Lecture 4:

Parallel Programming Models + Parallel Programming Basics

Parallel Computing

Stanford CS149, Fall 2022

REVIEW

Quiz: reviewing ISPC abstractions

```
export void ispc_sinx(
   uniform int N,
   uniform int terms,
   uniform float* x,
   uniform float* result)
{
   // assume N % programCount = 0
   for (uniform int i=0; i<N; i+=programCount)
   {
      int idx = i + programIndex;
      float value = x[idx];
      float numer = x[idx] * x[idx] * x[idx];
      uniform int denom = 6;  // 3!
      uniform int sign = -1;

      for (uniform int j=1; j<=terms; j++)
      {
         value += sign * numer / denom
         numer *= x[idx] * x[idx];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }
      result[idx] = value;
   }
}
```

This is an ISPC function.

It contains two nested for loops

Consider one ISPC program instance.
Which iterations of the two loops are executed in parallel by the ISPC program instance?

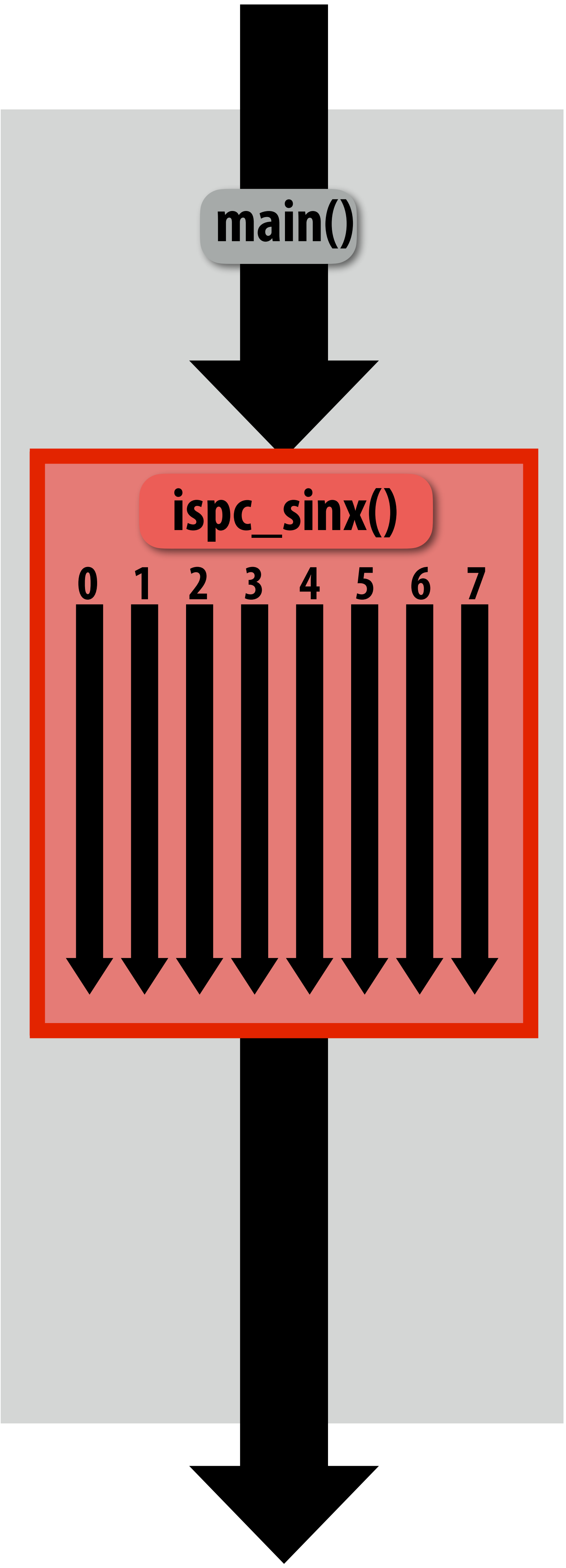
Hint: this is a trick question

Answer: none [or both] depending on how to think about it

Program instances (that run in parallel) are created when the ispc_sinx() ispc function is called

```
#include "sinx_ispc.h"

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
ispc_sinx(N, terms, x, result);
```

Each *ISPC program instance* executes the code in the function ispc_sinx serially.
(parallelism exists because there are multiple program instances, not because of parallelism in the code that defines an ispc function)



Sequential execution (C code)

Call to `ispc_sinx()`
Begin executing `programCount` instances of `ispc_sinx()`
(ISPC code)

`ispc_sinx()` returns.
Completion of ISPC program instances
Resume sequential execution

Sequential execution (C code)

Today's topics

- Three parallel programming abstractions (ways to think about the structure of parallel computation)
  - Shared address space
  - Message passing
  - Data parallel

- An example of writing and optimizing a program
  - Demonstrated in the shared address space and data parallel models

Programming models provide a way to think about the organization of parallel programs (by imposing structure)

- **Shared address space: very little structure to communication**
  - All threads can read and write to all shared variables
- **Message passing: communication is structured in the form of messages**
  - All communication occurs in the form of messages
  - Communication is explicit in source code—the sends and receives)
- **Data parallel structure: more rigid structure to computation**
  - Perform same function on elements of large collections

Shared address space model

Review: a program's memory address space

- A computer's memory is organized as a array of bytes
- Each byte is identified by its "address" in memory (its position in this array)

  (in this class we assume memory is byte-addressable)

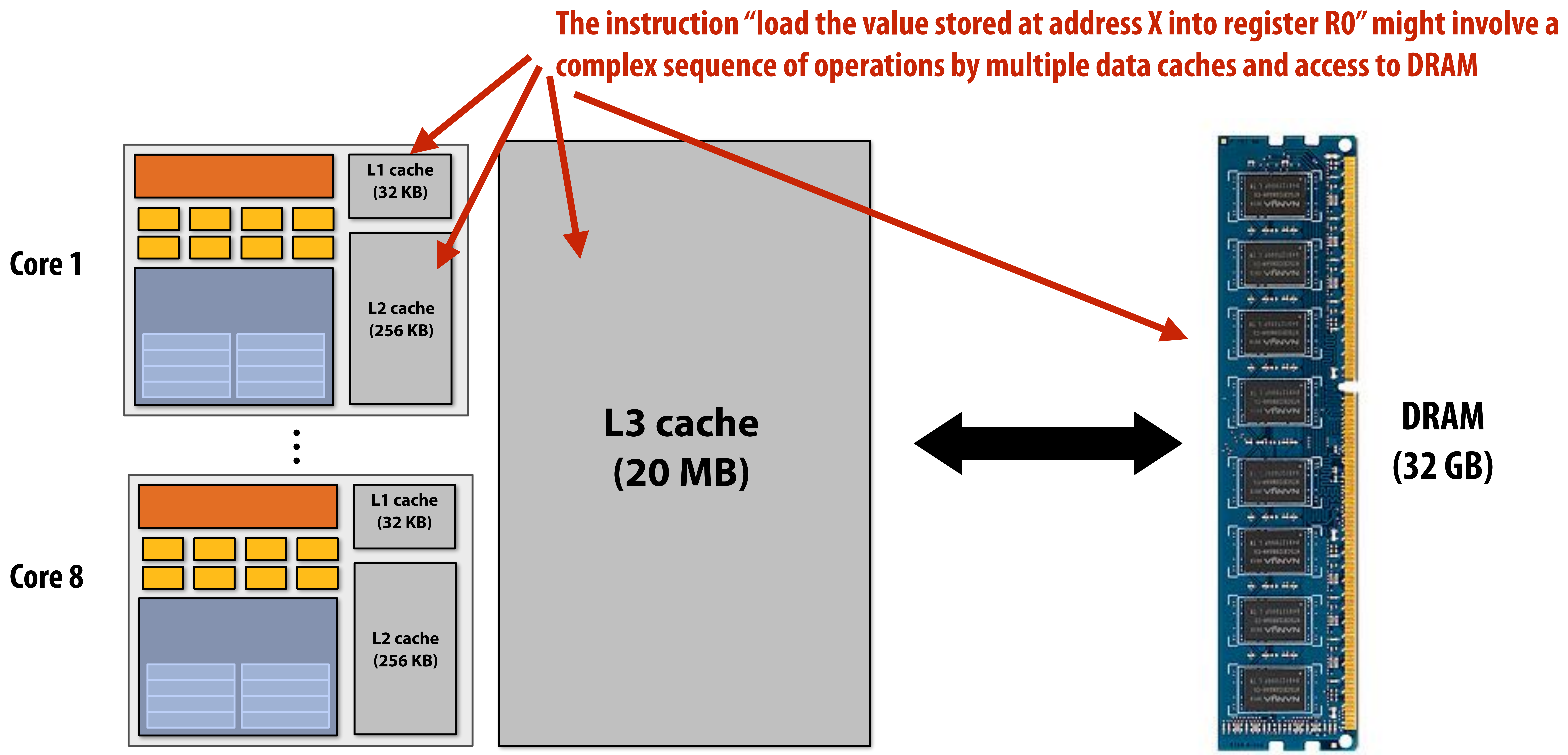
"The byte stored at address 0x8 has the value 32."

"The byte stored at address 0x10 (16) has the value 128."

In the illustration on the right, the program's memory address space is 32 bytes in size (so valid addresses range from 0x0 to 0x1F)

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
⋮	⋮
0x1F	0

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



Shared address space model (abstraction)

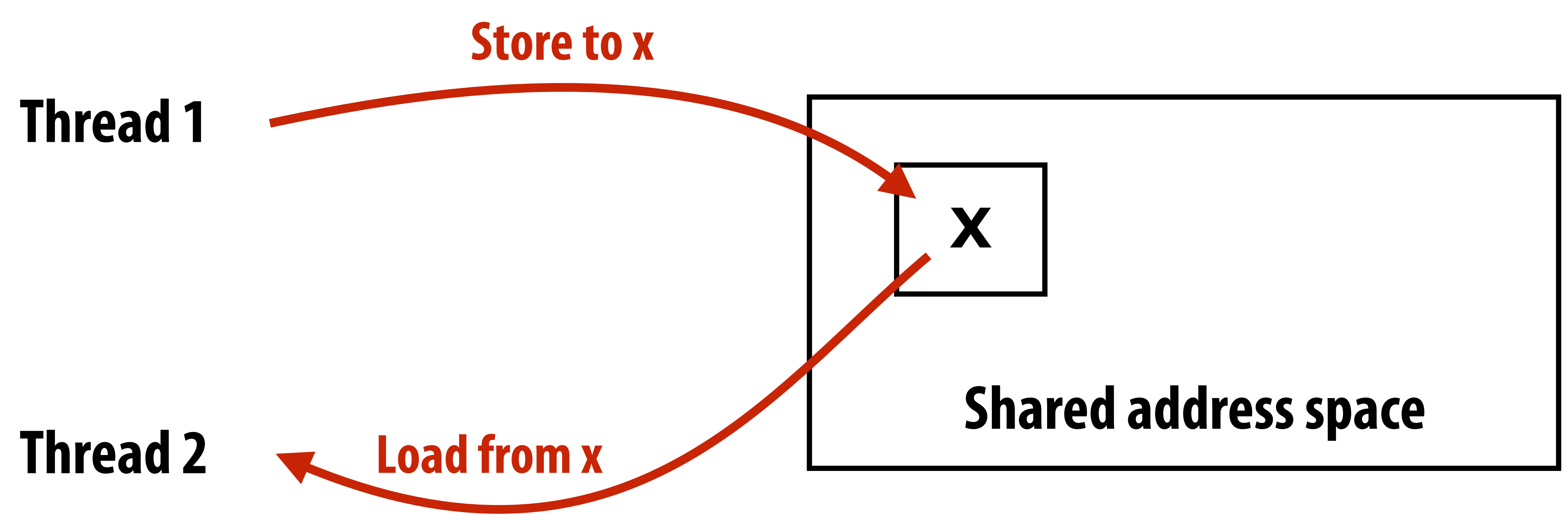
Threads communicate by reading/writing to locations in a shared address space (shared variables)

Thread 1:

```
int x = 0;
spawn_thread(foo, &x);

// write to address holding
// contents of variable x
x = 1;
```

Thread 2:

```
void foo(int* x) {

  // read from addr storing
  // contents of variable x
  while (x == 0) {}
  print x;
}
```



(Communication operations shown in red)

(Pseudocode provided in a fake C-like language for brevity.)

A common metaphor:
A shared address space is like a bulletin board

(Everyone can read/write)

Image credit:
https://thetab.com/us/stanford/2016/07/28/honest-packing-list-freshman-stanford-1278

Coordinating access to shared variables with synchronization

Thread 1:

```
int x = 0;
Lock my_lock;

spawn_thread(foo, &x, &my_lock);


mylock.lock();
x++;
mylock.unlock();
```

Thread 2:

```
void foo(int* x, Lock* my_lock) {
  my_lock->lock();
  x++;
  my_lock->unlock();

  print(x);
}
```

Review: why do we need mutual exclusion?

- Each thread executes:
  - Load the value of variable x from a location in memory into register r1
    (this stores a copy of the value in memory in the register)
  - Add the contents of register r2 to register r1
  - Store the value of register r1 into the address storing the program variable x
- One possible interleaving: (let starting value of x=0, r2=1)

T1	T2	
`r1 ← x`		`T1 reads value 0`
	`r1 ← x`	`T2 reads value 0`
`r1 ← r1 + r2`		`T1 sets value of its r1 to 1`
	`r1 ← r1 + r2`	`T2 sets value of its r1 to 1`
`X ← r1`		`T1 stores 1 to address of x`
	`X ← r1`	`T2 stores 1 to address of x`

- Need this set of three instructions must be "atomic"

Examples of mechanisms for preserving atomicity

- **Lock/unlock mutex around a critical section**

```
mylock.lock();
// critical section
mylock.unlock();
```

- **Some languages have first-class support for atomicity of code blocks**

```
atomic {
  // critical section
}
```

- **Intrinsics for hardware-supported atomic read-modify-write operations**

```
atomicAdd(x, 10);
```

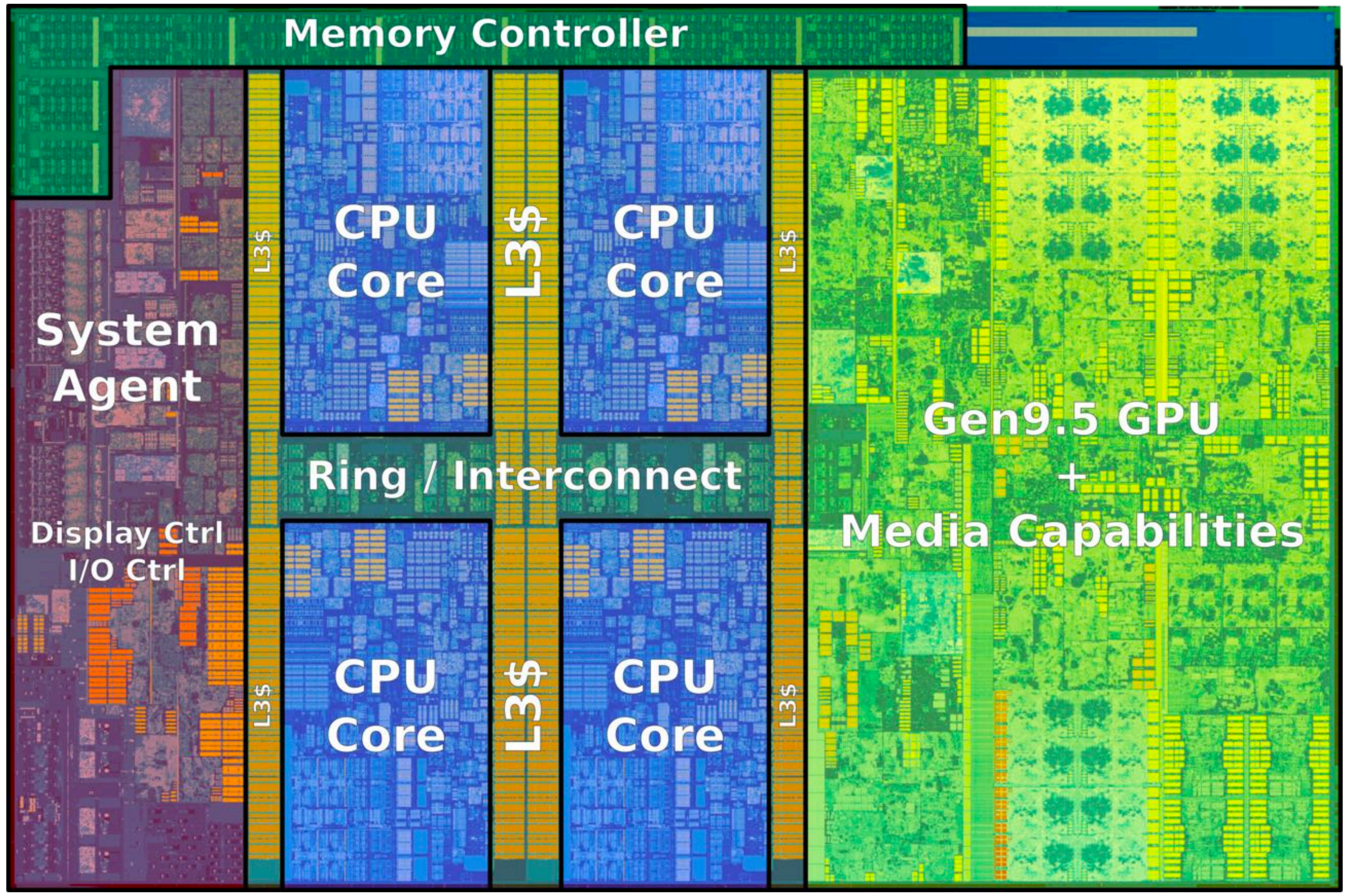
Review: shared address space model

- Threads communicate by:
  - Reading/writing to shared variables in a shared address space
    - Communication between threads is implicit in memory loads/stores
  - Manipulating synchronization primitives
    - e.g., ensuring mutual exclusion via use of locks

