Lecture 4:

Parallel Programming Basics

Parallel Computer Architecture and Programming
CMU 15-418, Spring 2013

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?

Quiz

```
export void sinx(
   uniform int N,
   uniform int terms,
   uniform float* x,
   uniform float* result)
{

   foreach (idx = 0 .. N)
   {

      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.

Which iterations of the loop are parallelized by ISPC? Which are not?

Creating a parallel program

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

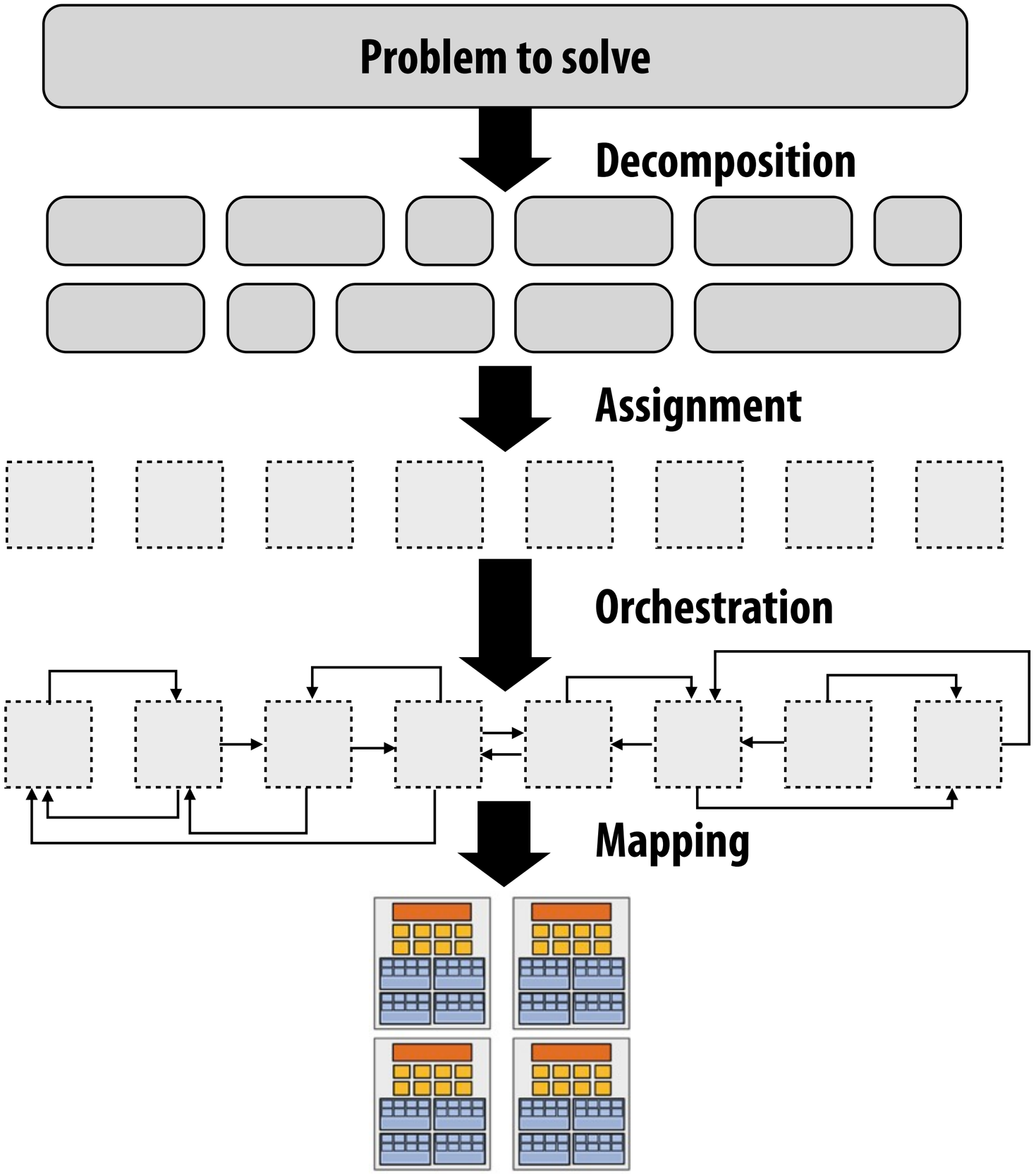
- Recall one of our main goals is speedup *

  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

Steps in creating a parallel program



These steps are performed by the programmer, by the system (compiler, runtime, hardware), or by both!

** I had to pick a term

Decomposition

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

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

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

Limited concurrency: Amdahl's law

