Lecture 4:

Parallel Programming Basics

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2015

Tunes

YACHT

Tripped and Fell in Love

(Shangri-La)

"I was so blown away by the experience of speeding my programs up by more than a factor of eight, I ran right to the keyboard and just had to compose a song."

- Claire Evans, on her tribute to writing her first parallel program.

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.

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**

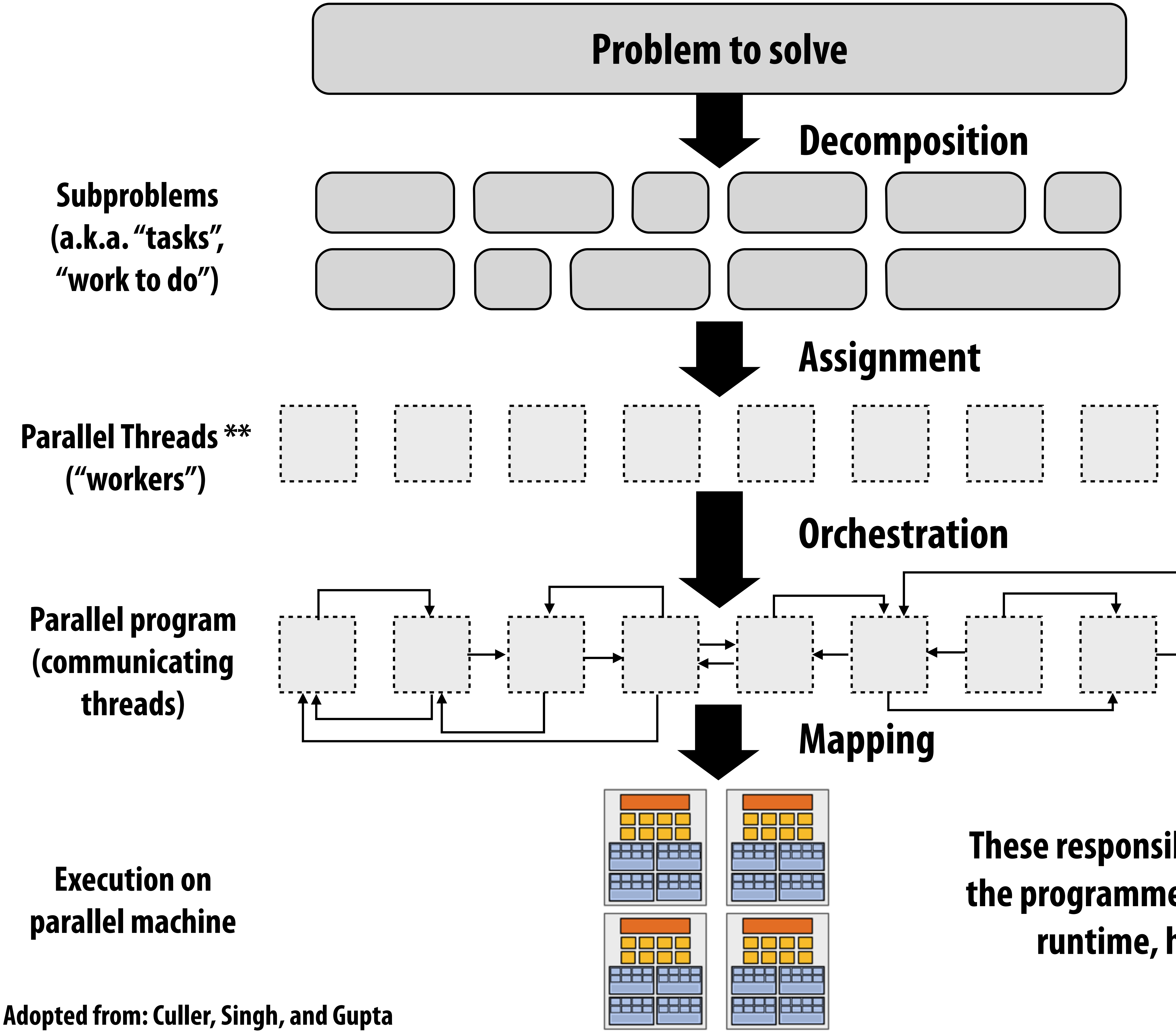
- **Recall one of our main goals is speedup ***

  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



** 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

Decomposition

- Break up problem into tasks that <u>can</u> be carried out in parallel
  - Decomposition need not happen statically
  - New tasks can be identified as program executes

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

Key aspect 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...

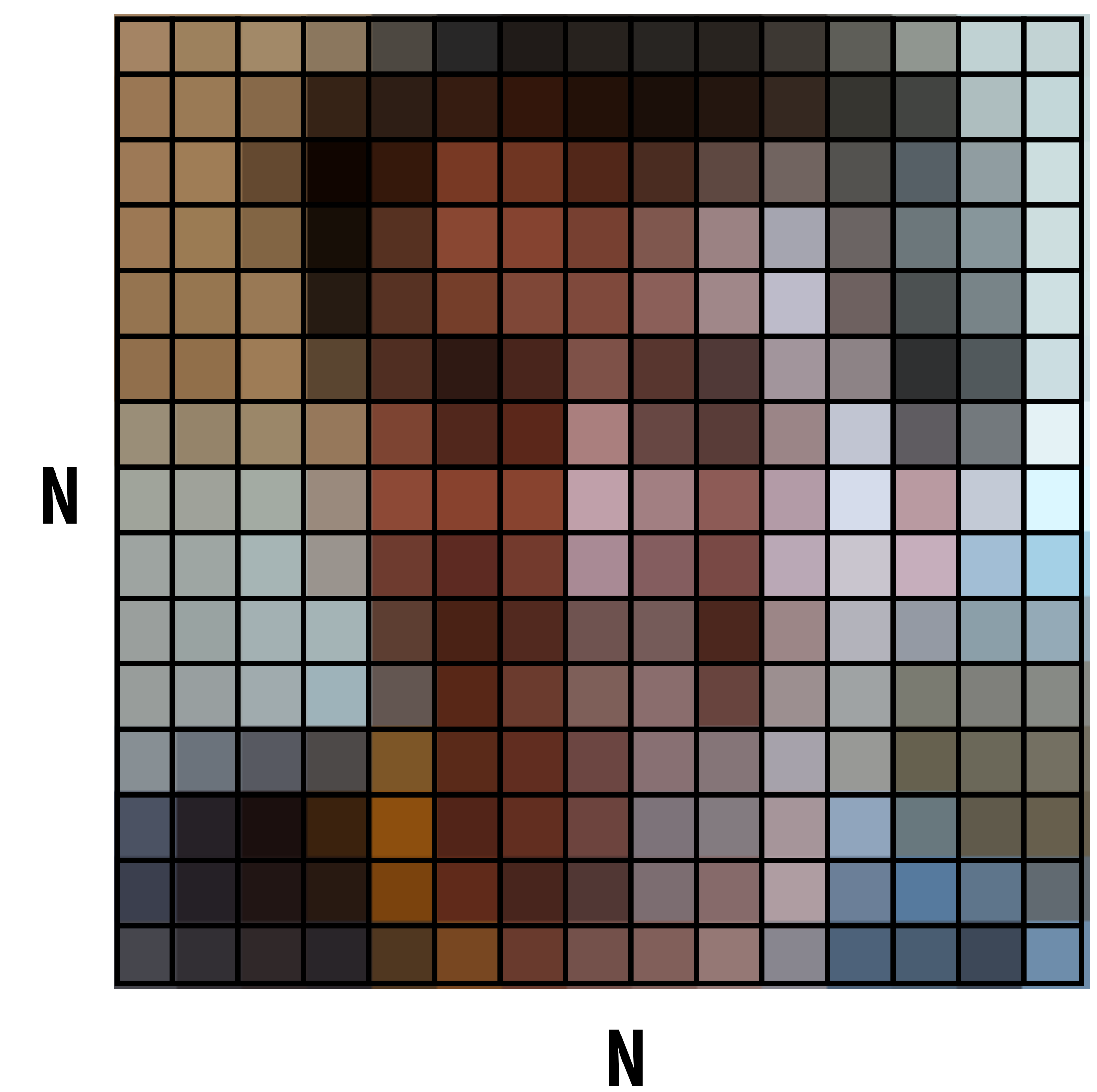
- Let $S =$ the fraction of sequential execution that is inherently sequential (dependencies prevent parallel execution)

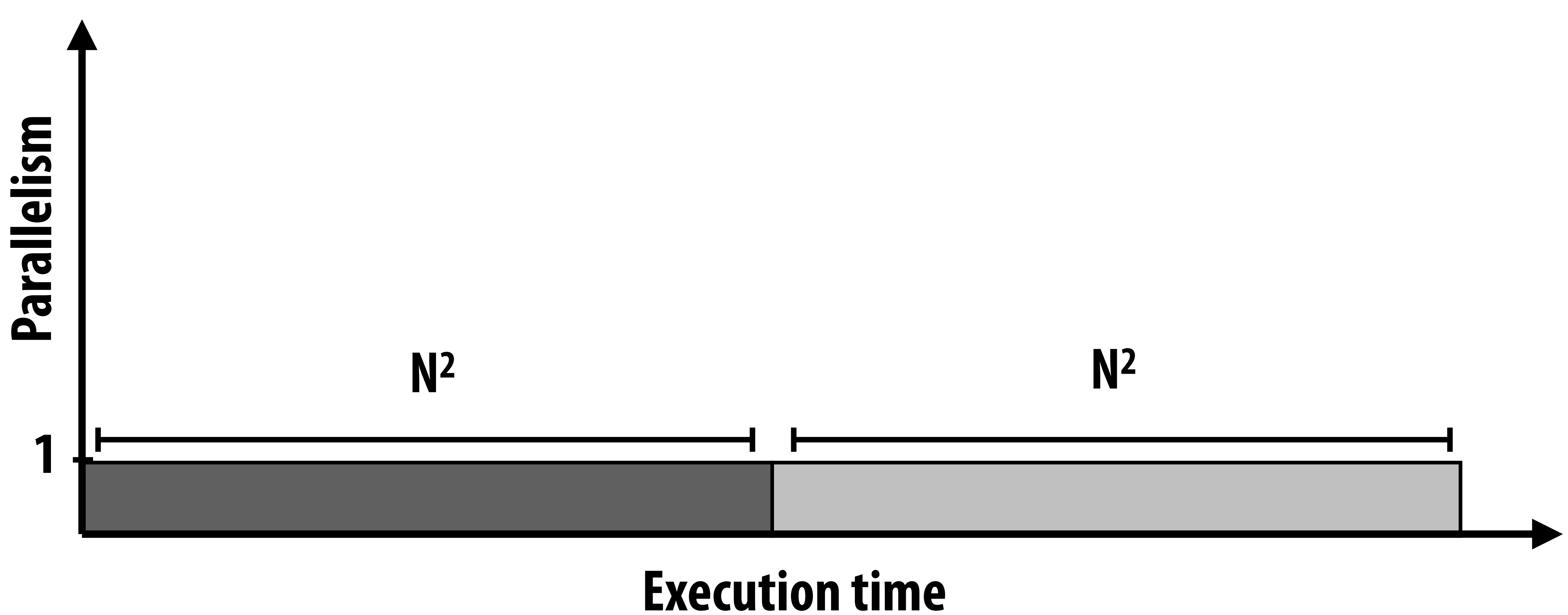
- Then maximum speedup due to parallel execution $\leq {}^{1}/_{S}$

