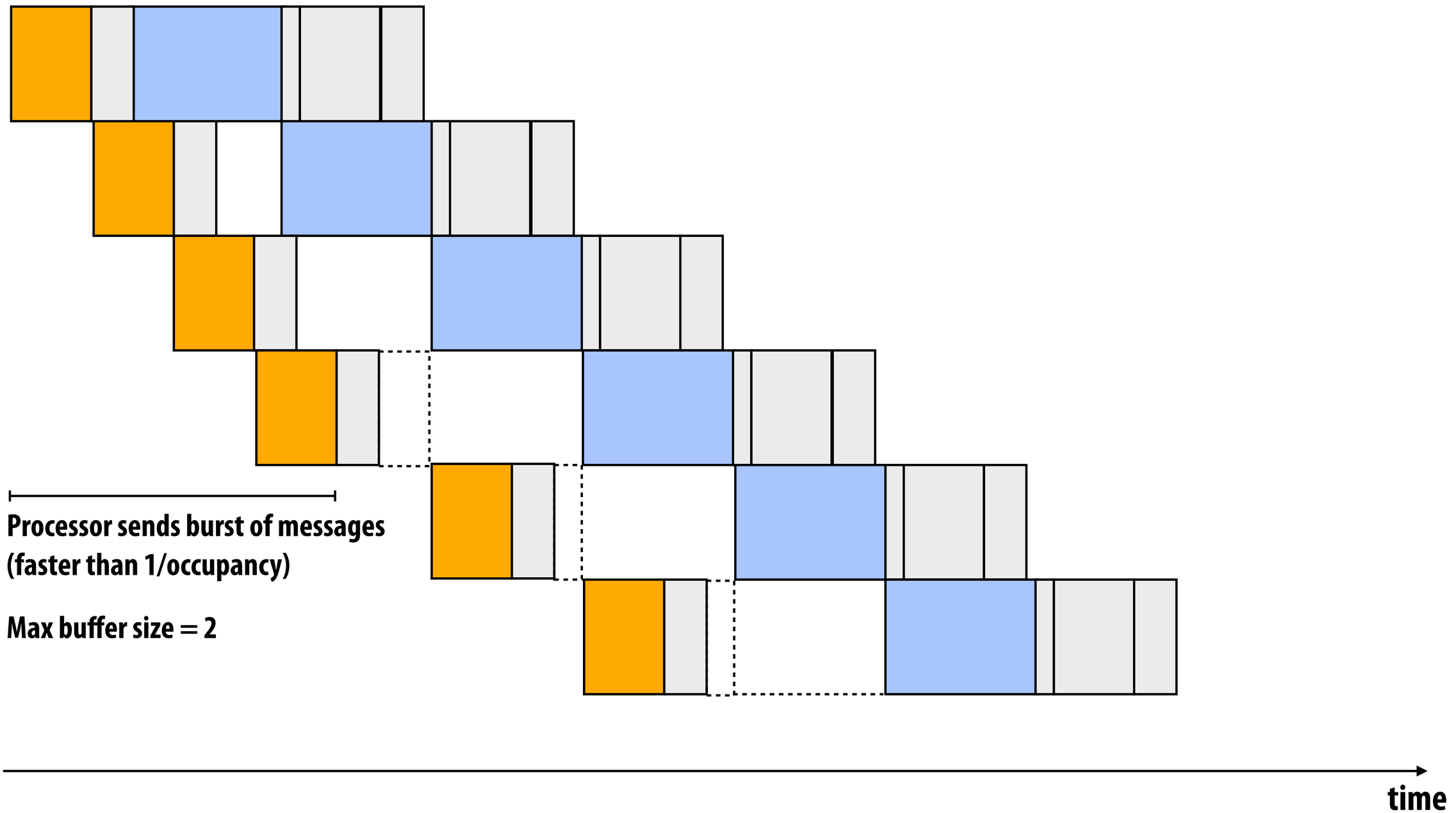
Occupancy determines communication rate (effective bandwidth)

= 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



Occupancy determines communication rate

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

Communication cost

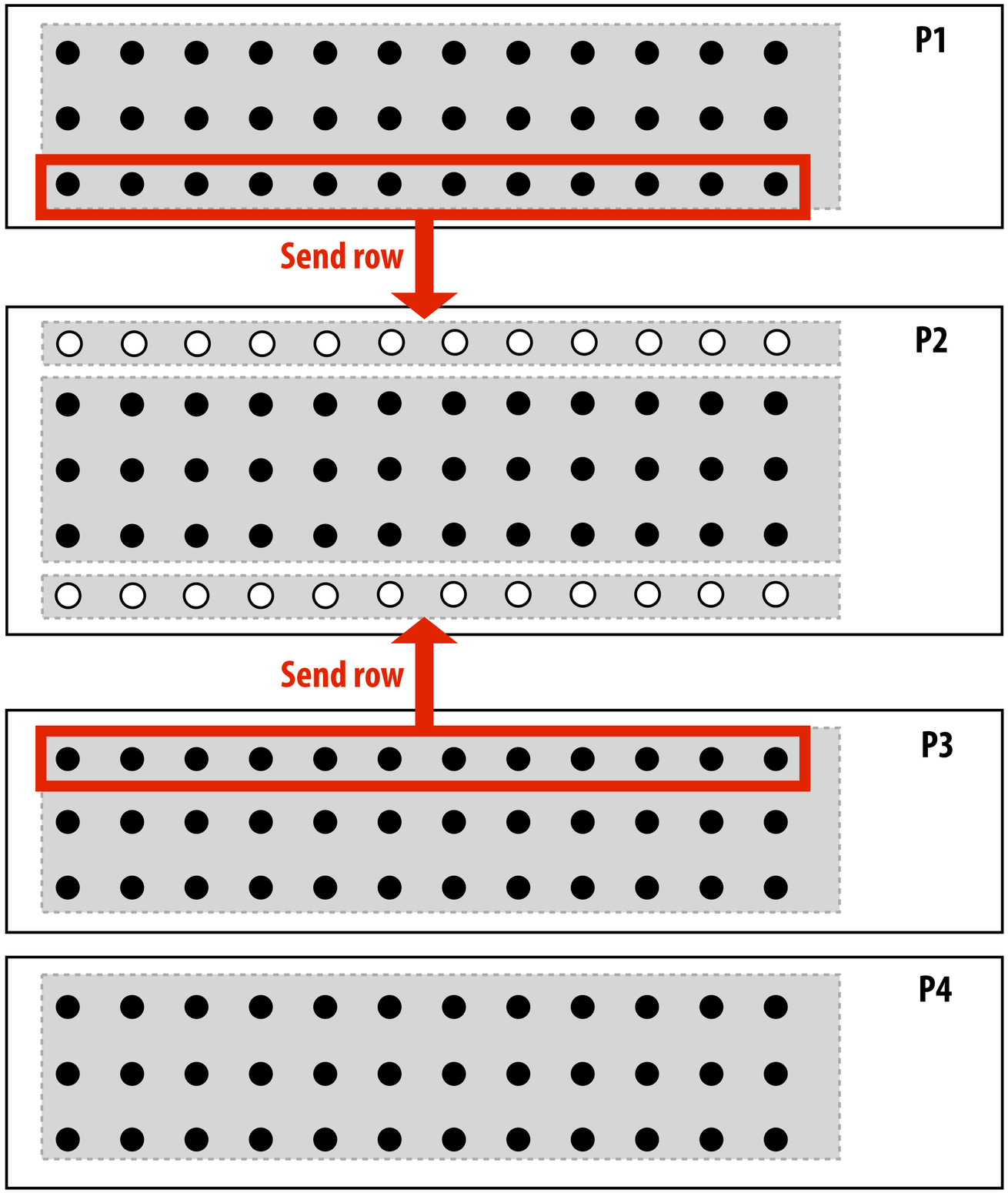
Total communication cost = frequency x (communication time - overlap)

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 it's cost relative to the cost of the computation fundamental to the problem being solved.

