

(how to be l33t)

Lecture 6:

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

Parallel Computing
Stanford CS149, Fall 2023

Today's topic

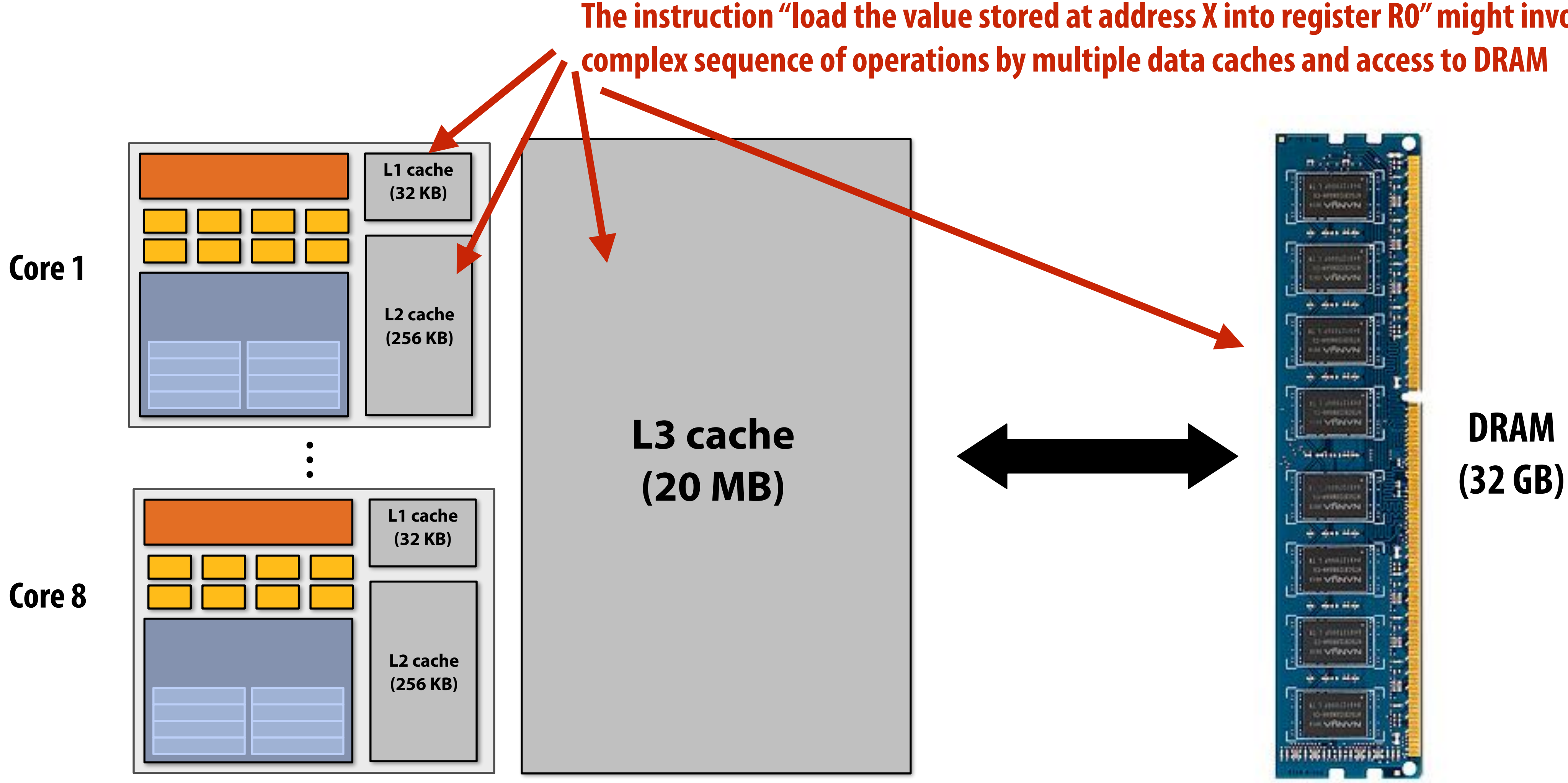
- **Techniques for reducing the costs of communication**
 - **Between processors**
 - **Between processor(s) and memory**
- **General program optimization tips**

So far in this course we've assumed all processors are connected to a memory system that provides the abstraction of a single shared address space

But the implementation of that abstraction can be quite complex.

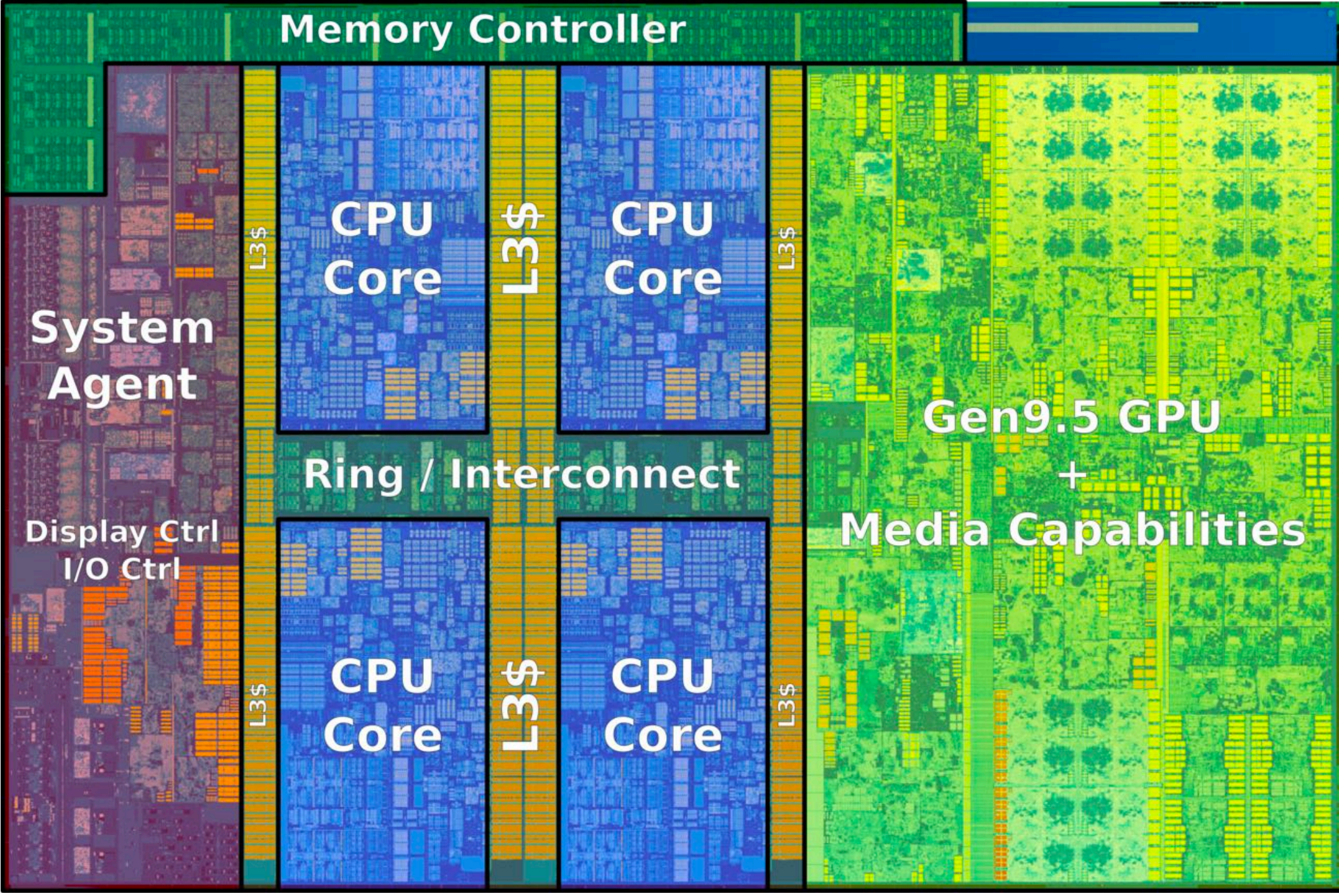
The implementation of the linear memory address space abstraction on a modern computer is complex

The instruction "load the value stored at address X into register R0" might involve a complex sequence of operations by multiple data caches and access to DRAM

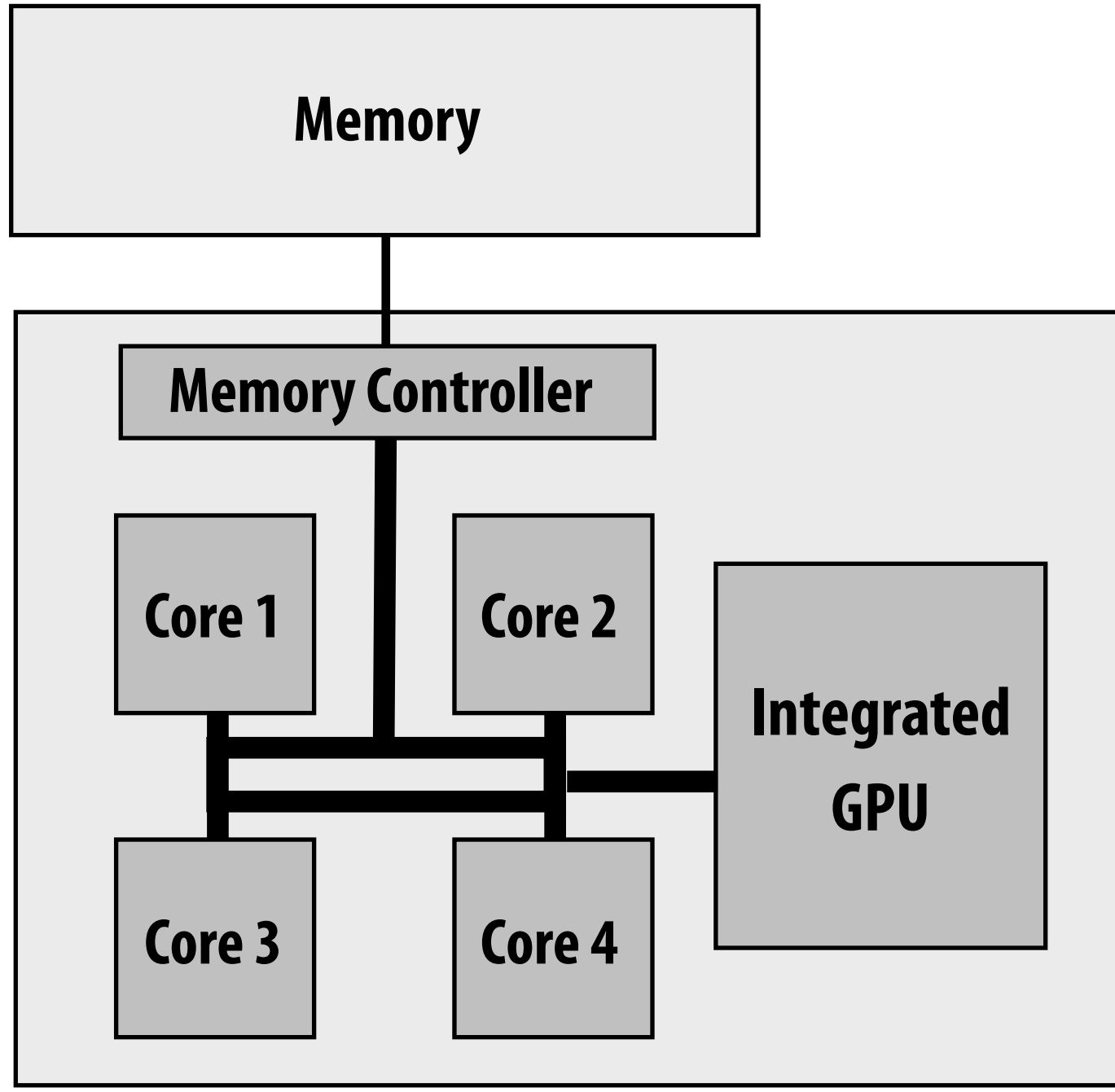


Shared address space hardware architecture

Any processor can directly reference any memory location



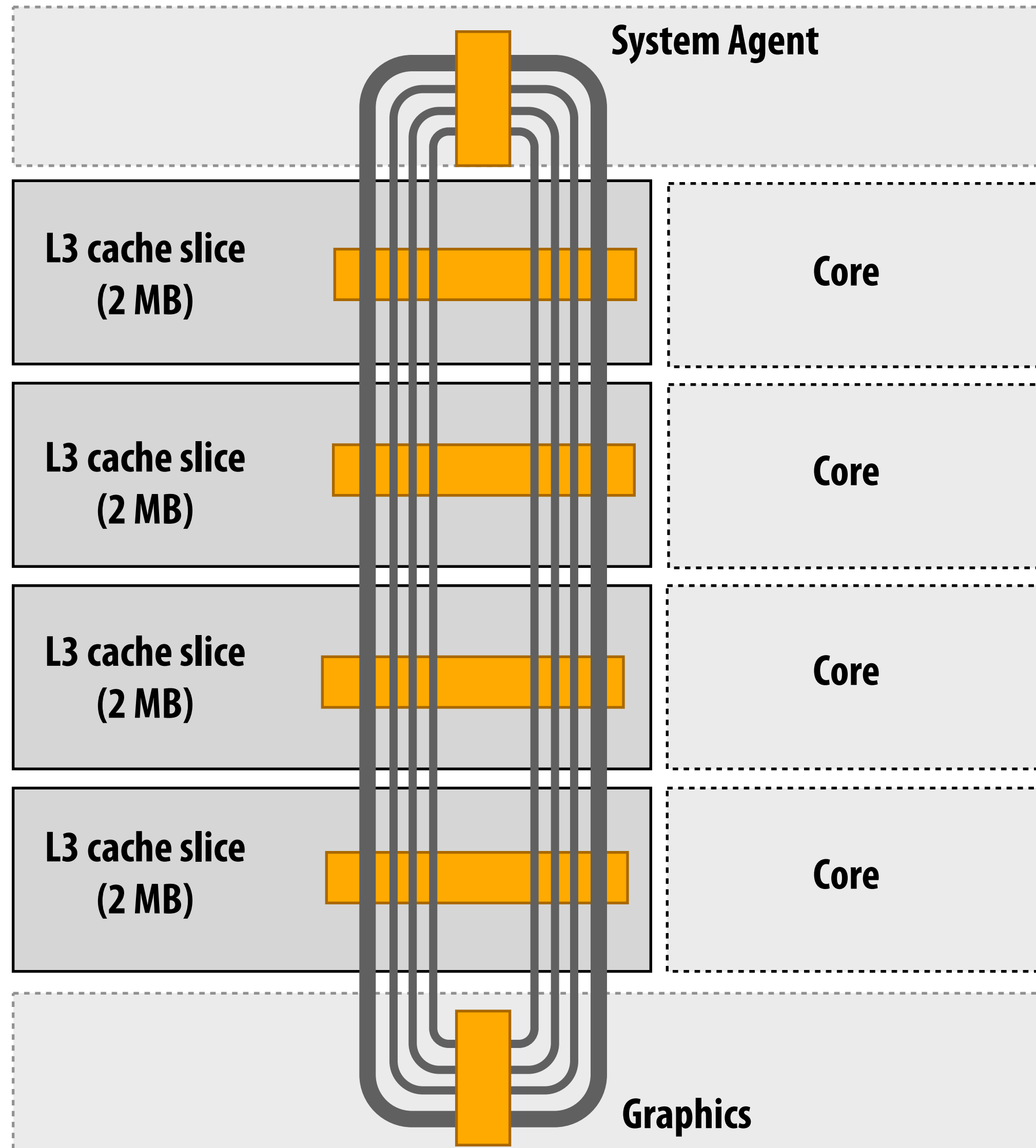
Example: Intel Core i7 processor (Kaby Lake)



Intel Core i7 (quad core)
(interconnect is a ring)

Intel's ring interconnect

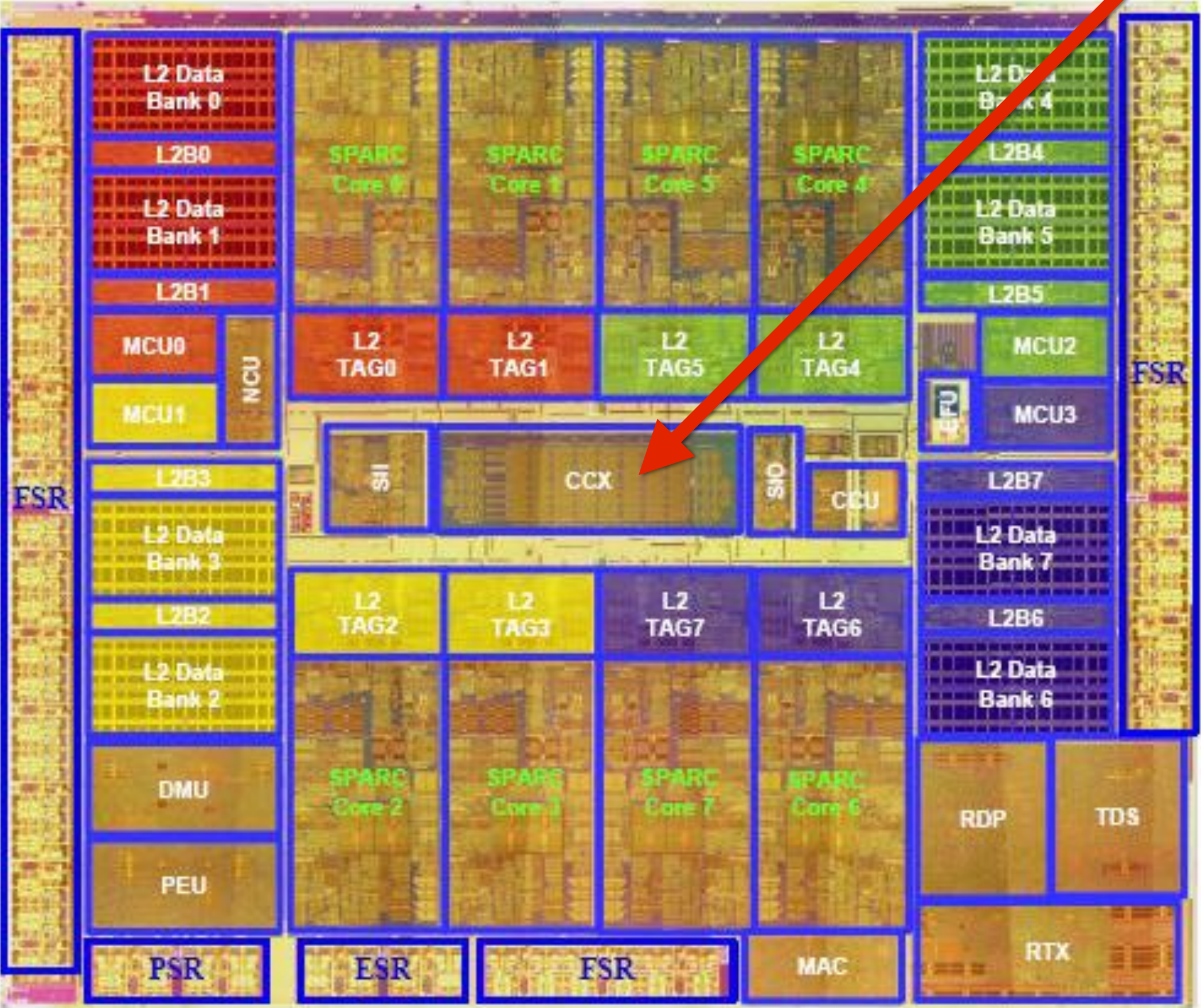
Introduced in Sandy Bridge microarchitecture