A simple example

- Consider a two-step computation on an N-by-N image
  - Step 1: double brightness of all pixels
    (independent computation on each grid element)
  - Step 2: compute average of all pixel values

- Sequential implementation of program
  - Both steps take ~ N^2 time, so total time is ~ $2N^2$





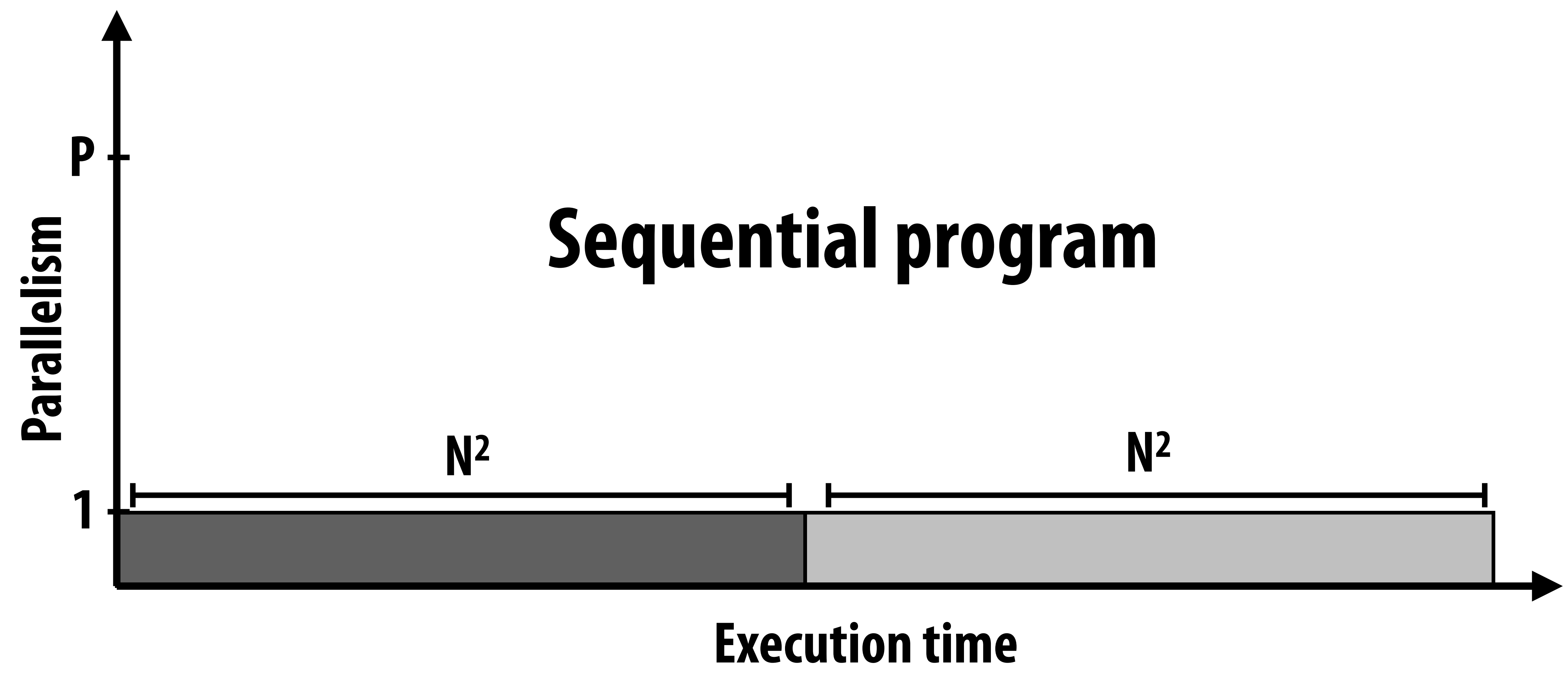
First attempt at parallelism (P processors)

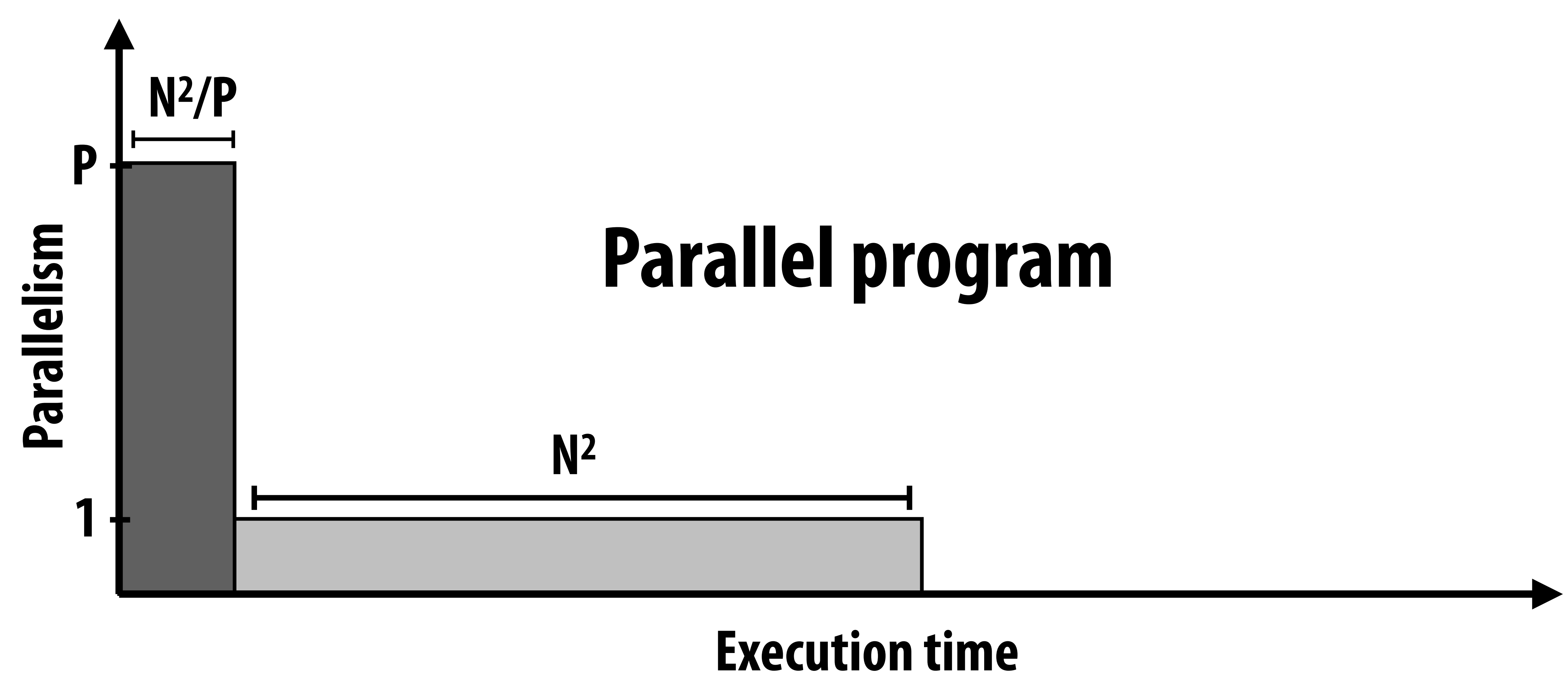
- **Strategy:**
  - Step 1: execute in parallel
    - time for phase 1: N^2/P
  - Step 2: execute serially
    - time for phase 2: N^2