- You run your favorite sequential program...

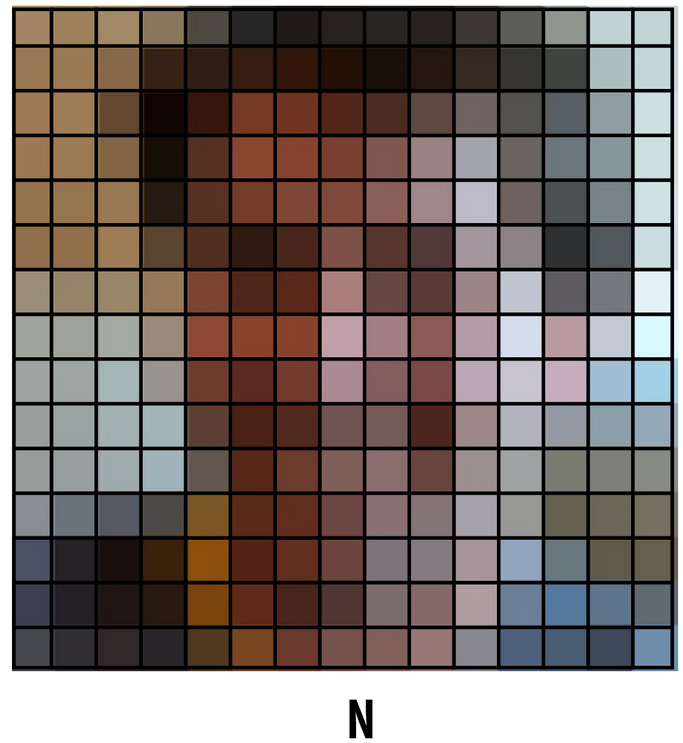
- Let S = the fraction of sequential execution that is inherently sequential (dependencies prevent parallel execution)

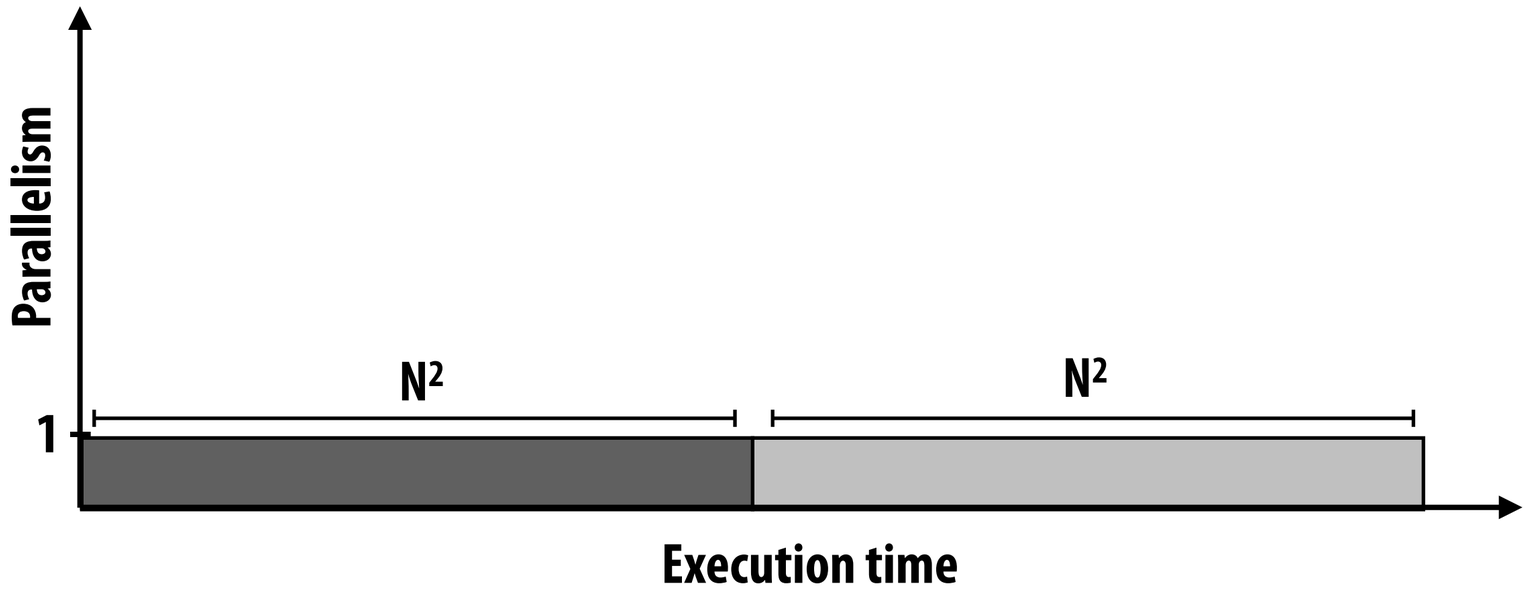
- Then maximum speedup due to parallel execution $\leq {}^{1}/_{S}$

Amdahl's law 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

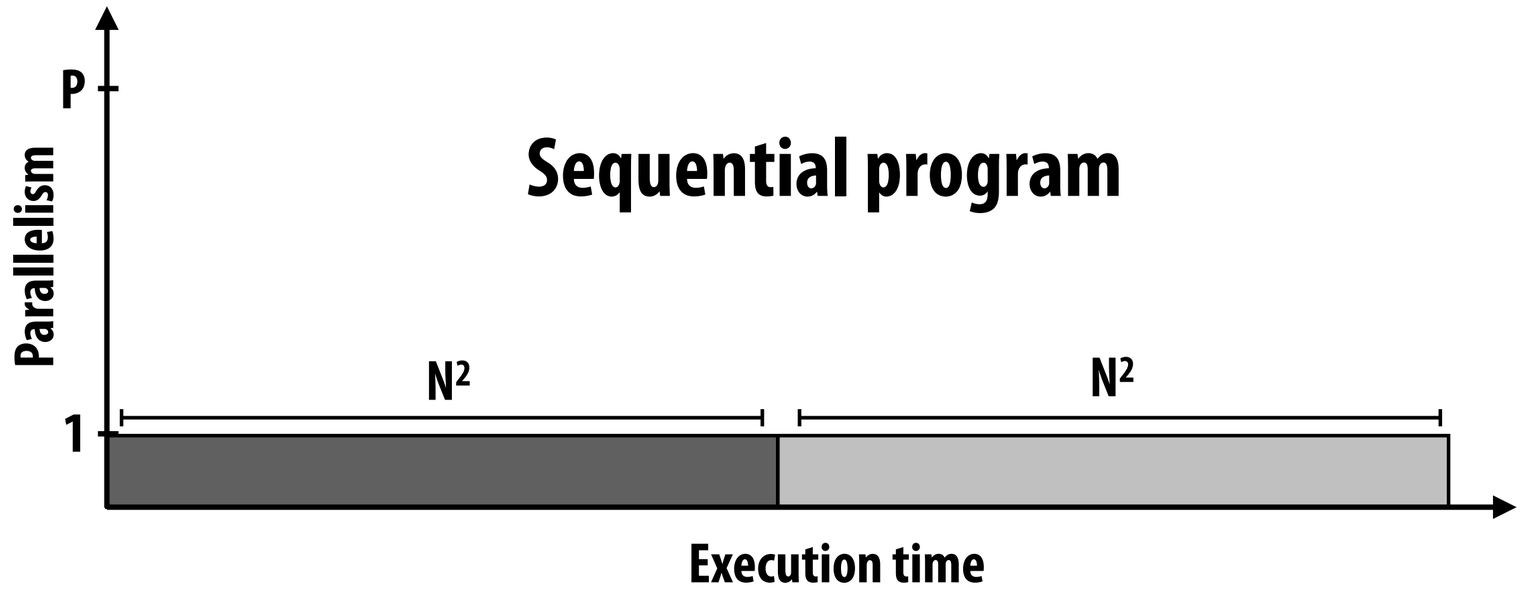
- Sequential implementation of program
  - Both steps take ~ N^2 time, so total time is ~ $2N^2$





First attempt at parallelism (P processors)

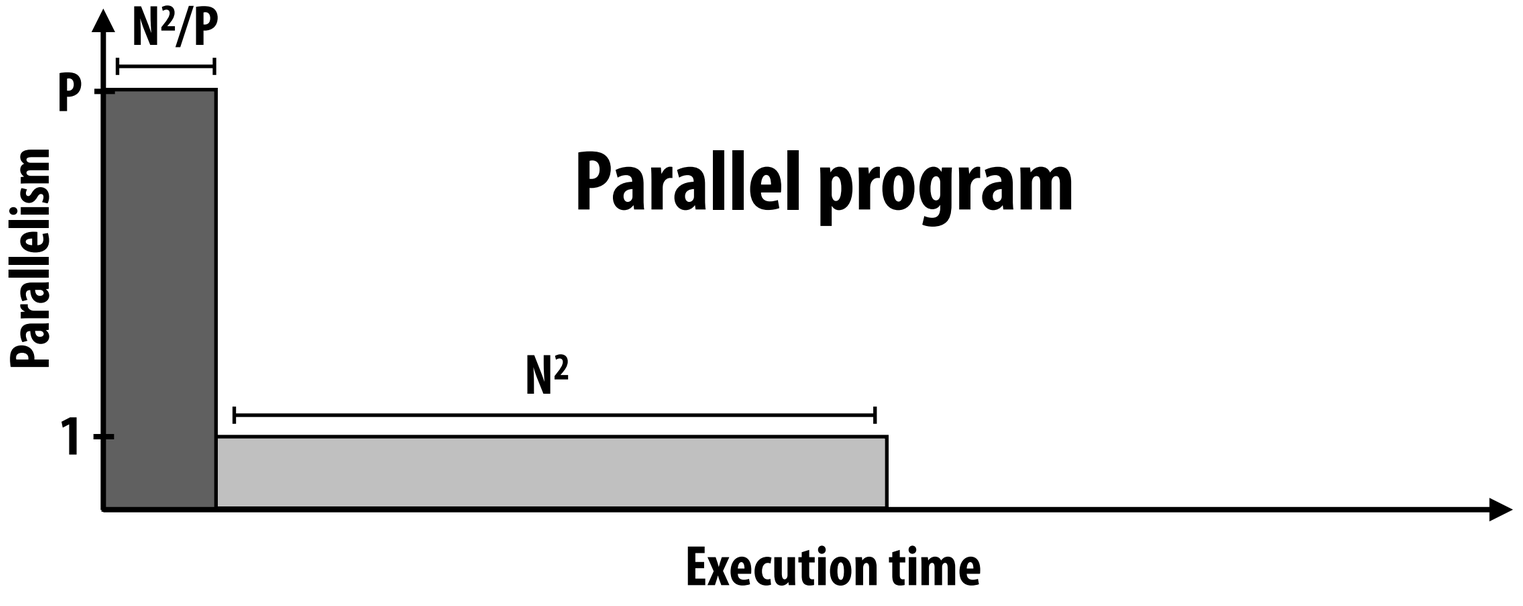
- **Strategy:**
  - Step 1: execute in parallel
    - time for phase 1: N^2/P
  - Step 2: execute serially
    - time for phase 2: N^2



- **Overall performance:**

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

Speedup ≤ 2

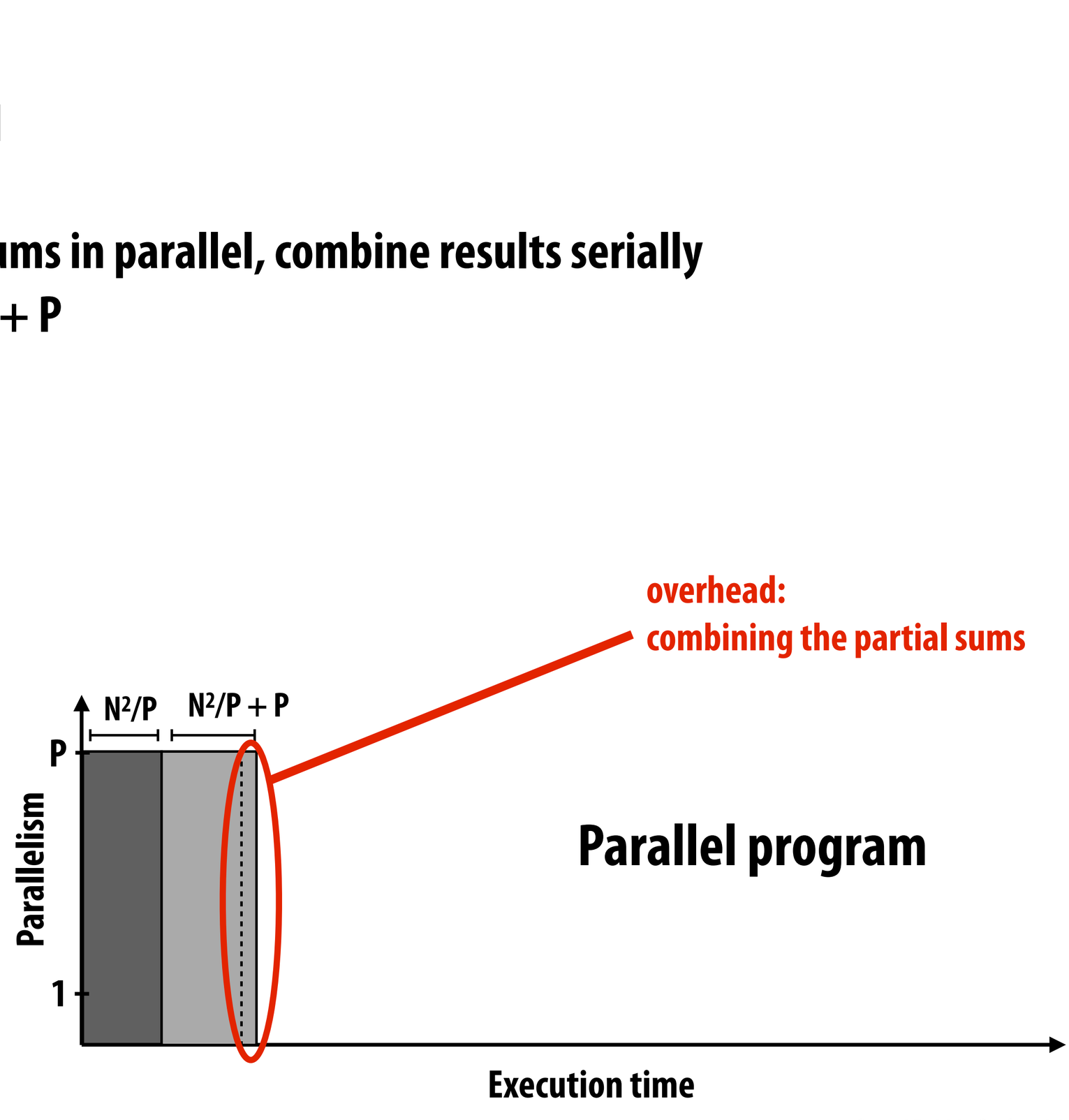


Parallelizing step 2

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

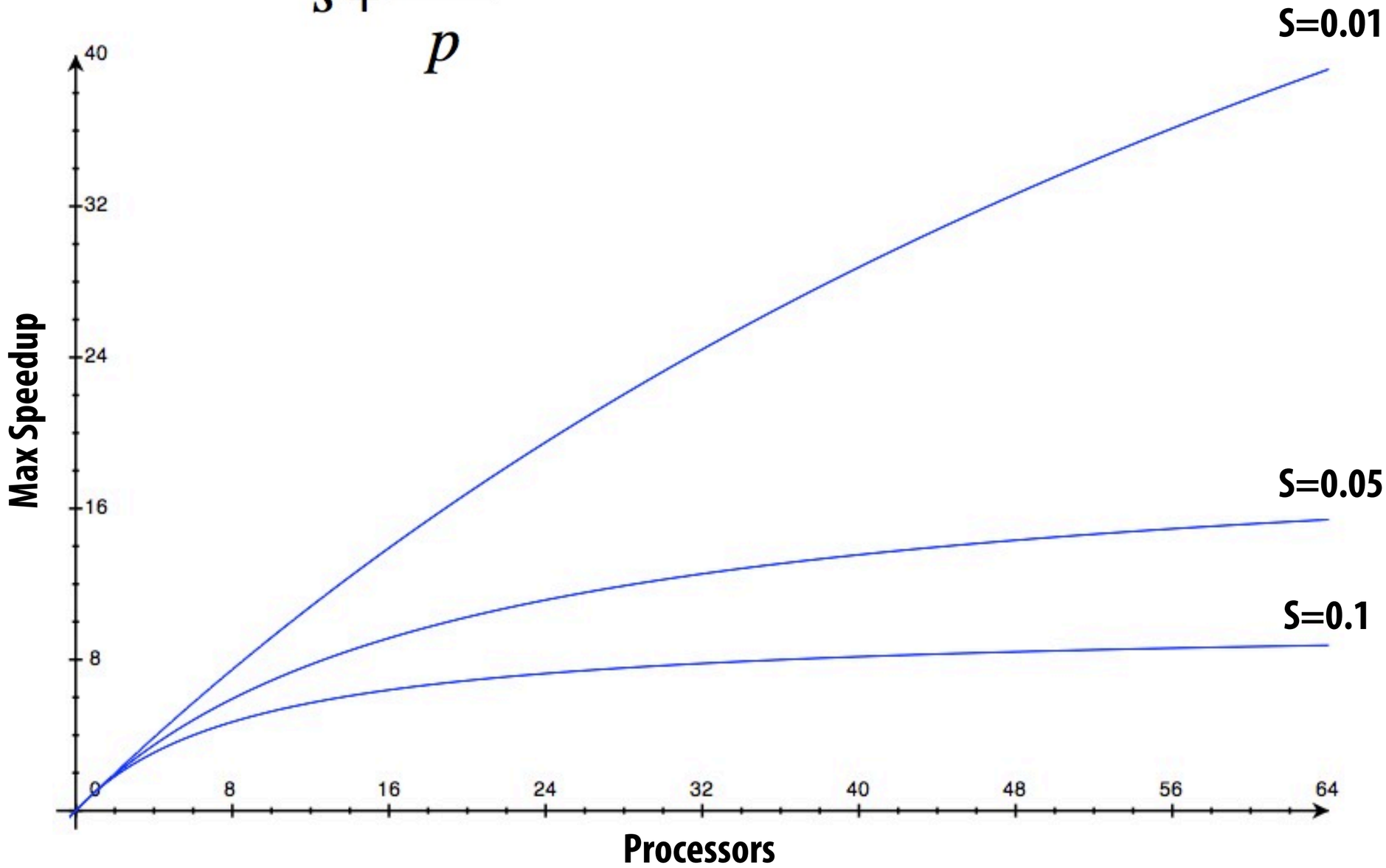
Note: speedup → P when N >> P



Amdahl's law

