(how to be l33t)

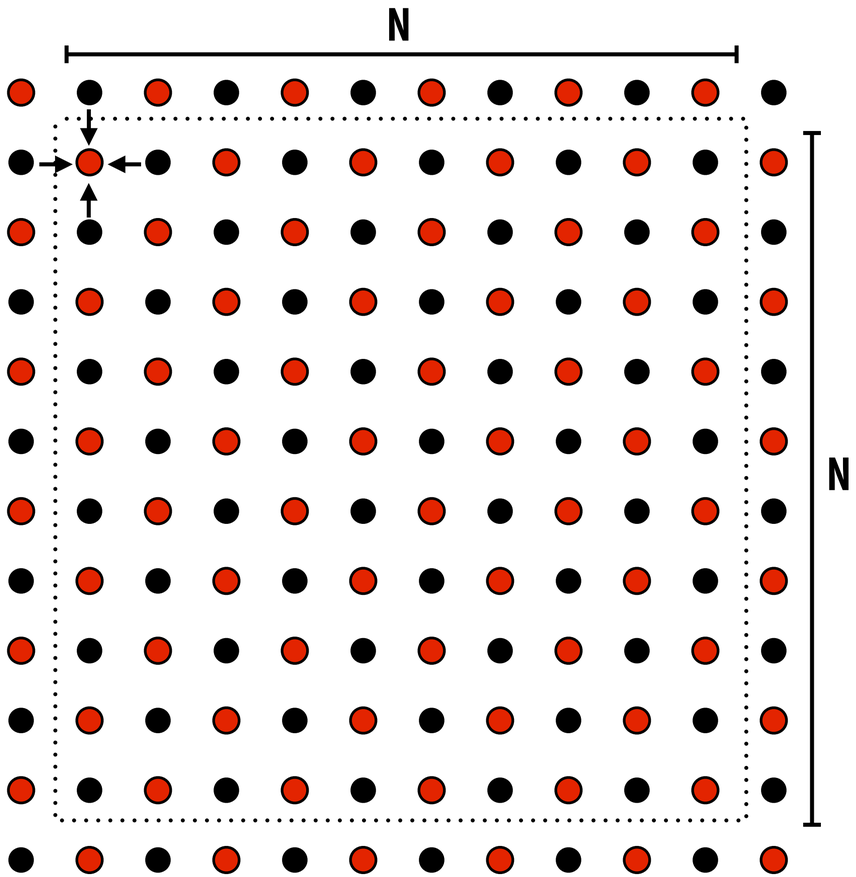
Lecture 6:

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

Parallel Computing

Stanford CS149, Fall 2022

Message passing expression of solver



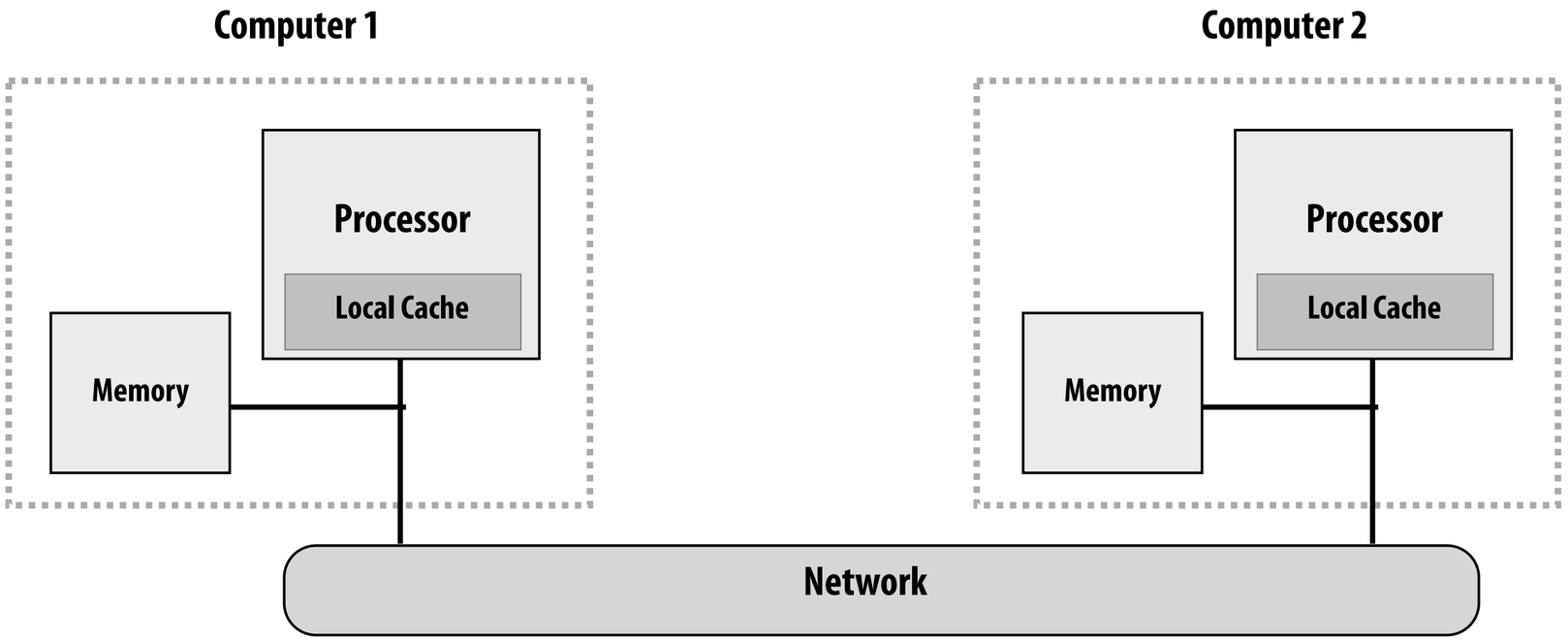
Update all red cells in parallel

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

Repeat until convergence

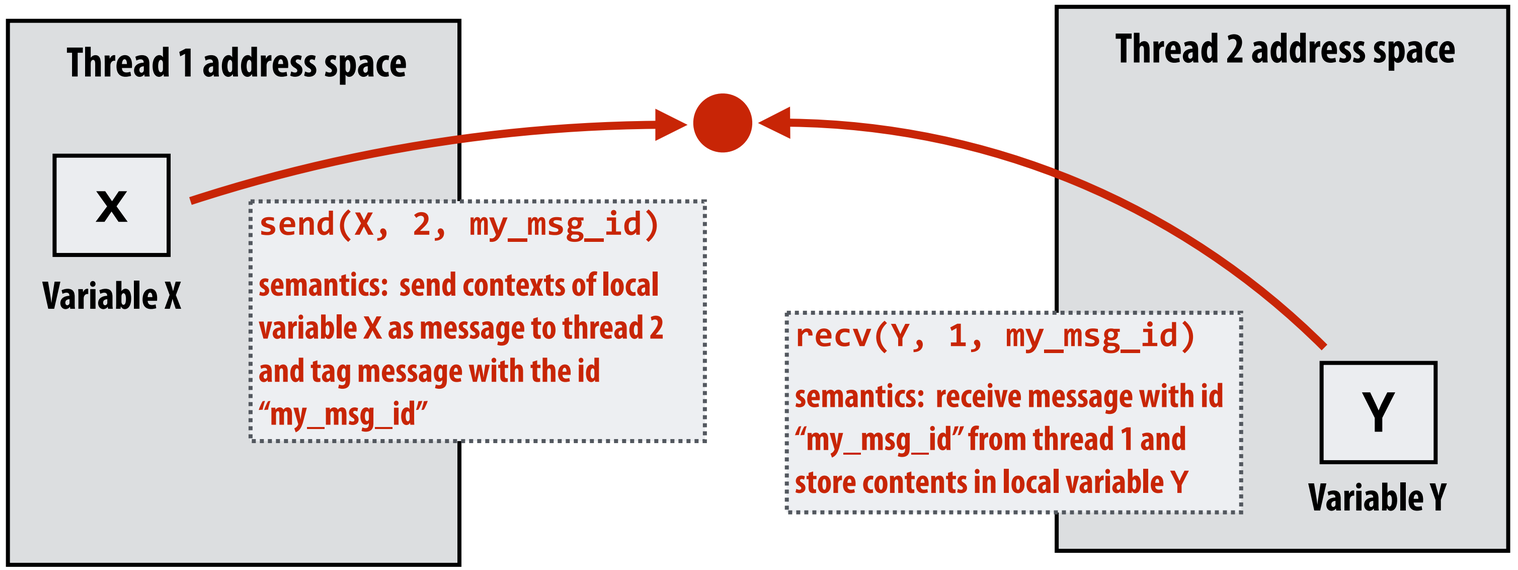
Let's think about expressing a parallel grid solver with communication via messages

