(how to be l33t)

Lecture 6:

~~Performance Optimization~~ Part II: Locality, Communication, and Contention

Parallel Computing

Stanford CS149, Fall 2021

Today: more parallel program optimization

- A case study in optimizing a parallel program (grid solver)

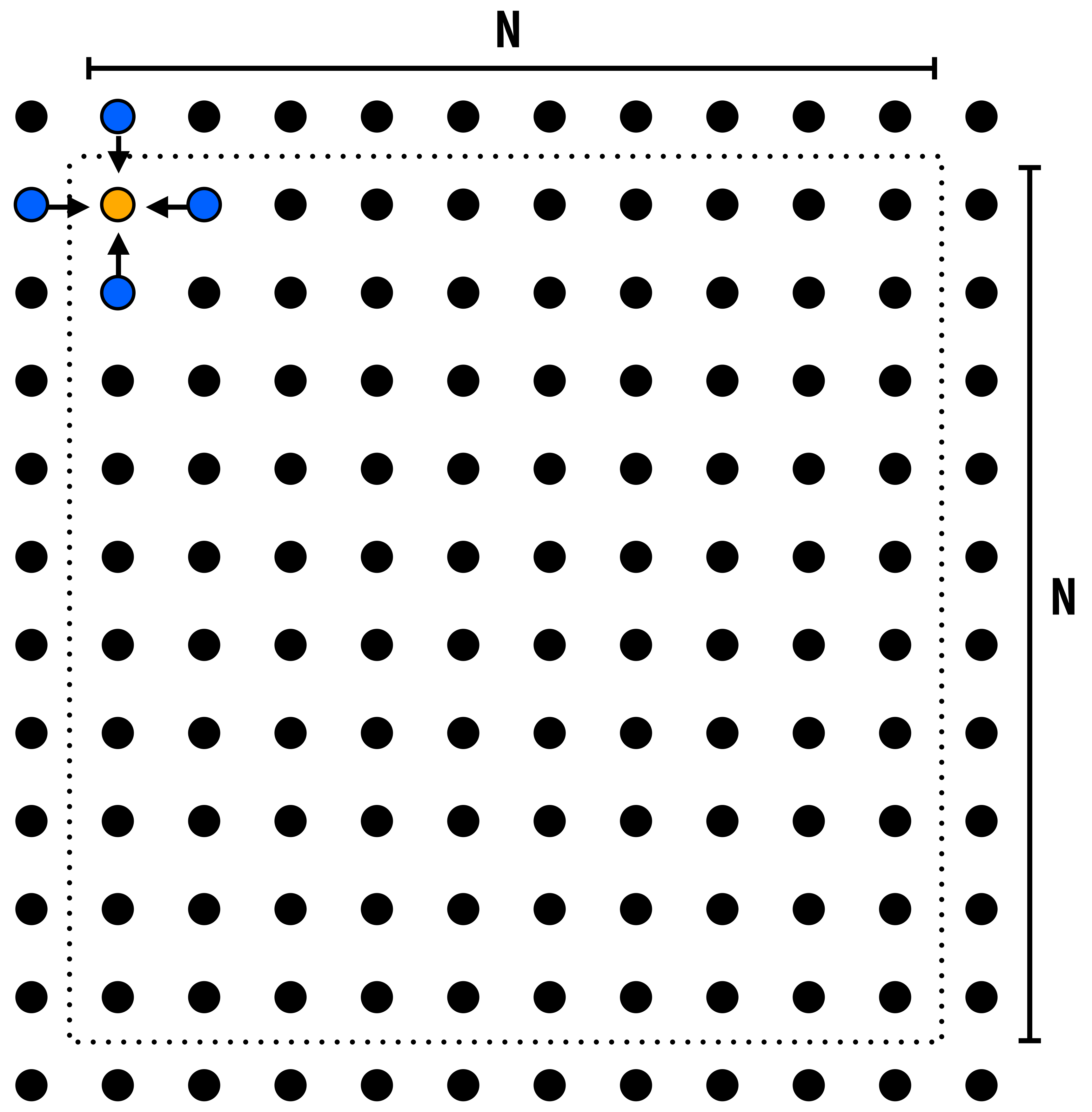
- Last lecture: strategies for assigning work to workers (threads, processors, etc.)
  - Goal: achieving good workload balance while also minimizing overhead
  - Discussed tradeoffs between static and dynamic work assignment
  - Reminder: keep it simple (implement, analyze, then tune/optimize if required)

- Today: strategies for minimizing communication costs

A parallel programming example

A 2D-grid based solver

- Problem: solve partial differential equation (PDE) on (N+2) x (N+2) grid
- Solution uses iterative algorithm:
  - Perform Gauss-Seidel sweeps over grid until convergence



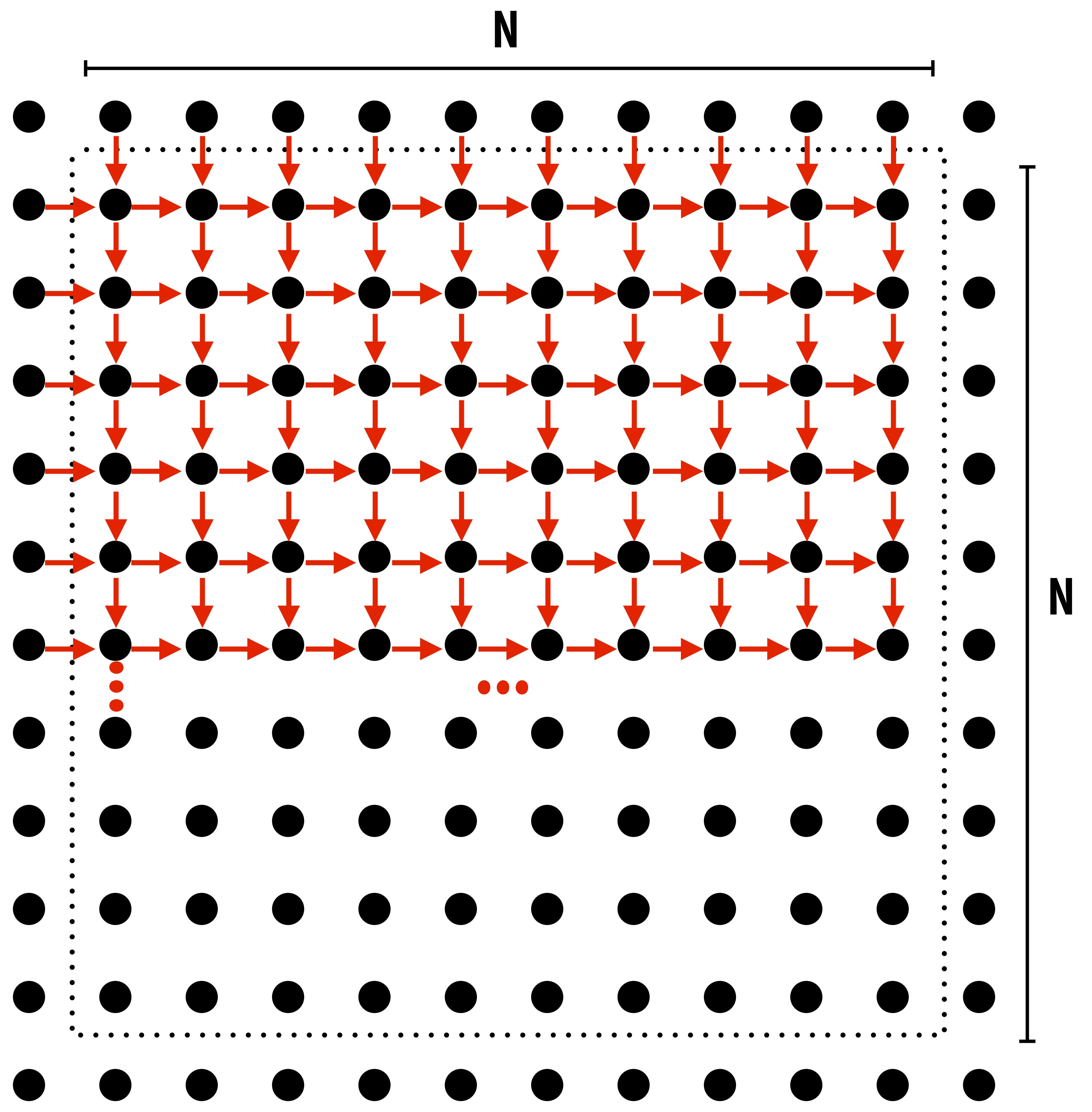
```
A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j]
                                   + A[i,j+1] + A[i+1,j]);
```

Grid solver algorithm: find the dependencies

C-like pseudocode for sequential algorithm is provided below

```
const int n;
float* A;                      // assume allocated for grid of N+2 x N+2 elements

void solve(float* A) {

  float diff, prev;
  bool done = false;

  while (!done) {                           // outermost loop: iterations
    diff = 0.f;
    for (int i=1; i<n i++) {                // iterate over non-border points of grid
      for (int j=1; j<n; j++) {
        prev = A[i,j];
        A[i,j] = 0.2f * (A[i,j] + A[i,j-1] + A[i-1,j] +
                                  A[i,j+1] + A[i+1,j]);
        diff += fabs(A[i,j] - prev);        // compute amount of change
      }
    }

    if (diff/(n*n) < TOLERANCE)             // quit if converged
      done = true;
  }
}
```

Step 1: identify dependencies (problem decomposition phase)

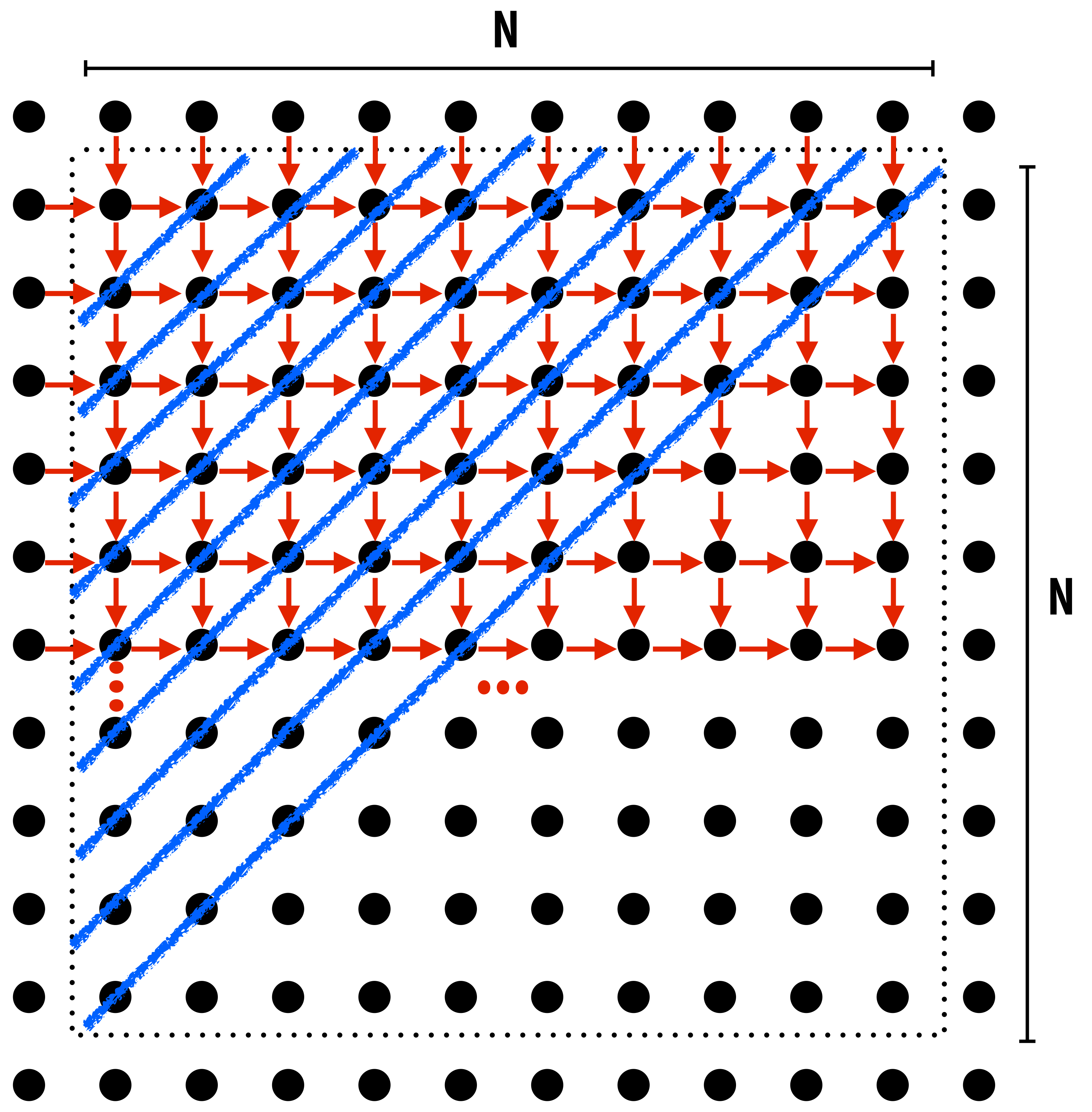


Each row element depends on element to left.

Each row depends on previous row.

Note: the dependencies illustrated on this slide are grid element data dependencies in one iteration of the solver (in one iteration of the "while not done" loop)

Step 1: identify dependencies (problem decomposition phase)



There is independent work along the diagonals!

Good: parallelism exists!

Possible implementation strategy:

1. Partition grid cells on a diagonal into tasks
2. Update values in parallel
3. When complete, move to next diagonal

Bad: independent work is hard to exploit

Not much parallelism at beginning and end of computation.

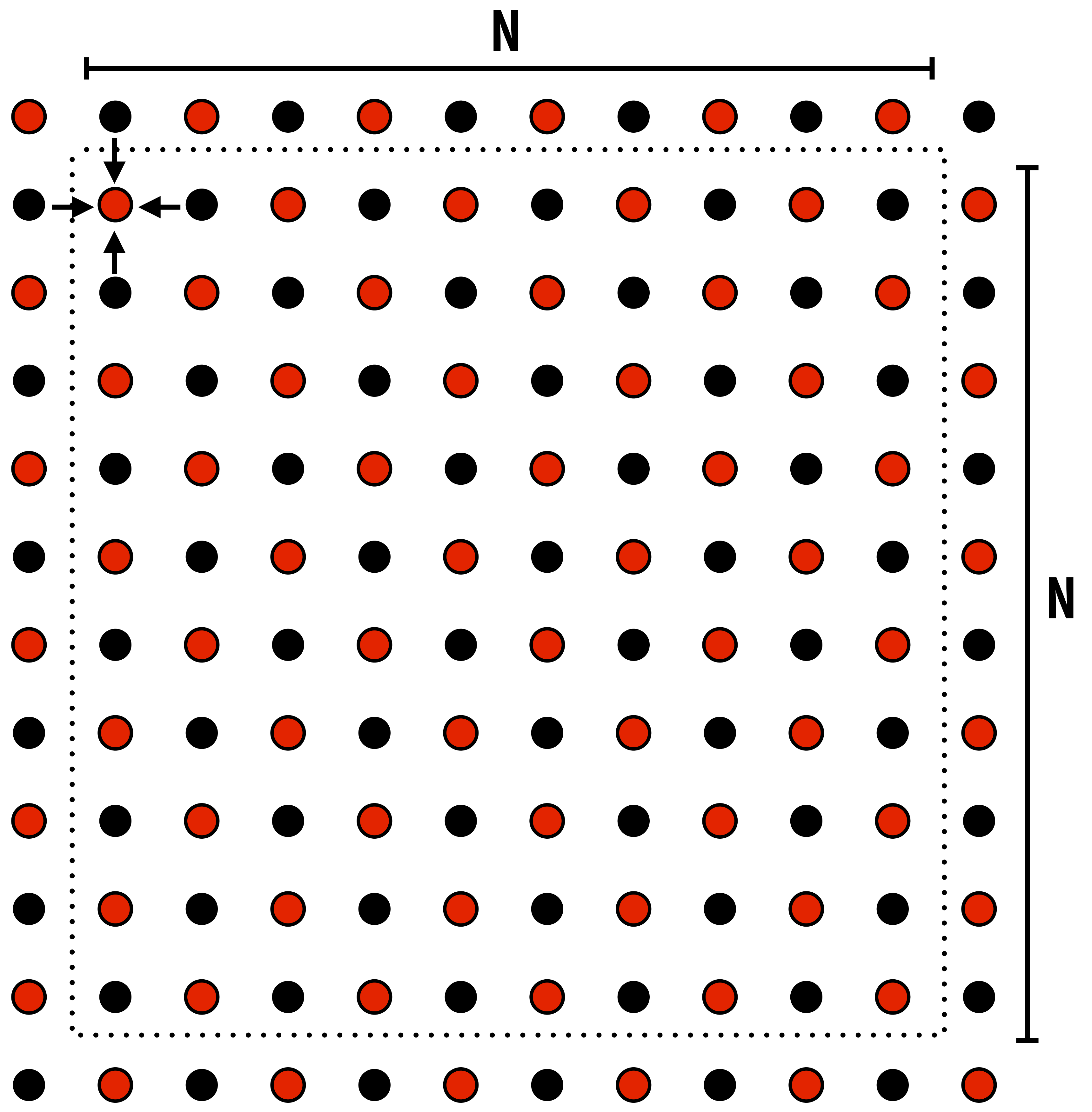
Frequent synchronization (after completing each diagonal)

Let's make life easier on ourselves

- Idea: improve performance by changing the algorithm to one that is more amenable to parallelism
  - Change the order that grid cell cells are updated
  - New algorithm iterates to same solution (approximately), but converges to solution differently
    - Note: floating-point values computed are different, but solution still converges to within error threshold
  - Yes, we needed domain knowledge of the Gauss-Seidel method to realize this change is permissible
    - But this is a common technique in parallel programming

New approach: reorder grid cell update via red-black coloring

Reorder grid traversal: red-black coloring



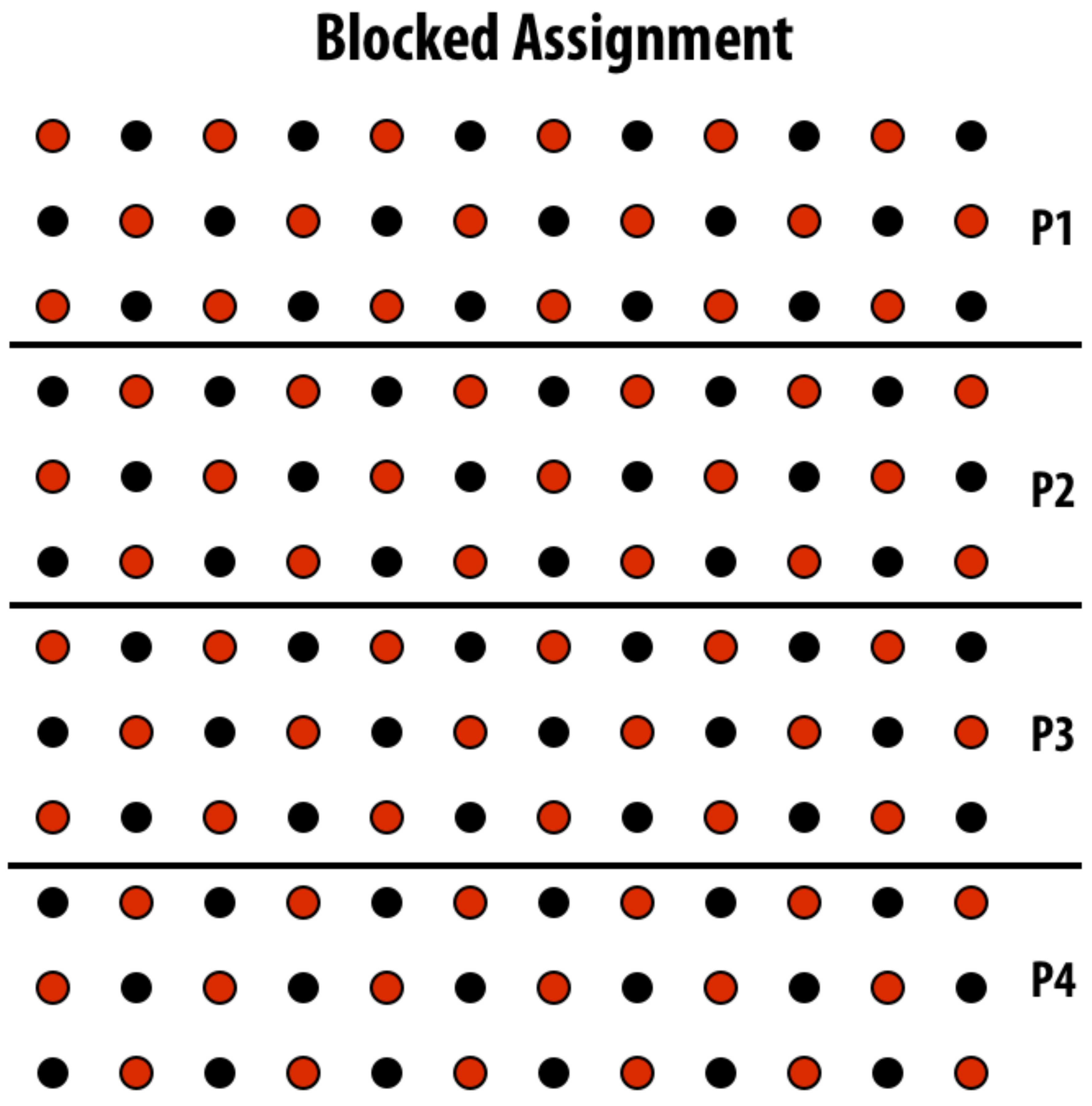
Update all red cells in parallel

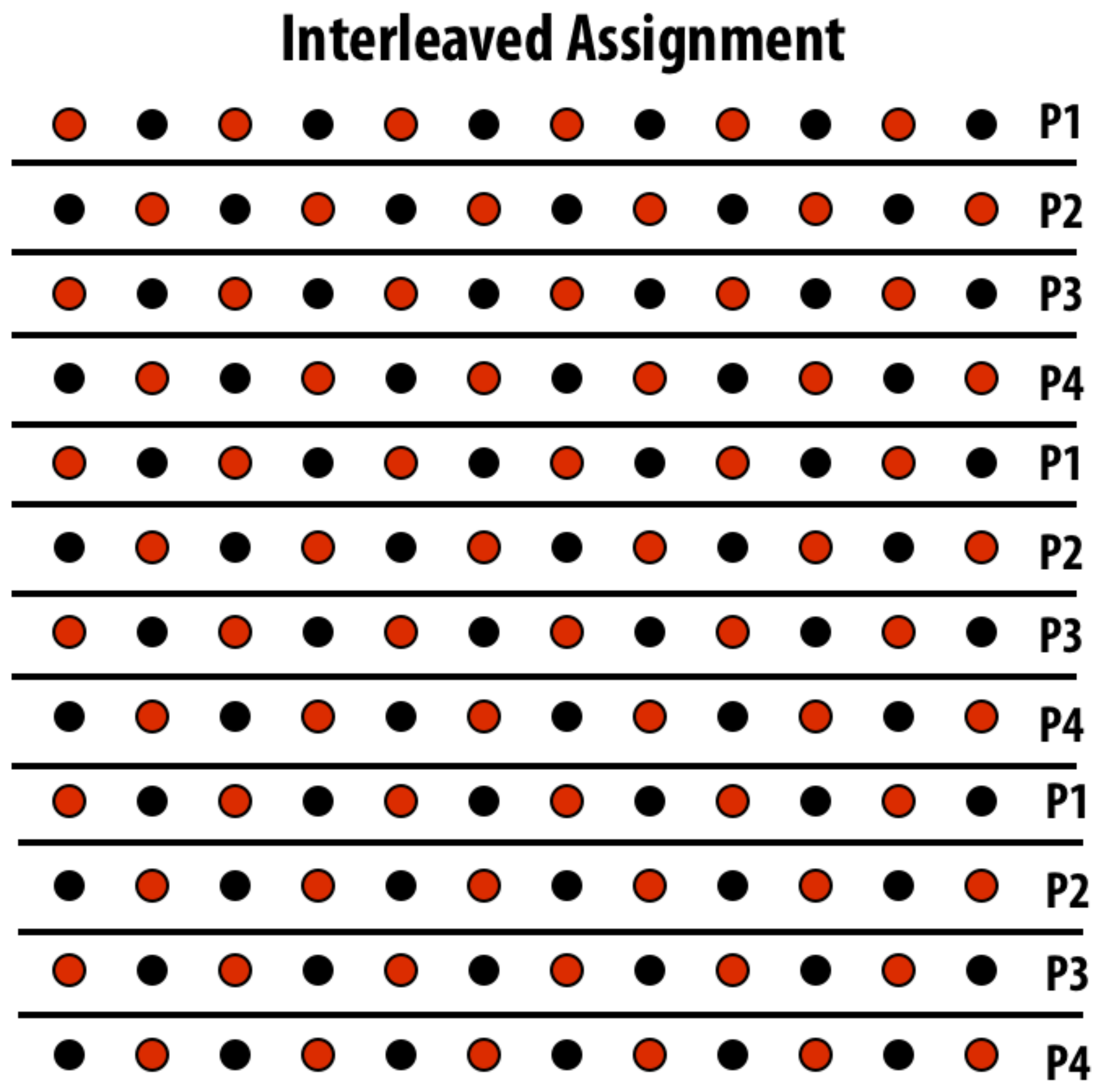
When done updating red cells , update all black cells in parallel (respect dependency on red cells)