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

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

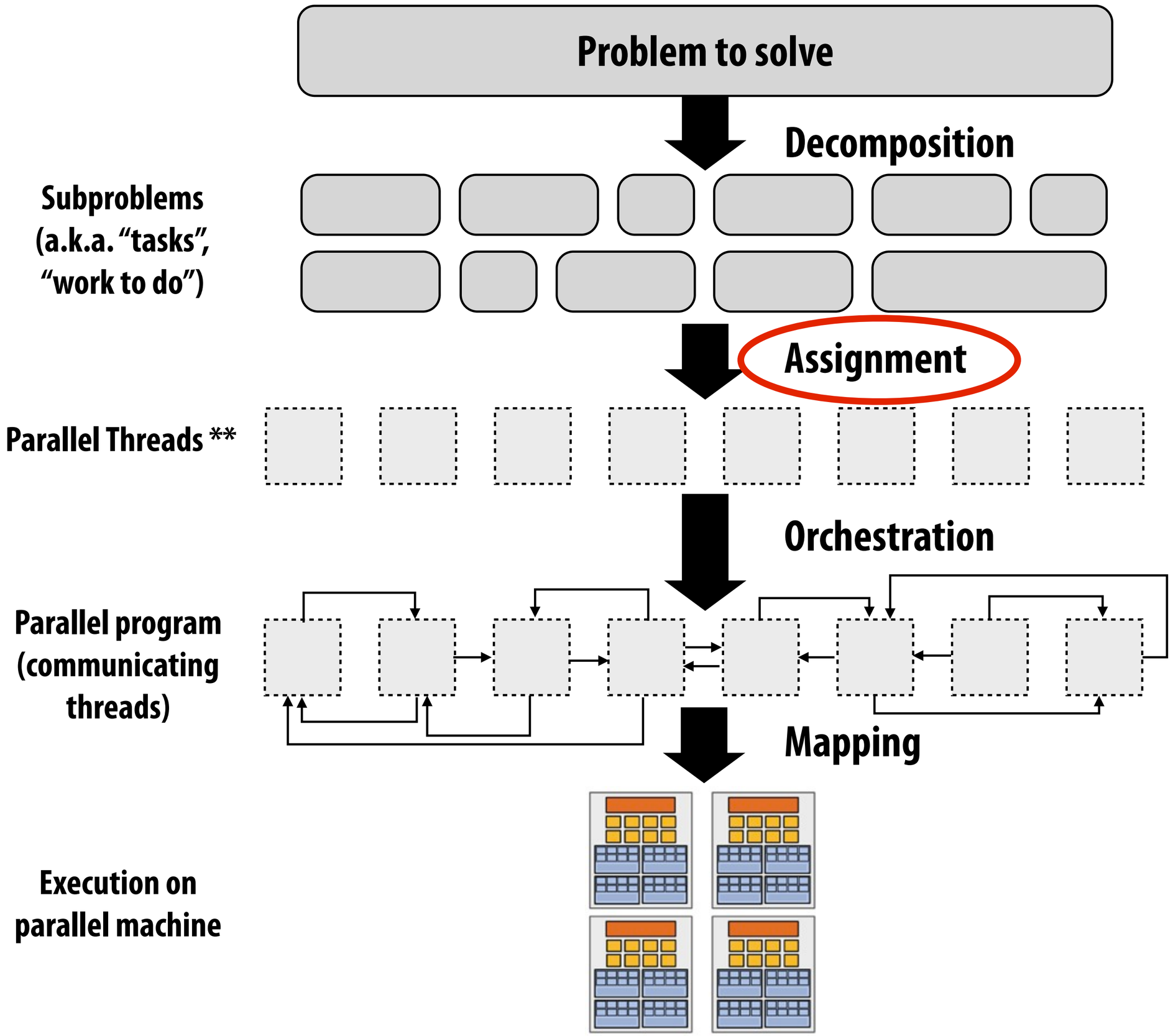


Decomposition

- Who is responsible for performing decomposition?
  - In many cases: the programmer

- Automatic decomposition of sequential programs continues to be a challenging research problem (very difficult in general case)
  - Compiler must analyze program, identifies dependencies
    - What if dependencies are data-dependent (not known at compile time)?
  - Modest success with simple loops, 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 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 by loop iteration

Programmer managed assignment:

- **Static assignment**
- **Assign iterations to instances in interleaved fashion**

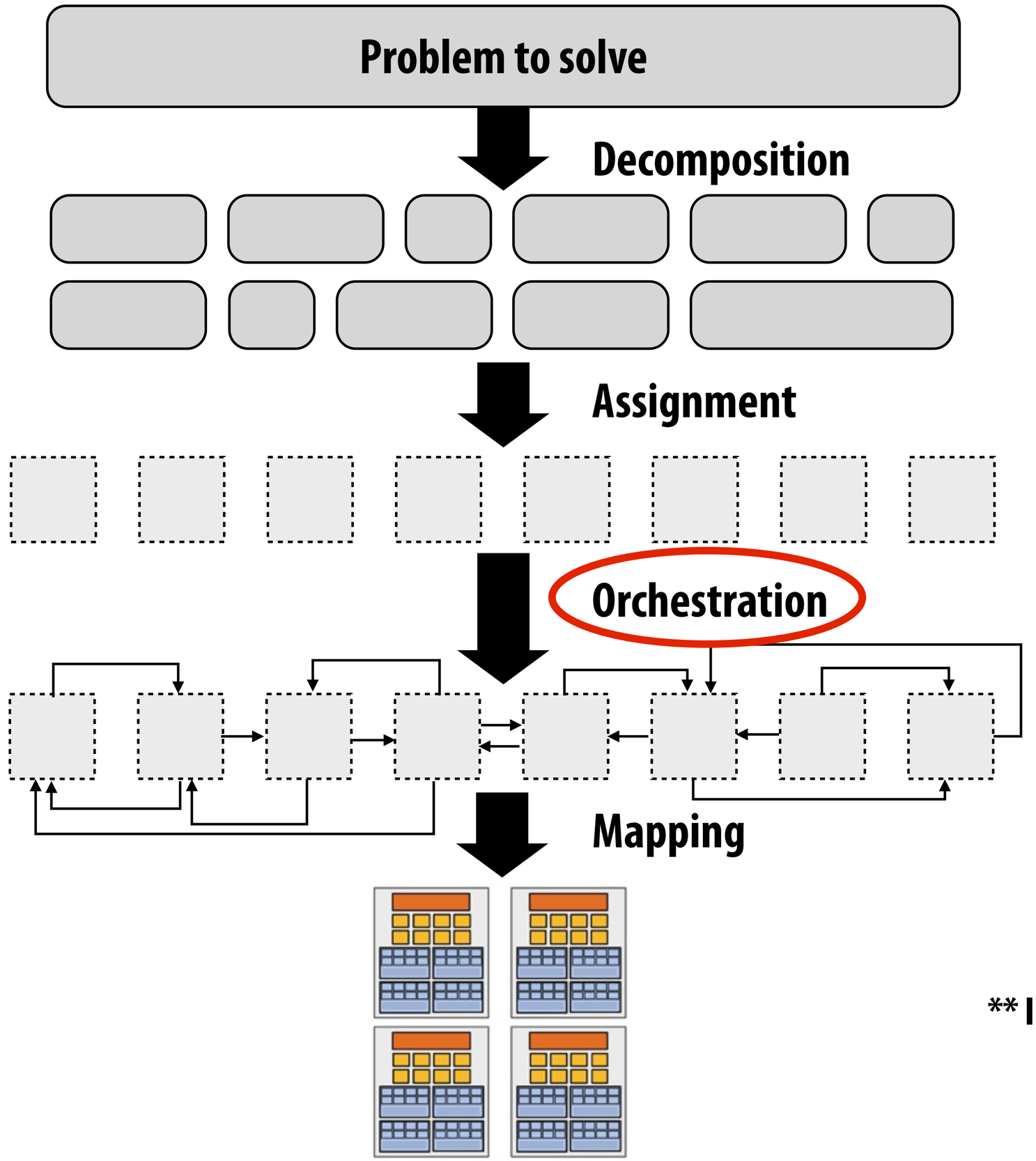
```
export void sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    foreach (i = 0 ... N)
    {
        float value = x[i];
        float numer = x[i] * x[i] * x[i];
        uniform int denom = 6;  // 3!
        uniform int sign = -1;