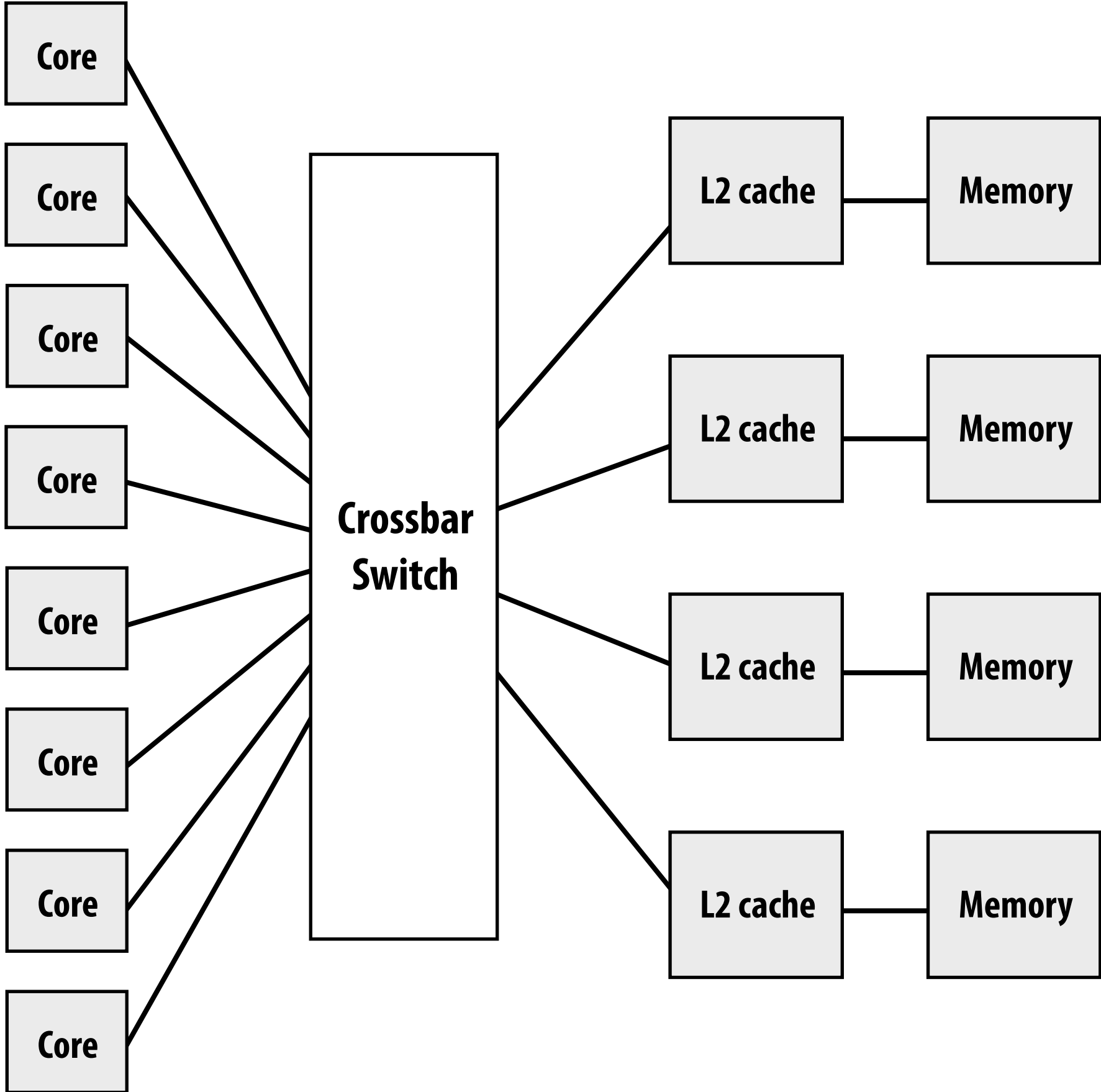
- **Four rings: for different types of messages**
 - request
 - snoop
 - ack
 - data (32 bytes)
- **Six interconnect nodes: four “slices” of L3 cache + system agent + graphics**
- **Each bank of L3 connected to ring bus twice**
- **Theoretical peak BW from cores to L3 at 3.4 GHz ~ 435 GB/sec**
 - When each core is accessing its local slice

SUN Niagara 2 (UltraSPARC T2): crossbar interconnect

Note area of crossbar (CCX):
about same area as one core on chip



Eight core processor



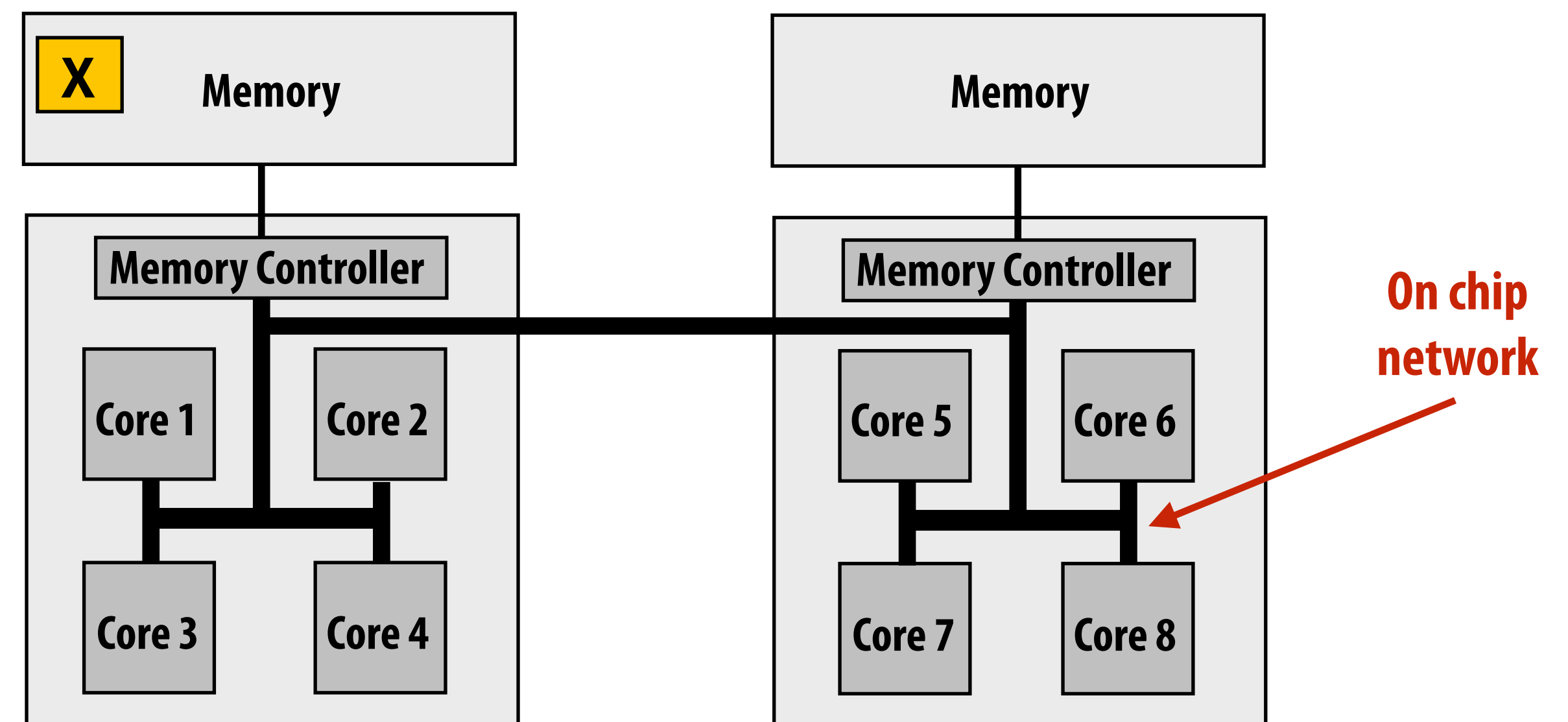
Crossbar = All cores connected directly to all others

Non-uniform memory access (NUMA)

The latency of accessing a memory location may be different from different processing cores in the system
Bandwidth from any one location may also be different to different CPU cores *



Example: modern multi-socket configuration



* In practice, you'll find NUMA behavior on a single-socket system as well (recall: different cache slices are a different distance from each core)

Summary: shared address space model

■ Communication abstraction

- Threads read/write variables in shared address space
- Threads manipulate synchronization primitives: locks, atomic ops, etc.
- Logical extension of uniprocessor programming *

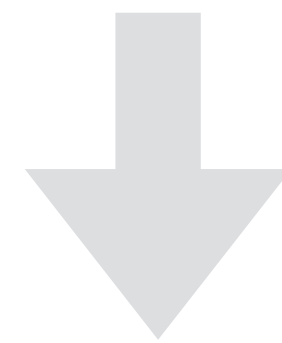
■ Requires hardware support to implement efficiently

- Any processor can load and store from any address
- Can be costly to scale to large numbers of processors
(one of the reasons why high-core count processors are expensive)

* But NUMA implementations require reasoning about locality for performance optimization

In the shared address space model, threads communicate by reading and writing to variables in the shared address space.

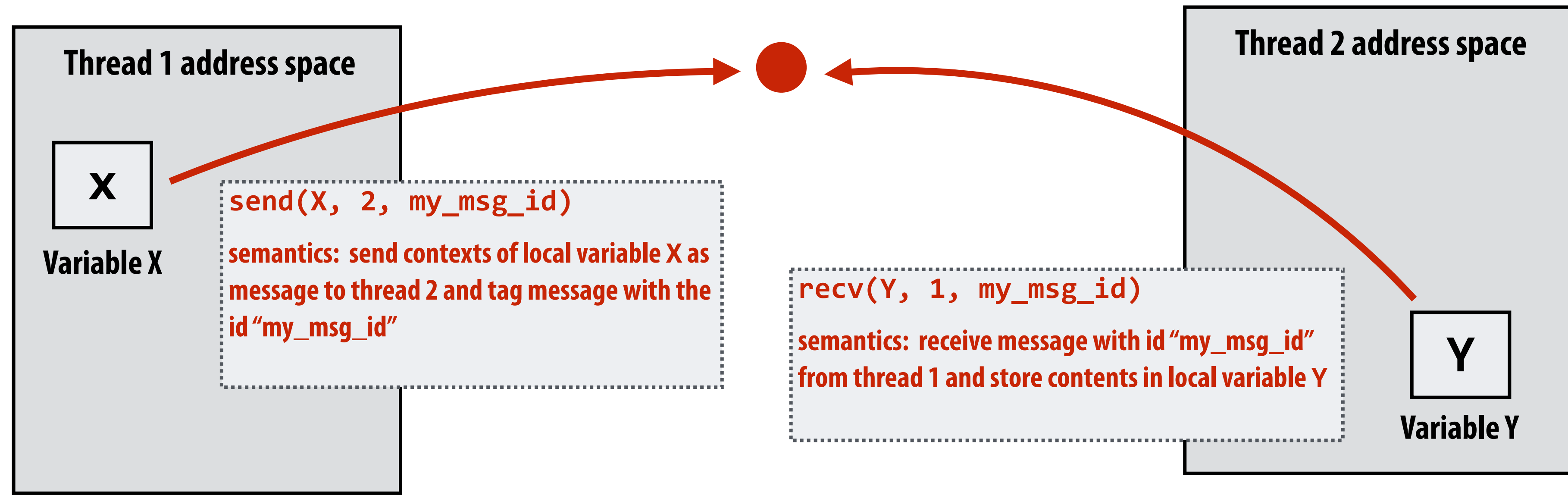
Let's consider a different abstraction that makes communication between processors more explicit.



Message passing

Message passing model (abstraction)

- 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



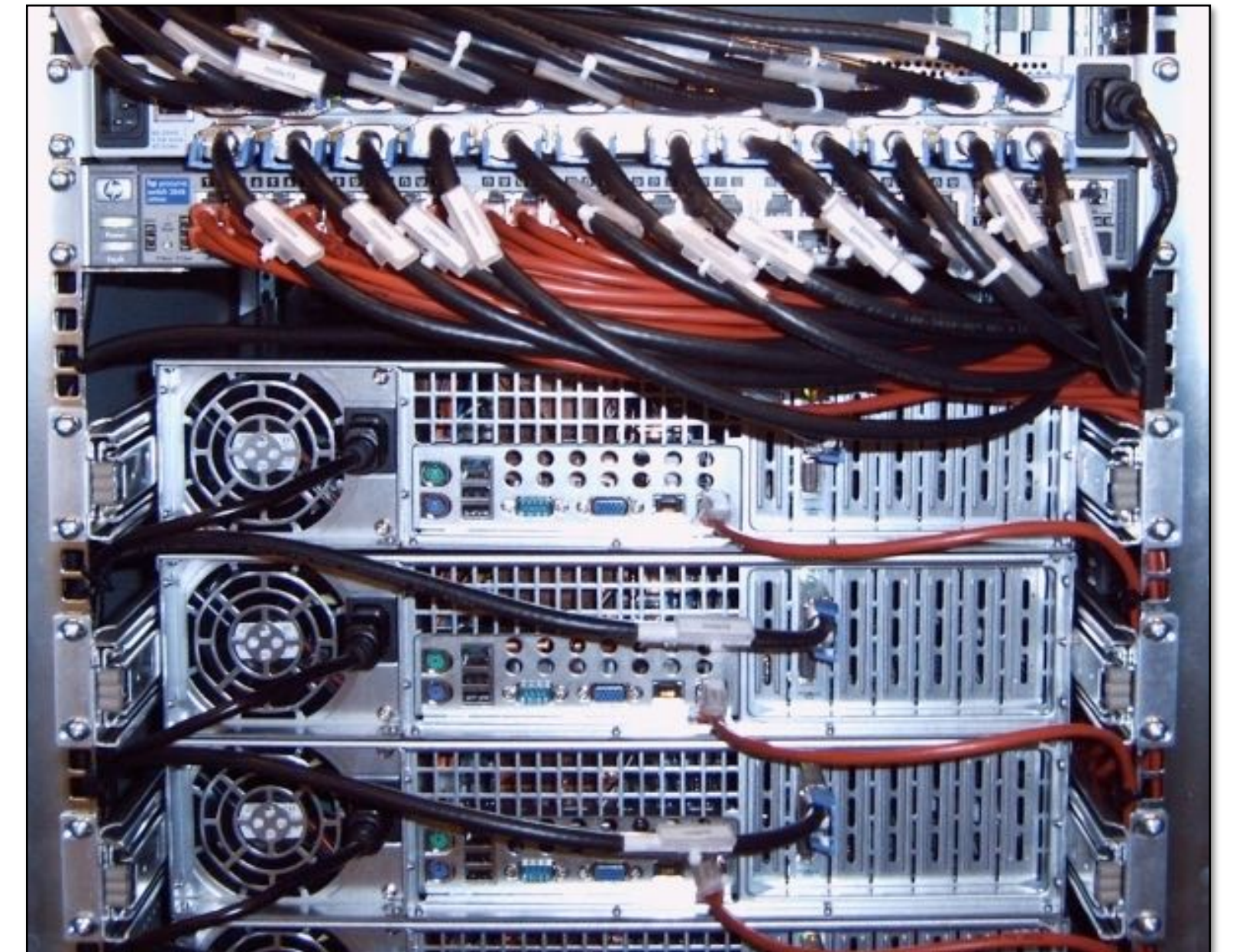
(Communication operations shown in red)