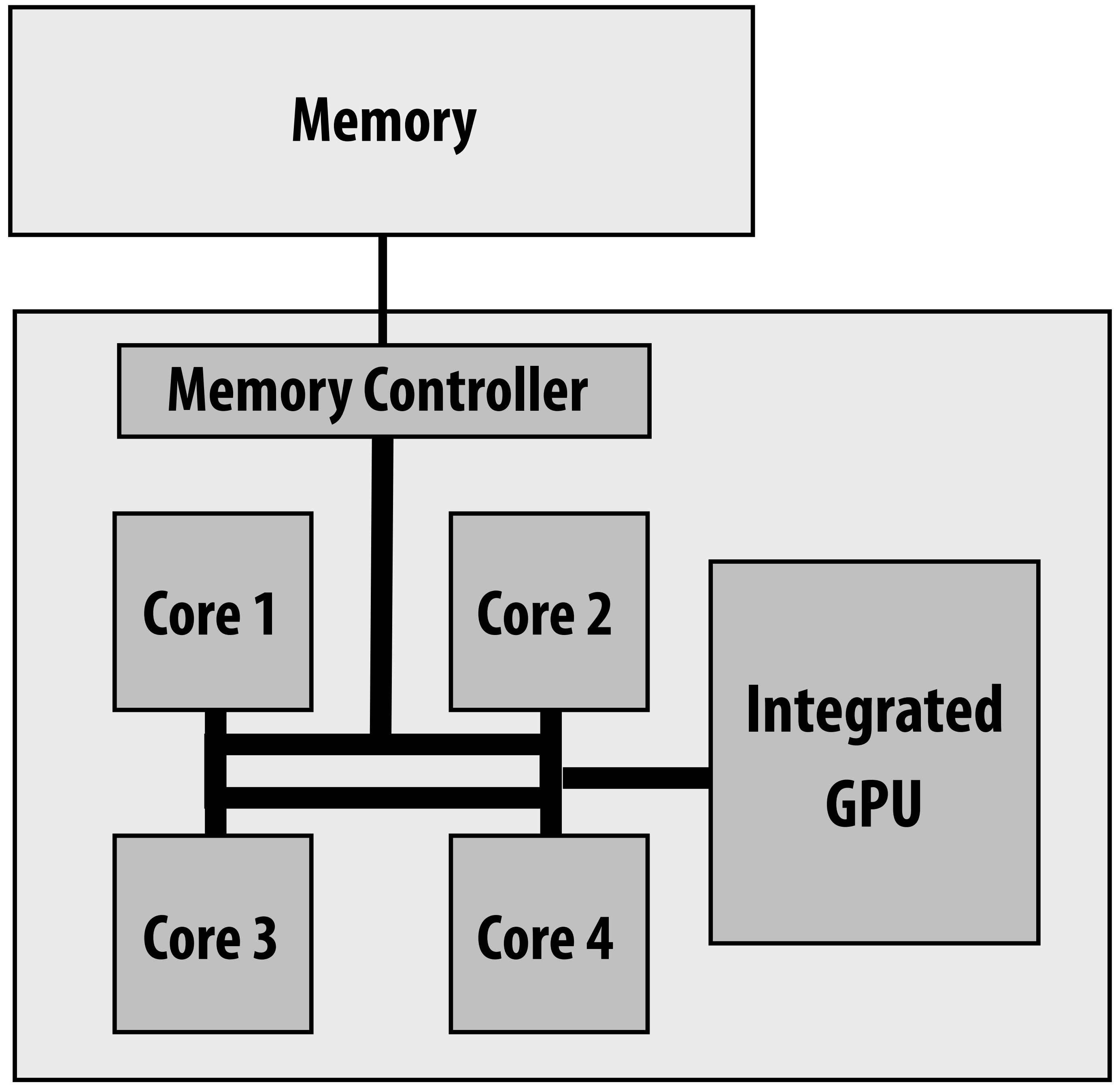
- This is a natural extension of sequential programming
  - In fact, all our discussions in class have assumed a shared address space so far!

Shared address space hardware architecture

Any processor can directly reference any memory location



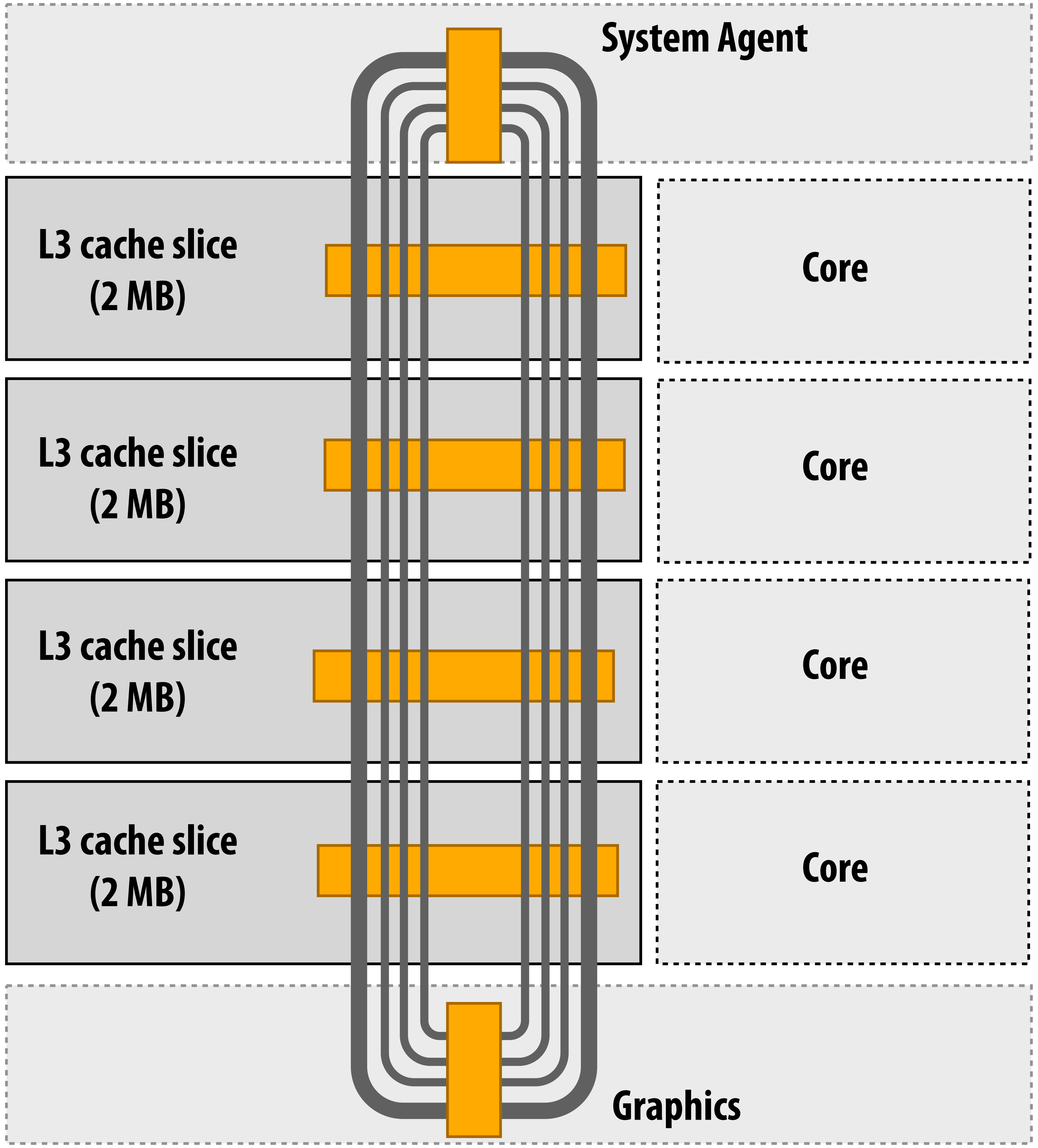
Example: Intel Core i7 processor (Kaby Lake)



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

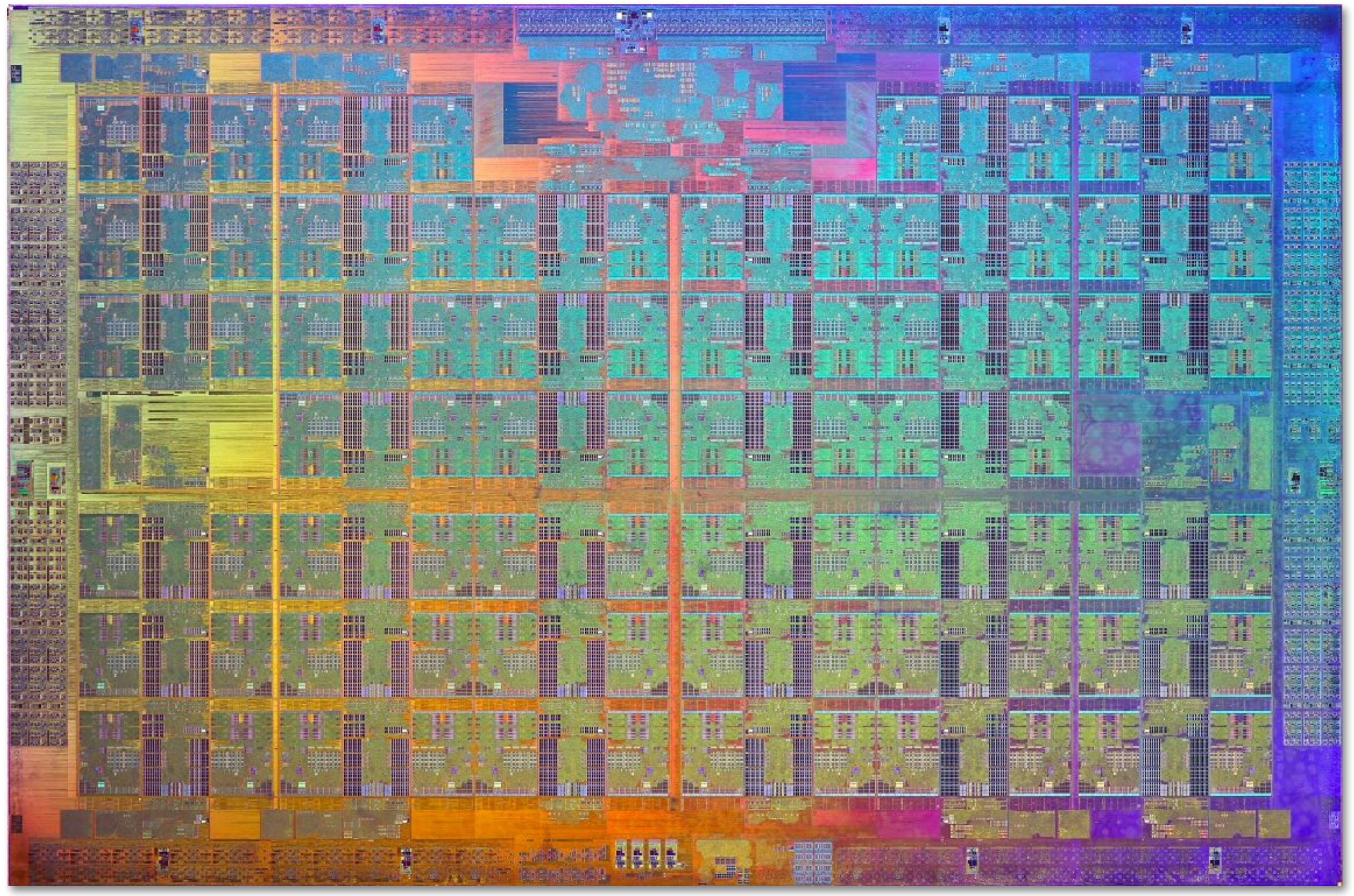
Intel's ring interconnect

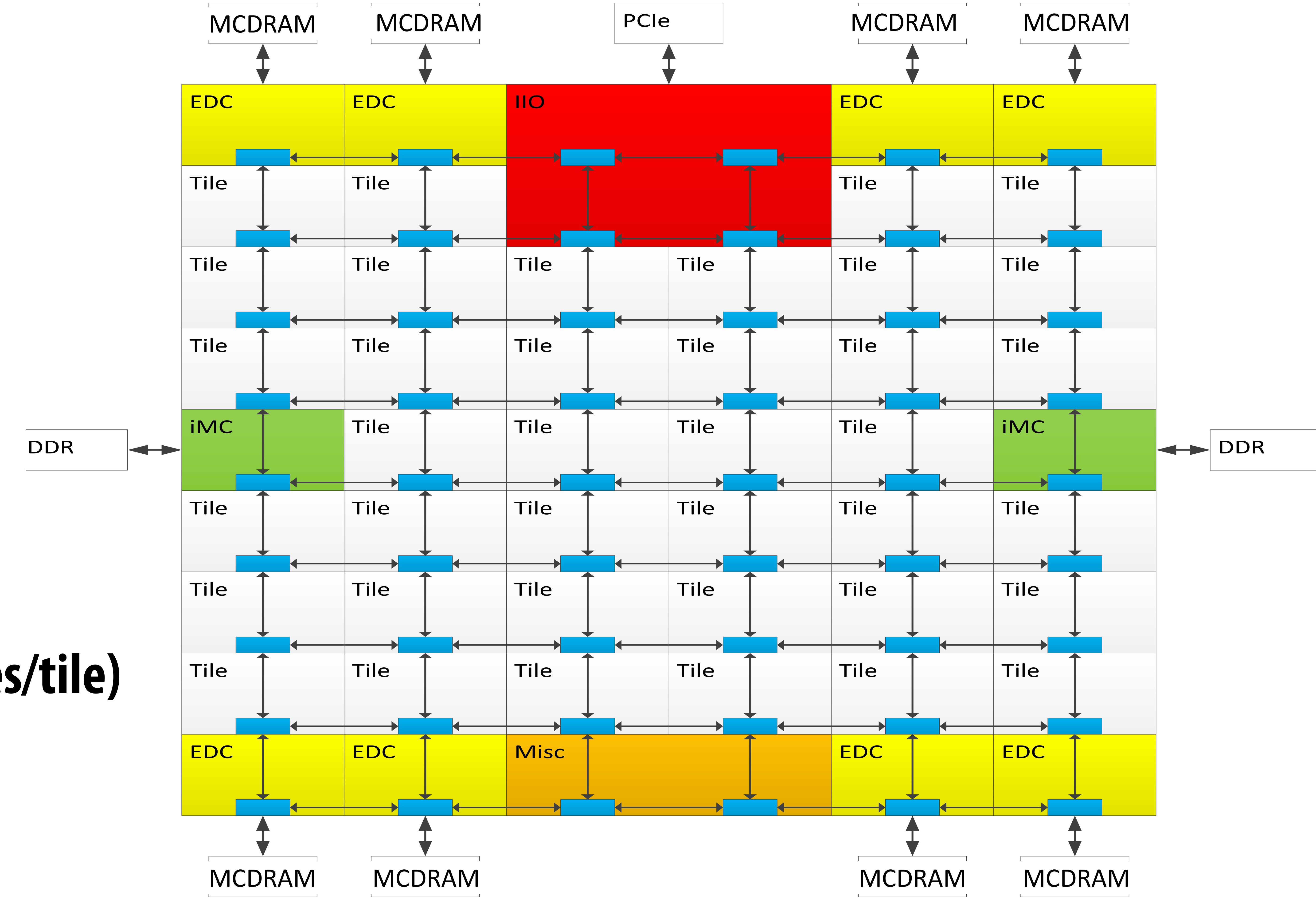
Introduced in Sandy Bridge microarchitecture



- 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

Intel Xeon Phi (Knights Landing)