Inherent communication

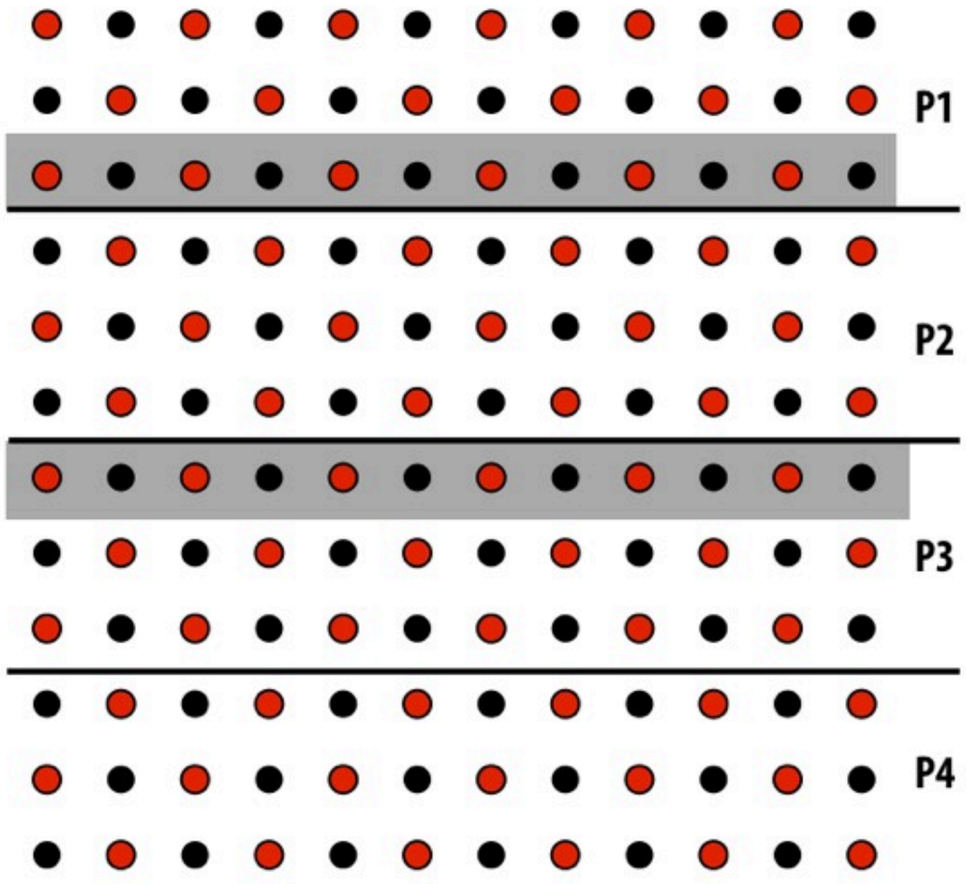


Communication that must occur in a parallel algorithm. Fundamental to the algorithm.

Reducing inherent communication

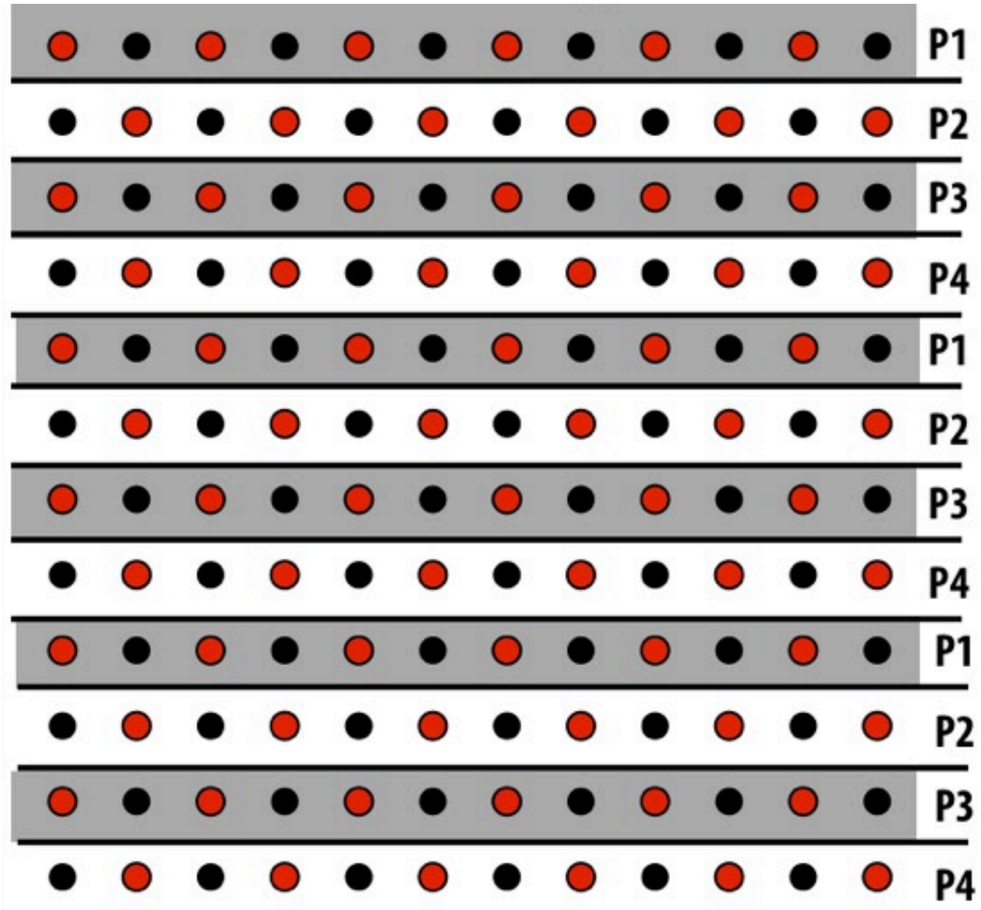
- Good assignment can reduce inherent communication (decrease communication to computation ratio)

1D blocked assignment



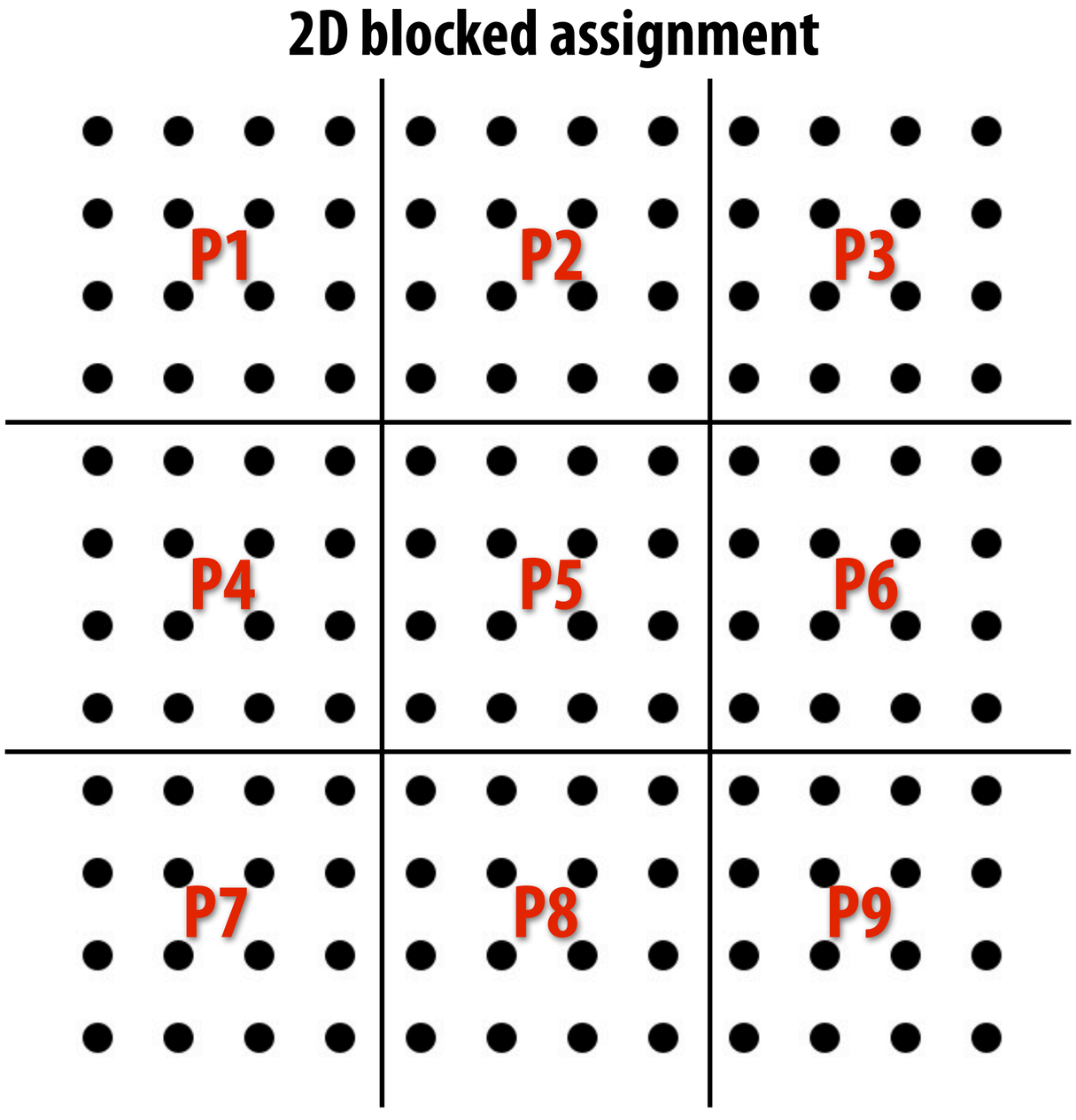
$$\frac{\text{elements communicated} \propto PN}{\text{elements computed} \propto N^2} \propto P / N$$