A common metaphor: snail mail



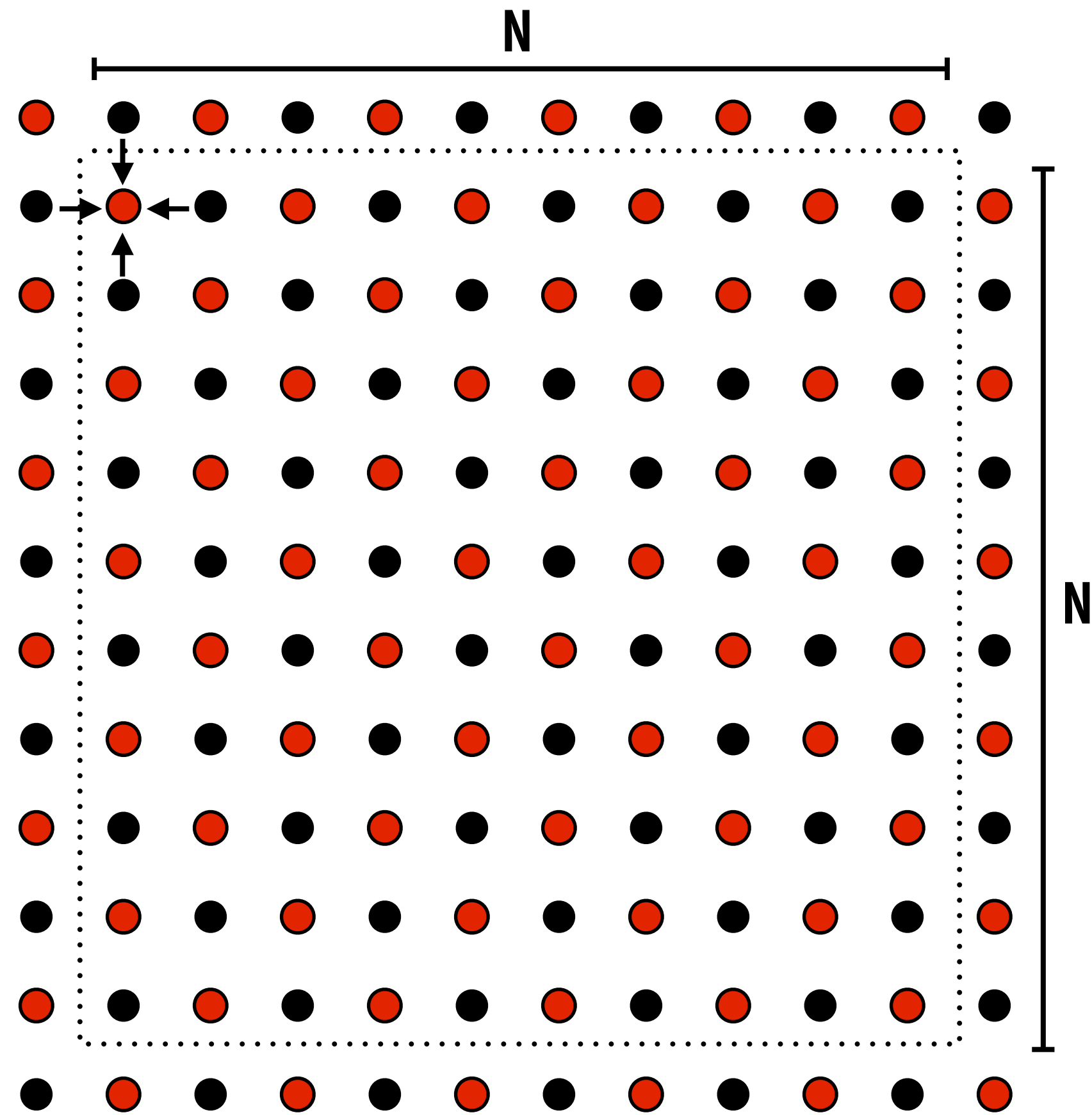
Message passing (implementation)

- **Hardware need not implement a single shared address space for all processors (it only needs to provide mechanisms to communicate messages between nodes)**
 - **Can connect commodity systems together to form a large parallel machine (message passing is a programming model for clusters and supercomputers)**



**Cluster of workstations
(Infiniband network)**

Message passing expression of solver



Recall the grid solver application:

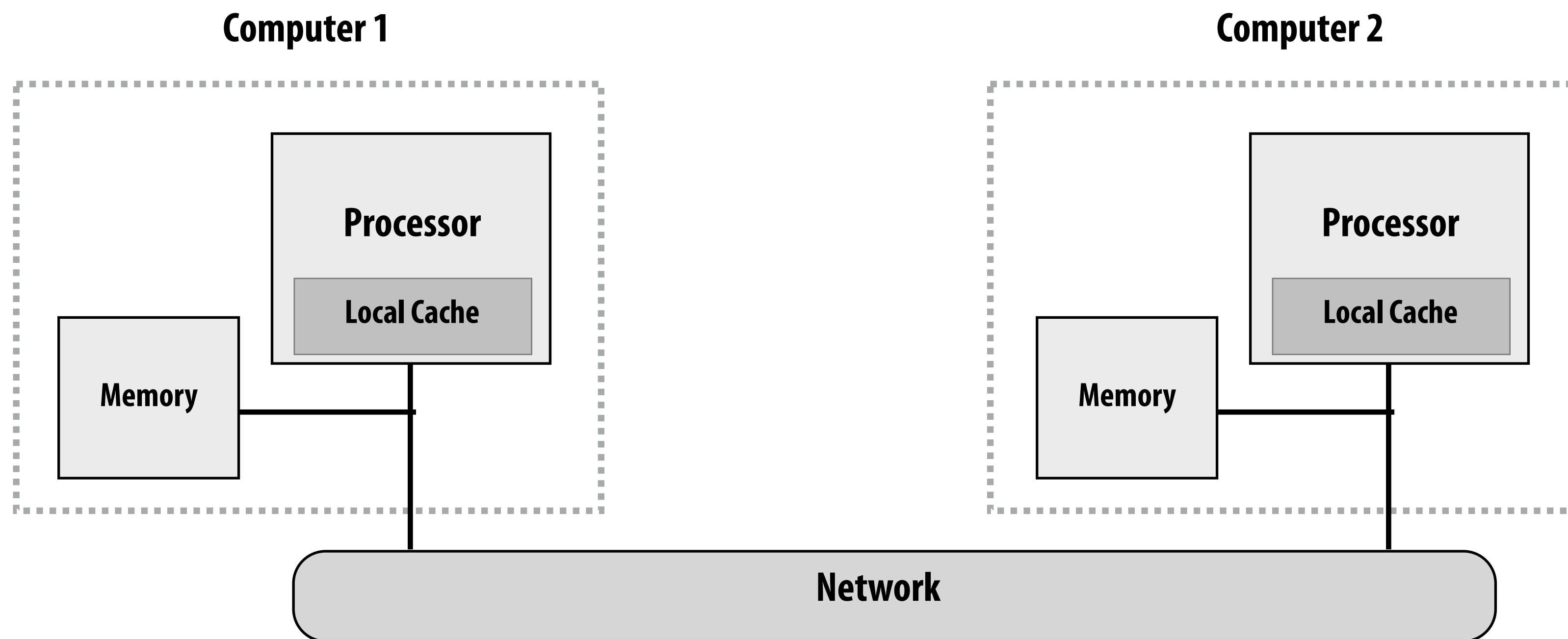
Update all red cells in parallel

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

Repeat until convergence

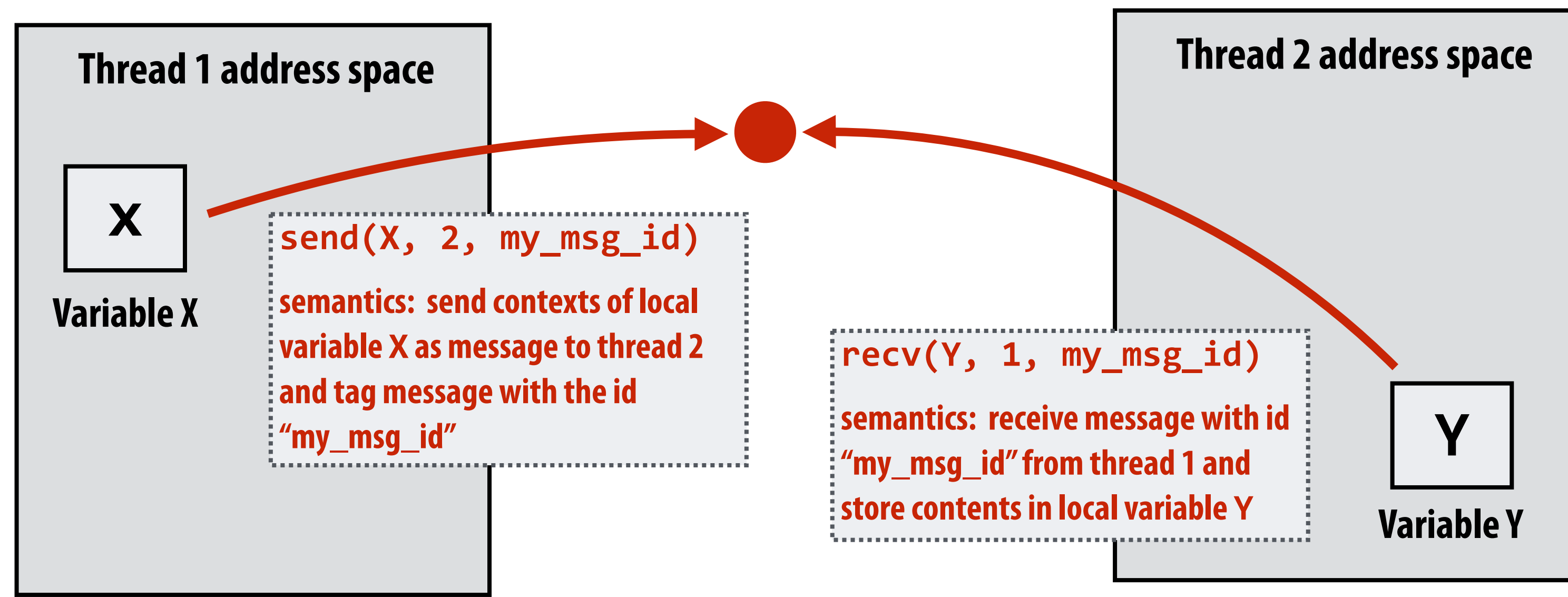
Let's think about expressing a parallel grid solver with communication via messages

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



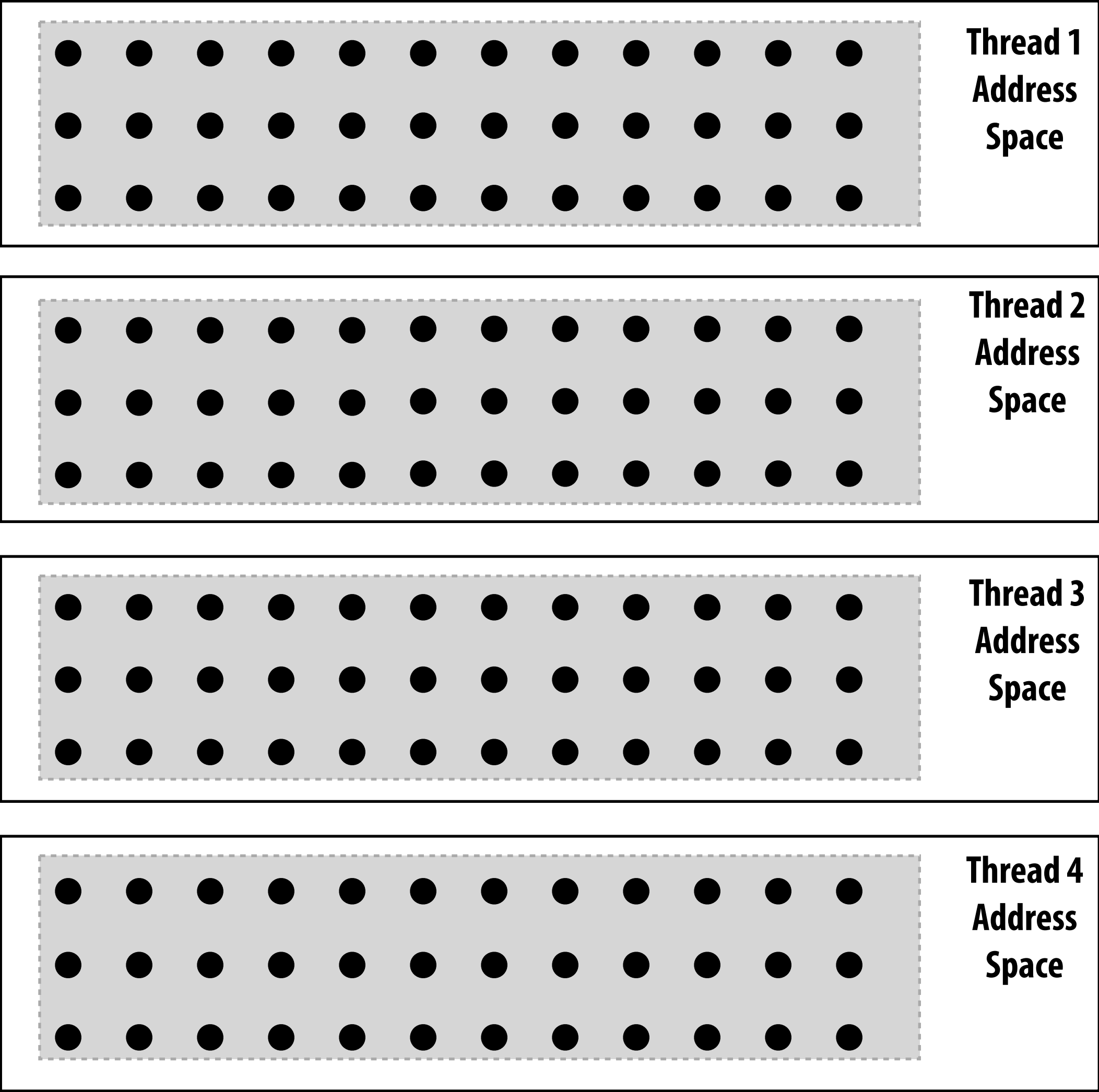
Review: message passing model

- 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?



(Communication operations shown in red)

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

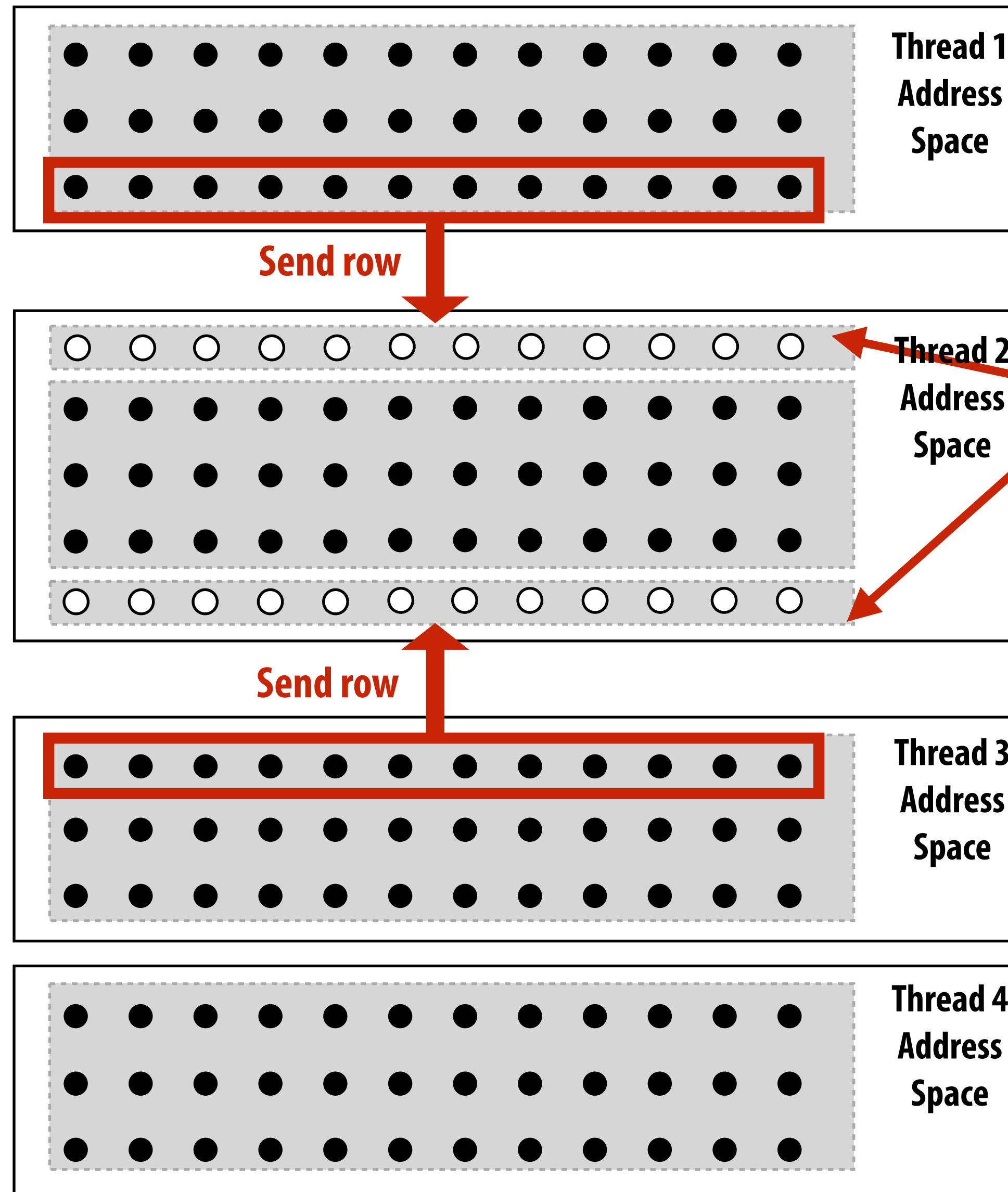


In this figure: four threads

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

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

```
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, MSG_ID_DONE);
        }
    }
}
```

Notes on the message passing example

■ Computation

- Array indexing is relative to local address space

■ Communication:

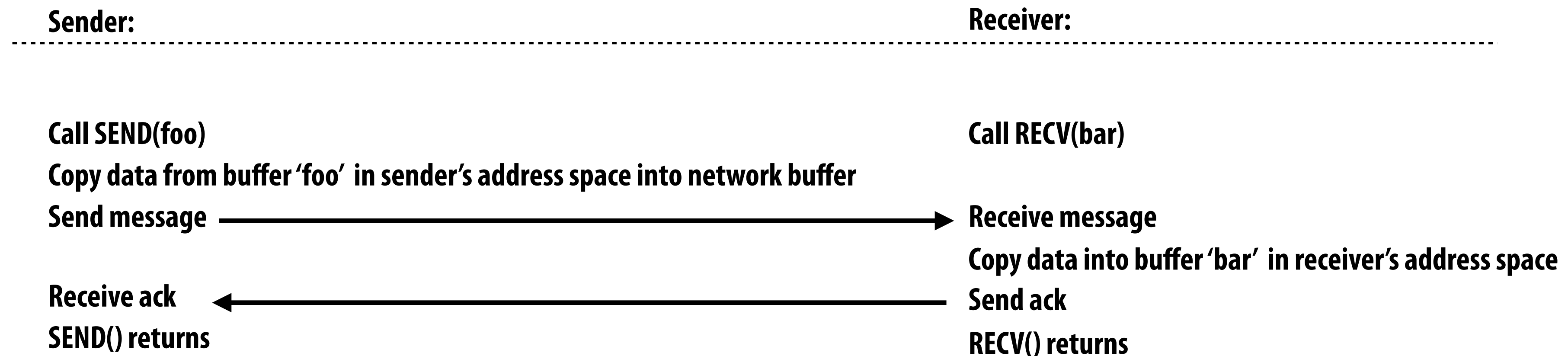
- Performed by sending and receiving messages
- Bulk transfer: communicate entire rows at a time