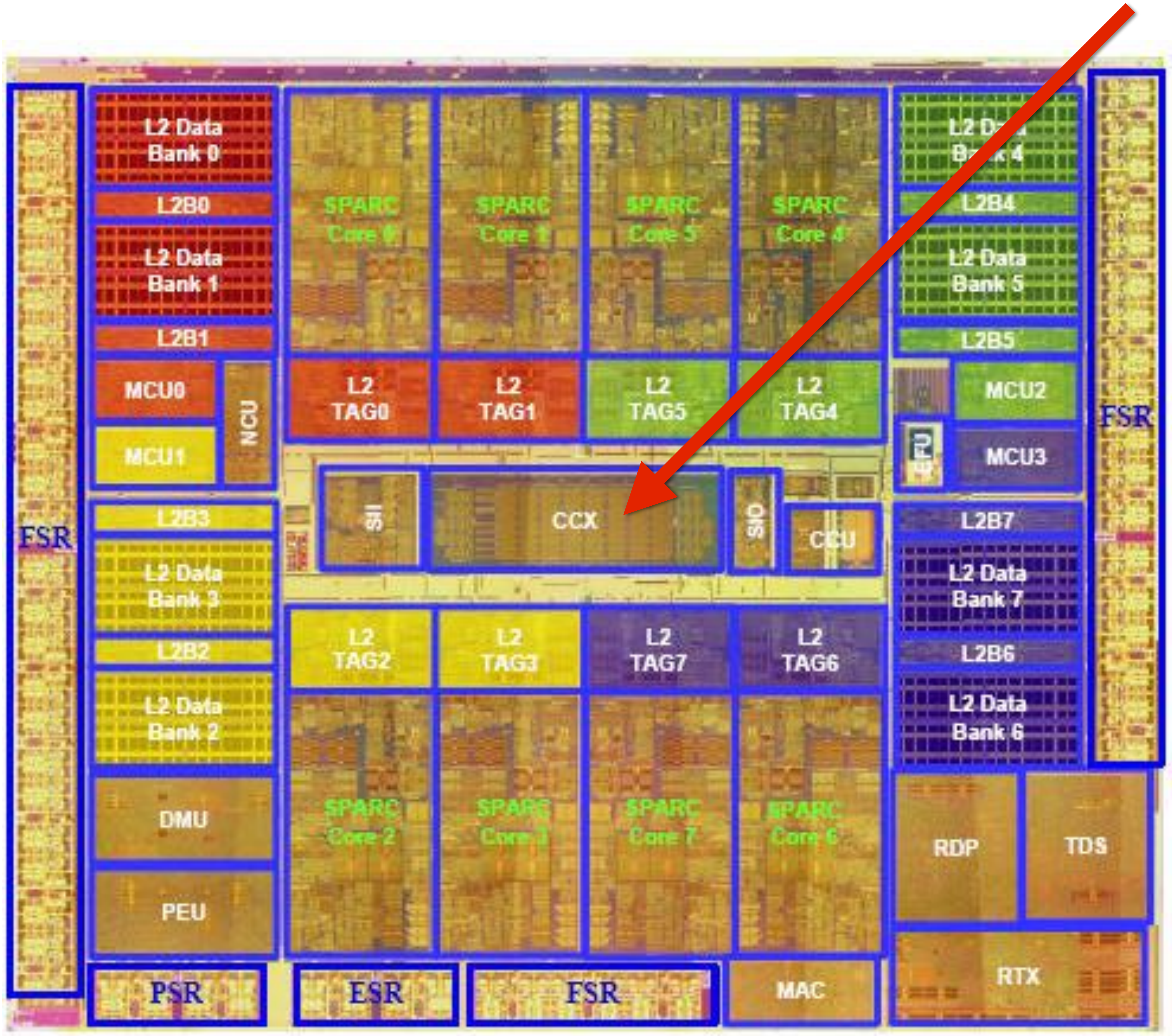




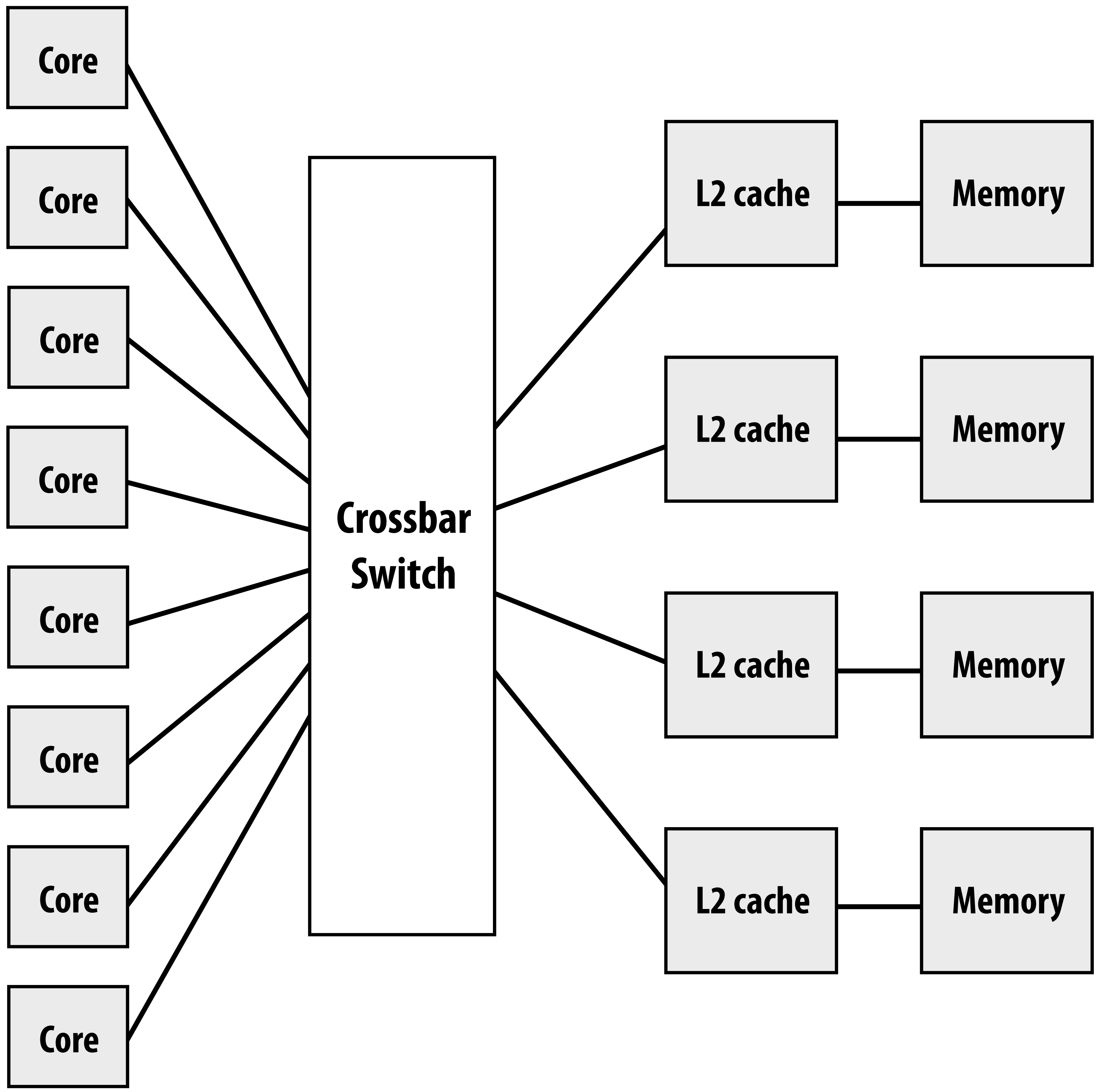
- 72 cores, arranged as 6x6 mesh of tiles (2 cores/tile)
- YX routing of messages:
  - Message travels in Y direction
  - "Turn"
  - Message traves in X direction

SUN Niagara 2 (UltraSPARC T2): crossbar interconnect

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



Eight core processor



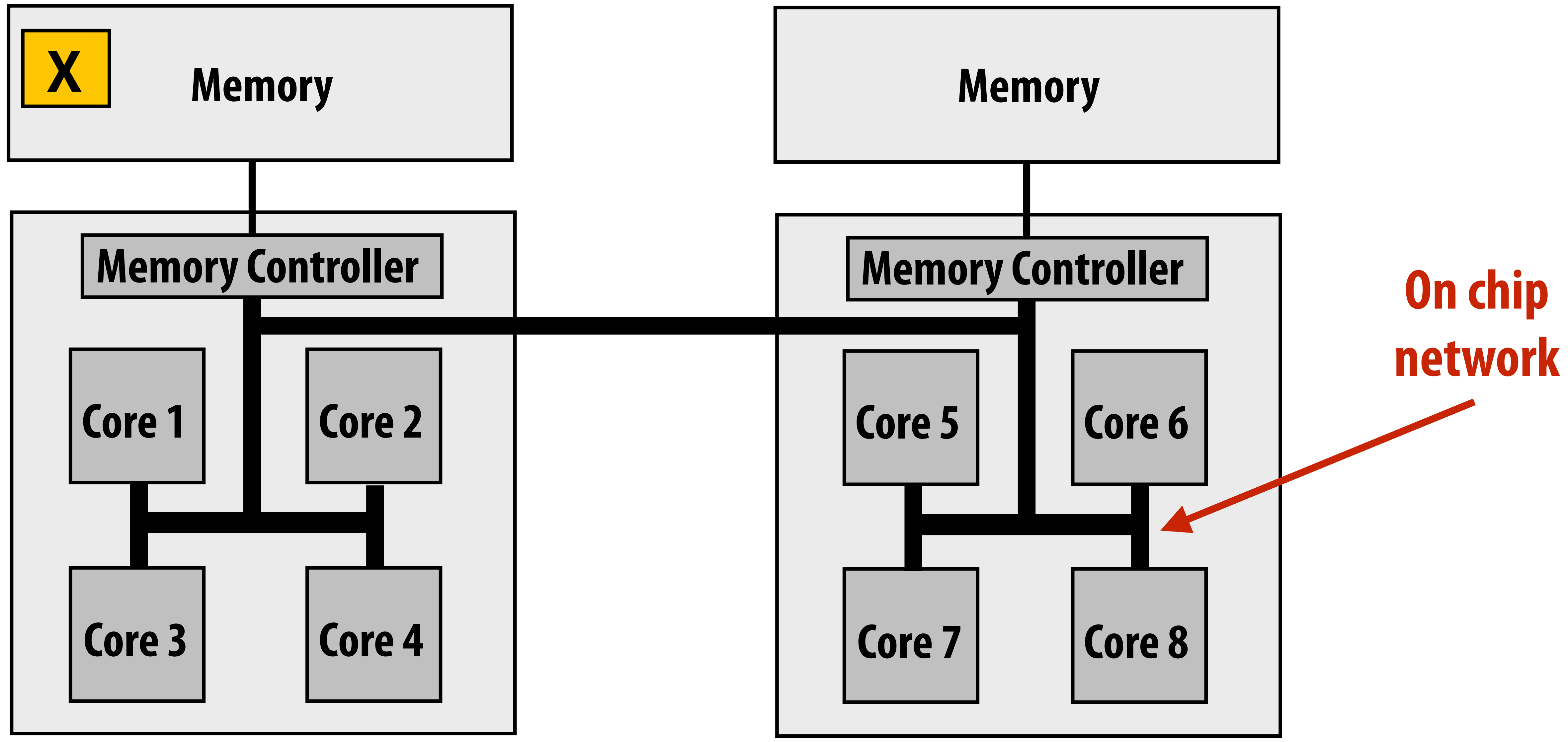
Crossbar = All cores connected directly to all others

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 *

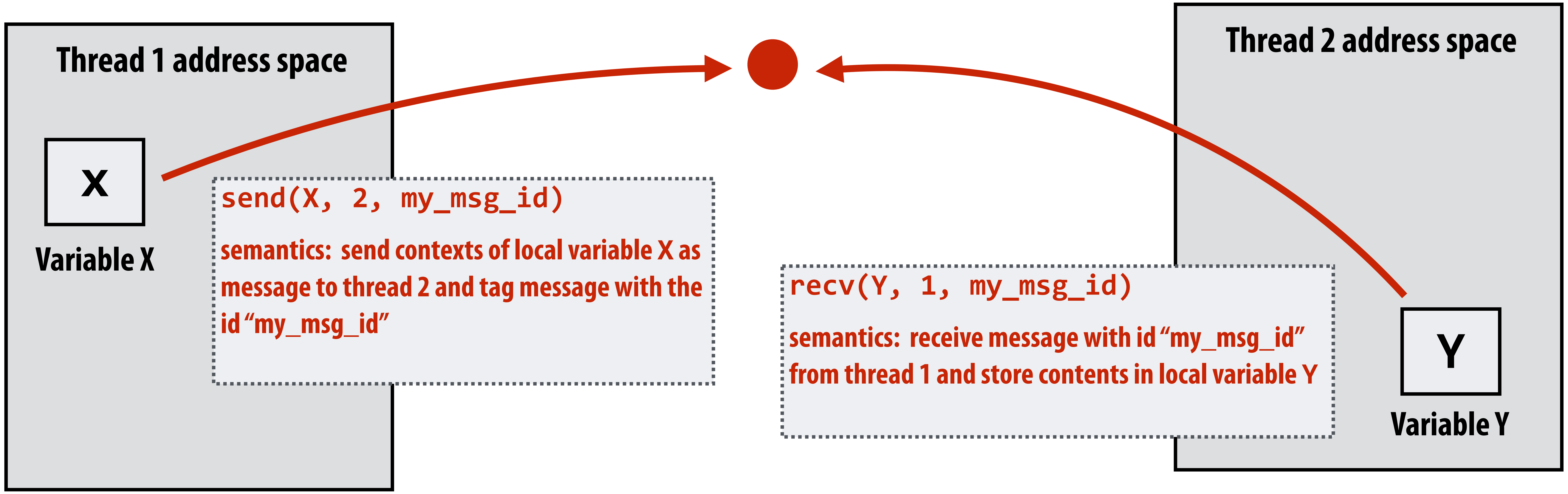
- **Requires hardware support to implement efficiently**
  - Any processor can load and store from any address
  - Can be costly to scale to large numbers of processors
    (one of the reasons why high-core count processors are expensive)

* But NUMA implementations require reasoning about locality for performance optimization

Message passing model of communication

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
    - Why?



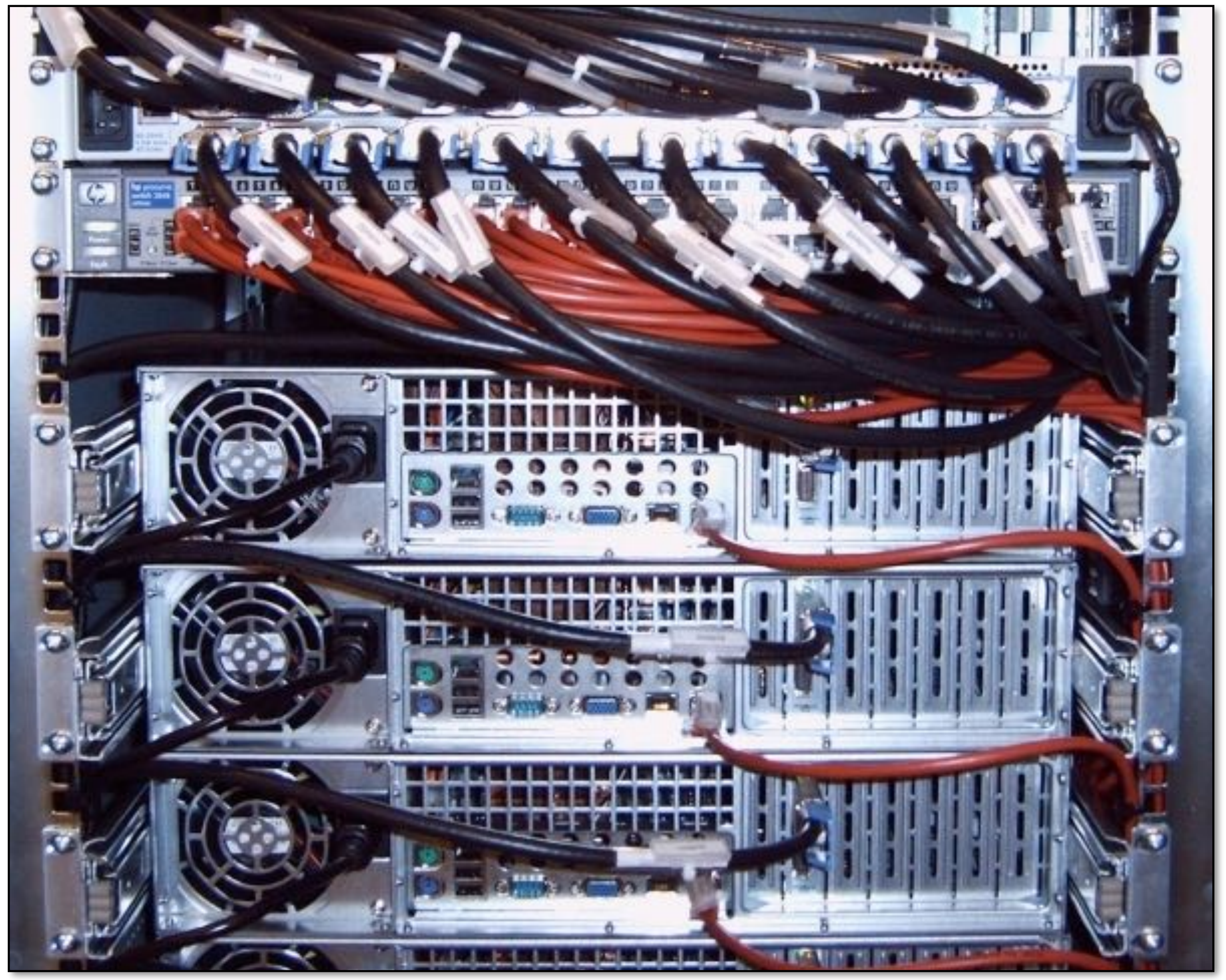
(Communication operations shown in red)

Illustration adopted from Culler, Singh, Gupta

A common metaphor: snail mail

Message passing (implementation)

- Hardware need not implement system-wide loads and stores to execute message passing programs (it need only 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)

The data-parallel model

Data-parallel model *

- **Organize computation as operations on sequences of elements**
  - **e.g., perform same function on all elements of a sequence**

- **A common example: NumPy: C = A + B**
  (A, B, and C are vectors of same length)

*** We'll have multiple lectures in the course about data-parallel programming and data-parallel thinking: this is just a taste**