1D interleaved assignment



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

Reducing inherent communication



N^2 elements

P processors

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

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

comm-to-comp ratio: $\frac{\sqrt{P}}{N}$

Asymptotically better communication scaling than 1D blocked assignment

Communication costs increase sub-linearly with P

Assignment captures 2D locality of algorithm

Communication-to-computation ratio

$$\frac{\text{amount of communication}}{\text{amount of computation}}$$

- 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

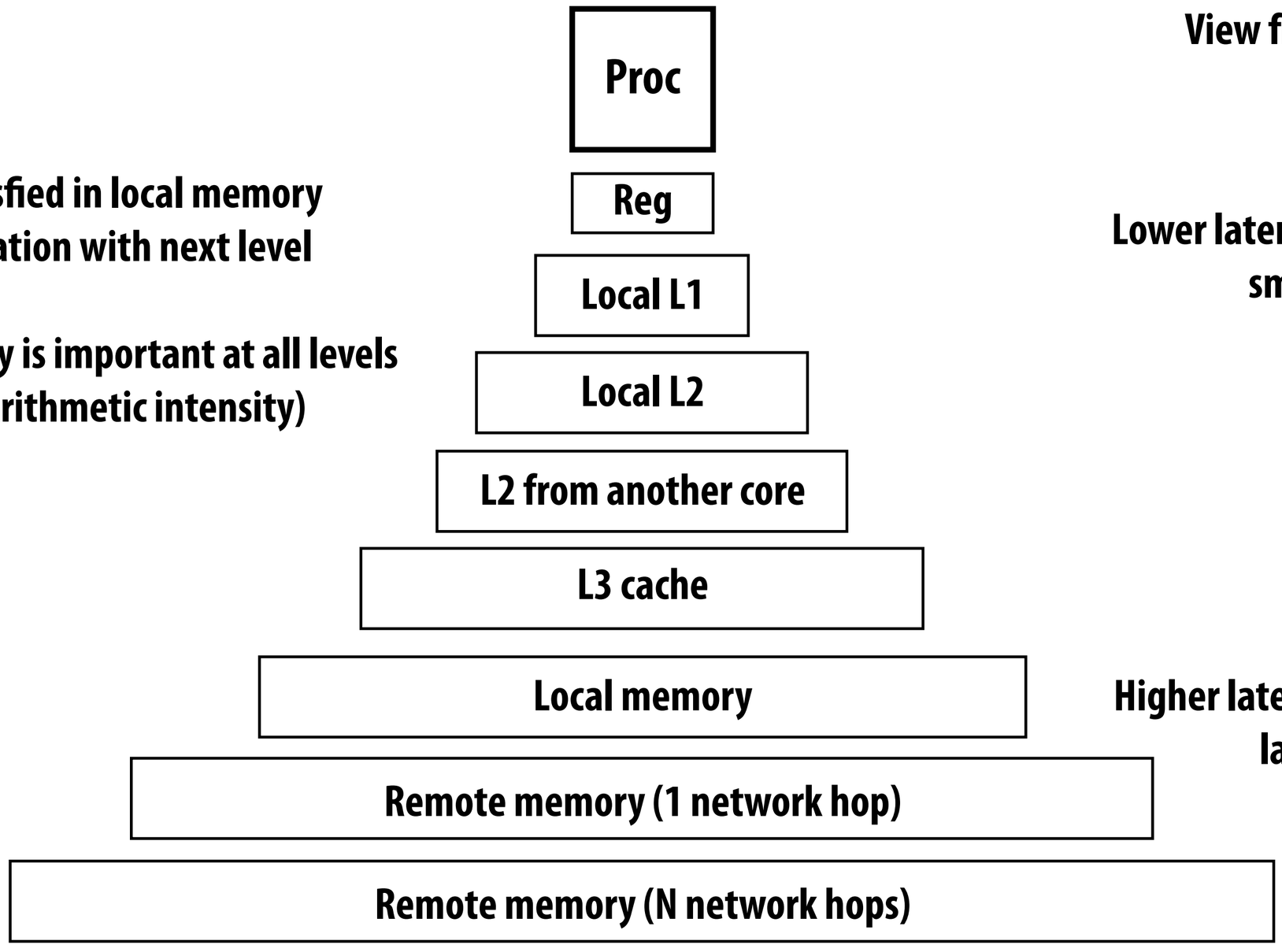
- High arithmetic intensity is needed on modern parallel processors since the ratio of compute capability to available bandwidth is very high (recall saxpy)

Parallel system as an extended memory hierarchy

- Up until now: I've described data "partitioned among processors", and that "non-local data" required communication
- In reality, parallel system is multi-memory, multi-cache system. Characteristics of data access latencies and bandwidths can have large effect on performance

Accesses not satisfied in local memory cause communication with next level

Managing locality is important at all levels (achieving high arithmetic intensity)



View from one processor

Lower latency, higher bandwidth, smaller capacity

Higher latency, lower bandwidth, larger capacity

Artifactual communication

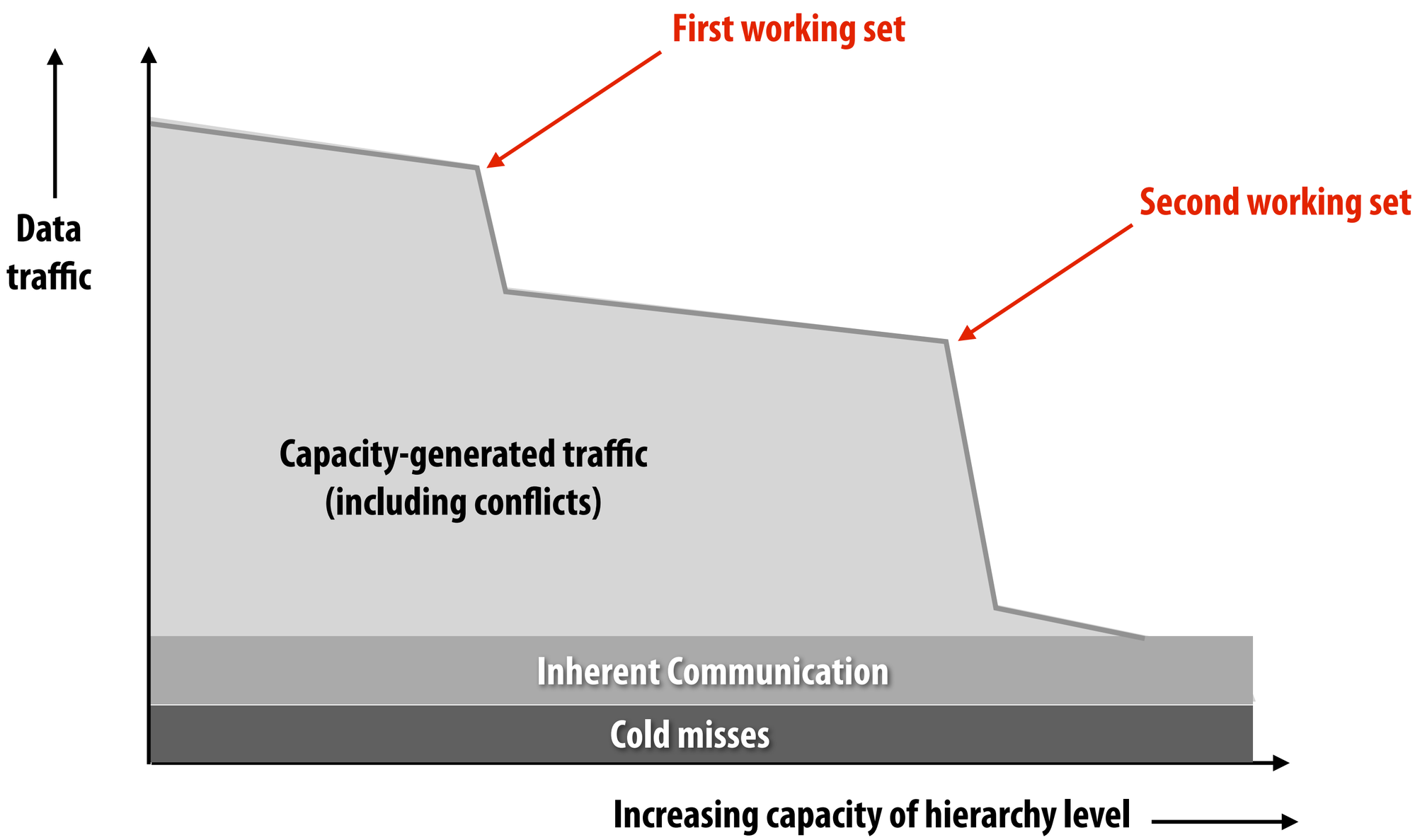
- Inherent communication: assumes unlimited capacity, small transfers, perfect knowledge of what is needed to communicate

