Lecture 4:

Parallel Programming Basics

Parallel Computing
Stanford CS149, Fall 2023

Today's topic: case study on writing an optimizing a parallel program

- Demonstrated in two programming models
  - data parallel
  - shared address space

Creating a parallel program

- **Your 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**

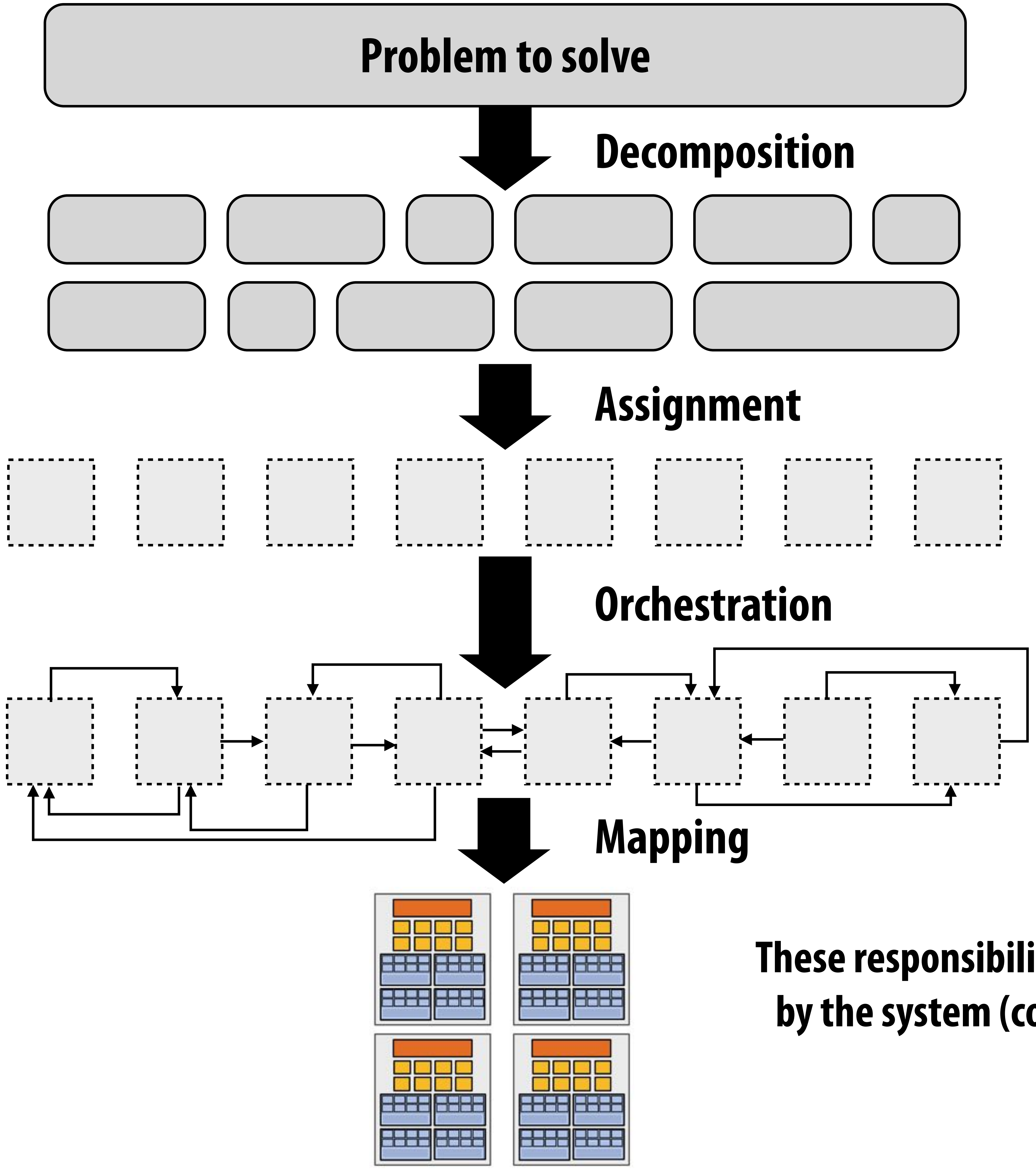
- **A common goal is maximizing speedup ***

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



Subproblems (a.k.a. “tasks”, “work to do”)

Parallel Threads ** (“workers”)

** I had to pick a term

Parallel program (communicating threads)

Execution on parallel machine

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 can 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...**

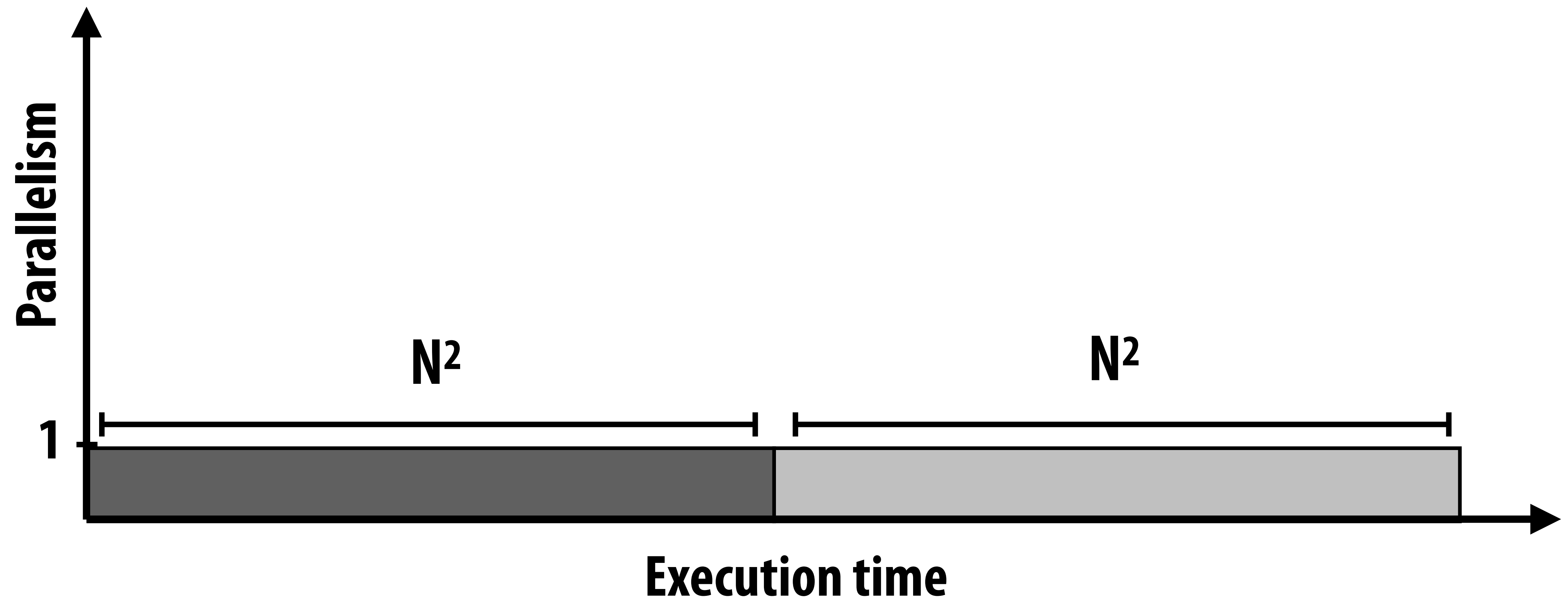
- **Let S = the fraction of sequential execution that is inherently sequential (dependencies prevent parallel execution)**

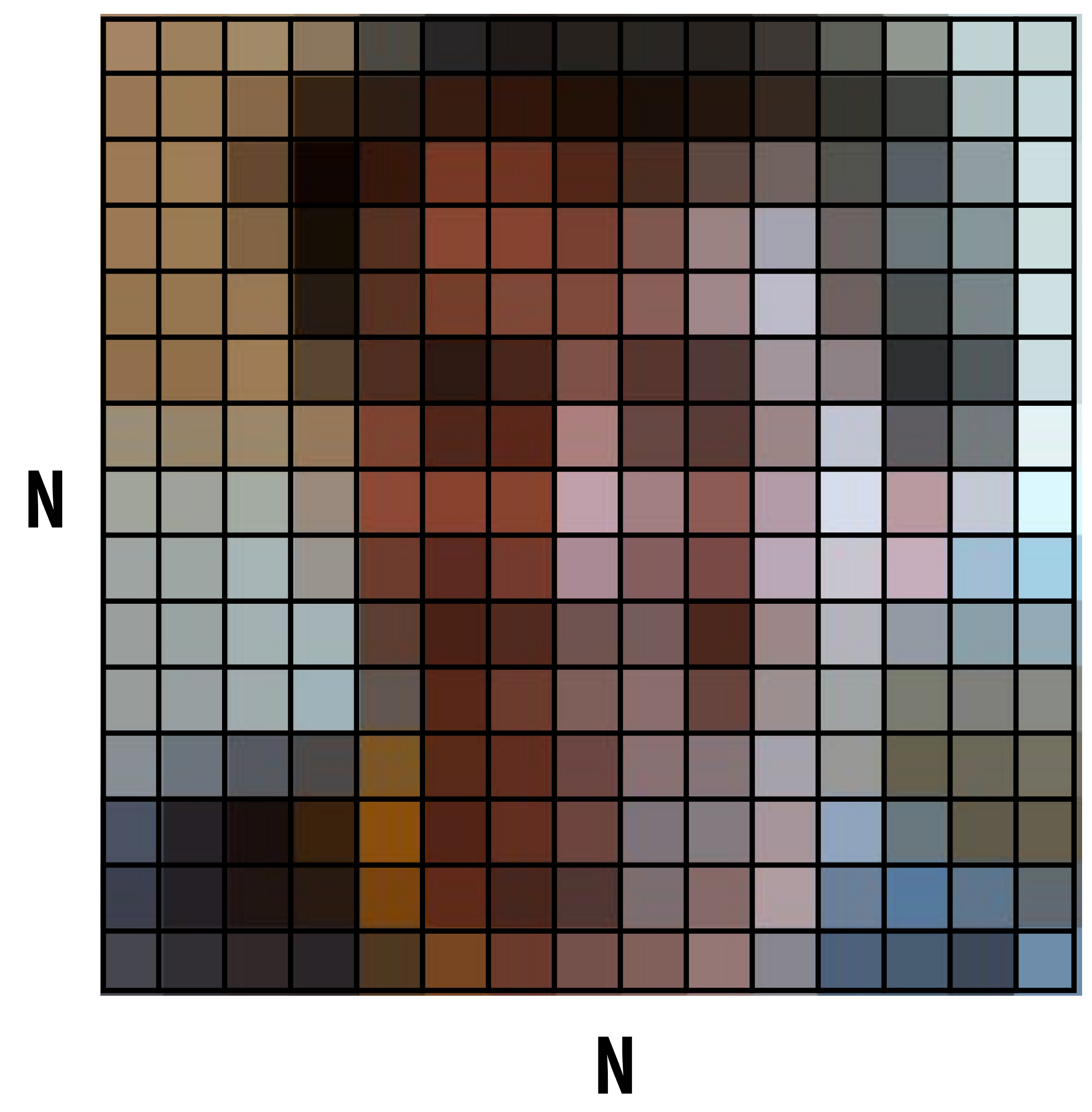
- **Then maximum speedup due to parallel execution $\leq 1/S$**