- **Overall performance:**

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

Speedup ≤ 2



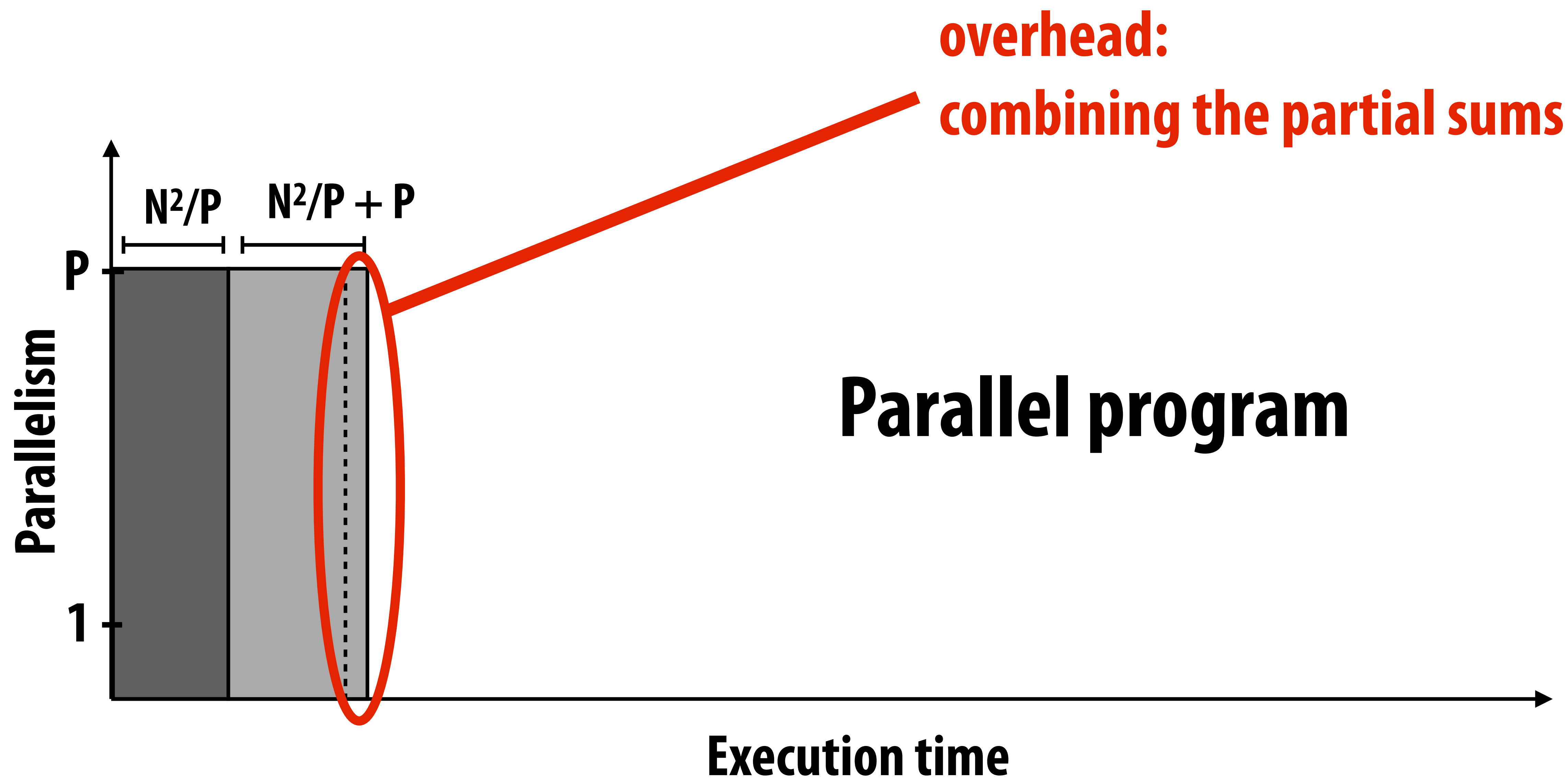


Parallelizing step 2

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

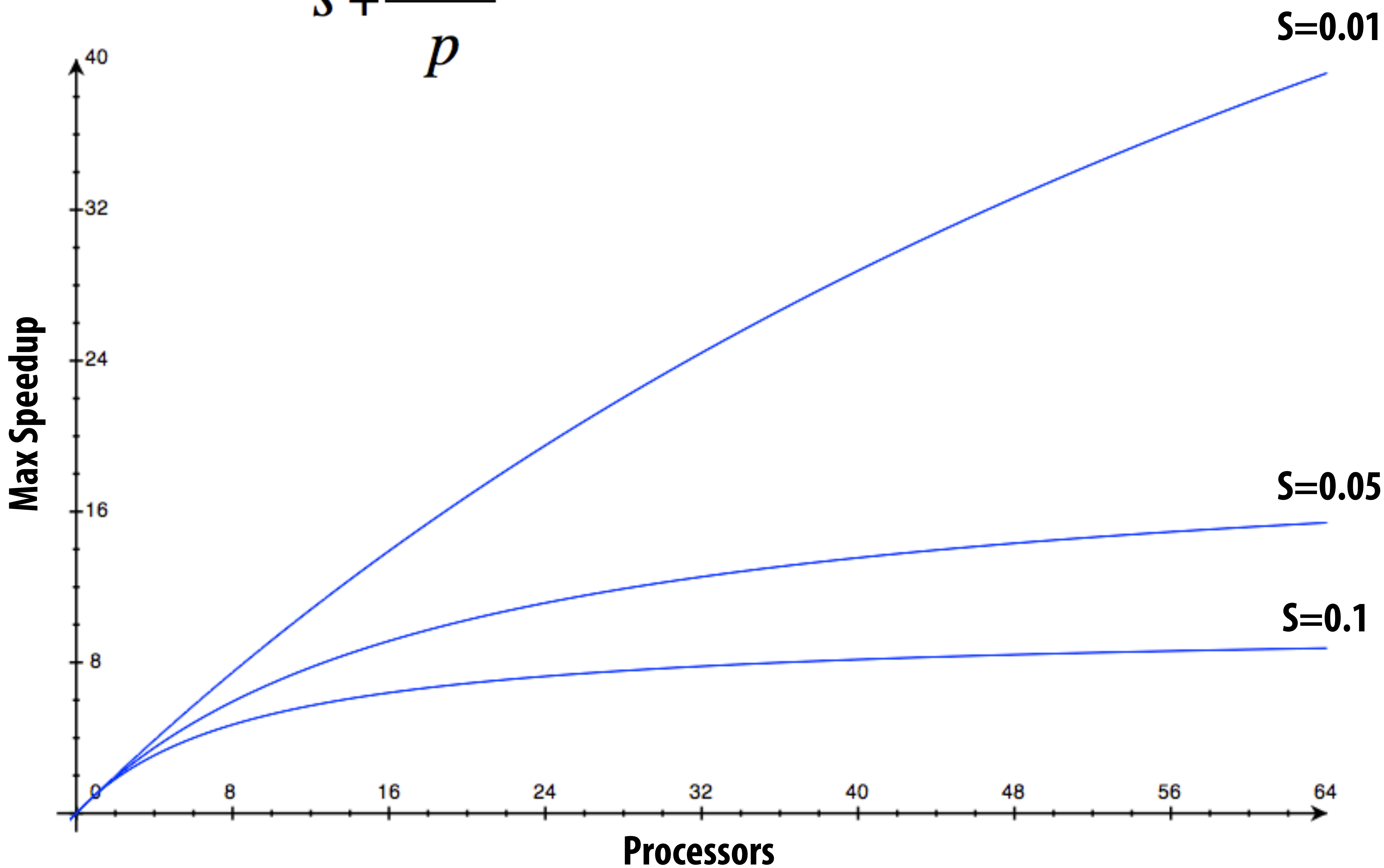
Note: speedup → P when N >> P



Amdahl's law

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

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

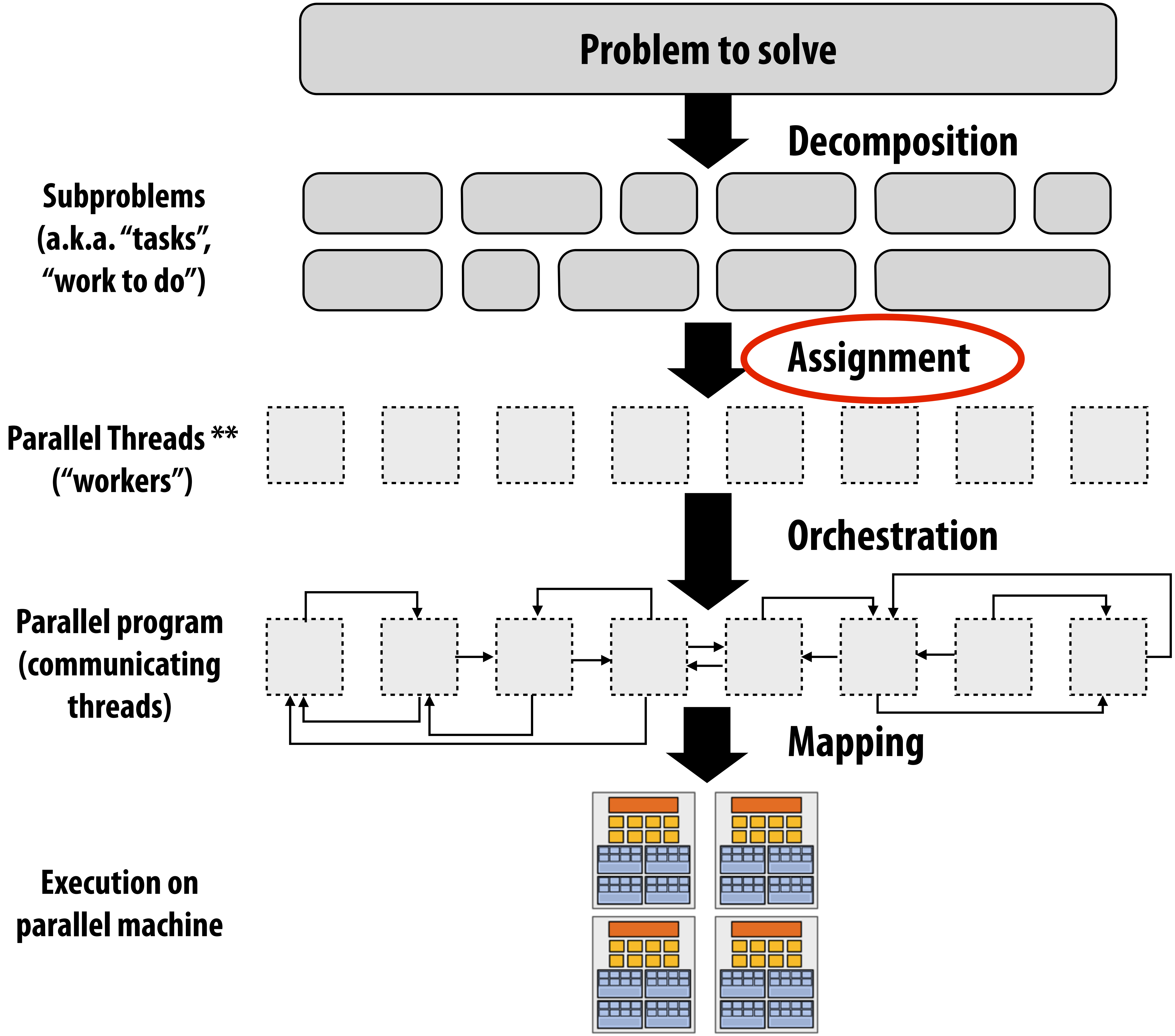


Decomposition

- **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

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

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

- Goals: balance workload, reduce communication costs

- Can be performed statically, or dynamically during execution

- While programmer 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:

Static 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)

Static assignment example using pthreads

```
typedef struct {
   int N, terms;
   float* x, *result;
} my_args;

void parallel_sinx(int N, int terms, float* x, float* result)
{
    pthread_t thread_id;
    my_args args;

    args.N = N/2;
    args.terms = terms;
    args.x = x;
    args.result = result;

    // launch second thread, do work on first half of array
    pthread_create(&thread_id, NULL, my_thread_start, &args);

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

    pthread_join(thread_id, NULL);
}

void my_thread_start(void* thread_arg)
{
   my_args* thread_args = (my_args*)thread_arg;
   sinx(args->N, args->terms, args->x, args->result); // do work
}
```