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



Review: message passing model

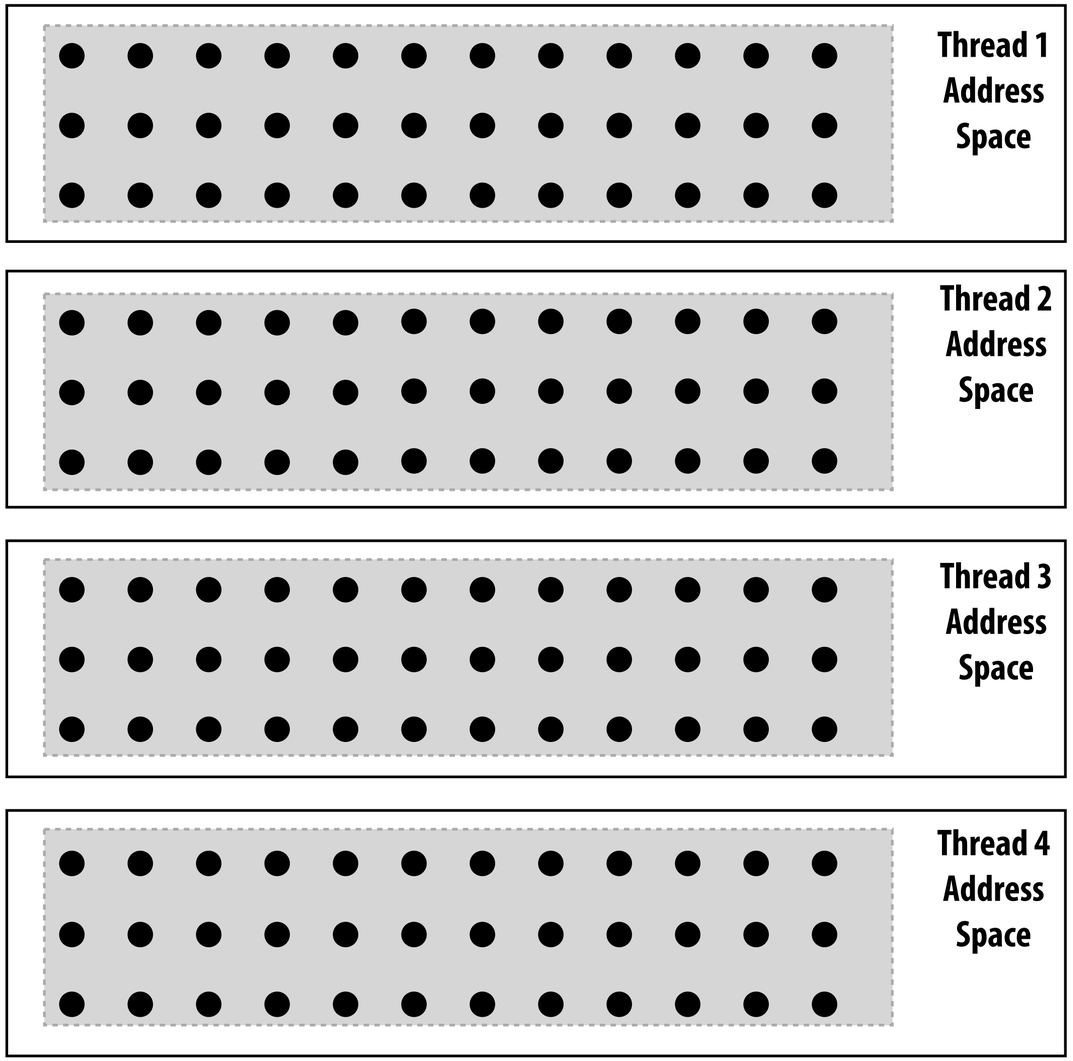
- Threads operate within their own private address spaces
- Threads communicate by sending/receiving messages
  - send: specifies recipient, buffer to be transmitted, and optional message identifier ("tag")
  - receive: sender, specifies buffer to store data, and optional message identifier
  - Sending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



(Communication operations shown in red)

Illustration adopted from Culler, Singh, Gupta

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



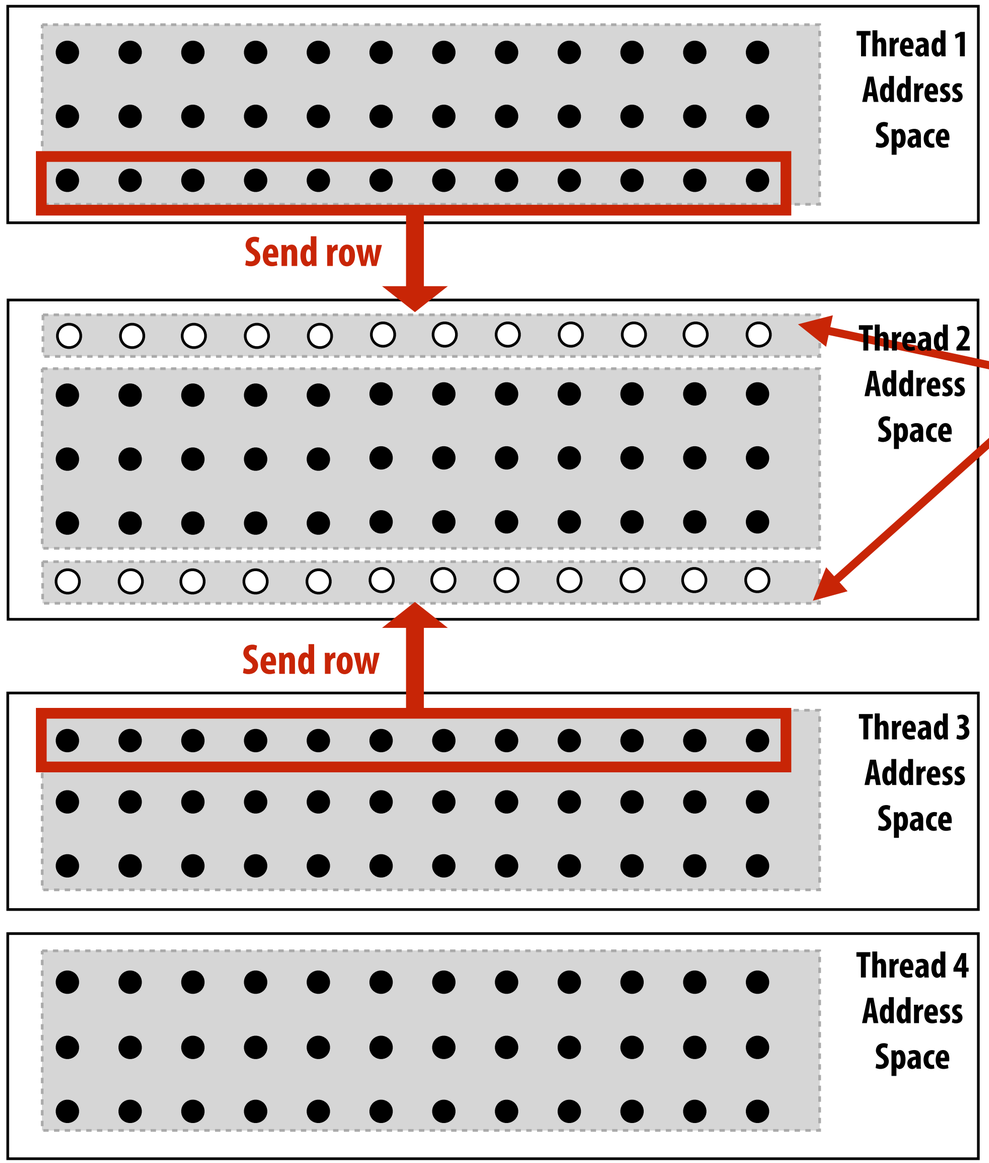
In this figure: four threads

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, MSD_ID_DONE);
    }
  }
}
```

Notes on message passing example

- **Computation**
  - Array indexing is relative to local address space

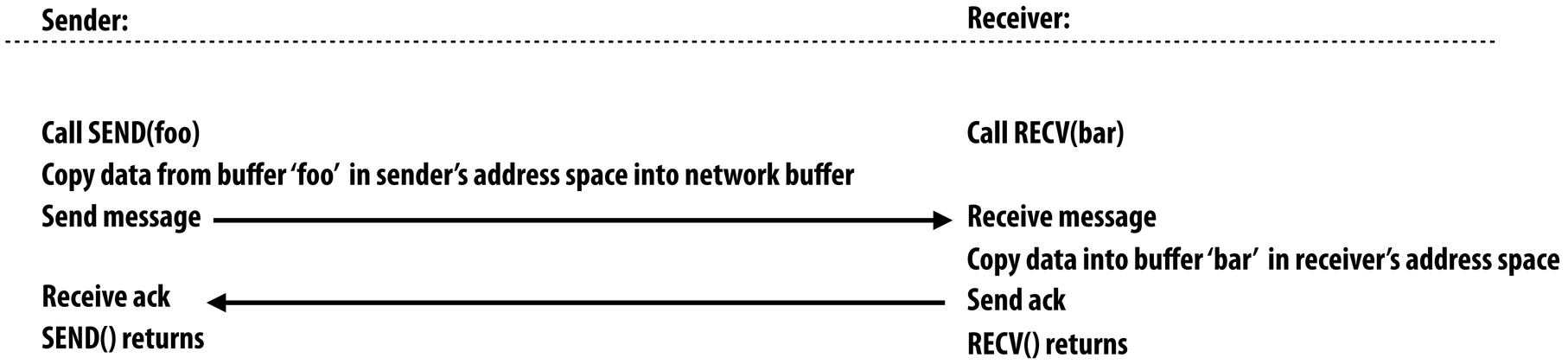
- **Communication:**
  - Performed by sending and receiving messages
  - Bulk transfer: communicate entire rows at a time

- **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



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

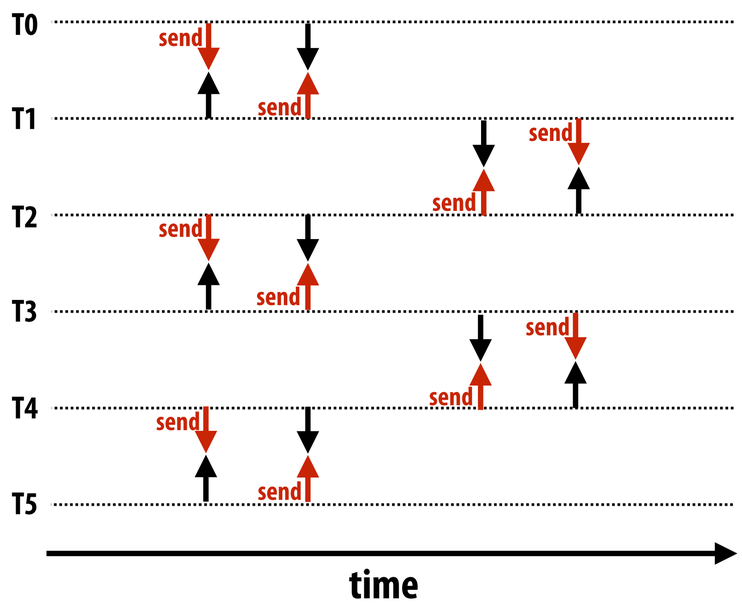
Why?

How can we fix it?

(while still using synchronous send/recv)

Message passing solver (fixed to avoid deadlock)

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

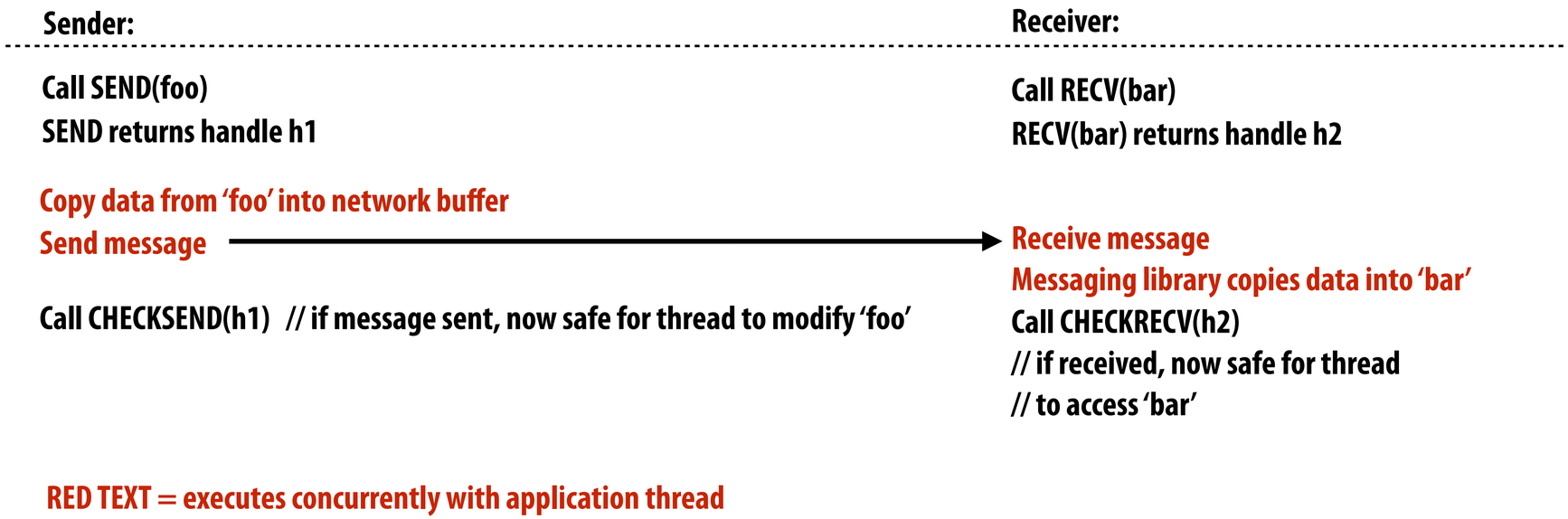