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 ~ $2N^2$





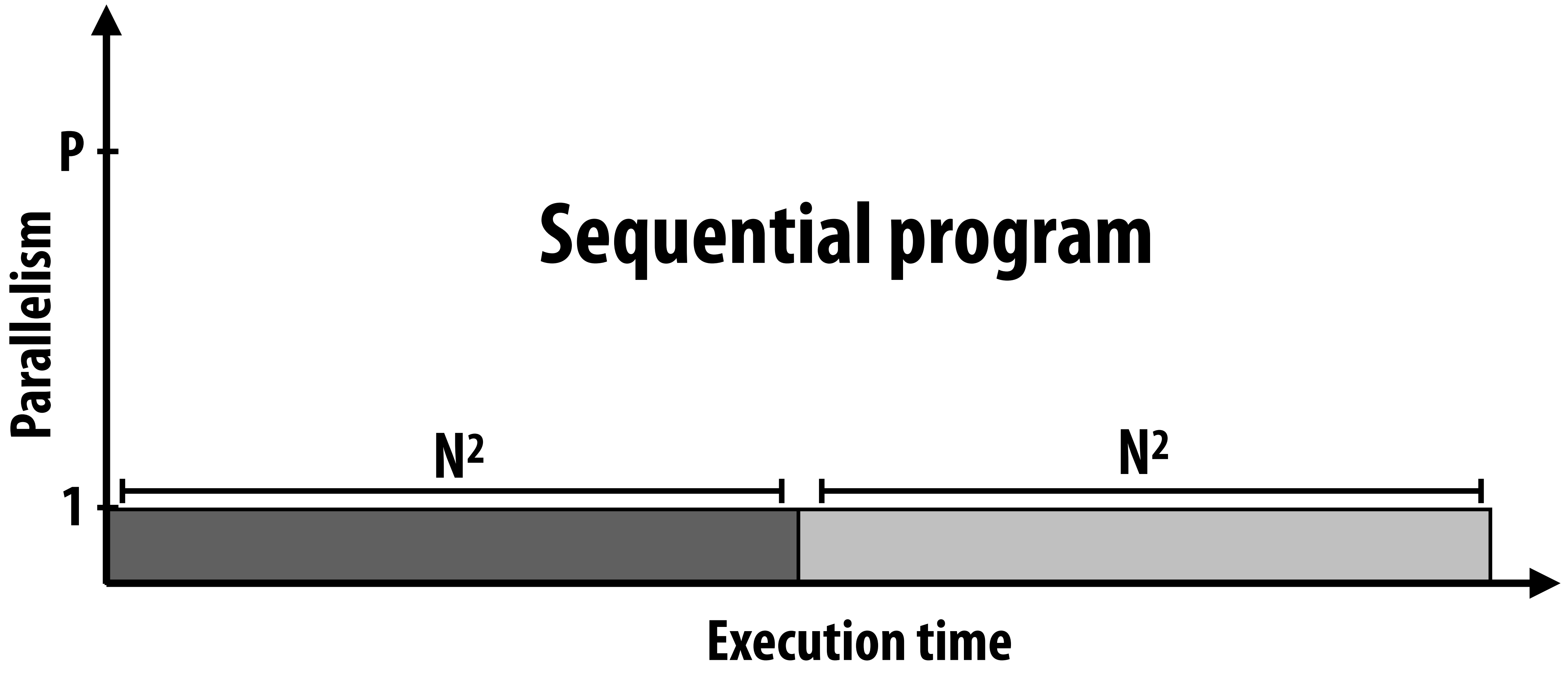
First attempt at parallelism (P processors)

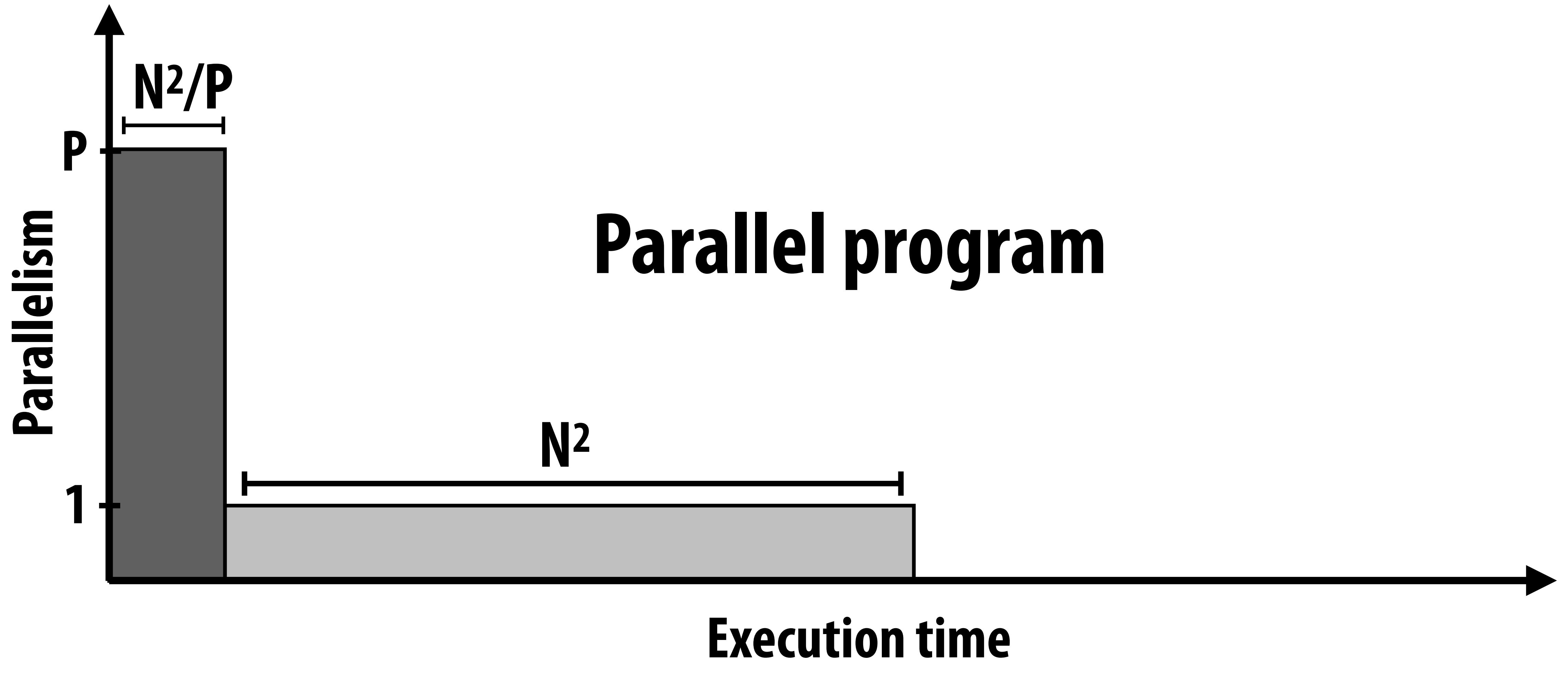
- **Strategy:**
  - Step 1: execute in parallel
    - time for phase 1: N^2/P
  - Step 2: execute serially
    - time for phase 2: N^2