- Artifactual communication is everything else (depends on interaction of application and system)
  - System might have a minimum granularity of transfer (result: system must send more then what's needed: e.g., program reads one word but entire cache line must be loaded)
  - Poor allocation of data among distributed memories (data doesn't reside near processor that accesses it most)
  - Finite replication capacity (same data communicated to processor multiple times because tier of memory hierarchy is too small to have kept it)

Review of the three (now four) Cs

- **Cold miss**
  - First time data touched. Unavoidable.
- **Capacity miss**
  - Working set larger than cache. Can be decreased by larger caches
- **Conflict miss**
  - Miss induced by cache management policy. Can reduce 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

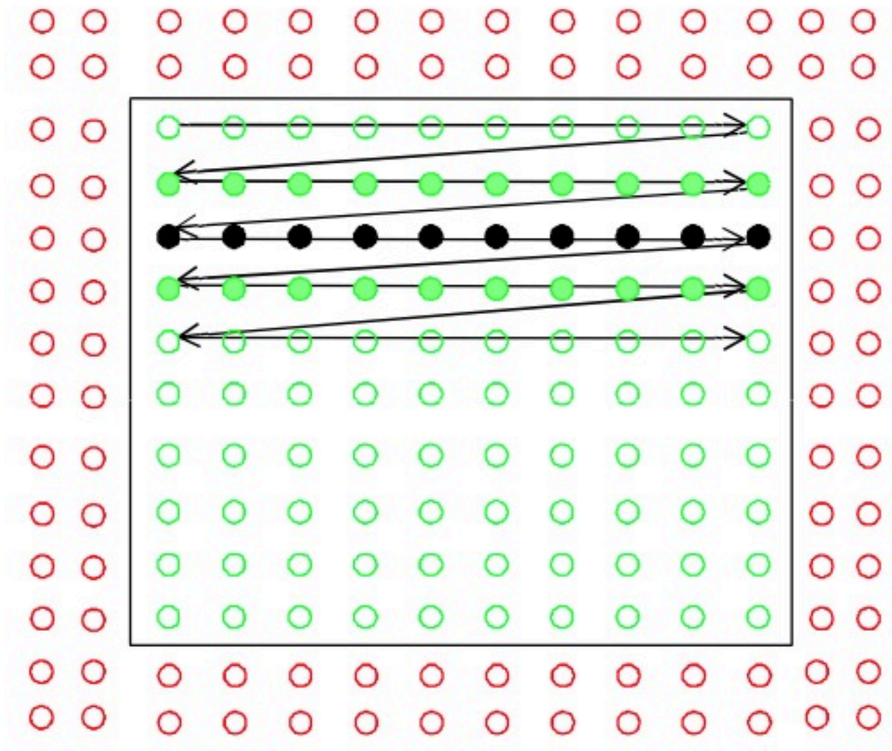
Question: how much capacity should an architect build for this workload?

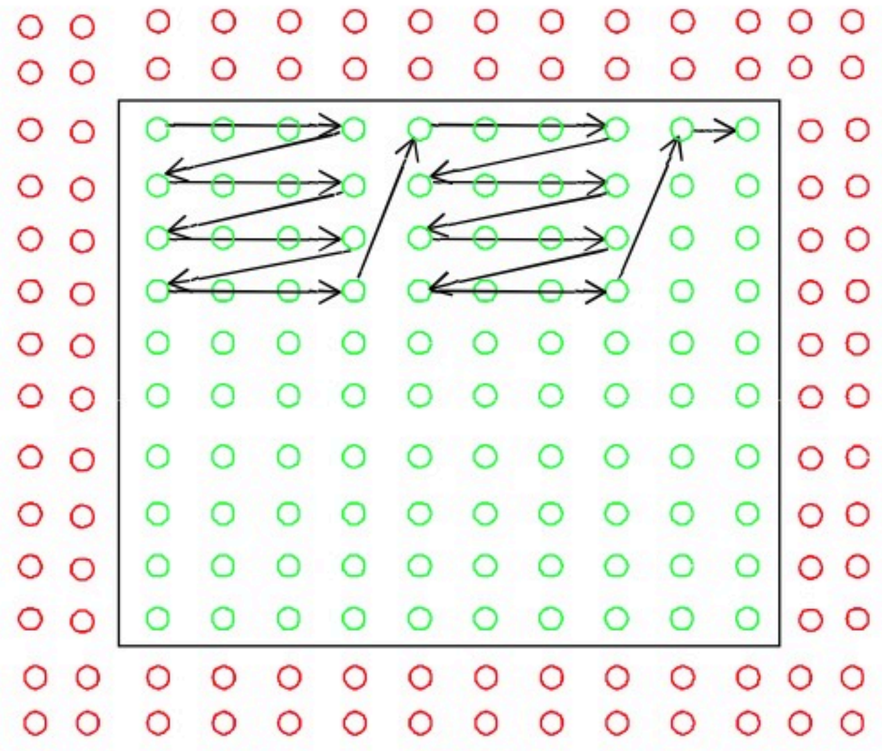
Reducing amount of communication

Improve temporal locality

- "Blocking": reorder computation to make working sets map well to system's memory hierarchy

Consider order in which elements are updated in solver example from previous class(es).





Common technique in sequential programs (recall matrix transpose assignment in 15-213)

Main idea: replicate block of data from in local memories (cache)

Process block of data in its entirety (accessing it many times) prior to moving only next block (goal: reduce capacity misses)

Improve temporal locality

- 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 to localize related processing in the machine
  - Threads in block often cooperate to perform an operation
  - Leverage fast access to / synchronization via CUDA shared memory