```
int N;
int tid = get_thread_id();
int rows_per_thread = N / get_num_threads();

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(); 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);
    }
  }
}
```

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



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

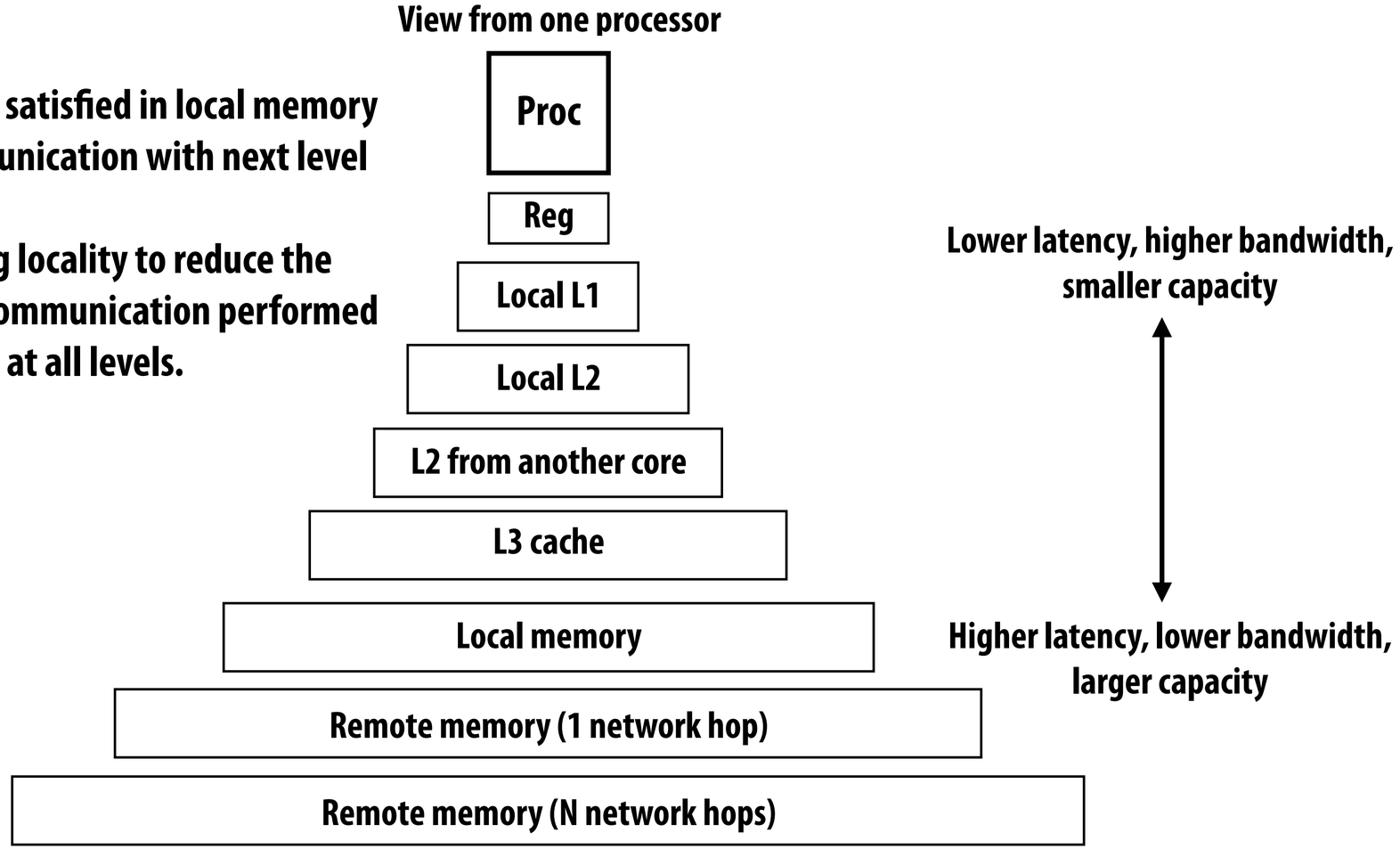
Think of a parallel system as an extended memory hierarchy

I want you to think of "communication" generally:

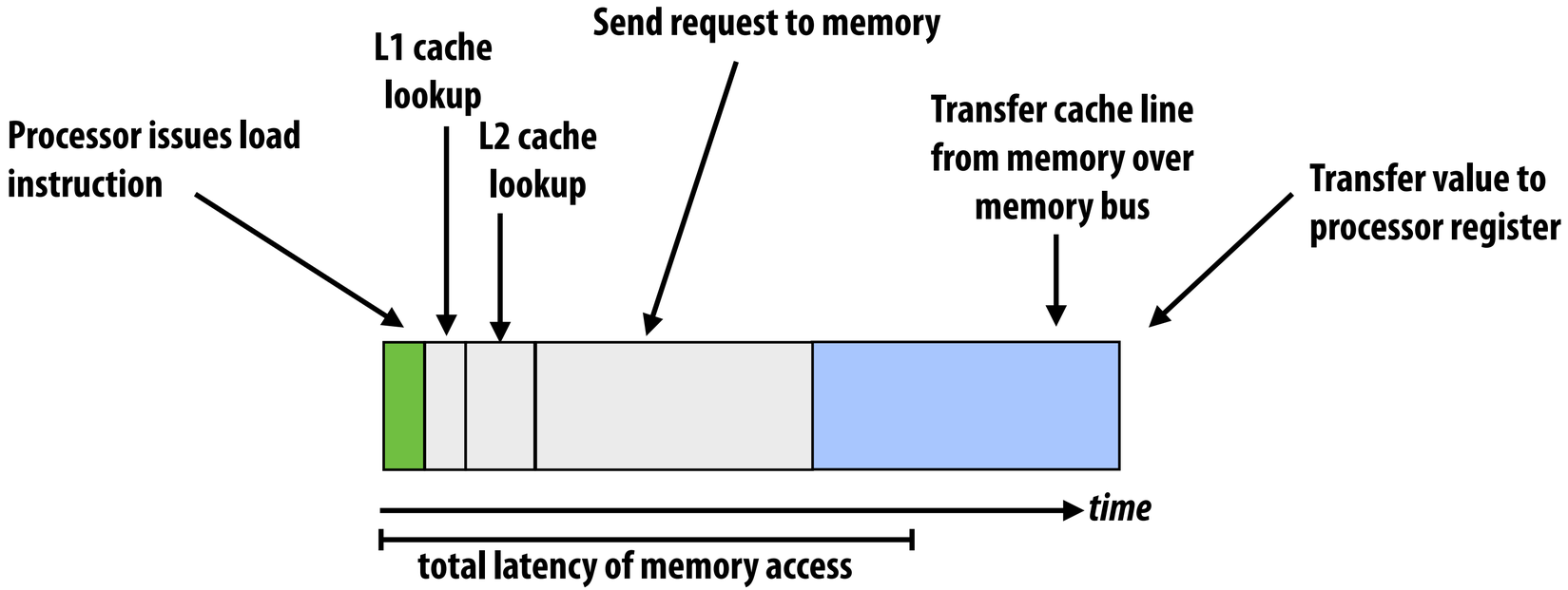
- Communication between a processor and its cache
- Communication between processor and memory (e.g., memory on same machine)
- Communication between processor and a remote memory
  (e.g., memory on another node in the cluster, accessed by sending a network message)