Key data type of data-parallel code: sequences

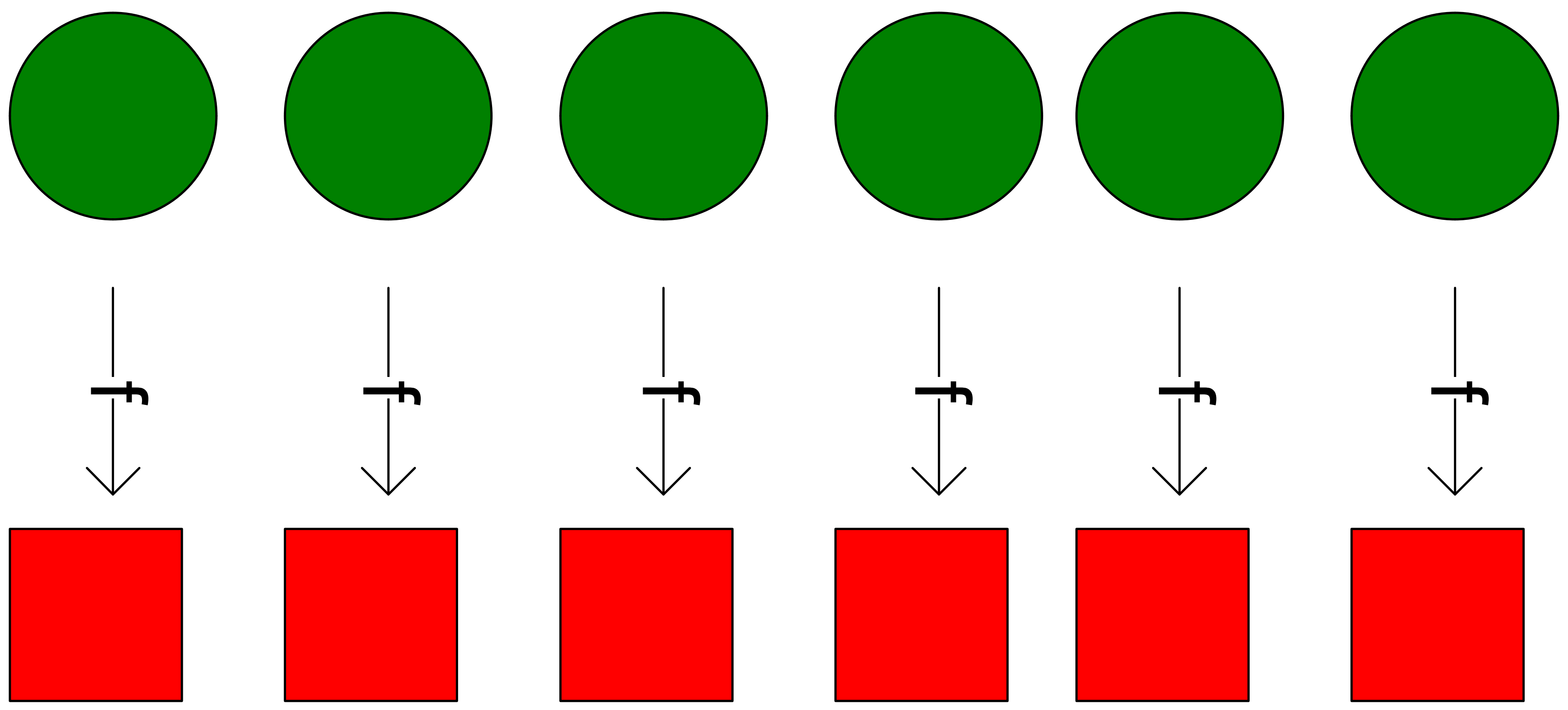
- A sequence is an ordered collection of elements
- For example, in a C++ like language: Sequence<T>
- Scala lists: List[T]
- In a functional language (like Haskell): seq T
- In numPy: An n-D array

- Program can only access elements of sequence through sequence operators:
  - map, reduce, scan, shift, etc.

Map

- Higher order function (function that takes a function as an argument) that operates on sequences
- Applies side-effect-free function `f :: a -> b` to all elements of input sequence, to produce output sequence of the same length (Note: c-style type signature of f is — B f(A)
- In a functional language (e.g., Haskell)
  - `map :: (a -> b) -> seq a -> seq b`
- In C++ (as a templated function):

```
template<class InputIt, class OutputIt, class UnaryOperation>
OutputIt transform(InputIt a_first, InputIt a_last, OutputIt b_first,
                   UnaryOperation f);
```



Parallelizing map

- Since `f :: a -> b` is a function (side-effect free), then applying `f` to all elements of the sequence can be done in any order without changing the output of the program

- The implementation of map has flexibility to reorder/parallelize processing of elements of sequence however it sees fit

Data parallelism in ISPC

```
// main C++ code:
const int N = 1024;
float* x = new float[N];
float* y = new float[N];

// initialize N elements of x here

absolute_value(N, x, y);
```

```
// ISPC code:
export void absolute_value(
   uniform int N,
   uniform float* x,
   uniform float* y)
{
   foreach (i = 0 ... N)
   {
      if (x[i] < 0)
         y[i] = -x[i];
      else
         y[i] = x[i];
   }
}
```

`foreach` **construct**

Think of loop body as a function

```
float loop_body(float input)
  if (input < 0.0)
    return -input;
  else
    return input;
}
```

Given this program, it is reasonable to think of the program as using `foreach` to "map the loop body onto each element" of the arrays X and Y.

But if we want to be more precise: a sequence is not a first-class ISPC concept. It is implicitly defined by how the program has implemented array indexing logic in the `foreach` loop.

(There is no operation in ISPC with the semantics: "map this code over all elements of this sequence")

Data parallelism in ISPC

```
// main C++ code:
const int N = 1024;
float* x = new float[N/2];
float* y = new float[N];

// initialize N/2 elements of x here

absolute_repeat(N/2, x, y);
```

Think of loop body as a function

The input/output sequences being mapped over are implicitly defined by array indexing logic

```
// ISPC code:
export void absolute_repeat(
   uniform int N,
   uniform float* x,
   uniform float* y)
{
   foreach (i = 0 ... N)
   {
      if (x[i] < 0)
         y[2*i] = -x[i];
      else
         y[2*i] = x[i];
      y[2*i+1] = y[2*i];
   }
}
```

This is also a valid ISPC program!

It takes the absolute value of elements of x, then repeats it twice in the output array y

(Less obvious how to think of this code as mapping the loop body onto existing sequences.)

Data parallelism in ISPC

```
// main C++ code:
const int N = 1024;
float* x = new float[N];
float* y = new float[N];

// initialize N elements of x

shift_negative(N, x, y);
```

```
// ISPC code:
export void shift_negative(
   uniform int N,
   uniform float* x,
   uniform float* y)
{
   foreach (i = 0 ... N)
   {
      if (i >= 1 && x[i] < 0)
        y[i-1] = x[i];
      else
        y[i] = x[i];
   }
}
```

Think of loop body as a function

The input/output sequences being mapped over are implicitly defined by array indexing logic

The output of this program is undefined!

Possible for multiple iterations of the loop body to write to same memory location

Data-parallel model (foreach) provides no specification of order in which iterations occur

ISPC discussion: sum "reduction"

Compute the sum of all array elements in parallel

```
export uniform float sumall1(uniform int N, uniform float* x)
{
   uniform float sum = 0.0f;
   foreach (i = 0 ... N)
   {
      sum += x[i];
   }

   return sum;
}
```

```
export uniform float sumall2(uniform int N, uniform float* x)
{
   uniform float sum;
   float partial = 0.0f;
   foreach (i = 0 ... N)
   {
      partial += x[i];
   }

   // from ISPC math library
   sum = reduce_add(partial);

   return sum;
}
```

Correct ISPC solution

`sum` is of type `uniform float` (one copy of variable for all program instances)

`x[i]` is not a uniform expression (different value for each program instance)

Result: compile-time type error

ISPC discussion: sum “reduction”

Each instance accumulates a private partial sum (no communication)

Partial sums are added together using the `reduce_add()` cross-instance communication primitive. The result is the same total sum for all program instances (`reduce_add()` returns a uniform float)

The ISPC code at right will execute in a manner similar to handwritten C + AVX intrinsics implementation below. *

```
export uniform float sumall2(
   uniform int N,
   uniform float* x)
{
   uniform float sum;
   float partial = 0.0f;
   foreach (i = 0 ... N)
   {
      partial += x[i];
   }

   // from ISPC math library
   sum = reduce_add(partial);

   return sum;
}
```

```
float sumall2(int N, float* x) {

  float tmp[8];  // assume 16-byte alignment
  __mm256 partial = _mm256_broadcast_ss(0.0f);

  for (int i=0; i<N; i+=8)
    partial = _mm256_add_ps(partial, _mm256_load_ps(&x[i]));

  _mm256_store_ps(tmp, partial);

  float sum = 0.f;
  for (int i=0; i<8; i++)
    sum += tmp[i];

  return sum;
}
```

*** Self-test: If you understand why this implementation complies with the semantics of the ISPC gang abstraction, then you’ve got a good command of ISPC**

Summary: data-parallel model

- Data-parallelism is about imposing rigid program structure to facilitate simple programming and advanced optimizations

- Basic structure: map a function onto a large collection of data
  - Functional: side-effect free execution
  - No communication among distinct function invocations
    (allow invocations to be scheduled in any order, including in parallel)

- Other data parallel operators express more complex patterns on sequences: gather, scatter, reduce, scan, shift, etc.
  - This will be a topic of a later lecture

- You will think in terms of data-parallel primitives often in this class, but many modern performance-oriented data-parallel languages do not <u>enforce</u> this structure in the language
  - Many languages (like ISPC, CUDA, etc.) choose flexibility/familiarity of imperative C-style syntax over the safety of a more functional form

Summary

- Programming models provide a way to think about the organization of parallel programs.

- They provide abstractions that permit multiple valid implementations.

- *I want you to always be thinking about abstraction vs. implementation for the remainder of this course.*

Parallel Programming Basics

Creating a parallel program

- **Thought process:**
  1. Identify work that can be performed in parallel
  2. Partition work (and also data associated with the work)
  3. Manage data access, communication, and synchronization

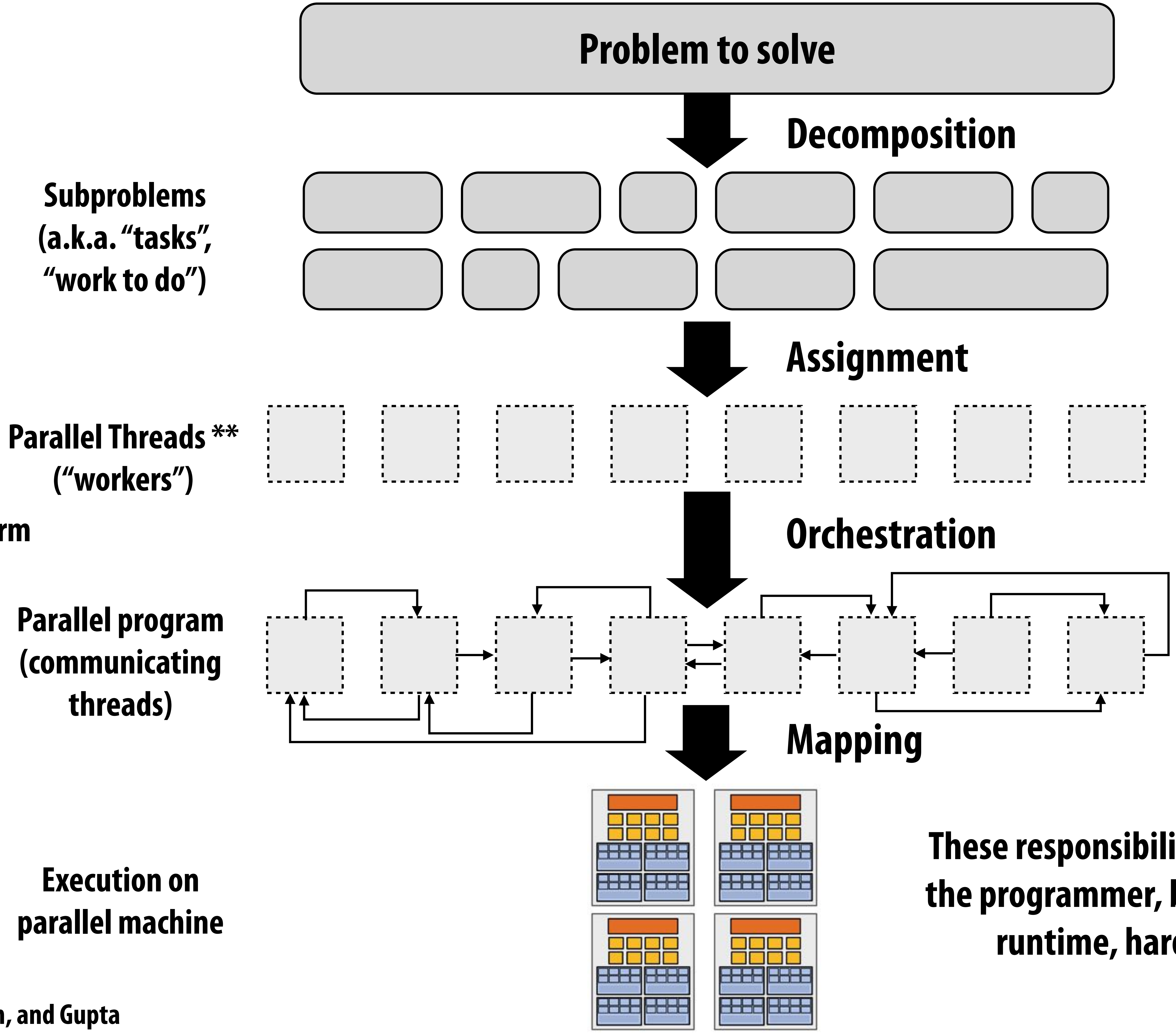
- **A common goal is maximizing speedup ***

For a fixed computation:

$$\text{Speedup(P processors)} = \frac{\text{Time (1 processor)}}{\text{Time (P processors)}}$$

* Other goals include achieving high efficiency (cost, area, power, etc.) or working on bigger problems than can fit on one machine

Creating a parallel program



** I had to pick a term

These responsibilities may be assumed by the programmer, by the system (compiler, runtime, hardware), or by both!

Adopted from: Culler, Singh, and Gupta

Problem decomposition

- Break up problem into tasks that <u>can</u> be carried out in parallel

- In general: create at least enough tasks to keep all execution units on a machine busy

Key challenge of decomposition:
identifying dependencies
(or... a lack of dependencies)

Amdahl's Law: dependencies limit maximum speedup due to parallelism

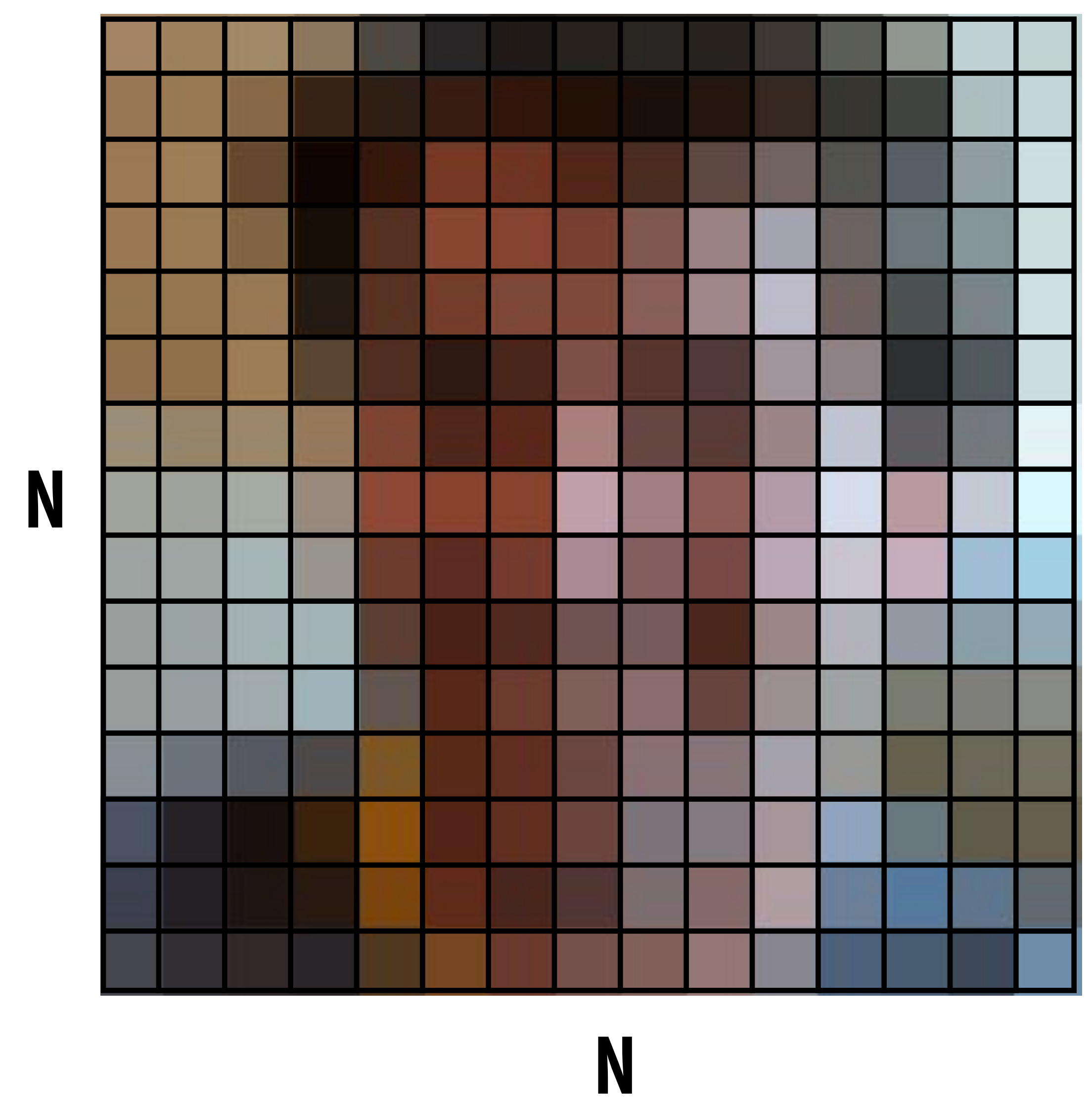
- You run your favorite sequential program...