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

Decomposition by loop iteration

Foreach construct exposes independent work to system

System-manages assignment of iterations (work) to instances

Orchestration



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

Parallel Threads **

Parallel program (communicating threads)

Execution on parallel machine

** I had to pick a term

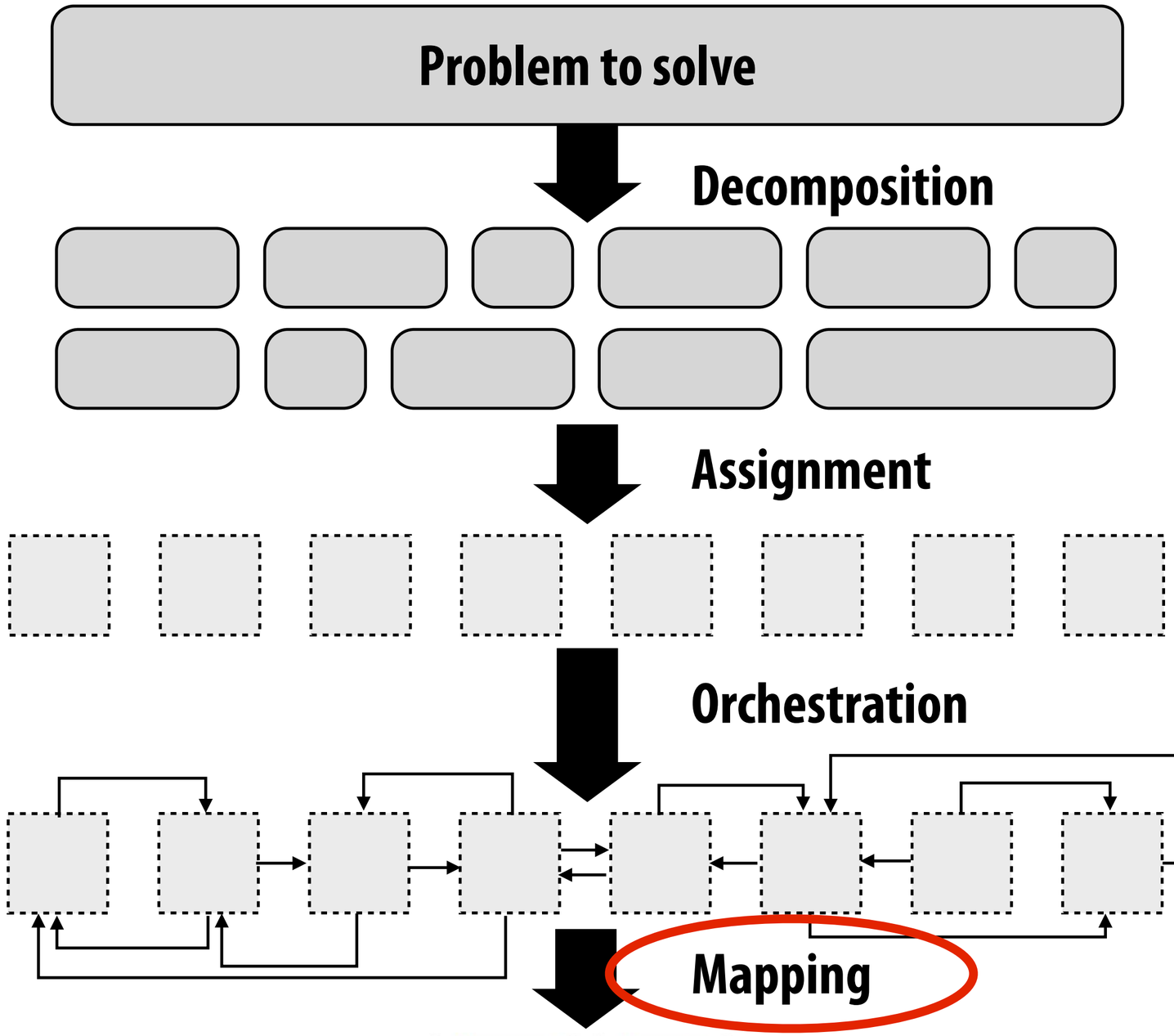
Orchestration

- Involves:
  - Structuring communication
  - Adding synchronization to preserve dependencies
  - 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

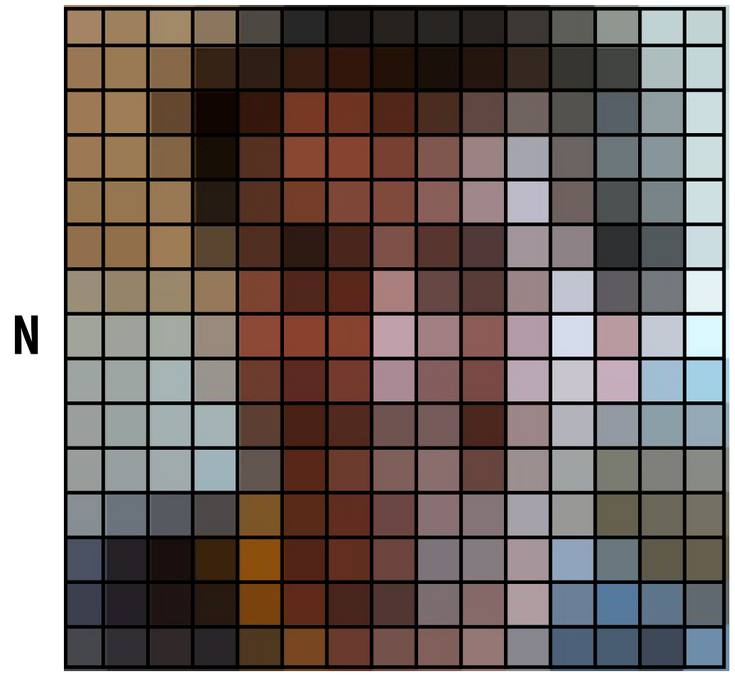
Mapping

- Mapping "threads" to hardware execution units

- Traditionally, a responsibility of the OS
  - e.g., map kernel thread to CPU core execution context
  - Counter example 1: mapping ISPC program instances to vector instruction lanes
  - Counter example 2: mapping CUDA thread blocks to GPU core (future lecture)

- Some interesting mapping decisions:
  - Place related threads (cooperating threads) on the same processor (maximize locality, data sharing, minimize costs of comm/sync)
  - Place unrelated 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 partitioning computation.