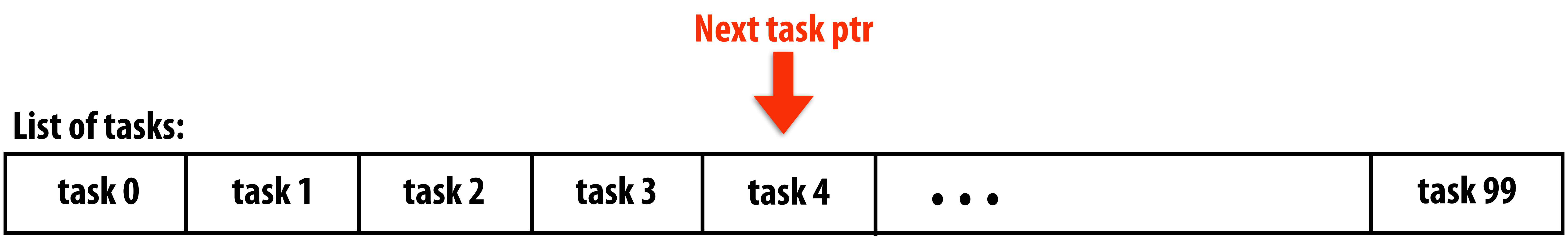
Decomposition of work by loop iteration

Programmer-managed assignment:
- **Static assignment**
- **Assign iterations to pthreads in blocked fashion (first half of array to spawned thread, second half to main thread)**

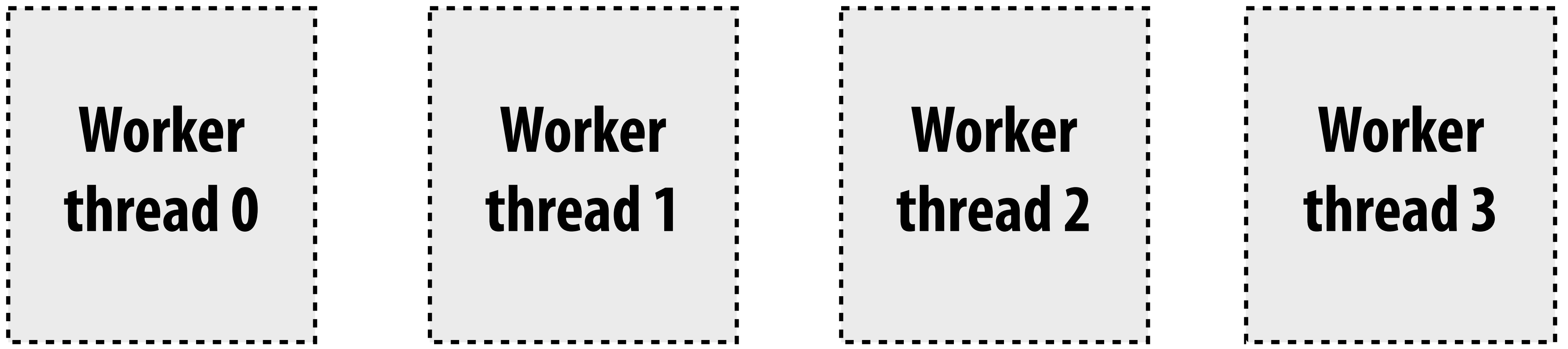
Dynamic assignment using ISPC tasks

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

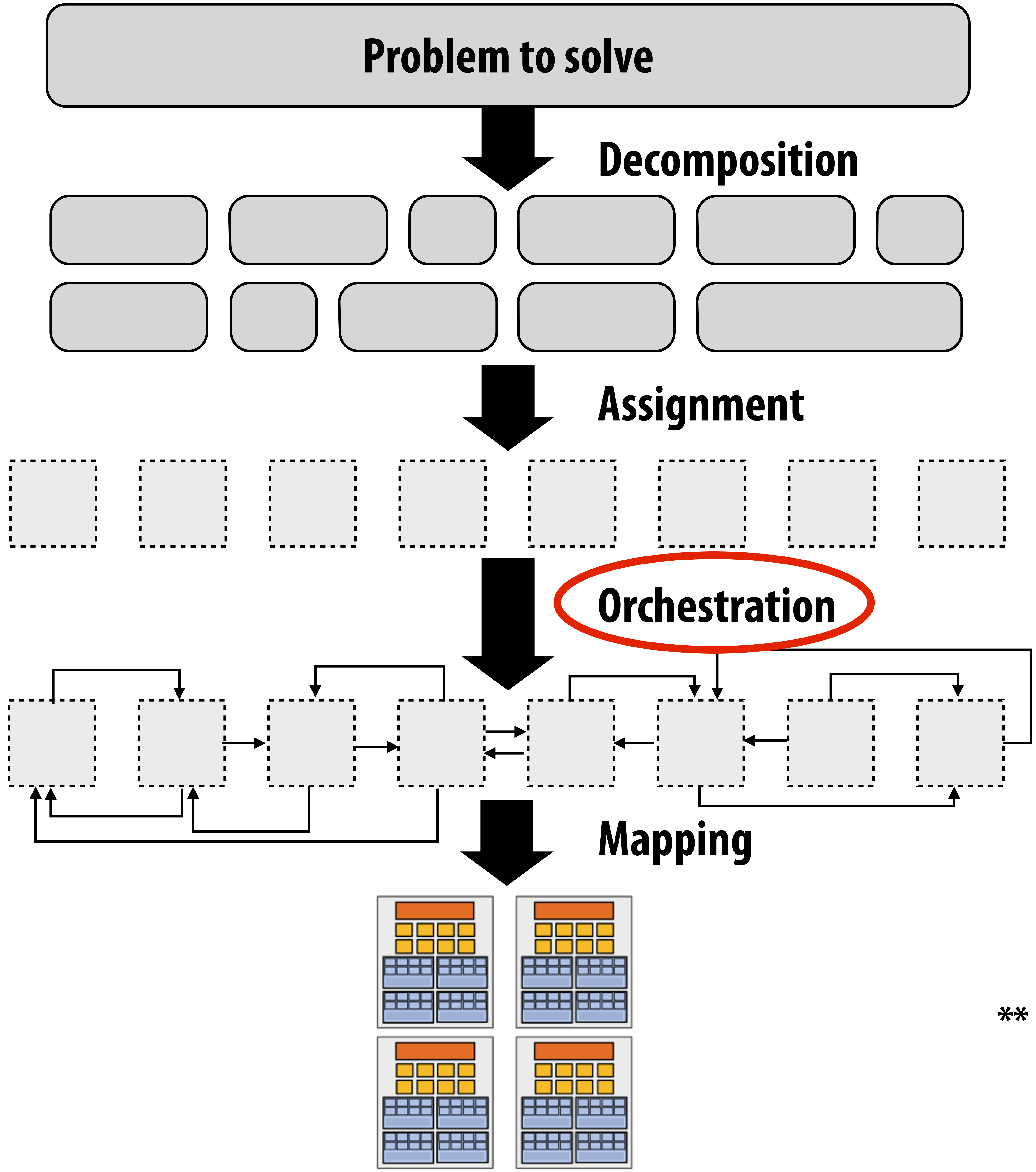
Assign tasks to worker threads.



Assignment policy: after completing current task, worker thread inspects list and assigns itself the next uncompleted task.