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.

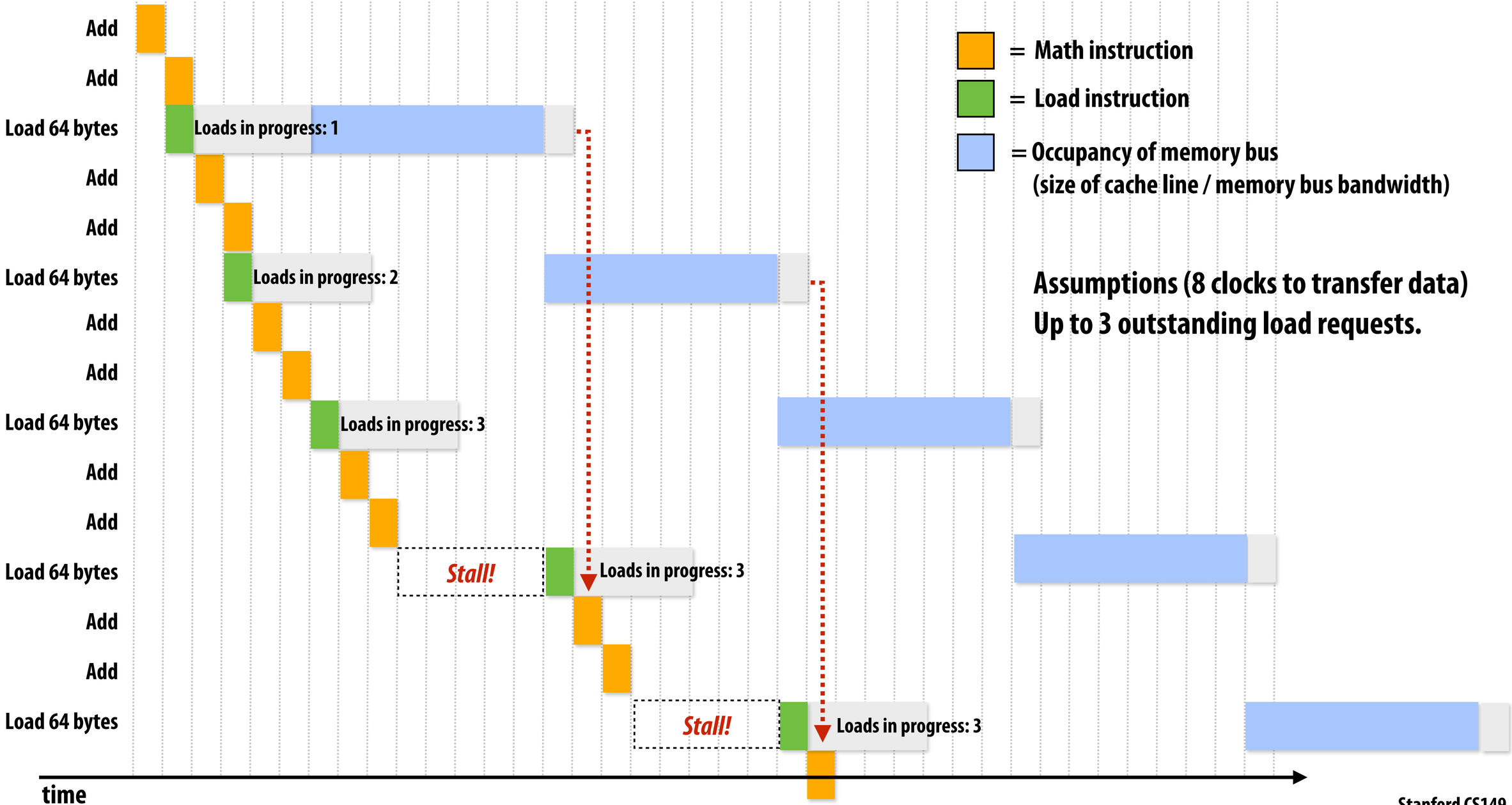


One example: CPU to memory communication

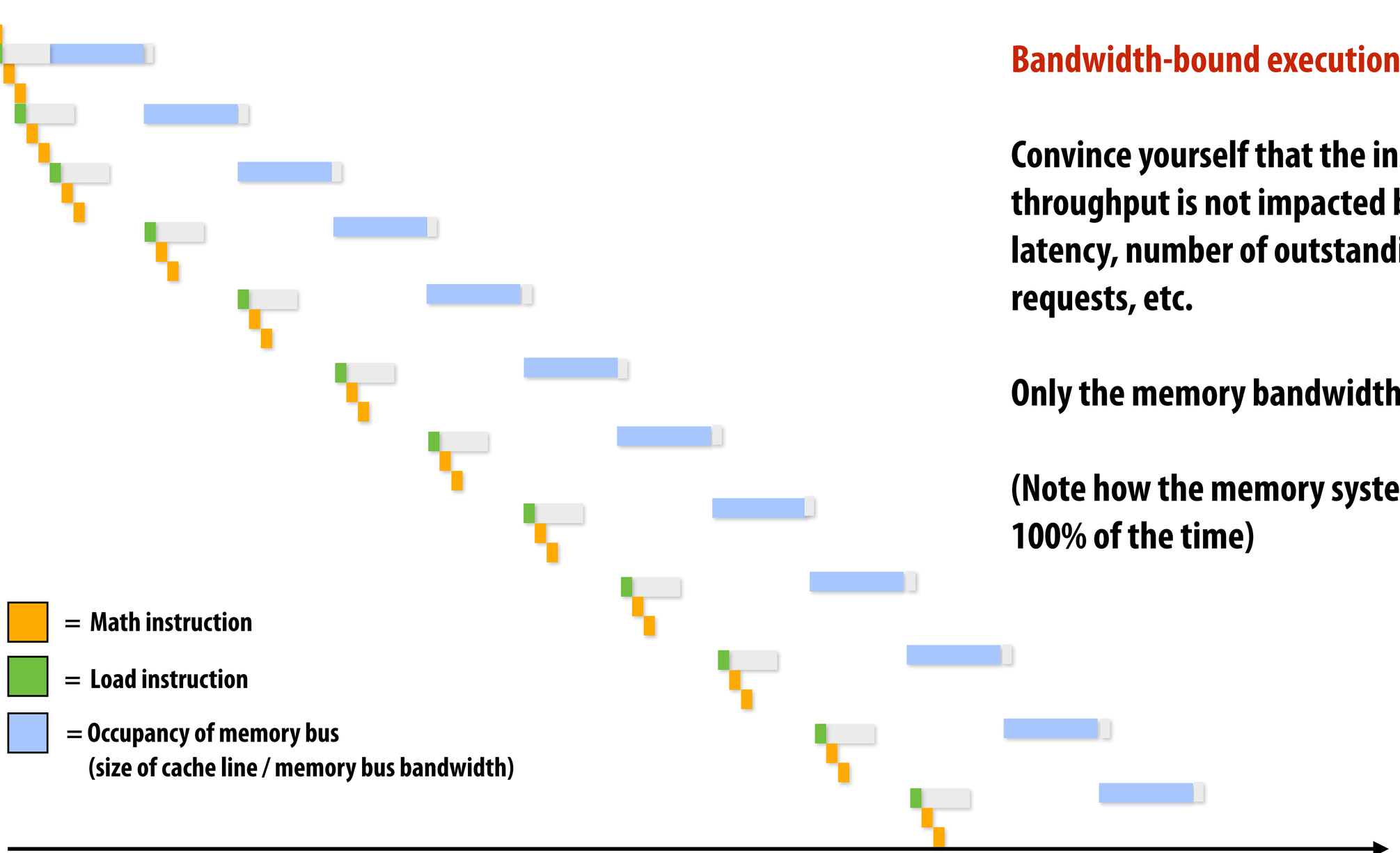


= Time to send cache line over memory bus

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



Rate of math instructions limited by available bandwidth



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?

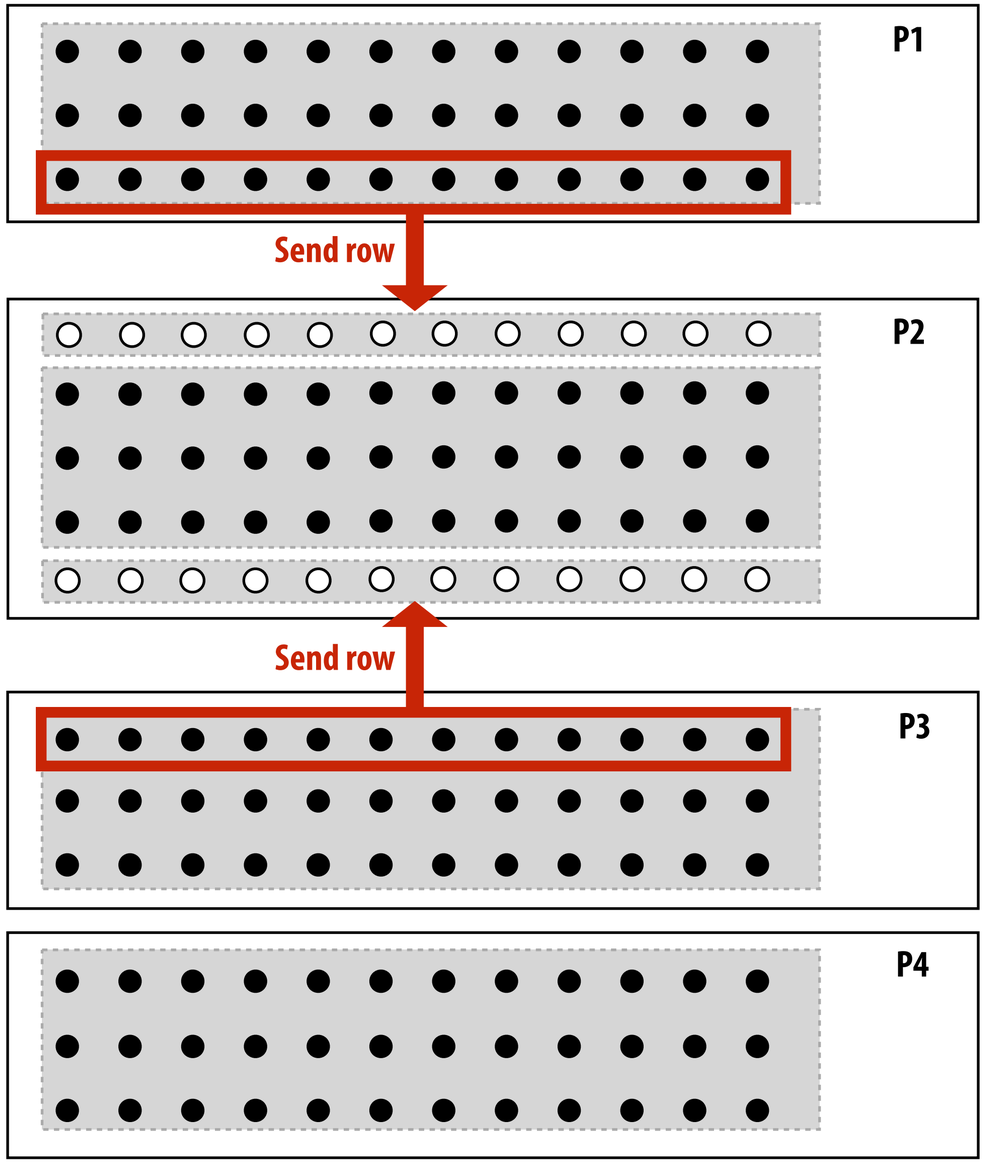
Communication-to-computation ratio

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

- 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



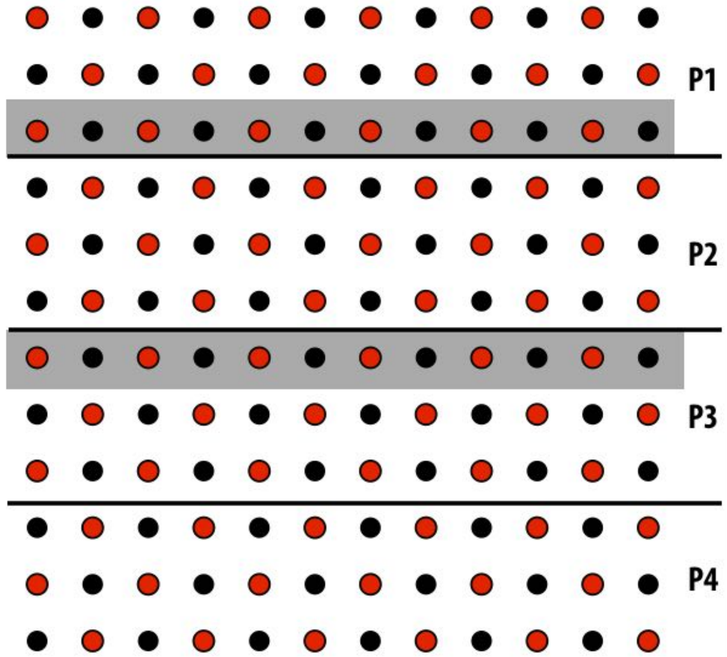
Communication that must occur in a parallel algorithm. The communication is fundamental to the algorithm.