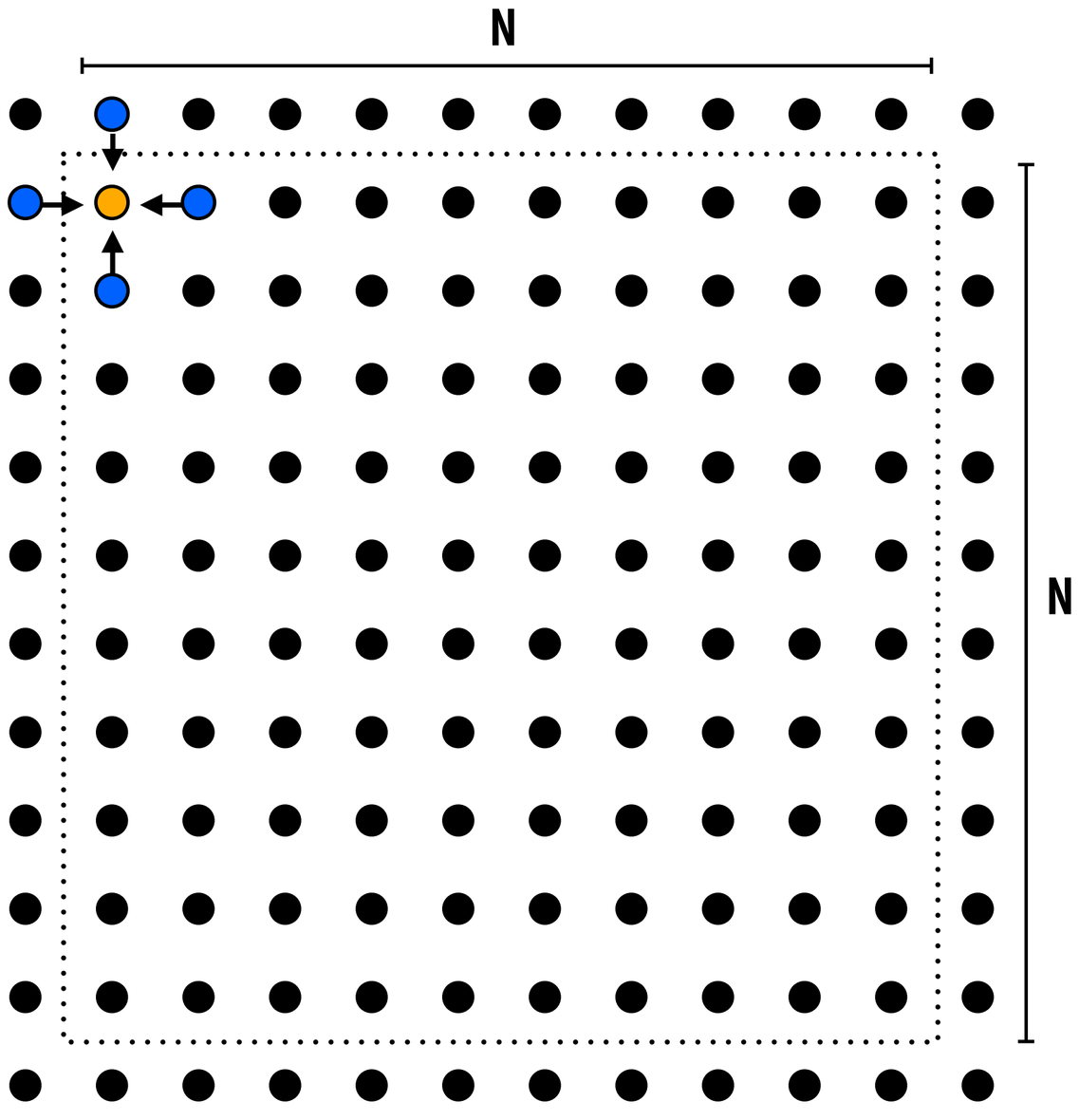
Often, it's equally valid to think of partitioning data. (computations go with the data)

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 on N+2 x N+2 grid
- Iterative solution
  - Perform Gauss-Seidel sweeps over grid until convergence



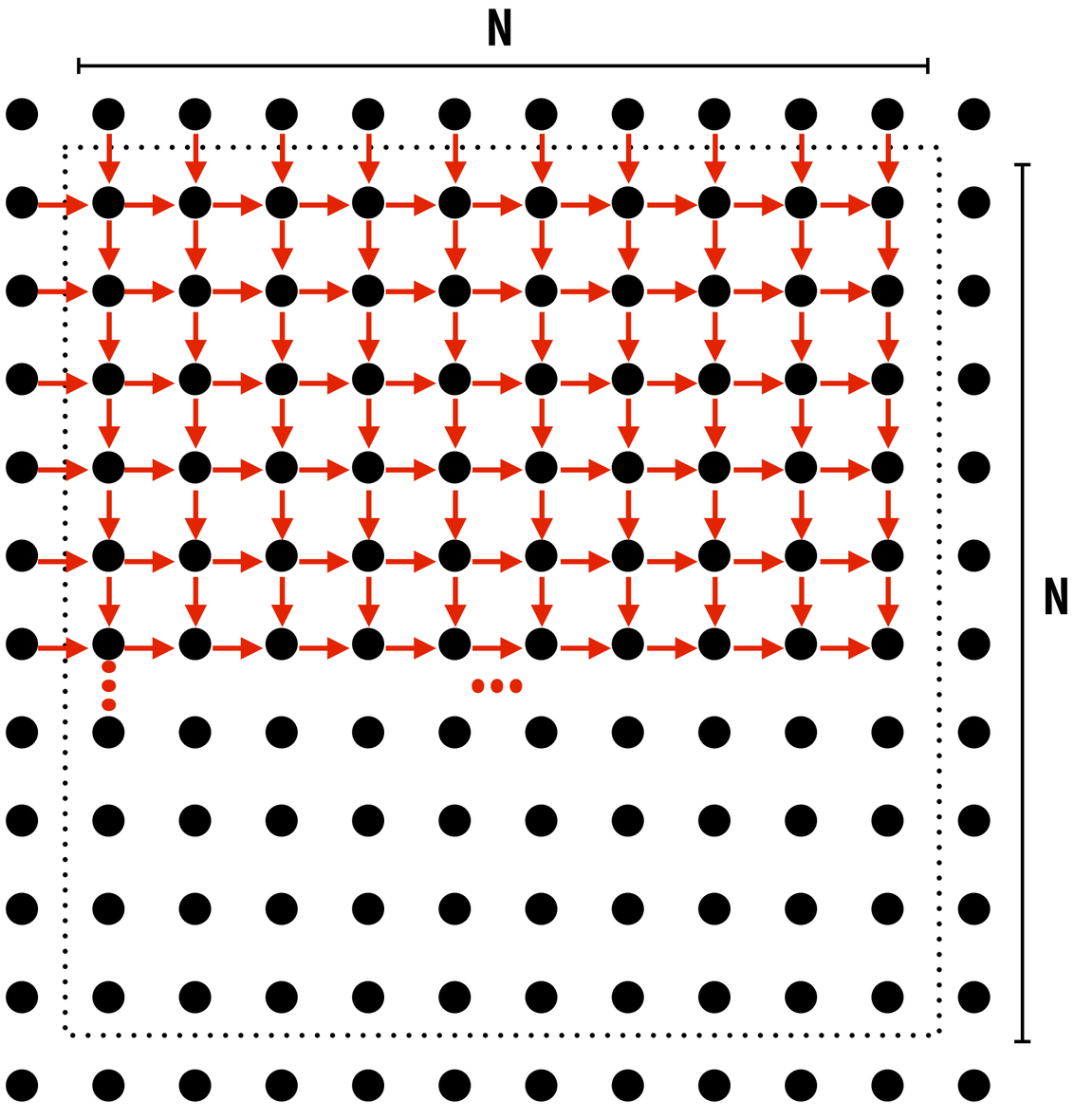

```
A[i,j] = 0.2 * A[i,j] + A[i,j-1] + A[i-1,j]
                      + A[i,j+1] + A[i+1,j];
```

Grid solver algorithm

(generic syntax)

```
1.  int n;                                   /*size of matrix: (n + 2-by-n + 2) elements*/
2.  float **A, diff = 0;

3.  main()
4.  begin
5.     read(n) ;                             /*read input parameter: matrix size*/
6.     A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
7.     initialize(A);                        /*initialize the matrix A somehow*/
8.     Solve (A);                            /*call the routine to solve equation*/
9.  end main

10. procedure Solve (A)                      /*solve the equation system*/
11.    float **A;                            /*A is an (n + 2)-by-(n + 2) array*/
12. begin
13.    int i, j, done = 0;
14.    float diff = 0, temp;
15.    while (!done) do                      /*outermost loop over sweeps*/
16.       diff = 0;                          /*initialize maximum difference to 0*/
17.       for i ← 1 to n do                  /*sweep over nonborder points of grid*/
18.          for j ← 1 to n do
19.             temp = A[i,j];              /*save old value of element*/
20.             A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                A[i,j+1] + A[i+1,j]); /*compute average*/
22.             diff += abs(A[i,j] - temp);
23.          end for
24.       end for
25.       if (diff/(n*n) < TOL) then done = 1;
26.    end while
27. end procedure
```