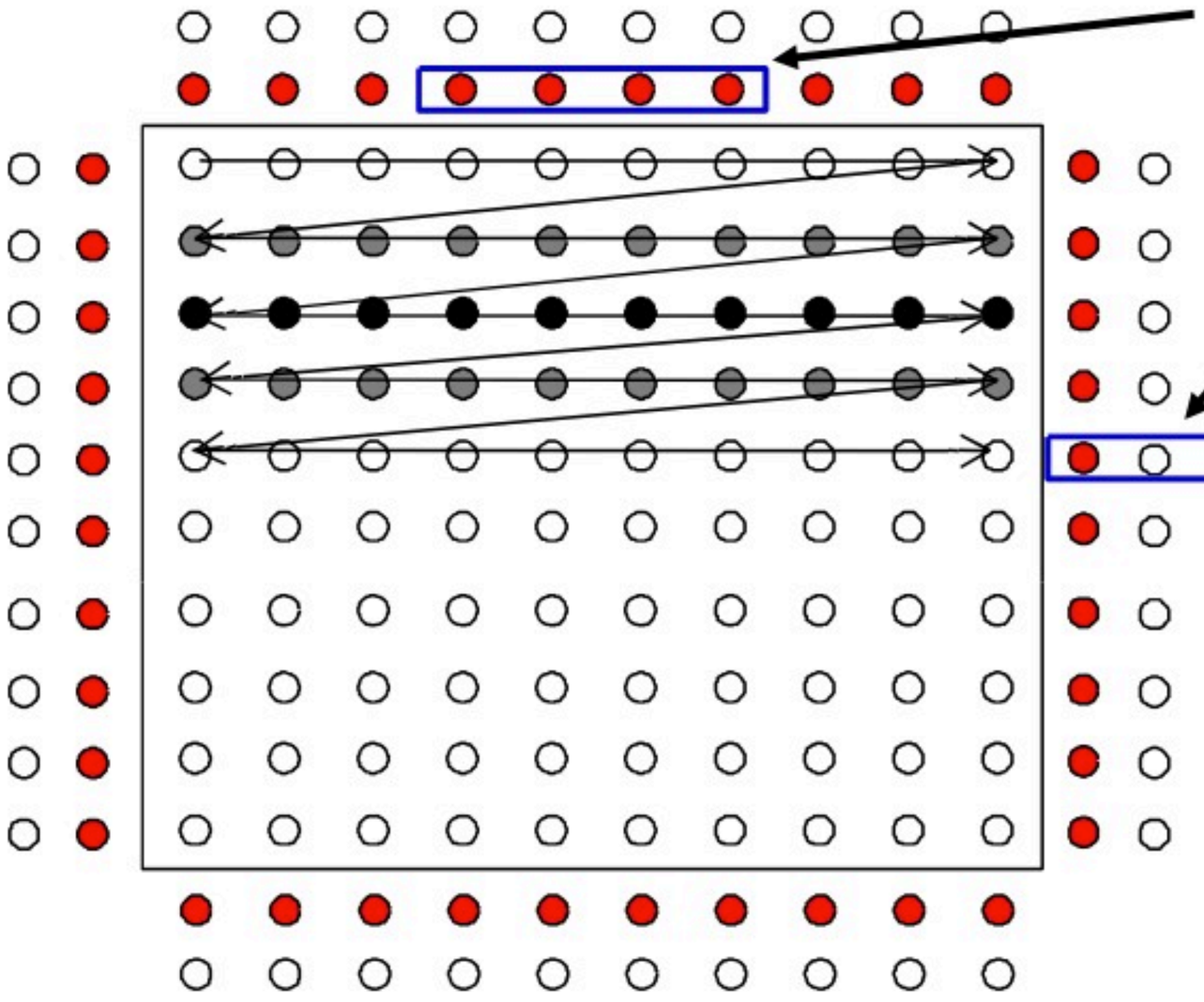
Exploiting spatial locality

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

Artifactual communication due to comm. granularity

Shared memory system, cache line communication granularity (assume line contains four elements)

2D blocked partitioning of data



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

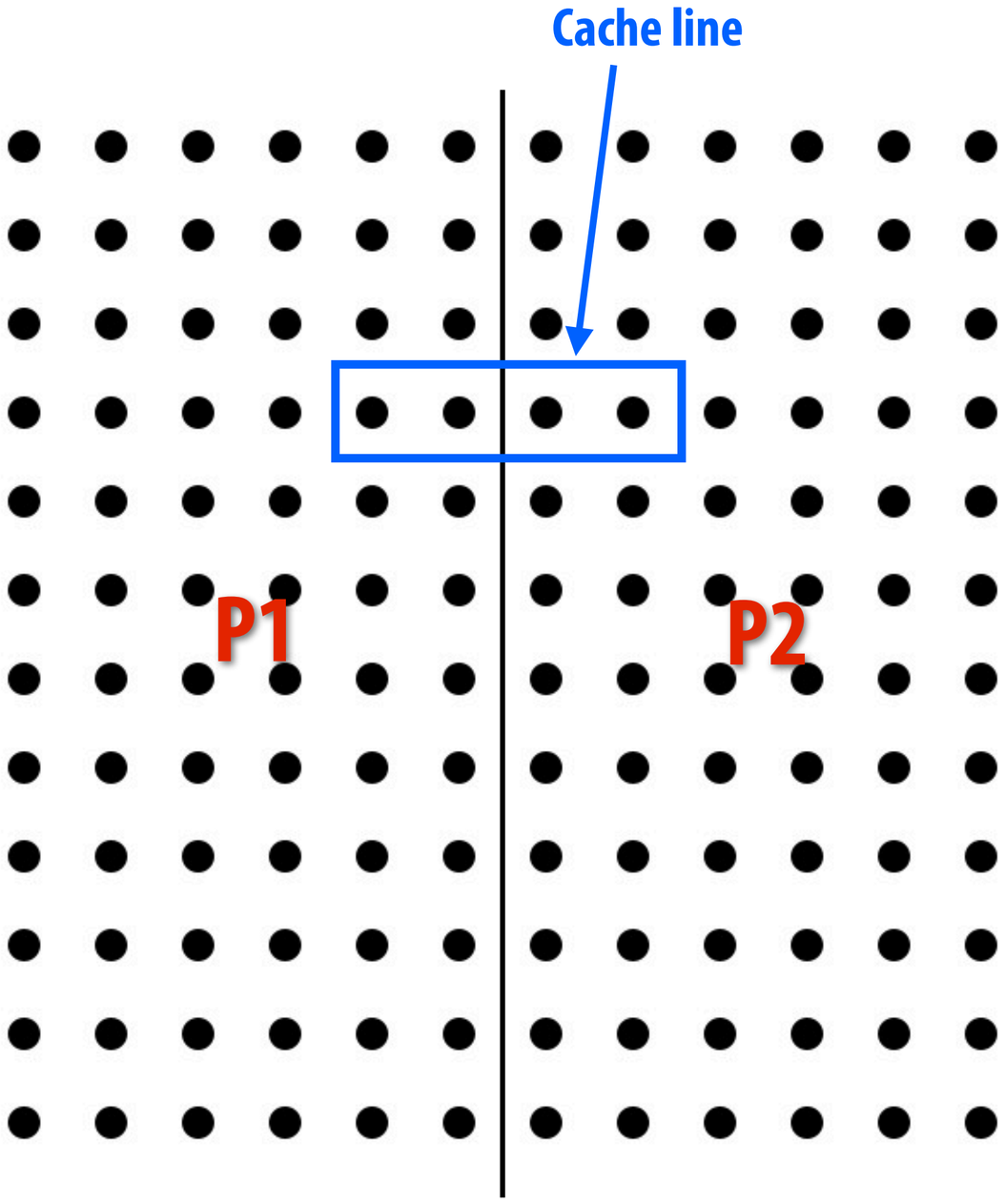
Poor spatial locality for non-local accesses to left-right columns

Need one element from line. Must communicate four.

Implication: 1D blocked layout may perform better despite higher inherent communication-to-computation ratio.

● = required elements assigned to other processors

Artifactual communication
(due to communication/coherence granularity)



Data partitioned among memories local to processors

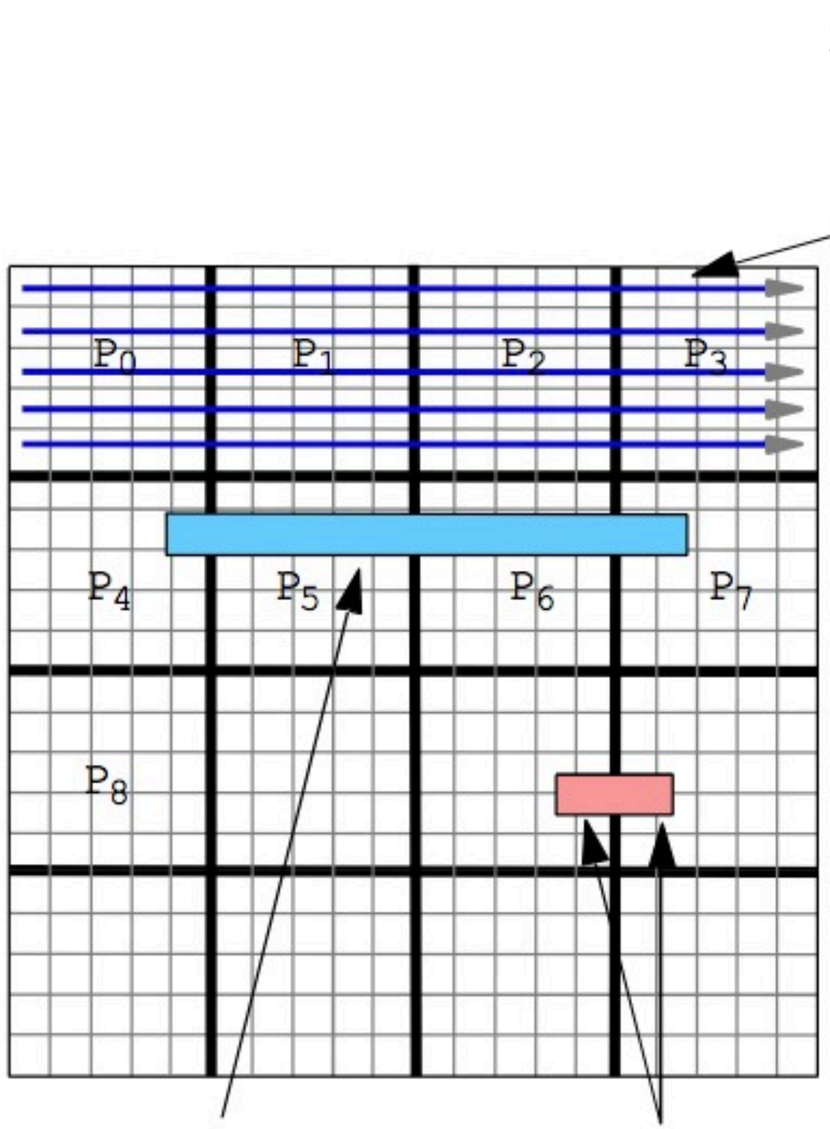
Processors access their assigned elements (no inherent communication)