Repeat until convergence

Possible assignments of work to processors

Reorder grid traversal: red-black coloring



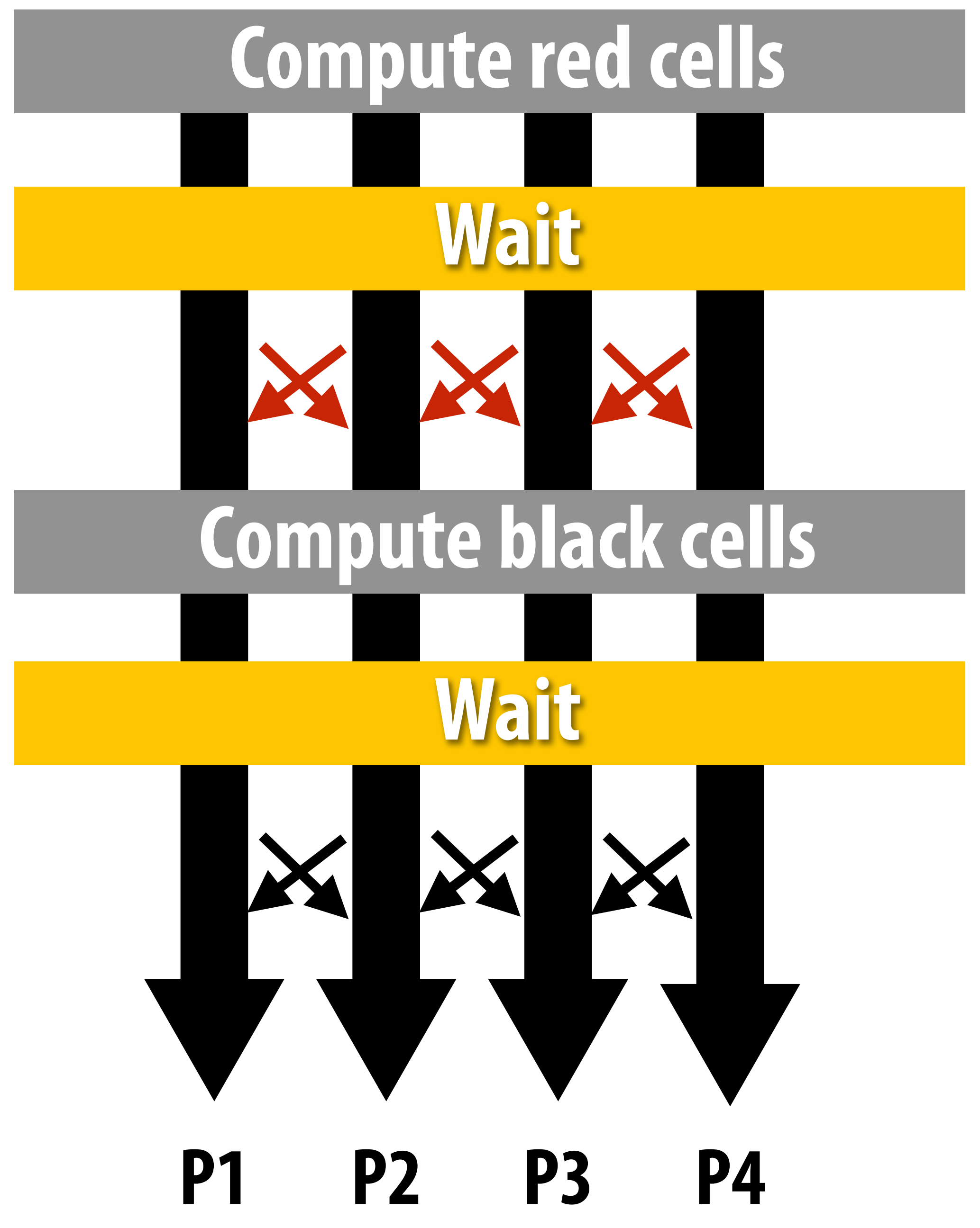


Question: Which is better? Does it matter?

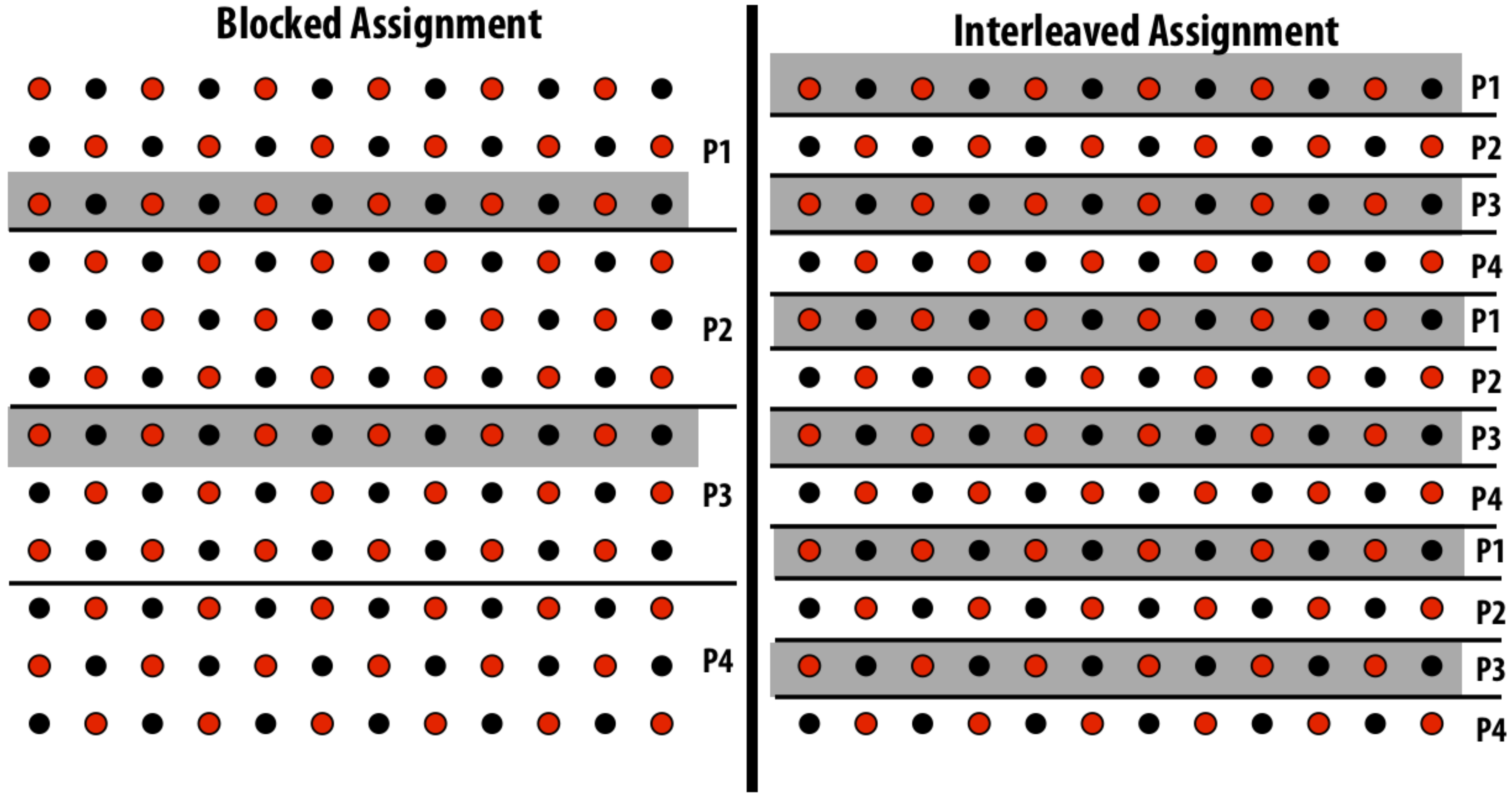
Answer: it depends on the system this program is running on

Consider dependencies in the program

1. Perform red cell update in parallel
2. Wait until all processors done with update
3. Communicate updated red cells to other processors
4. Perform black cell update in parallel
5. Wait until all processors done with update
6. Communicate updated black cells to other processors
7. Repeat



Communication resulting from assignment



▨ = data that must be sent to P2 each iteration

Blocked assignment requires less data to be communicated between processors

Three ways to think about writing this program

- **Data parallel thinking**
- **SPMD / shared address space**
- **Message passing**

Data-parallel expression of solver

Data-parallel expression of grid solver

Note: to simplify pseudocode: just showing red-cell update

```
const int n;

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

void solve(float* A) {

   bool done = false;
   float diff = 0.f;
   while (!done) {
     for_all (red cells (i,j)) {
        float prev = A[i,j];
        A[i,j] = 0.2f * (A[i-1,j] + A[i,j-1] + A[i,j] +
                         A[i+1,j] + A[i,j+1]);
        reduceAdd(diff, abs(A[i,j] - prev));
     }

     if (diff/(n*n) < TOLERANCE)
        done = true;
    }
}
```

Decomposition:
processing individual grid elements constitutes independent work

Assignment: handled by system
One implementation: assign iterations to threads running on each processor

Orchestration: handled by system
(builtin communication primitive: reduceAdd)

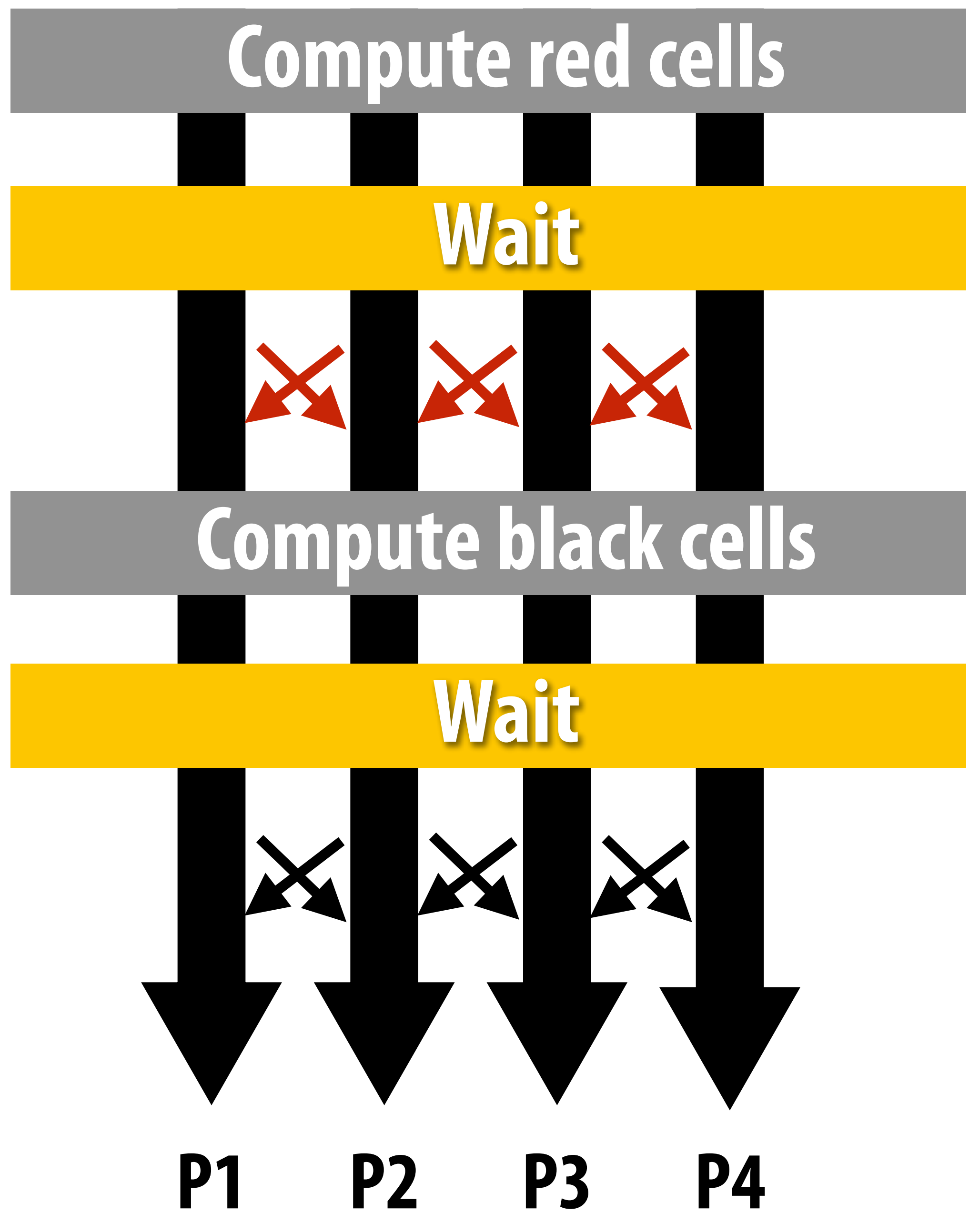
Orchestration: handled by system
(End of for_all block is an implicit wait for all workers before returning to sequential control)

Shared address space (with SPMD threads) expression of solver

Shared address space expression of solver

SPMD execution model

- Programmer is responsible for synchronization
- Common synchronization primitives:
  - Locks (provide mutual exclusion): only one thread in the critical region at a time
  - Barriers: wait for threads to reach this point



Shared address space solver

(pseudocode in SPMD execution model)

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

Assume these are global variables (accessible to all threads)

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

void solve(float* A) {

   int threadId = getThreadId();
   int myMin = 1 + (threadId * n / NUM_PROCESSORS);
   int myMax = myMin + (n / NUM_PROCESSORS)

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

Assume solve function is executed by all threads. (SPMD-style)

Value of threadId is different for each SPMD instance: use value to compute region of grid to work on

Each thread computes the rows it is responsible for updating

Shared address space solver

(pseudocode in SPMD execution model)

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

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

void solve(float* A) {

   int threadId = getThreadId();
   int myMin = 1 + (threadId * n / NUM_PROCESSORS);
   int myMax = myMin + (n / NUM_PROCESSORS)

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

Do you see a potential performance problem with this implementation?

Shared address space solver

(pseudocode in SPMD execution model)

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

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

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

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

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

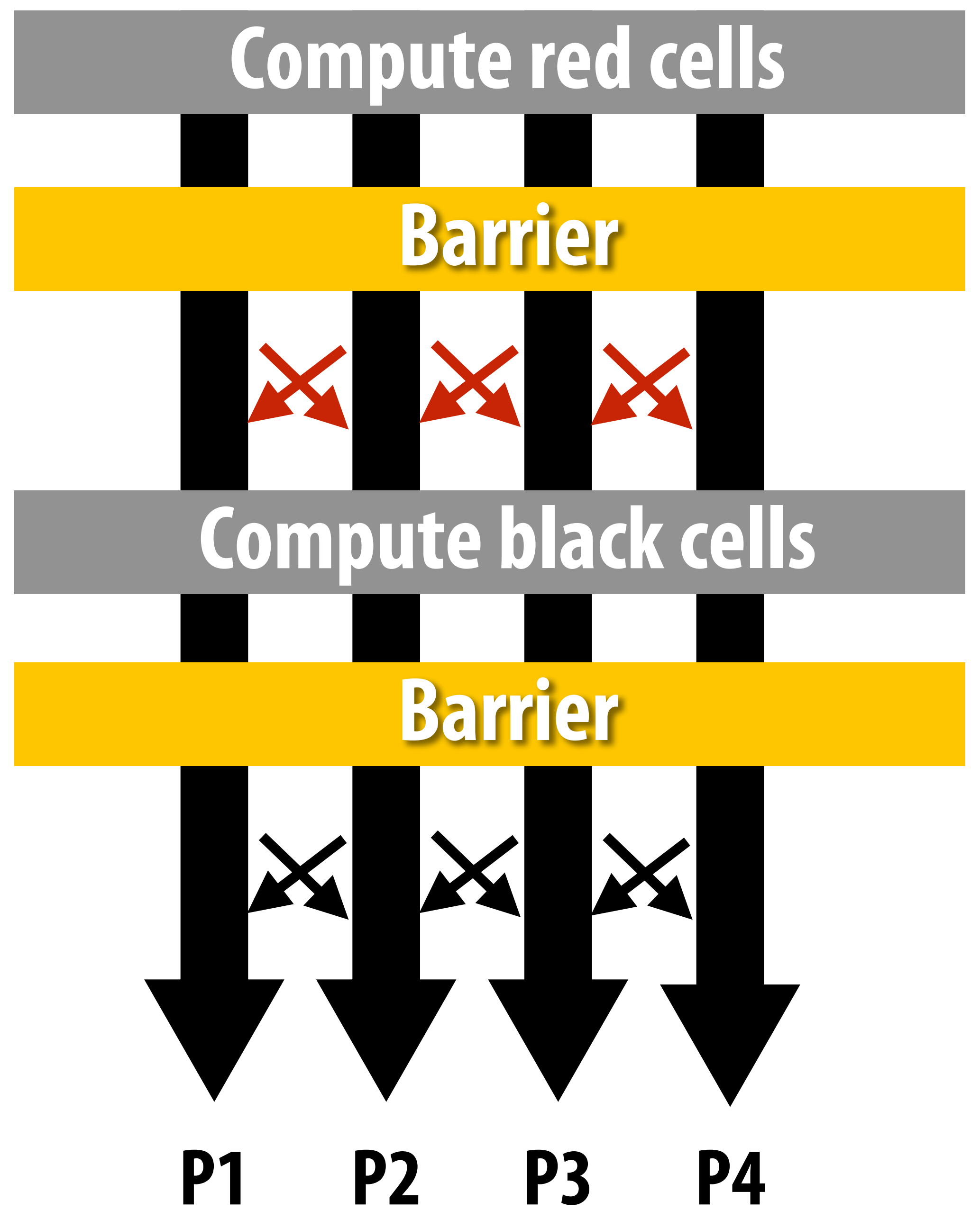
Compute partial sum per worker

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

Grid solver example from: Culler, Singh, and Gupta

Barrier synchronization primitive

- `barrier(num_threads)`
- **Barriers are a way to express dependencies**
- **Barriers divide computation into phases**
- **All computations by all threads before the barrier complete before any computation in any thread after the barrier begins**
  - **In other words, all computations after the barrier are assumed to depend on all computations before the barrier**



Shared address space solver: three barriers

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

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

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

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

Why are there three barriers?

Grid solver example from: Culler, Singh, and Gupta

Shared address space solver: one barrier

```
int     n;                // grid size
bool    done = false;
LOCK    myLock;
BARRIER myBarrier;
float diff[3];  // global diff, but now 3 copies

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


void solve(float* A) {
  float myDiff;   // thread local variable
  int index = 0;  // thread local variable

  diff[0] = 0.0f;
  barrier(myBarrier, NUM_PROCESSORS);  // one-time only: just for init

  while (!done) {
    myDiff = 0.0f;
    //
    // perform computation (accumulate locally into myDiff)
    //
    lock(myLock);
    diff[index] += myDiff;    // atomically update global diff
    unlock(myLock);
    diff[(index+1) % 3] = 0.0f;
    barrier(myBarrier, NUM_PROCESSORS);
    if (diff[index]/(n*n) < TOLERANCE)
      break;
    index = (index + 1) % 3;
  }
}
```

Idea:

Remove dependencies by using different `diff` variables in successive loop iterations

Trade off footprint for removing dependencies! (a common parallel programming technique)

Grid solver implementation in two programming models

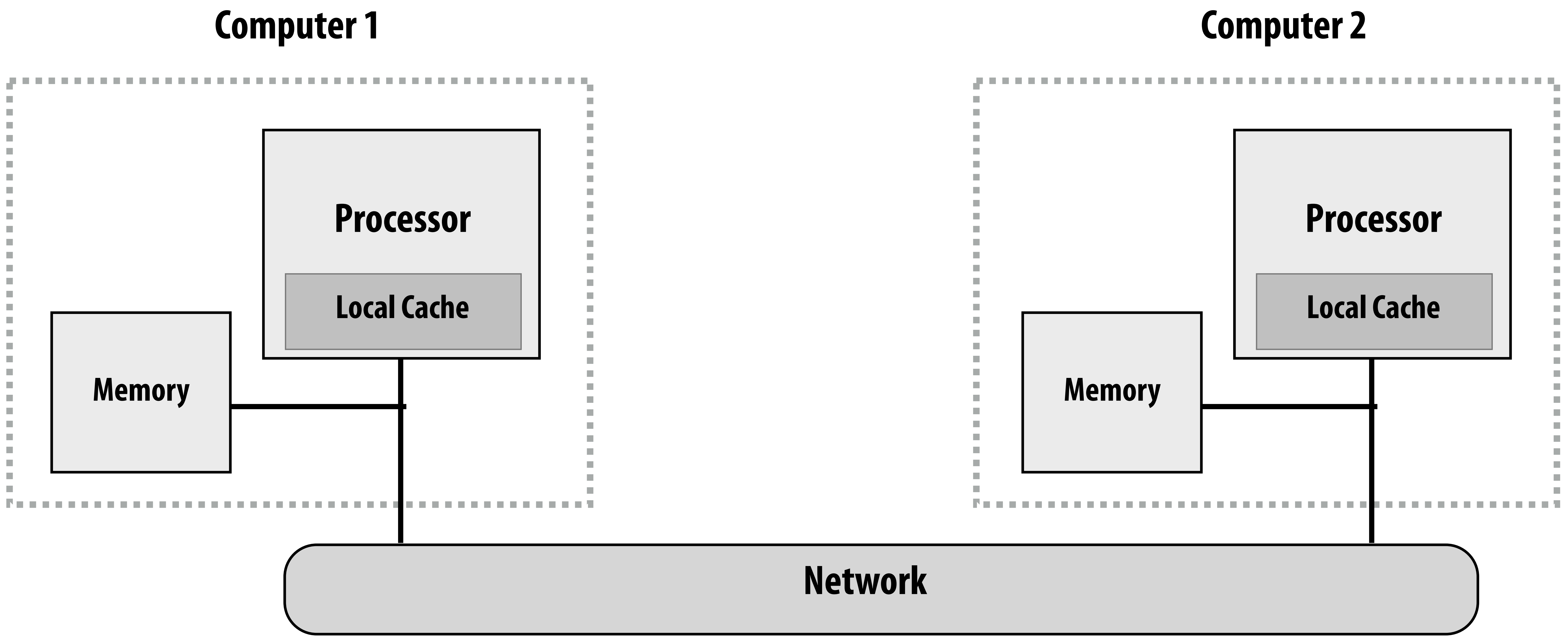
- **Data-parallel programming model**
  - Synchronization:
    - Single logical thread of control, but iterations of `forall` loop <u>may</u> be parallelized by the system (implicit barrier at end of `forall` loop body)
  - Communication
    - Implicit in loads and stores (like shared address space)
    - Special built-in primitives for more complex communication patterns: e.g., reduce

- **Shared address space**
  - Synchronization:
    - Mutual exclusion required for shared variables (e.g., via locks)
    - Barriers used to express dependencies (between phases of computation)
  - Communication
    - Implicit in loads/stores to shared variables