■ 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**



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

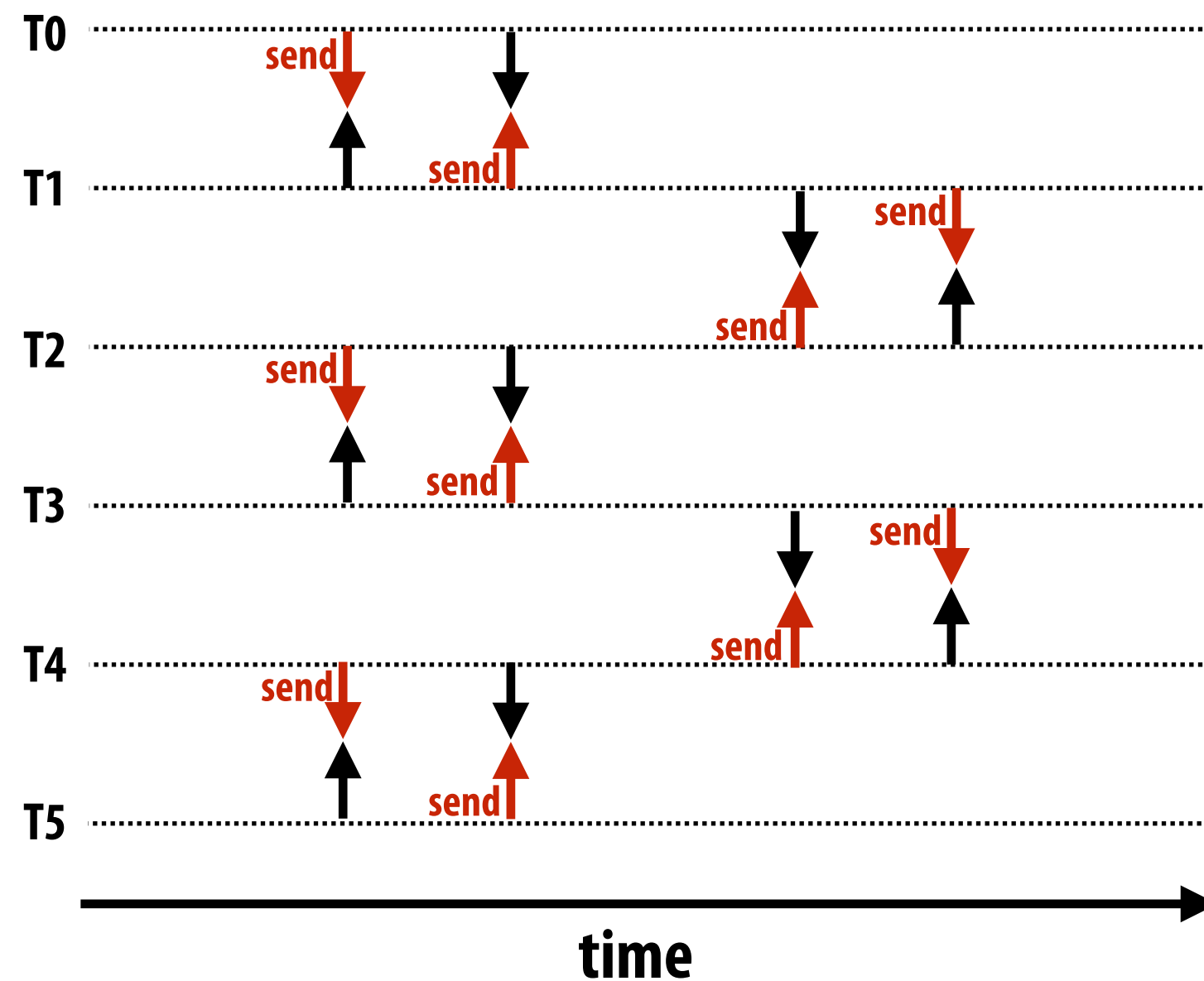
Why?

How can we fix it?

(while still using synchronous send/rcv)

Message passing solver (fixed to avoid deadlock)

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



```

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(); rcvDown();
            sendUp();   rcvUp();
        } else {
            rcvUp();   sendUp();
            rcvDown(); sendDown();
        }

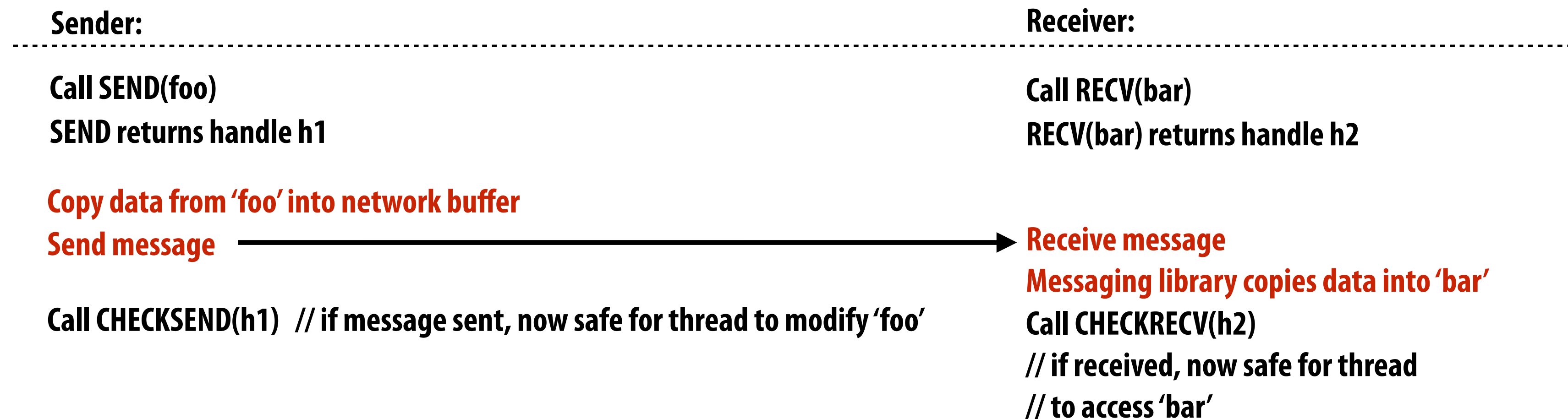
        for (int i=1; i<rows_per_thread-1; i++) {
            for (int j=1; j<n+1; j++) {
                float prev = localA[i,j];
                localA[i,j] = 0.2 * (localA[i-1,j] + localA[i,j] + localA[i+1,j] +
                                   localA[i,j-1] + localA[i,j+1]);
                my_diff += fabs(localA[i,j] - prev);
            }
        }

        if (tid != 0) {
            send(&mydiff, sizeof(float), 0, MSG_ID_DIFF);
            rcv(&done, sizeof(bool), 0, MSG_ID_DONE);
        } else {
            float remote_diff;
            for (int i=1; i<get_num_threads()-1; i++) {
                rcv(&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);
        }
    }
}

```

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



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:

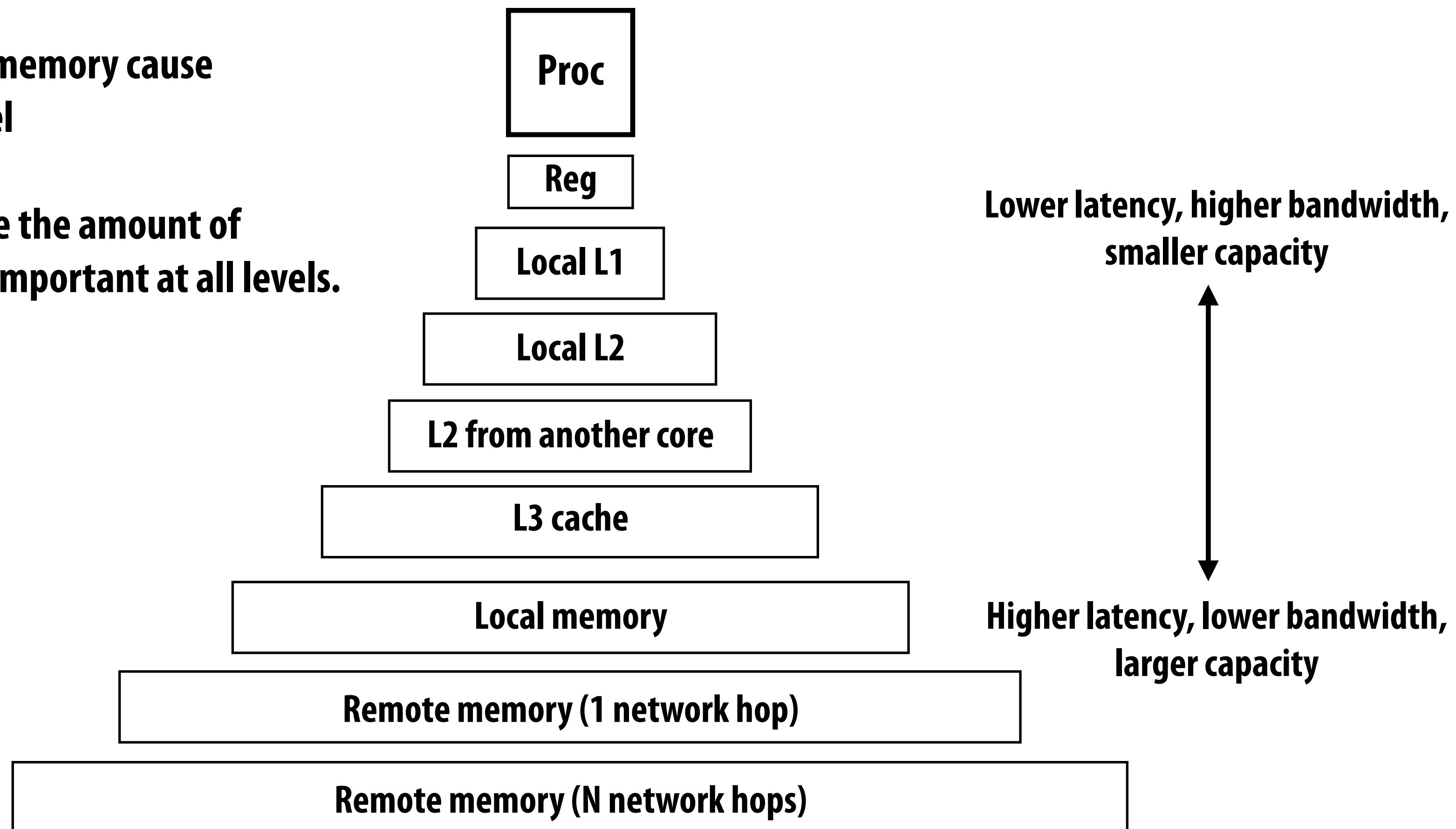
Communication between cores on a chip
Communication between a core and its cache
Communication between a core and memory

Think of a parallel system as an extended memory hierarchy

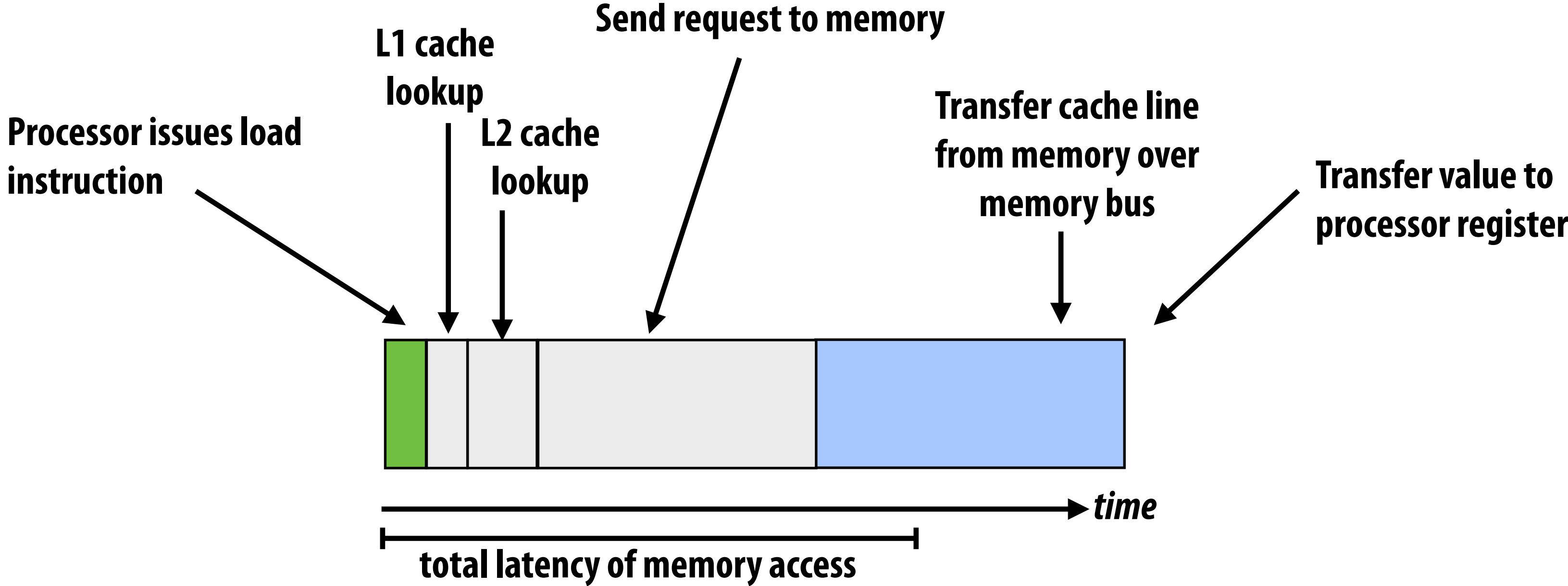
I want you to think of “communication” generally:

- 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)

View from one processor







One example: CPU to memory communication



 = Time to send cache line over memory bus

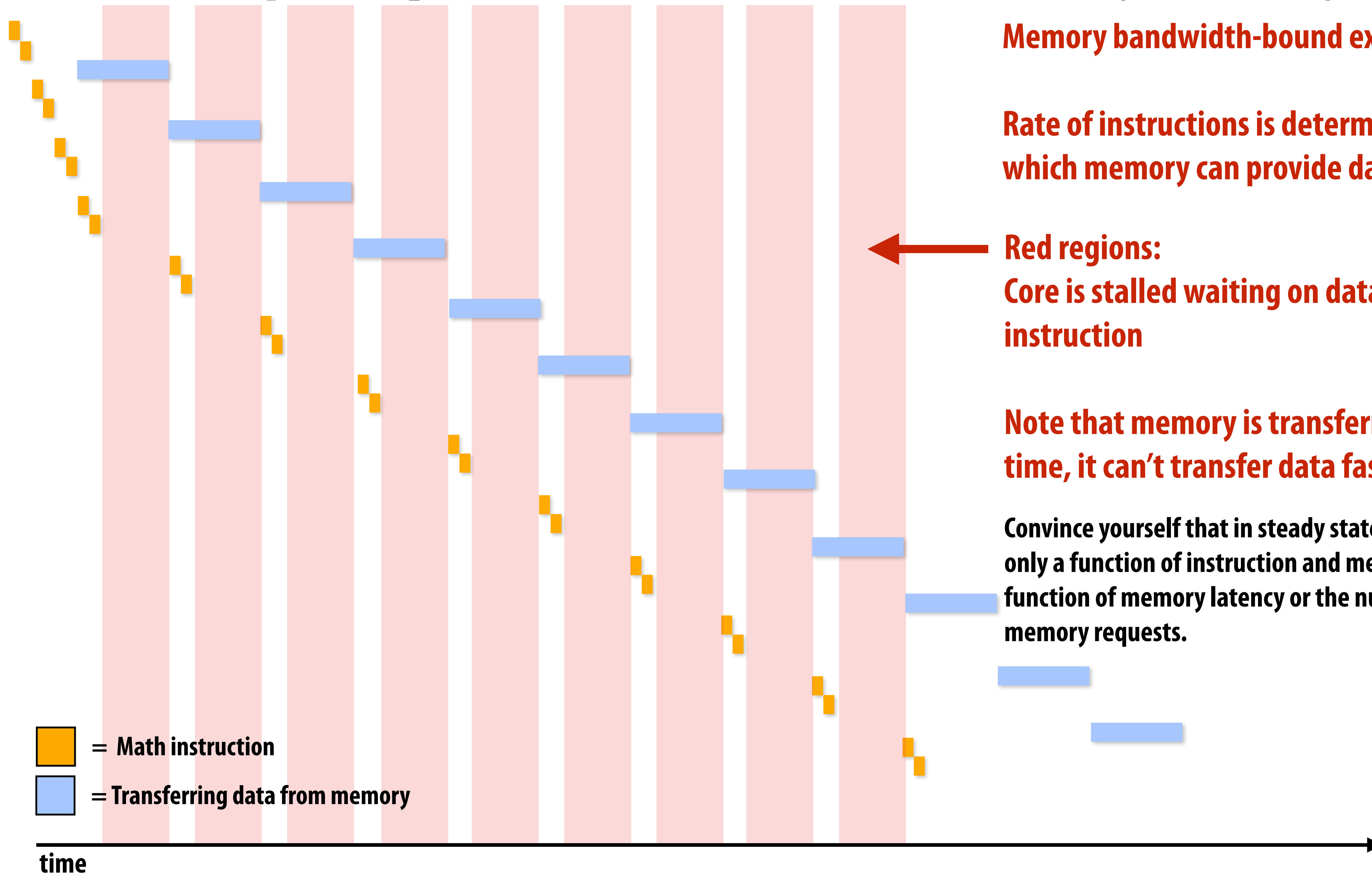
Recall discussion of bandwidth limited execution:

This was an example where the processor executed 2 instructions for each cache line load

-  = Math instruction
-  = Load instruction
-  = Load command sent to memory (part of mem latency)
-  = Transferring data from memory

time

Rate of completing math instructions is limited by memory bandwidth



Memory bandwidth-bound execution!