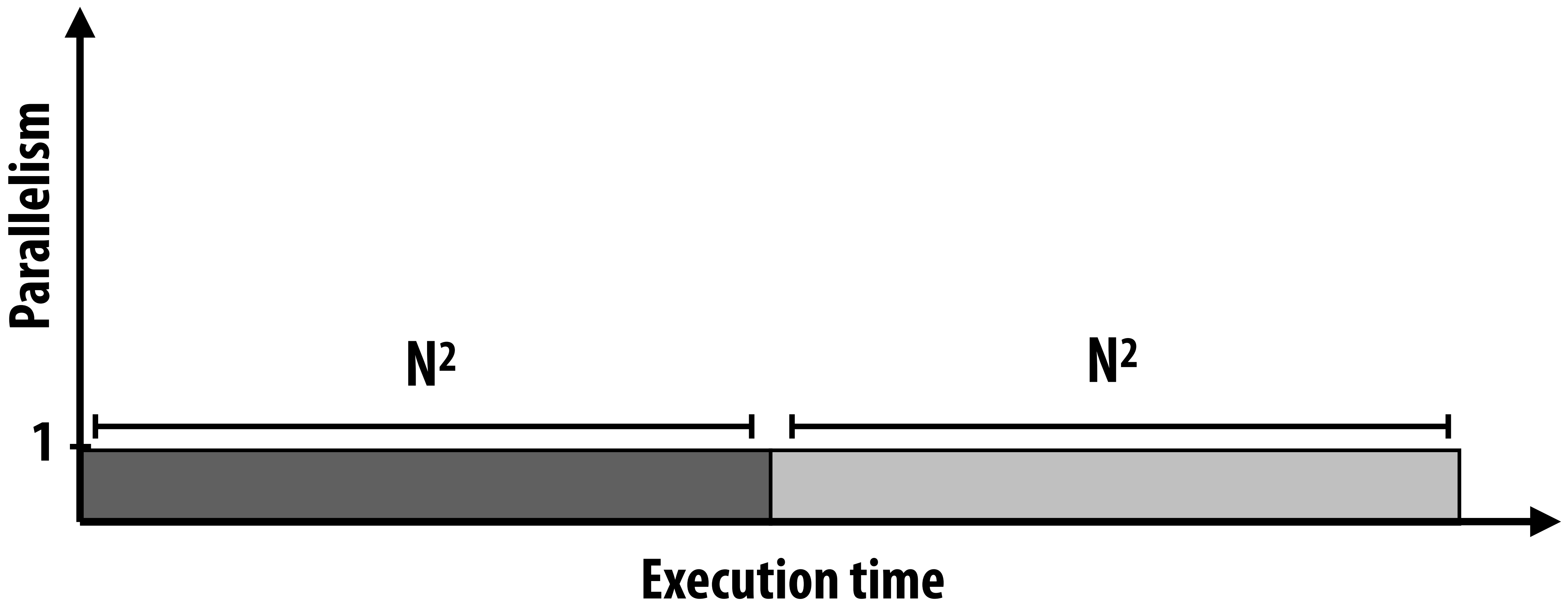
- Let S = the fraction of sequential execution that is inherently sequential (dependencies prevent parallel execution)

- Then maximum speedup due to parallel execution $\leq 1/S$

A simple example

- Consider a two-step computation on a N x N image
  - Step 1: multiply brightness of all pixels by two (independent computation on each pixel)
  - Step 2: compute average of all pixel values
- Sequential implementation of program
  - Both steps take ~ N^2 time, so total time is ~ $2N^2$





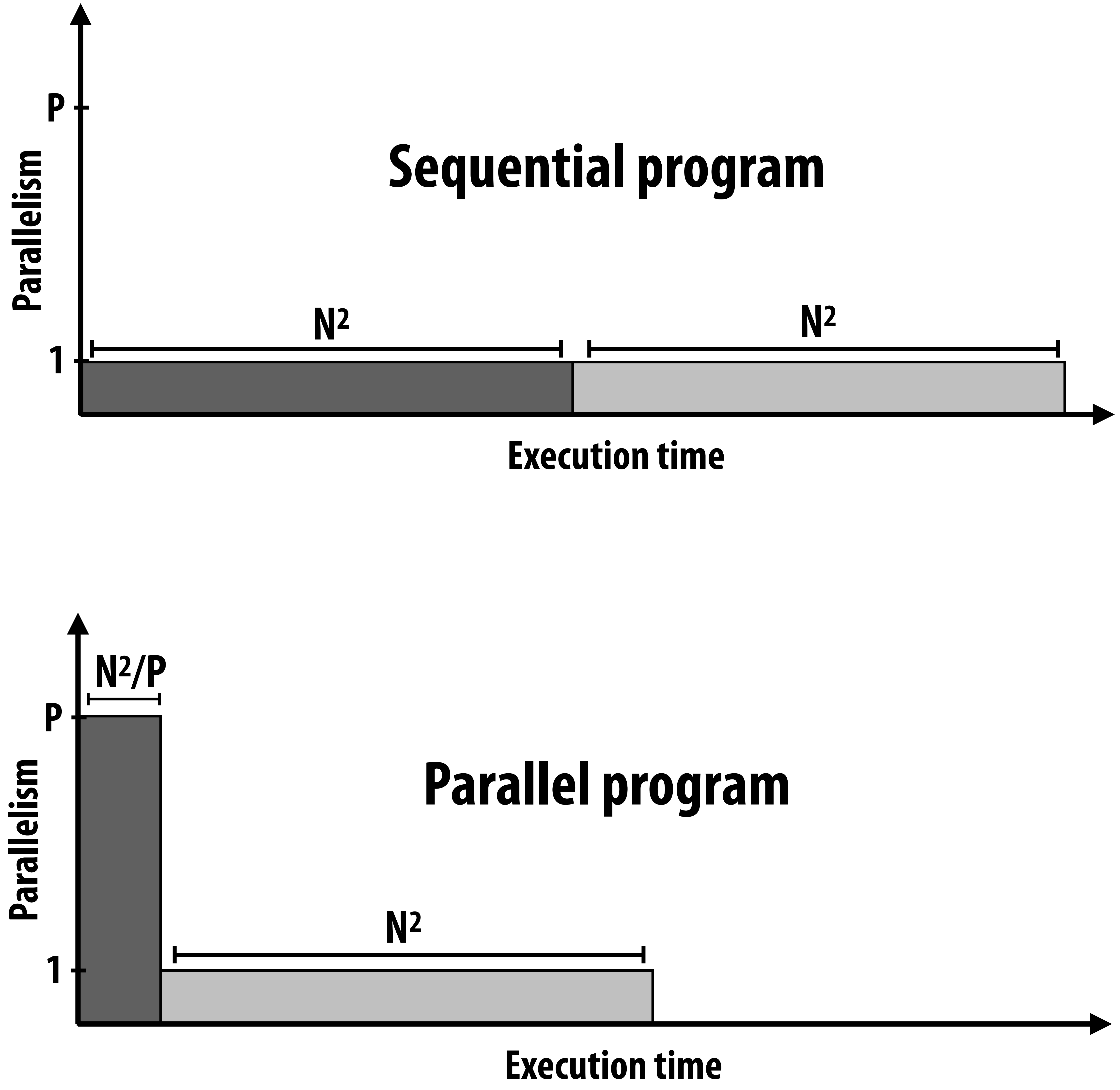
First attempt at parallelism (P processors)

- **Strategy:**
  - Step 1: execute in parallel
    - time for phase 1: N^2/P
  - Step 2: execute serially
    - time for phase 2: N^2

- **Overall performance:**

Speedup $\leq \dfrac{2n^2}{\dfrac{n^2}{p} + n^2}$

Speedup ≤ 2

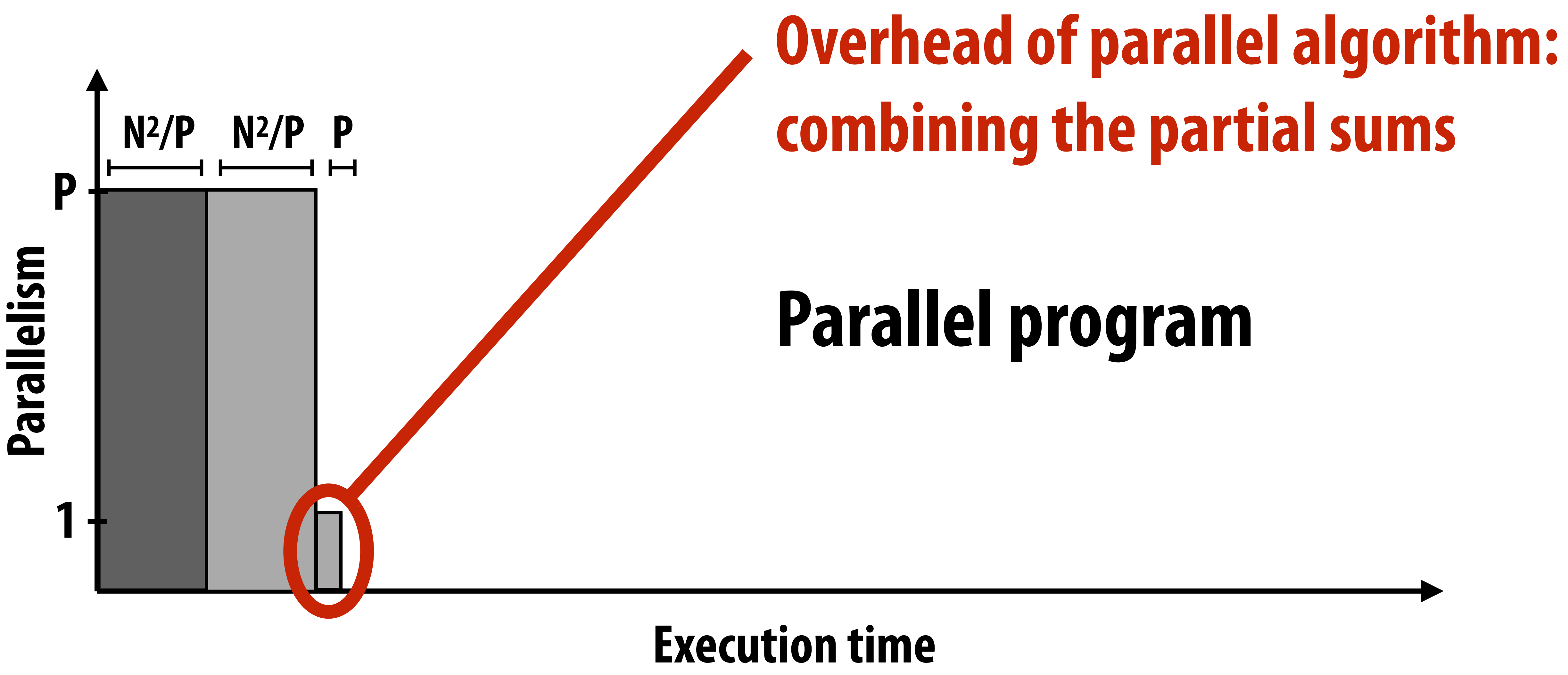


Parallelizing step 2

- **Strategy:**
  - Step 1: execute in parallel
    - time for phase 1: N^2/P
  - Step 2: compute partial sums in parallel, combine results serially
    - time for phase 2: $N^2/P + P$

- **Overall performance:**
  - Speedup $\leq \dfrac{2n^2}{\dfrac{2n^2}{p} + p}$

Note: speedup → P when N >> P

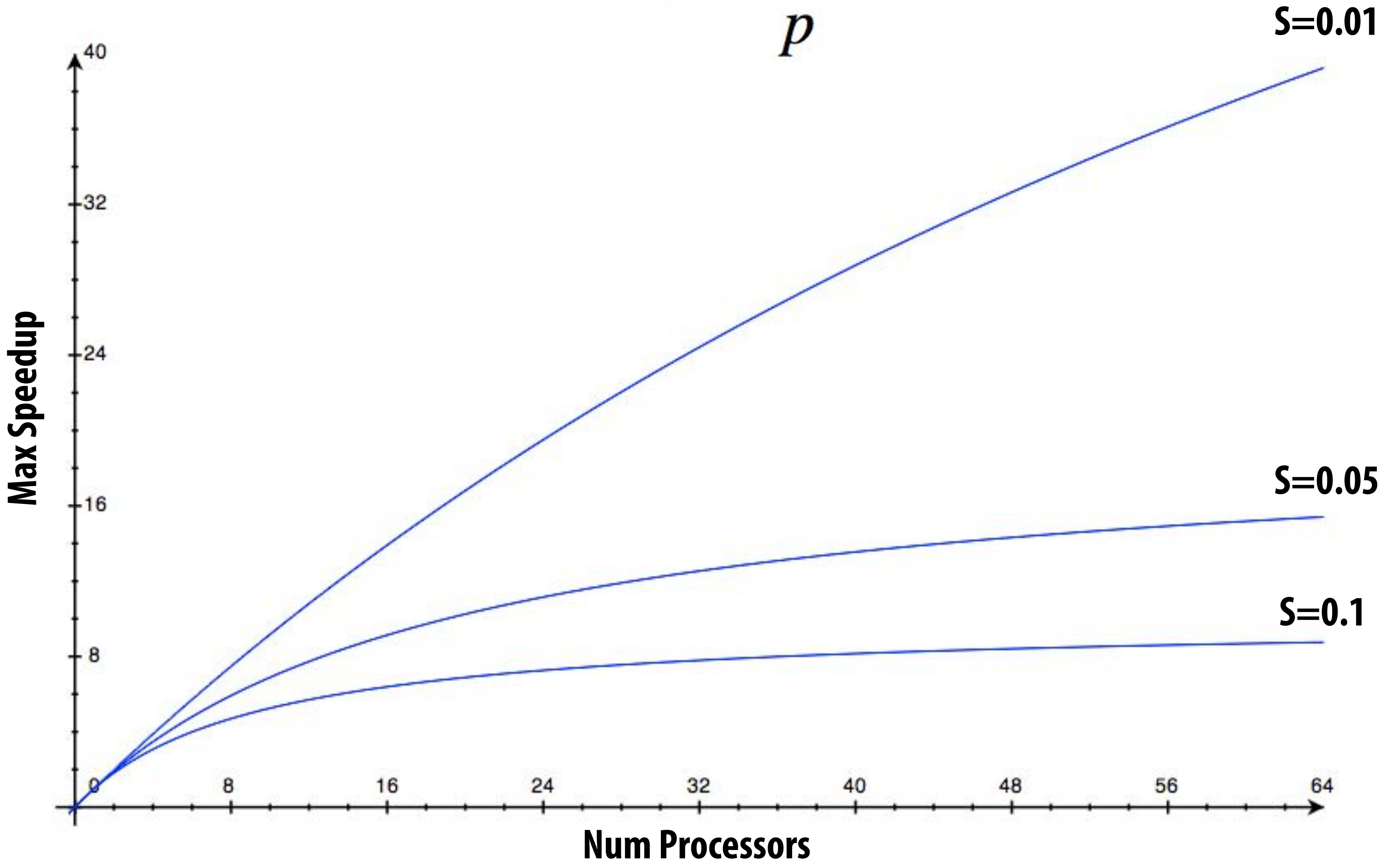


Parallel program

Amdahl's law

- **Let S = the fraction of total work that is inherently sequential**
- **Max speedup on P processors given by:**

$$\textbf{speedup} \leq \frac{1}{s + \frac{1-s}{p}}$$



A small serial region can limit speedup on a large parallel machine

Summit supercomputer: 27,648 GPUs x (5,376 ALUs/GPU) = 148,635,648 ALUs
Machine can perform 148 million single precision operations in parallel
What is max speedup if 0.1% of application is serial?