Message passing expression of solver

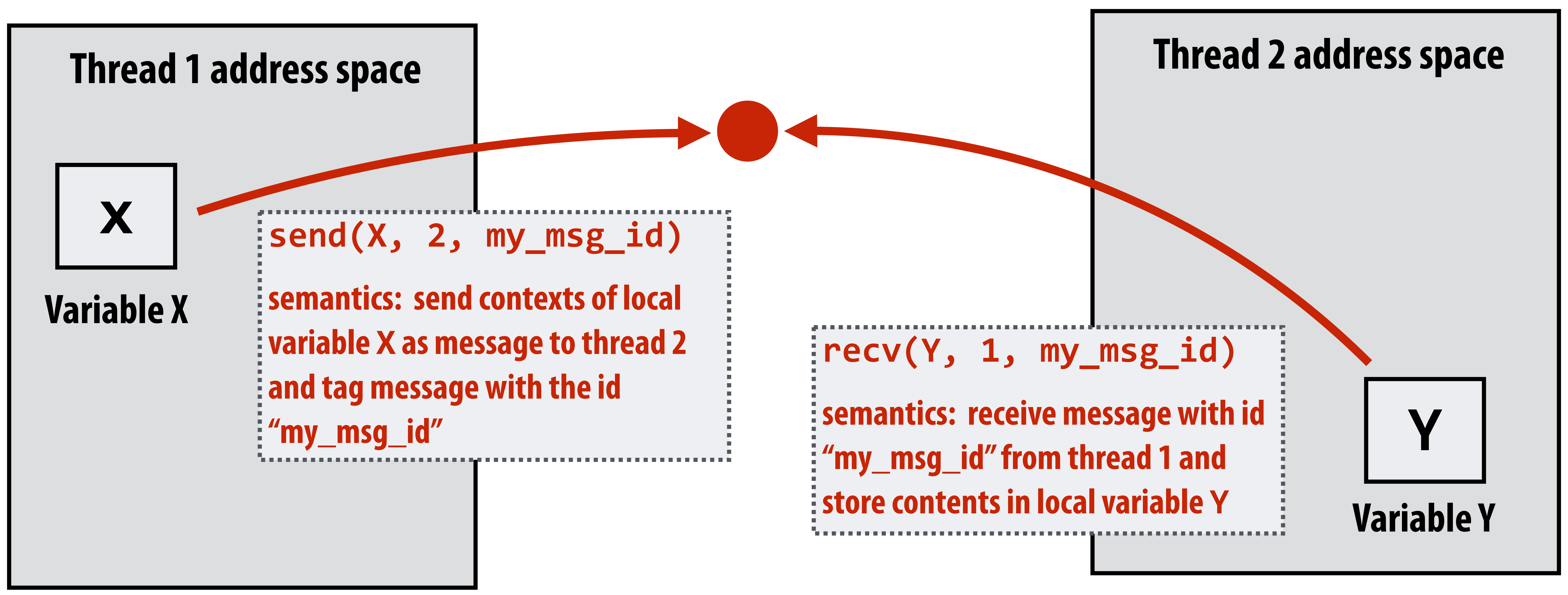
Let's think about expressing a parallel grid solver with communication via messages

One possible message passing machine configuration: a cluster of two machines



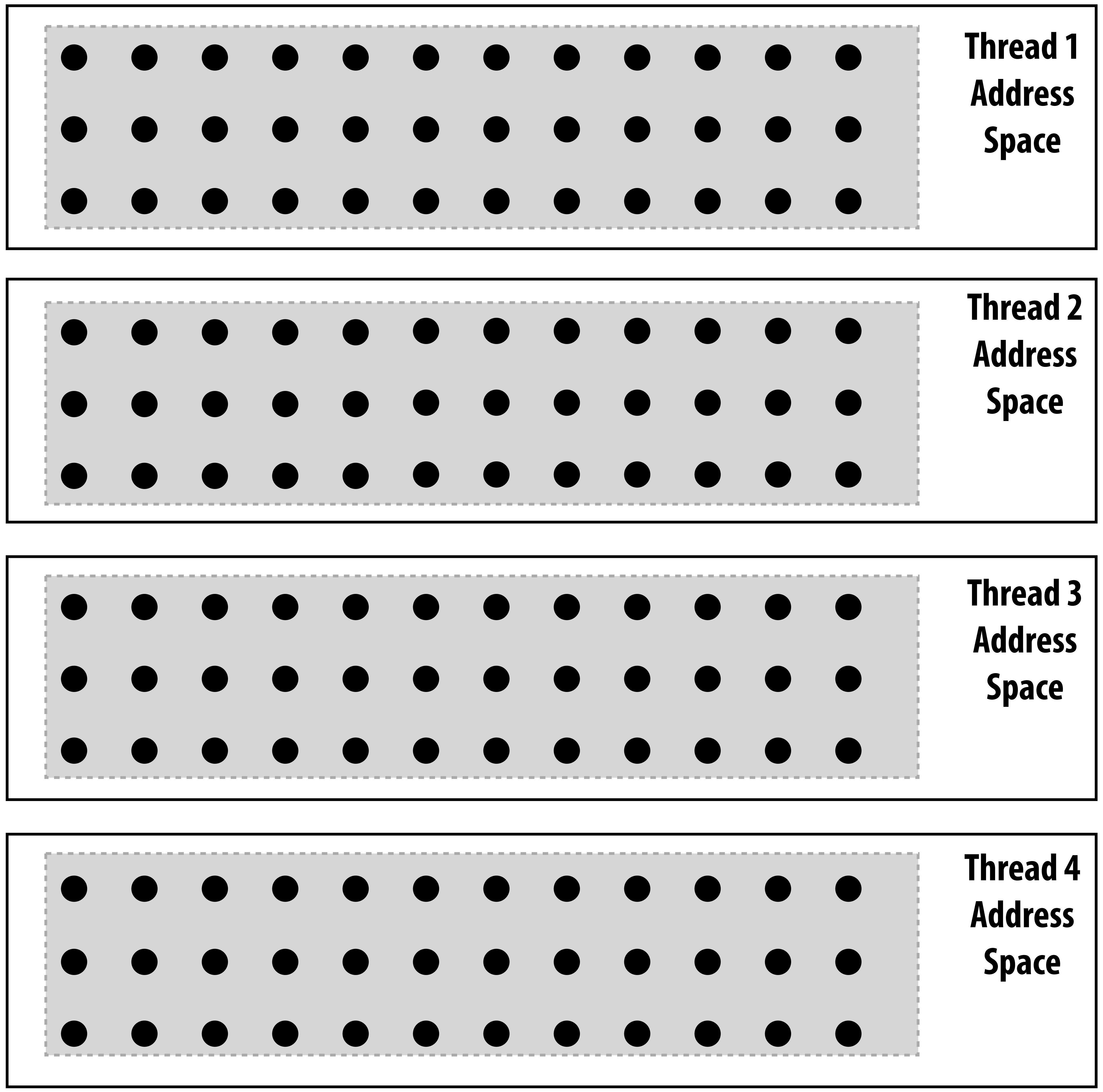
Review: message passing model

- Threads operate within their own private address spaces
- Threads communicate by sending/receiving messages
  - send: specifies recipient, buffer to be transmitted, and optional message identifier ("tag")
  - receive: sender, specifies buffer to store data, and optional message identifier
  - Sending messages is the only way to exchange data between threads 1 and 2 Why?


Thread 1 address space
X
Variable X
send(X, 2, my_msg_id)
semantics: send contexts of local variable X as message to thread 2 and tag message with the id "my_msg_id"
Thread 2 address space
recv(Y, 1, my_msg_id)
semantics: receive message with id "my_msg_id" from thread 1 and store contents in local variable Y
Y
Variable Y

(Communication operations shown in red)

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



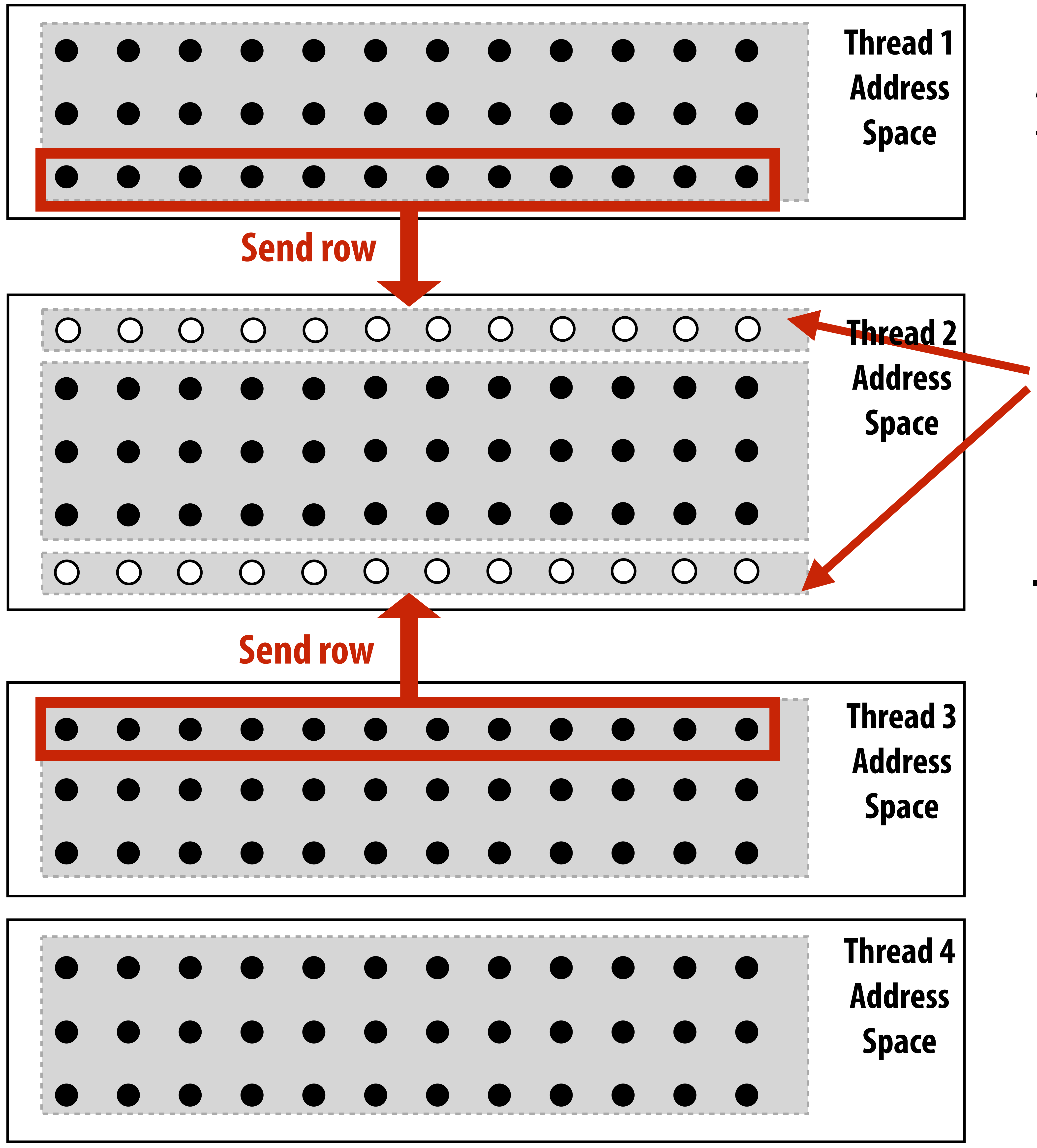
In this figure: four threads

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

Grid data stored in four separate address spaces (four private arrays)



Example:

After processing of red cells is complete, thread 1 and thread 3 send one 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], bytes, tid-1);
recv(&local_data[rows_per_thread+1], bytes, tid+1);

// Thread 2 now has data necessary to perform
// its 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

Example pseudocode from: Culler, Singh, and Gupta

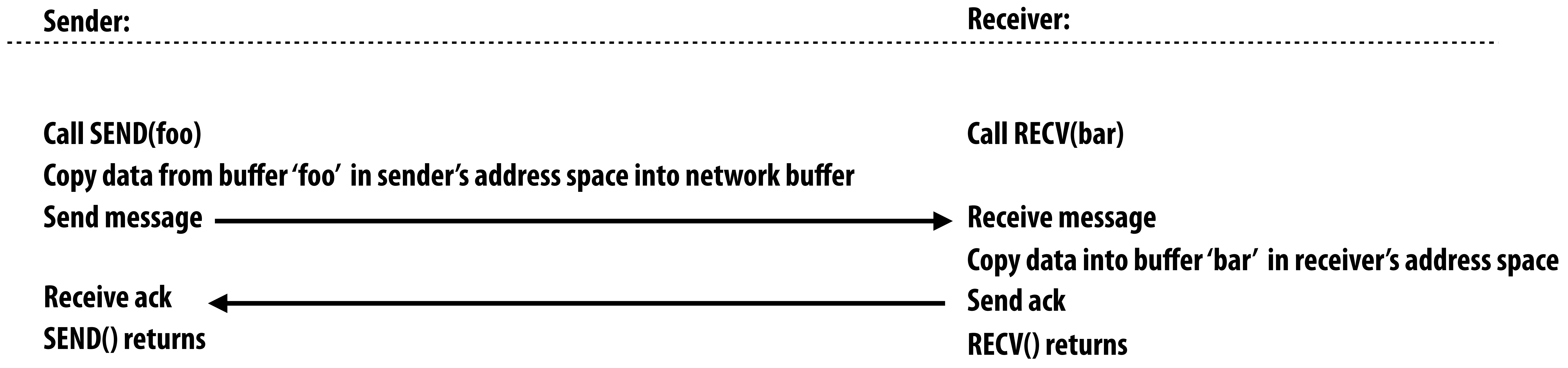
Notes on message passing example

- **Computation**
  - Array indexing is relative to local address space
- **Communication:**
  - Performed by sending and receiving messages
  - Bulk transfer: communicate entire rows at a time
- **Synchronization:**
  - Performed by sending and receiving messages
  - Consider how to implement mutual exclusion, barriers, flags using messages

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
Send message
Receive ack
SEND() returns
Call RECV(bar)
Receive message
Copy data into buffer 'bar' in receiver's address space
Send ack
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?

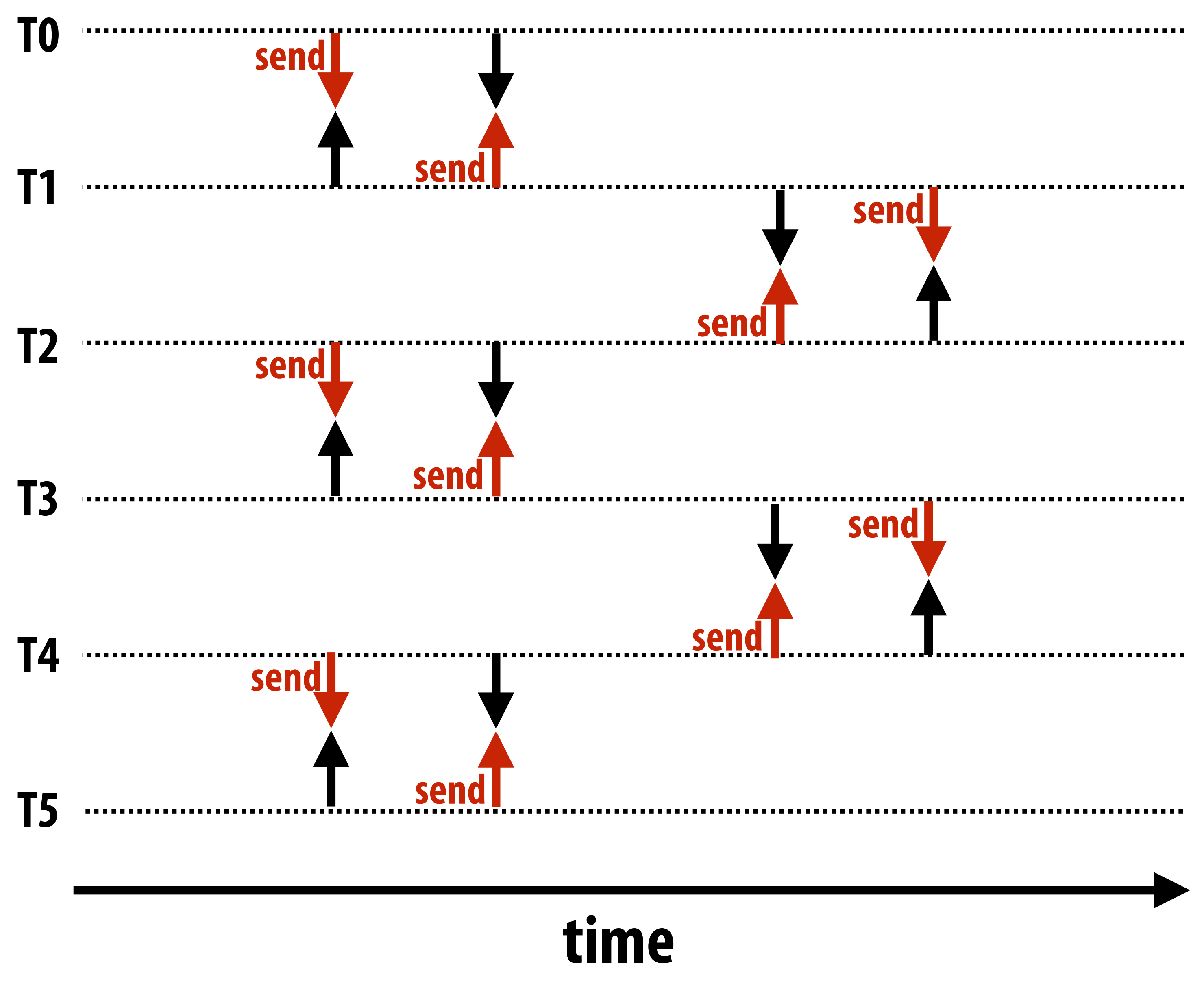
How can we fix it?