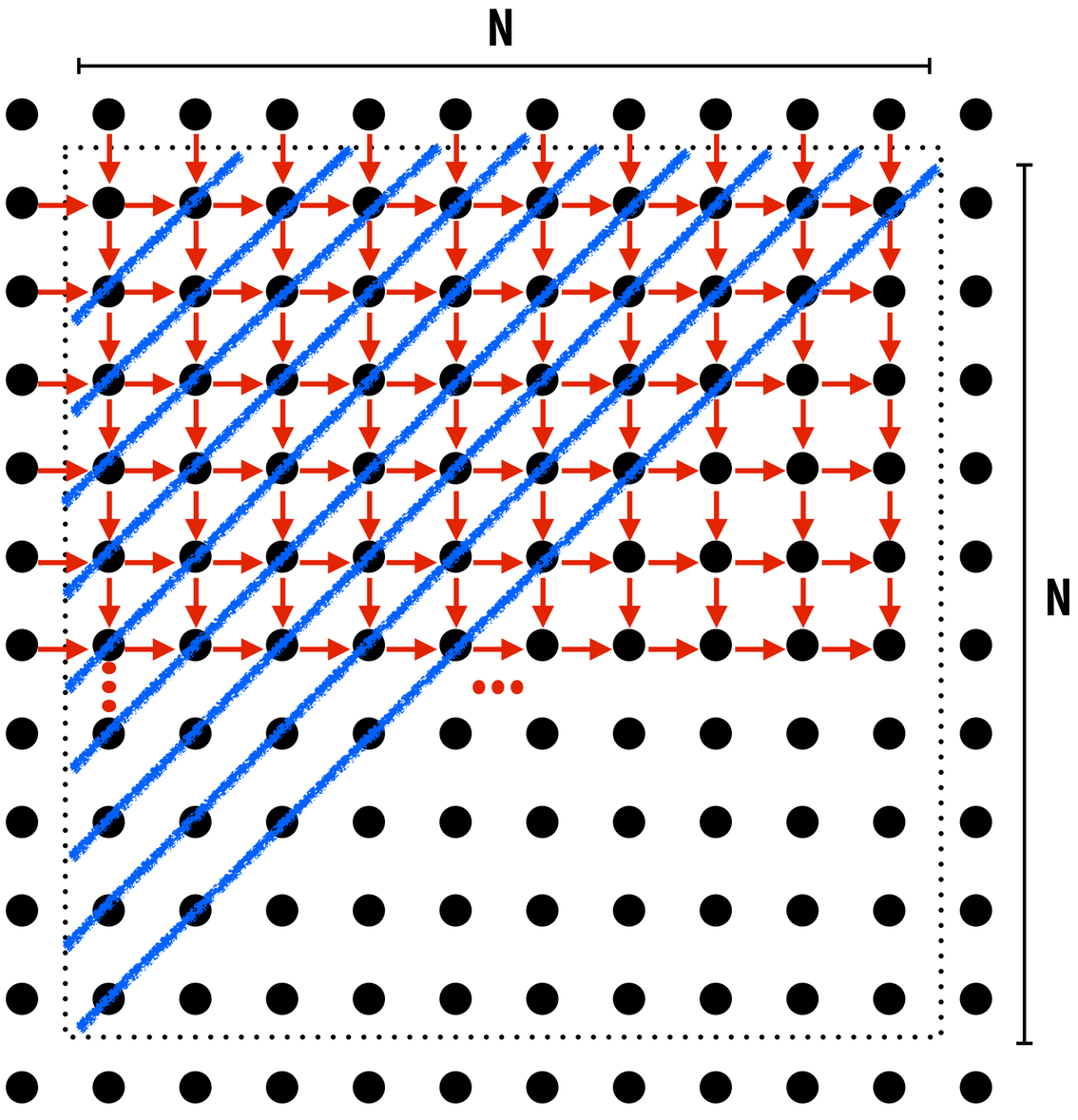
Step 1: identify dependencies (problem decomposition phase)



Each row element depends on element to left.

Each column depends on previous column.

Step 1: identify dependencies (problem decomposition phase)



Parallelism along the diagonals.

Good: parallelism exists!

Possible 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

Early in computation: not much parallelism

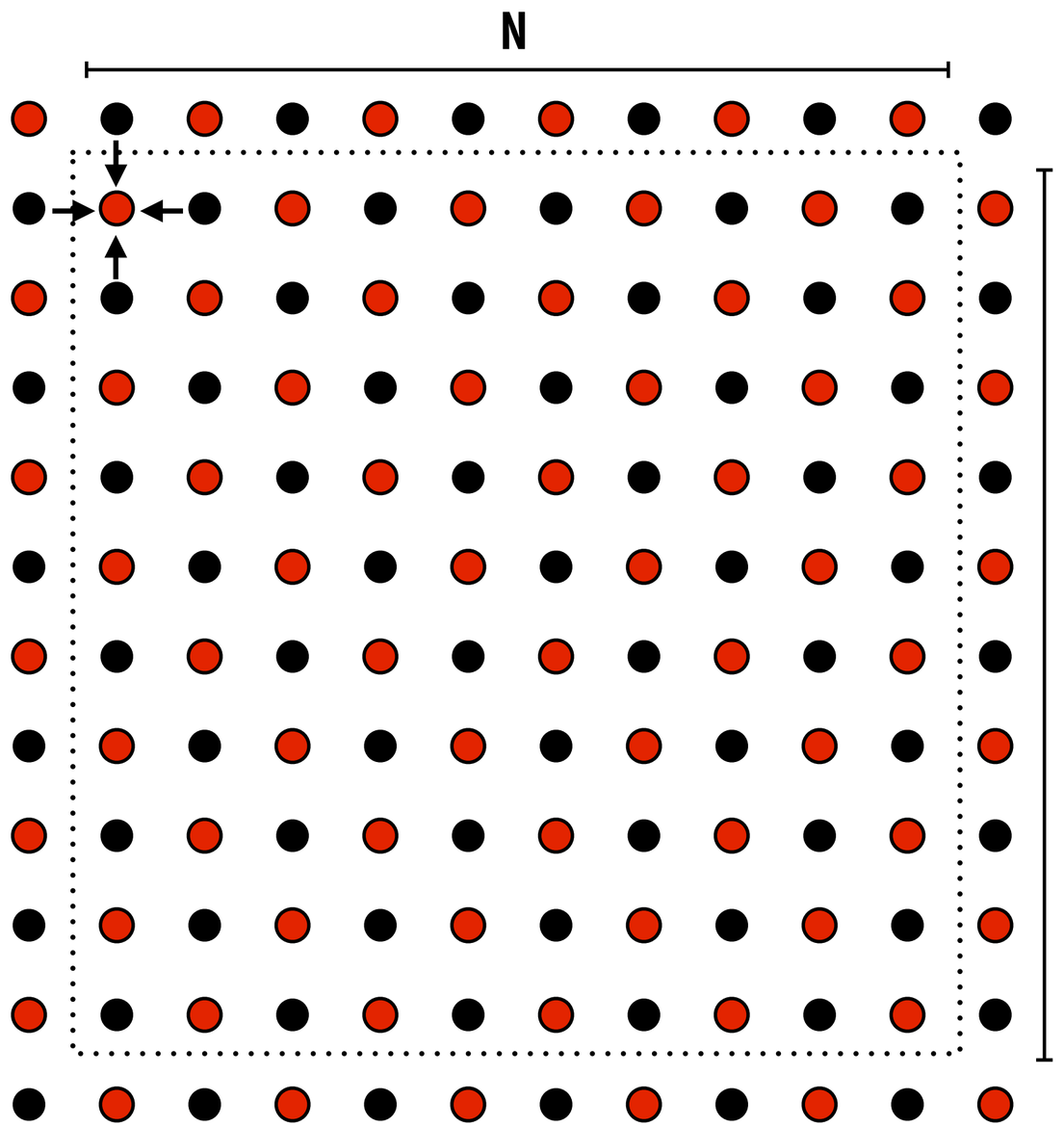
Frequent synchronization (each diagonal)

Key idea: change algorithm

- Change order grid cell cells are updated
- New algorithm iterates to (approximately) same solution, but converges to solution differently
  - Note: floating point values computed are different, but solution still converges to within error threshold
- Domain knowledge: needed knowledge of Gauss-Seidel iteration to realize this change is okay for application's needs