Orchestration



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

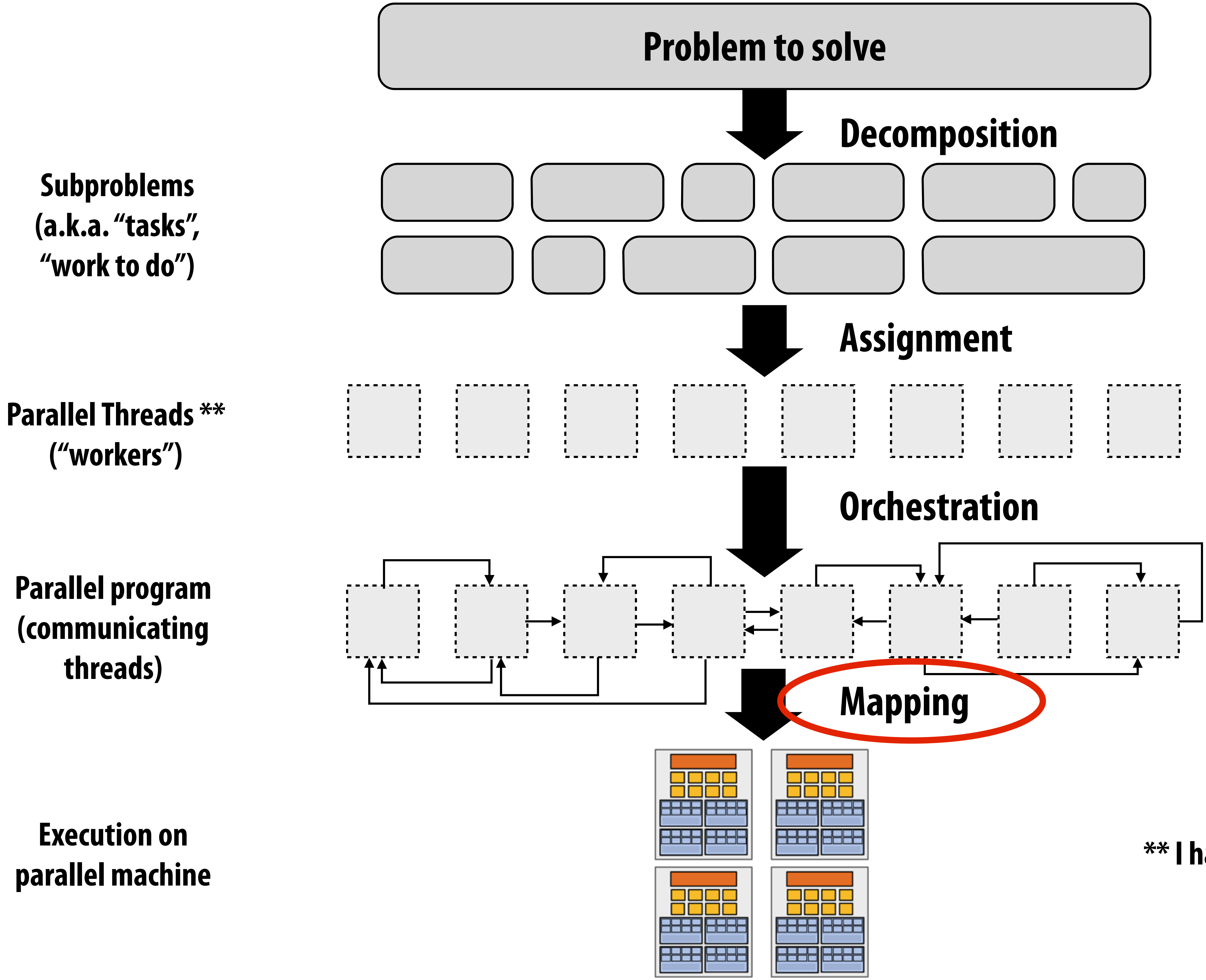
Orchestration

- **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



** I had to pick a term

Mapping

- **Mapping "threads" ("workers") to hardware execution units**

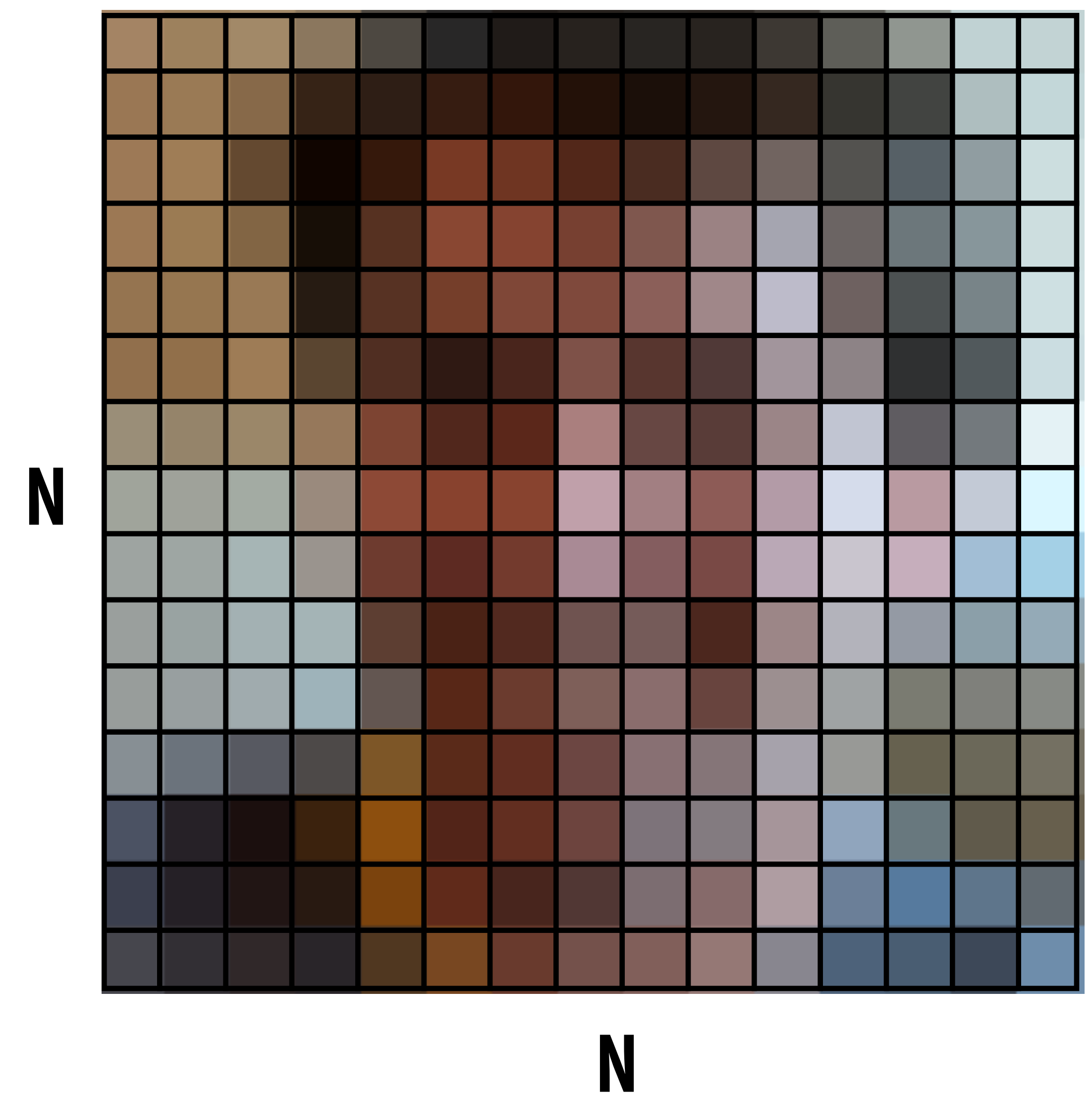
- **Example 1: mapping by the operating system**
  - e.g., map kernel thread to CPU core execution context

- **Example 2: mapping by the compiler:**
  - Mapping ISPC program instances to vector instruction lanes

- **Example 3: mapping by the hardware:**
  - mapping 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

Decomposing computation or data?



Often, the reason a problem requires lots of computation (and needs to be parallelized) is that it involves manipulating a lot of data.

I've described the process of parallelizing programs as an act of <u>partitioning computation</u>.

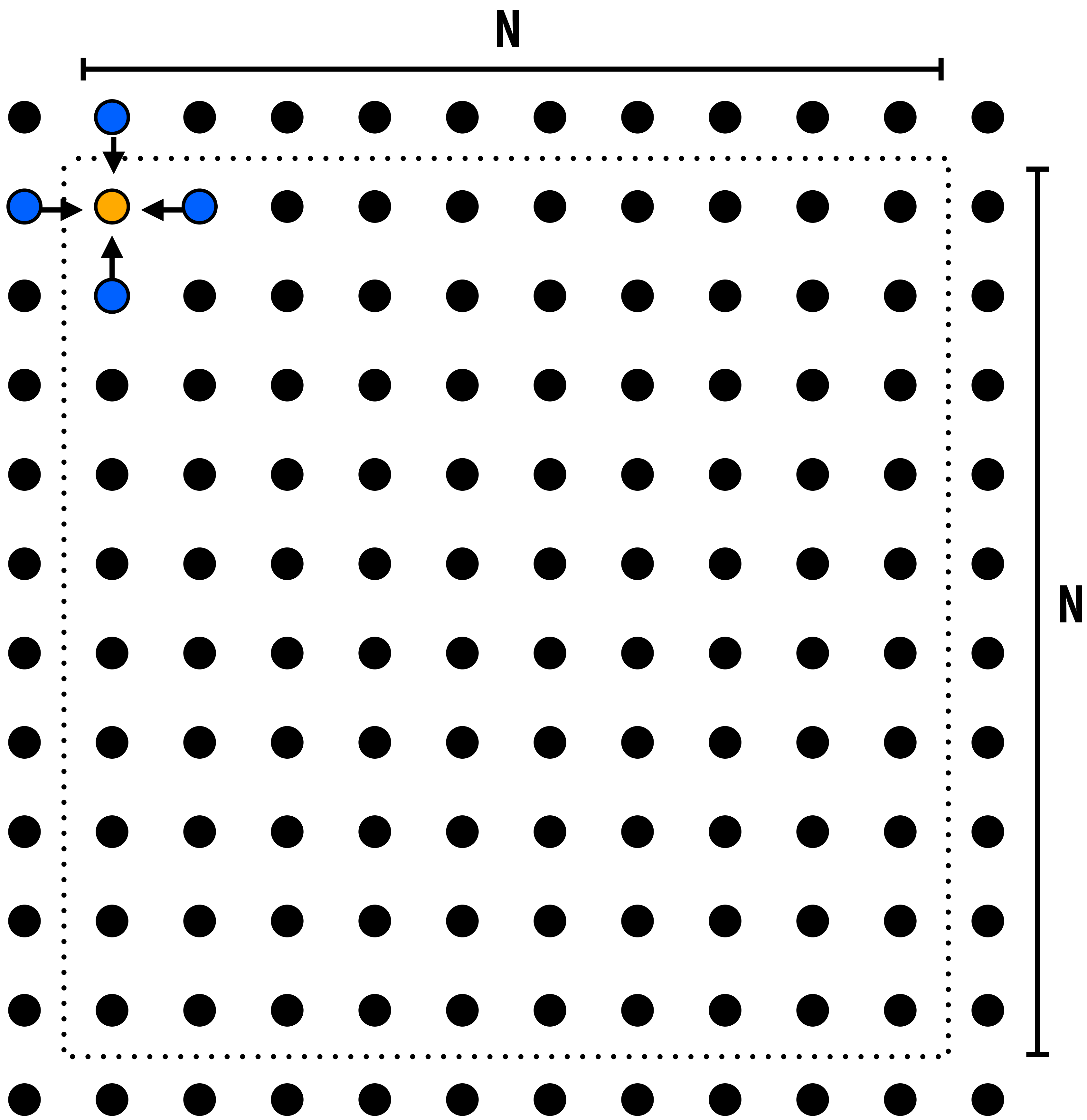
Often, it's equally valid to think of <u>partitioning data</u>. (computations go with the data)

But there are many computations where the correspondence between work-to-do ("tasks") and data is less clear. In these cases it's natural to think of partitioning computation.

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]);
```

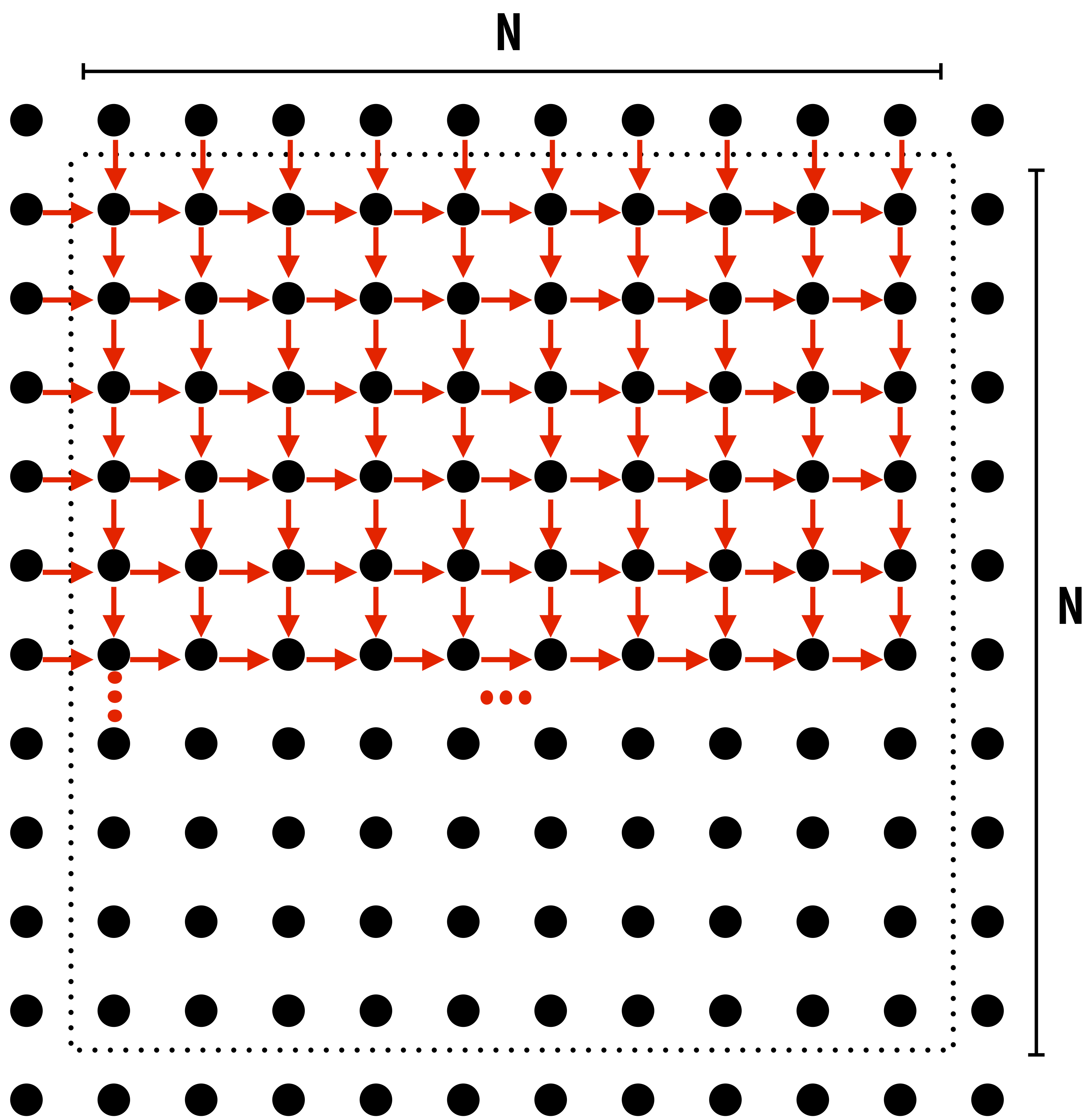
Grid solver example from: Culler, Singh, and Gupta

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;
  }
}
```

