Lecture 4:

Parallel Programming Basics

Parallel Computing
Stanford CS149, Winter 2019

Quiz

```
export void 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.

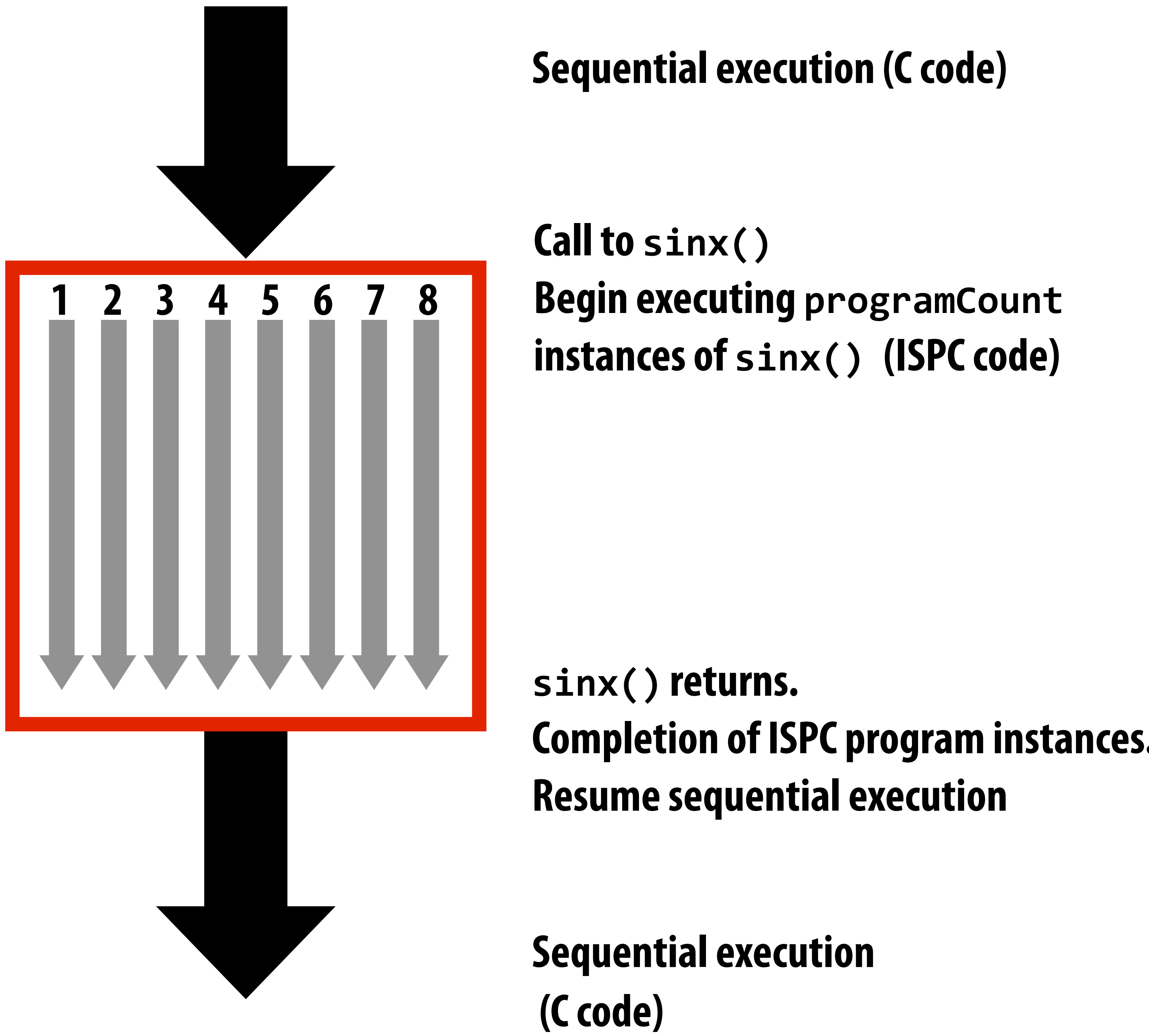
It contains a loop nest.

Which iterations of the loop(s) are parallelized by ISPC? Which are not?

Answer: none

Parallel program instances were created when the sinx() ispc function was 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
sinx(N, terms, x, result);
```



Each instance will run the code in the ispc function serially.
(parallelism exists because there are multiple program instances, not the code that defines an ispc function)

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**

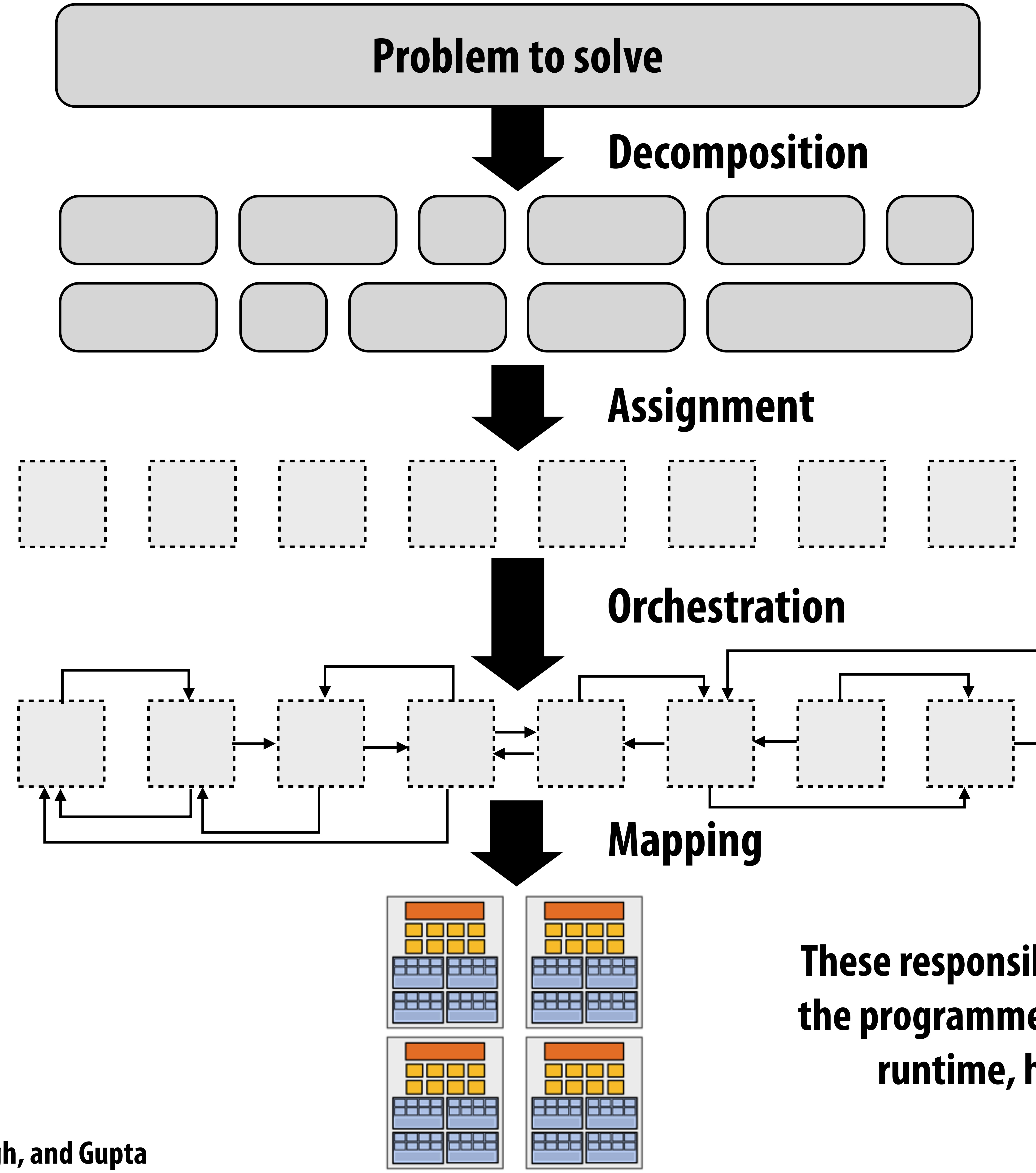
- **Recall one of our main goals is speedup ***

  For a fixed computation:

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

* Other goals include 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")

Parallel program (communicating threads)

Execution on parallel machine

** 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 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

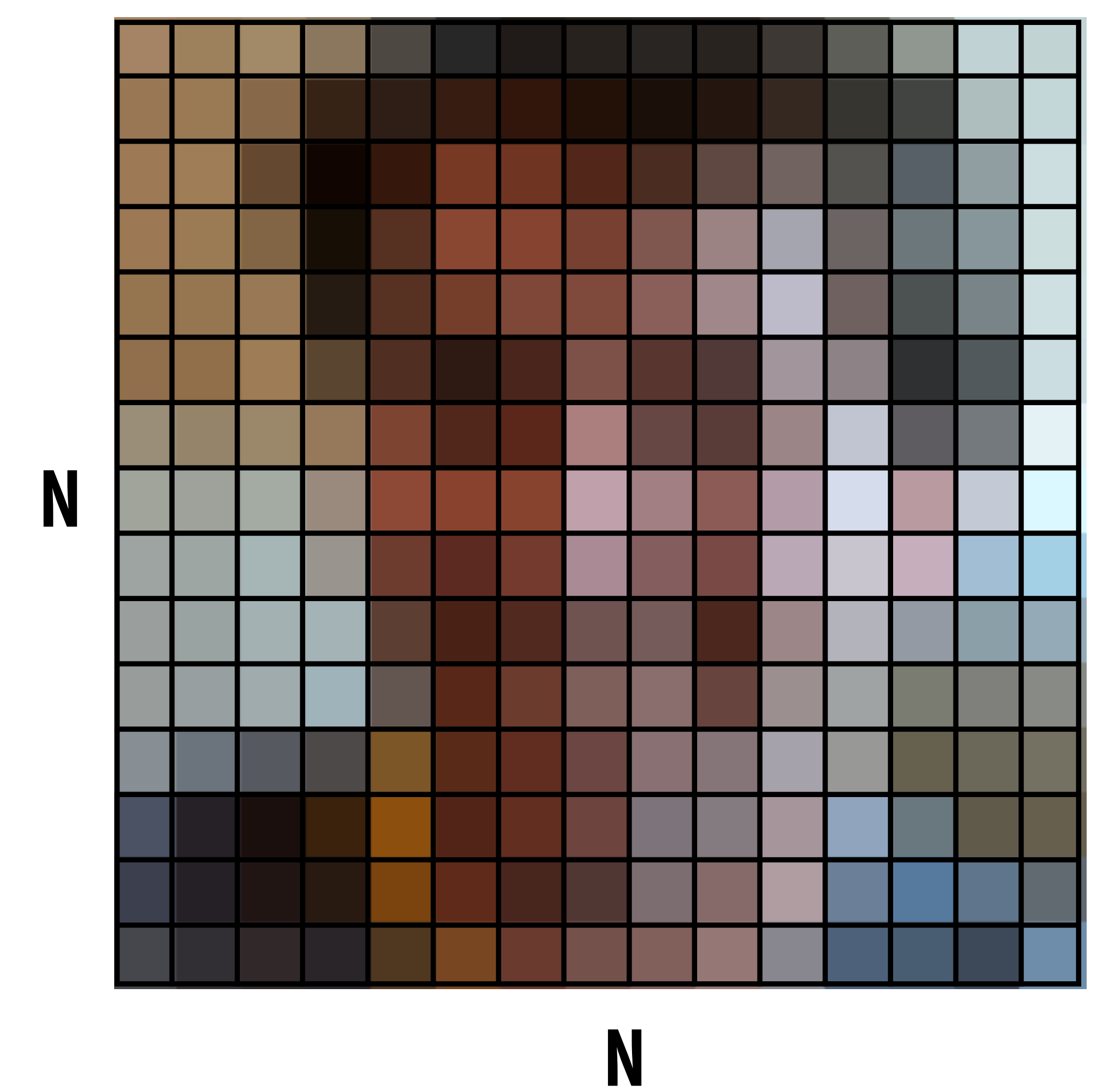
- You run your favorite sequential program...