(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();
    }

    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;
       if (int i=1; i<gen_num_threads()-1; i++)
         send(&done, sizeof(bool), i, MSD_ID_DONE);
    }
  }
}
```

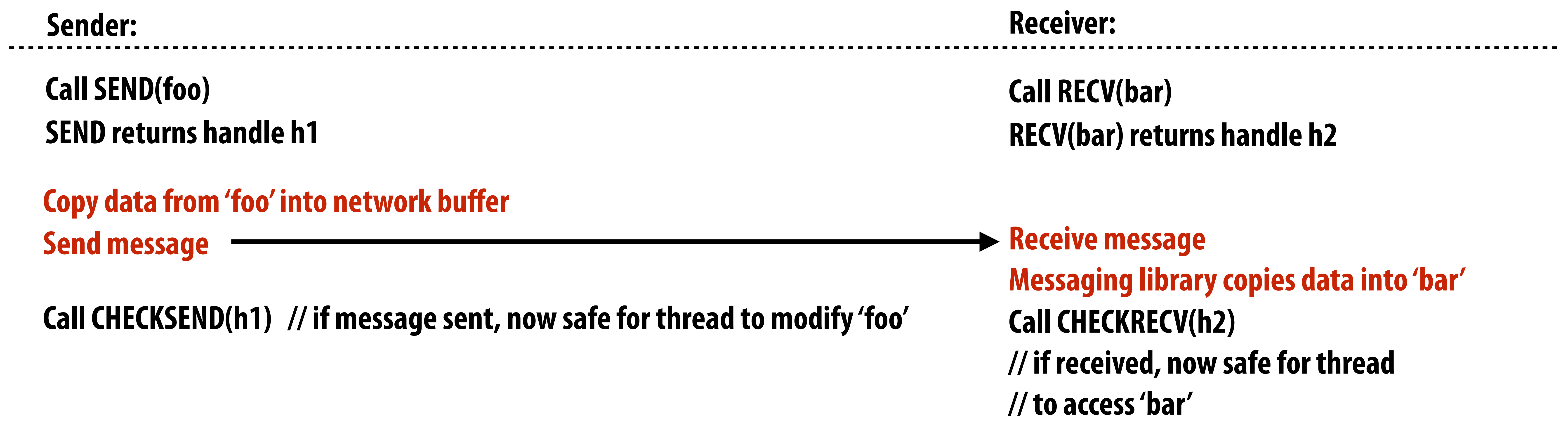
Send and receive ghost rows to "neighbor threads"
Even-numbered threads send, then receive
Odd-numbered thread recv, then send



Example pseudocode from: Culler, Singh, and Gupta

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:

Call SEND(foo)
SEND returns handle h1

Copy data from 'foo' into network buffer
Send message

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

Receiver:

Call RECV(bar)
RECV(bar) returns handle h2

Receive message
Messaging library copies data into 'bar'
Call CHECKRECV(h2)
// if received, now safe for thread
// to access 'bar'

RED TEXT = executes concurrently with application thread

When I talk about communication, I'm not just referring to messages between machines.
(e.g., in a datacenter)

More examples:
Communication between cores on a chip
Communication between a core and its cache
Communication between a core and memory

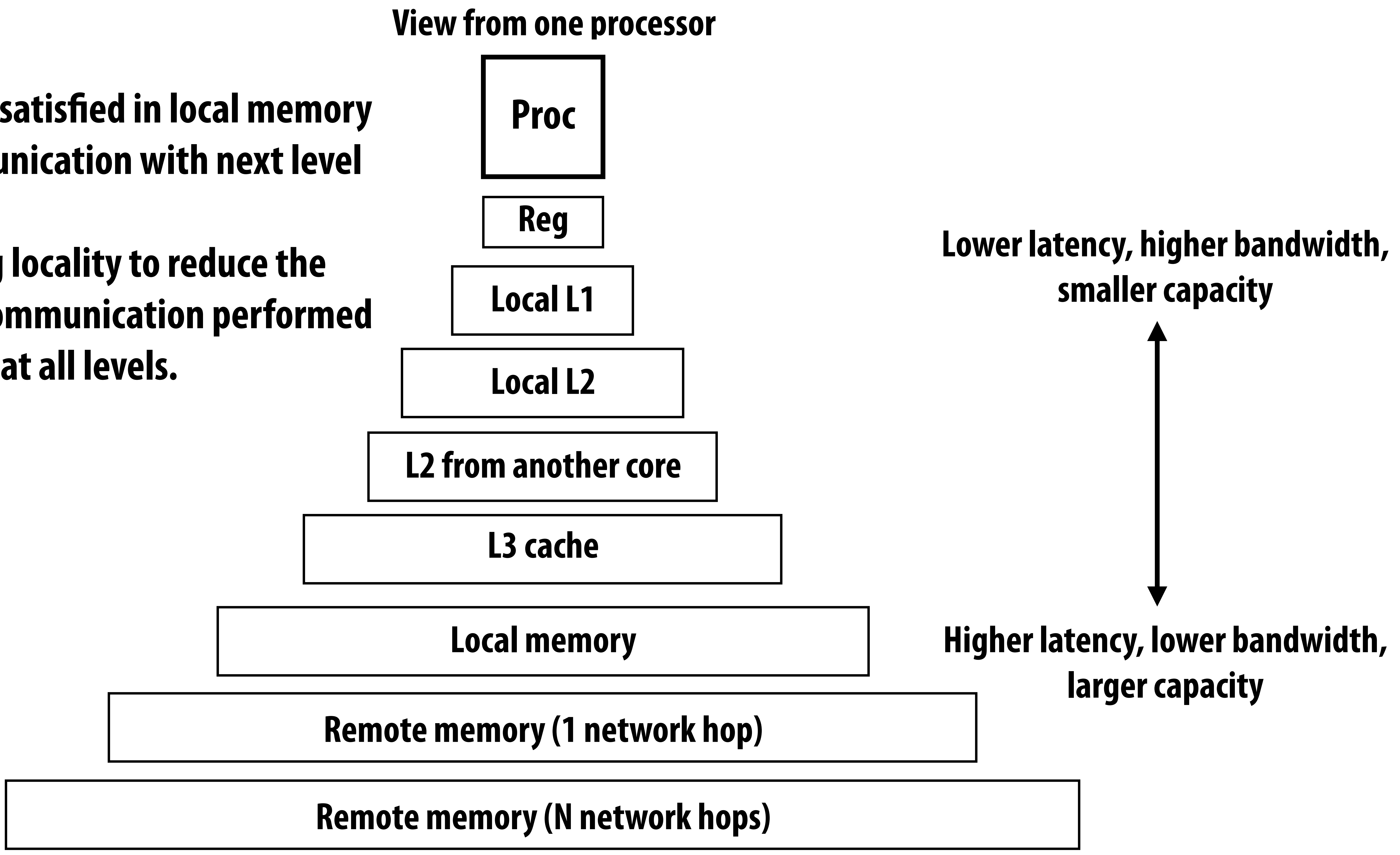
Think of a parallel system as an extended memory hierarchy

I want you to think of "communication" generally:

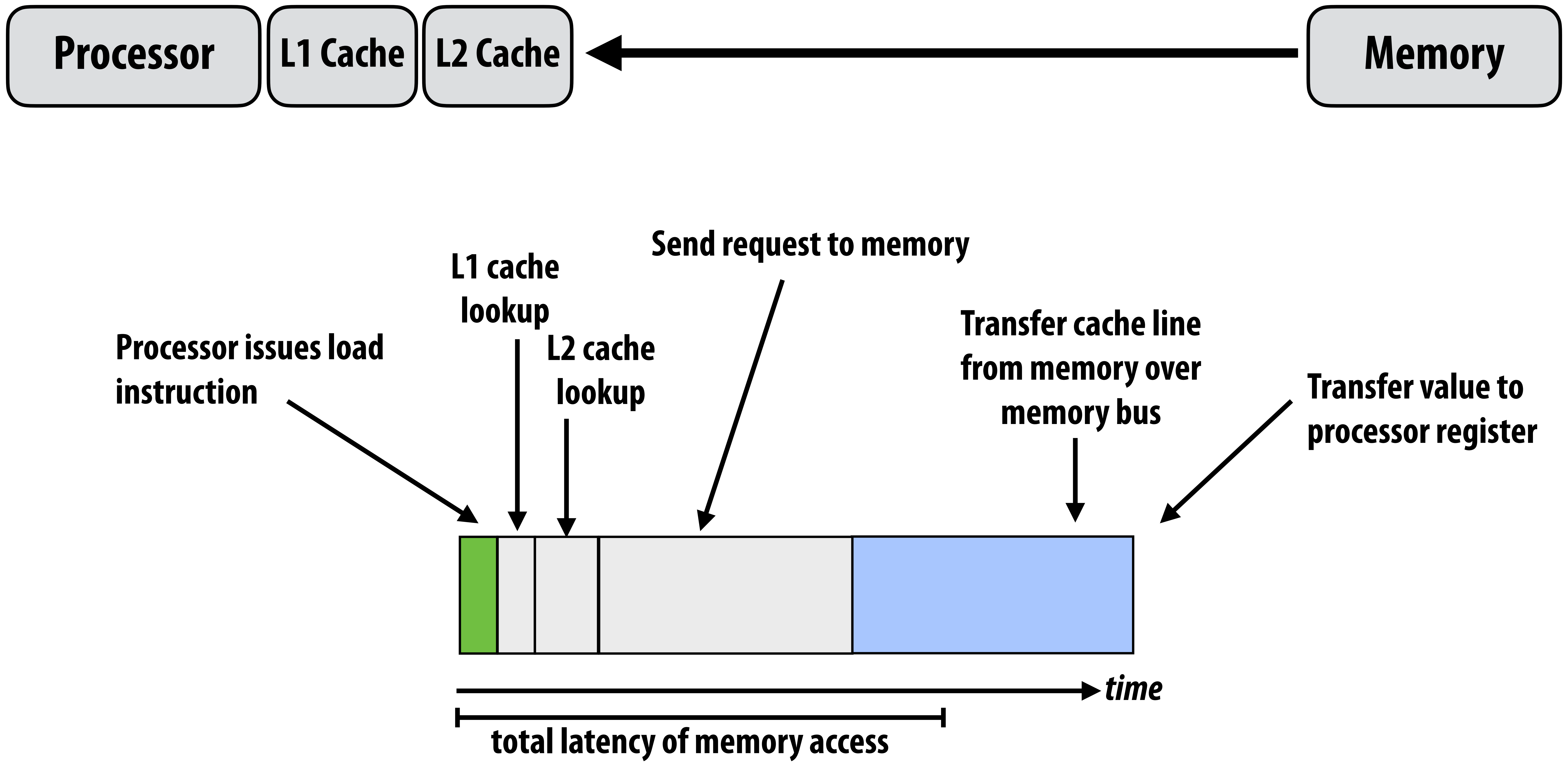
- Communication between a processor and its cache
- Communication between processor and memory (e.g., memory on same machine)
- Communication between processor and a remote memory
  (e.g., memory on another node in the cluster, accessed by sending a network message)

Accesses not satisfied in local memory cause communication with next level

So managing locality to reduce the amount of communication performed is important at all levels.

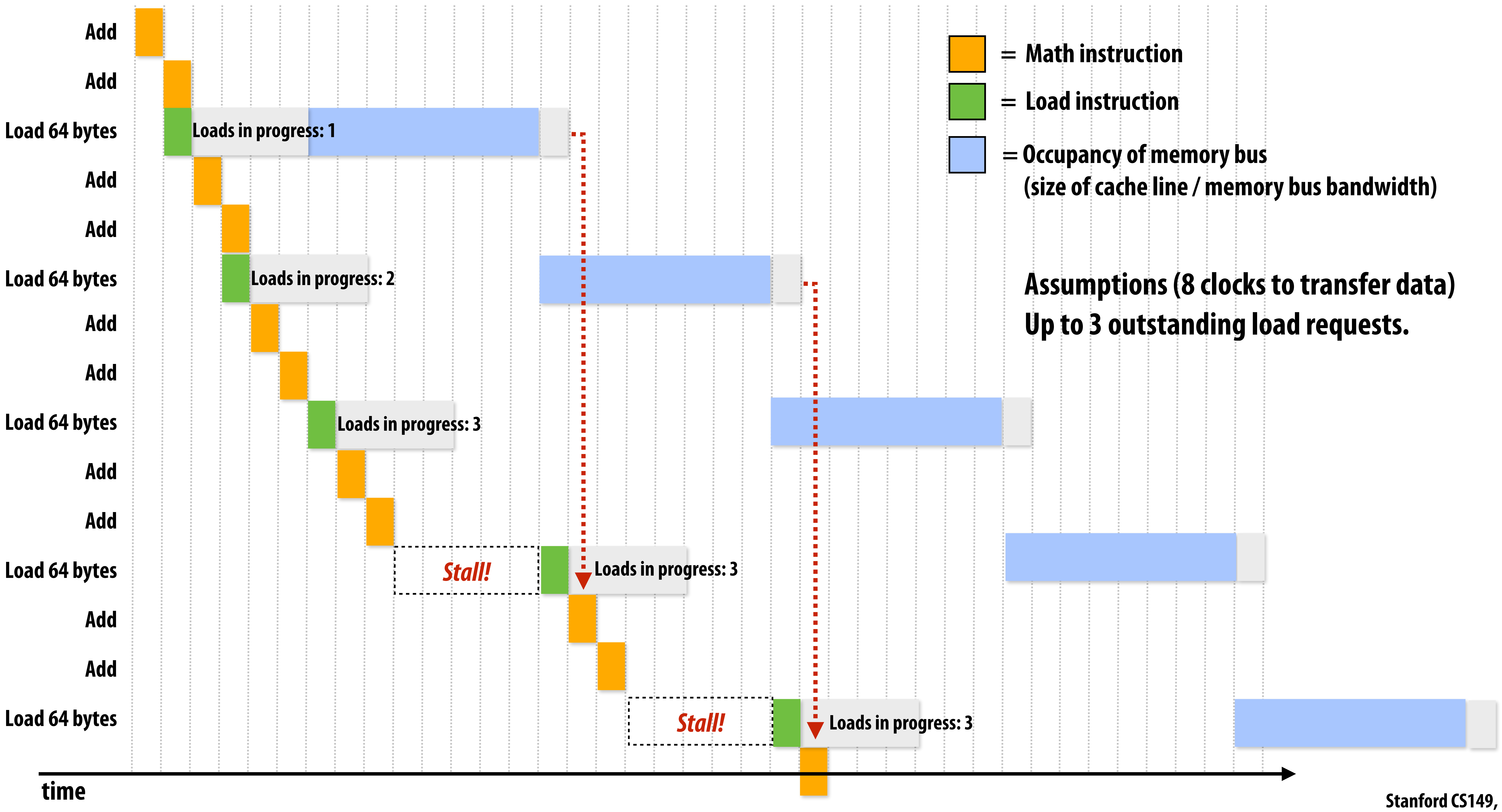

View from one processor
Proc
Reg
Local L1
Local L2
L2 from another core
L3 cache
Local memory
Remote memory (1 network hop)
Remote memory (N network hops)
Lower latency, higher bandwidth, smaller capacity
Higher latency, lower bandwidth, larger capacity

One example: CPU to memory communication


Processor
L1 Cache
L2 Cache
Memory
Send request to memory
L1 cache lookup
L2 cache lookup
Processor issues load instruction
Transfer cache line from memory over memory bus
Transfer value to processor register
time
total latency of memory access

🟦 = Time to send cache line over memory bus

Consider a processor that can do one add per clock (+ can co-issue LD)



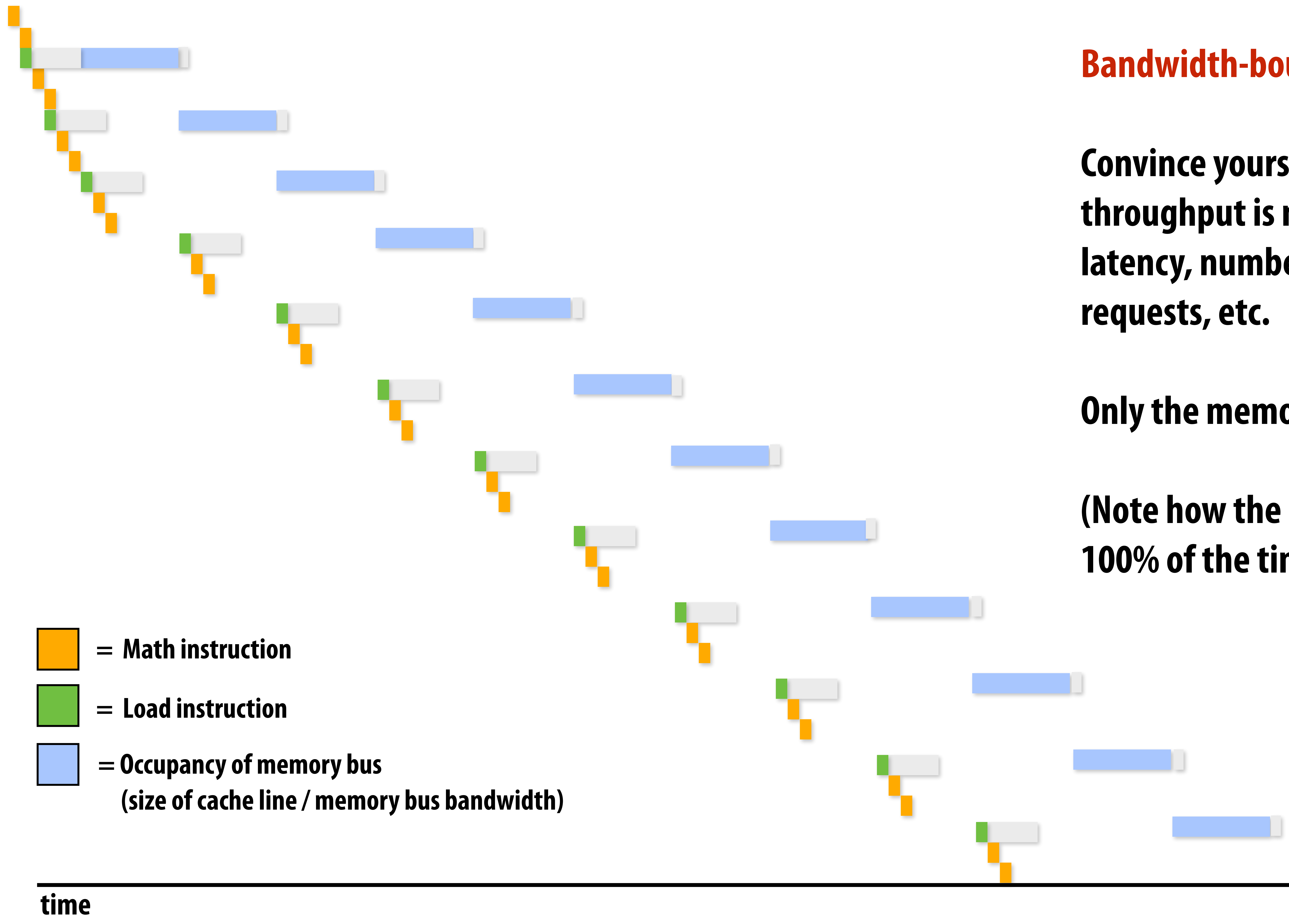
🟧 = Math instruction

🟩 = Load instruction

🟦 = Occupancy of memory bus (size of cache line / memory bus bandwidth)

Assumptions (8 clocks to transfer data)
Up to 3 outstanding load requests.

Rate of math instructions limited by available bandwidth


🟧 = Math instruction
🟩 = Load instruction
🟦 = Occupancy of memory bus
(size of cache line / memory bus bandwidth)
time

Bandwidth-bound execution!

Convince yourself that the instruction throughput is not impacted by memory latency, number of outstanding memory requests, etc.

Only the memory bandwidth!!!

(Note how the memory system is occupied 100% of the time)

Good questions about the previous slide

- **How do you tell from the figure that the memory bus is fully utilized?**
- **How would you illustrate higher memory latency (keep in mind memory requests are pipelined and memory bus bandwidth is not changed)?**
- **How would the figure change if memory bus bandwidth was increased?**
- **Would there still be processor stalls if the ratio of math instructions to load instructions was significantly increased? Why?**

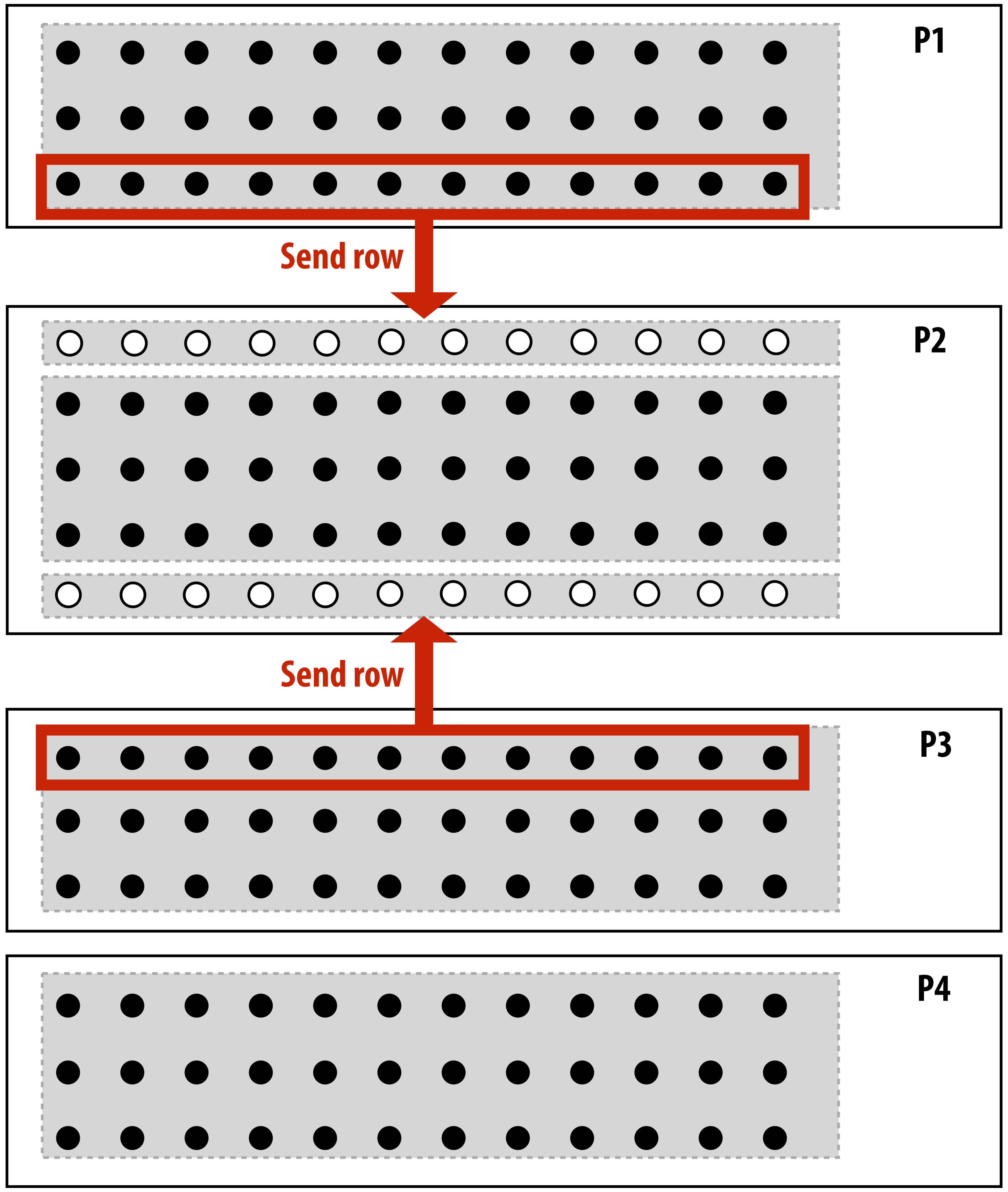
Communication-to-computation ratio

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

- 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 multiply example from the end of lecture 2)

Two reasons for communication: inherent vs. artifactual communication