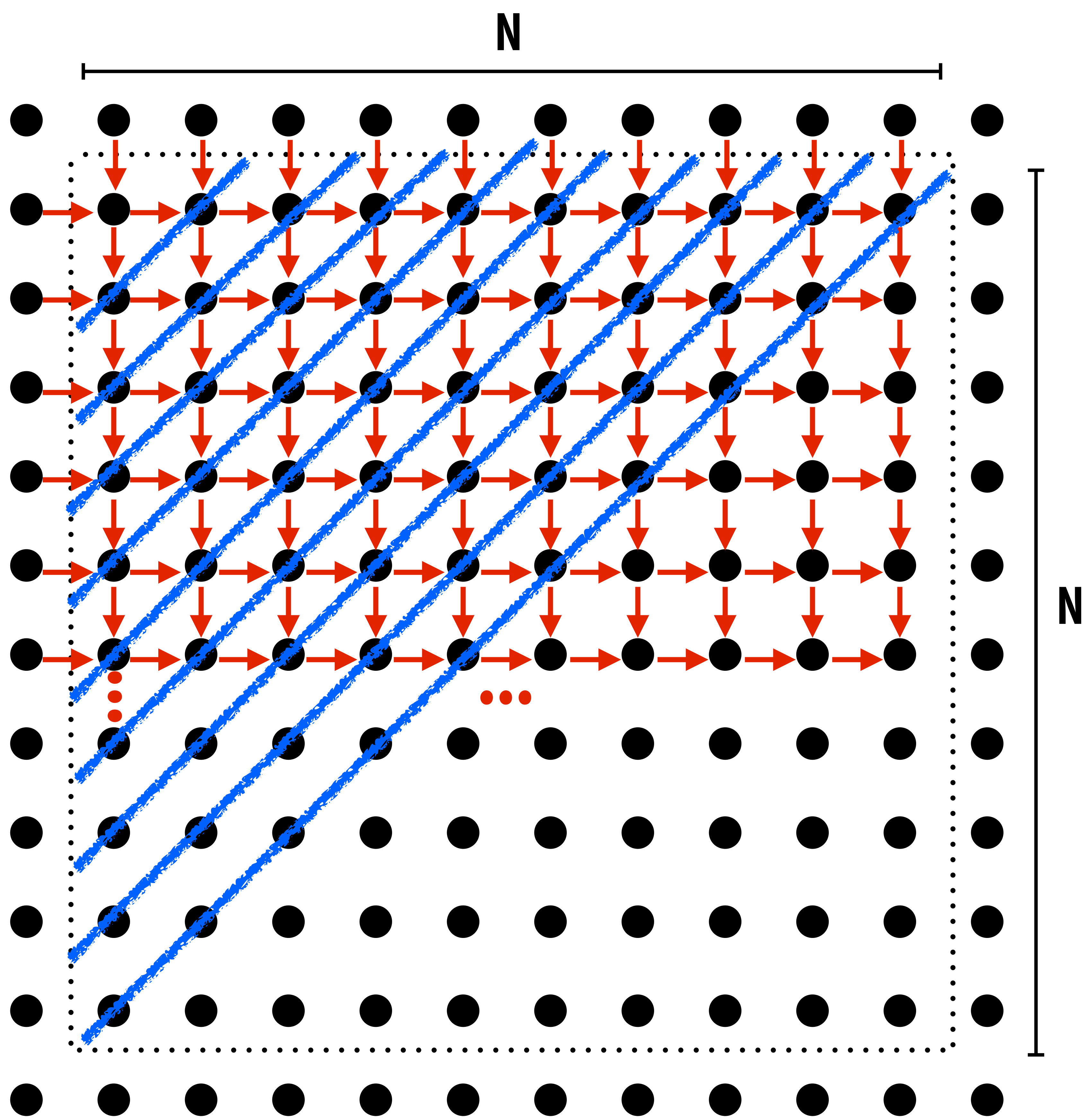
Step 1: identify dependencies (problem decomposition phase)



Each row element depends on element to left.

Each column depends on previous column.

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: hard to exploit

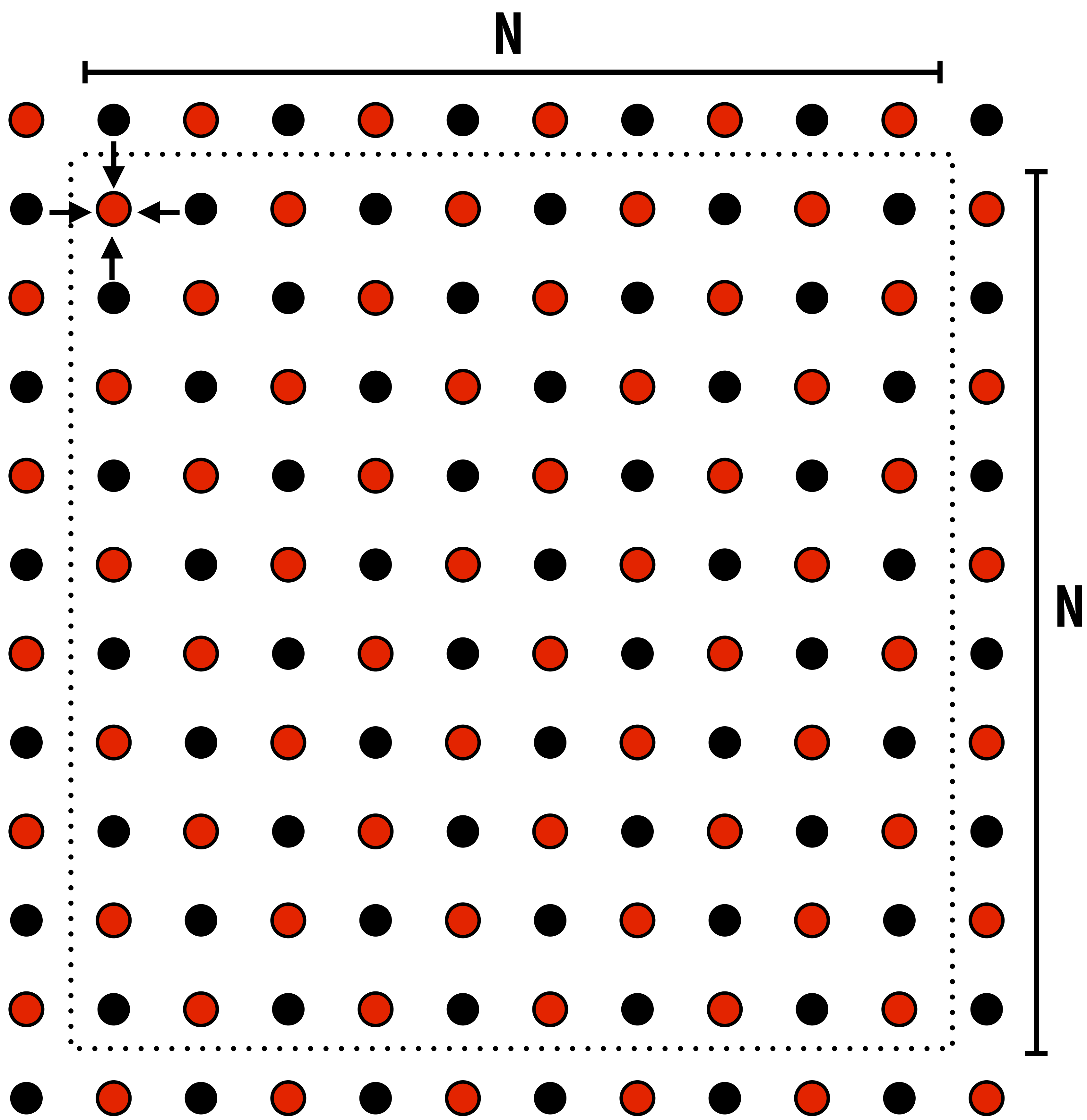
Not much parallelism at beginning and end of computation.