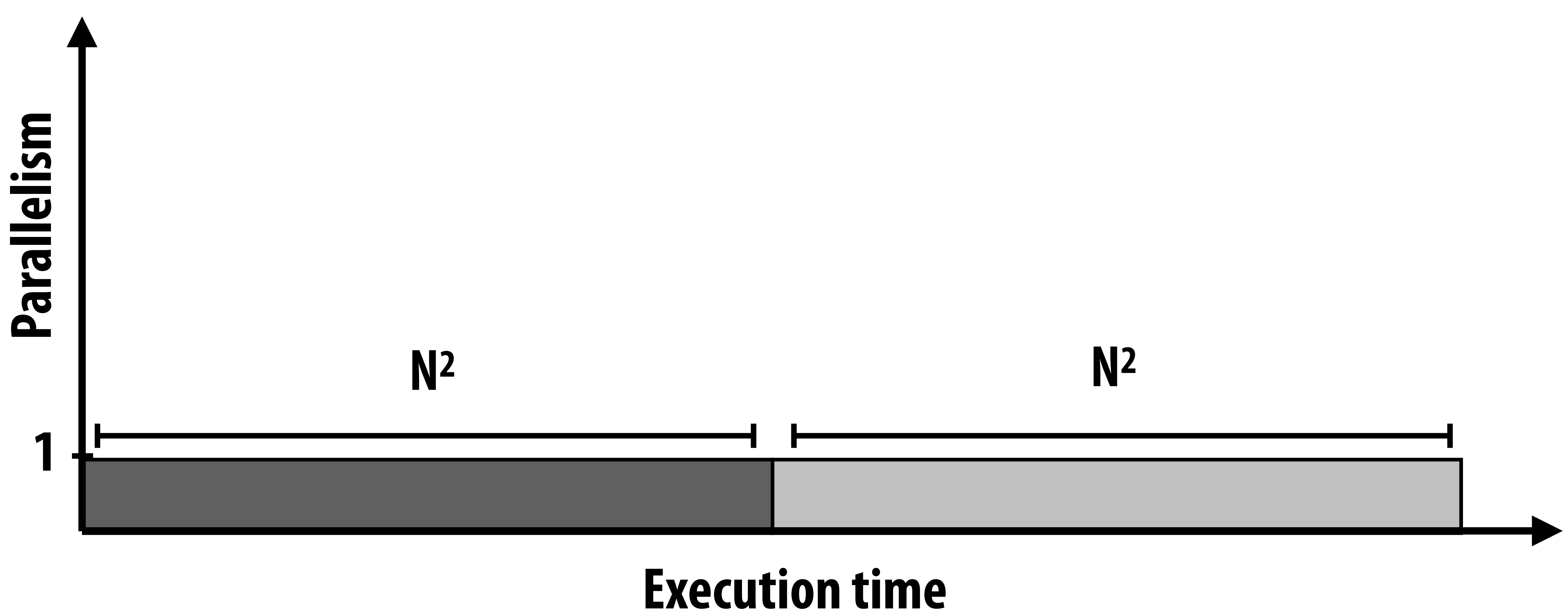
- Let S = the fraction of sequential execution that is inherently sequential (dependencies prevent parallel execution)

- Then maximum speedup due to parallel execution $\leq {}^{1}/_{S}$

A simple example

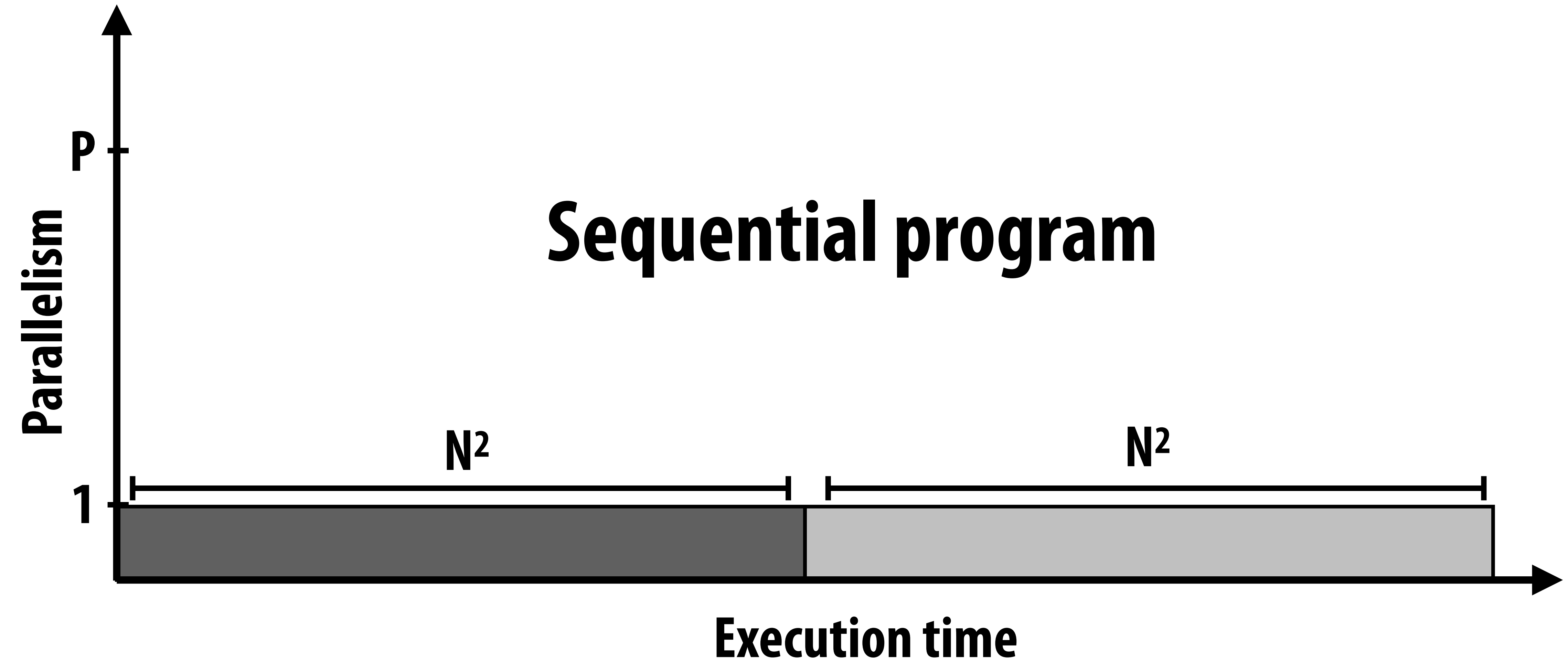
- **Consider a two-step computation on a N x N image**
  - Step 1: double brightness of all pixels (independent computation on each pixel)
  - Step 2: compute average of all pixel values
- **Sequential implementation of program**
  - Both steps take ~ N^2 time, so total time is ~ $2N^2$





First attempt at parallelism (P processors)

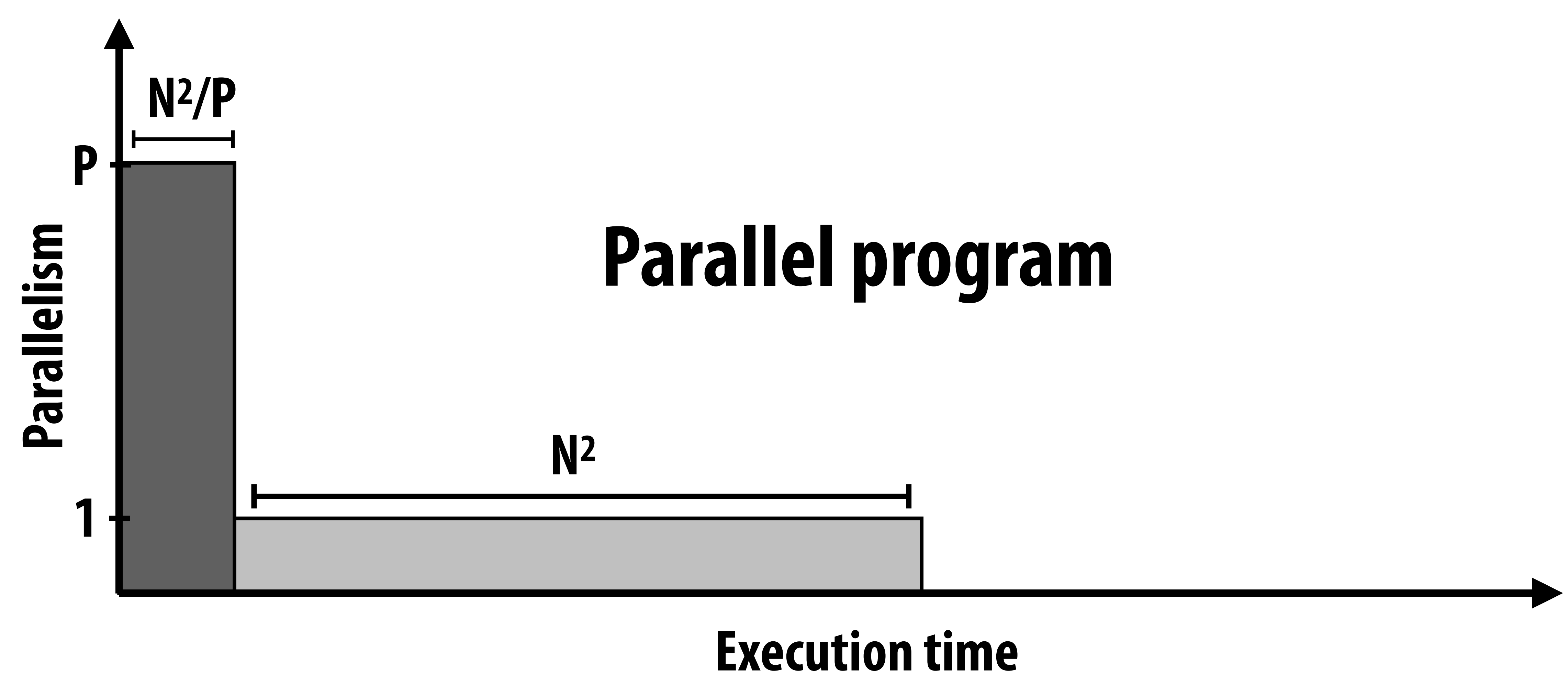
- **Strategy:**
  - Step 1: execute in parallel
    - time for phase 1: N^2/P
  - Step 2: execute serially
    - time for phase 2: N^2