Inherent communication



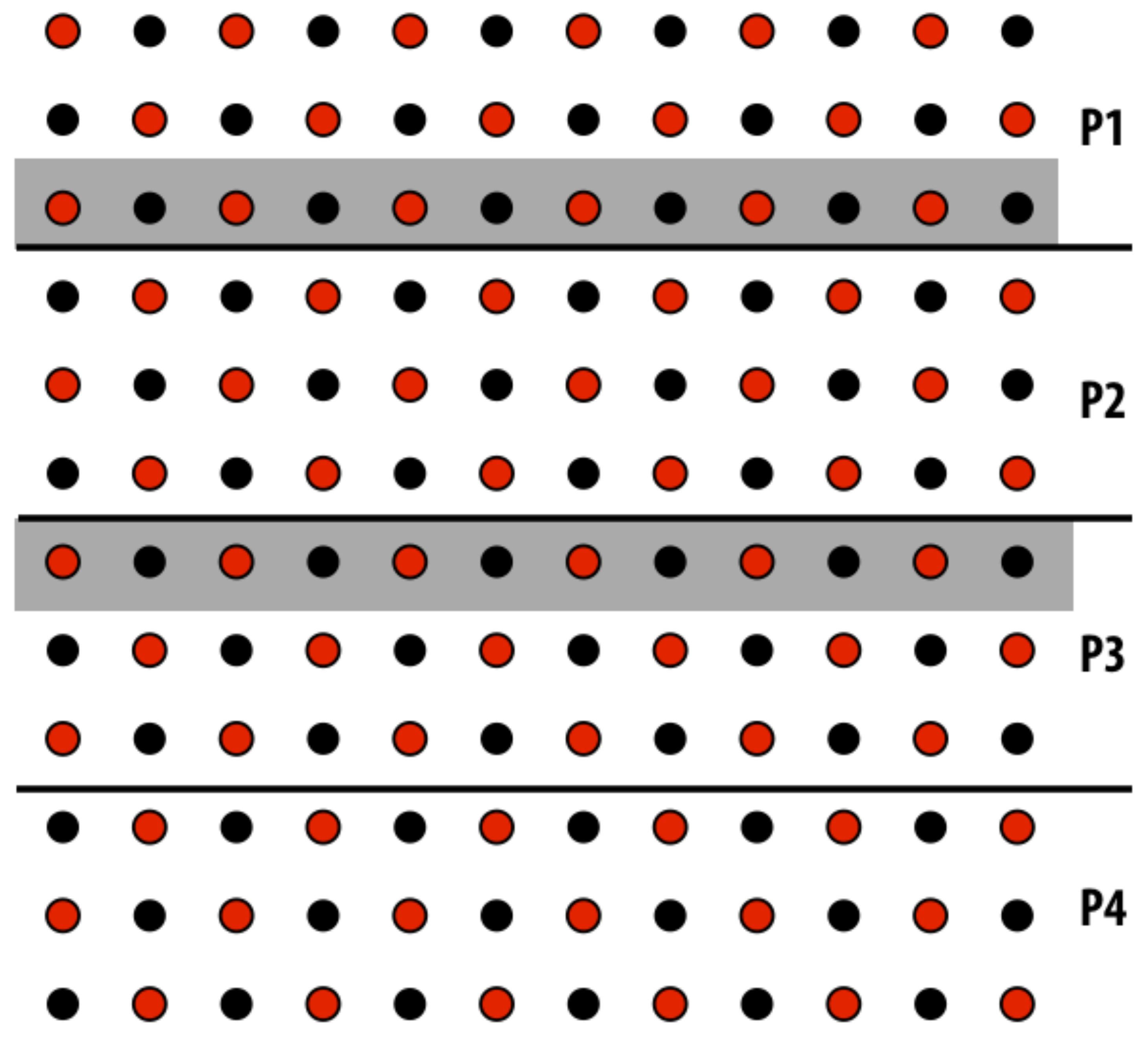
Communication that must occur in a parallel algorithm. The communication is fundamental to the algorithm.

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

Reducing inherent communication

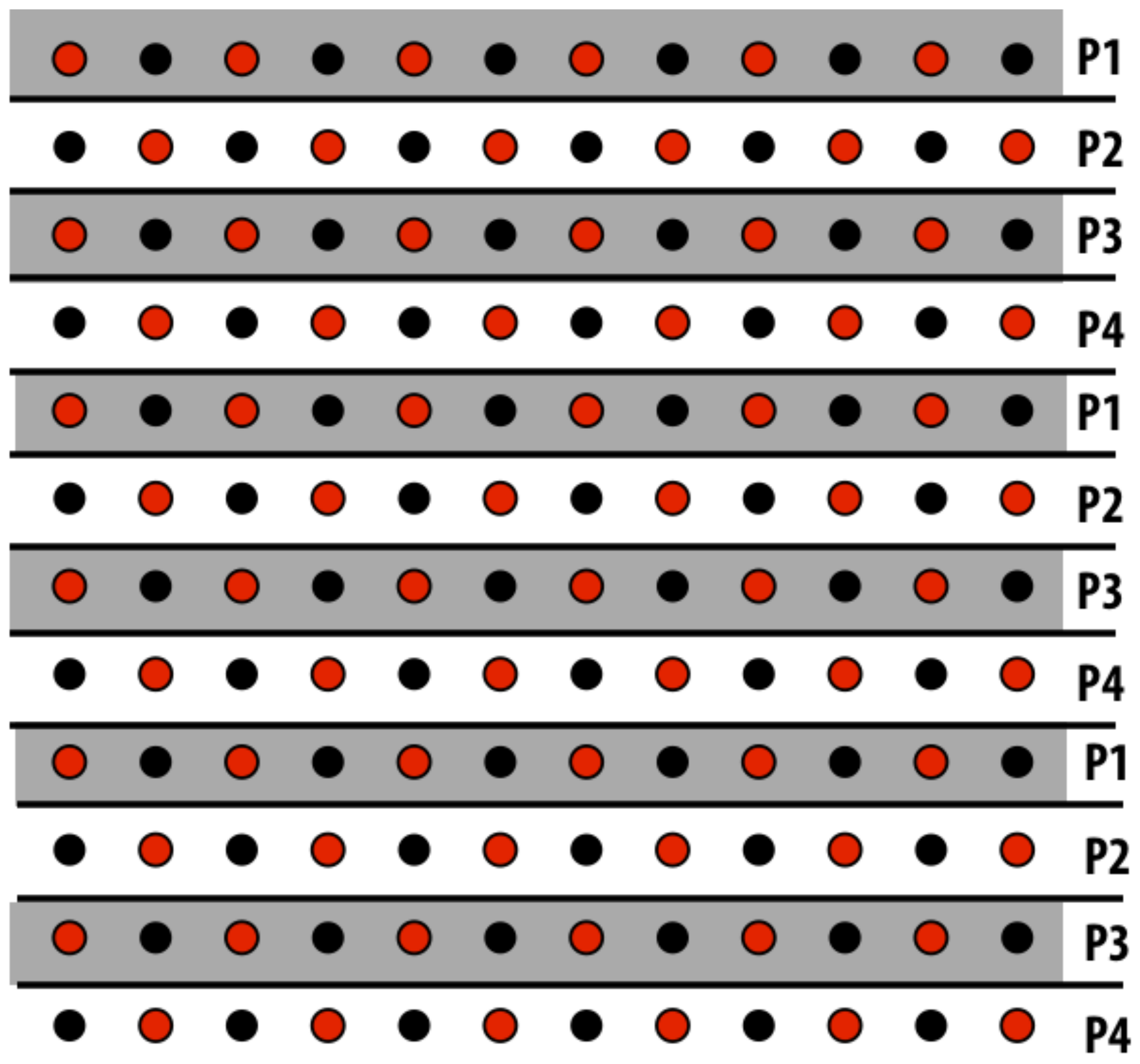
Good assignment decisions can reduce inherent communication (increase arithmetic intensity)

1D blocked assignment: N x N grid



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

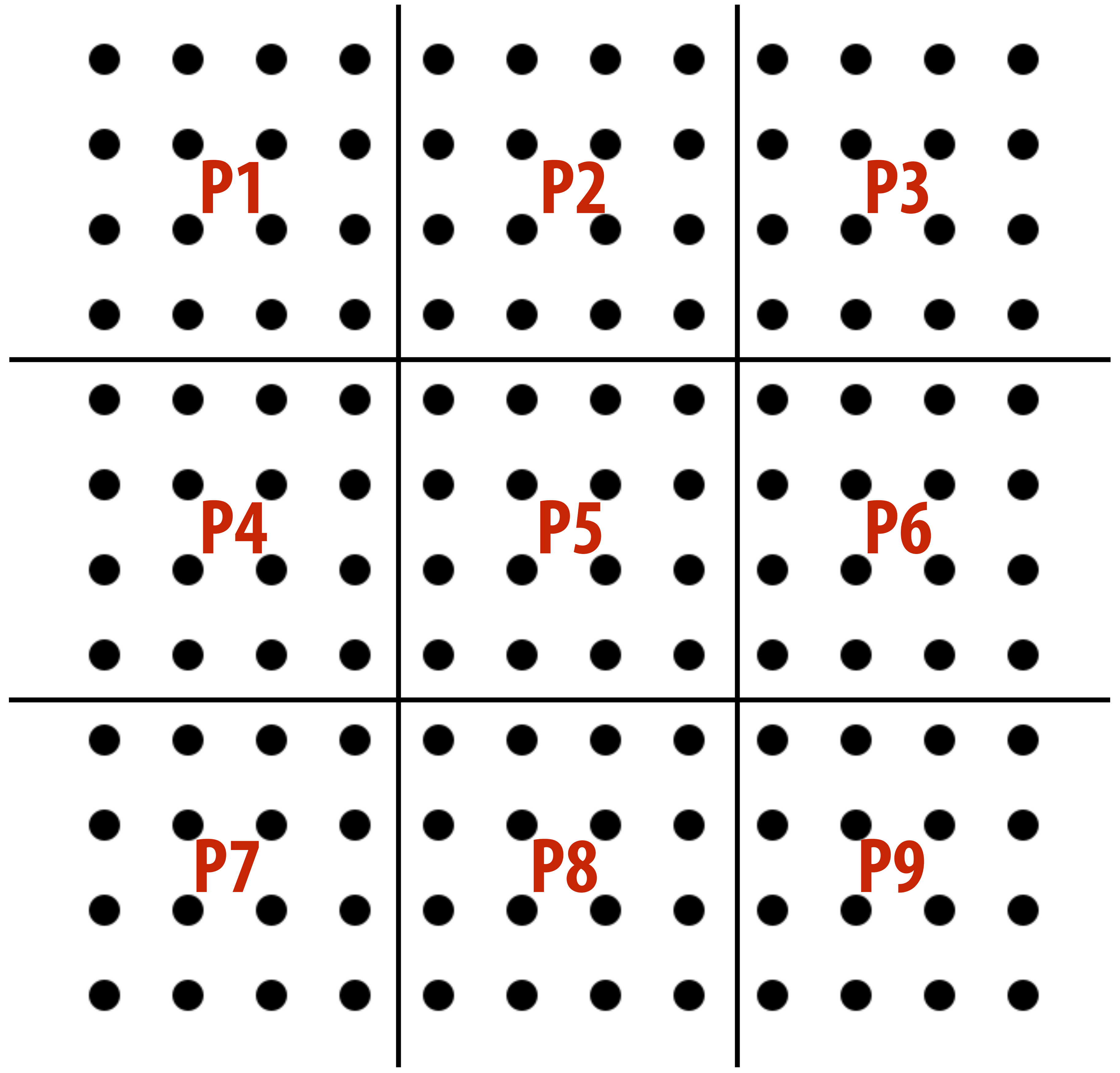
1D interleaved assignment: N x N grid



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

Reducing inherent communication

2D blocked assignment: N x N grid



N^2 elements

P processors

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

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

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

Asymptotically better communication scaling than 1D blocked assignment

Communication costs increase sub-linearly with P

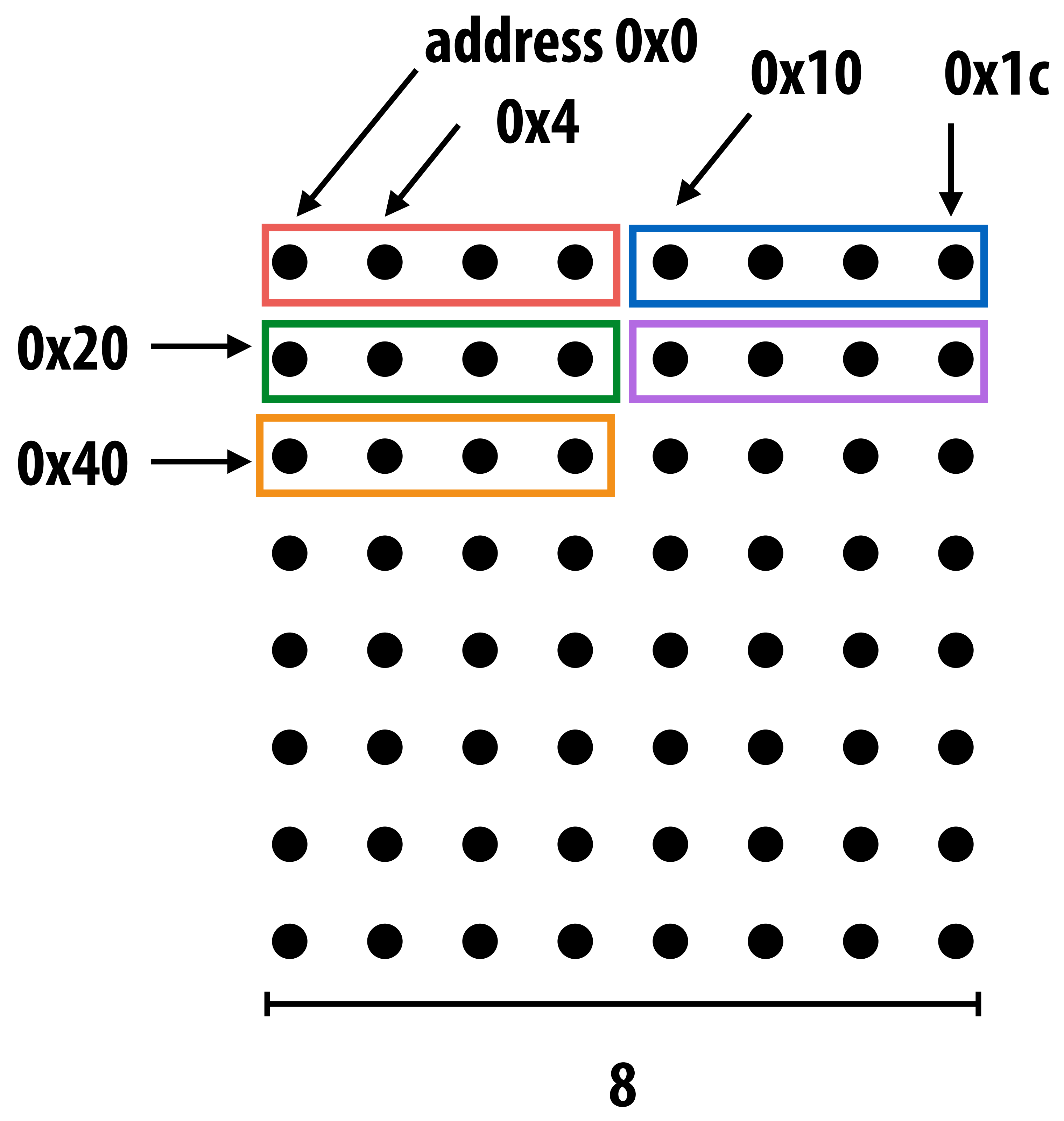
Assignment captures 2D locality of algorithm

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)**

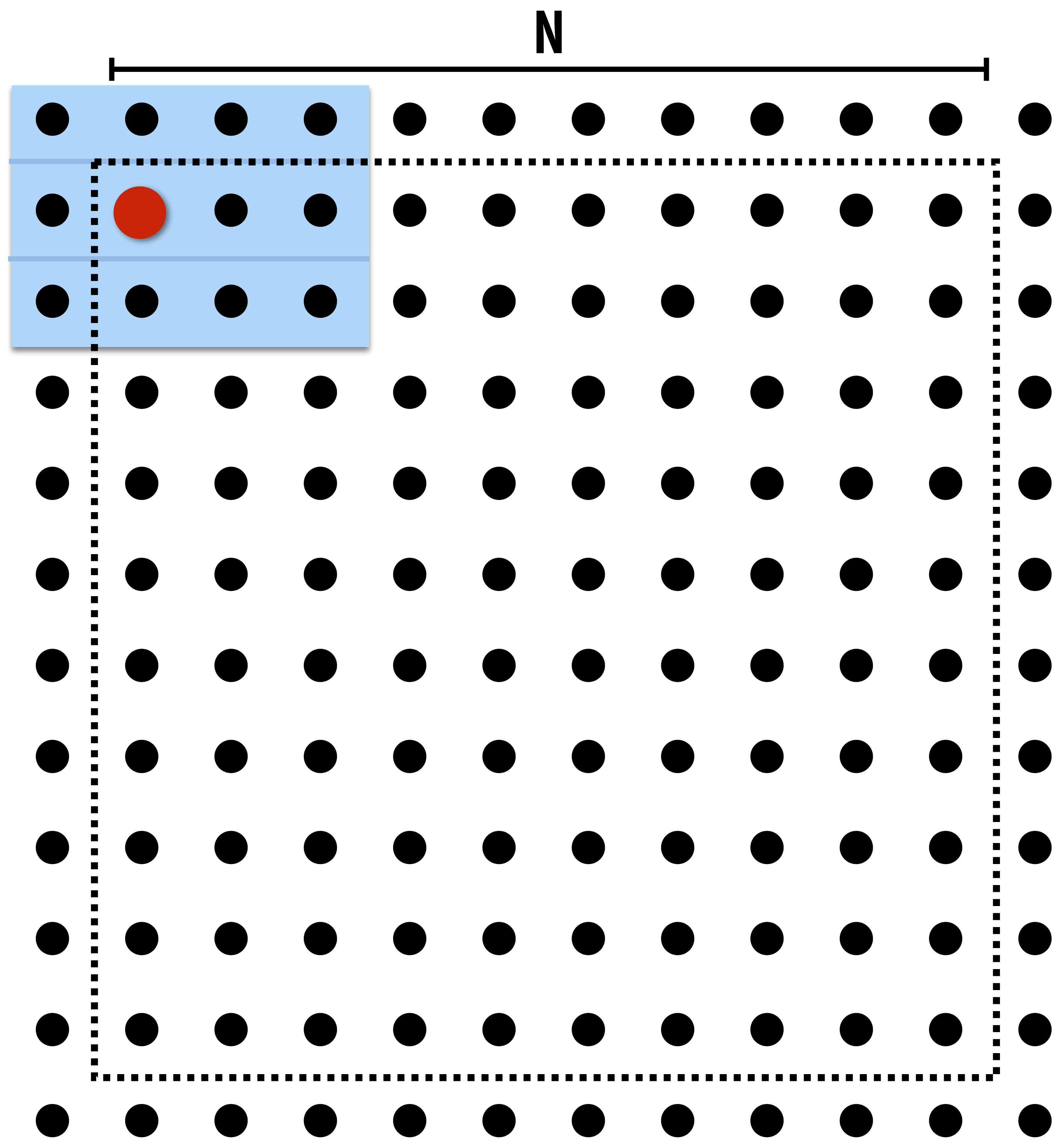
Cache review



Consider 4-byte elements
Consider a cache with 16-byte cache lines and a total capacity of 32 bytes (2 lines fit in cache)
Least recently used (LRU) replacement policy

Address accessed	Cache state (after load is complete)		
0x0	0x0 ●●●●		"cold miss"
0x4	0x0 ●●●●		hit
0x8	0x0 ●●●●		hit
0xc	0x0 ●●●●		hit
0x10	0x0 ●●●●	0x10 ●●●●	cold miss
0x14	0x0 ●●●●	0x10 ●●●●	hit
0x18	0x0 ●●●●	0x10 ●●●●	hit
0x1c	0x0 ●●●●	0x10 ●●●●	hit
0x20	0x20 ●●●●	0x10 ●●●●	cold miss (evict 0x0)
0x24	0x20 ●●●●	0x10 ●●●●	hit
0x28	0x20 ●●●●	0x10 ●●●●	hit
0x2c	0x20 ●●●●	0x10 ●●●●	hit
0x30	0x20 ●●●●	0x30 ●●●●	cold miss (evict 0x10)
0x34	0x20 ●●●●	0x30 ●●●●	hit
0x38	0x20 ●●●●	0x30 ●●●●	hit
0x3c	0x20 ●●●●	0x30 ●●●●	hit
0x40	0x40 ●●●●	0x30 ●●●●	cold miss (evict 0x20)

Data access in grid solver: row-major traversal



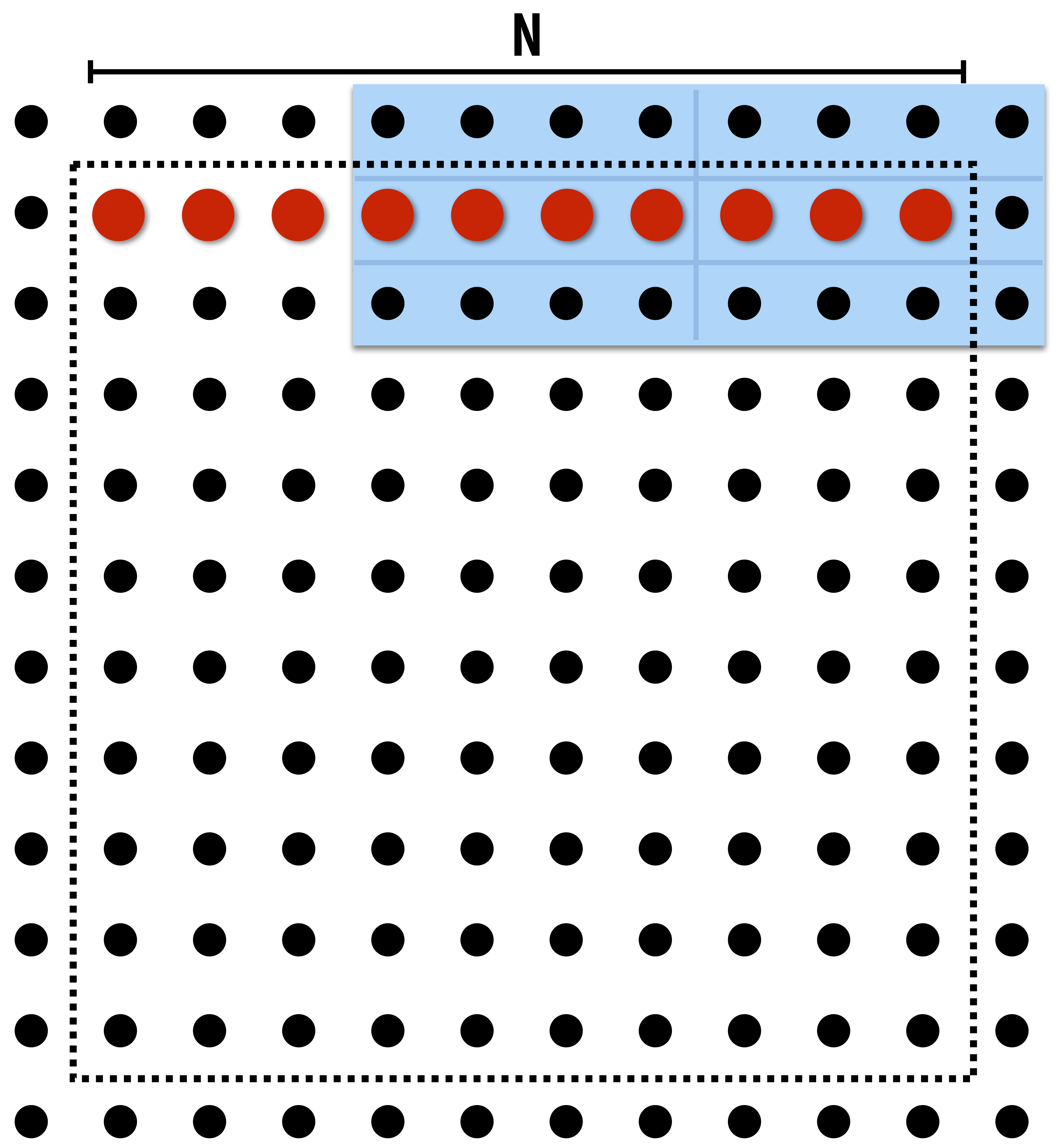
Assume row-major grid layout.

Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

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

Data access in grid solver: row-major traversal



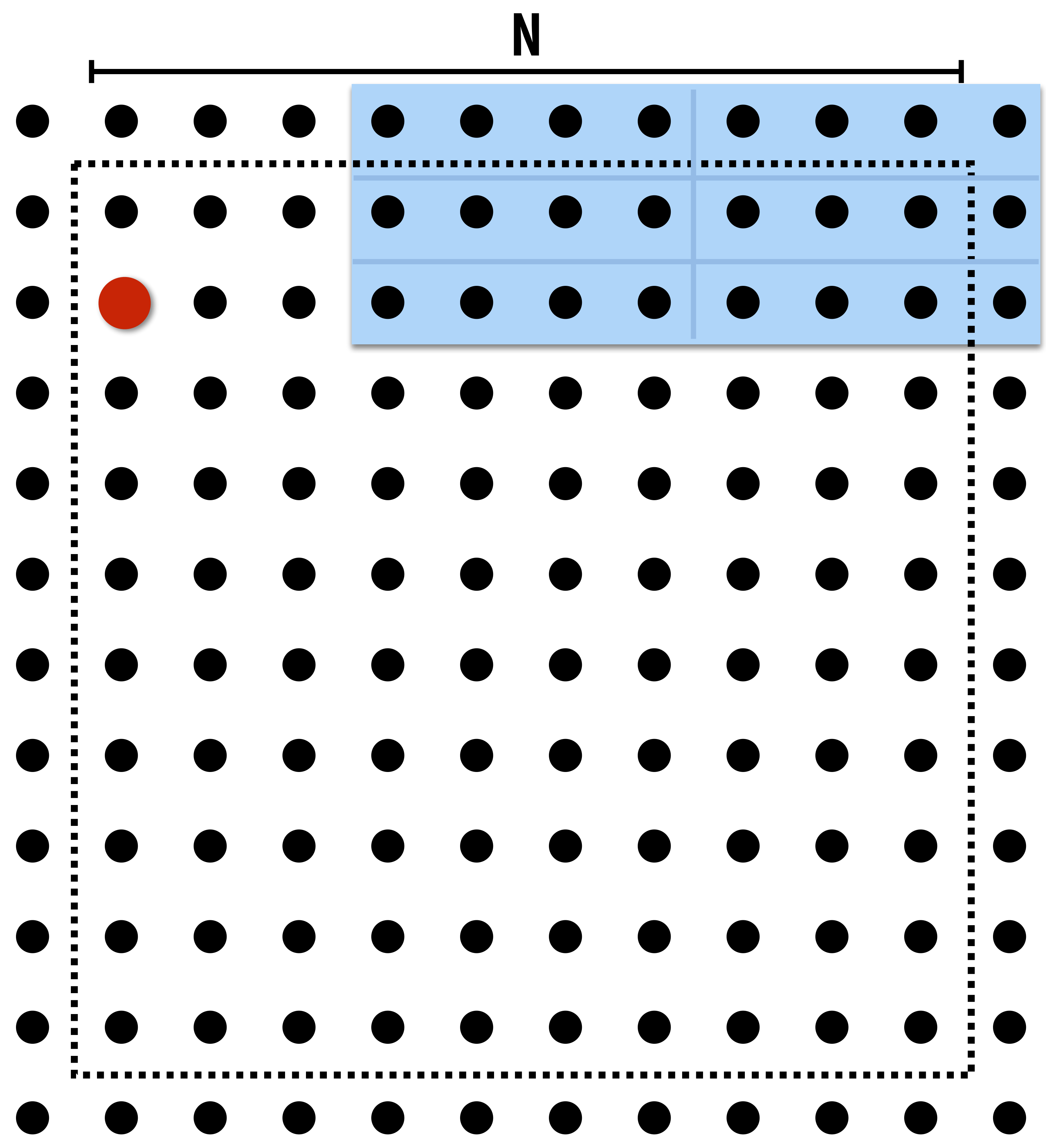
Assume row-major grid layout.

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 (0,1) 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 of output.

Artifactual communication examples

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

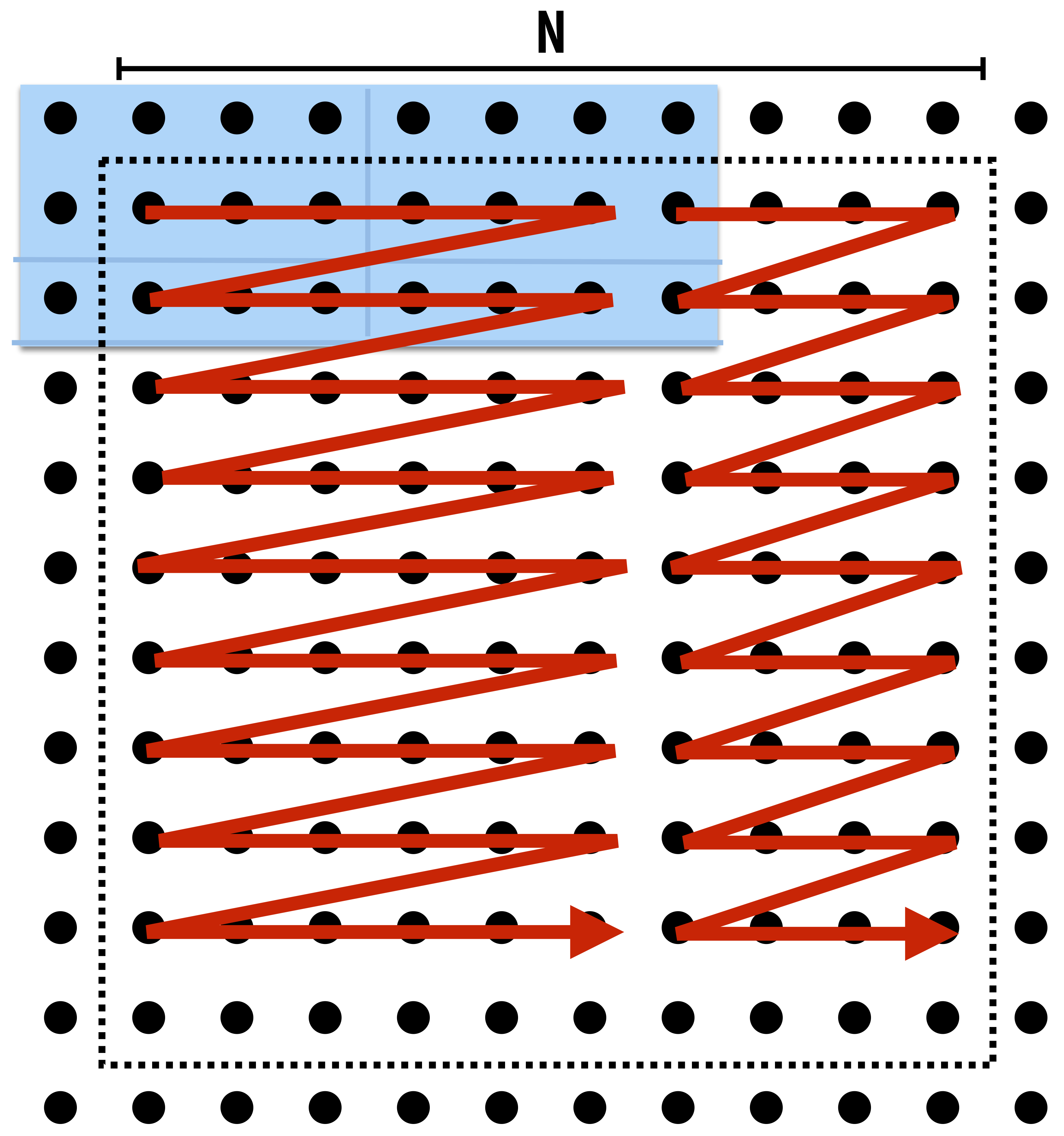
- **System operation might result in unnecessary communication:**
  - **Program stores 16 consecutive 4-byte float values, and as a result the entire 64-byte cache line is loaded from memory, entirely overwritten, then subsequently stored to memory (2x overhead… load was unnecessary)**

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

Techniques for reducing communication

Improving temporal locality by changing grid traversal order

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



Assume row-major grid layout.

Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

"Blocked" iteration order

(diagram shows state of cache after finishing work from first row of first block)

Now load two cache lines for every six elements of output

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 numPy in Python)
Code on bottom performs much better. Why?

Optimization: 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

Contention

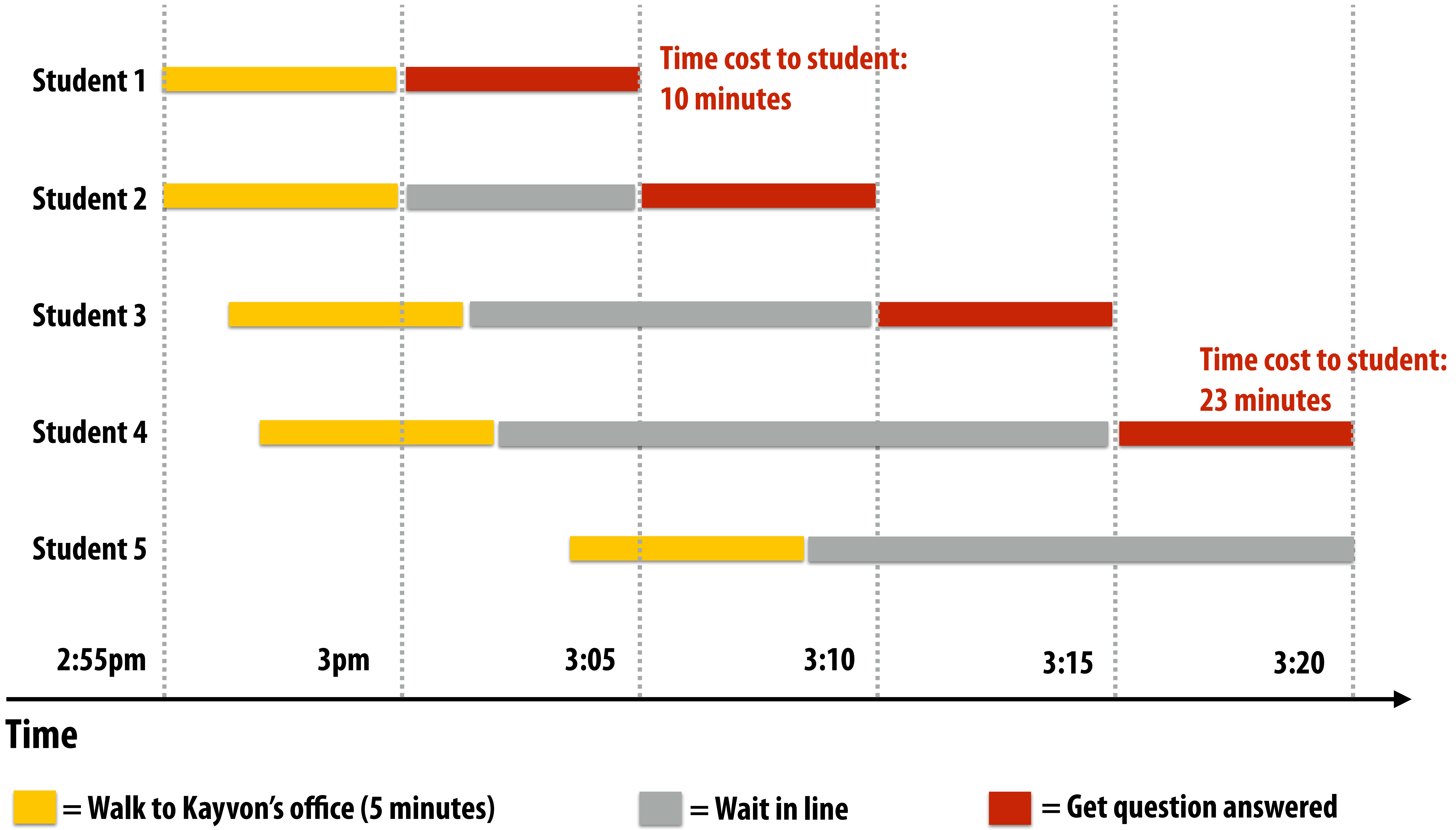
Example: office hours from 3-3:20pm (no appointments)

- **Operation to perform: Professor Kayvon helps a student with a question**

- **Execution resource: Professor Kayvon**

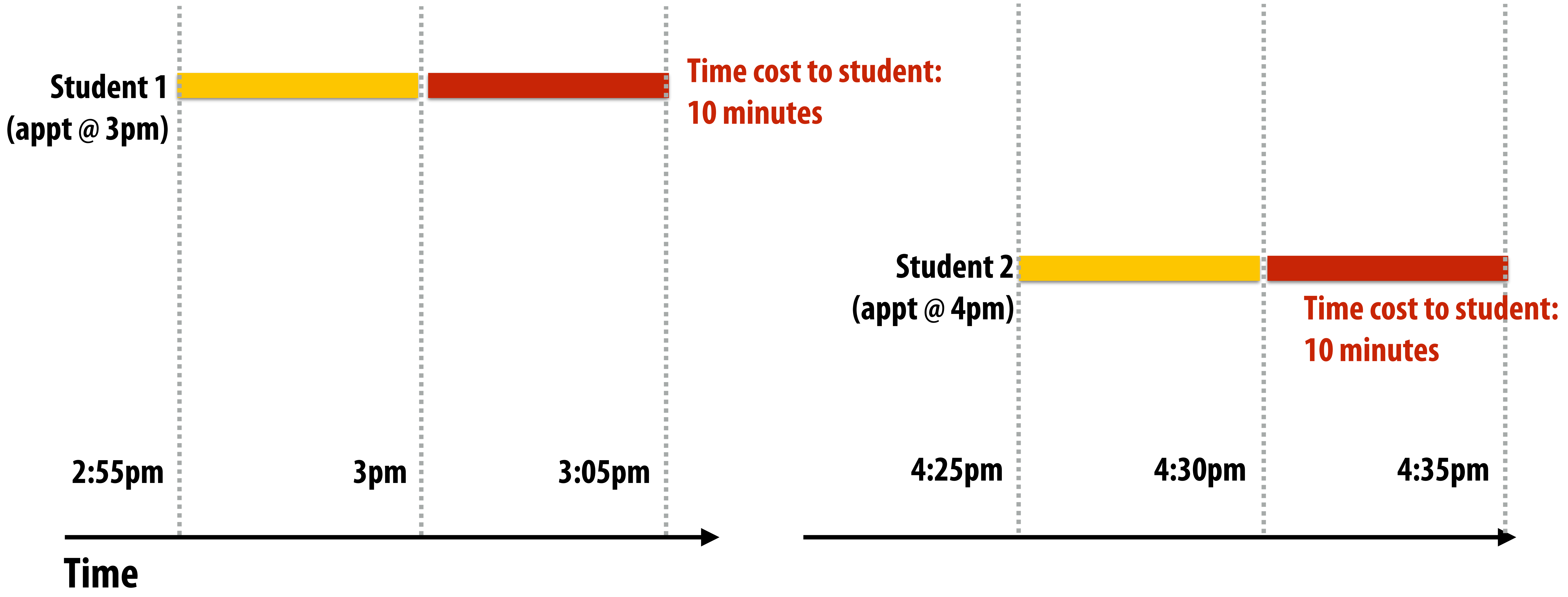
- **Steps in operation:**
  1. **Student walks from Bytes Cafe to Kayvon's office (5 minutes)**