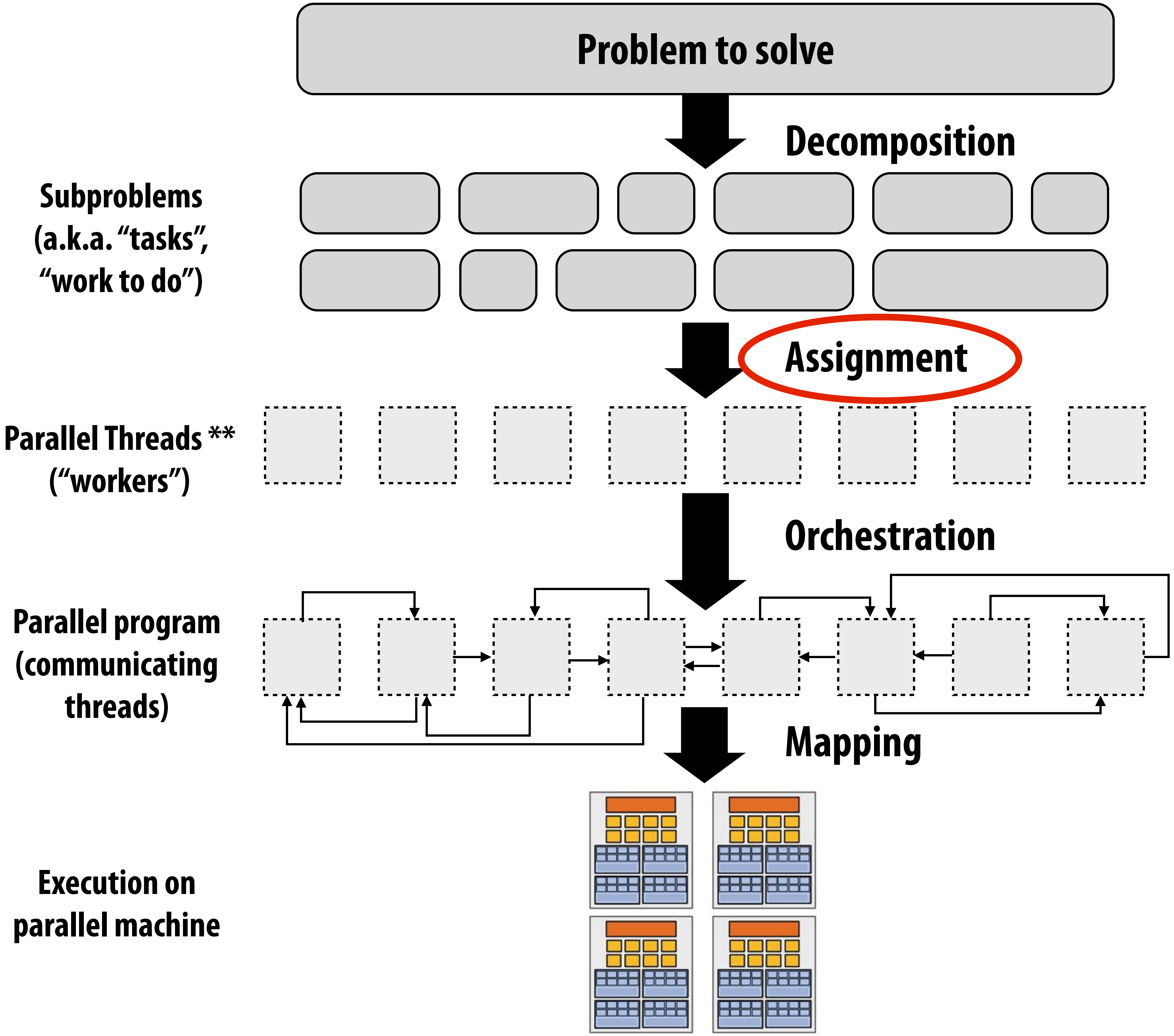
Decomposition

- **Who is responsible for decomposing a program into independent tasks?**
  - In most cases: the programmer

- **Automatic decomposition of sequential programs continues to be a challenging research problem**

  (very difficult in general case)
  - Compiler must analyze program, identify dependencies
    - What if dependencies are data dependent (not known at compile time)?
  - Researchers have had modest success with simple loop nests
  - The "magic parallelizing compiler" for complex, general-purpose code has not yet been achieved

Assignment



** I had to pick a term

Assignment

** I had to pick a term
(will explain in a second)

- Assigning tasks to threads **
  - Think of "tasks" as things to do
  - Think of threads as "workers"

- Goals: achieve good workload balance, reduce communication costs

- Can be performed statically (before application is run), or dynamically as program executes

- Although programmer is often responsible for decomposition, many languages/runtimes take responsibility for assignment.

Assignment examples in ISPC

```
export void ispc_sinx_interleaved(
   uniform int N,
   uniform int terms,
   uniform float* x,
   uniform float* result)
{
   // assumes N % programCount = 0
   for (uniform int i=0; i<N; i+=programCount)
   {
      int idx = i + programIndex;
      float value = x[idx];
      float numer = x[idx] * x[idx] * x[idx];
      uniform int denom = 6;  // 3!
      uniform int sign = -1;

      for (uniform int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[idx] * x[idx];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }
      result[i] = value;
   }
}
```

Decomposition of work by loop iteration

Programmer-managed assignment:

- **<u>Static</u> assignment**
- **Assign iterations to ISPC program instances in interleaved fashion**

```
export void ispc_sinx_foreach(
   uniform int N,
   uniform int terms,
   uniform float* x,
   uniform float* result)
{
   foreach (i = 0 ... N)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      uniform int denom = 6;  // 3!
      uniform int sign = -1;

      for (uniform int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }
      result[i] = value;
   }
}
```

Decomposition of work by loop iteration

`foreach` construct exposes independent work to system

System-manages assignment of iterations (work) to ISPC program instances (abstraction leaves room for dynamic assignment, but current ISPC implementation is static)

Example 2: static assignment using C++11 threads

```
void my_thread_start(int N, int terms, float* x, float* results) {
  sinx(N, terms, x, result); // do work
}



void parallel_sinx(int N, int terms, float* x, float* result) {

    int half = N/2.

    // launch thread to do work on first half of array
    std::thread t1(my_thread_start, half, terms, x, result);

    // do work on second half of array in main thread
    sinx(N - half, terms, x + half, result + half);

    t1.join();
}
```

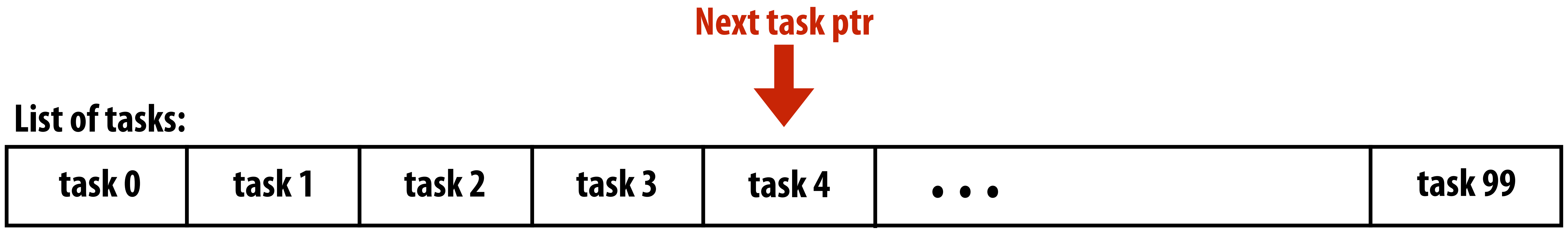
Decomposition of work by loop iteration

Programmer-managed static assignment
This program assigns loop iterations to threads in a blocked fashion (first half of array assigned to the spawned thread, second half assigned to main thread)

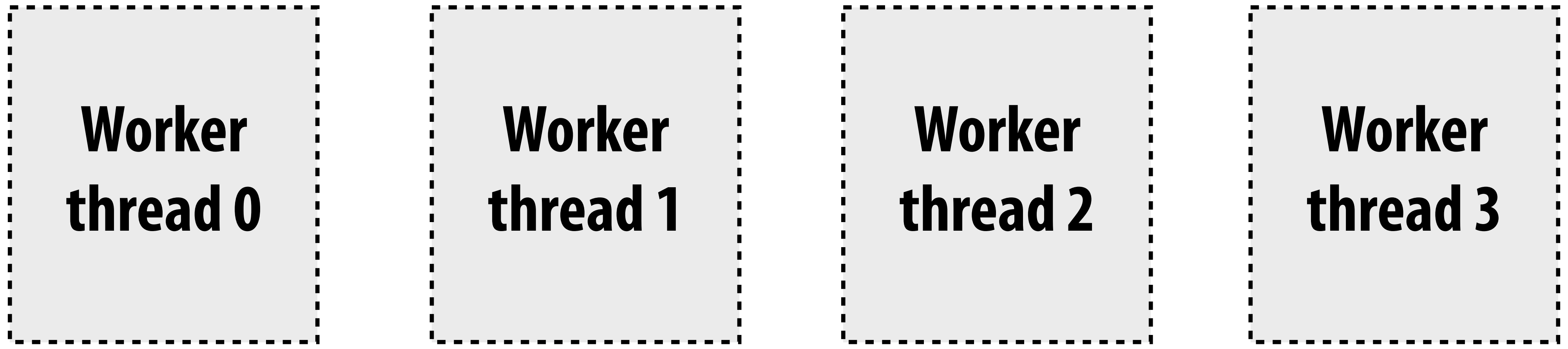
Dynamic assignment using ISPC tasks

```
void foo(uniform float* input,
         uniform float* output,
         uniform int N)
{
  // create a bunch of tasks
  launch[100] my_ispc_task(input, output, N);
}
```

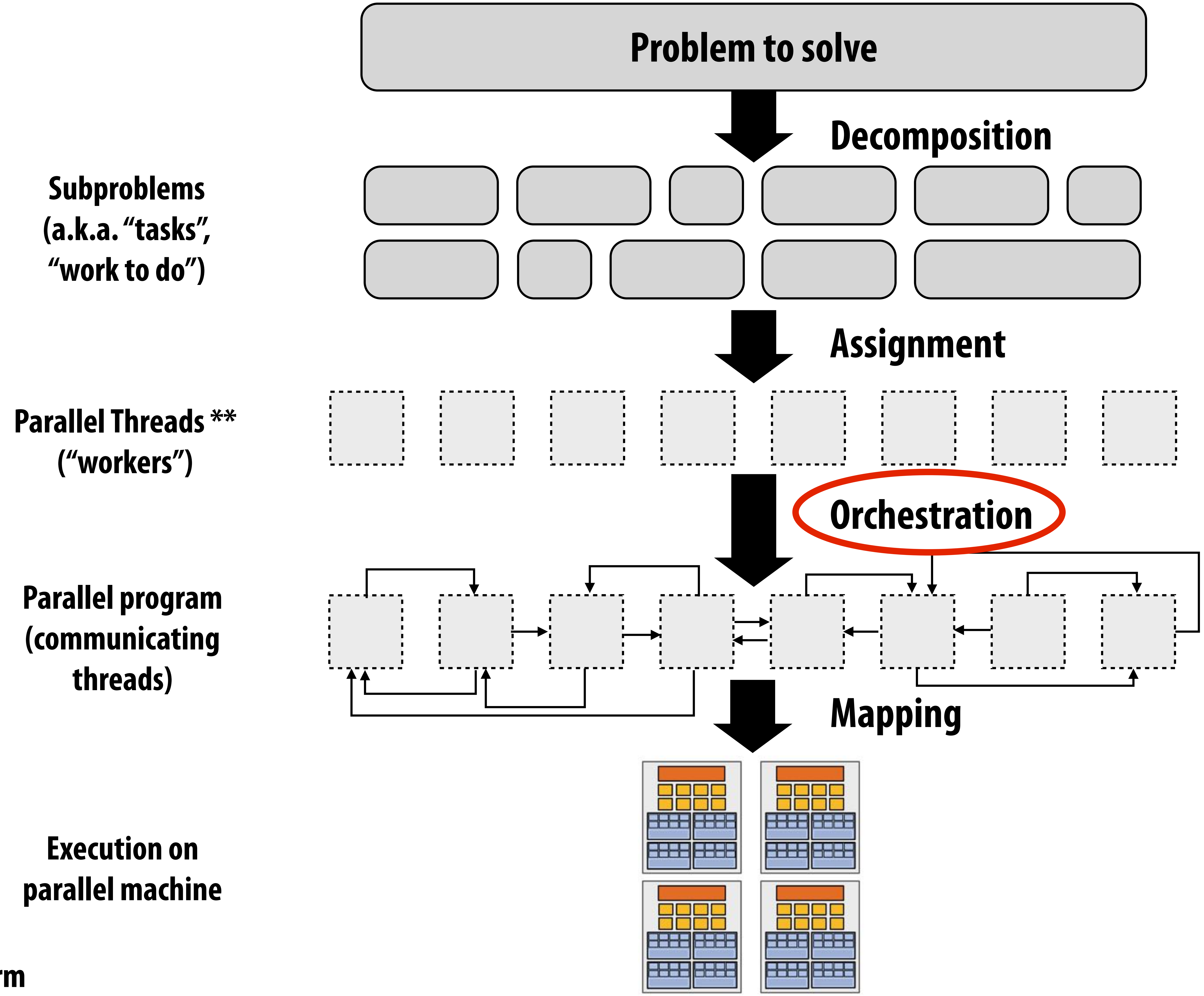
ISPC runtime assigns tasks to worker threads



Implementation of task assignment to threads: after completing current task, worker thread inspects list and assigns itself the next uncompleted task.



Orchestration



** I had to pick a term

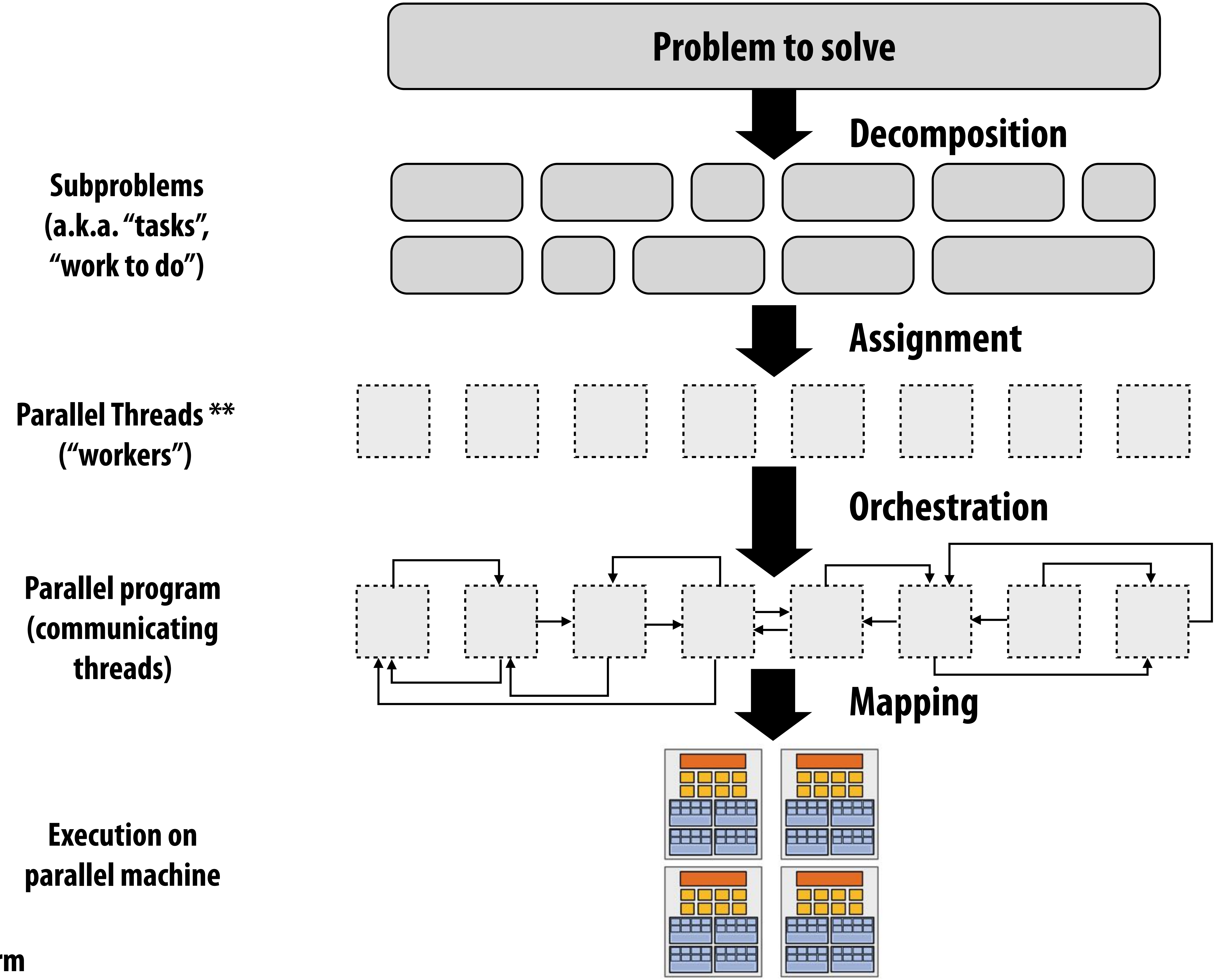
Orchestration

- Involves:
  - Structuring communication
  - Adding synchronization to preserve dependencies if necessary
  - Organizing data structures in memory
  - Scheduling tasks

- Goals: reduce costs of communication/sync, preserve locality of data reference, reduce overhead, etc.