But data access on real machine triggers communication (artifactual)

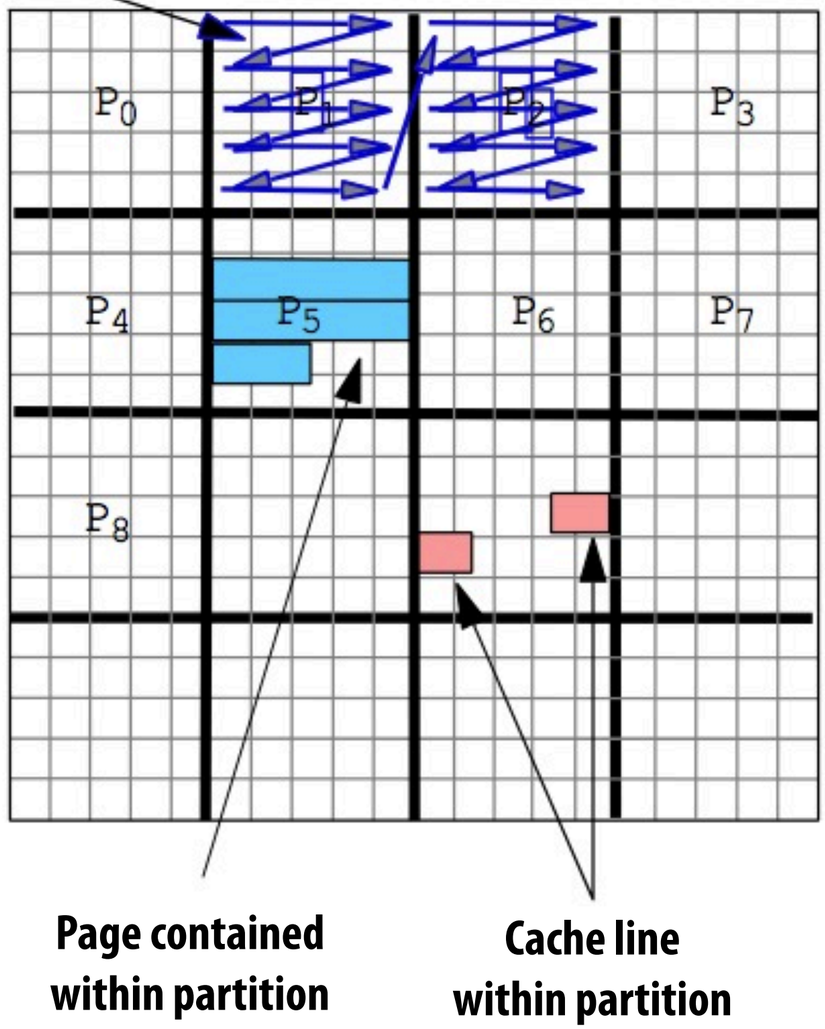
Also, writes by different processors require cache coherence **

** Implementing cache coherence will be a topic of future lectures.

Reducing artifactual comm: better layout



2D, row-major array layout



4D array layout (block-major)

Structuring communication to reduce cost

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

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

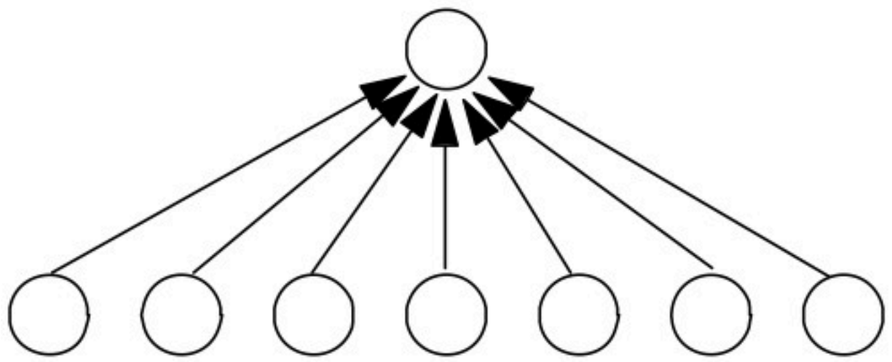
Total communication cost = frequency x (overhead + (n/B + contention) + network delay - overlap)

Occupancy dominated by time it takes to transfer message (n bytes) over slow link + delays due to contention for link

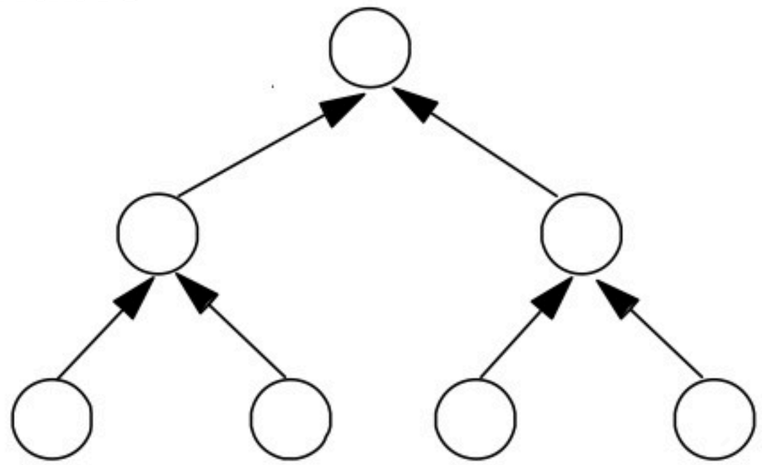
Contention

- All resources have non-zero occupancy
  - Memory, communication links, comm. controller. etc.
  - Each has fixed number of transactions per unit time
- Contention occurs when many requests to a resource are made within a small window of time
  - Resource is a "hot spot"