- **Overall performance:**

$$\text{Speedup} \leq \frac{2n^2}{\frac{n^2}{p} + n^2}$$

Speedup ≤ 2



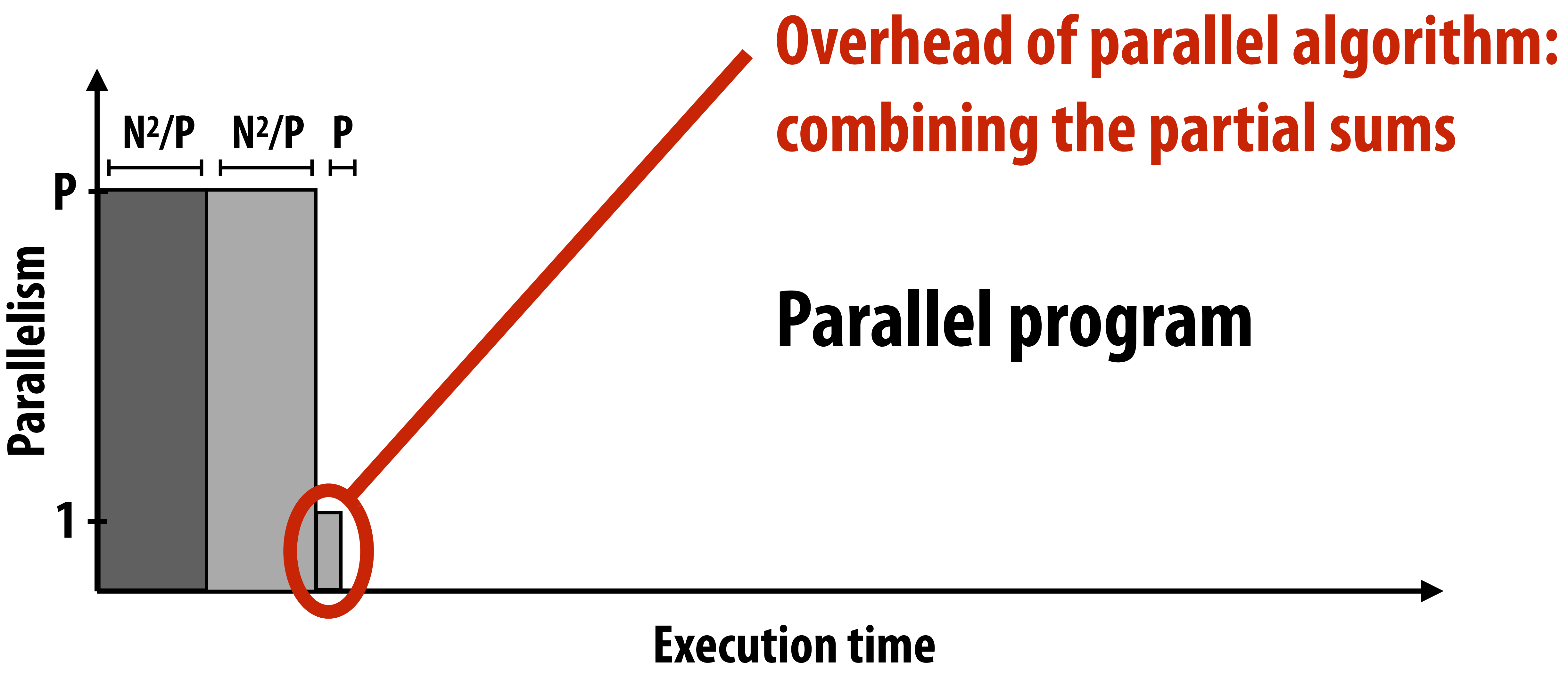


Parallelizing step 2

- **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 $\leq \dfrac{2n^2}{\dfrac{2n^2}{p} + p}$

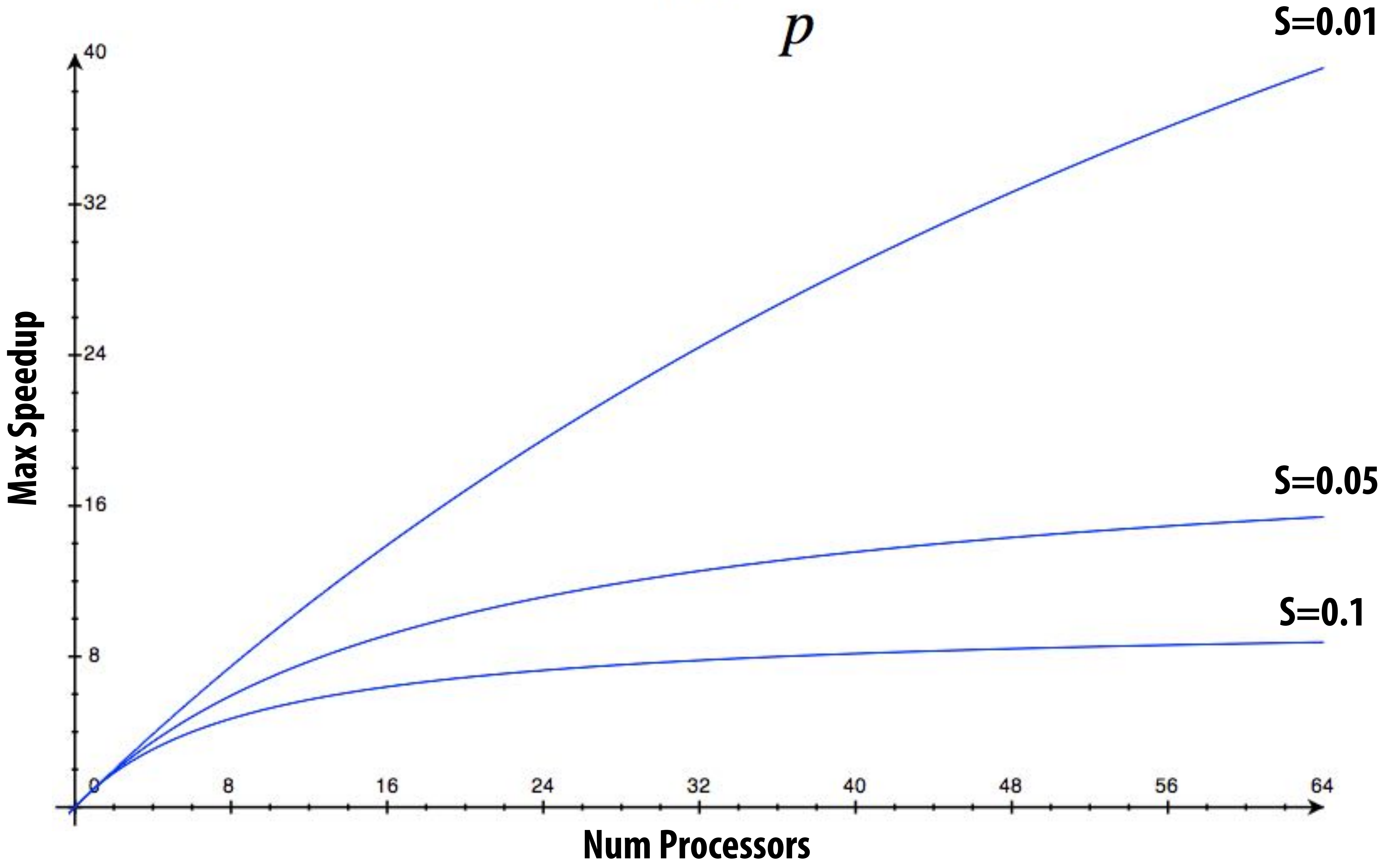
Note: speedup → P when N >> P



Amdahl's law

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

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



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

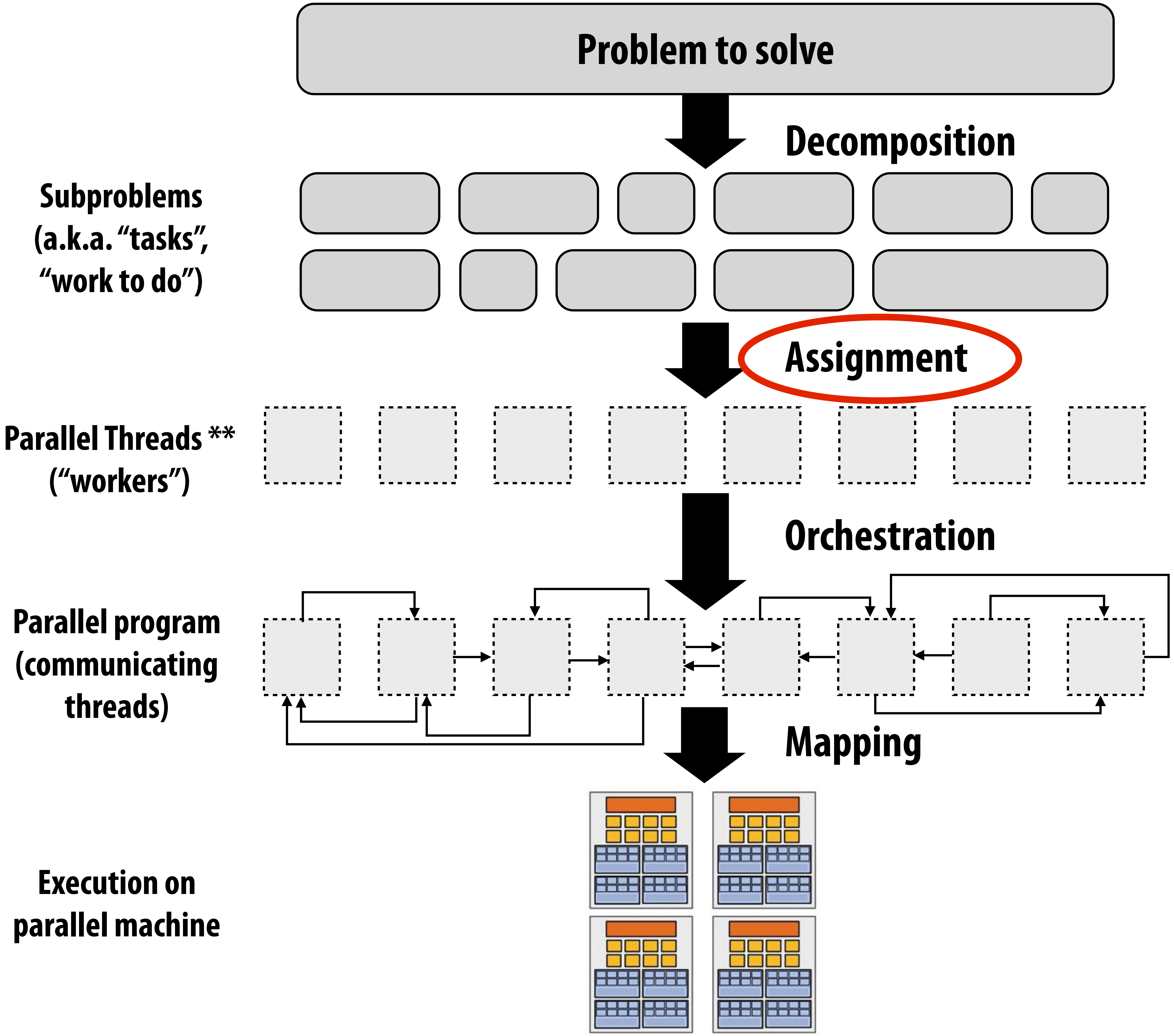
What is max speedup if 0.1% of application is serial?