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

Reducing inherent communication

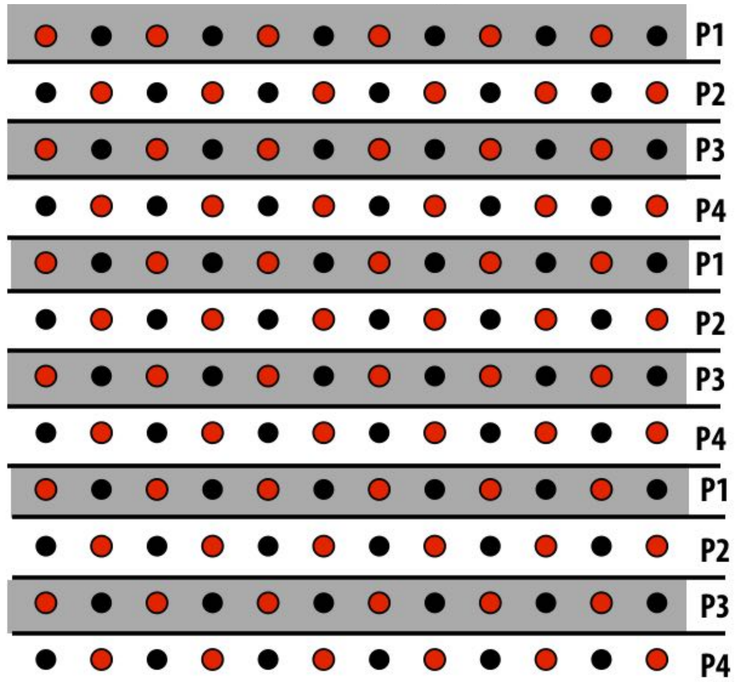
Good assignment decisions can reduce inherent communication (increase arithmetic intensity)

1D blocked assignment: N x N grid



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

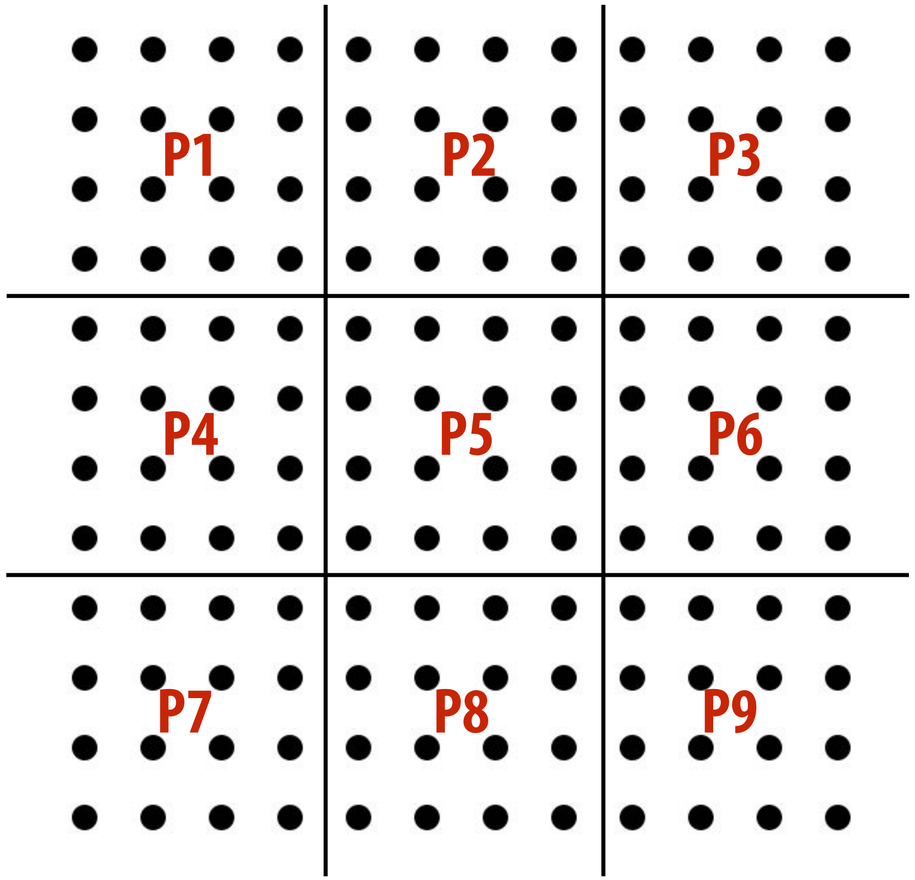
1D interleaved assignment: N x N grid



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

Reducing inherent communication

2D blocked assignment: N x N grid



N^2 elements

P processors

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

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

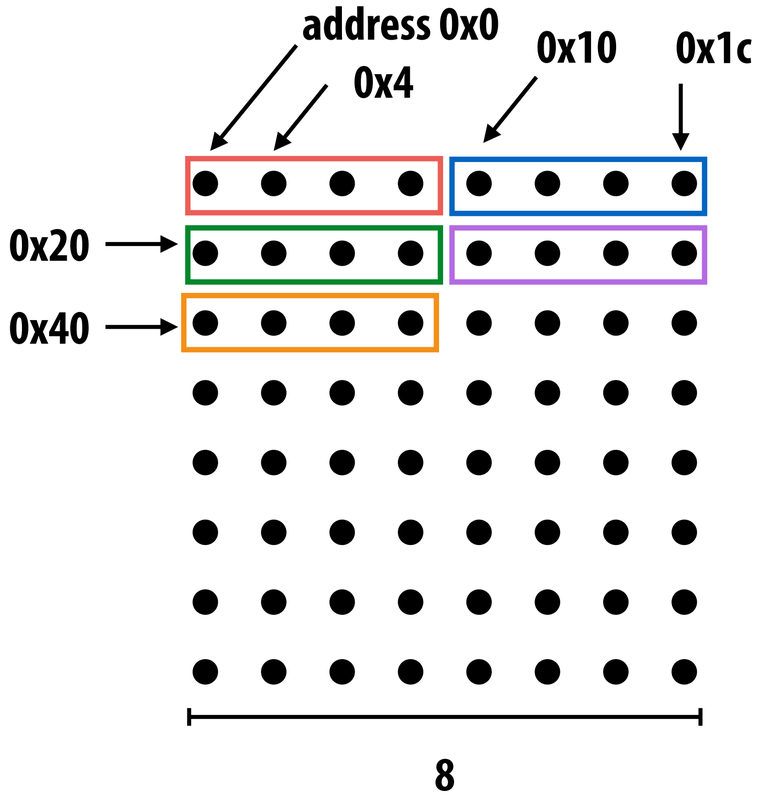
Assignment captures 2D locality of algorithm