Rate of instructions is determined by the rate at which memory can provide data.

**Red regions:
Core is stalled waiting on data for next instruction**

Note that memory is transferring data 100% of time, it can't transfer data faster.

Convince yourself that in steady state core underutilization is only a function of instruction and memory throughput, not a function of memory latency or the number of outstanding memory requests.

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?**

Arithmetic intensity

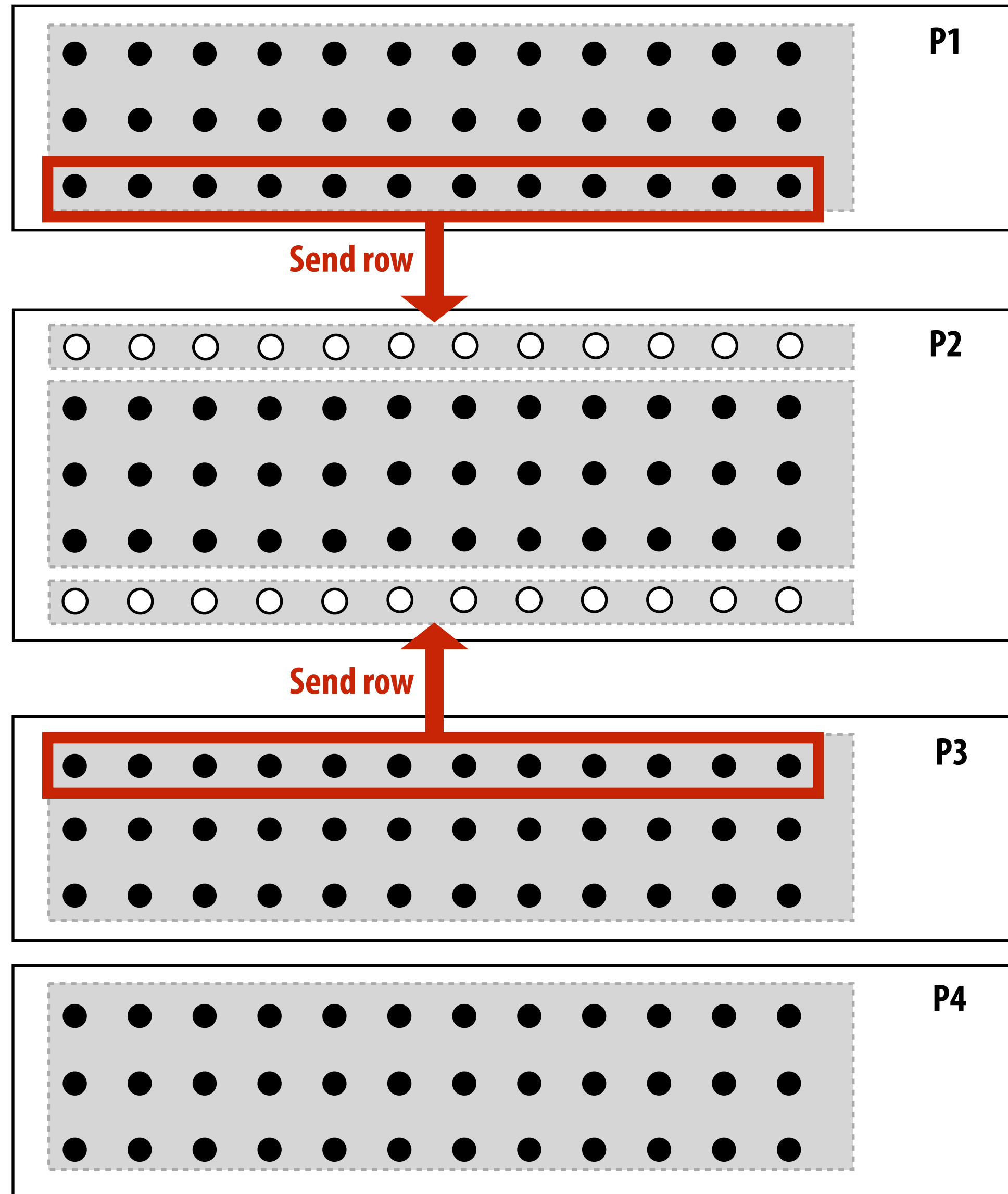
amount of computation (e.g., instructions)

amount of communication (e.g., bytes)

- If numerator is the execution time of computation, ratio gives average bandwidth requirement of code
- $1 / \text{“Arithmetic intensity”} = \text{communication-to-computation ratio}$
 - Some people like to refer to 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 lecture 3)

Two reasons for communication: inherent vs. artifactual communication

Inherent communication



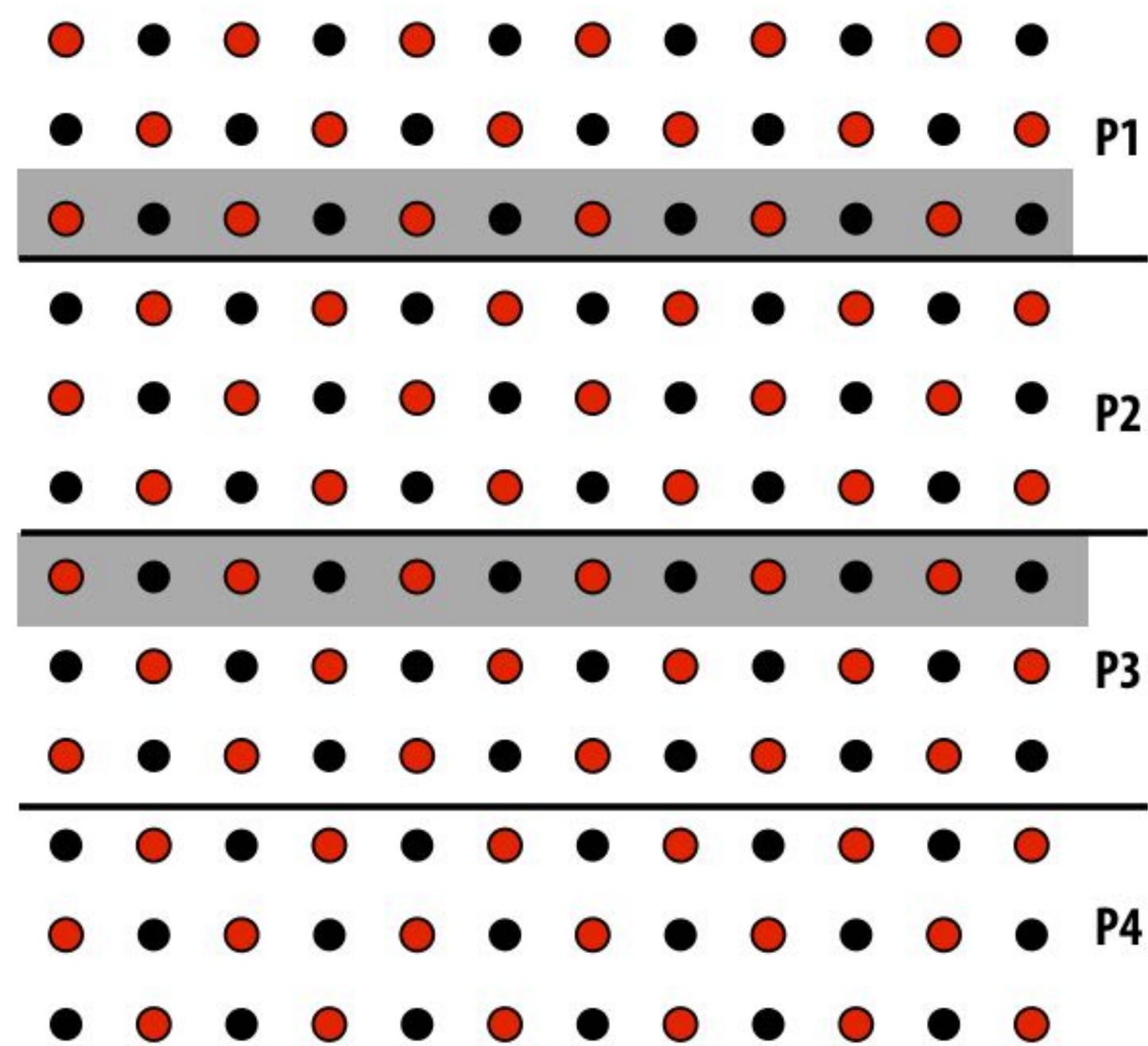
Communication that must occur in a parallel algorithm. The communication is fundamental to the algorithm.

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

Reducing inherent communication

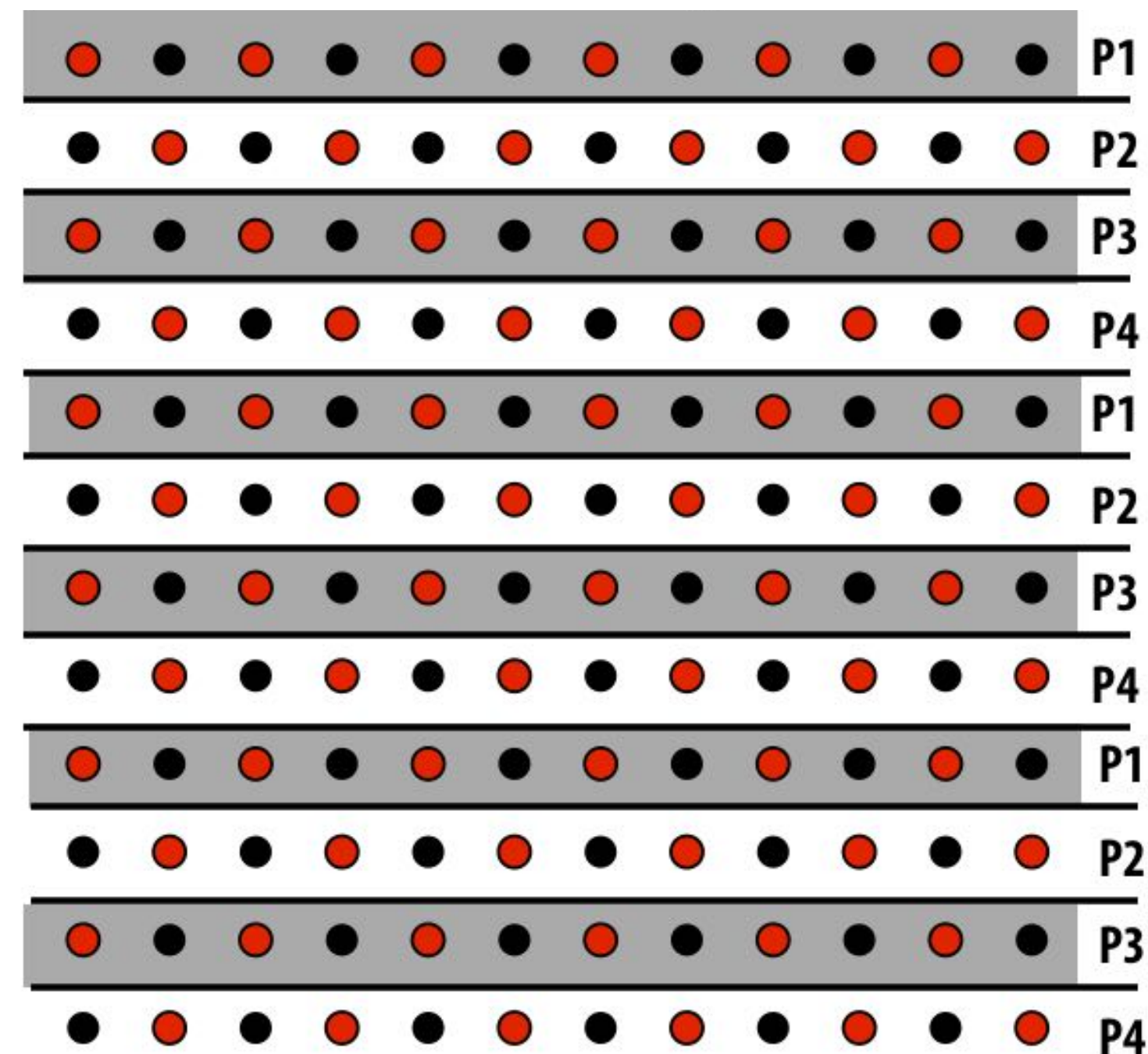
Good assignment decisions can reduce inherent communication
(increase arithmetic intensity)

1D blocked assignment: N x N grid



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

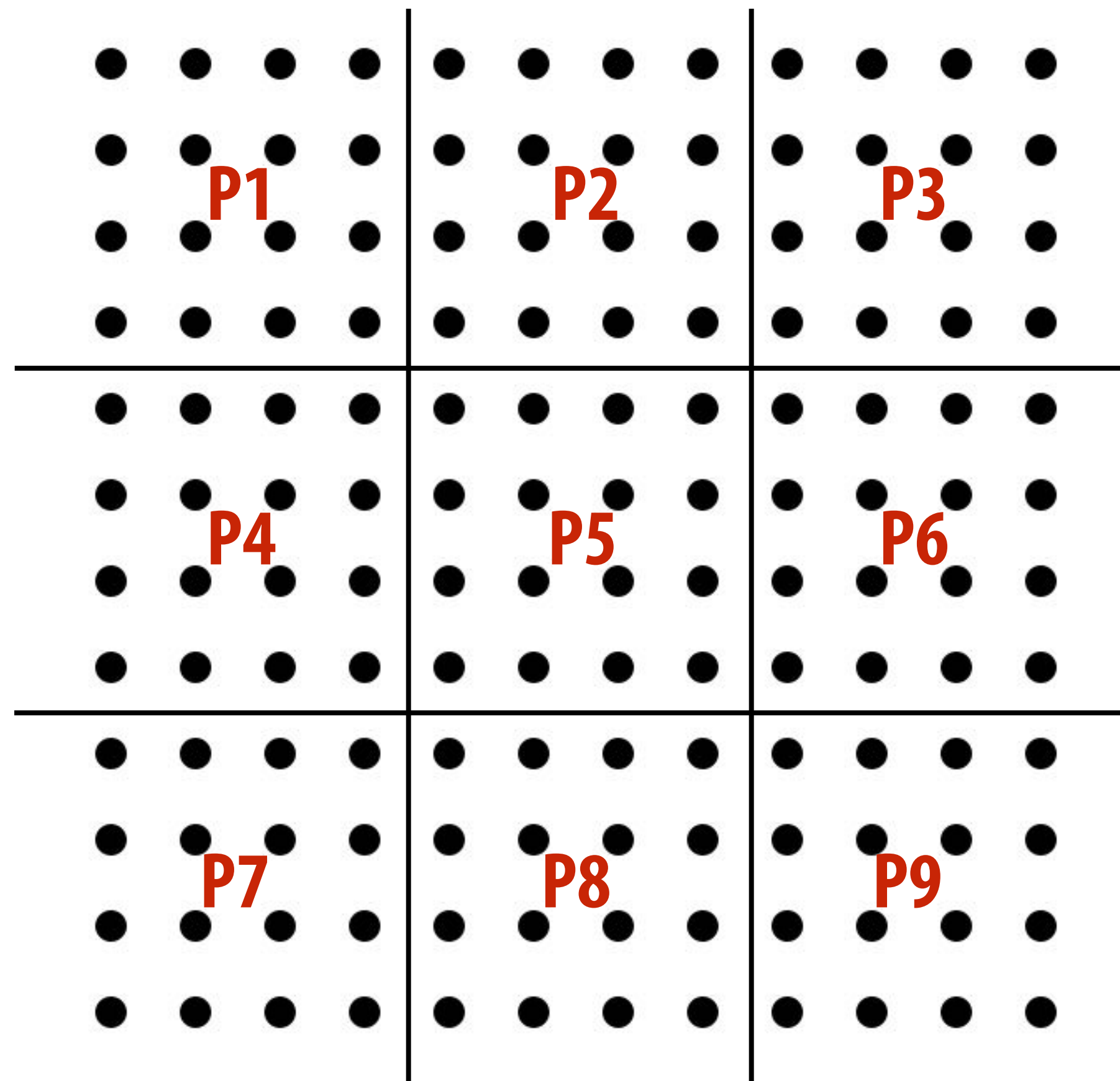
1D interleaved assignment: N x N grid



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

Reducing inherent communication

2D blocked assignment: $N \times N$ 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

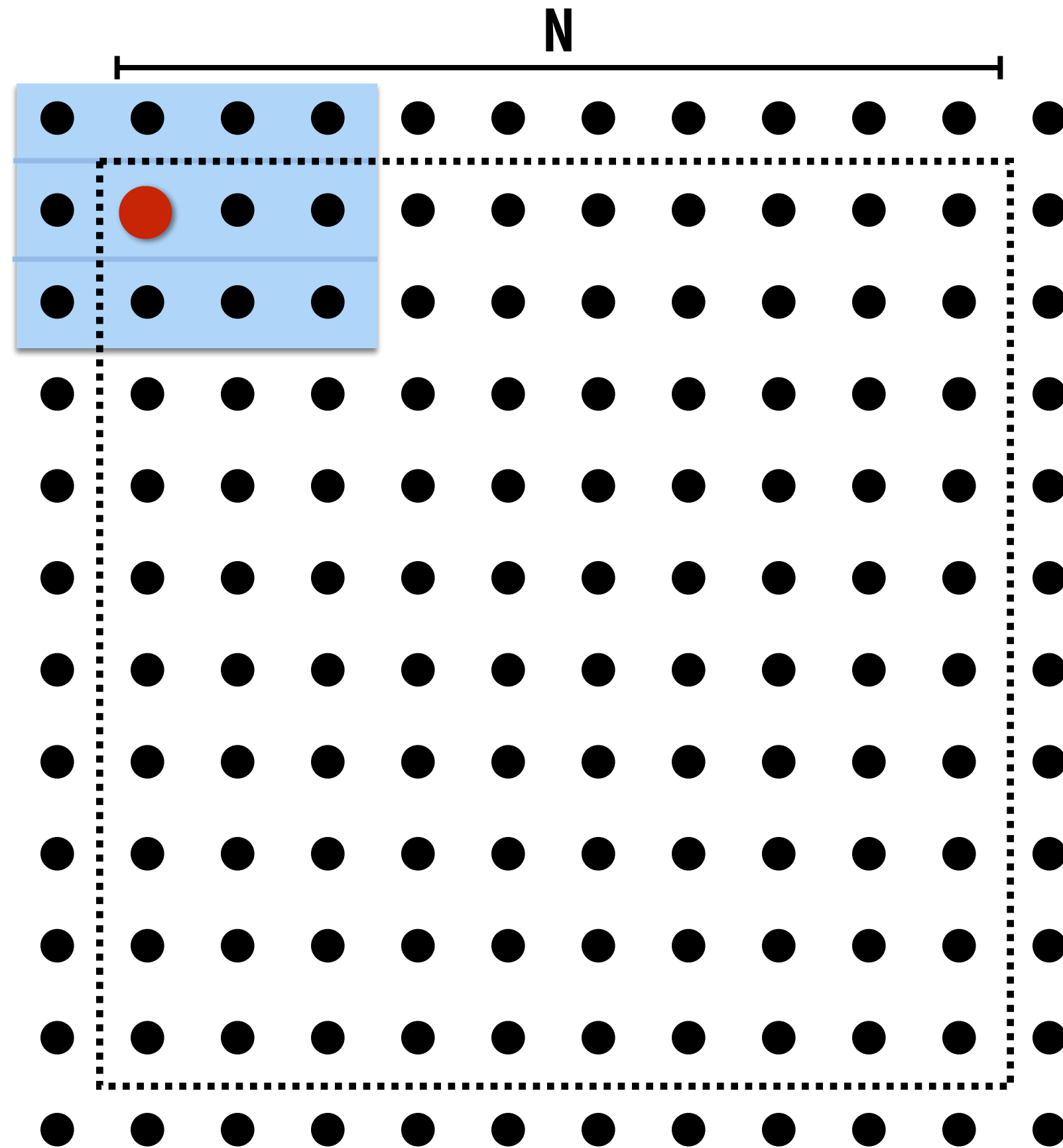
Artifactual communication

- **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)**

Example:
**Artifactual communication arises from
the behavior of caches**

In this case: the communication is between memory and the processor.