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 the 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: assigning tasks to workers



** I had to pick a term

Assignment

- Assigning tasks to workers
  - Think of "tasks" as things to do
  - What are "workers"? (Might be threads, program instances, vector lanes, etc.)
- 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();
}
```

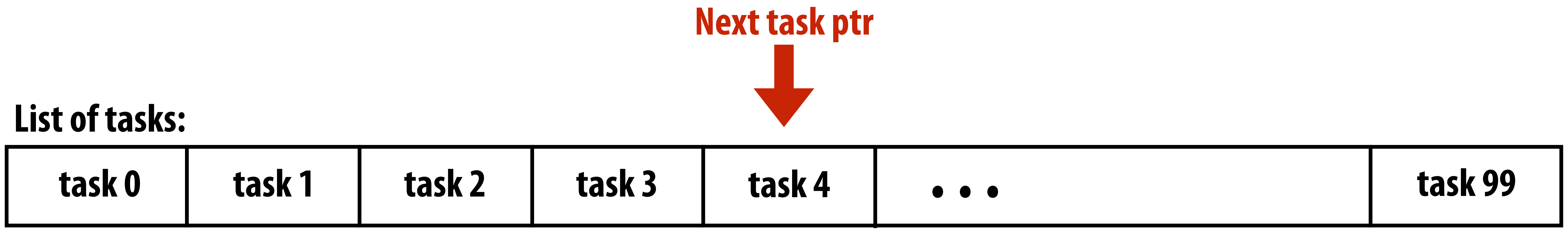
Decomposition of work by loop iteration

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)

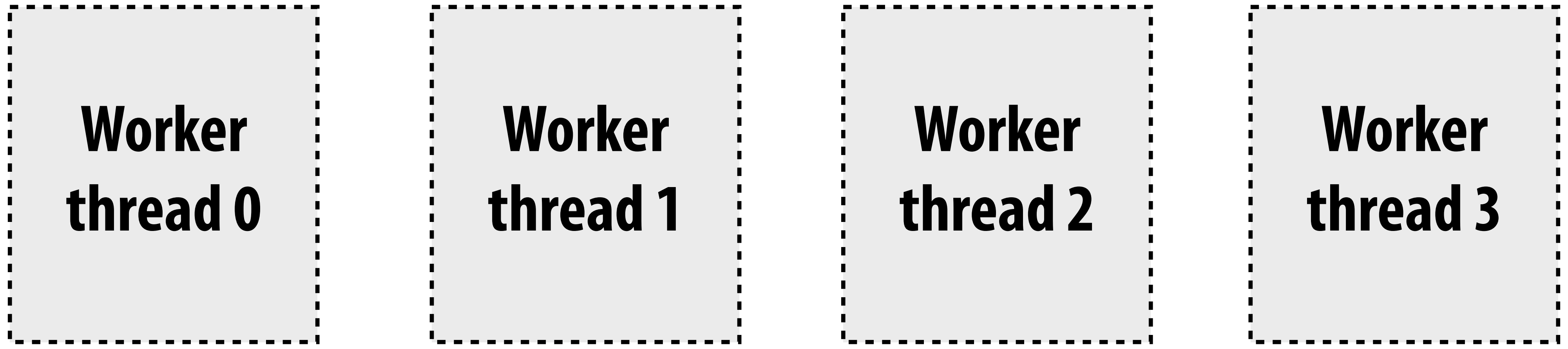
Dynamic assignment using ISPC tasks

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

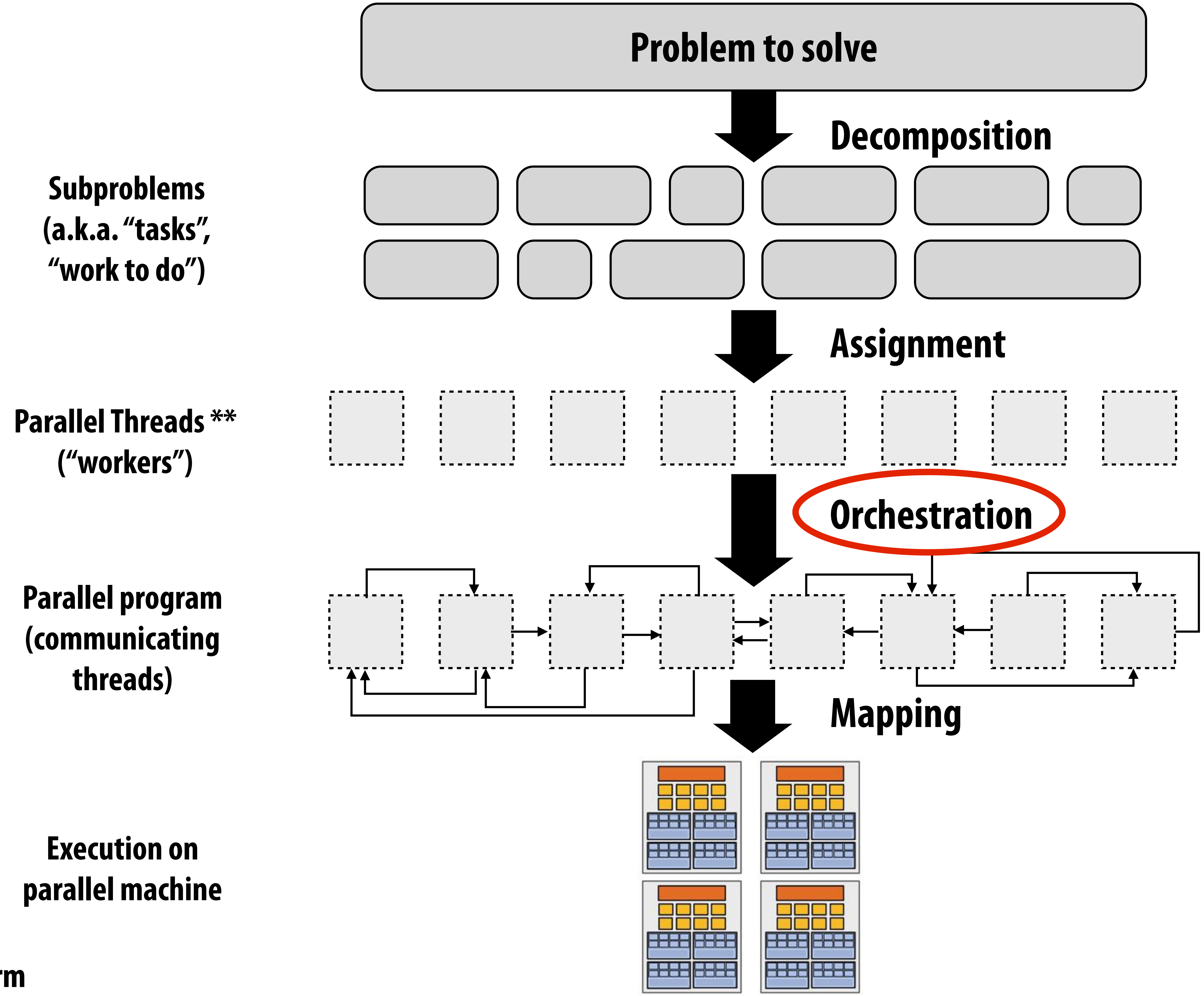