  2. **Student waits in line (if necessary)**
  3. **Student gets question answered with insightful answer (5 minutes)**

Example: office hours from 3-3:20pm (no appointments)



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

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

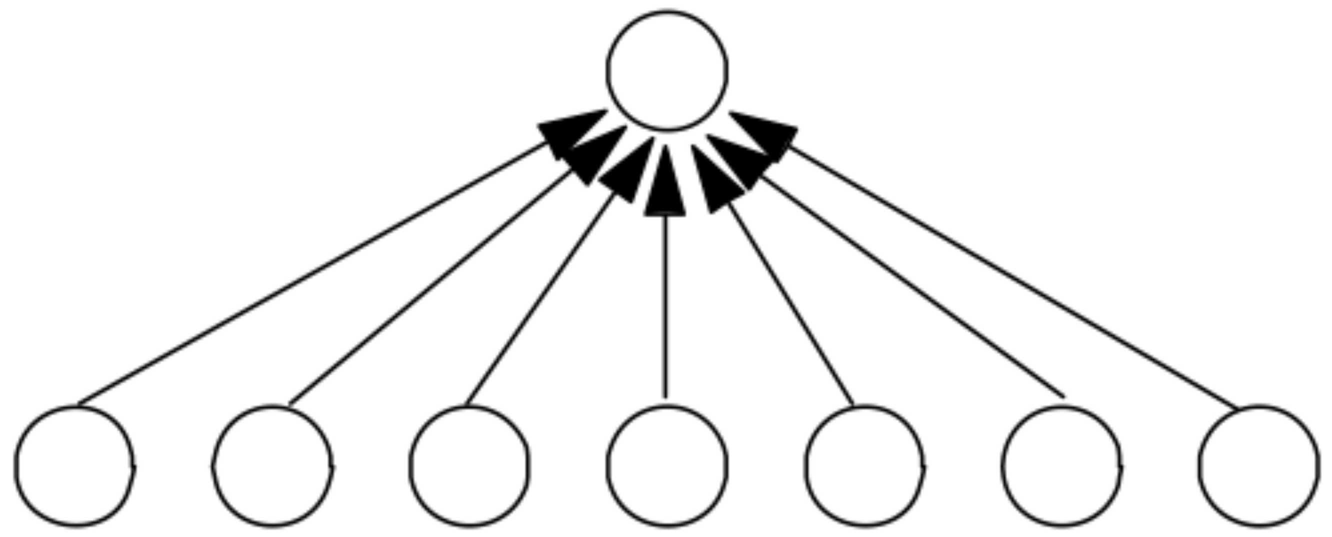


Contention

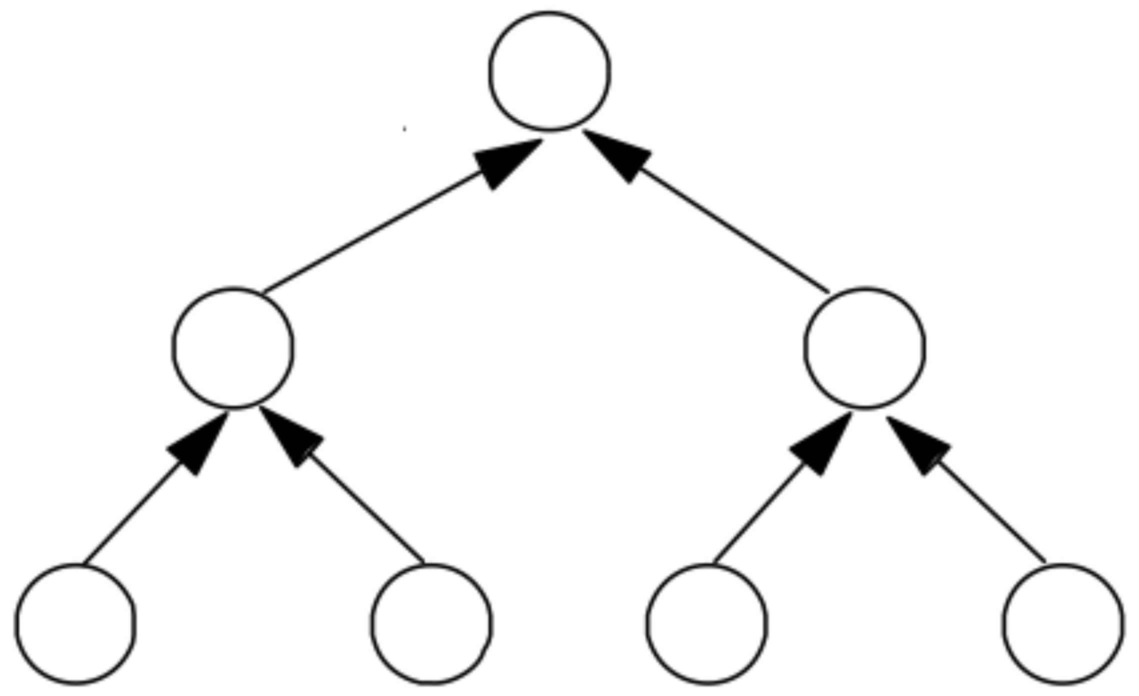
- A resource can perform operations at a given throughput (number of transactions per unit time)
  - Memory, communication links, servers, TA's at office hours, etc.

- 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 reduce contention

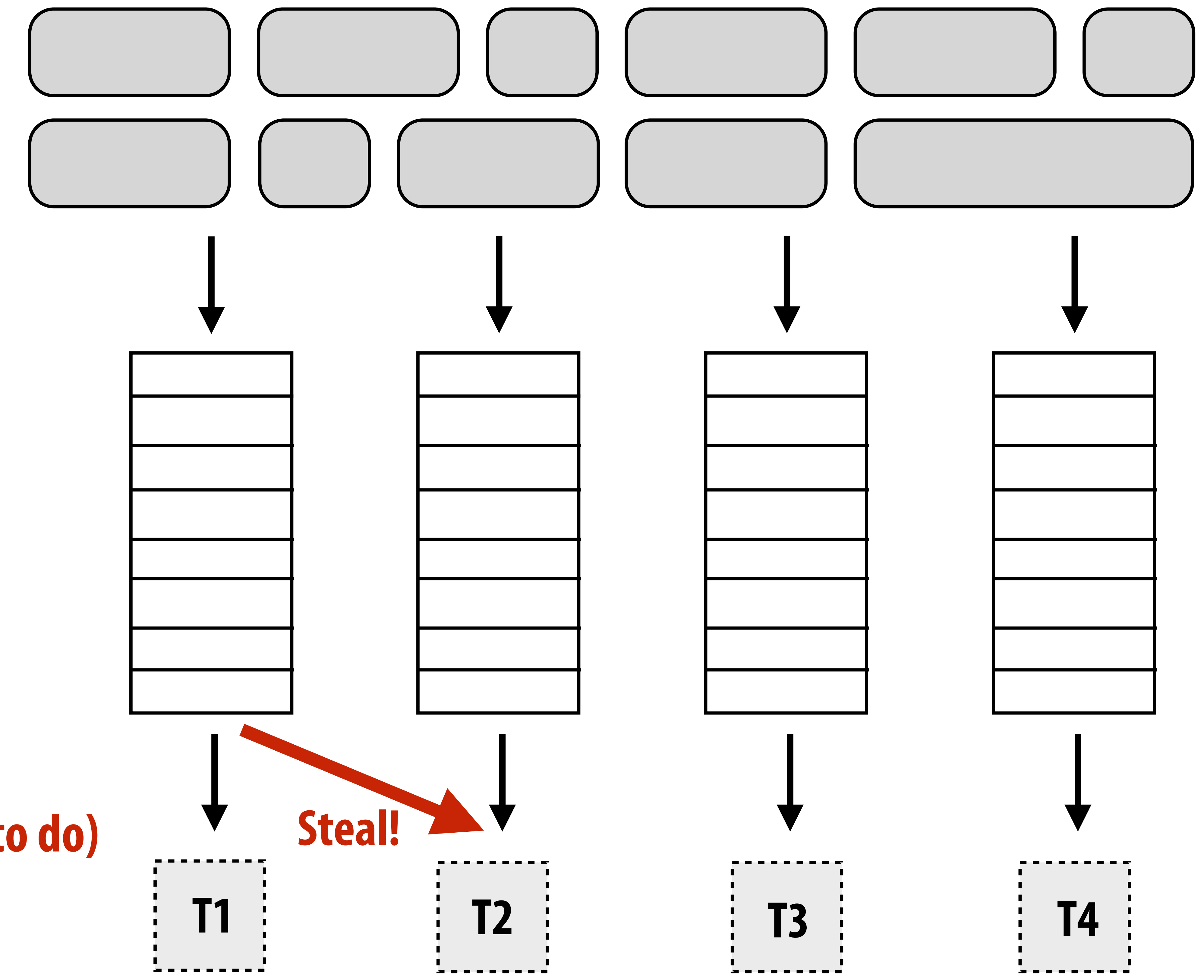
(contention in access to single shared work queue)

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
(no contention when all processors have work to do)

When local work queue is empty...
STEAL work from random work queue
(synchronization okay since processor would have sat idle anyway)



Summary: 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 latency of communication
  - Application writer: restructure code to exploit locality
  - Hardware 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)

Here are some tricks for understanding the performance of parallel software

Remember:
Always, always, always try the simplest parallel solution first, then measure performance to see where you stand.

A useful performance analysis strategy

- Determine if your performance is limited by computation, memory bandwidth (or memory latency), or synchronization?

- Try and establish "high watermarks"
  - What's the best you can do in practice?
  - How close is your implementation to a best-case scenario?

Roofline model

- Use microbenchmarks to compute peak performance of a machine as a function of arithmetic intensity of application
- Then compare application's performance to known peak values

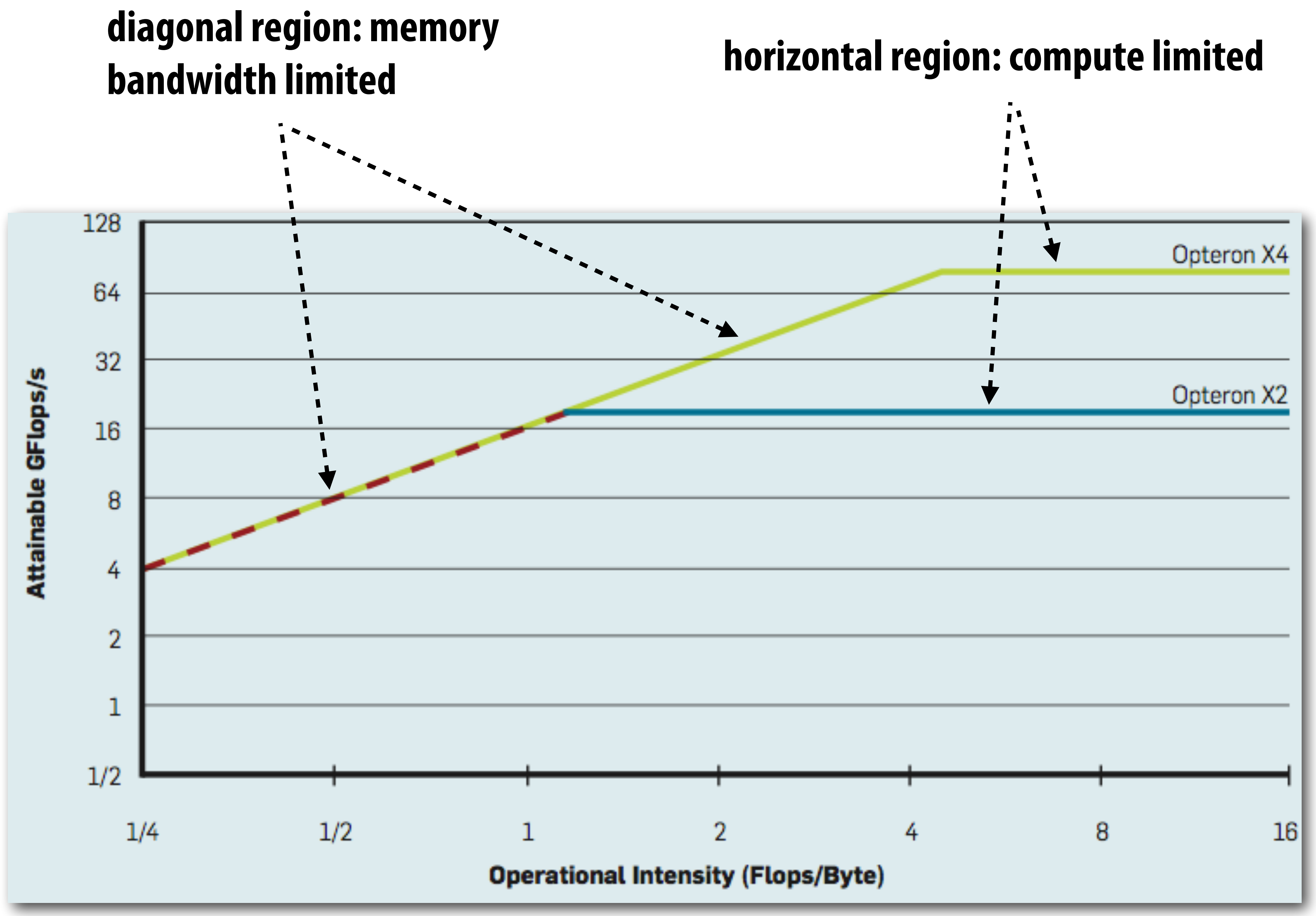


Figure credit: Williams et al. 2009

Roofline model: optimization regions

Use various levels of optimization in benchmarks (e.g., best performance with and without using SIMD instructions)

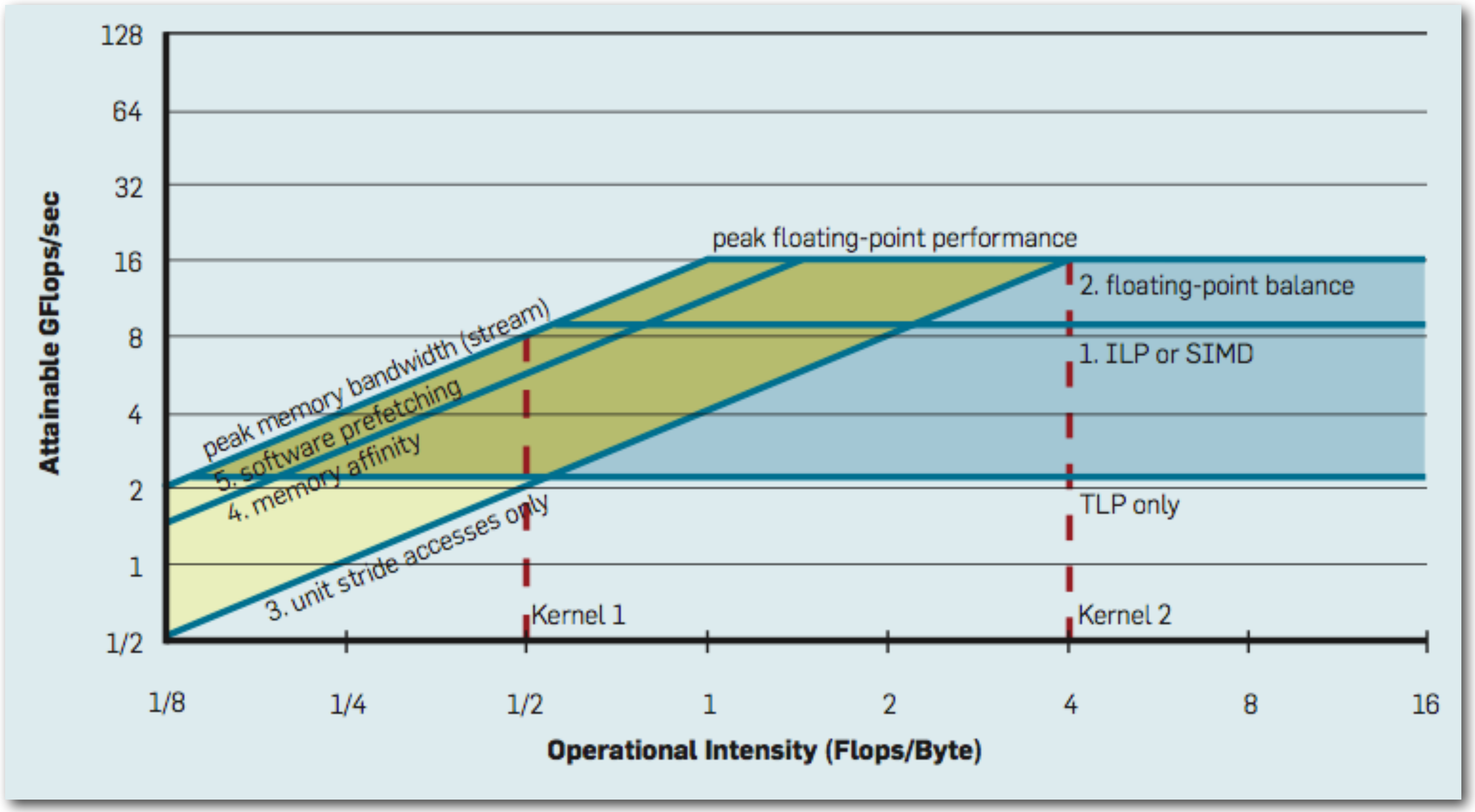


Figure credit: Williams et al. 2009

Establishing high watermarks *

Add "math" (non-memory instructions)

Does execution time increase linearly with operation count as math is added?
(If so, this is evidence that code is instruction-rate limited)

Remove almost all math, but load same data

How much does execution-time decrease? If not much, suspect memory bottleneck

Change all array accesses to A[0]

How much faster does your code get?
(This establishes an upper bound on benefit of improving locality of data access)

Remove all atomic operations or locks

How much faster does your code get? (provided it still does approximately the same amount of work)
(This establishes an upper bound on benefit of reducing sync overhead.)

* Computation, memory access, and synchronization are almost never perfectly overlapped. As a result, overall performance will rarely be dictated entirely by compute or by bandwidth or by sync. Even so, the sensitivity of performance change to the above program modifications can be a good indication of dominant costs

Use profilers/performance monitoring tools

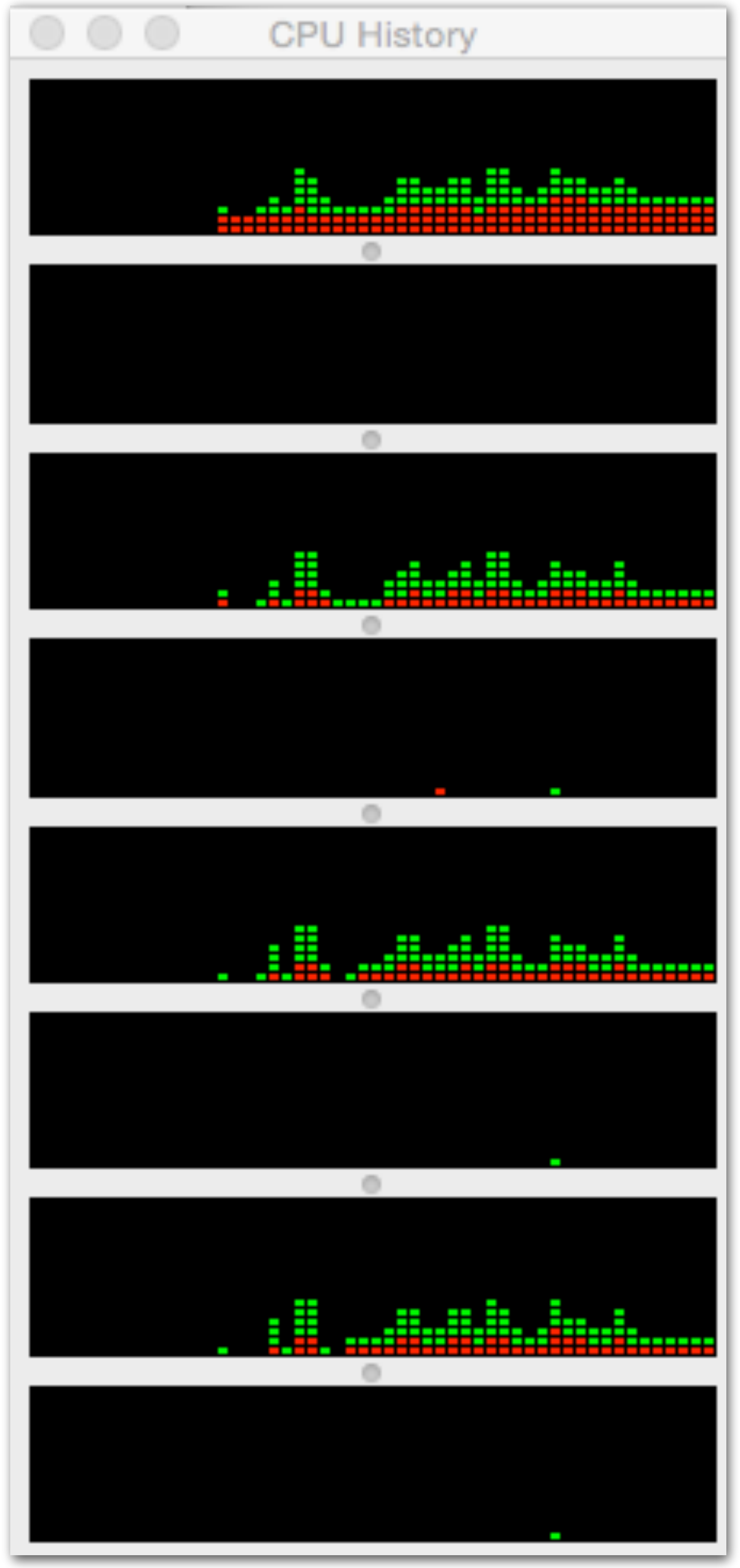
- Image at left is "CPU usage" from activity monitor in OS X while browsing the web in Chrome (from a laptop with a quad-core Core i7 CPU)
  - Graph plots percentage of time OS has scheduled a process thread onto a processor execution context
  - Not very helpful for optimizing performance

- All modern processors have low-level event "performance counters"
  - Registers that count important details such as: instructions completed, clock ticks, L2/L3 cache hits/misses, bytes read from memory controller, etc.

- Example: Intel's Performance Counter Monitor Tool provides a C++ API for accessing these registers.