Frequent synchronization (after completing each diagonal)

Let's make life easier on ourselves

- Idea: improve performance by changing 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
  - Needed domain knowledge of Gauss-Seidel iteration to realize this change is permissible for the application

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

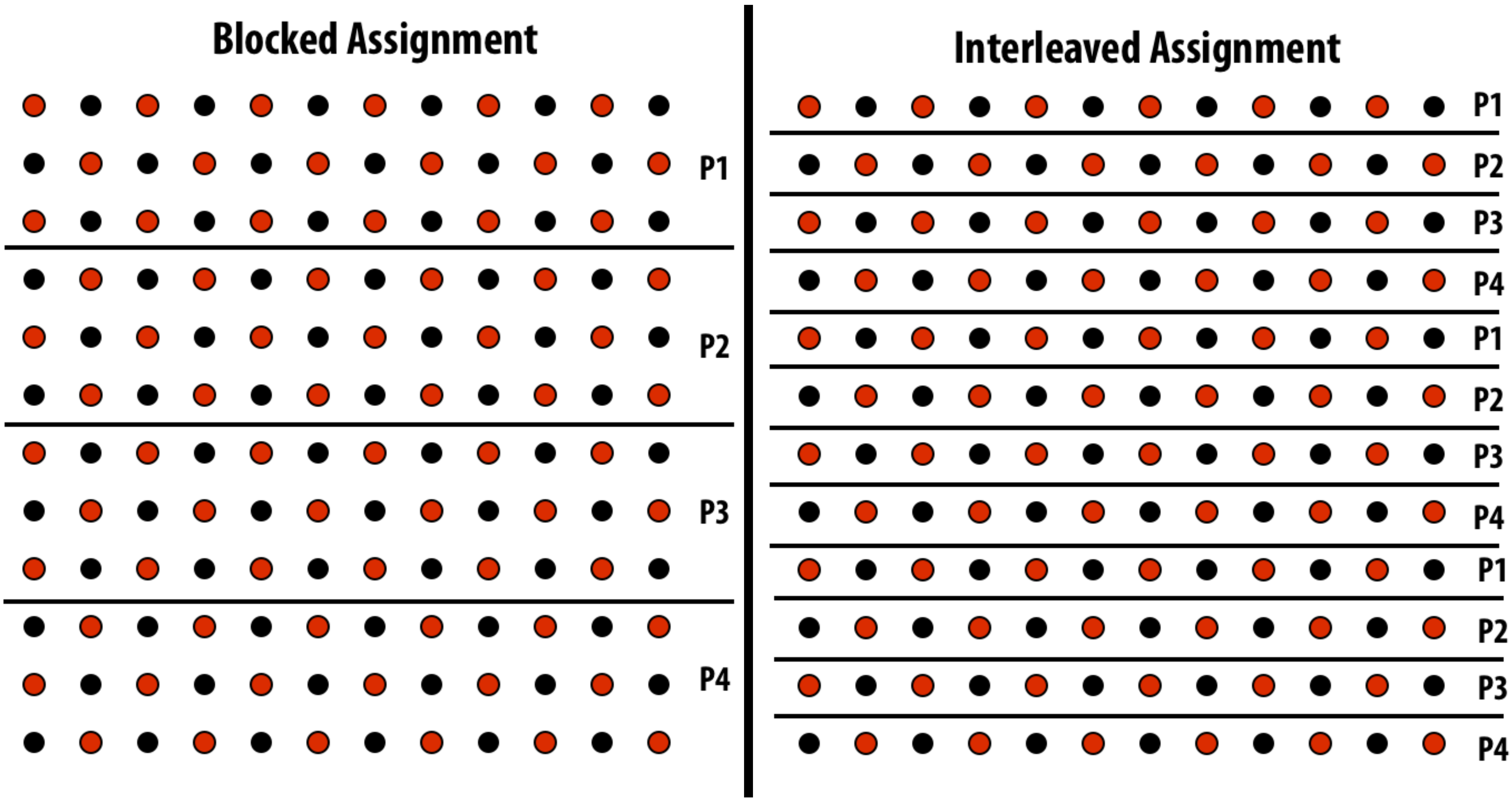


Update all red cells in parallel

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

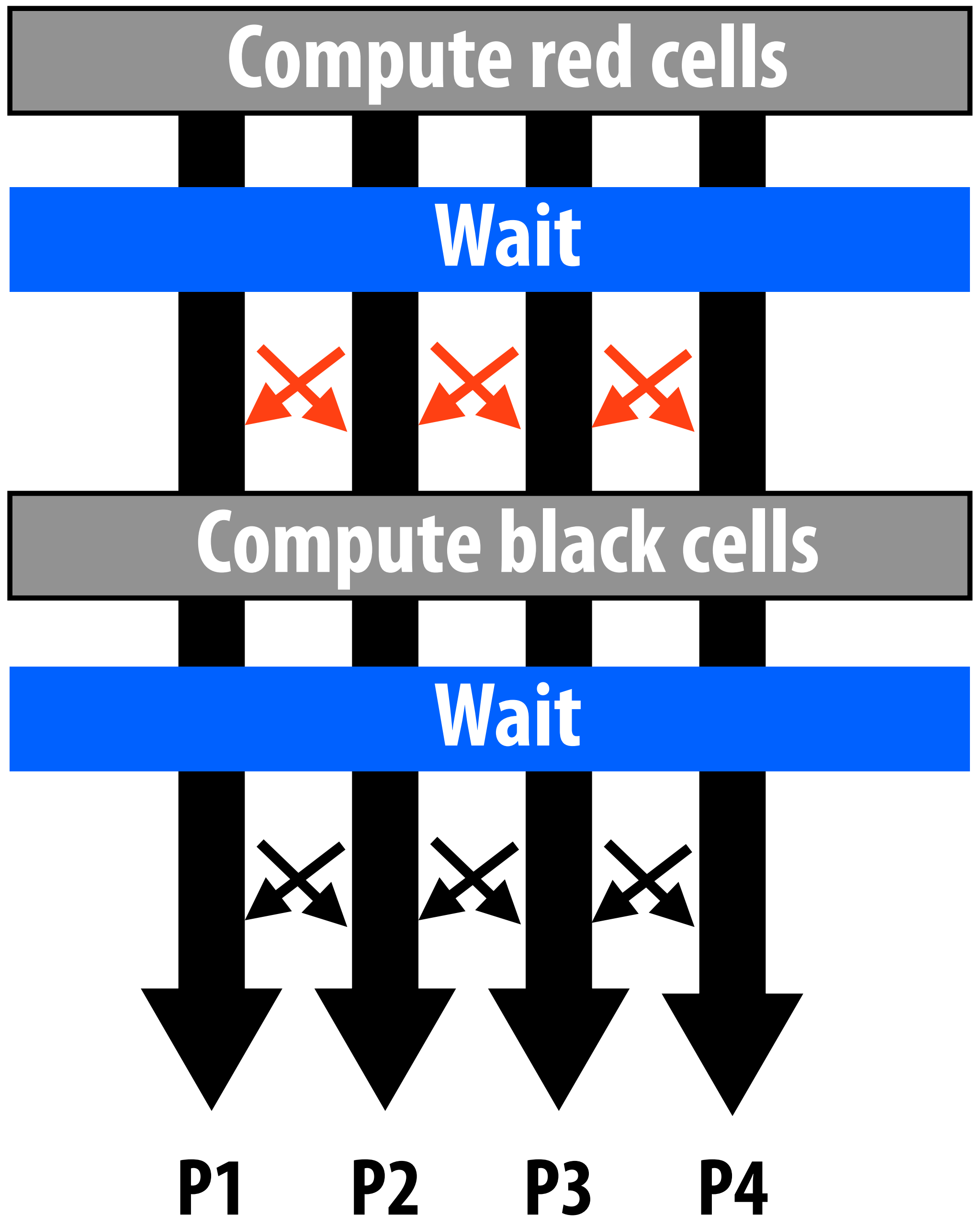


Question: Which is better? Does it matter?

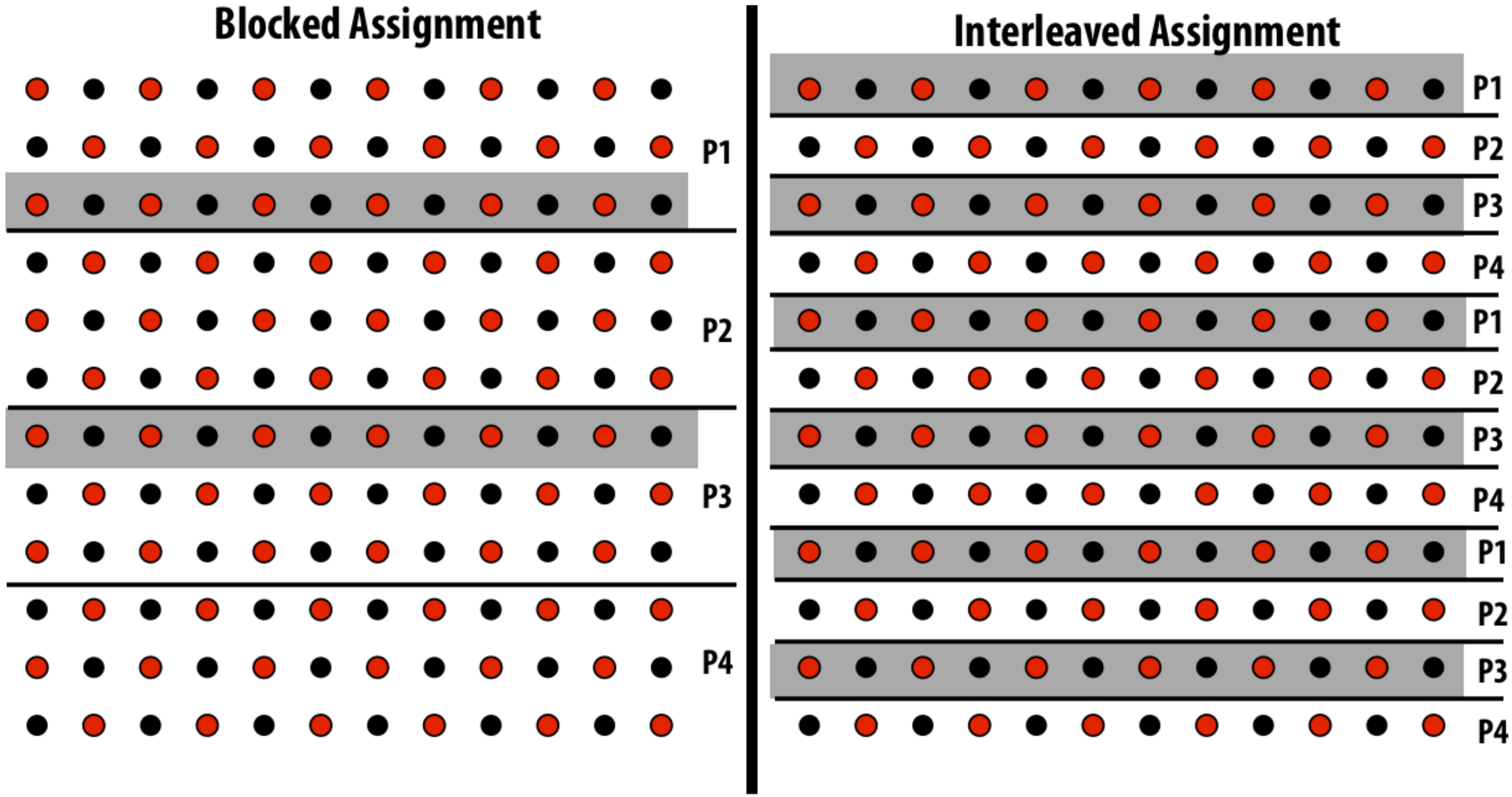
Answer: it depends on the system this program is running on