- **Overall performance:**

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

$$\text{Speedup} \leq 2$$

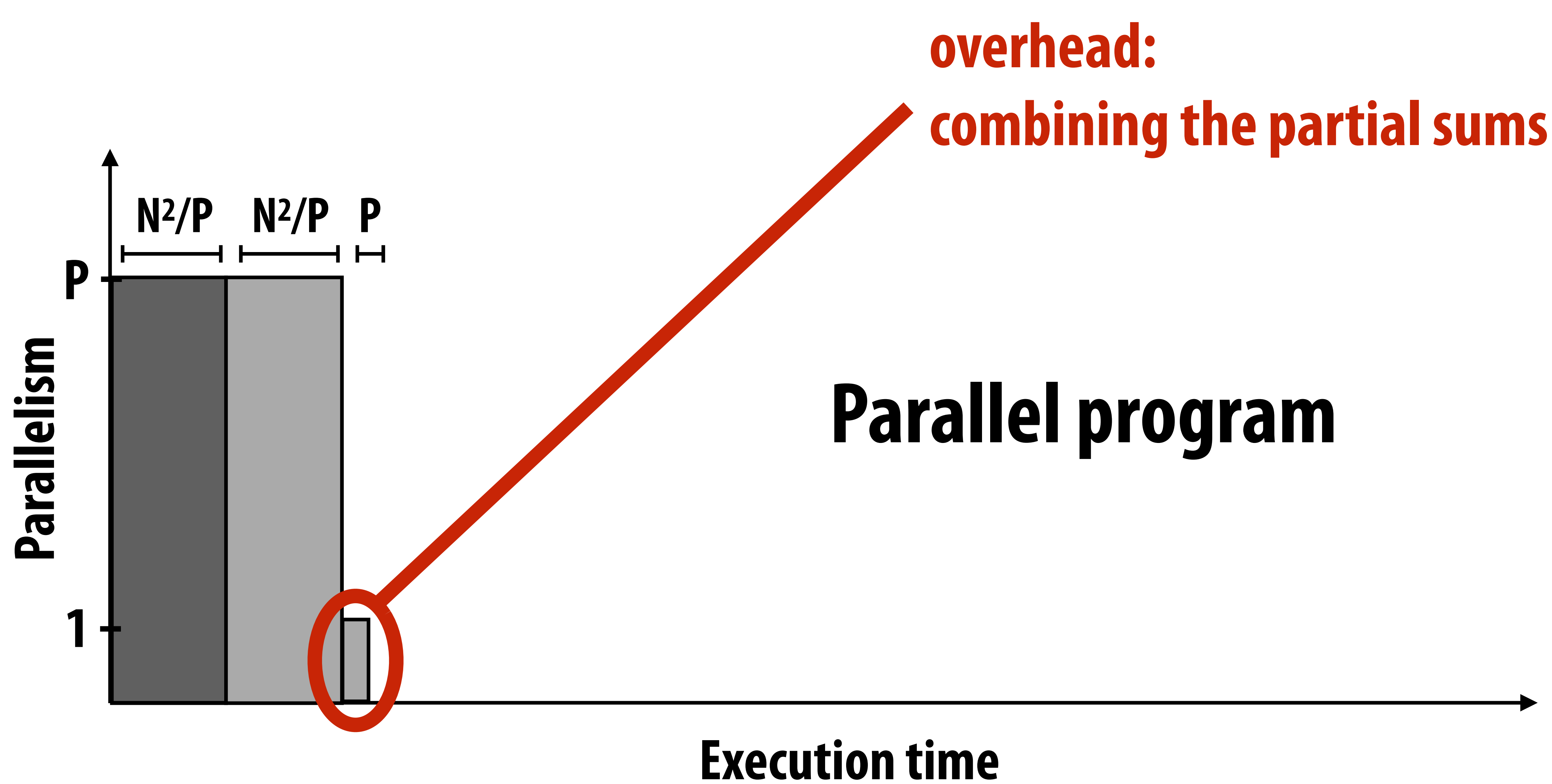


Parallelizing step 2

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

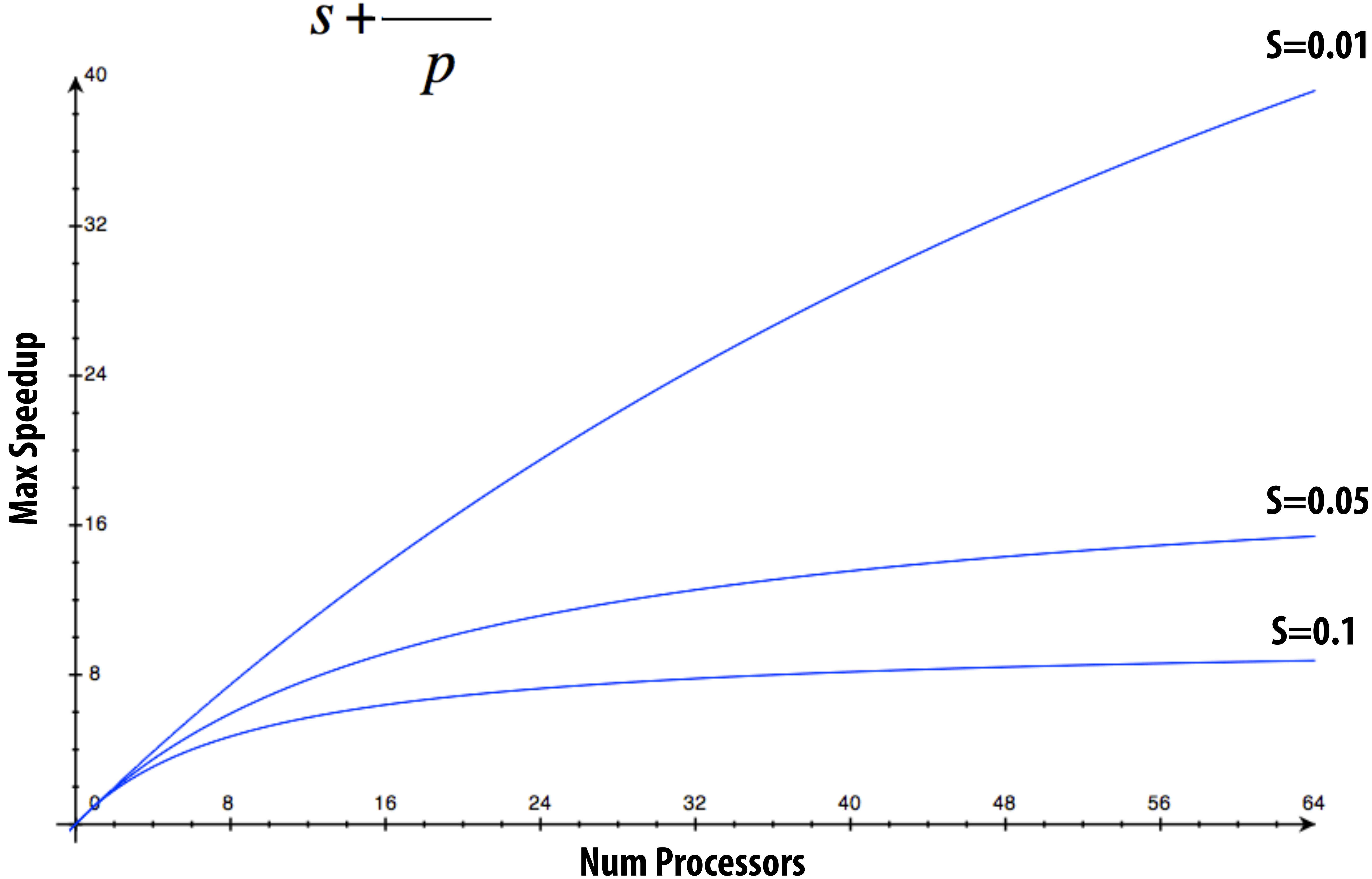
Note: speedup → P when N >> P



Amdahl's law

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

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

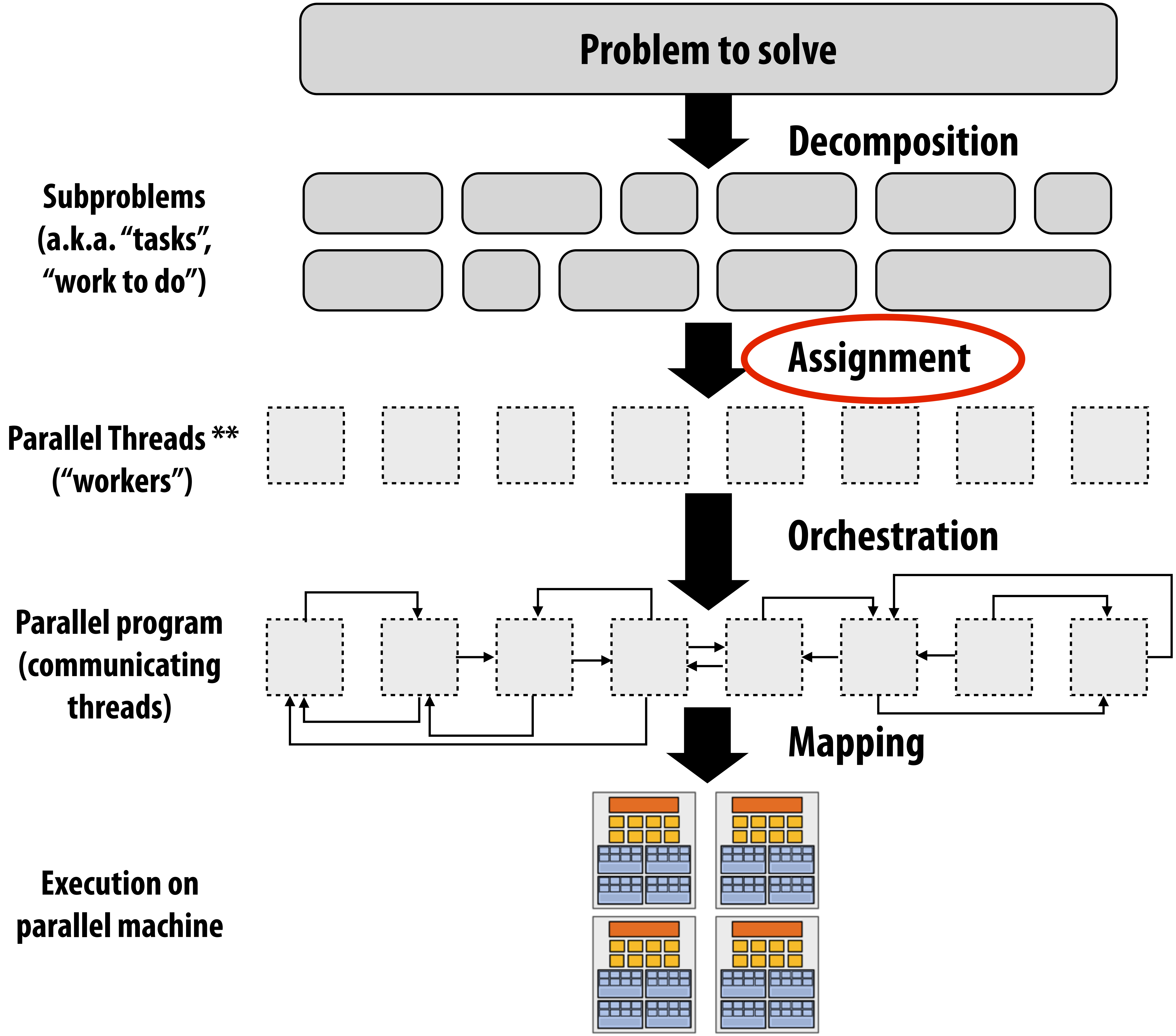


Decomposition

- **Who is responsible for performing decomposition?**
  - 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

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

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

- Goals: balance workload, reduce communication costs
- Can be performed statically, or dynamically during execution
- Although programmer is often responsible for decomposition, many languages/runtimes take responsibility for assignment.