```
PCM *m = PCM::getInstance();
SystemCounterState begin = getSystemCounterState();

// code to analyze goes here

SystemCounterState end = getSystemCounterState();

printf("Instructions per clock: %f\n", getIPC(begin, end));
printf("L3 cache hit ratio: %f\n", getL3CacheHitRatio(begin, end));
printf("Bytes read: %d\n", getBytesReadFromMC(begin, end));
```

- Also see Intel VTune, PAPI, oprofile, etc.



Bonus slides:
Understanding problem size issues can very helpful when assessing program performance

You are hired by [insert your favorite chip company here].

You walk in on day one, and your boss says
"All of our senior architects have decided to take the year off. Your job is to lead the design of our next parallel processor."

What questions might you ask?

Your boss selects the application that matters most to the company
"I want you to demonstrate good performance on this application."

How do you know if you have a good design?

- **Absolute performance?**
  - Often measured as wall clock time
  - Another example: operations per second
- **Speedup: performance improvement due to parallelism?**
  - Execution time of sequential program / execution time on P processors
  - Operations per second on P processors / operations per second of sequential program
- **Efficiency?**
  - Performance per unit resource
  - e.g., operations per second per chip area, per dollar, per watt

Measuring scaling

- Consider the grid solver example from last week's class
  - We changed the algorithm to allow for parallelism
  - The new algorithm might converge more slowly, requiring more iterations of the solver

- Should speedup be measured against the performance of a parallel version of a program running on one processor, or the best sequential program?

Common pitfall: compare parallel program speedup to parallel algorithm running on one core (easier to make yourself look good)

Speedup of solver application: 258 x 258 grid

Execution on 32 processor SGI Origin 2000

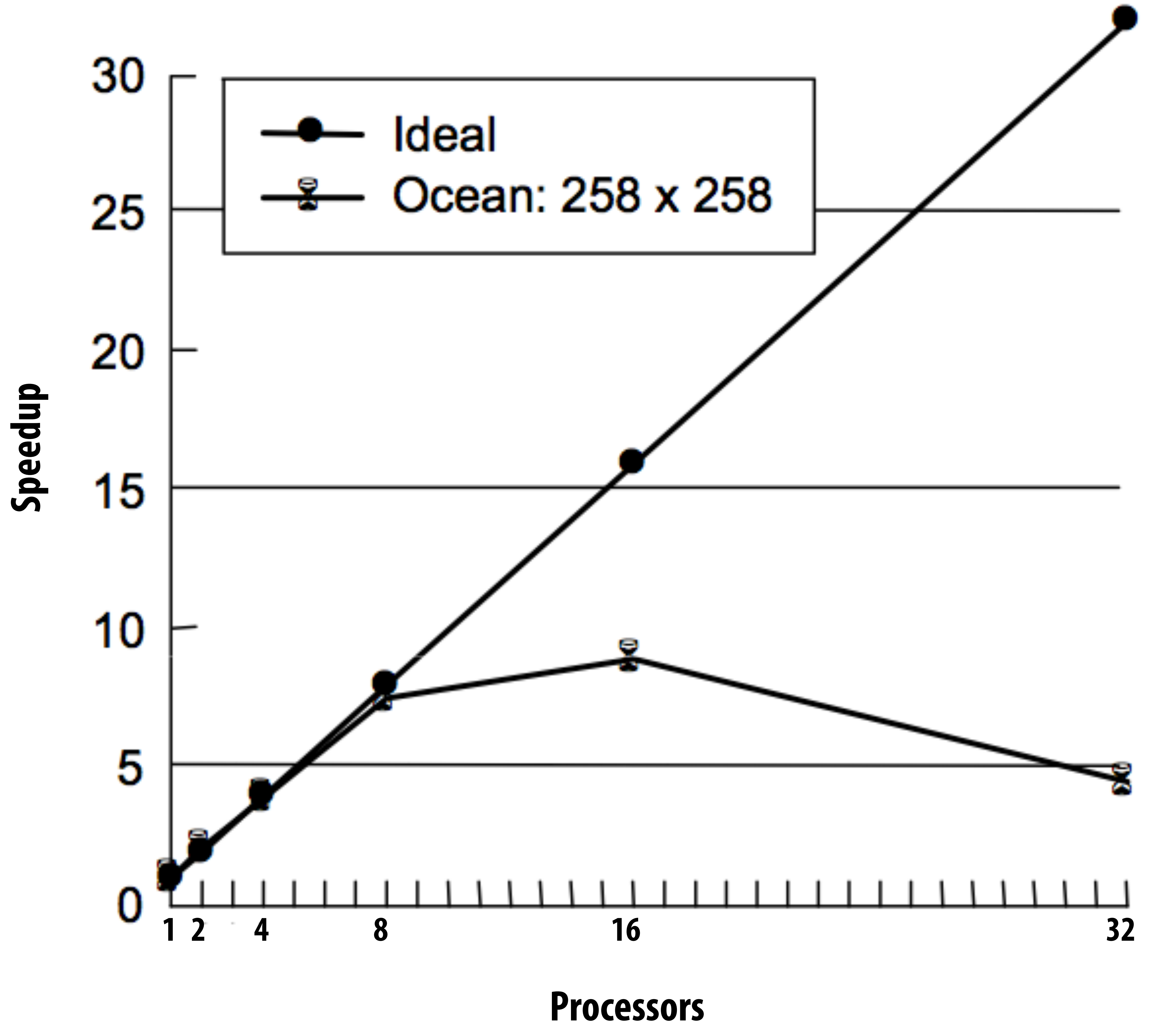
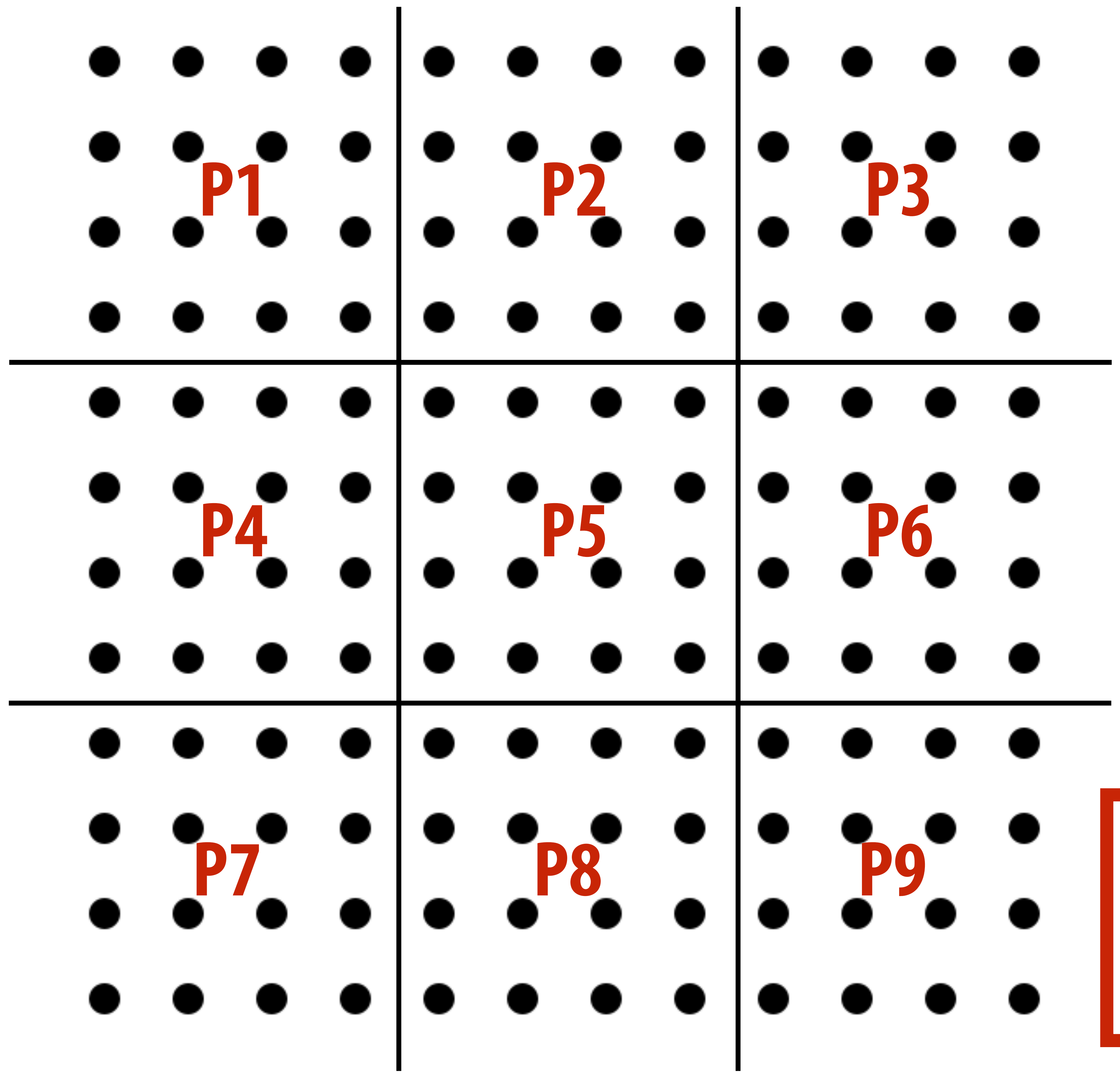


Figure credit: Culler, Singh, and Gupta

Remember: work assignment in solver

2D blocked assignment: N x N grid



N^2 elements

P processors

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

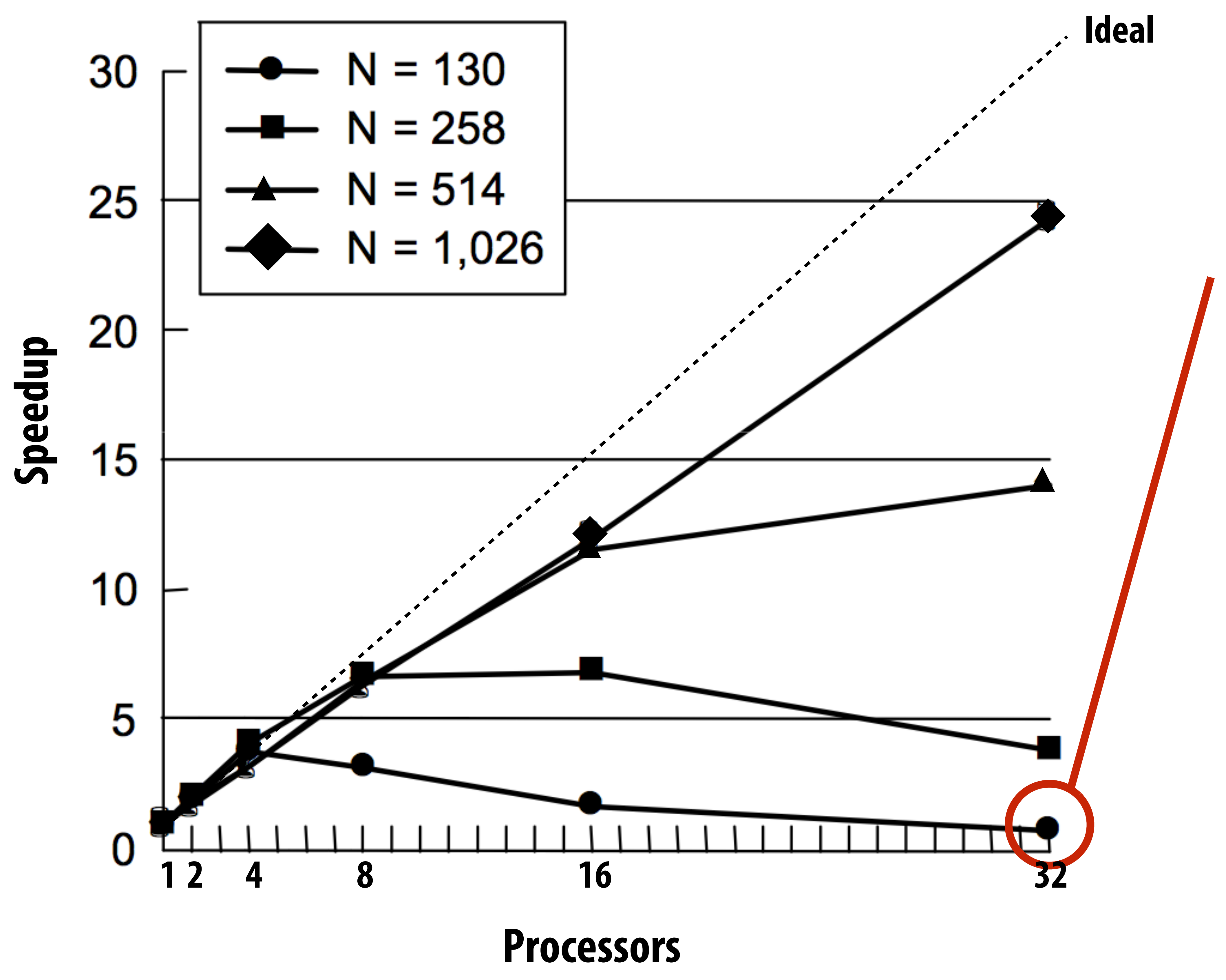
elements communicated: (per processor) $\propto \frac{N}{\sqrt{P}}$

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

Small N (or large P) yields low arithmetic intensity!

Pitfalls of fixed problem size speedup analysis

Solver execution on 32 processor SGI Origin 2000



No benefit! (slight slowdown)

Problem size is just too small for the machine (large communication-to-computation ratio)

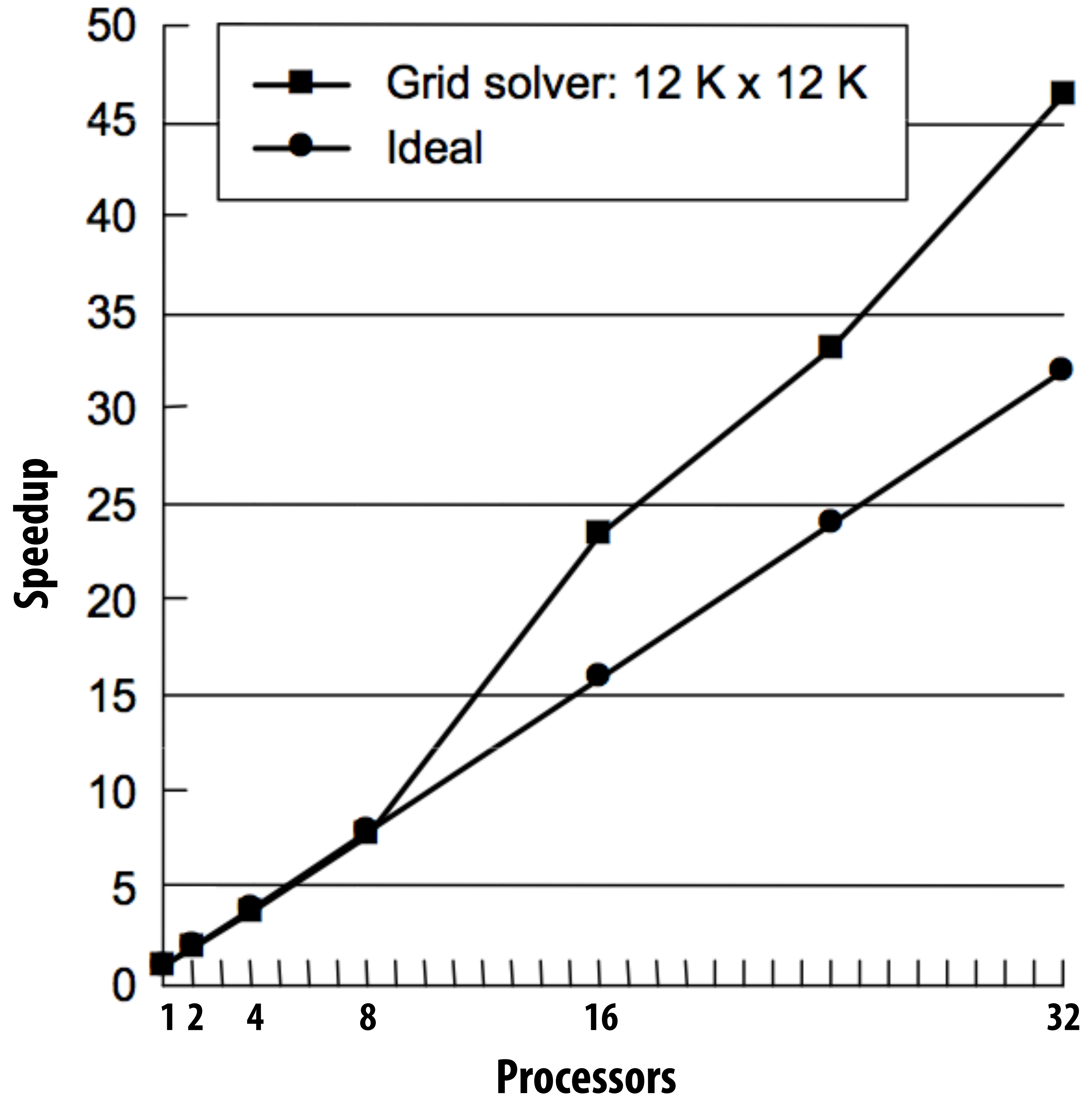
Scaling the performance of small problem may not be all that important anyway (it might already execute fast enough on a single core)

258 x 258 grid on 32 processors:	~ 310 grid cells per processor
1K x 1K grid on 32 processors:	~ 32K grid cells per processor

Figure credit: Culler, Singh, and Gupta

Pitfalls of fixed problem size speedup analysis

Execution on 32 processor SGI Origin 2000



Here: super-linear speedup! with enough processors, chunk of grid assigned to each processor begins to fit in cache (key working set fits in per-processor cache)

Another example: if problem size is too large for a single machine, working set may not fit in memory: causing thrashing to disk

(this would make speedup on a bigger parallel machine with more memory look amazing!)

Figure credit: Culler, Singh, and Gupta

Understanding scaling

- **There can be complex interactions between the size of the problem to solve and the size of the parallel computer**
  - Can impact load balance, overhead, arithmetic intensity, locality of data access
  - Effects can be dramatic and application dependent

- **Evaluating a machine with a fixed problem size can be problematic**
  - Too small a problem:
    - Parallelism overheads dominate parallelism benefits (may even result in slow downs)
    - Problem size may be appropriate for small machines, but inappropriate for large ones (does not reflect realistic usage of large machine!)

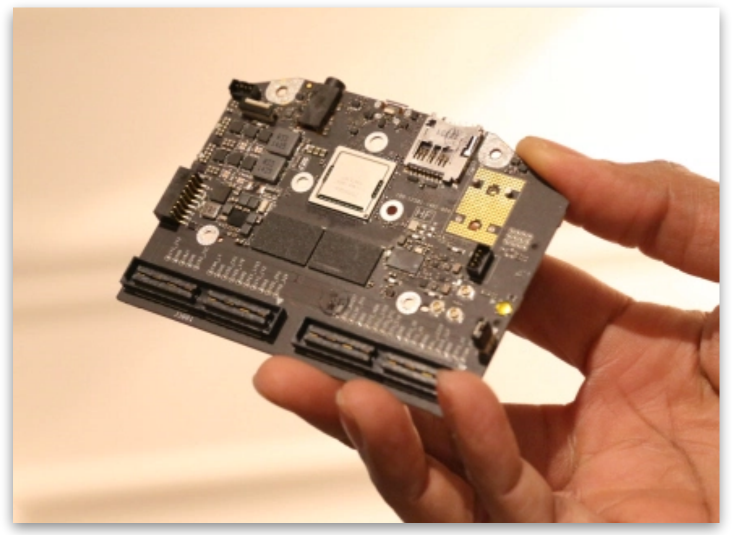
  - Too large a problem: (problem size chosen to be appropriate for large machine)
    - Key working set may not "fit" in small machine (causing thrashing to disk, or key working set exceeds cache capacity, or can't run at all)
    - When problem working set "fits" in a large machine but not small one, super-linear speedups can occur

- **Can be desirable to scale problem size as machine sizes grow**
  (buy a bigger machine to compute <u>more</u>, rather than just compute the same problem faster)

Architects also think about scaling

A common question: "Does an architecture scale?"

- Scaling up: how does architecture's performance scale with increasing core count?
  - Will design scale to the high end?
- Scaling down: how does architecture's performance scale with decreasing core count?
  - Will design scale to the low end?
- Parallel architectures are designed to work in a range of contexts
  - Same architecture used for low-end, medium-scale, and high-end systems
  - GPUs are a great example
    - Same SM core architecture, different numbers of SM cores per chip


Tegra X1: 2 SM cores (mobile SoC)

GTX 950: 6 SM cores (90 watts)

GTX 980: 16 SM cores (165 watts)

Titan X: 24 SM cores (250 watts)

Questions to ask when scaling a problem

- **Under what constraints should the problem be scaled?**
  - "Work done by program" may no longer be the quantity that is fixed