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




= data that must be sent to P2 each iteration

Blocked assignment requires less data to be communicated between processors

Data-parallel expression of solver

Data-parallel expression of grid solver

Note: to simplify pseudocode: just showing red-cell update

```
const int n;

// allocate grid, use block decomposition across processors
float* A = allocate(n+2, n+2, BLOCK_Y, NUM_PROCESSORS);

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:
specified explicitly (blocks of consecutive rows for same processor)

decomposition:
independent tasks are individual elements

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

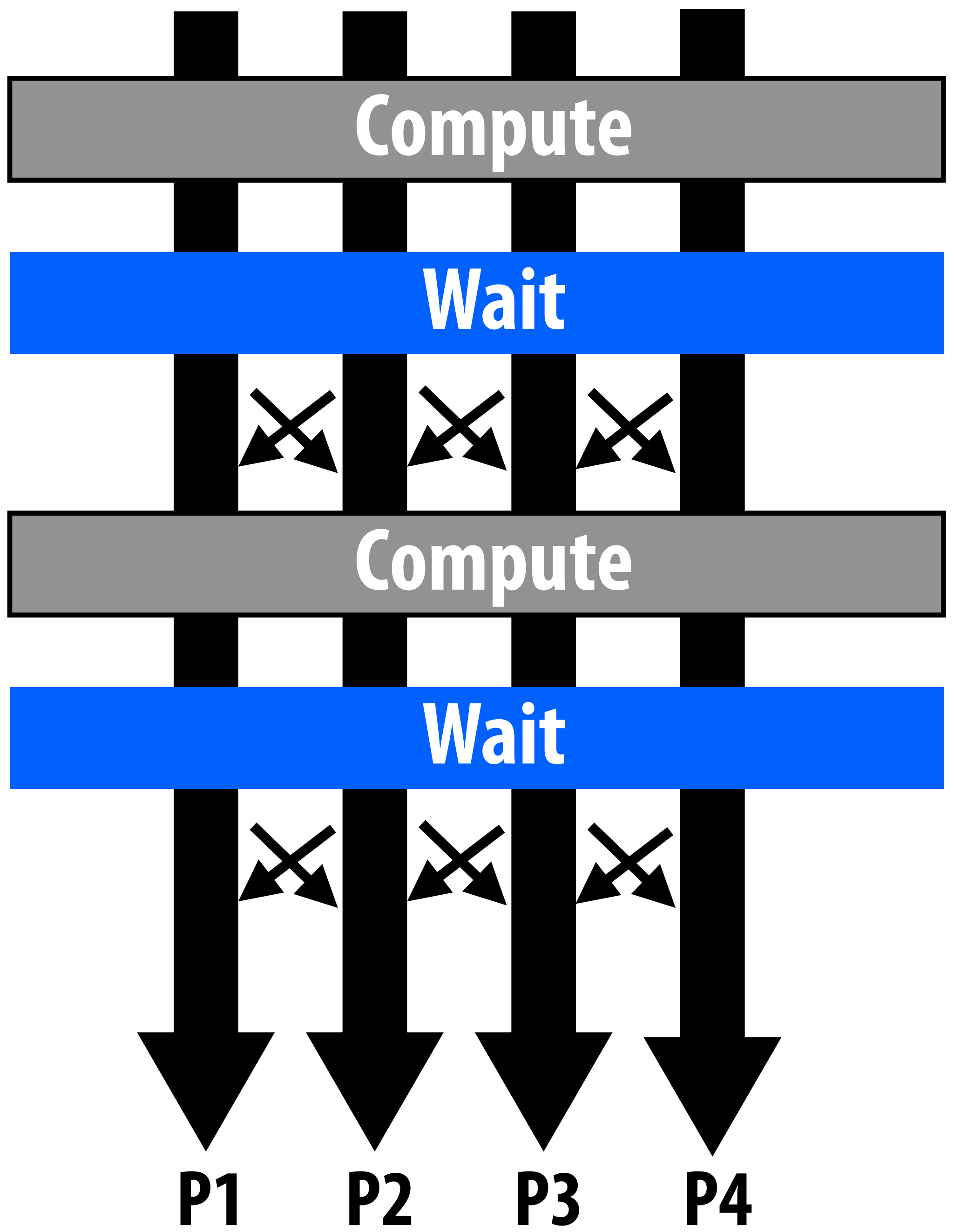
Orchestration:
handled by system
(builtin communication primitive: reduceAdd)

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) {
     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);
   }
}
```

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

Grid solver example from: Culler, Singh, and Gupta

Review: need for mutual exclusion

- Each thread executes
  - Load the value of diff into register r1
  - Add the register r2 to register r1
  - Store the value of register r1 into diff
- One possible interleaving: (let starting value of diff=0, r2=1)

T0	T1	
`r1 ← diff`		`T0 reads value 0`
	`r1 ← diff`	`T1 reads value 0`
`r1 ← r1 + r2`		`T0 sets value of its r1 to 1`
	`r1 ← r1 + r2`	`T1 sets value of its r1 to 1`
`diff ← r1`		`T0 stores 1 to diff`
	`diff ← r1`	`T1 stores 1 to diff`

- Need this set of three instructions to be atomic

Mechanisms for atomicity

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

```
LOCK(mylock);
// critical section
UNLOCK(mylock);
```

- **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);
```

- **Access to critical section will be serialized across all threads**
  - **High contention will cause performance problems (recall Amdahl's Law)**
  - **Note partial accumulation into private `myDiff` reduces contention**

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) {
     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 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);
   }
}
```

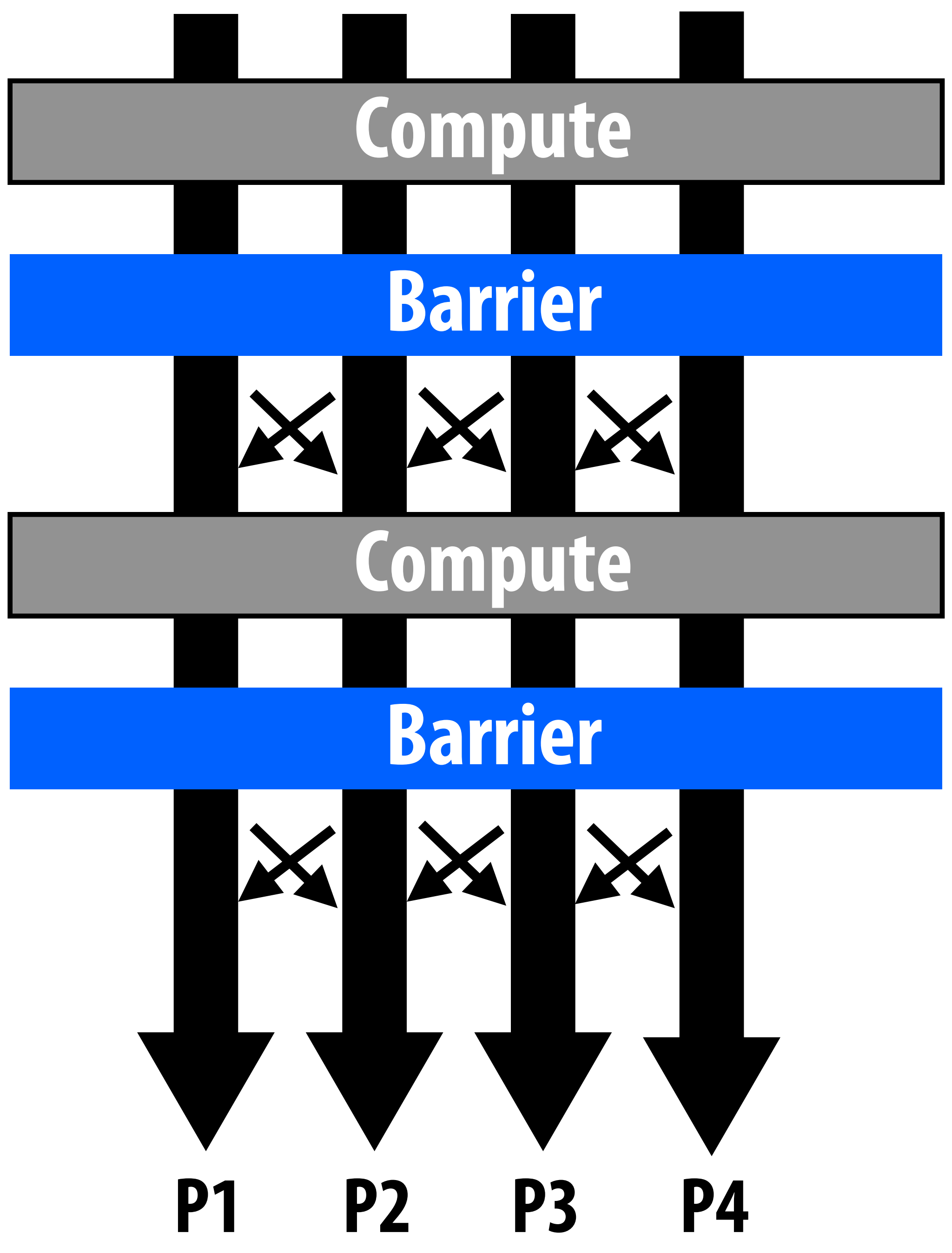
Improve performance by accumulating locally first, then globally only at the end of the iteration.

compute per worker partial sum

Now only only lock once per thread, not once per (i,j) loop iteration!

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



Shared address space solver (SPMD execution model)

Why are there three barriers?

```
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);
   }
}
```

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)

More on specifying dependencies

- **Barriers: simple, but conservative (coarse granularity dependencies)**
  - **All work in program up until this point (for all threads) must finish before any thread begins next phase**

- **Specifying specific dependencies can increase performance (by revealing more parallelism)**
  - **Example: two threads. One produces a result, the other consumes it.**

T0

```
// produce x, then let T1 know
x = 1;
flag = 1;
```

T1

```
// do stuff independent
// of x here

while (flag == 0);
print x;
```

- **We just implemented a message queue (of length 1)**

Solver implementation in two programming models

- **Data-parallel programming model**
  - Synchronization:
    - Single logical thread of control, but iterations of `forall` loop can be parallelized (implicit barrier at end of outer `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
    - 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, assignment, orchestration, mapping
  - 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**