Assignment examples in ISPC

```
export void sinx(
   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 sinx(
   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

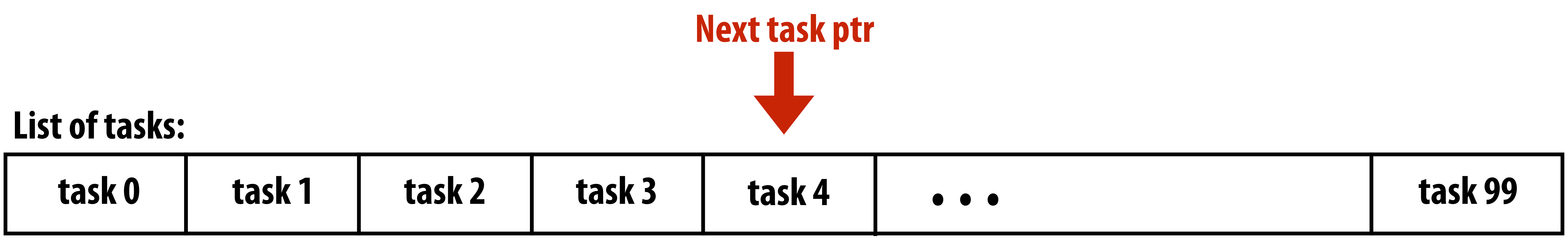
Programmer-managed static assignment

This program assigns iterations to threads in a blocked fashion (first half of array assigned to the spawned thread, second half assigned to main thread)

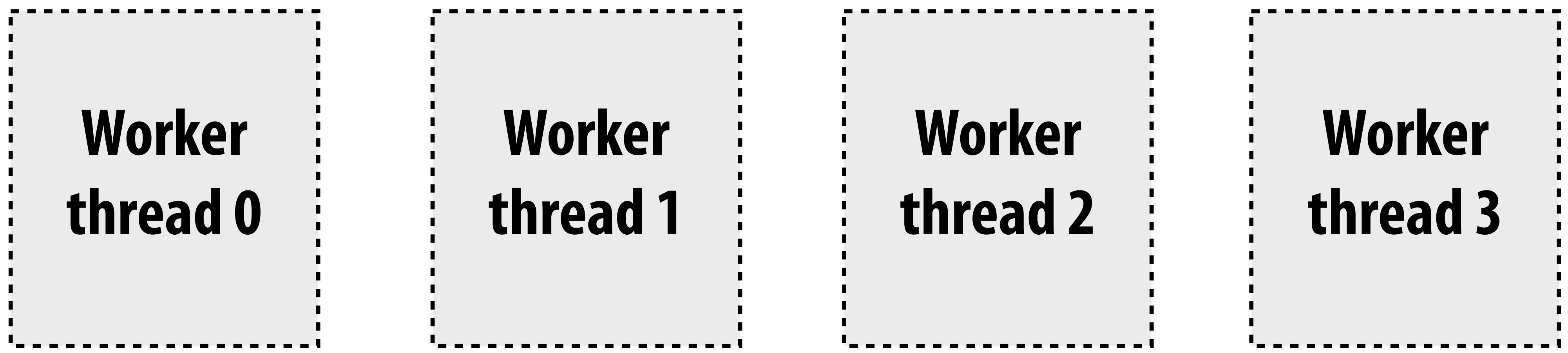
Dynamic assignment using ISPC tasks

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

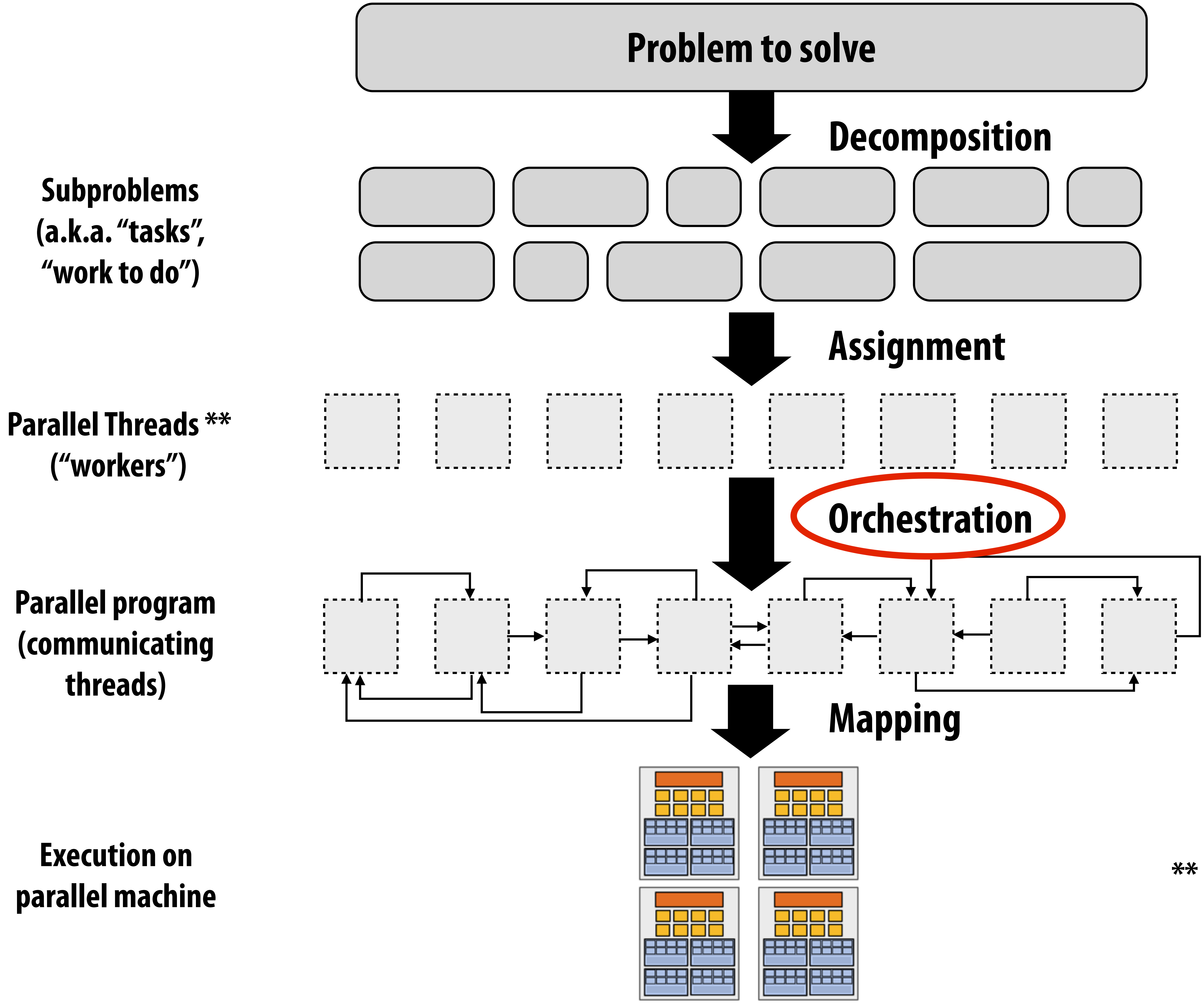
ISPC runtime assign tasks to worker threads



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

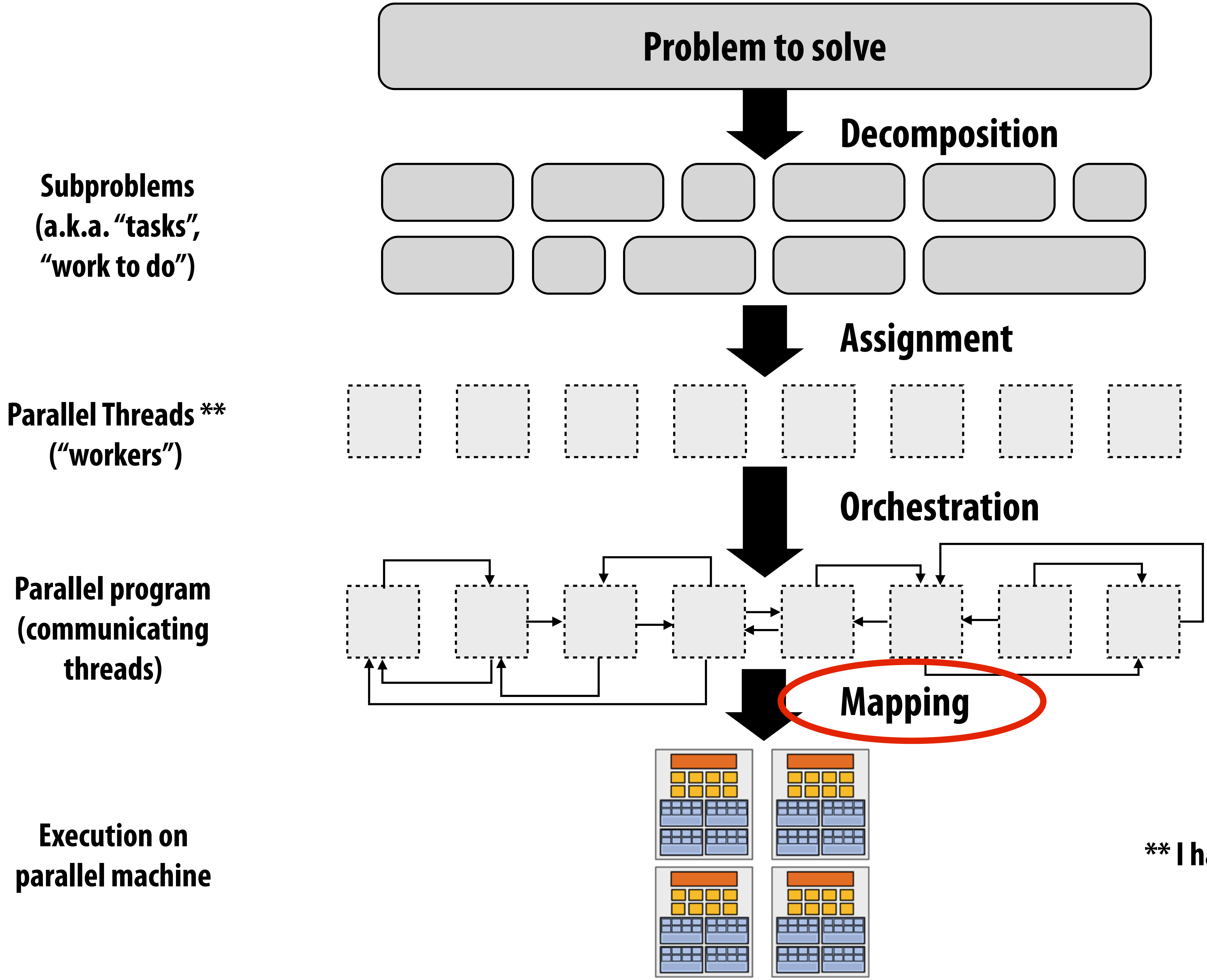
Orchestration

- **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, 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 pthread 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 (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