  - Fixed data set size, fixed memory usage per processor, fixed execution time, etc.?

- **How should be the problem be scaled?**
  - Problem size is often determined by more than one parameter
  - Solver example: problem defined by $(N, \varepsilon, \Delta t, T)$

total time simulated by program
(one hour of fluid flow)

grid size: (N x N)

time step size (of overall fluid simulation that uses solver)

convergence threshold of solver

Problem-constrained scaling *

- Focus: use a parallel computer to solve the same problem faster

$$\text{Speedup} = \frac{\text{time 1 processor}}{\text{time P processors}}$$

- Recall pitfalls from earlier in lecture (small problems may not be realistic workloads for large machines, big problems may not fit on small machines)

- Examples of problem-constrained scaling:
  - Almost everything we've considered parallelizing in class so far

* Problem-constrained scaling is often called "hard scaling".

Time-constrained scaling

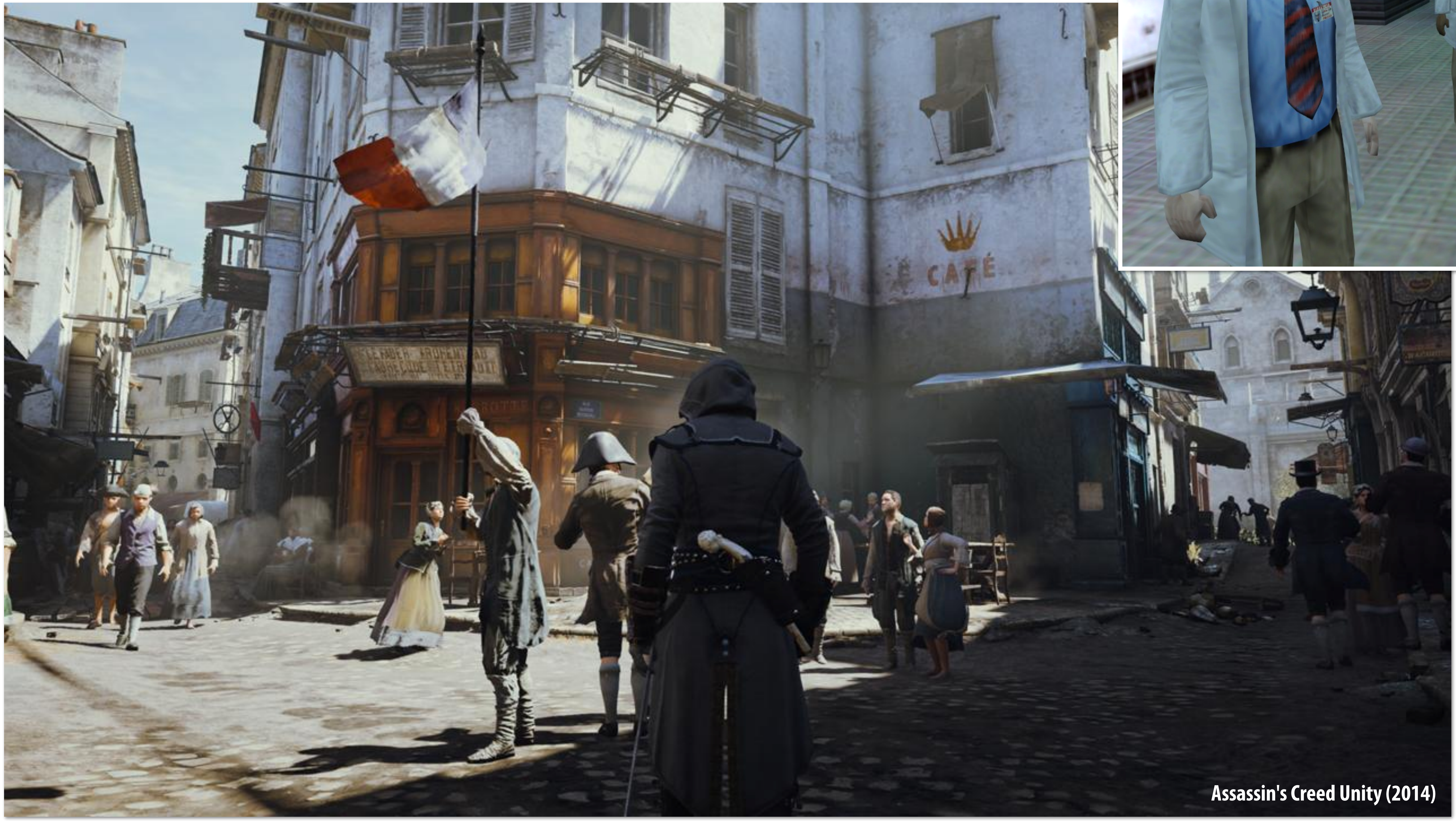
- **Focus: completing more work in a fixed amount of time**
  - Execution time kept fixed as the machine (and problem) scales

$$\text{Speedup} = \frac{\text{work done by P processors}}{\text{work done by 1 processor}}$$

- **How to measure "work"?**
  - Challenge: "work done" may not be linear function of problem inputs (e.g. matrix multiplication is $O(N^3)$ work for $O(N^2)$ sized inputs)
  - One approach: "work done" is defined by execution time of same computation on a single processor (but consider effects of thrashing if problem too big)
  - Ideally, a measure of work is:
    - Simple to understand
    - Scales linearly with sequential run time (so ideal speedup remains linear in P)

Time-constrained scaling example

Real-time 3D graphics: more compute power allows for rendering of much more complex scene
Problem size metrics: number of polygons, texels sampled, shader length, etc.


Assassin's Creed Unity (2014)

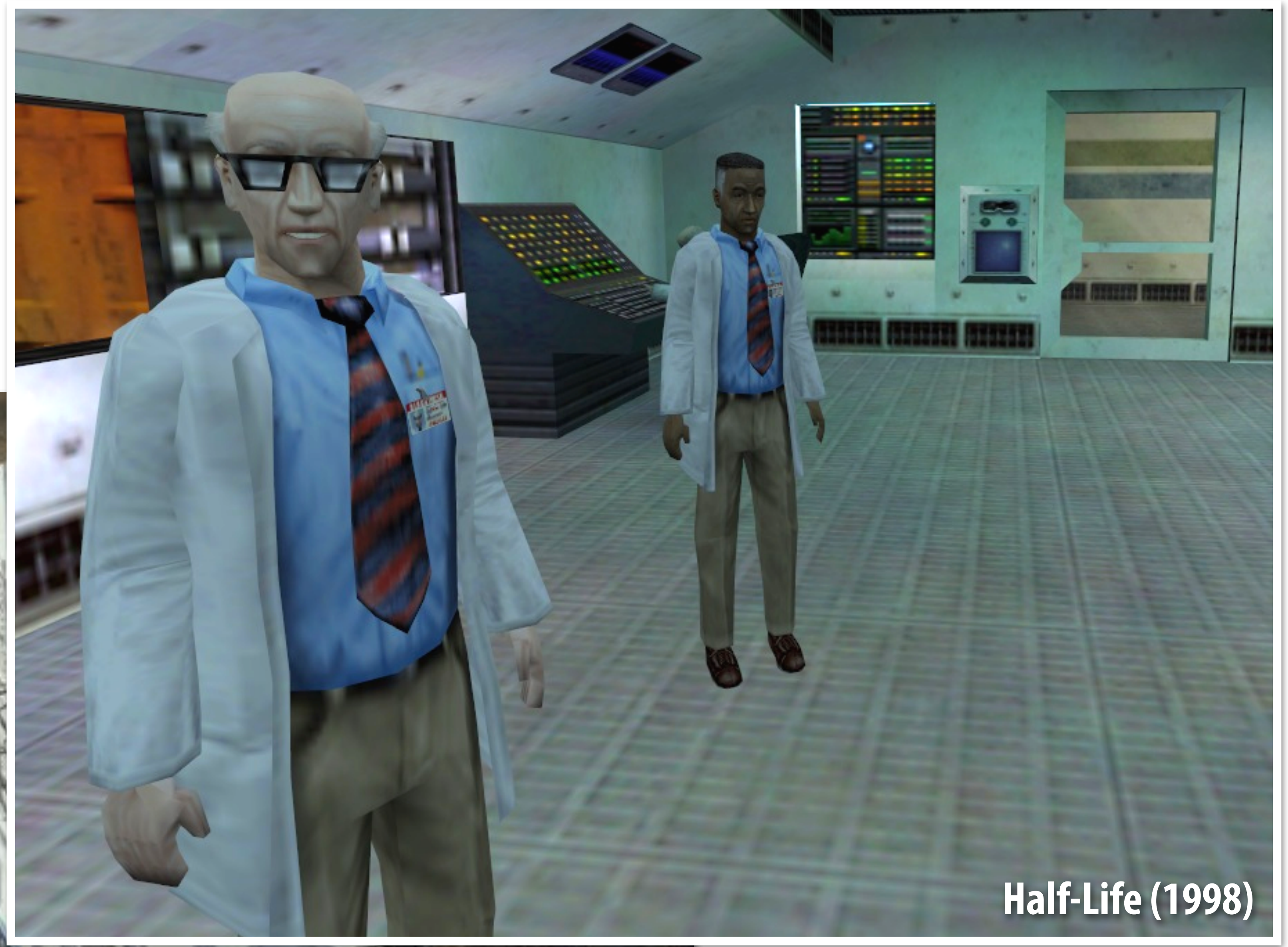
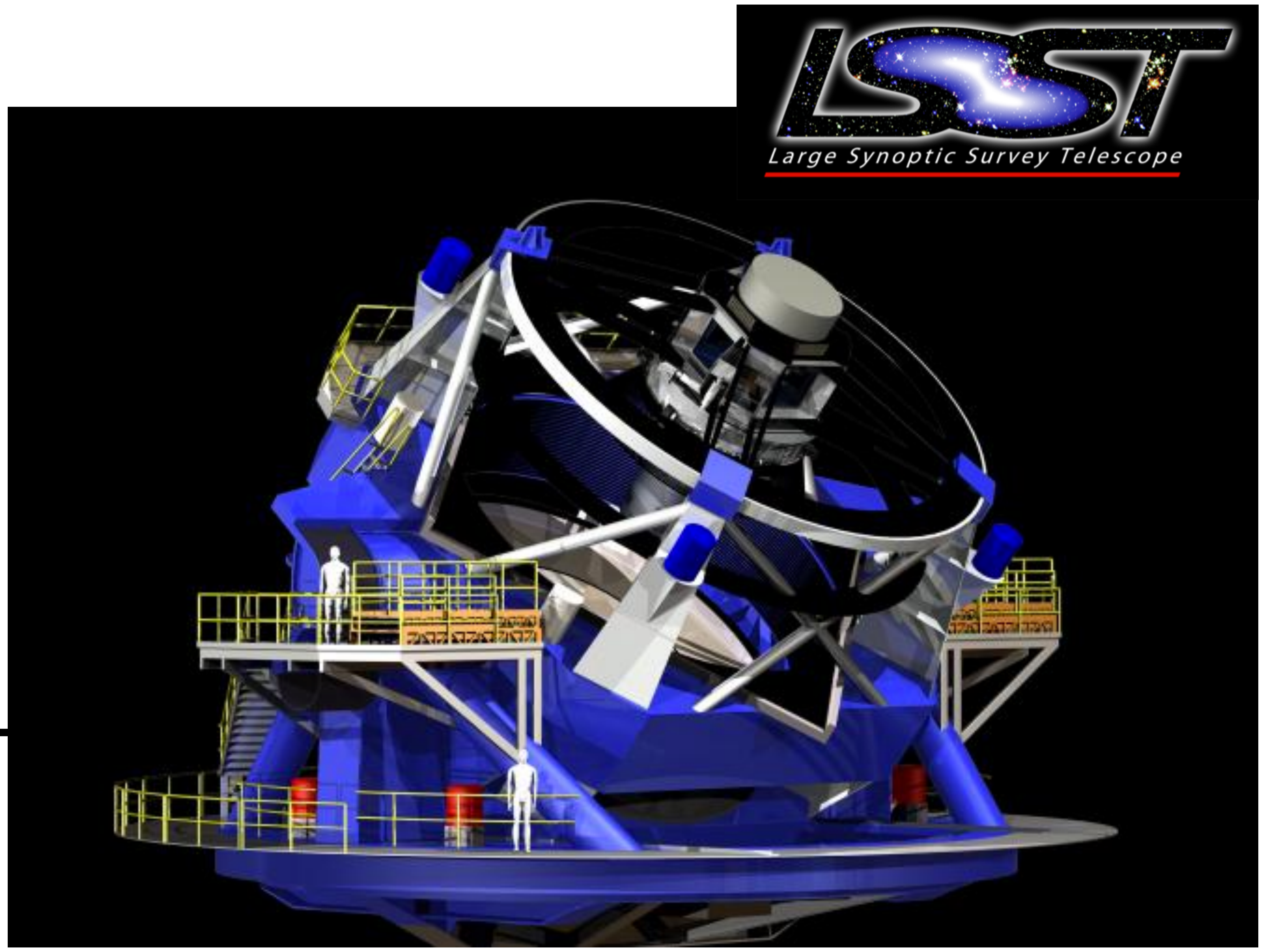

Half-Life (1998)

Image credits:
http://www.gamespot.com/forums/system-wars-314159282/assassin-s-creed-unity-best-graphics-of-2014-31696528/
http://www.game-weavers.com/?page_id=490

Time-constrained scaling example

Large Synoptic Survey Telescope (LSST)

- Estimated completion in 2019
- Acquire high-resolution survey of sky (3-gigapixel image every 15 seconds, every night for many years)



LSST will be located on top of Cerro Pachón Mountain, Chile

Rapid Image analysis compute platform (detect "potentially" interesting events)

Notify other observatories if potential event detected.

Increasing compute capability allows for more sophisticated detection algorithms (fewer false positives, detect broader class of events)

Image credits:
http://www.lsst.org
http://mcdonaldobservatory.org

More time-constrained scaling examples

- **Computational finance**
  - Run most sophisticated model possible in: 1 ms, 1 minute, overnight, etc.

- **Modern web sites**
  - Want to generate complex page, respond to user in X milliseconds (studies show site usage directly corresponds to page load latency)

- **Real-time computer vision for robotics**
  - Consider self-driving car: want best-quality obstacle detection in 5 ms

Memory-constrained scaling *

- Focus: run the largest problem possible without overflowing main memory **
- Memory per processor is held fixed (e.g., add more machines to cluster)
- Neither work or execution time are held constant

$$\text{Speedup} = \frac{\text{work (P processors)} \times \text{time (1 processor)}}{\text{time (P processors)} \times \text{work (1 processor)}}$$

$$= \frac{\text{work per unit time on P processors}}{\text{work per unit time on 1 processor}}$$

- Note: scaling problem size can make runtimes very large
  - Consider $O(N^3)$ matrix multiplication on $O(N^2)$ matrices

* Memory-constrained scaling is often called "weak scaling"

** Assumptions: (1) memory resources scale with processor count (2) spilling to disk is infeasible behavior (too slow)

Memory-constrained scaling examples

- One motivation to use supercomputers and large clusters is simply to be able to fit large problems in memory
- Large N-body problems
  - 2012 Supercomputing Gordon Bell Prize Winner: 1,073,741,824,000 particle N-body simulation on Japan's K-Computer
- Large-scale machine learning
  - Billions of clicks, documents, etc.
- Memcached (in memory caching system for web apps)
  - More servers = more available cache

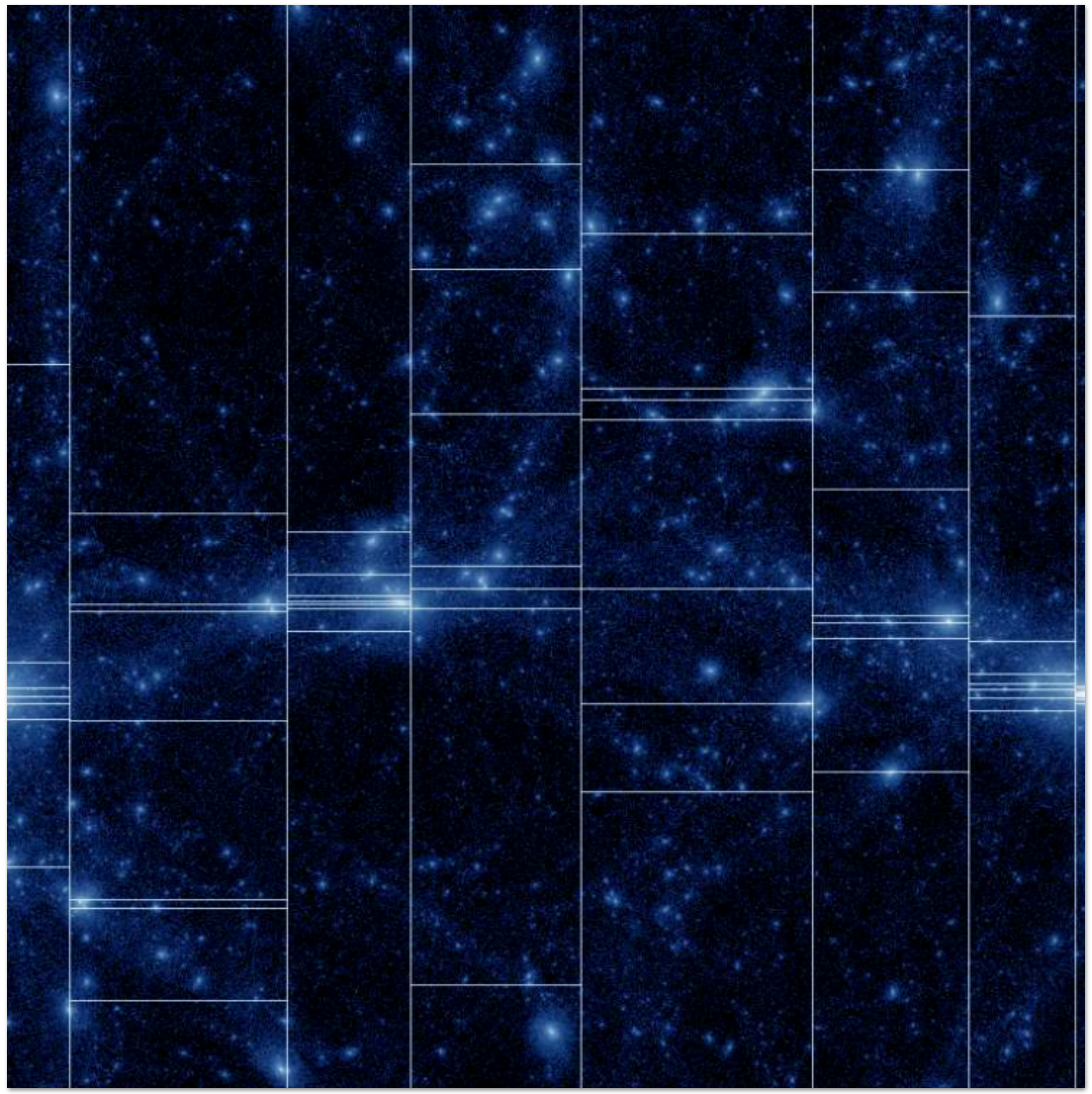

2D domain decomposition of N-body simulation

Image credit: Ishiyama et al. 2012

Scaling examples at PIXAR

- Rendering a "shot" (a sequence of frames) in a movie
  - Goal: minimize time to completion (problem constrained)
  - Assign each frame to a different machine in the cluster
- Artists working to design lighting for a scene
  - Provide interactive frame rate to artist (time constrained)
  - More performance = higher fidelity representation shown to artist in allotted time
- Physical simulation: e.g., fluid simulation
  - Parallelize simulation across multiple machines to fit simulation grid in aggregate memory of processors (memory constrained)
- Final render of images for movie
  - Scene complexity is typically bounded by memory available on farm machines
  - One barrier to exploiting additional parallelism within a machine is that required footprint often increases with number of processors

Summary of tips

- Measure, measure, measure…

- Establish high watermarks for your program
  - Are you compute, synchronization, or bandwidth bound?

- Be aware of scaling issues. Is the problem well matched for the machine?