Exploit application knowledge

Reorder grid traversal: red-black coloring

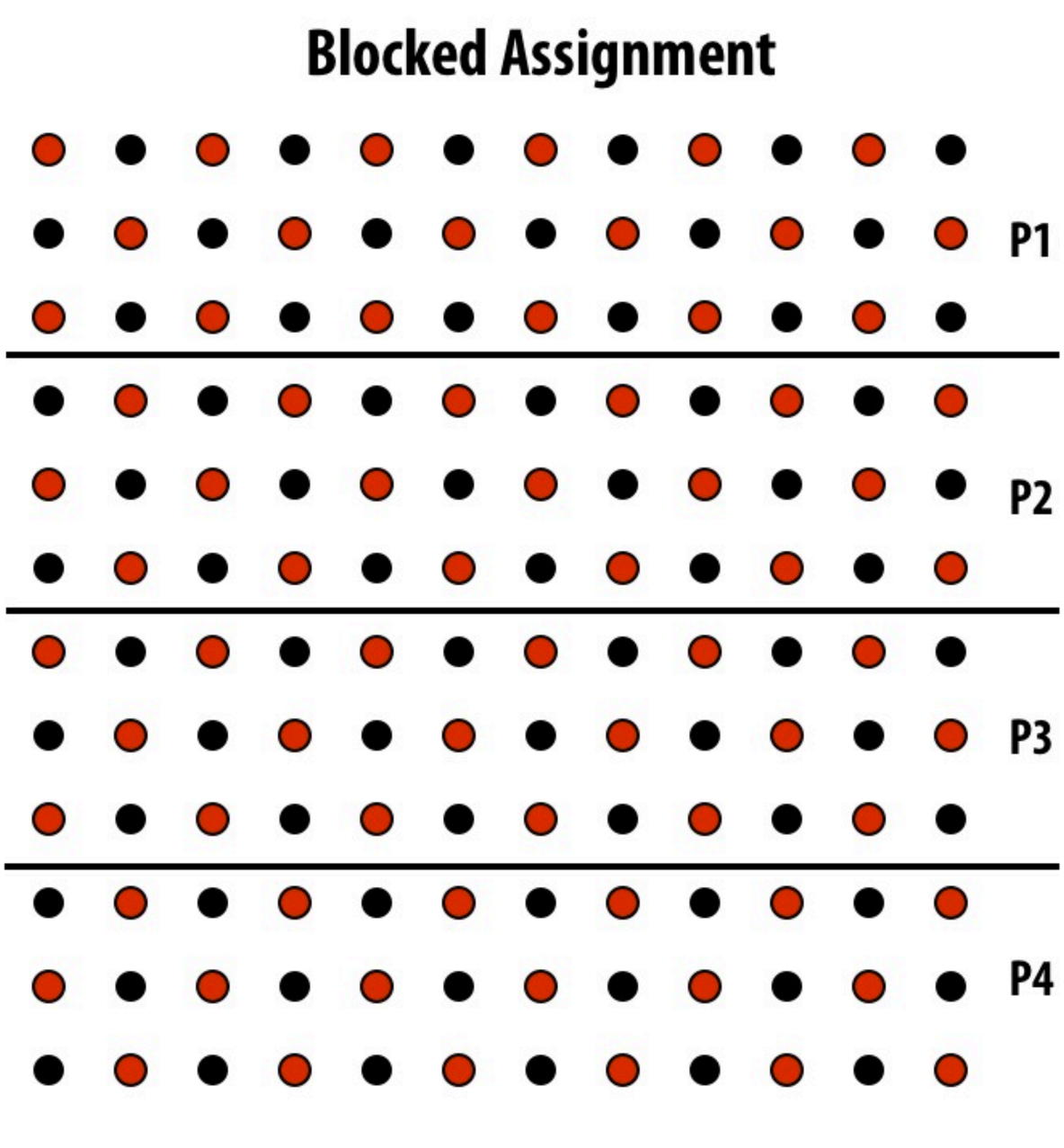


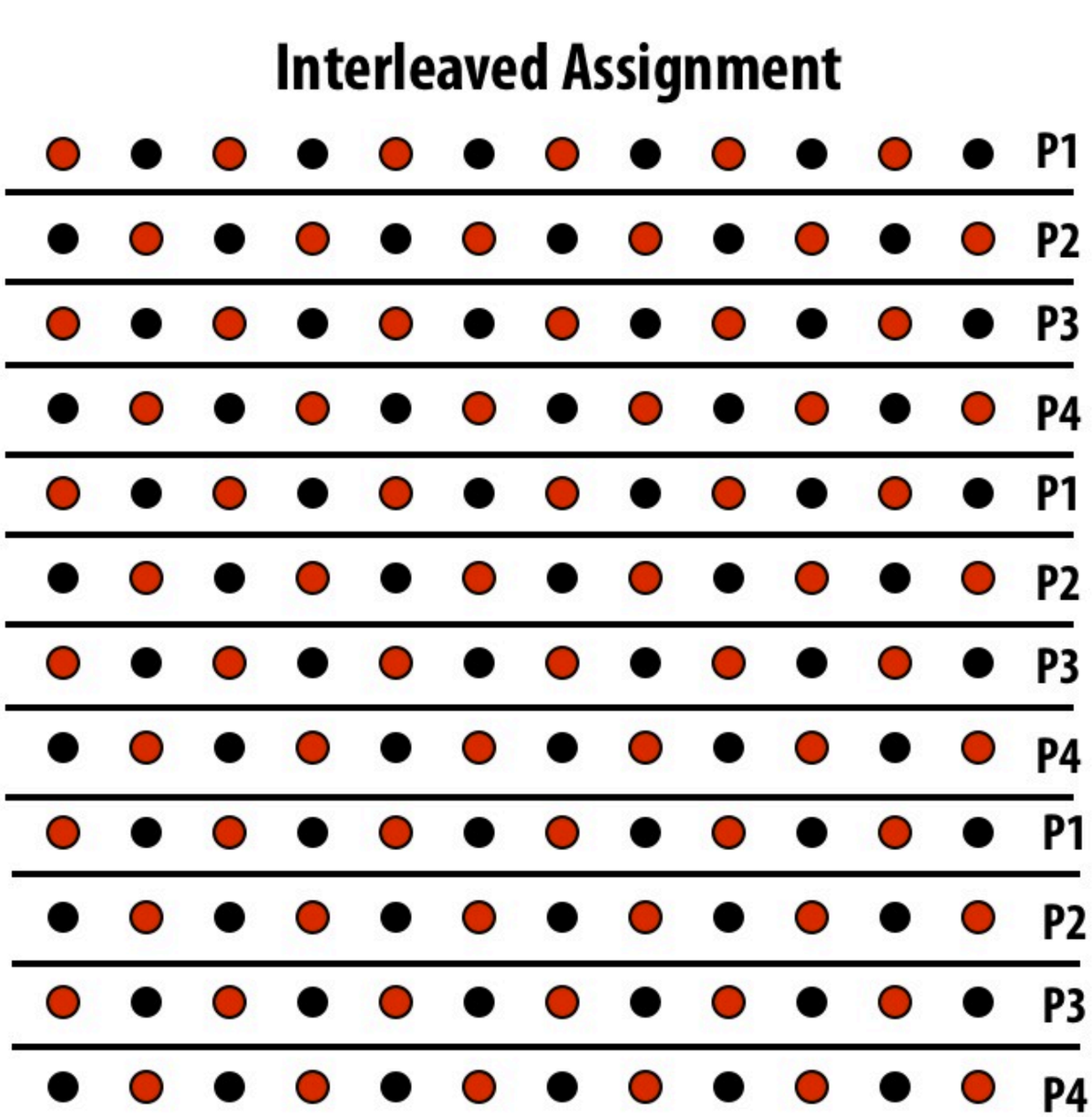
Update all red cells in parallel

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

Repeat until convergence

Assignment

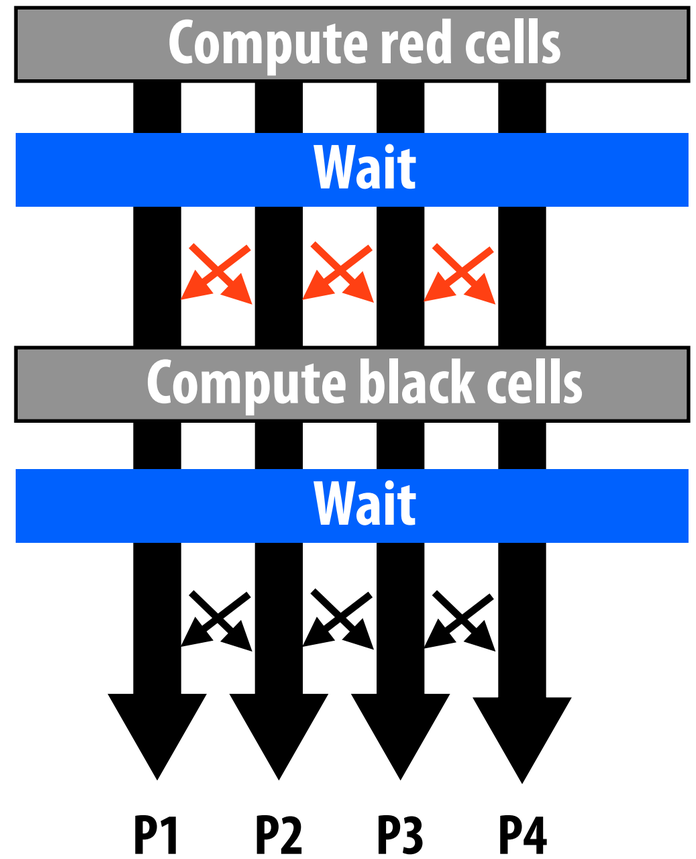




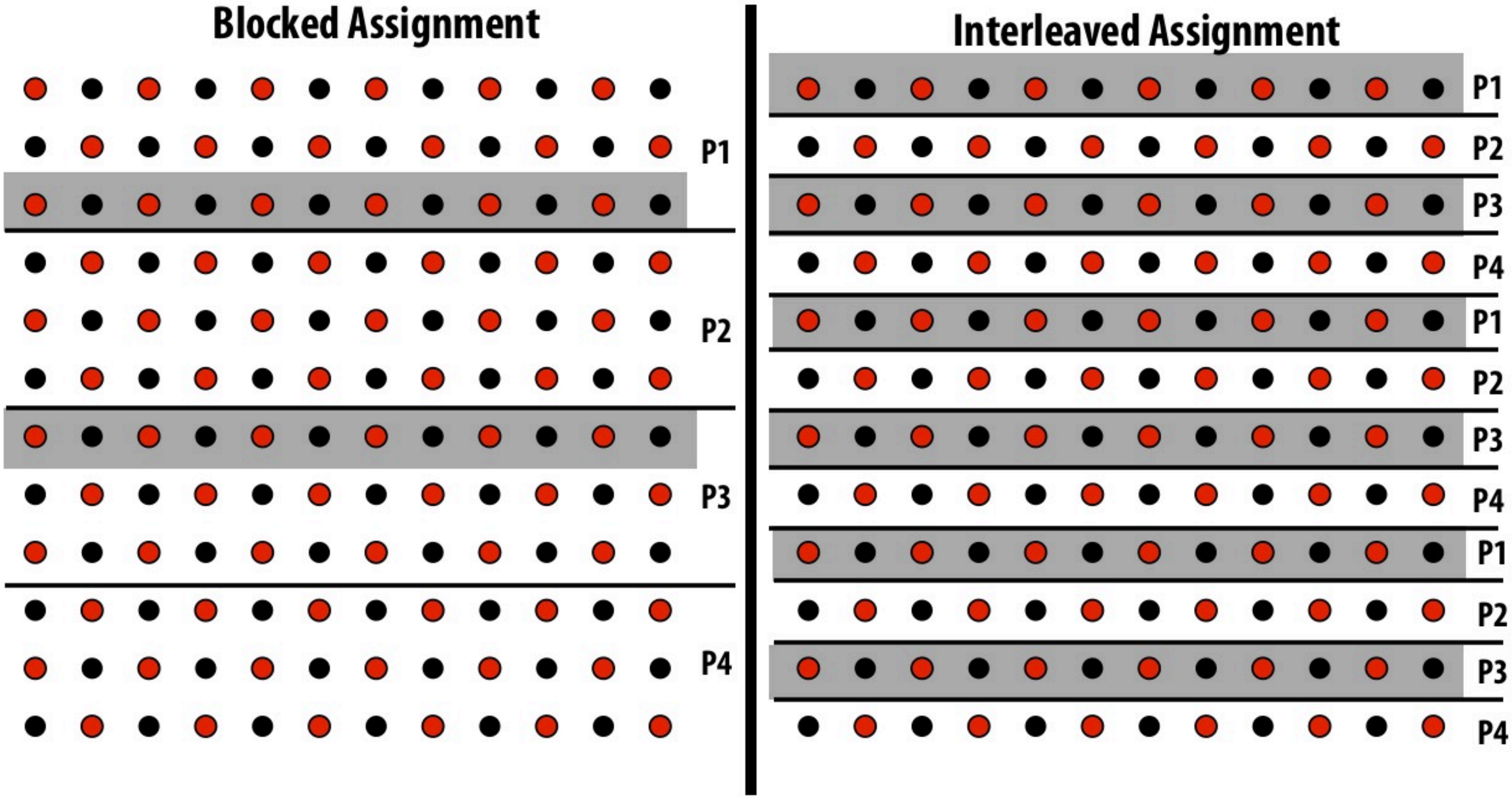
Which is better? Does it matter?

Consider dependencies (data flow)

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



Assignment



= data that must be sent to P2 each iteration

Blocked assignment requires less data to be communicated between processors

Grid solver: data-parallel expression

To simplify code: we've dropped red-black separation, now ignoring dependencies

```
1.  int n, nprocs;                          /*grid size (n + 2-by-n + 2) and number of processes*/
2.  float **A, diff = 0;

3.  main()
4.  begin
5.    read(n); read(nprocs);      ;         /*read input grid size and number of processes*/
6.    A ← G_MALLOC (a 2-d array of size n+2 by n+2 doubles);
7.    initialize(A);                        /*initialize the matrix A somehow*/
8.    Solve (A);                            /*call the routine to solve equation*/
9.  end main

10. procedure Solve(A)                      /*solve the equation system*/
11.     float **A;                          /*A is an (n + 2-by-n + 2) array*/
12.   begin
13.   int i, j, done = 0;
14.   float mydiff = 0, temp;
14a.    DECOMP A[BLOCK,*, nprocs];
15.   while (!done) do                      /*outermost loop over sweeps*/
16.     mydiff = 0;                         /*initialize maximum difference to 0*/
17.     for_all i ← 1 to n do               /*sweep over non-border points of grid*/
18.       for all j ← 1 to n do