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)

Cache review



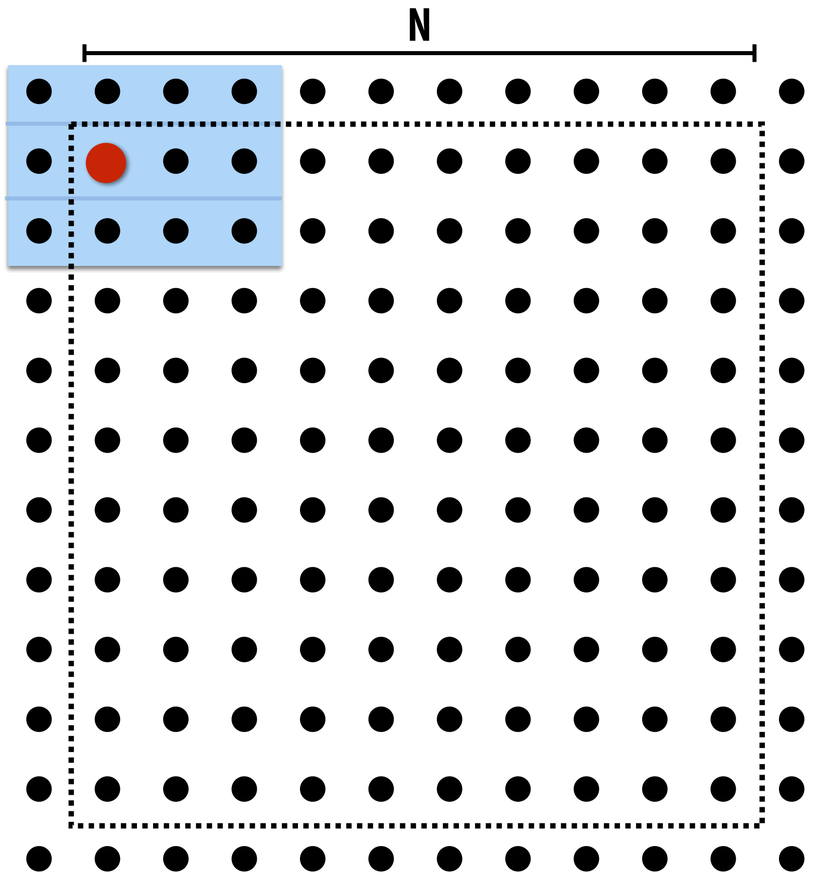
Consider 4-byte elements

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



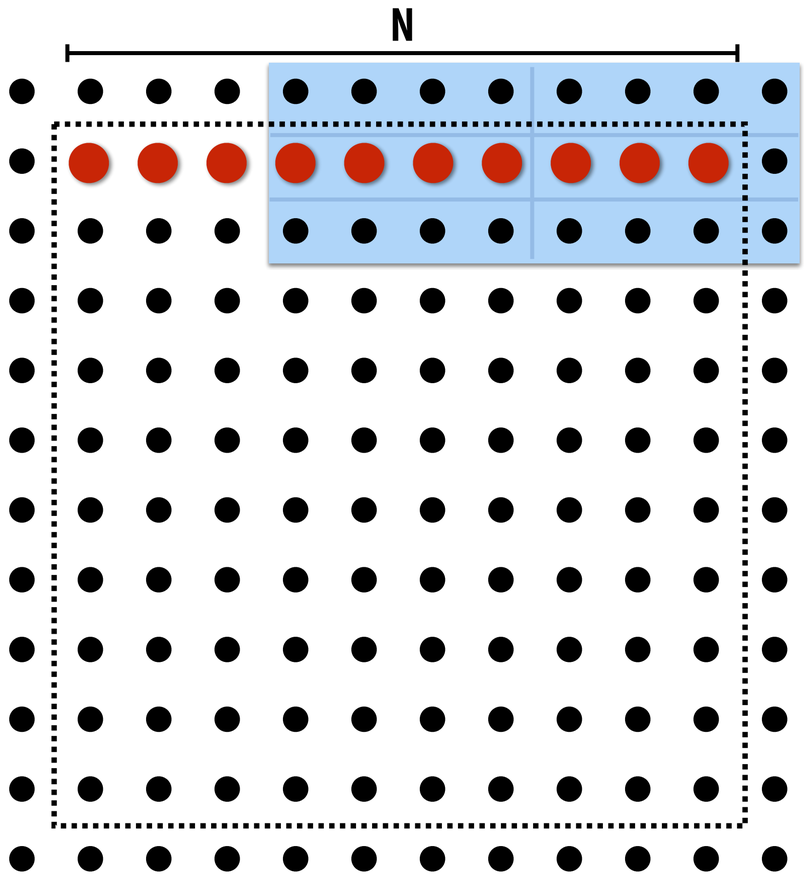
Assume row-major grid layout.

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



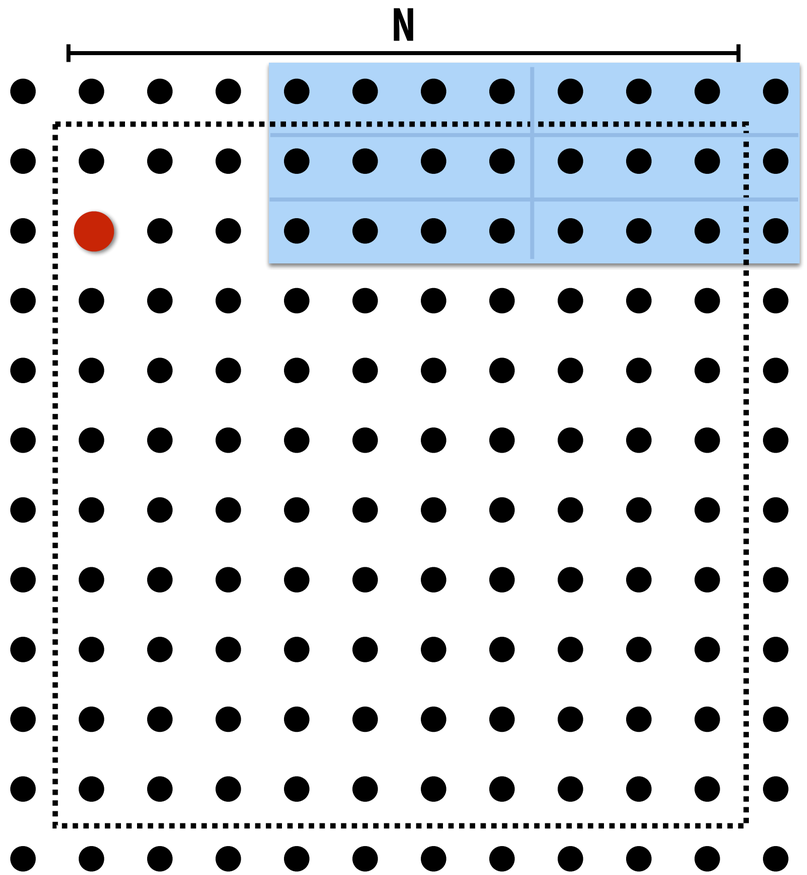
Assume row-major grid layout.

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)

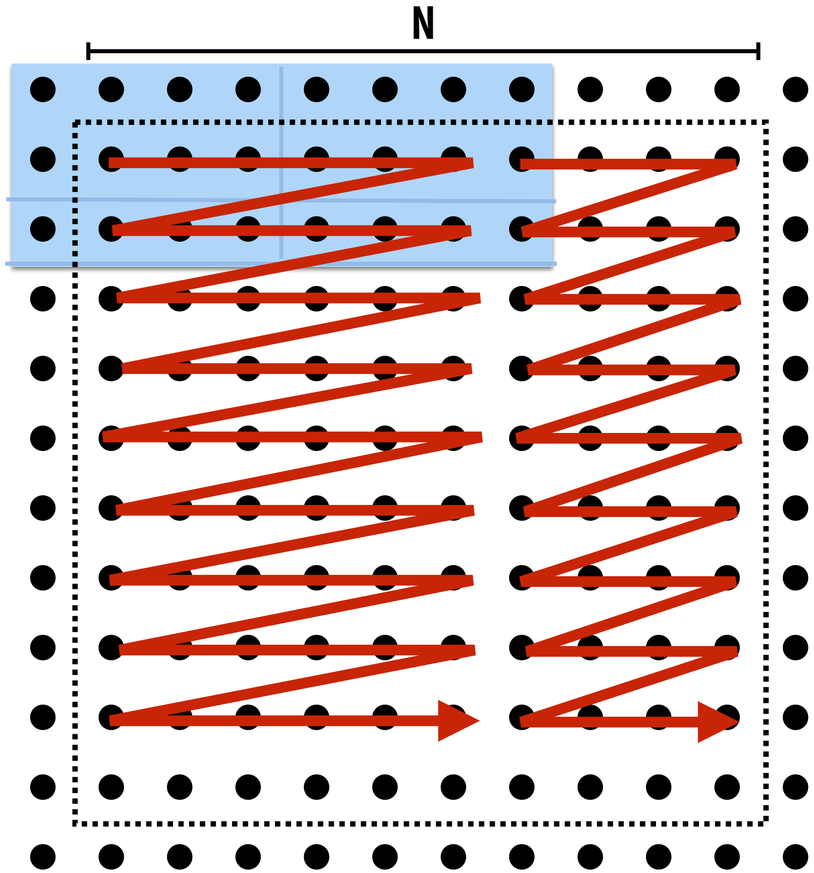
- System operation might result in unnecessary communication:
  - Program stores 16 consecutive 4-byte float values, and as a result the entire 64-byte cache line is loaded from memory, entirely overwritten, then subsequently stored to memory (2x overhead… load was unnecessary)