Example: last class I asked you a question about mapping

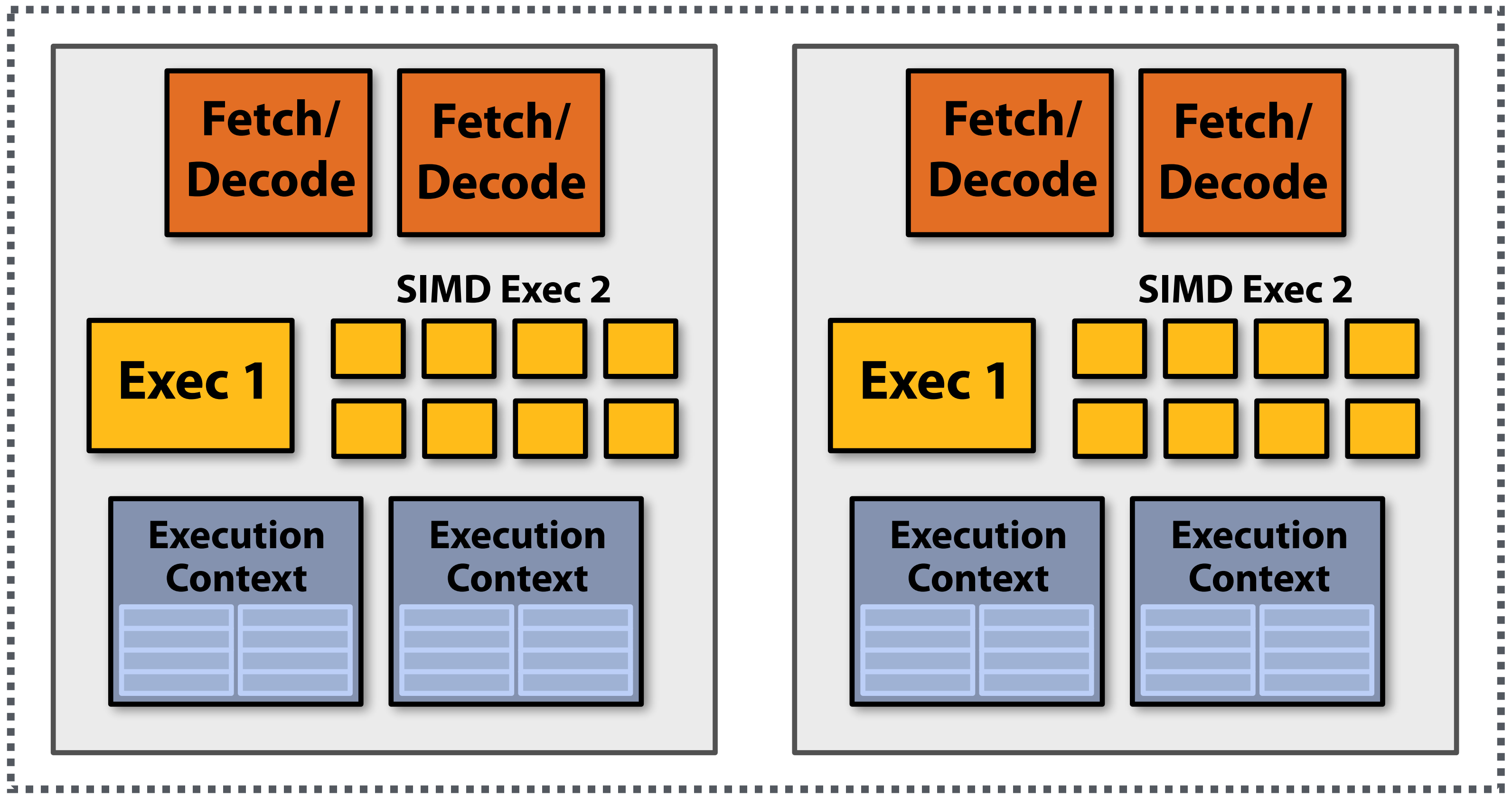
- Consider an application that creates two threads
- The application runs on the processor shown below
  - Two cores, two-execution contexts per core, up to instructions per clock, one instruction is an 8-wide SIMD instruction.

- Question: "who" is responsible for mapping the applications's pthreads to the processor's thread execution contexts?

  Answer: the operating system

- Question: If you were implementing the OS, how would to map the two threads to the four execution contexts?

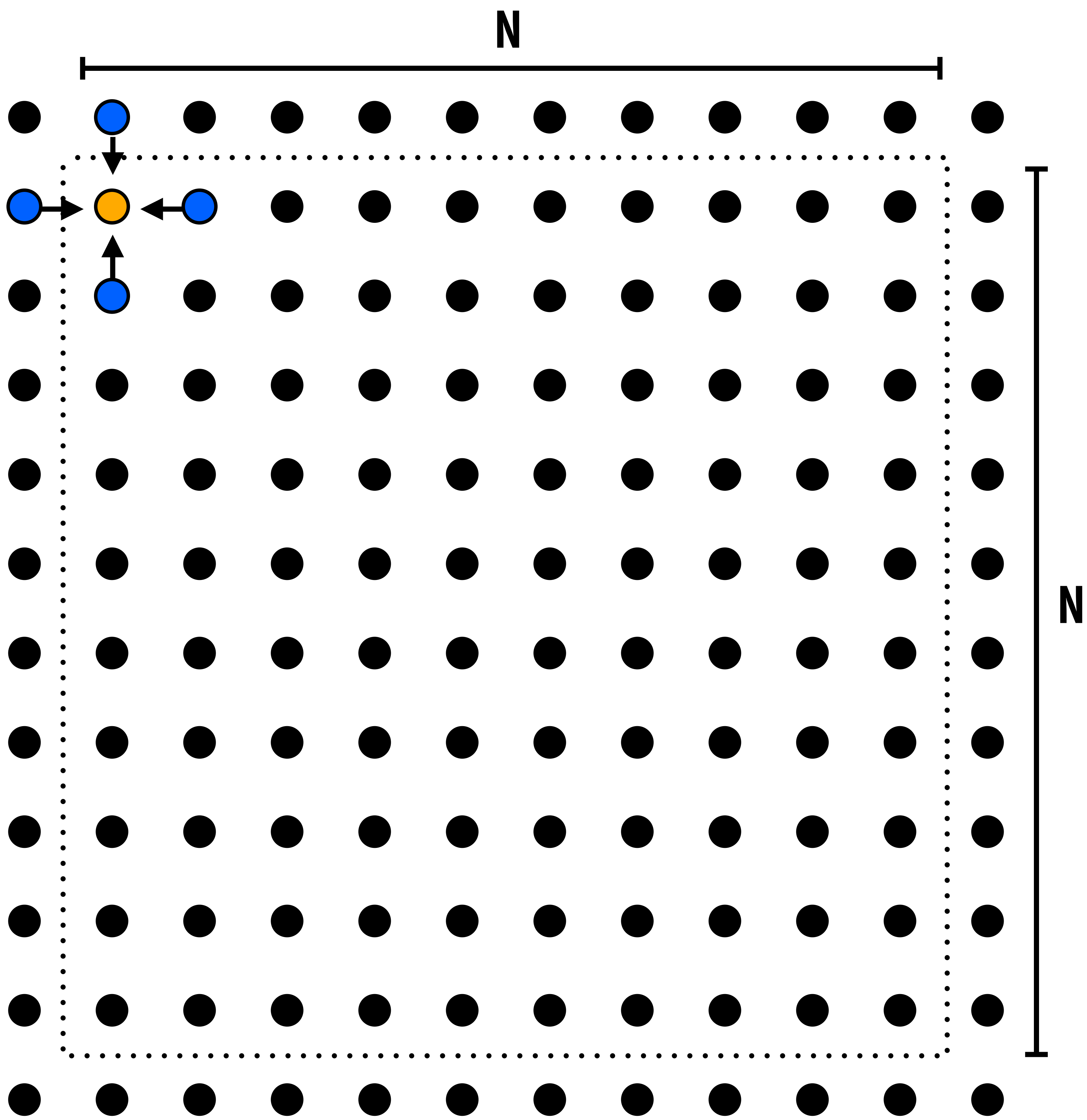
- Another question: How would you map threads to execution contexts if your C program spawned five threads?



A parallel programming example

A 2D-grid based solver

- Solve partial differential equation (PDE) on (N+2) x (N+2) grid
- Iterative solution
  - 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]);
```