Example: updating a shared variable



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

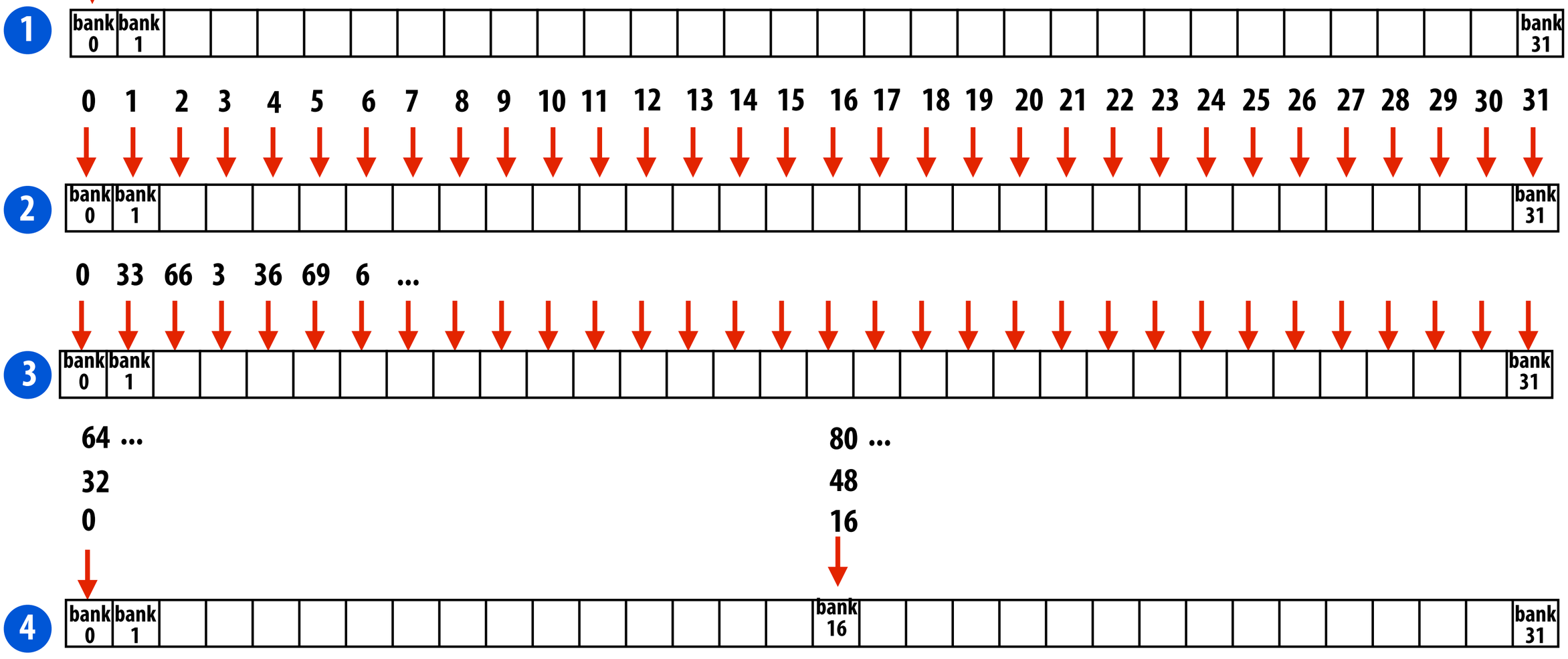


Tree structured communication:
reduces contention
(but higher latency under no contention)

NVIDIA GTX 480 contention example

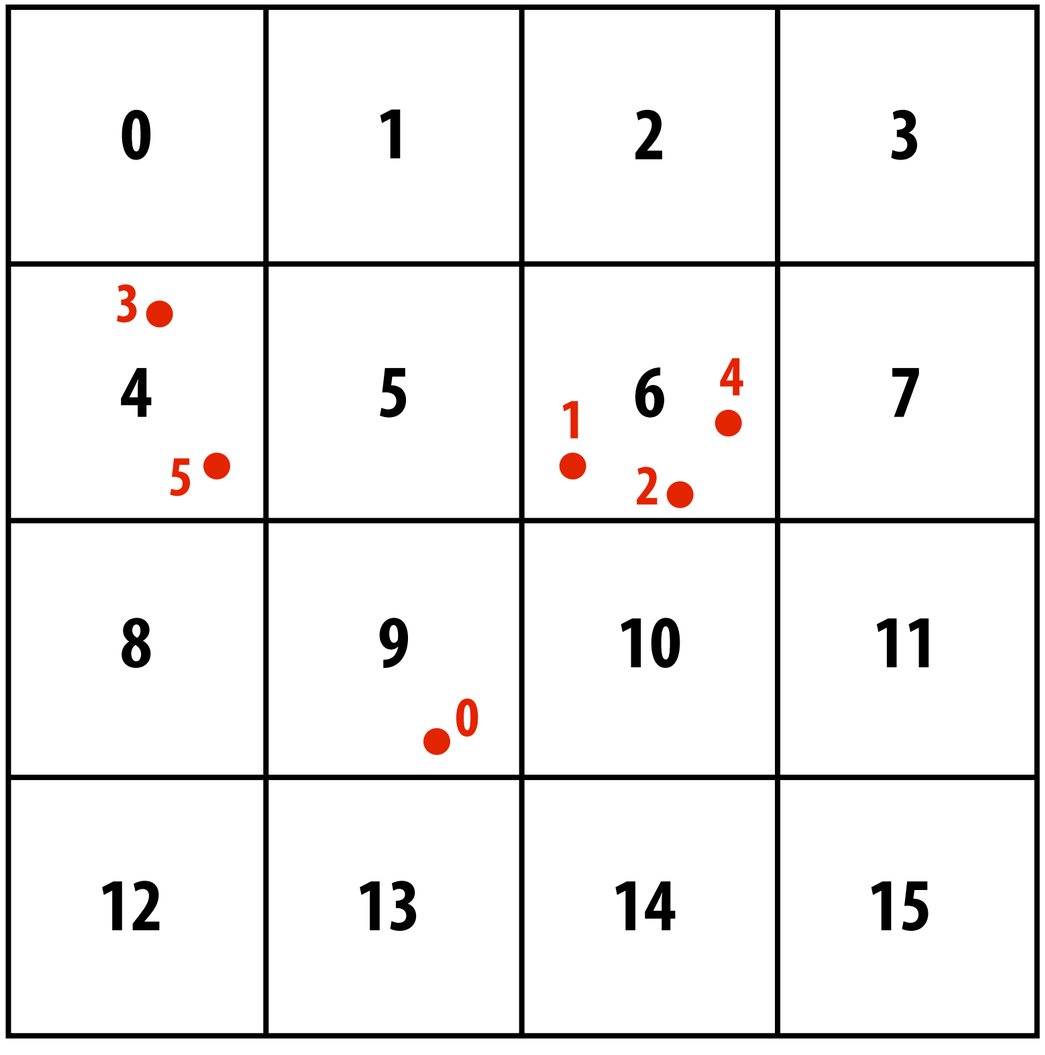
- Shared memory implementation
  - 48 KB on-chip indexable storage, divided into 32 banks
  - Address X is stored in bank b, where b = X % 32
  - Each bank can be accessed in parallel (one word per bank per clock)
  - Special broadcast mode if all addresses are the same (example 1)
- Figure shows addresses 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
```



Another contention example

- Problem: place 100K point particles in a 16-cell uniform grid
  - Parallel data structure manipulation problem: build a 2D array of lists
- Recall: 15 cores, up to 1024 threads per core on GTX 480 GPU



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	

Solution 1: parallelize over cells

- One answer: partition work by cells: for each cell, independently compute contained particles
  - 16 parallel tasks (insufficient parallelism: need thousands of independent tasks for GPU)
  - Also: performs 16 times more particle-in-cell computations than sequential algorithm

    (15 parallel units, but 16 times more work: it's no faster!)