- 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

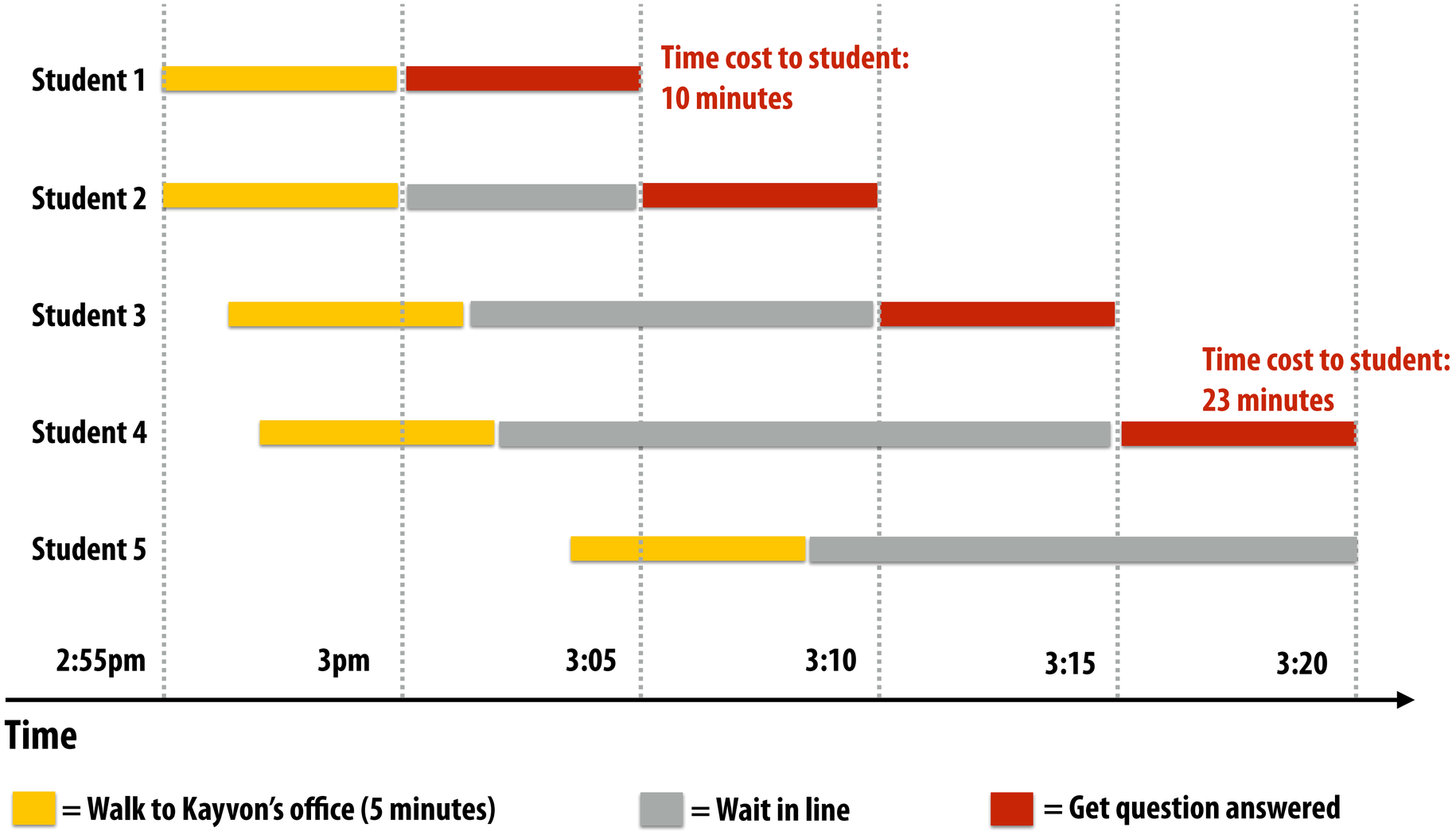
Example: office hours from 3-3:20pm (no appointments)

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

- Execution resource: Professor Kayvon

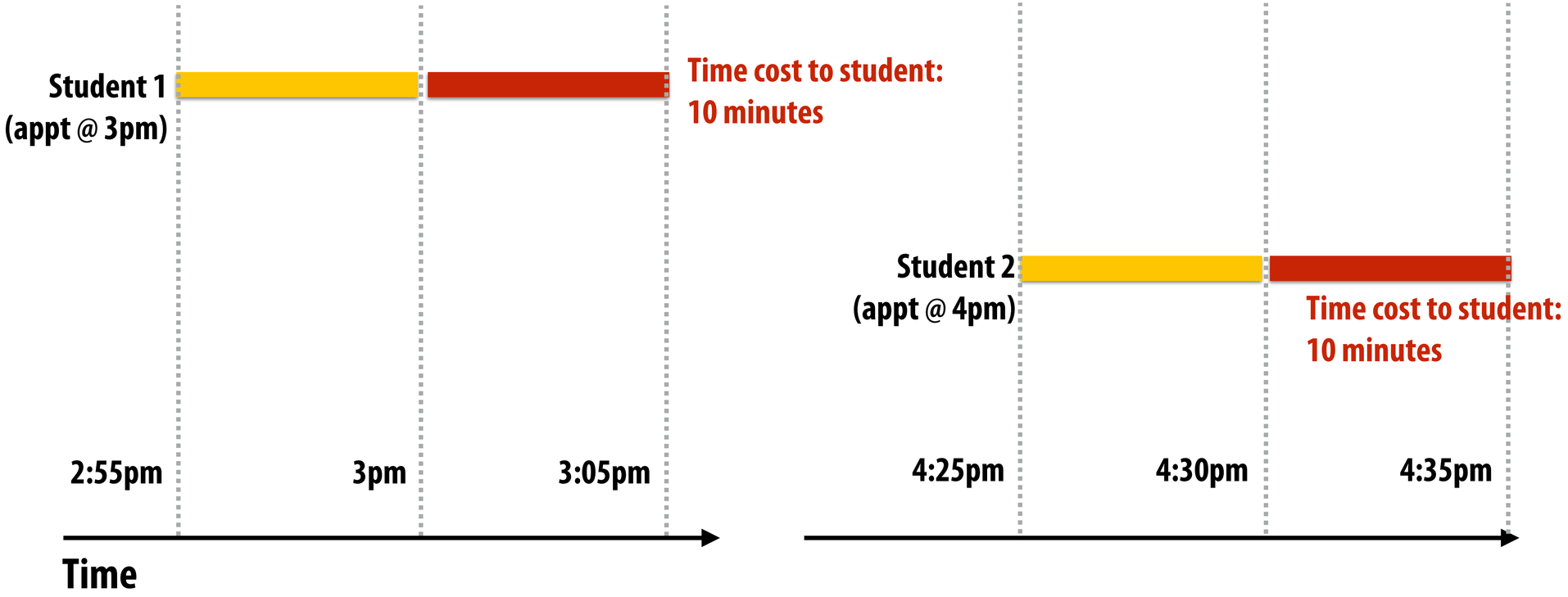
- Steps in operation:
  1. Student walks from Bytes Cafe to Kayvon's office (5 minutes)
  2. Student waits in line (if necessary)
  3. Student gets question answered with insightful answer (5 minutes)

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)

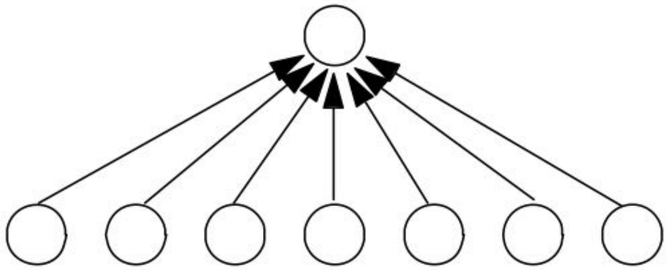


Contention

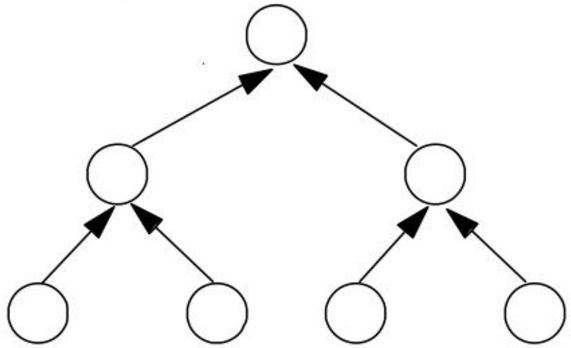
- A resource can perform operations at a given throughput (number of transactions per unit time)