- Machine details impact many of these decisions
  - If synchronization is expensive, programmer might use it more sparsely

Mapping to hardware



** I had to pick a term

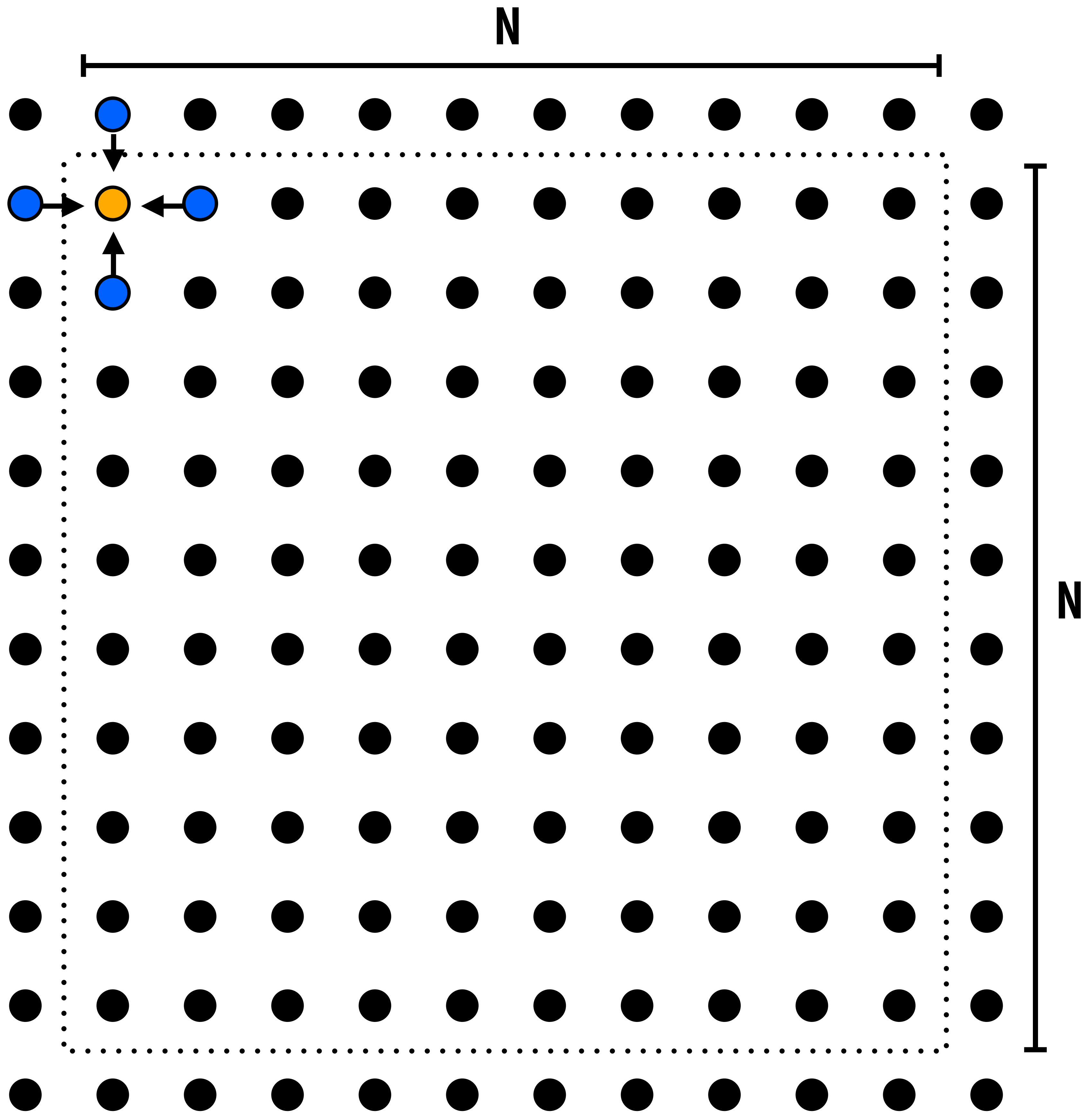
Mapping to hardware

- **Mapping "threads" ("workers") to hardware execution units**
- **Example 1: mapping by the operating system**
  - e.g., map a thread to HW execution context on a CPU core
- **Example 2: mapping by the compiler**
  - Map ISPC program instances to vector instruction lanes
- **Example 3: mapping by the hardware**
  - Map CUDA thread blocks to GPU cores (discussed in future lecture)
- **Some interesting mapping decisions:**
  - Place <u>related</u> threads (cooperating threads) on the same processor
    (maximize locality, data sharing, minimize costs of comm/sync)
  - Place <u>unrelated</u> threads on the same processor (one might be bandwidth limited and another might be compute limited) to use machine more efficiently

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



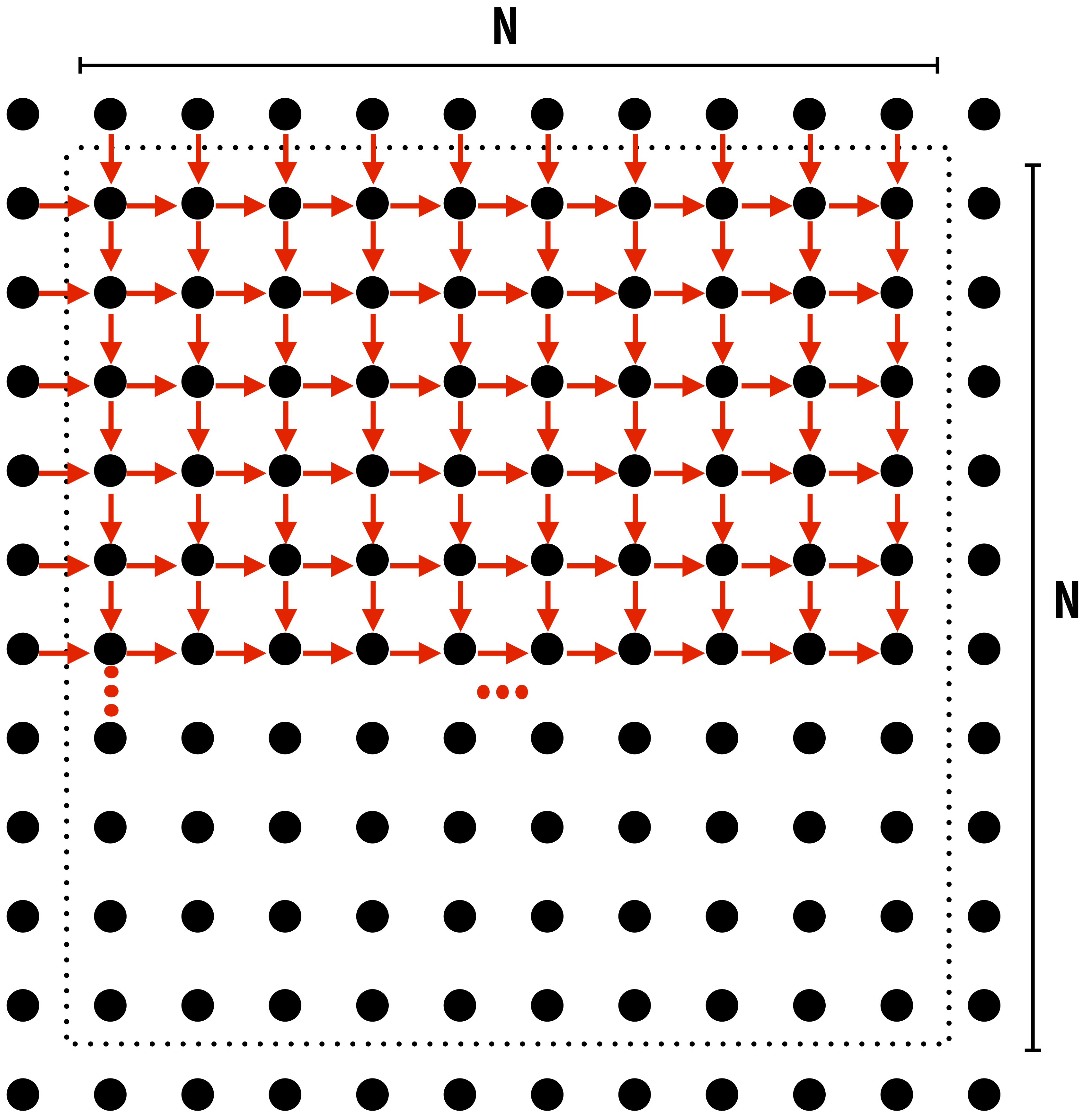

```
A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j]
                        + A[i,j+1] + A[i+1,j]);
```

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)

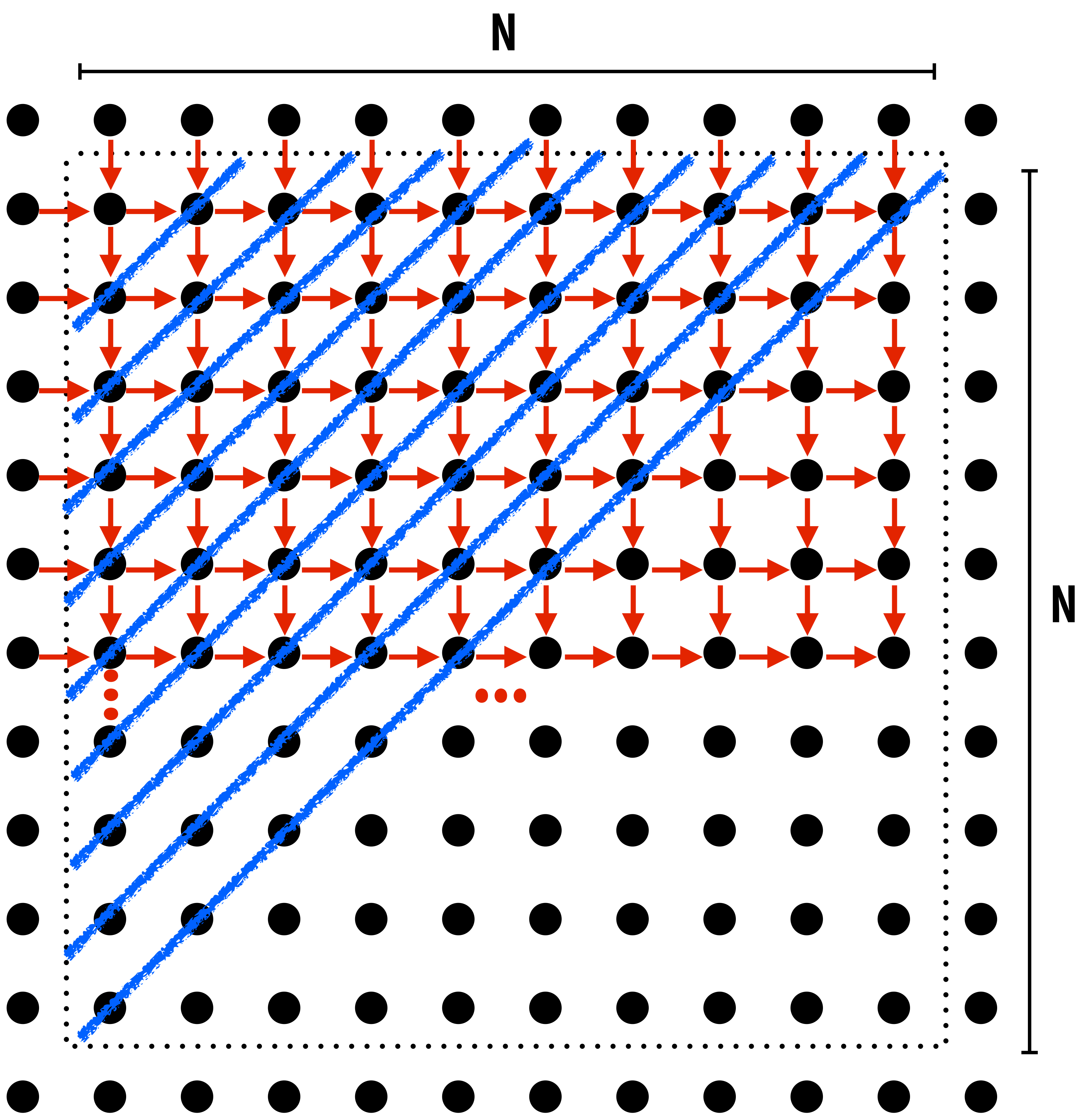


Each row element depends on element to left.

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.

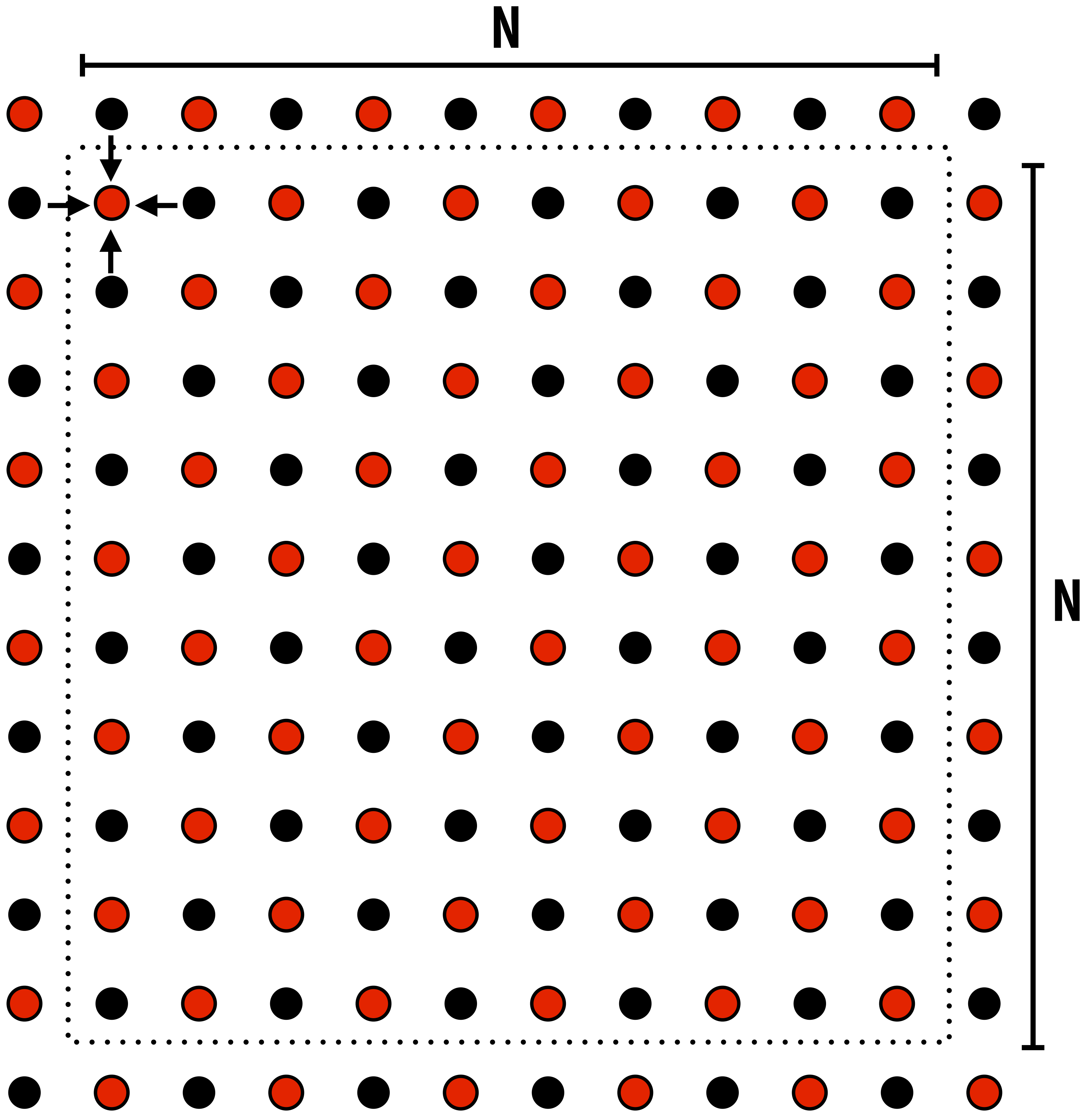
Frequent synchronization (after completing each diagonal)