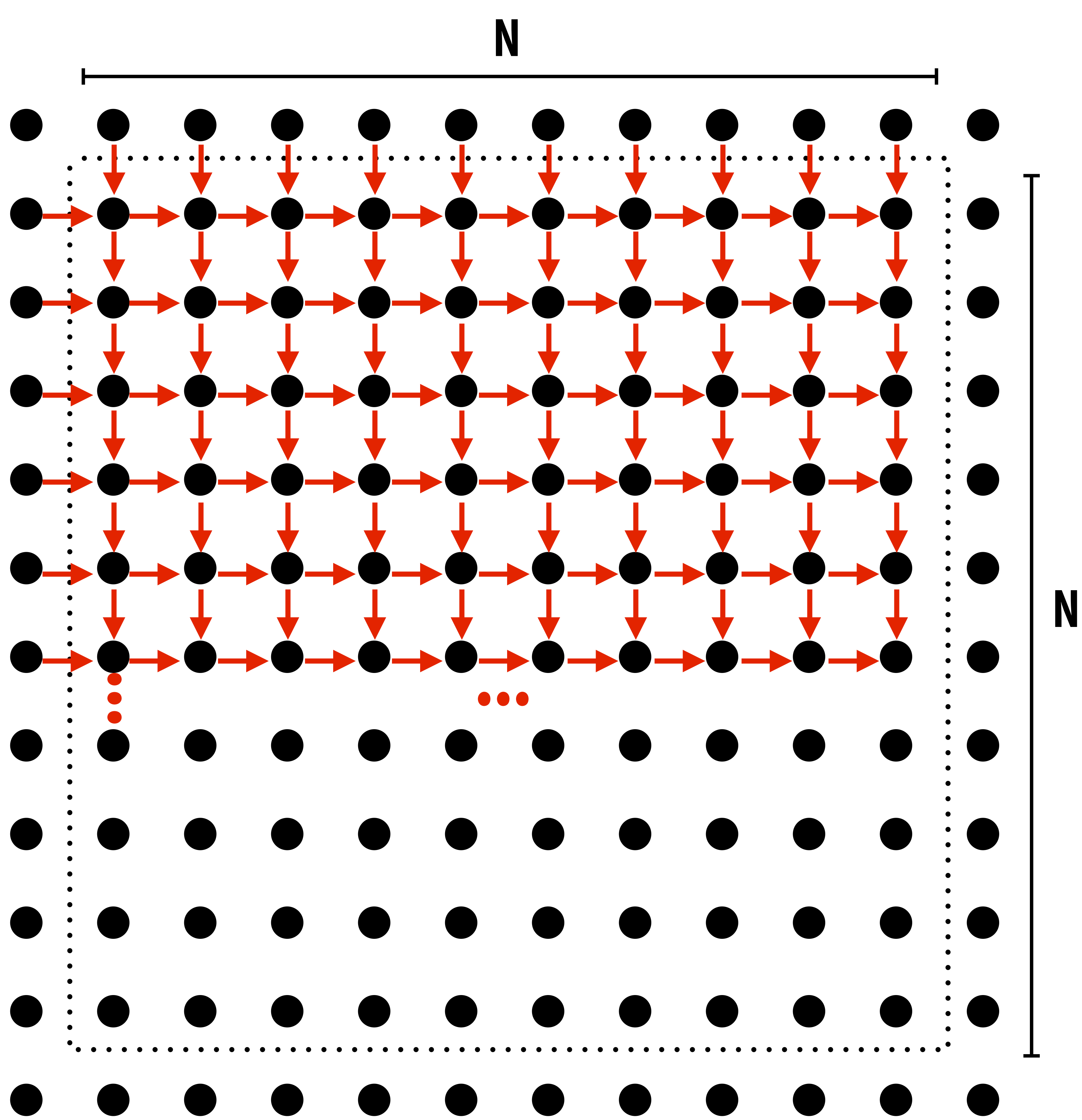
Grid solver algorithm

C-like pseudocode for sequential algorithm is provided below

```
const int n;
float* A;                      // assume allocated to 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;
  }
}
```

Grid solver example from: Culler, Singh, and Gupta

Step 1: identify dependencies (problem decomposition phase)

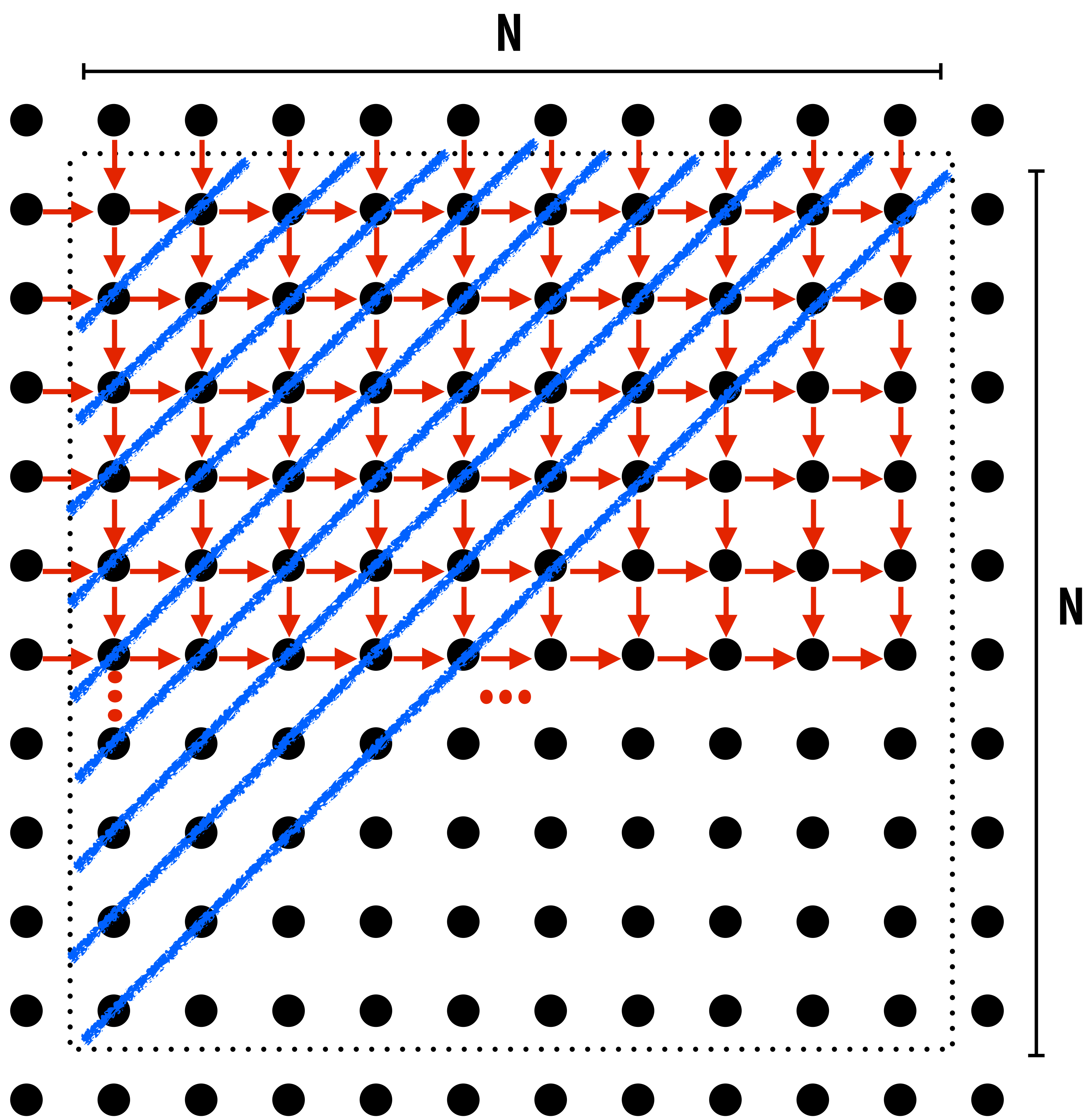


Each row element depends on element to left.

Each row depends on previous row.

Note: the dependencies illustrated on this slide are 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

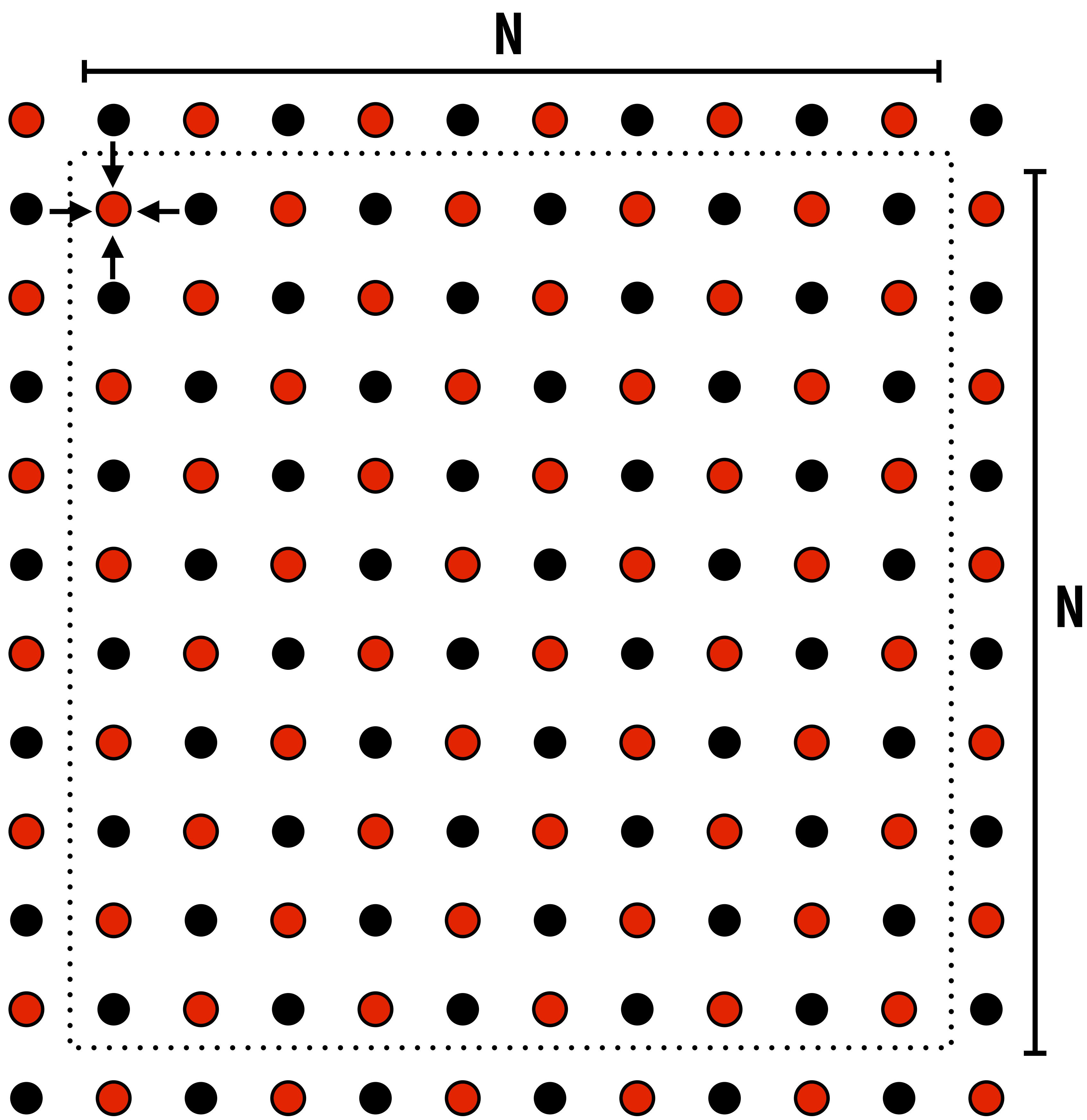
Not much parallelism at beginning and end of computation.