Data access in grid solver: row-major traversal



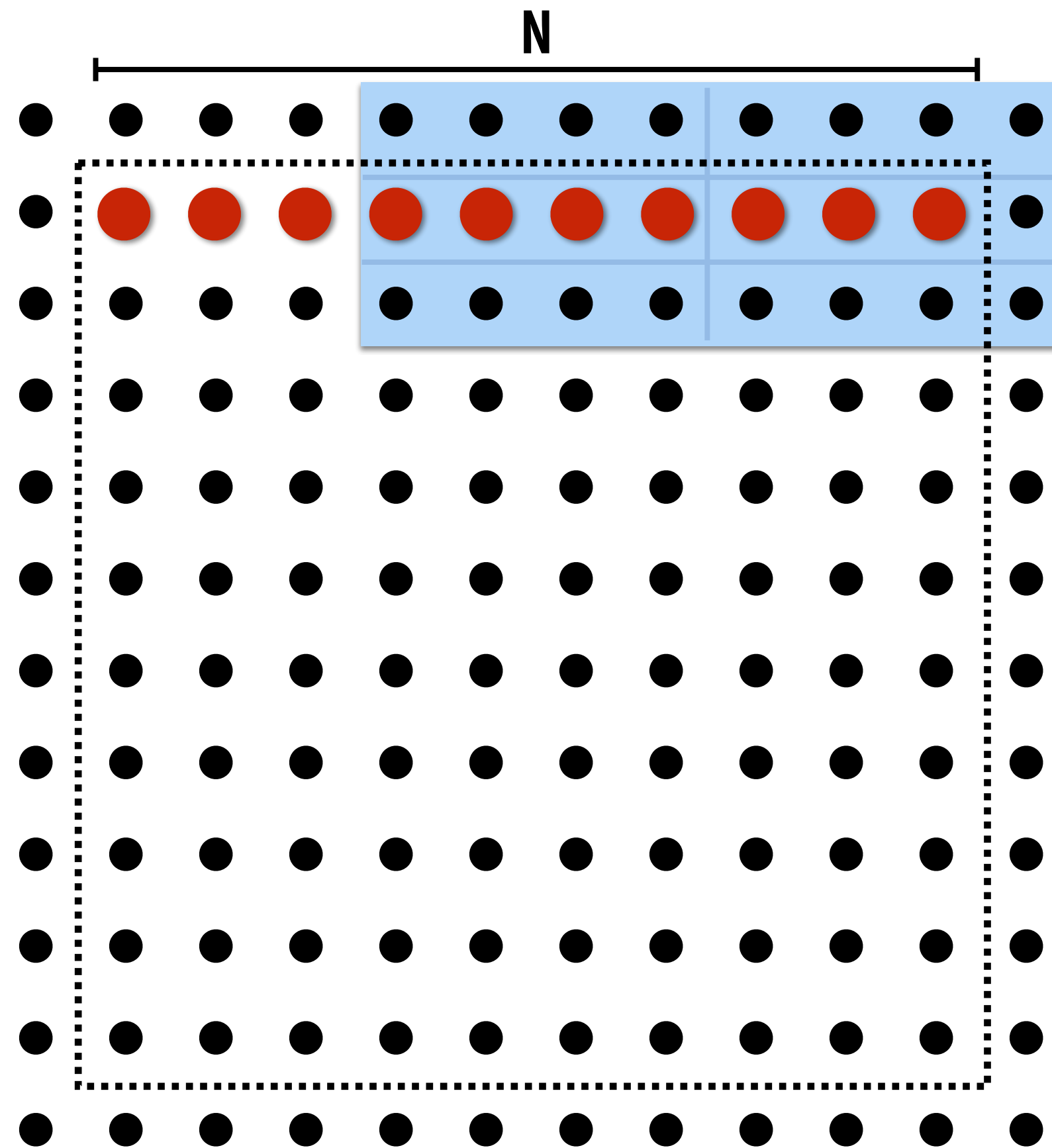
Assume row-major grid layout.

Assume cache line is 4 grid elements.

Cache capacity is 24 grid elements (6 lines)

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

Data access in grid solver: row-major traversal



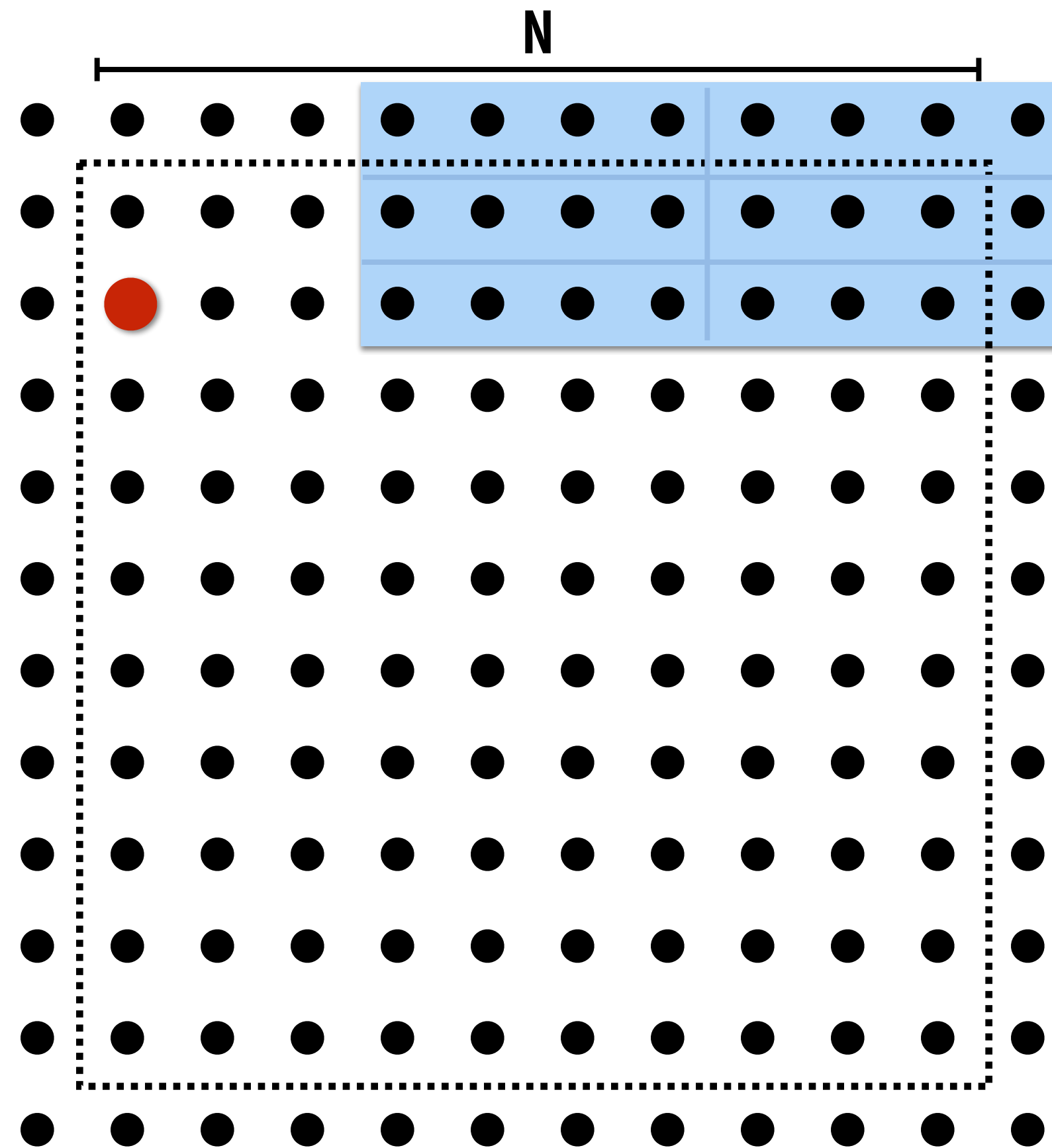
Assume row-major grid layout.

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 $(x,y)=(0,1)$, $(1,1)$, $(2,1)$, $(0,2)$, and $(2,2)$ have been accessed previously, they are no longer present in cache at start of processing the first output element in row 2.

As a result, this program loads three cache lines for every four elements of output.

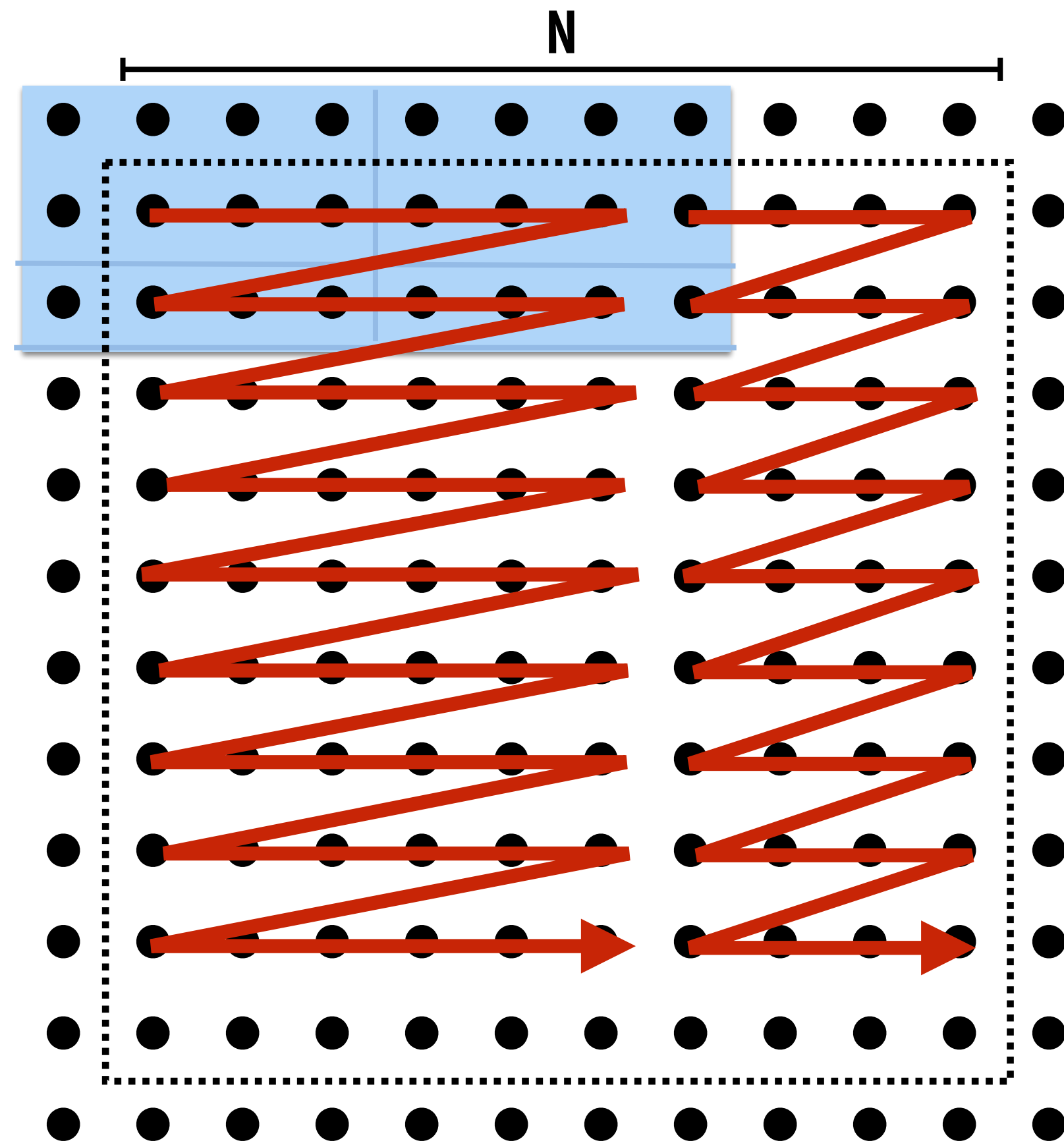
Artifactual communication examples

- **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 since entire cache line was overwritten)**
- **Finite replication capacity: the same data communicated to processor multiple times because cache is too small to retain it between accesses (capacity misses)**

Techniques for reducing communication

Improving temporal locality by changing grid traversal order

“Blocking”: reorder computation to reduce capacity misses



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];  
}
```

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

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

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

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

```
// assume arrays are allocated here
```

```
// compute E = D + ((A + B) * C)
```

```
add(n, A, B, tmp1);
```

```
mul(n, tmp1, C, tmp2);
```

```
add(n, tmp2, D, E);
```

Overall arithmetic intensity = 1/3

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

Four loads, one store per 3 math ops
(arithmetic intensity = 3/5)