ISPC runtime (invisible to the programmer) assigns tasks to worker threads in a thread pool



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

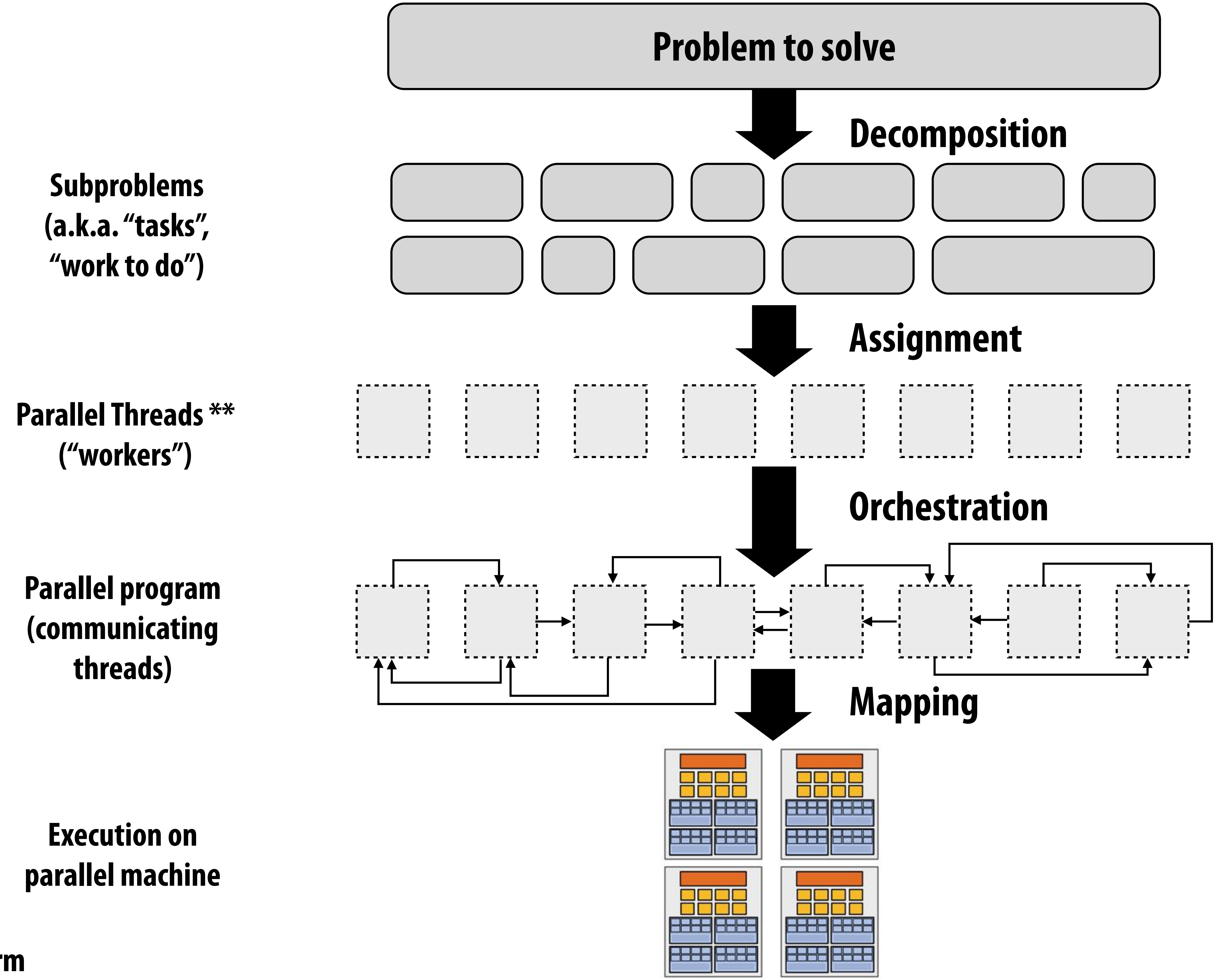
Orchestration

- **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

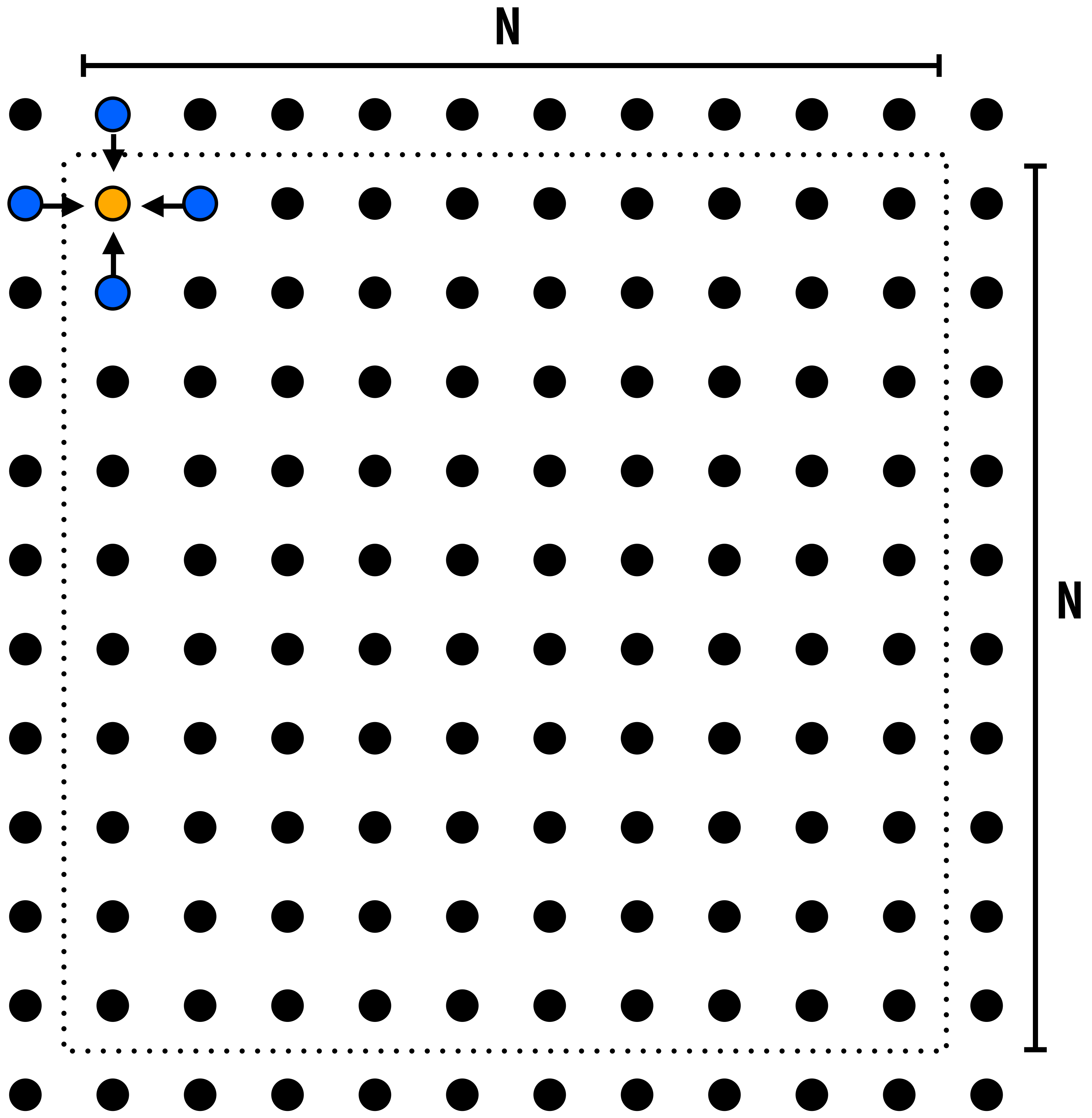
- **Example 3: mapping by the hardware**
  - Map CUDA thread blocks to GPU cores (discussed in a future lecture)

- **Many interesting mapping decisions:**
  - Place <u>related</u> threads (cooperating threads) on the same core
    (maximize locality, data sharing, minimize costs of comm/sync)
  - Place <u>unrelated</u> threads on the same core (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



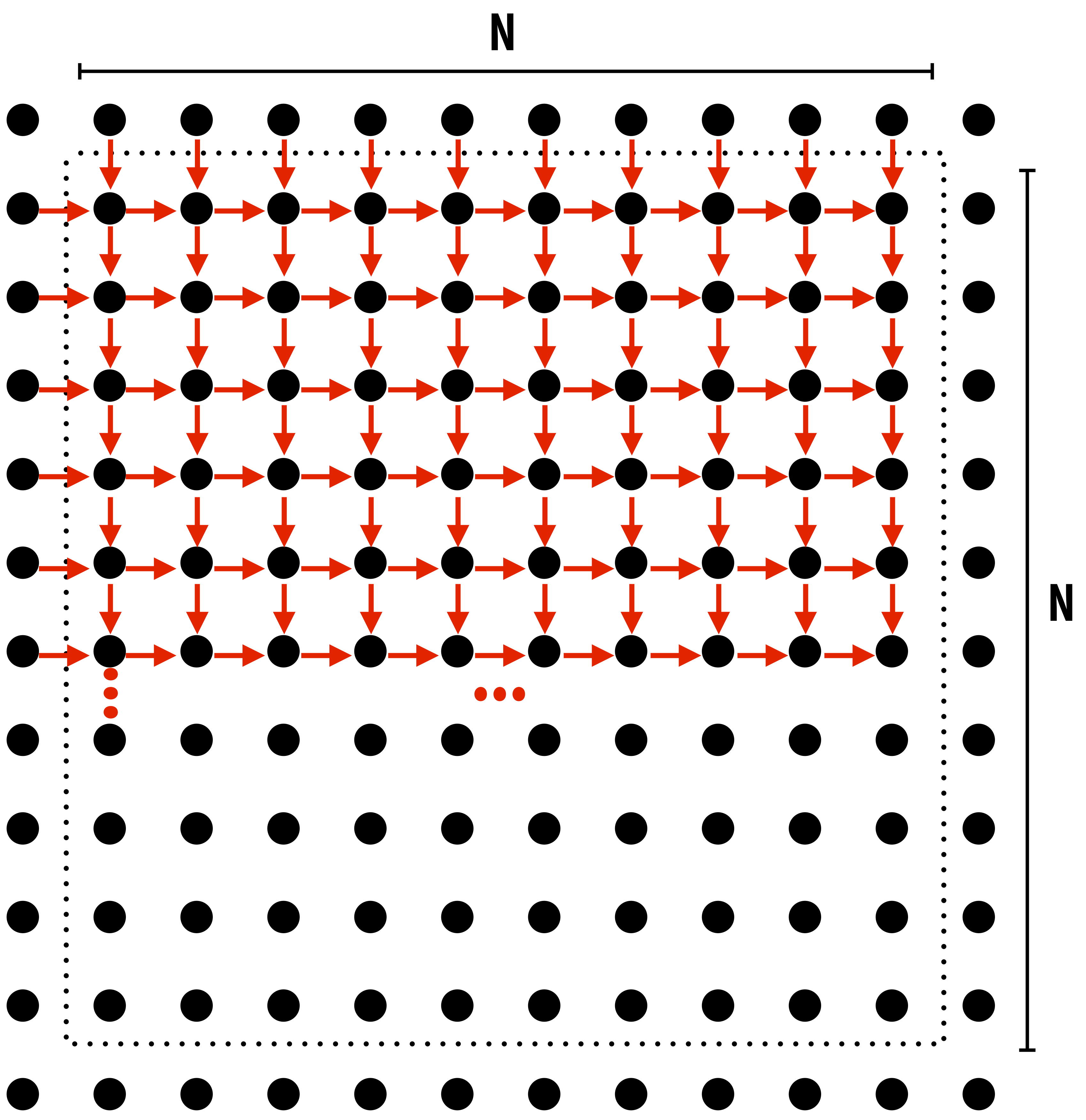