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 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 Gauss-Seidel method for solving a linear system to realize this change is permissible for the application

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

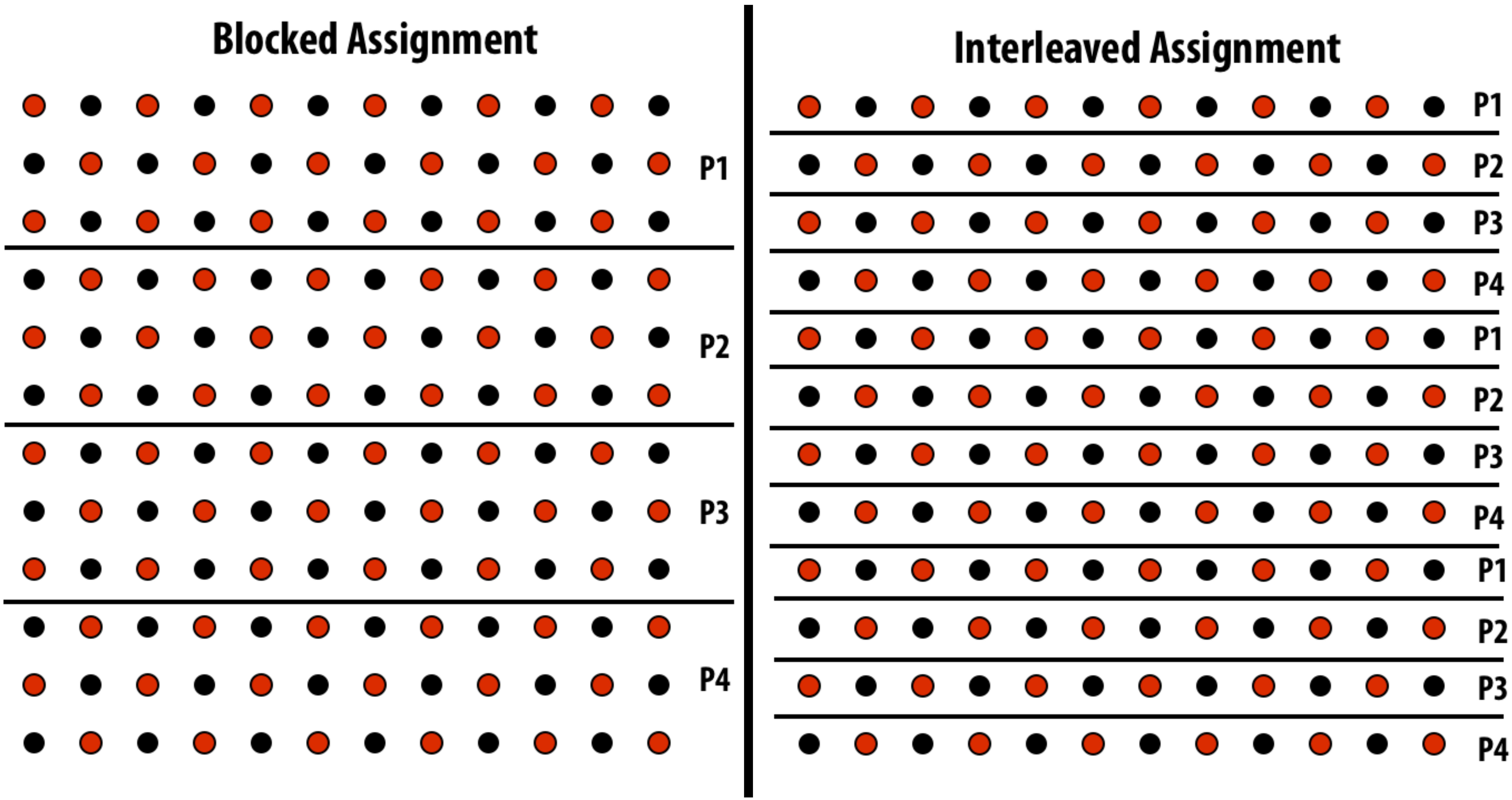


Update all red cells in parallel

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

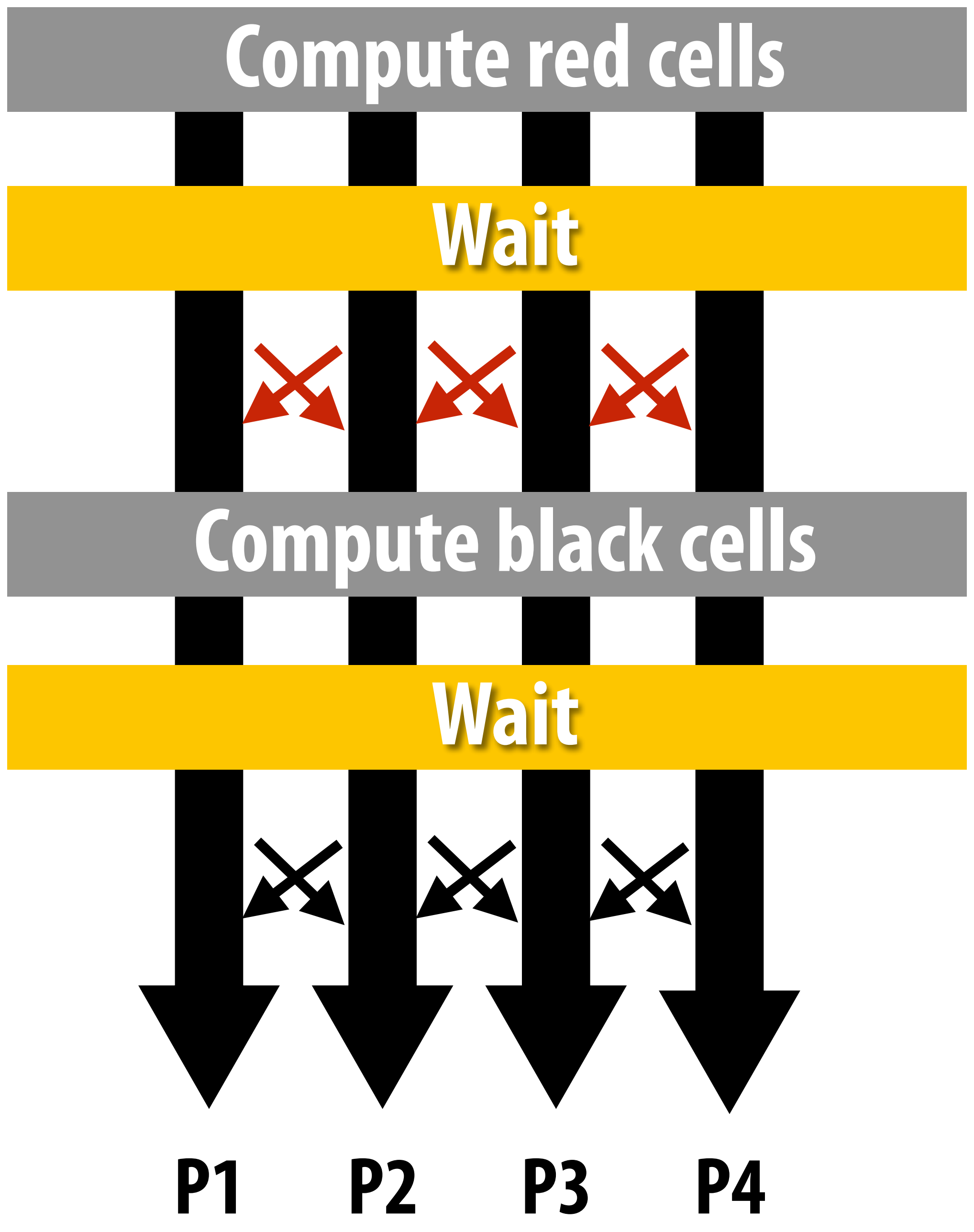


Question: Which is better? Does it matter?

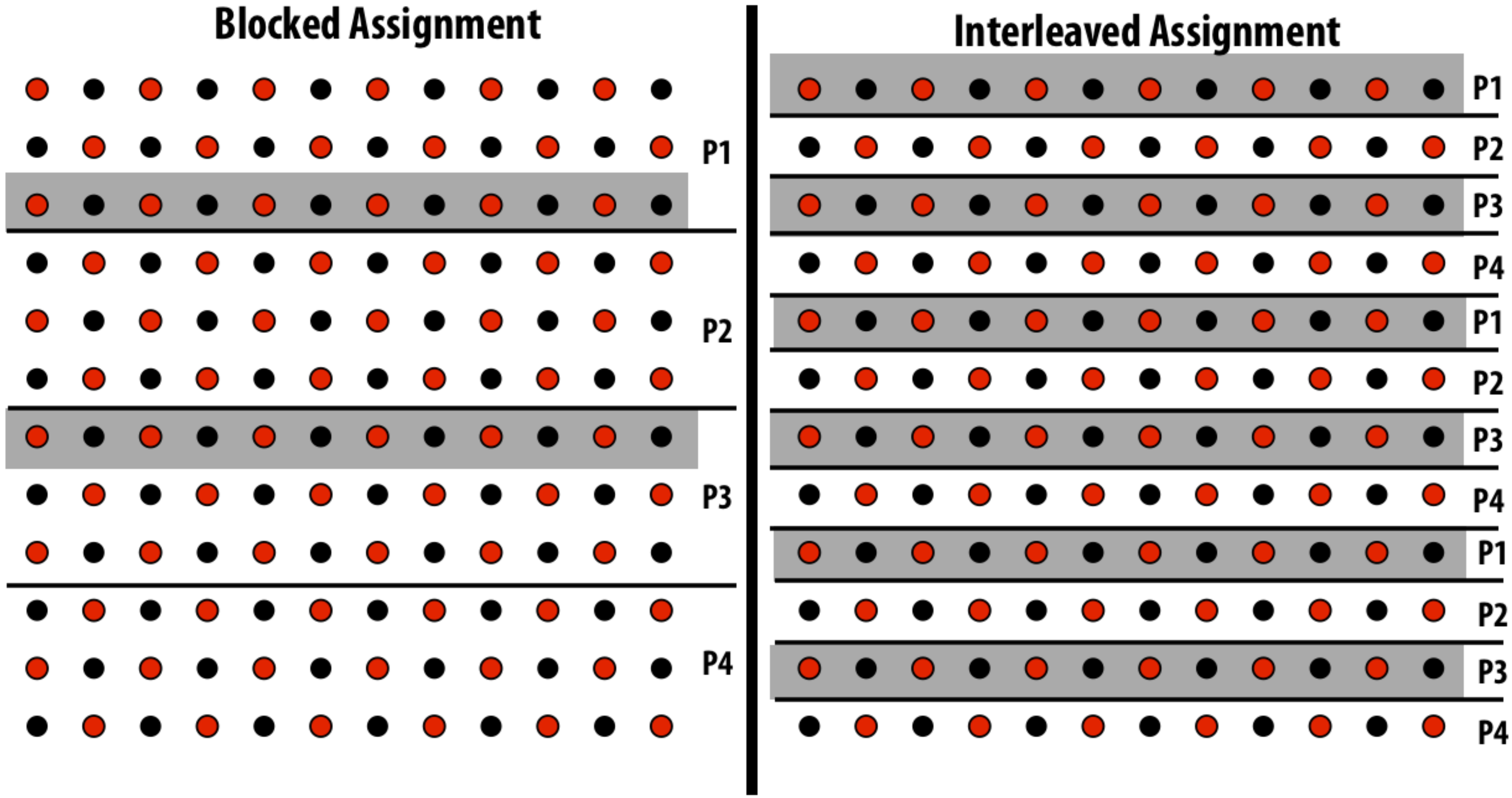
Answer: it depends on the system this program is running on