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



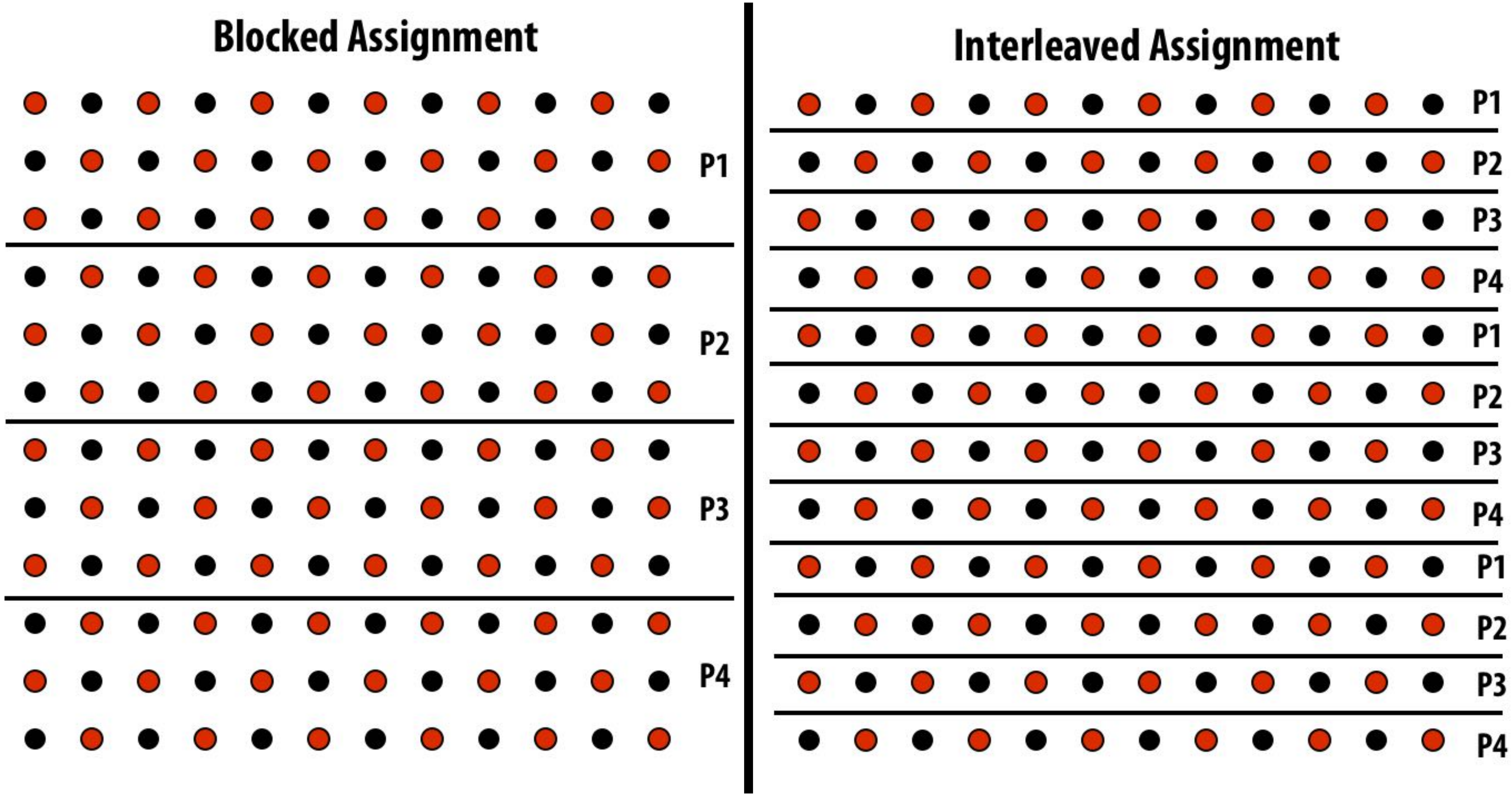
Update all red cells in parallel

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

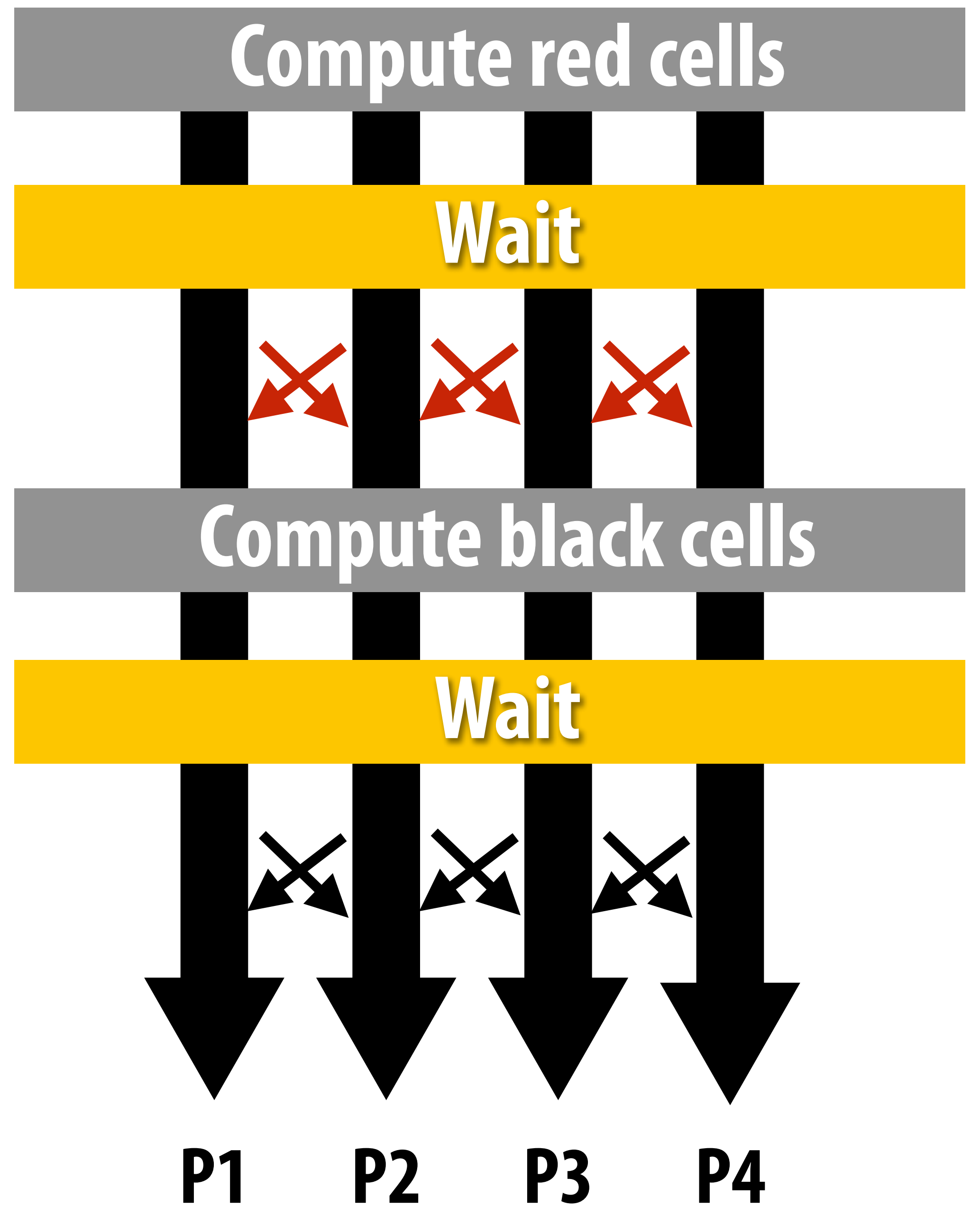


Question: Which is better? Does it matter?

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

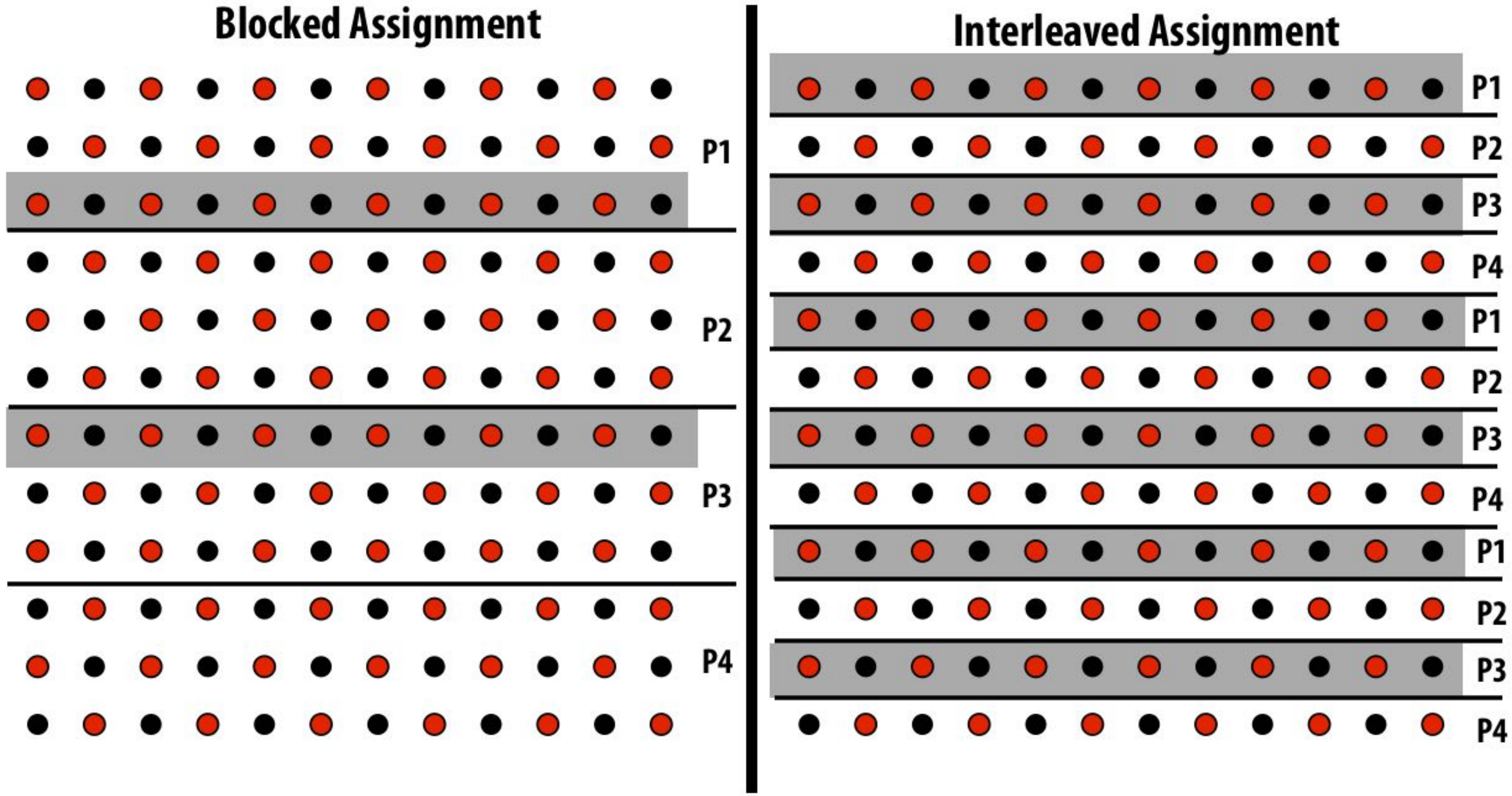
Consider dependencies in the program

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

Reorder grid tra



= data that must be sent to P2 each iteration

Blocked assignment requires less data to be communicated between processors

Two ways to think about writing this program

- Data parallel thinking
- SPMD / shared address space

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;
    }
}
```

Assignment: ???

Decomposition: processing individual grid elements constitutes independent work

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

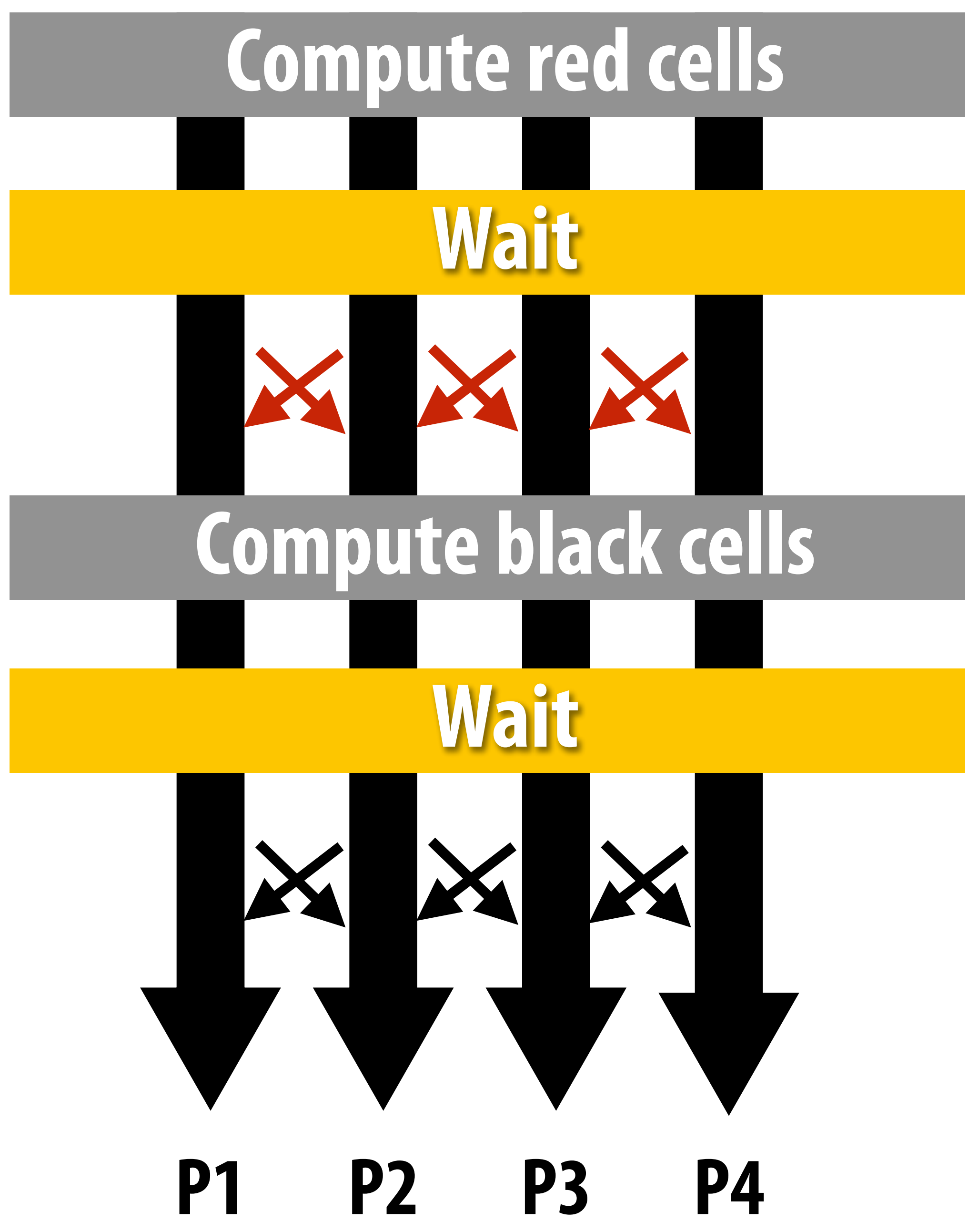
Orchestration: handled by system
(End of for_all block is implicit wait for all workers before returning to sequential control)

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;

// 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);
   }
}
```

Assume these are global variables (accessible to all threads)

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) {
   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);
   }
}
```

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.

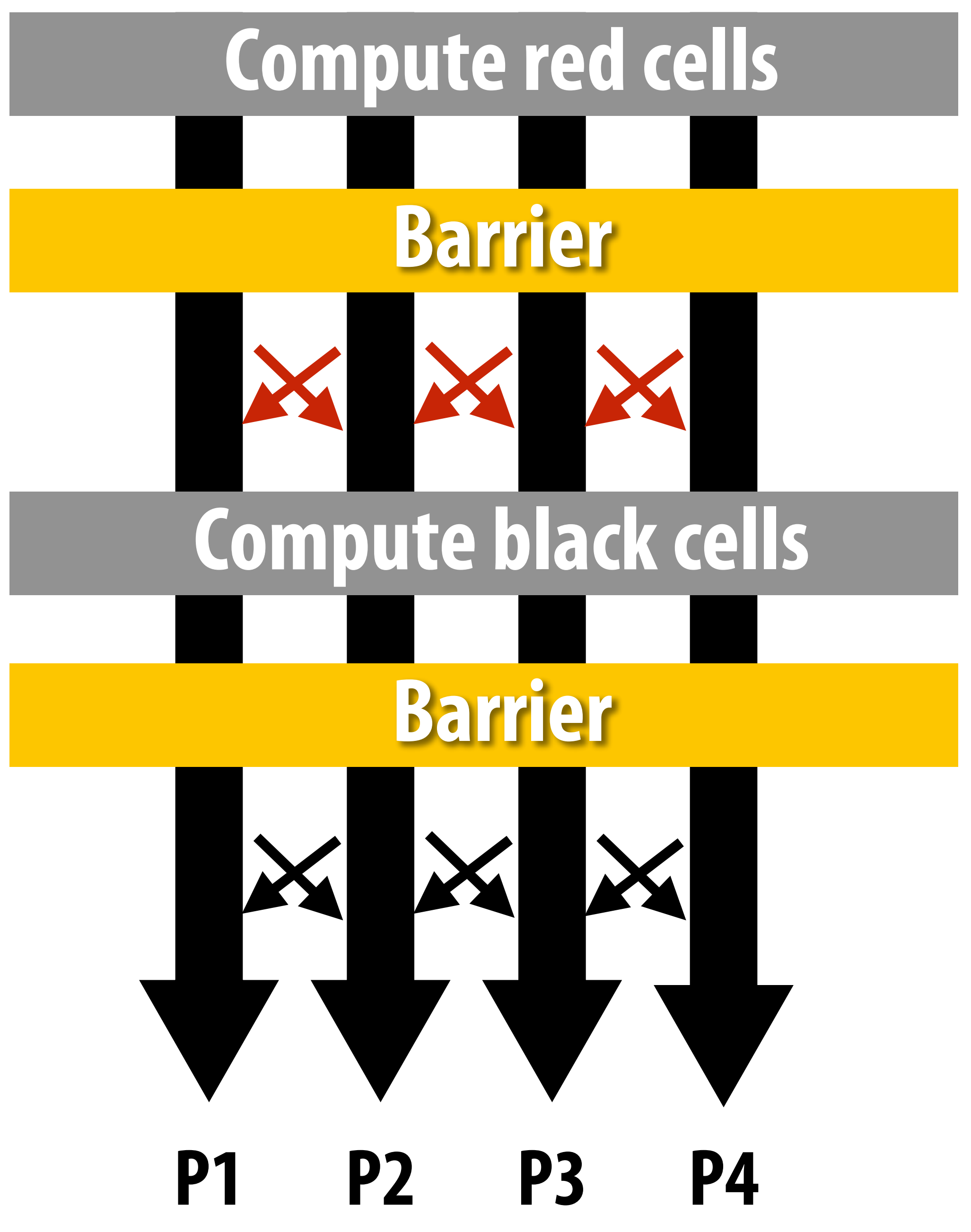
Compute partial sum per worker

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 conservative 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

```
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?

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

Summary

- **Amdahl's Law**
  - Overall maximum speedup from parallelism is limited by amount of serial execution in a program

- **Aspects of creating a parallel program**
  - Decomposition to create independent work, assignment of work to workers, orchestration (to coordinate processing of work by workers), mapping to hardware
  - We'll talk a lot about making good decisions in each of these phases in the coming lectures (in practice, they are very inter-related)

- **Focus today: identifying dependencies**

- **Focus soon: identifying locality, reducing synchronization**