  - Memory, communication links, servers, TA's at office hours, etc.

- 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

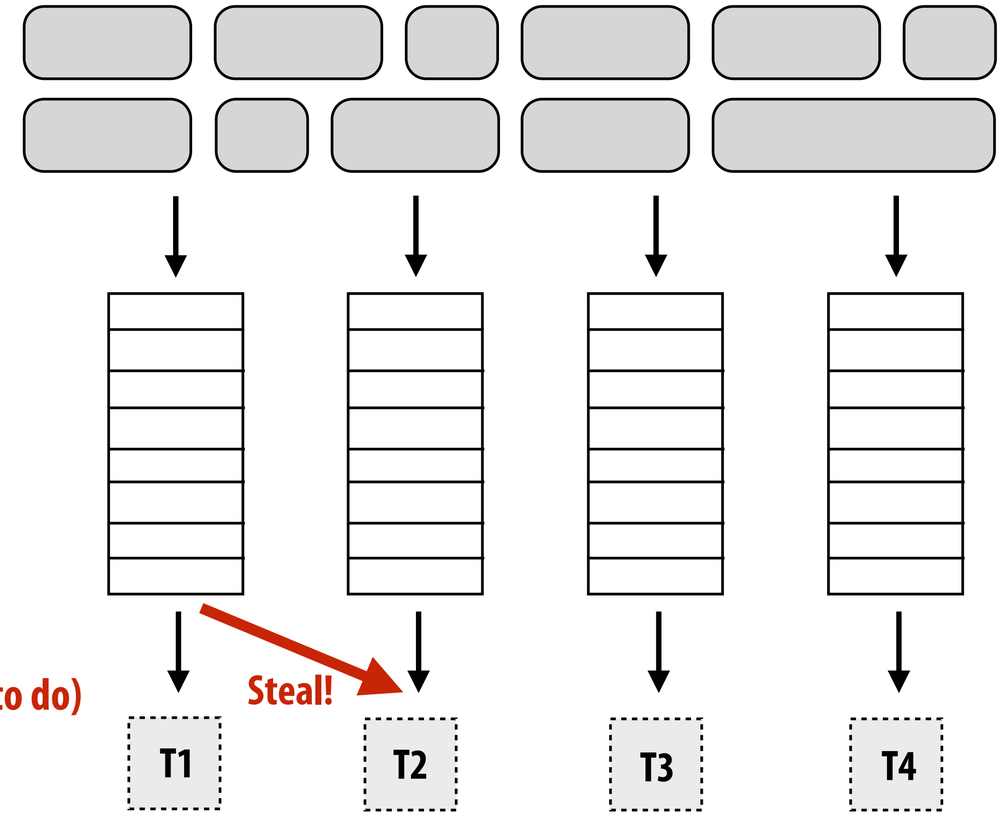
(contention in access to single shared work queue)

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

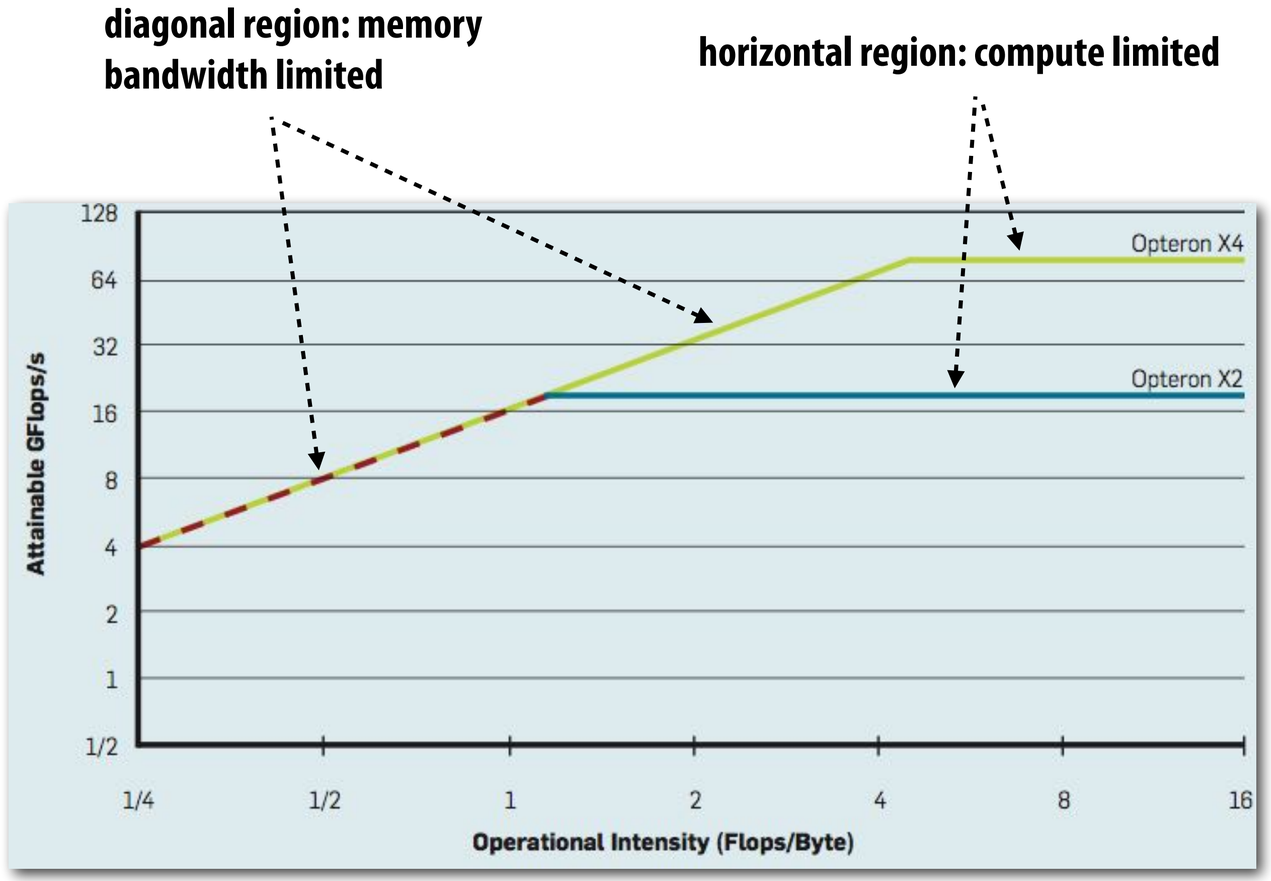


Figure credit: Williams et al. 2009

Roofline model: optimization regions

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

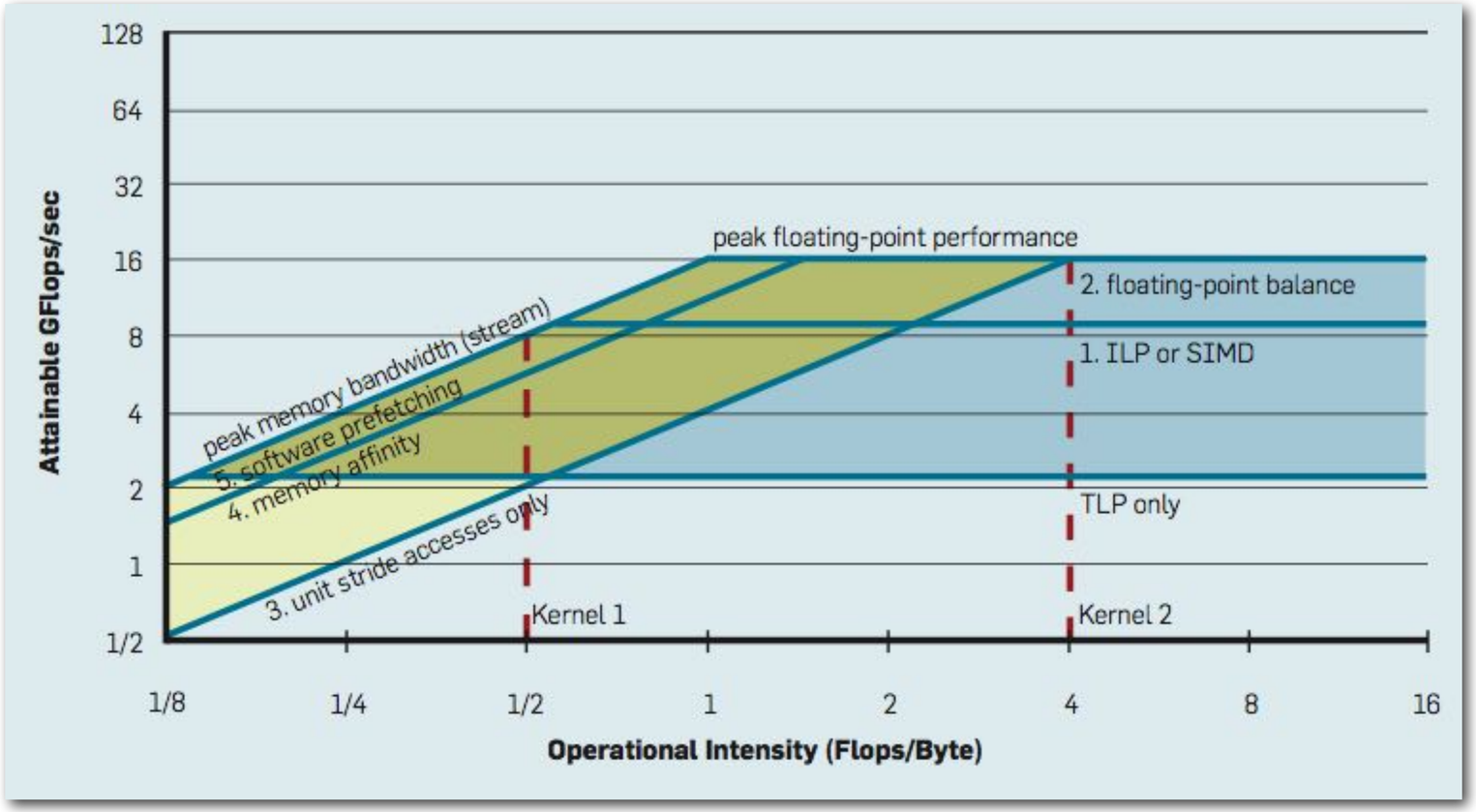


Figure credit: Williams et al. 2009

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)

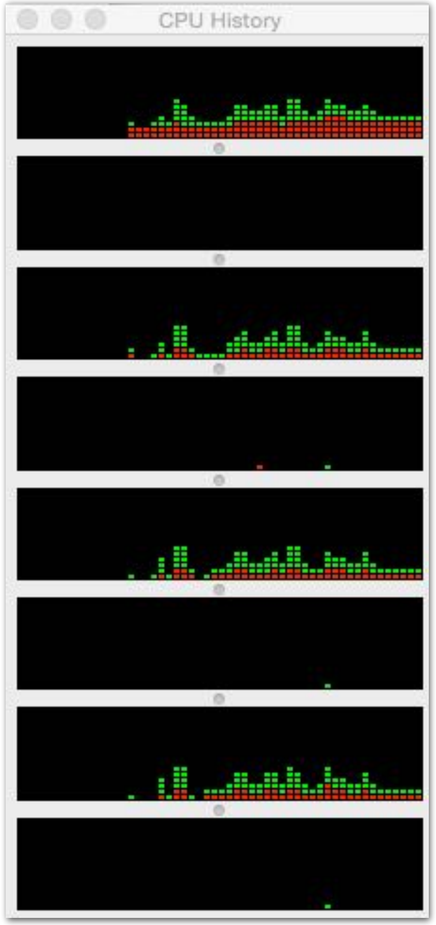
Remove all atomic operations or locks

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.

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?