Consider dependencies (data flow)

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



Blocked assignment requires less data to be communicated between processors

Three ways to think about writing this program

- Data parallel
- SPMD / shared address space
- Message passing (will wait until a future class)

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: individual grid elements constitute independent work

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

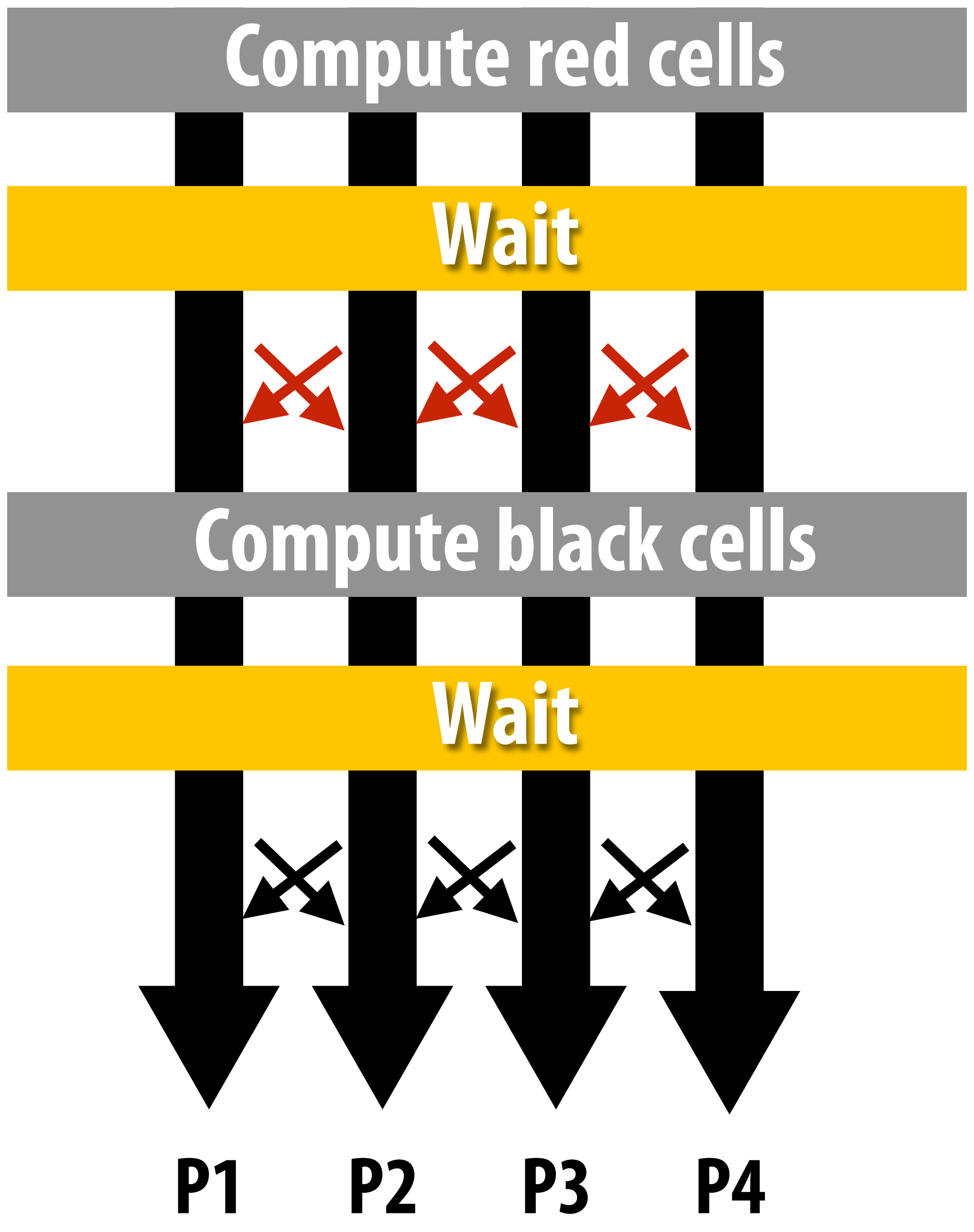
Orchestration: handled by system (End of for_all block is implicit wait for all workers before returning to sequential control)

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);
```

Assume these are global variables (accessible to all threads)

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

```
void solve(float* A) {

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

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

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

Grid solver example from: Culler, Singh, and Gupta

Shared address space solver (pseudocode in SPMD execution model)

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

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

void solve(float* A) {

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

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

Do you see a potential performance problem with this implementation?

Grid solver example from: Culler, Singh, and Gupta

Shared address space solver (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 reduction globally at the end of the iteration.

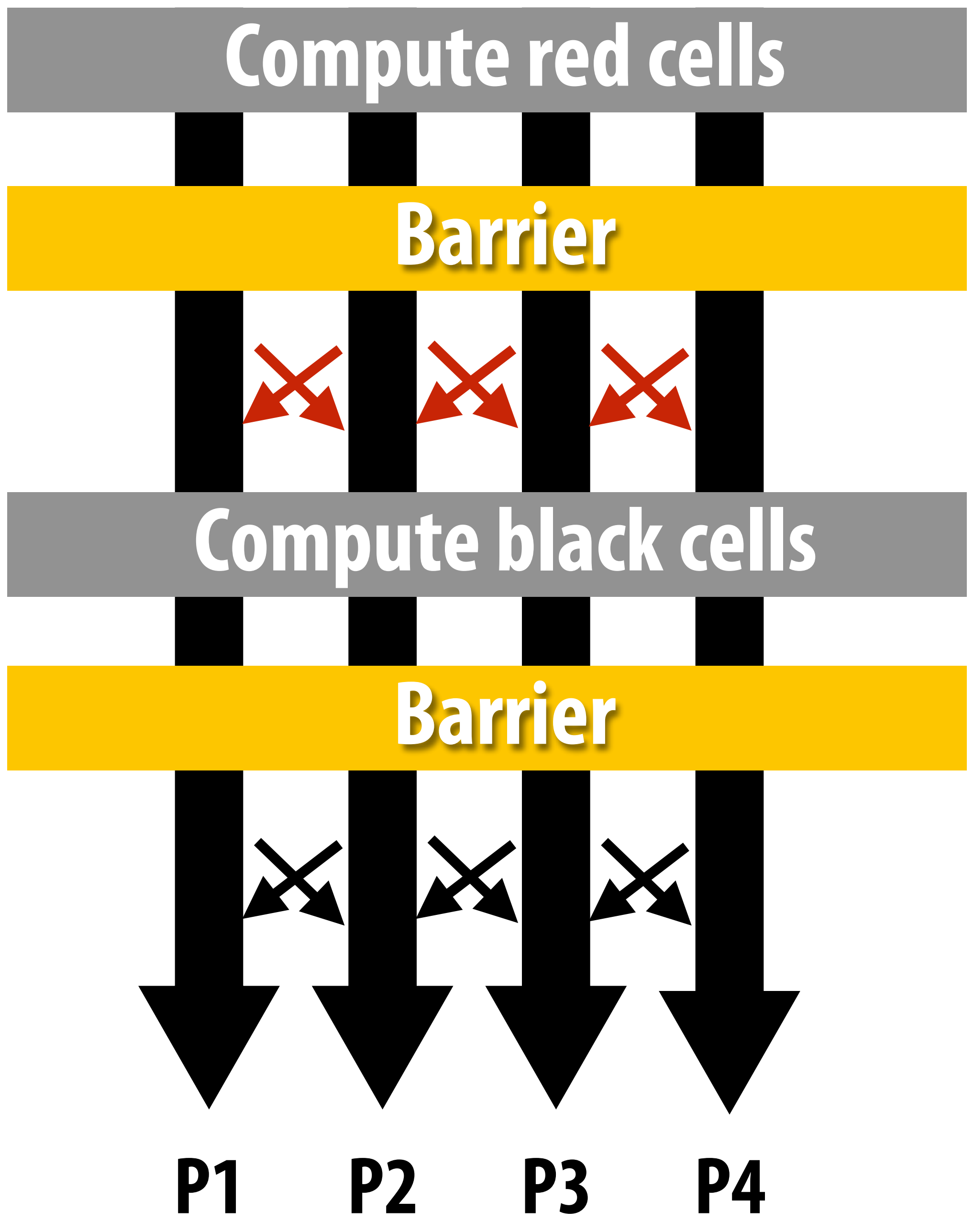
compute per worker partial sum (→ `myDiff += abs(A[i,j] - prev));`)

Now only only lock once per thread, not once per (i,j) loop iteration! (→ `lock(myLock); diff += myDiff; unlock(myLock);`)

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 (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);
   }
}
```

Why are there three barriers?

Grid solver example from: Culler, Singh, and Gupta

Shared address space solver: one barrier

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

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


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

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

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

Idea:

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

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

Grid solver example from: Culler, Singh, and Gupta

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

We will defer discussion of the message passing expression of solver to a later class.

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**