```
// compute E = D + (A + B) * C
```

```
fused(n, A, B, C, D, E);
```

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

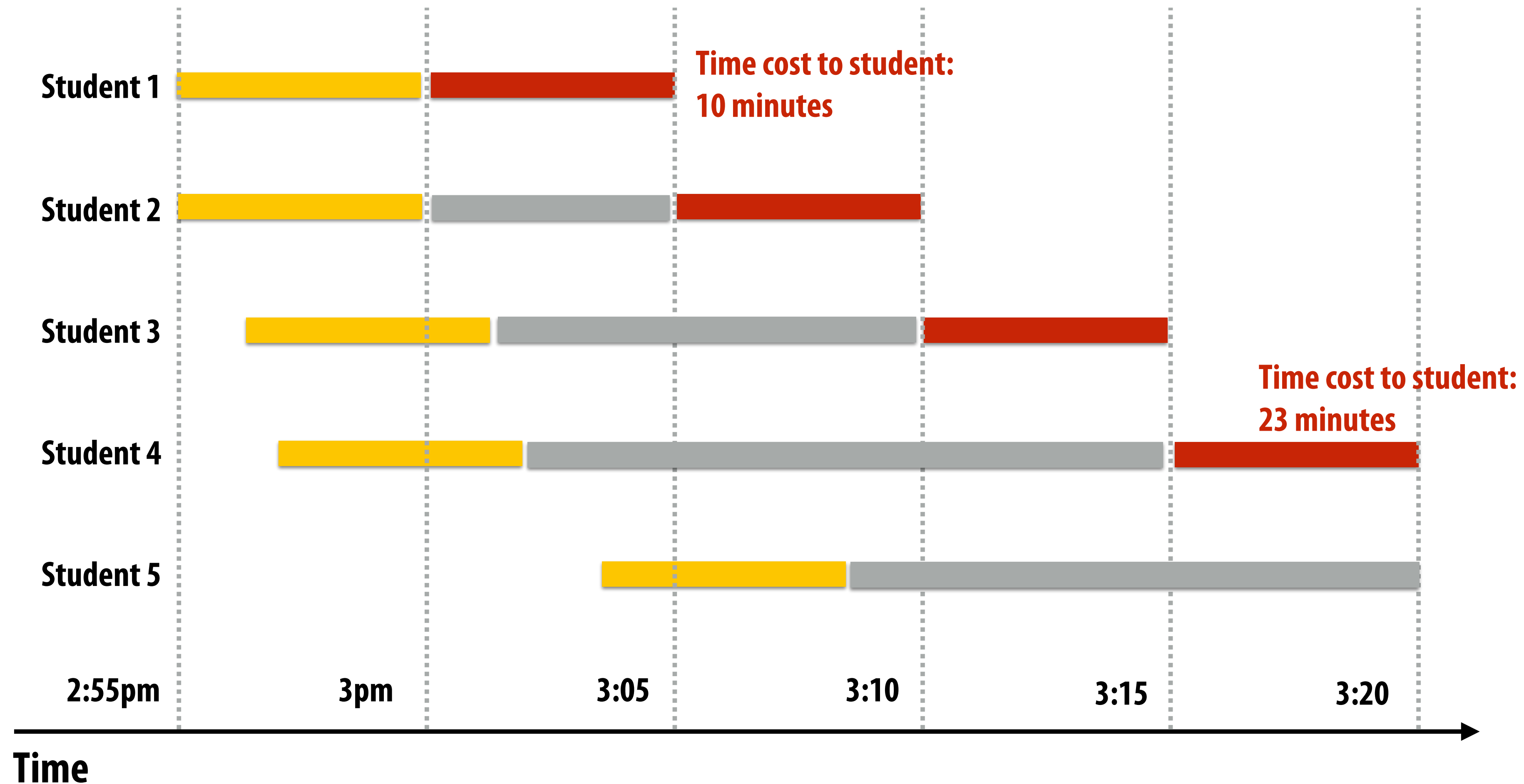
- **Exploit sharing: co-locate tasks that operate on the same data**
 - **Schedule threads working on the same data structure at the same time on the same processor**
 - **Reduces inherent communication**

Contention

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

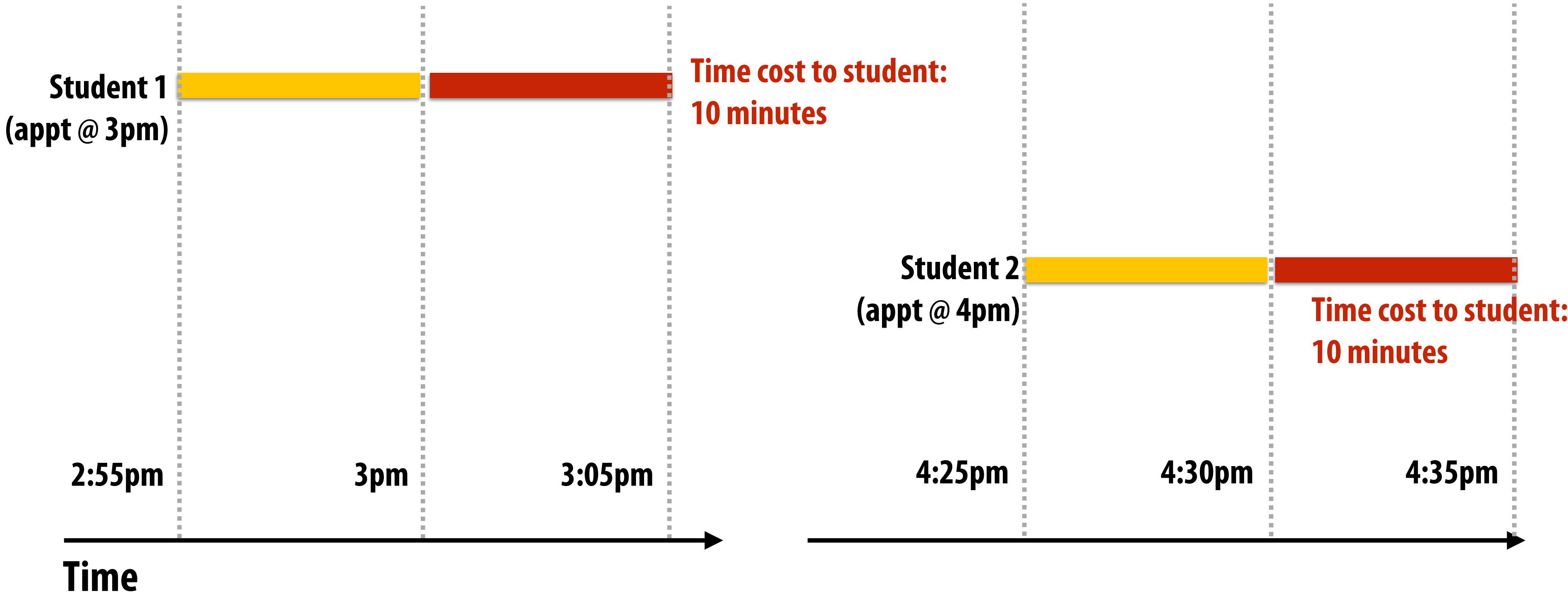
- **Operation to perform: Professor Kayvon helps a student with a question**
- **Execution resource: Professor Kayvon**
- **Steps in operation:**
 1. **Student walks from Bytes Cafe to Kayvon's office (5 minutes)**
 2. **Student waits in line (if necessary)**
 3. **Student gets question answered with 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)

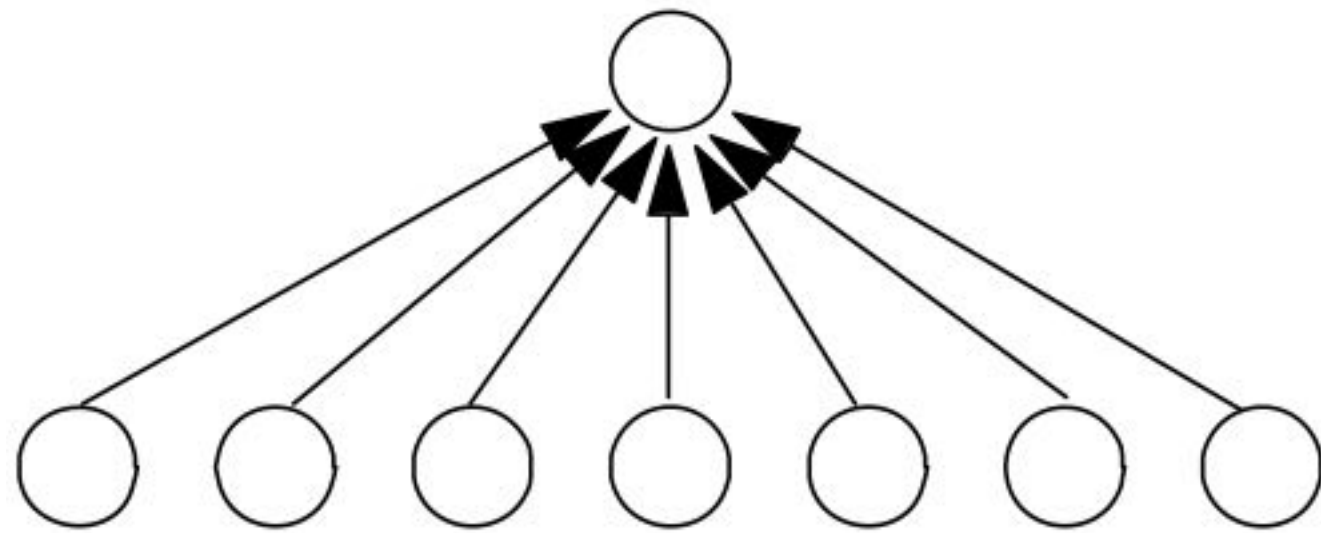
Example: two students make appointments to talk to me about course material (at 3pm and at 4:30pm)



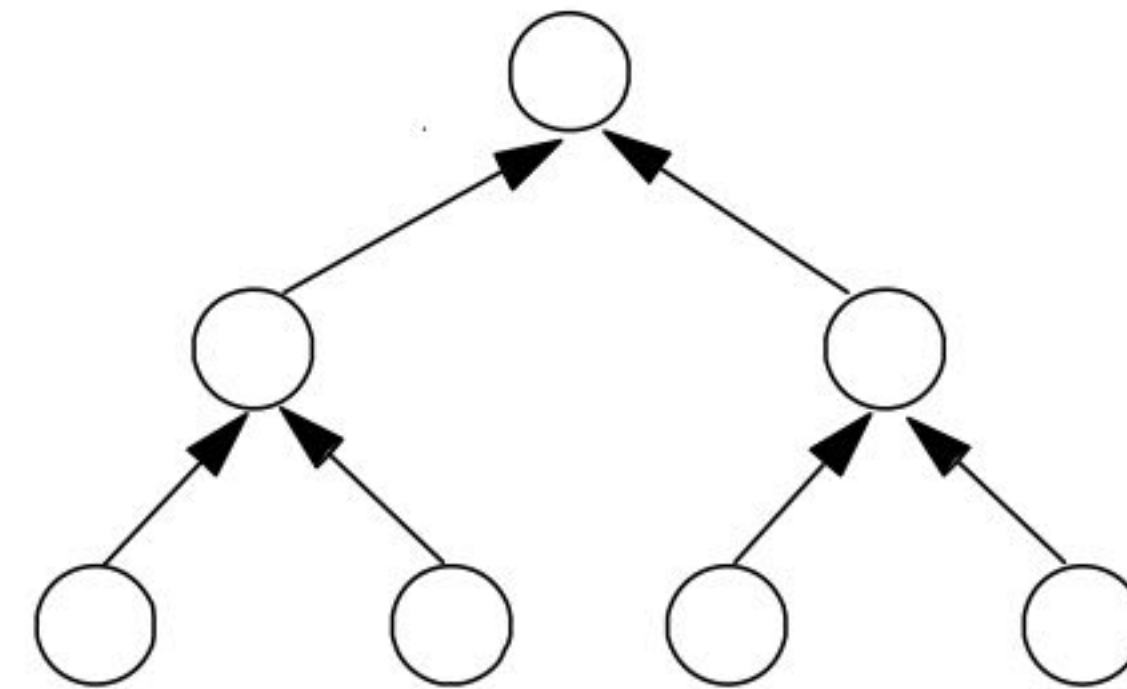
Contention

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

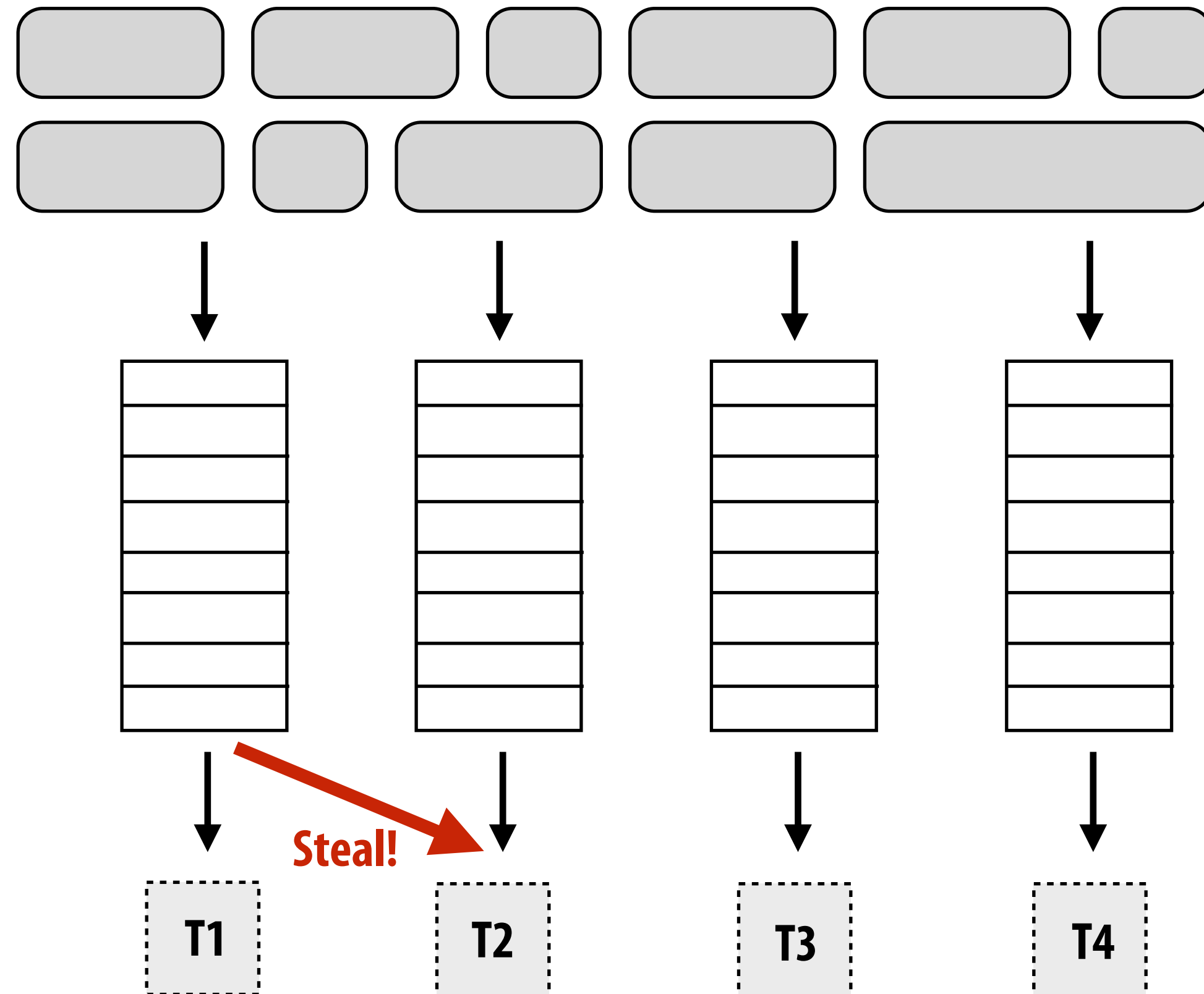
Push new work to OWN work queue

(no contention when all processors have work to do)

When local work queue is empty...

STEAL work from random work queue

(synchronization okay at this point since the thread 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

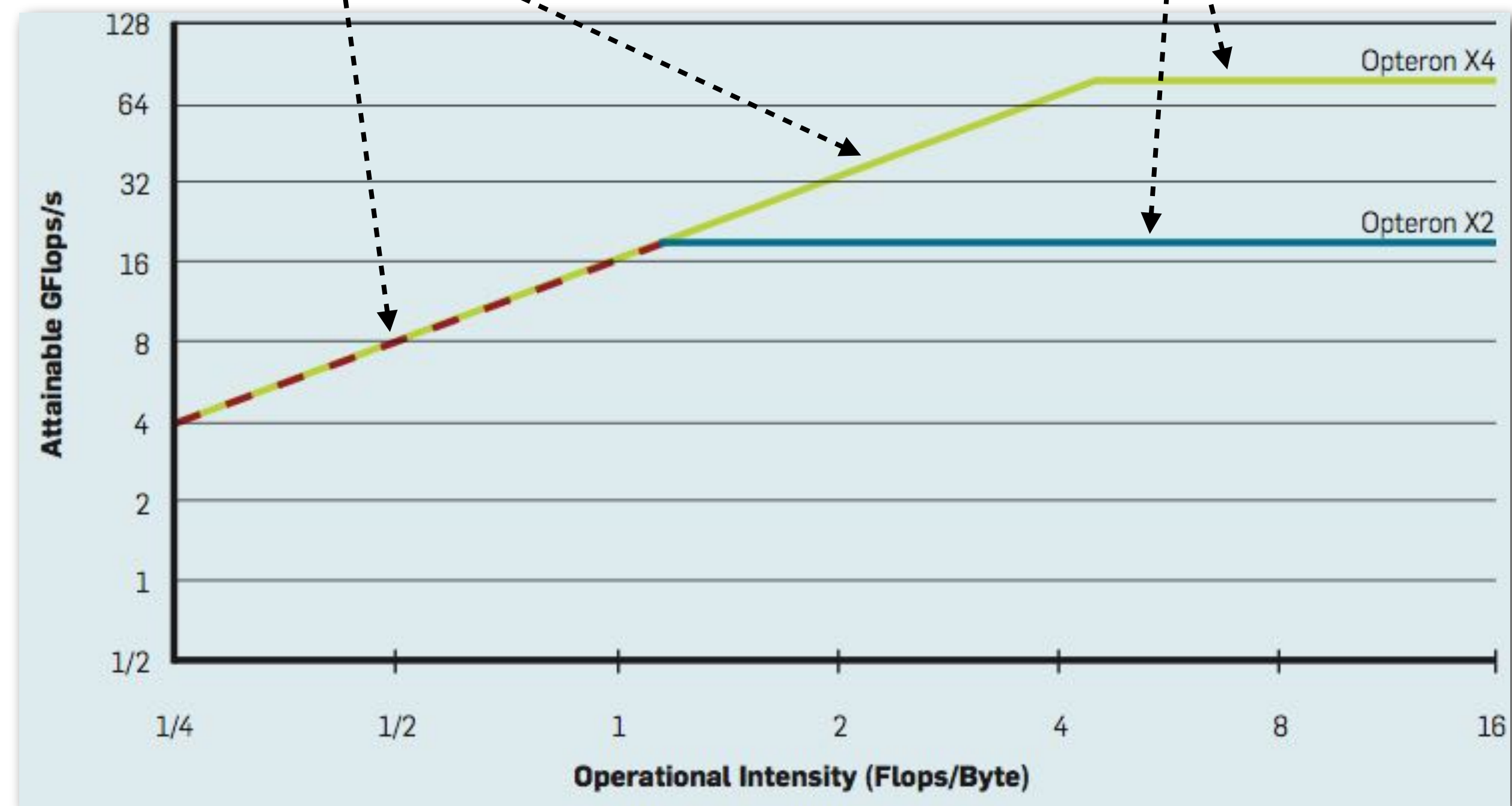
- **Determine if your performance is limited by computation, memory bandwidth (or memory latency), or synchronization?**
- **Try and establish “high watermarks”**
 - **What’s the best you can do in practice?**
 - **How close is your implementation to a best-case scenario?**