```
A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j]
                       + A[i,j+1] + A[i+1,j]);
```

Grid solver algorithm: find the dependencies

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)

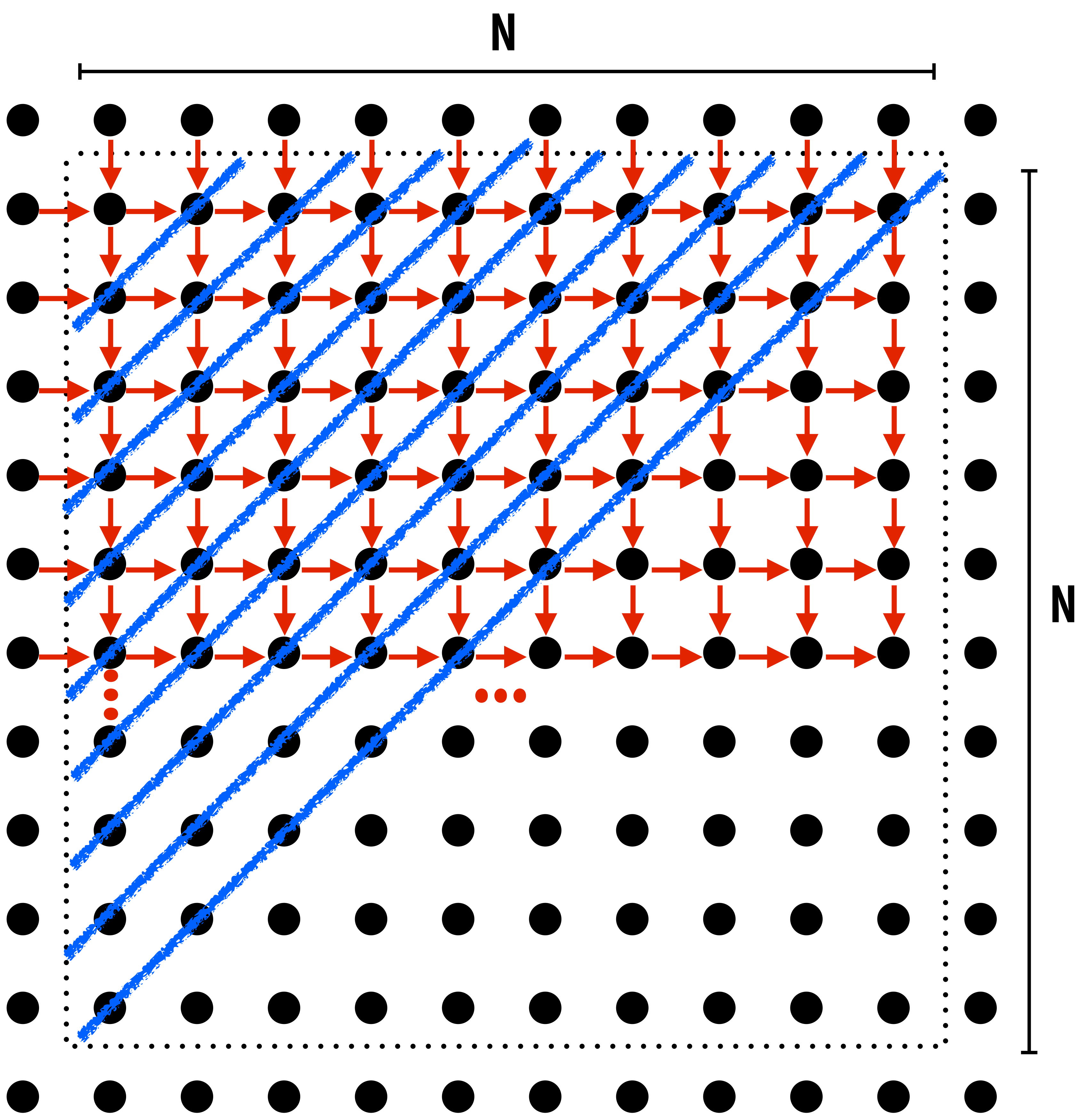


Each row element depends on element to left.

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.

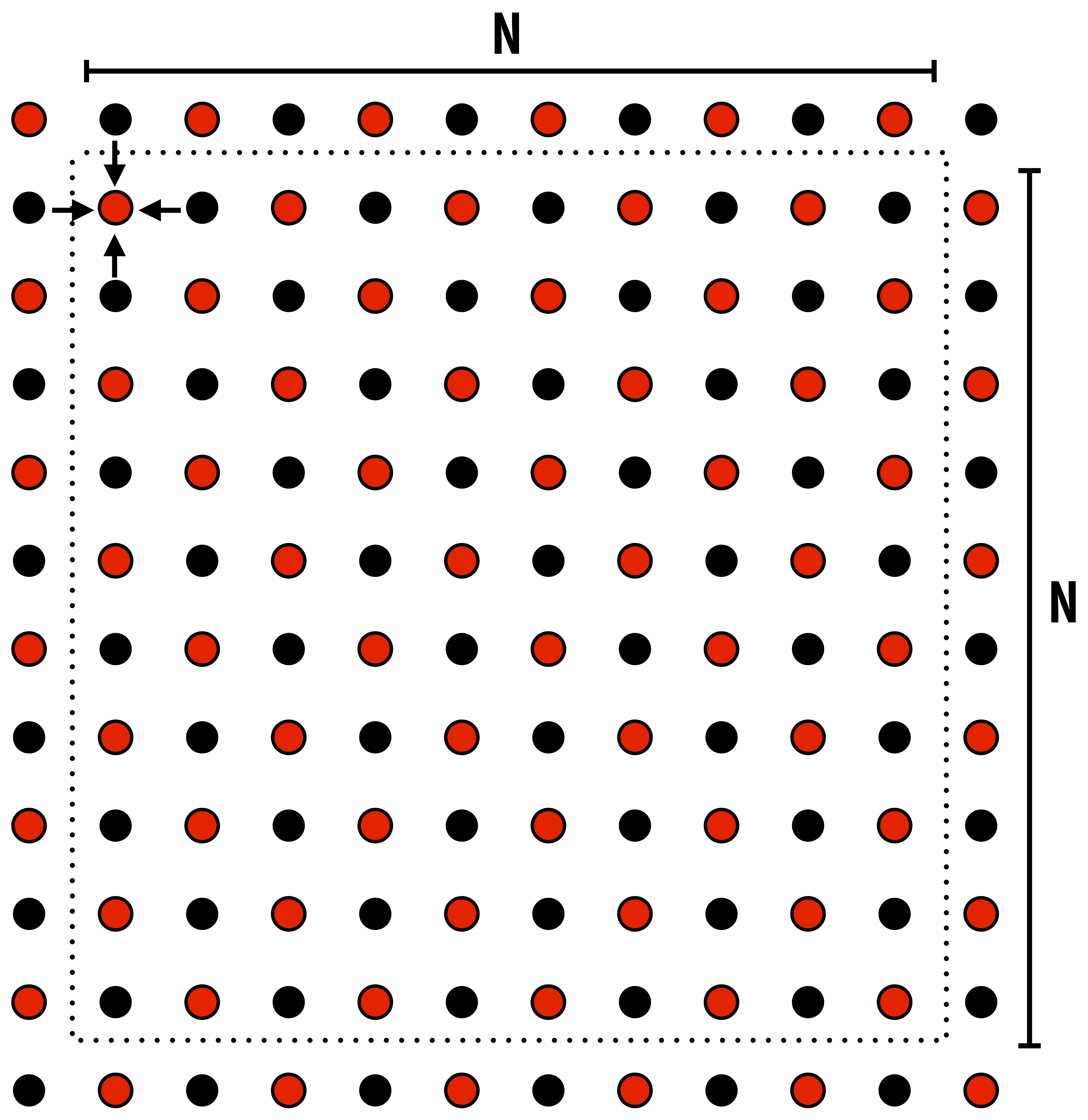
Frequent synchronization (after completing each diagonal)

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



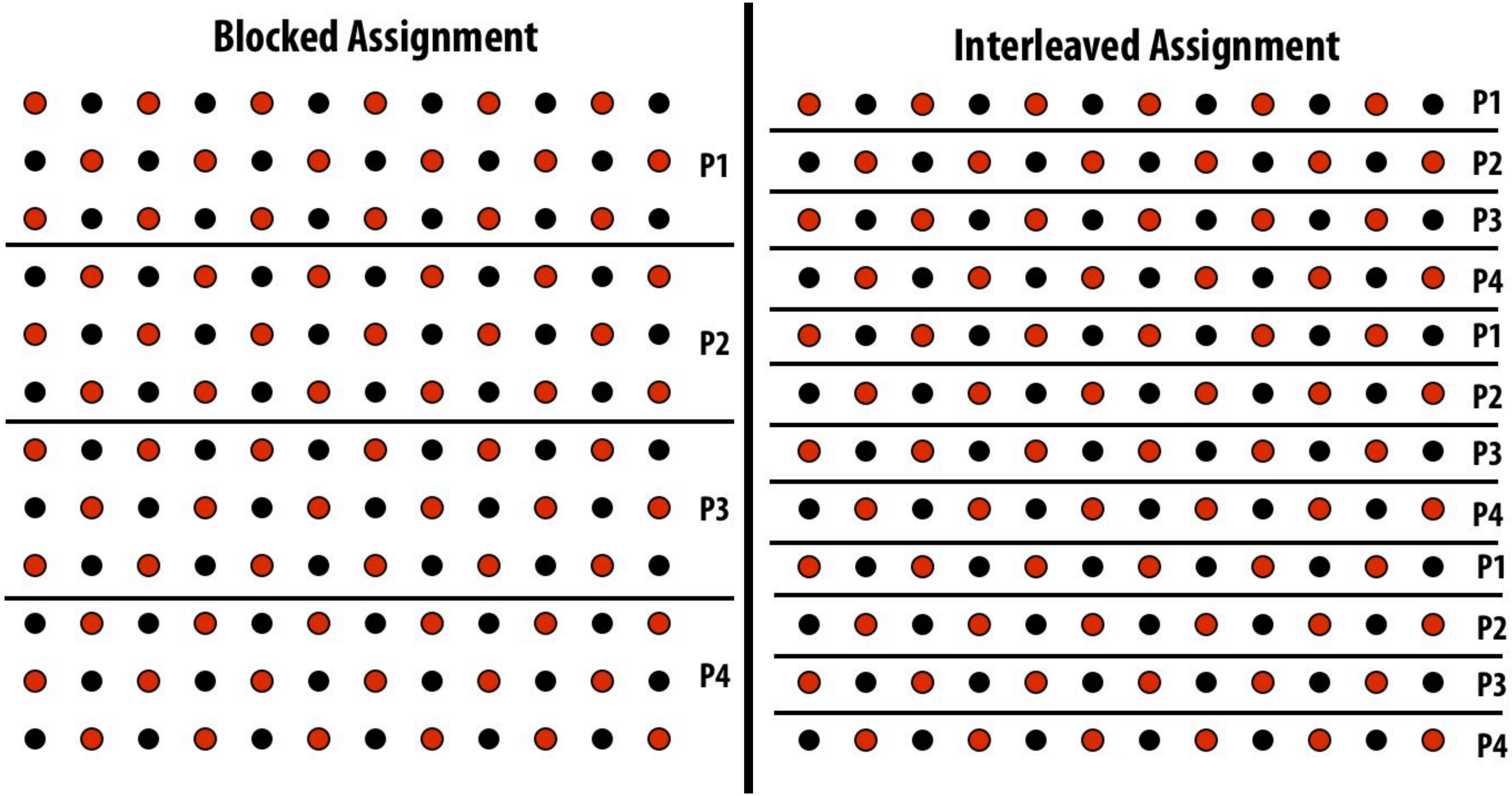
Update all red cells in parallel

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

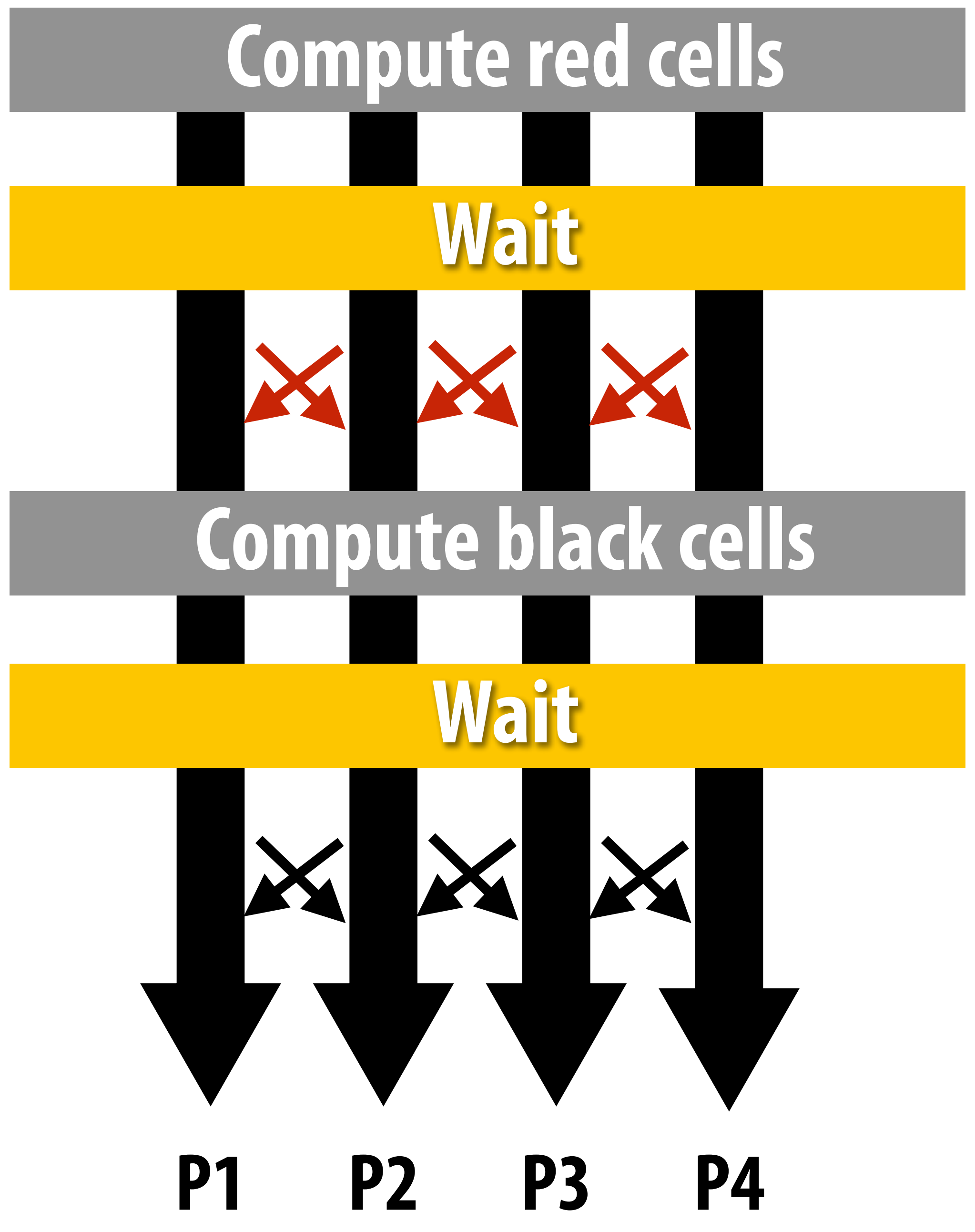


Question: Which is better? Does it matter?