```
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. Compute cell containing particle. Atomically update list.
  - Massive contention: thousands of threads contending for access to update 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)
```

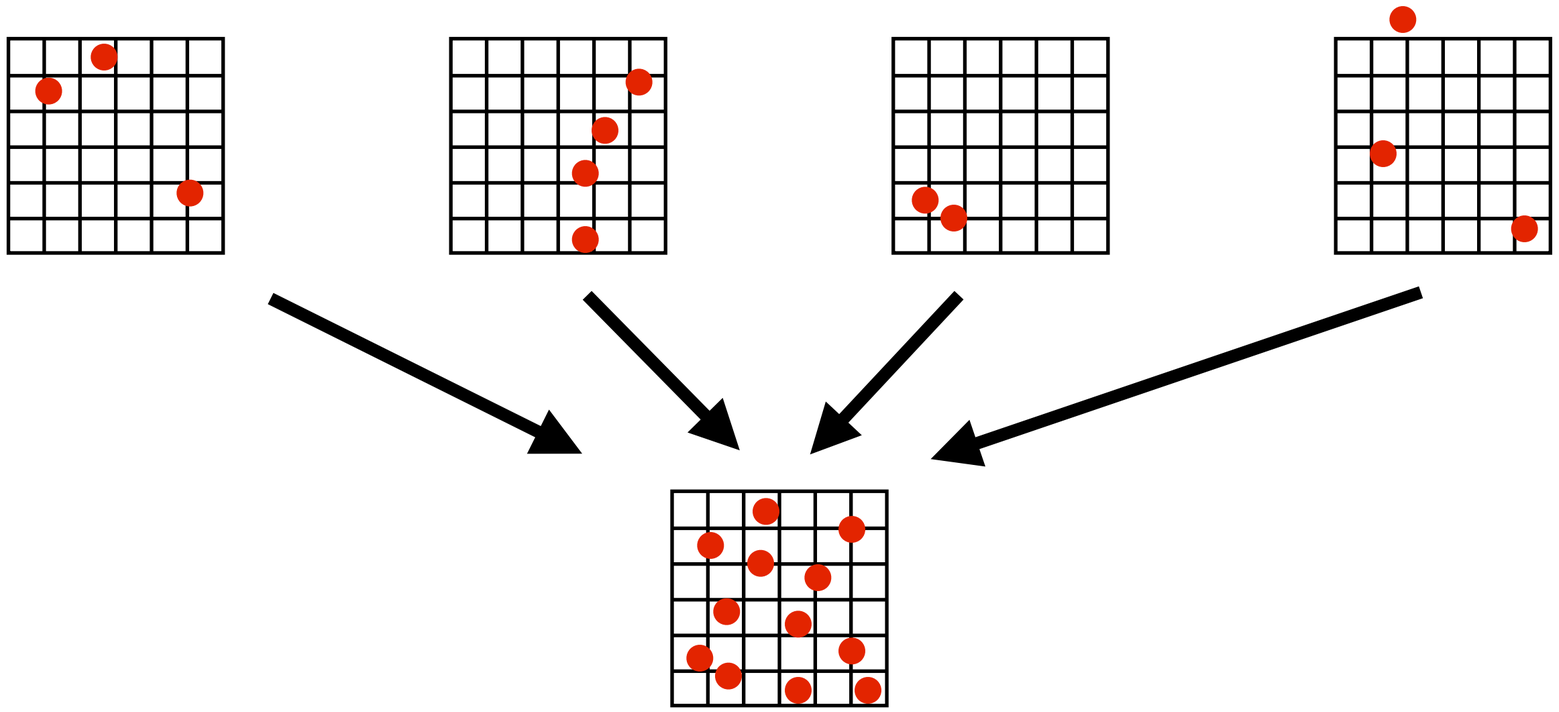
Solution 3: use finer-granularity locks

- Alleviate contention by using per-cell locks
  - Assuming uniform distribution of particles... ~16x less contention

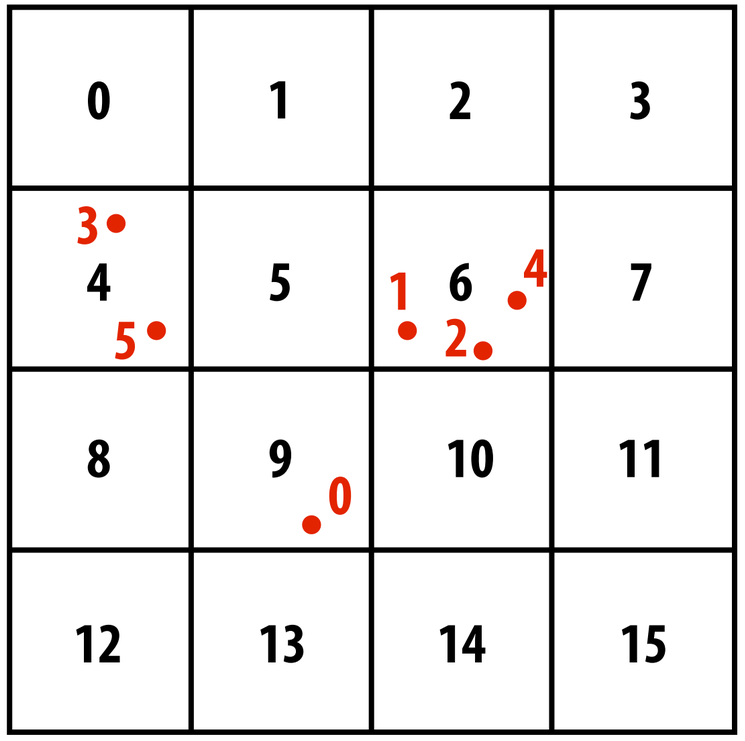
```
list cell_list[16];    // 2D array of lists
lock cell_list_lock[16];

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 15 grids on GTX 480 (one per core)
  - All threads in thread block update same grid
    - Faster synchronization: contention reduced by factor of N and synchronization performed on local variables
  - Extra work: merging the grids at the end of the computation
  - Extra memory footprint: Store 15 grids, rather than 1



Solution 5: data-parallel approach



Step 1: compute cell containing each particle

Array Index:	0	1	2	3	4	5
result:	9	6	6	4	6	4

Step 2: sort results by cell

Array Index:	3	5	1	2	4	0
result:	4	4	6	6	6	9

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

Step 3: find start/end of each cell

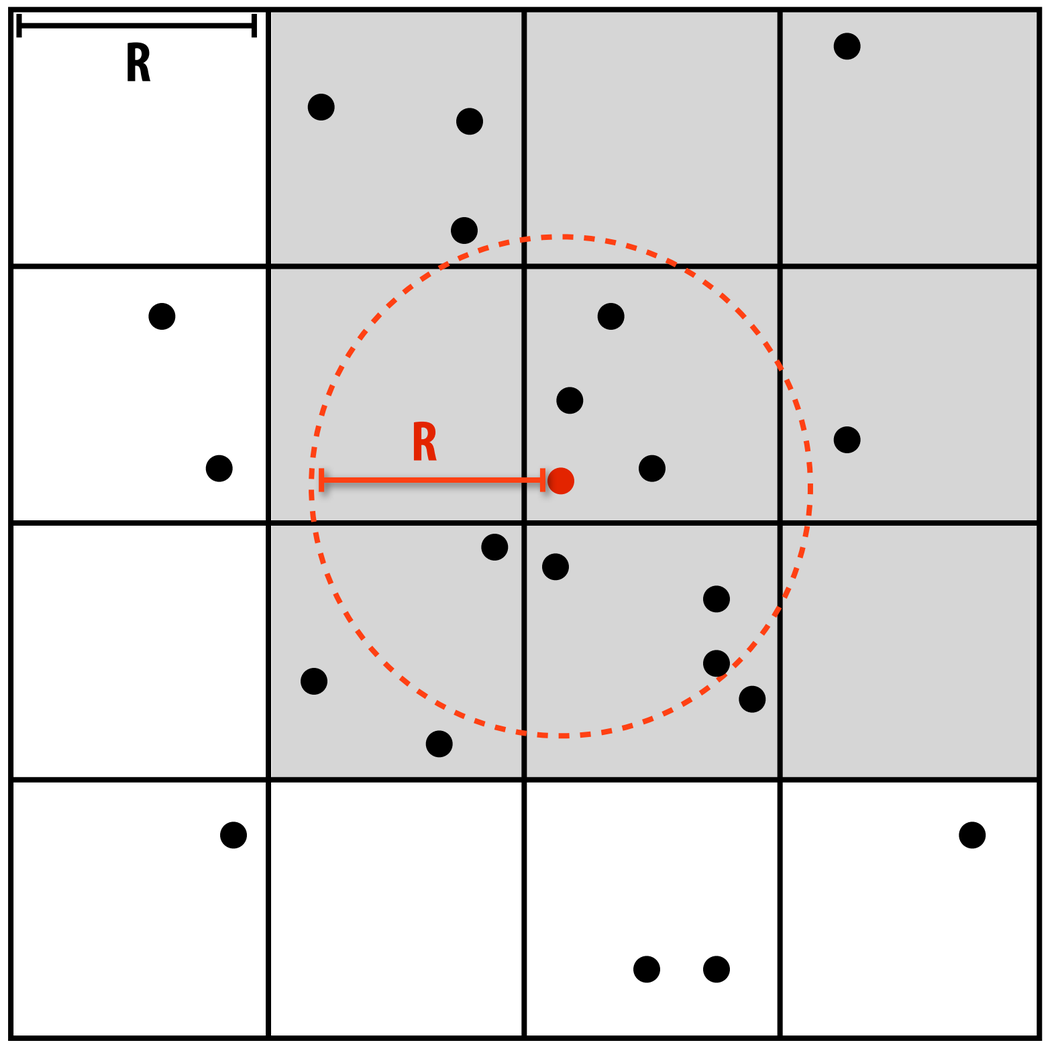
```
cell = result[index]
if (index == 0 || cell != result[index-1]) {
    cell_starts[cell] = index;
    if (index > 0)            // special case for first cell
       cell_ends[result[index-1]] = index;
}
if (index == numParticles-1) // special case for last cell
    cell_ends[cell] = index+1;
```

This code is run for each element of 'result'

	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	• • •

Common use: N-body problems

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



Reducing communication costs

- Reduce overhead to sender/receiver
  - Send fewer messages, make messages larger
  - Coalesce small messages into large ones

- Reduce delay
  - HW implementor: improve communication architecture
  - Application writer: exploit locality

- Reduce contention

- Increase overlap
  - HW implementation: multi-threading, pre-fetching, out-of-order execution
  - Application: asynchronous communication
  - Requires additional concurrency in application (more concurrency than number of processors)

Summary: optimizing communication

- Inherent vs. artifactual communication
  - 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 (few, large messages)
  - Reduce contention
  - Maximize overlap (hide latency so as to not incur cost)