Roofline model

- In plot below, different points on the X axis correspond to different programs with different arithmetic intensities
- The Y axis is the maximum obtainable instruction throughput for a program with a given arithmetic intensity

diagonal region: memory bandwidth limited execution

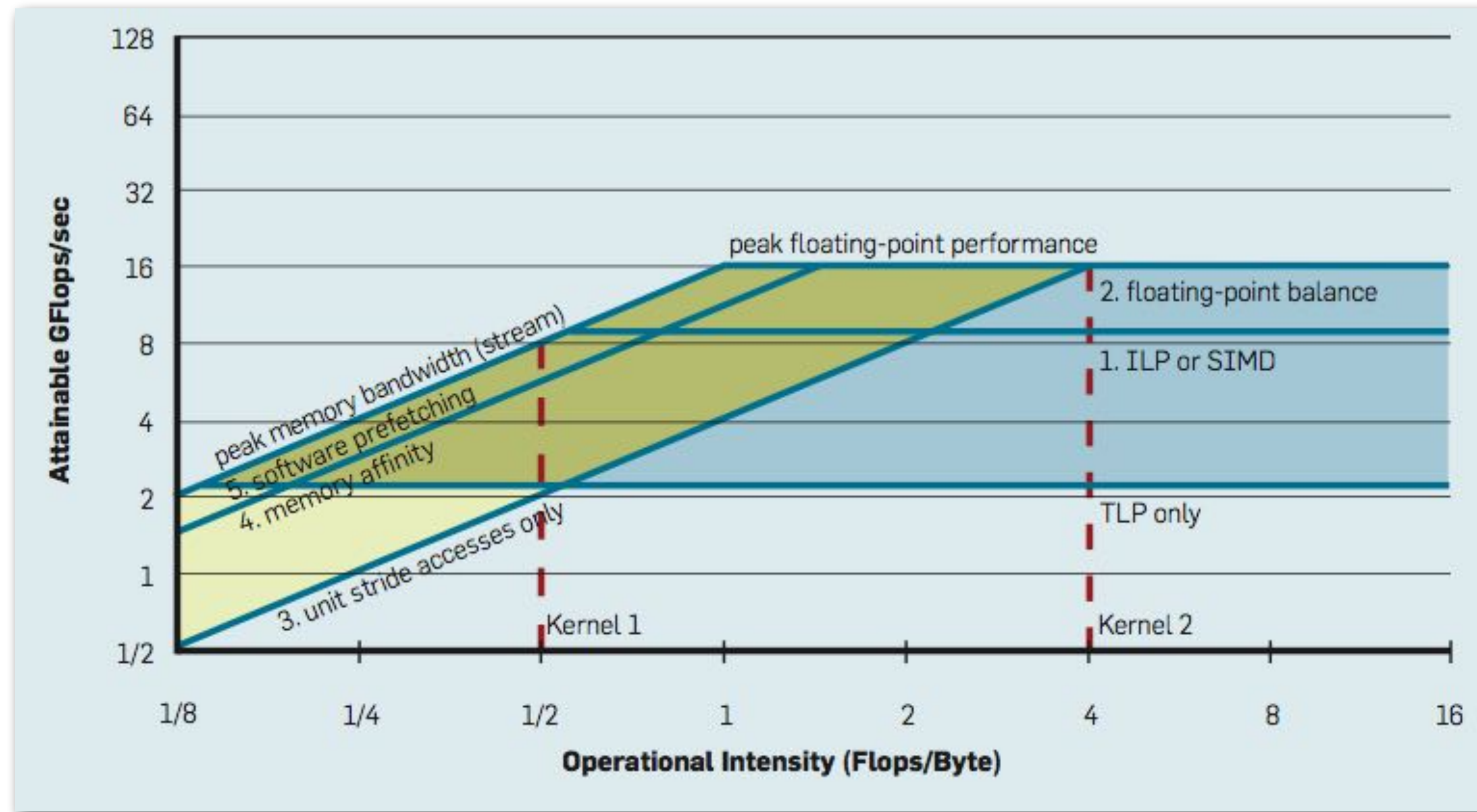
horizontal region: compute limited execution



Roofline model: optimization regions

Use various levels of optimization in benchmarks

(e.g., best performance with and without using SIMD instructions)



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, you might 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

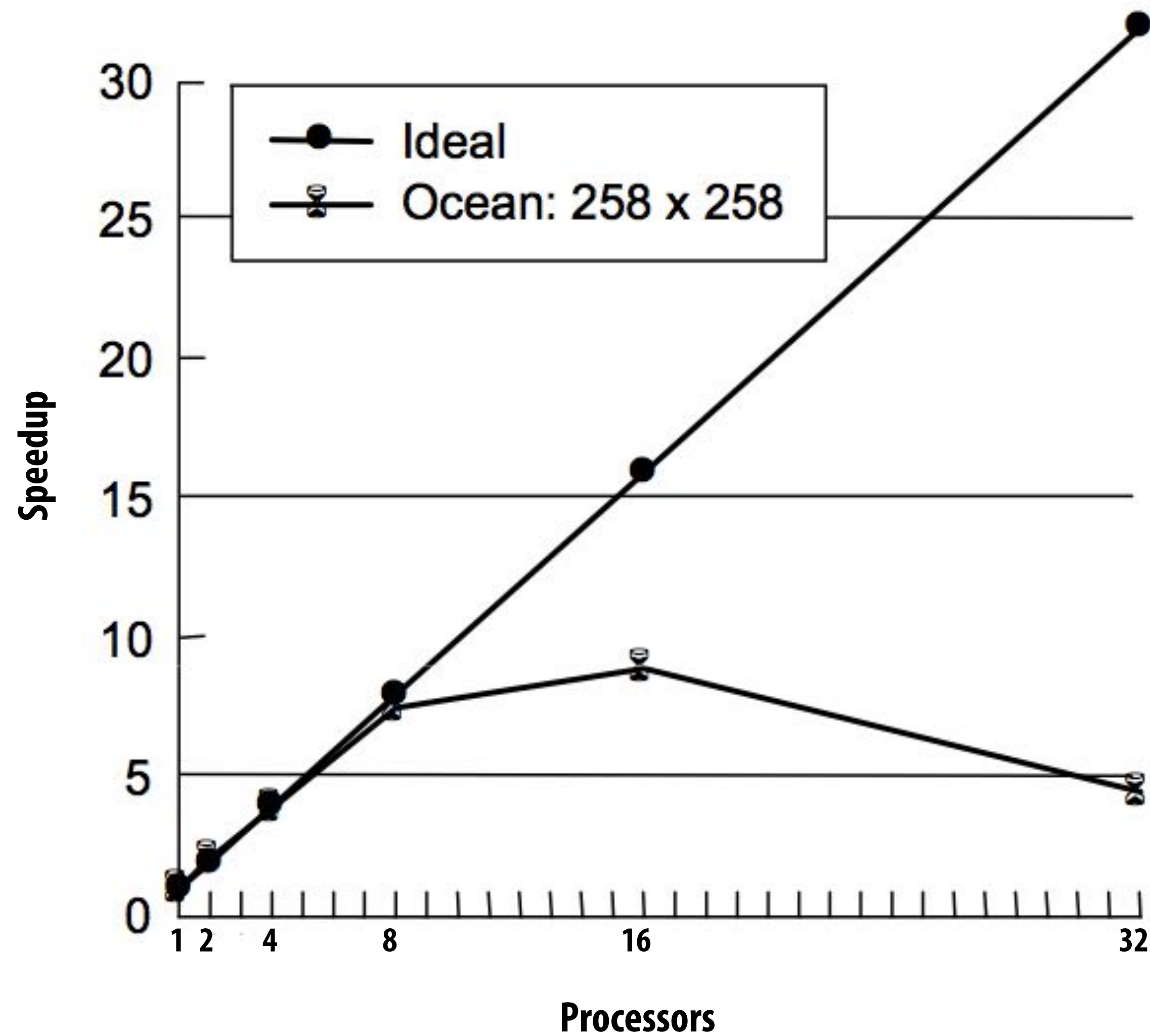
Measuring scaling

- **Consider the grid solver example**
 - **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?**

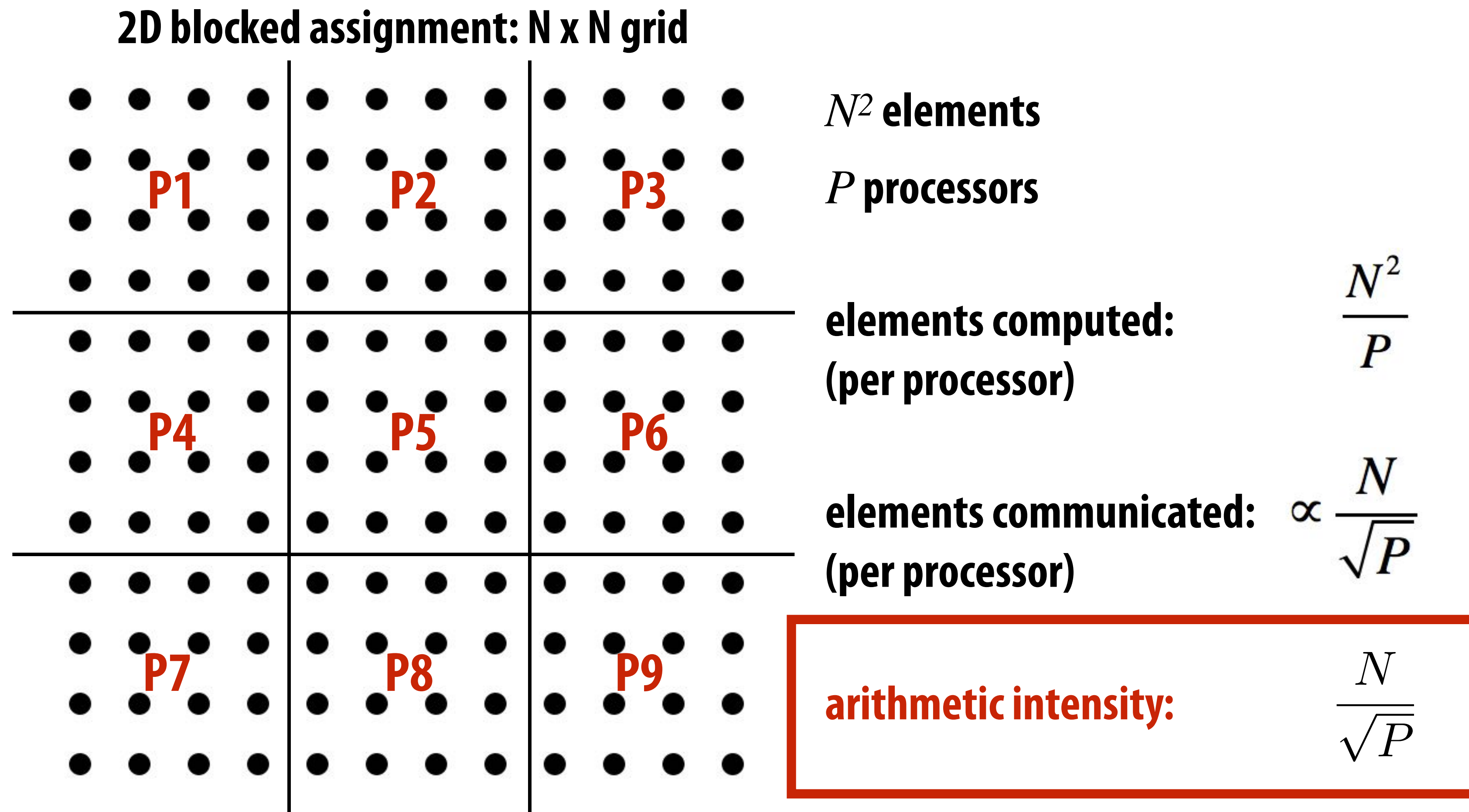
Common pitfall: compare parallel program speedup to parallel algorithm running on one core (easier to make yourself look good)

Speedup of solver application: 258 x 258 grid

Execution on 32 processor SGI Origin 2000



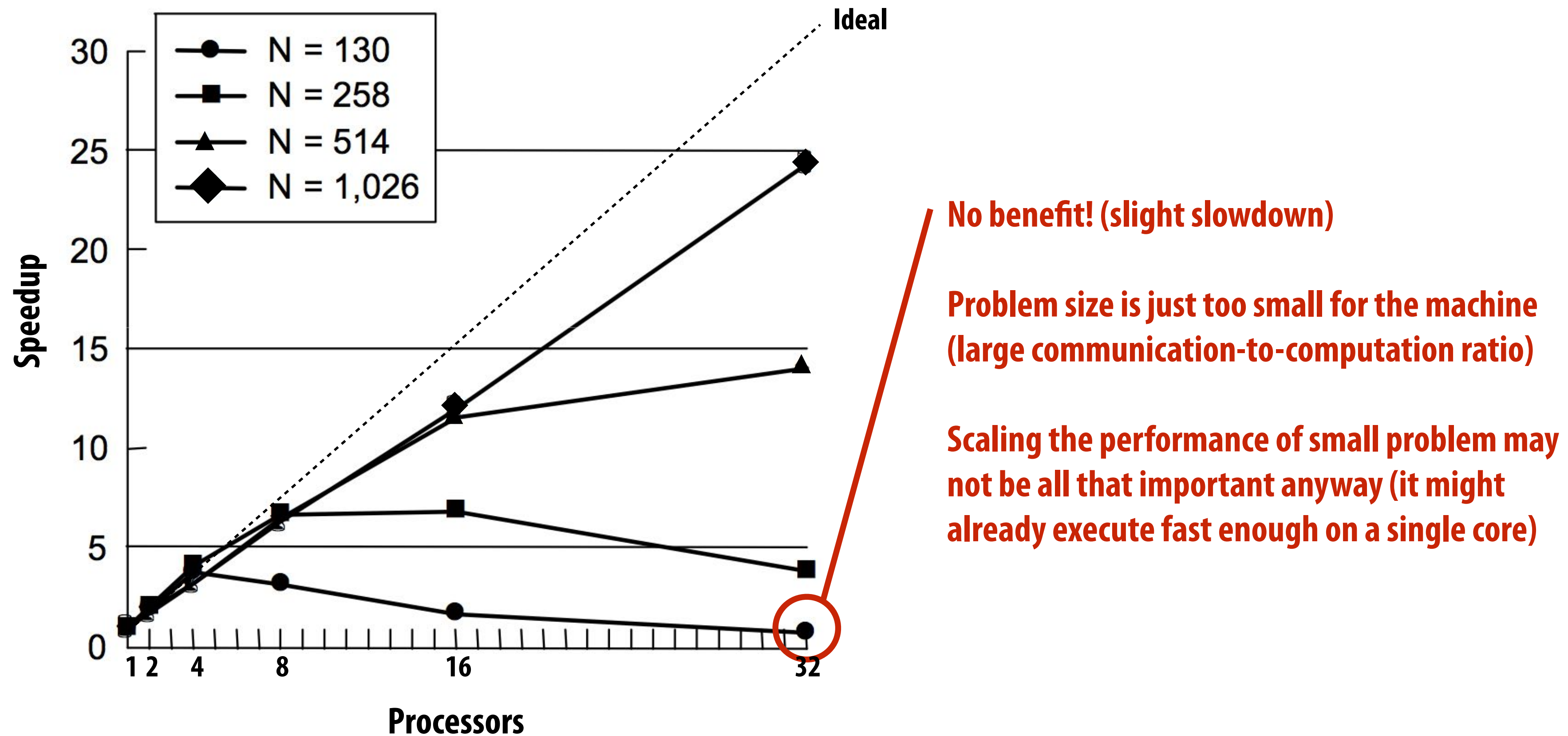
Remember: work assignment in solver



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

Pitfalls of fixed problem size speedup analysis

Solver execution on 32 processor SGI Origin 2000

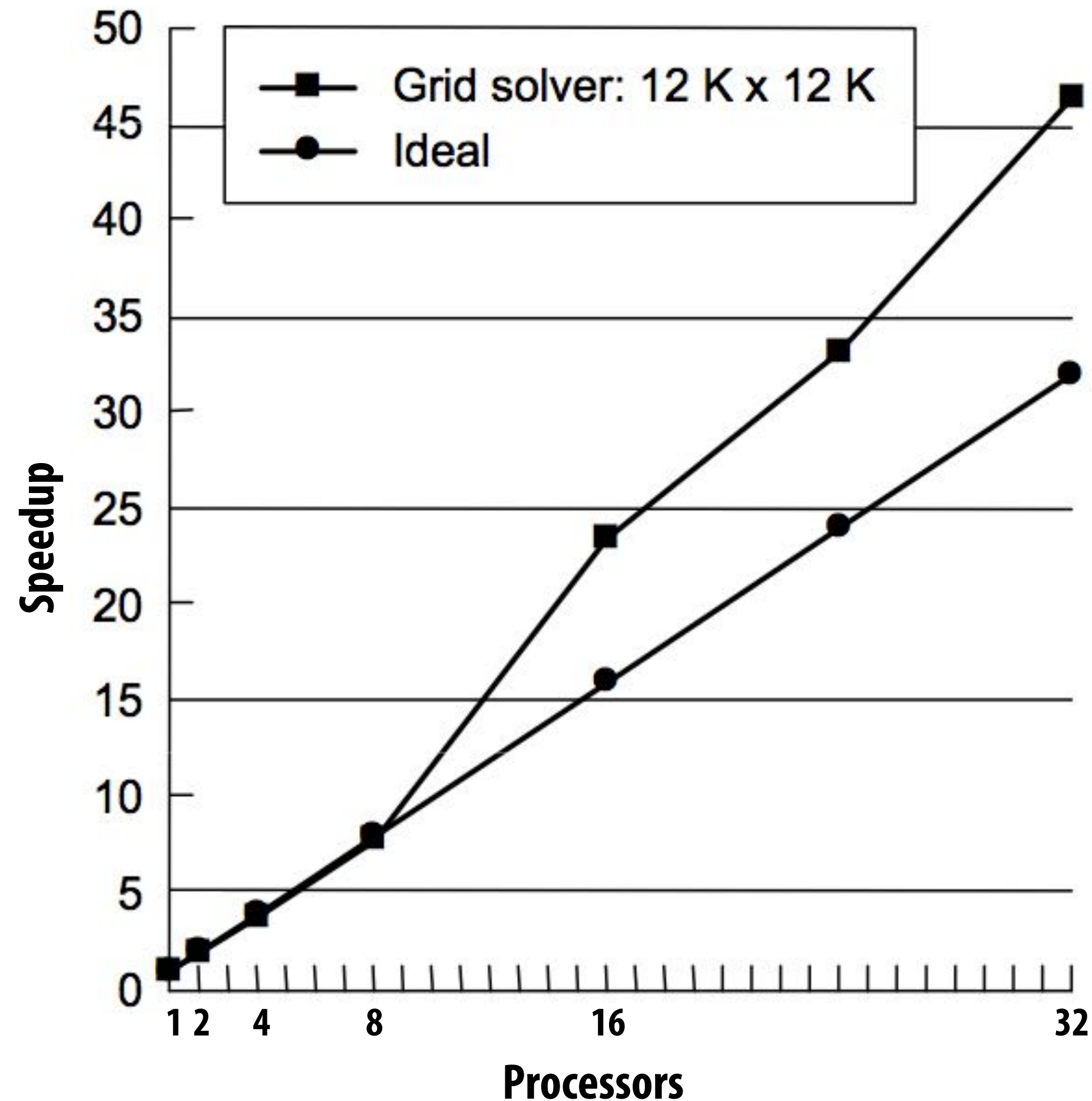


258 x 258 grid on 32 processors: ~ 310 grid cells per processor

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

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!)

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 more, rather than just compute the same problem faster)

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?**