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

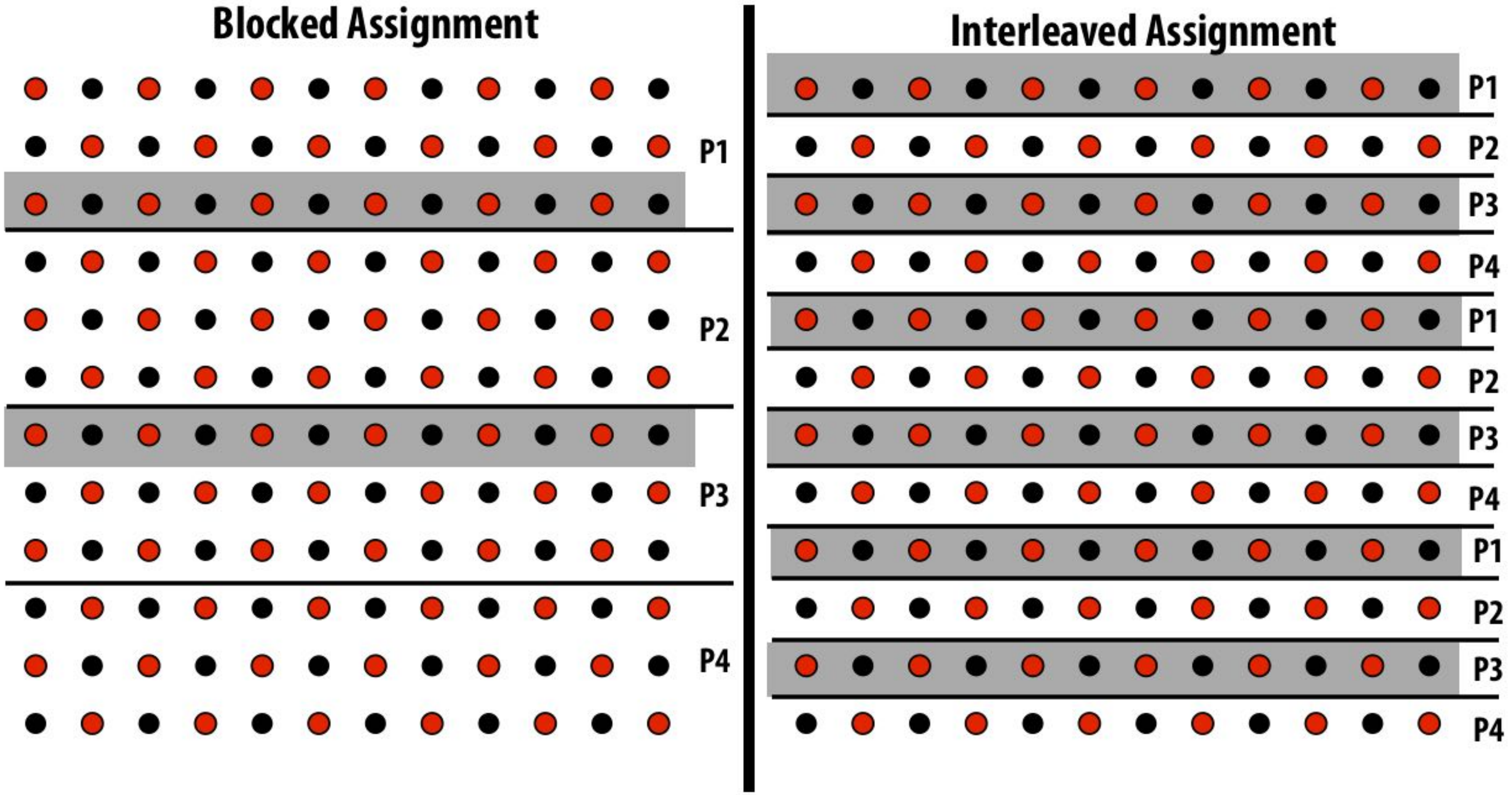
Consider dependencies in the program

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)

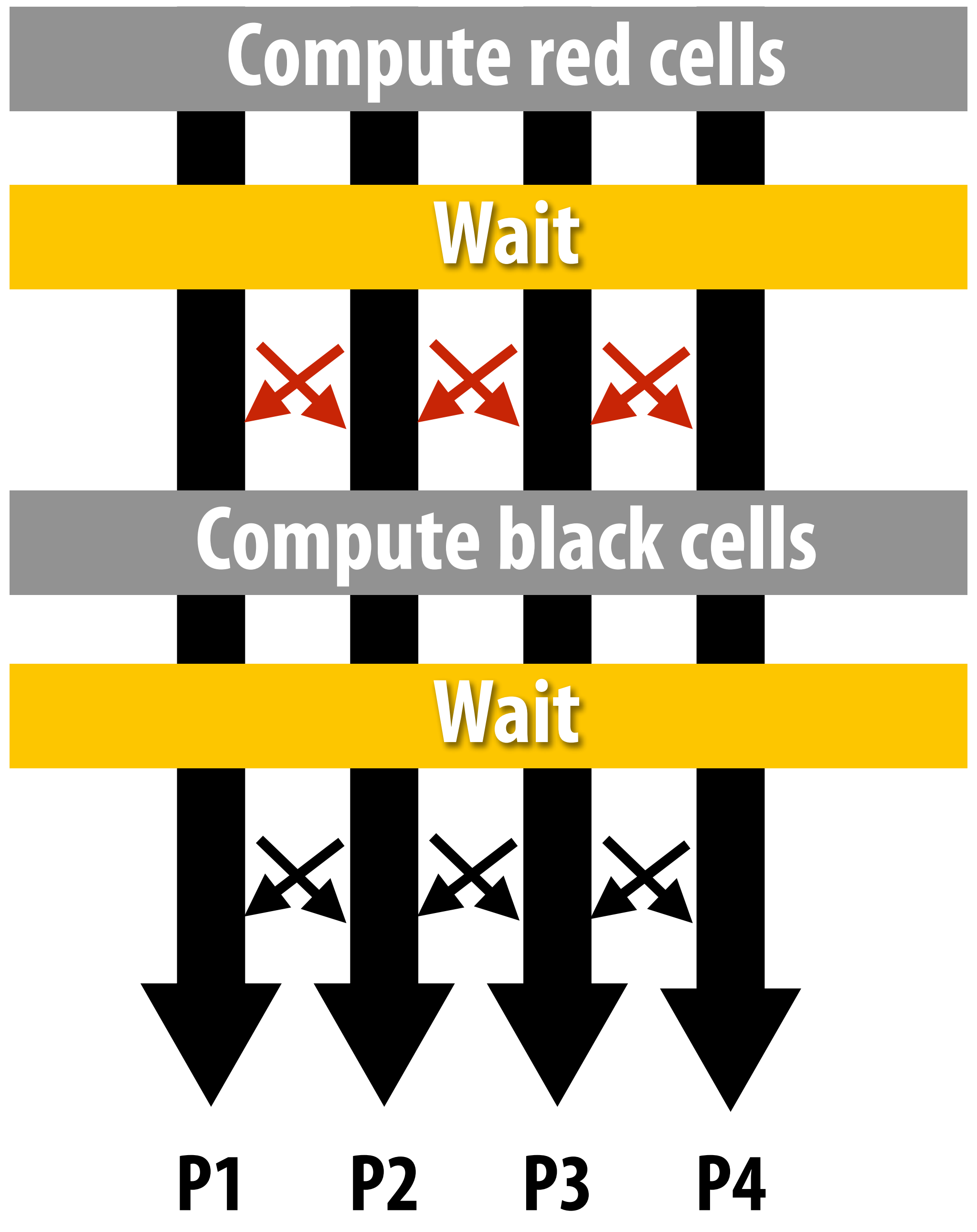
Orchestration: handled by system
(End of for_all block is implicit wait for all workers before returning to sequential control)

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

What's this lock doing here ?????

And these barriers?

Synchronization in a shared address space

Shared address space model (abstraction)

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

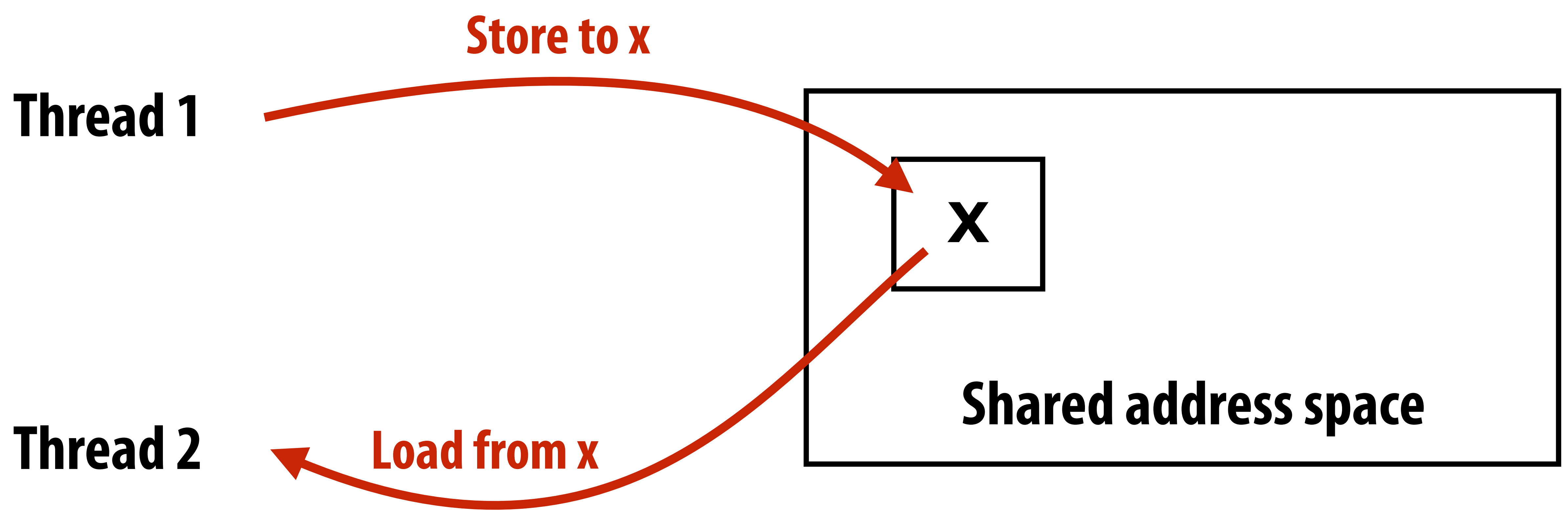
Assume x=0 when threads are launched

Thread 1:

```
// Do work here…

// 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

Shared (among all threads) variables:

```
int x = 0;
Lock my_lock;
```

Thread 1:

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

print(x);
```

Thread 2:

```
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"

Example 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);
```

Summary: 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 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);
   }
}
```

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

Lock for mutual exclusion

Hint

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.

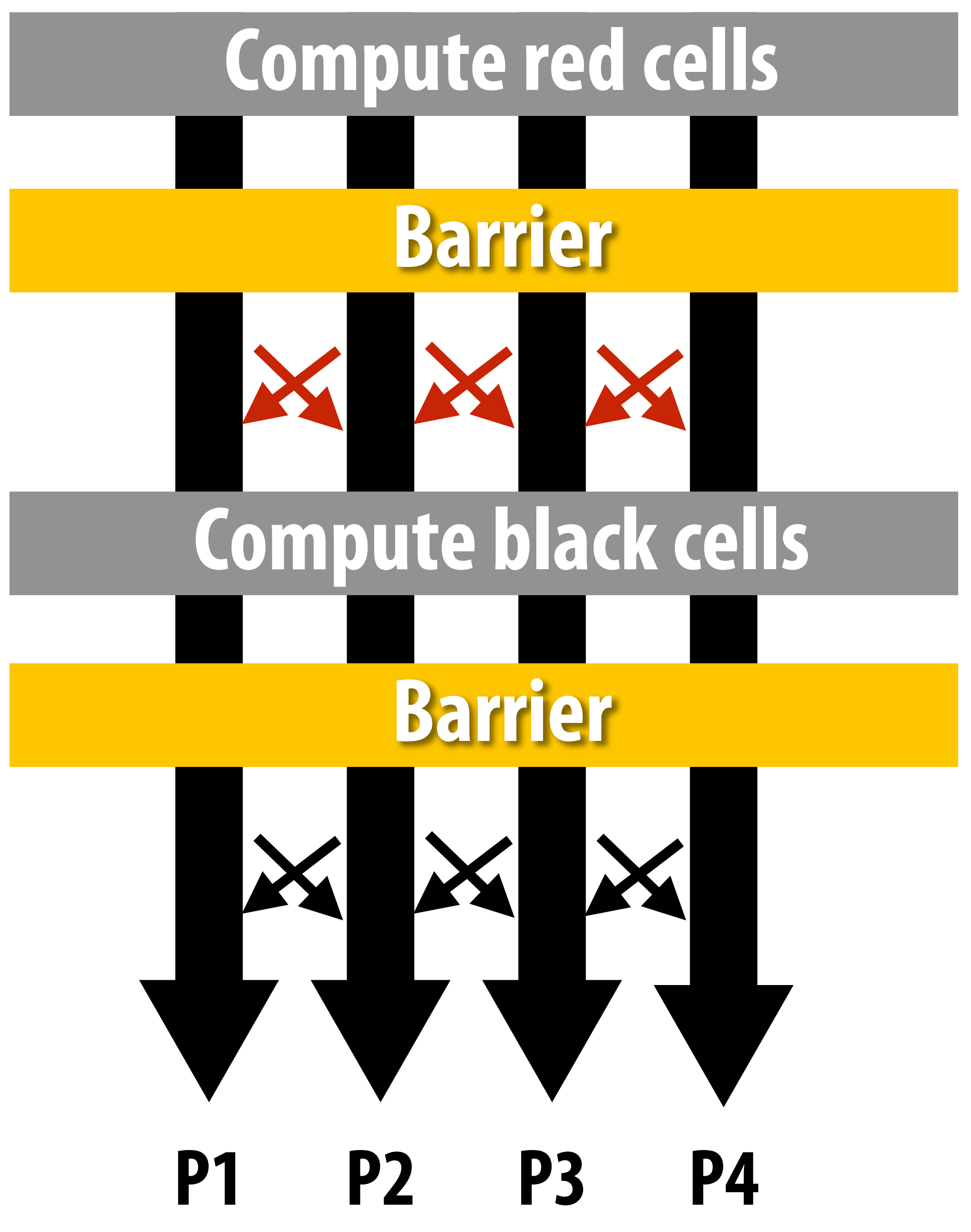
Compute partial sum per worker

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 computation 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

- **Focus today: identifying dependencies**

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