19.         temp = A[i,j];                  /*save old value of element*/
20.         A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.           A[i,j+1] + A[i+1,j]);         /*compute average*/
22.         mydiff += abs(A[i,j] - temp);
23.       end for_all
24.     end for_all
24a.      REDUCE (mydiff, diff, ADD);
25.     if (diff/(n*n) < TOL) then done = 1;
26.   end while
27. end procedure
```

assignment: specified explicitly (block assignment)

decomposition: tasks are individual elements

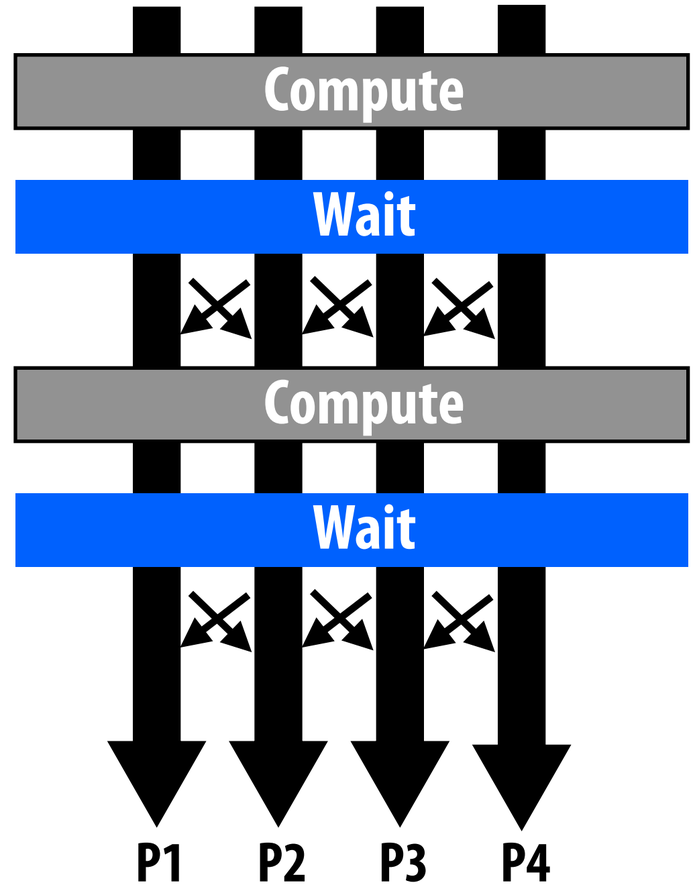
Orchestration: handled by system

(End of for_all block is implicit wait for all workers before returning to sequential control)

Shared address space solver

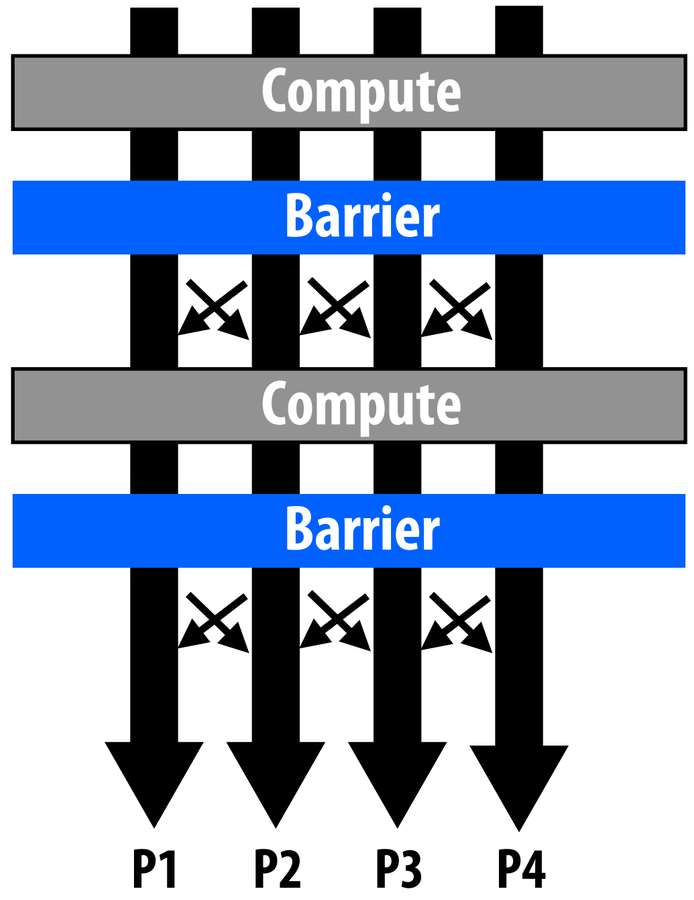
SPMD execution model

- Programmer is responsible for synchronization
- Common synchronization primitives:
  - Locks (mutual exclusion): only one thread in the critical region at a time
  - Barriers: wait for threads to reach this point



Barriers

- `Barrier(nthreads)`
- 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)

```
LOCKDEC(diff_lock);          /*declaration of lock to enforce mutual exclusion*/
BARDEC (bar1);               /*barrier declaration for global synchronization between
                             sweeps*/

procedure Solve(A)
   float **A;                                  /*A is entire n+2-by-n+2 shared array,
                                               as in the sequential program*/
begin
   int i,j, pid, done = 0;
   float temp, mydiff = 0;                     /*private variables*/
   int mymin = 1 + (pid * n/nprocs);           /*assume that n is exactly divisible by*/
   int mymax = mymin + n/nprocs - 1            /*nprocs for simplicity here*/

   while (!done) do                            /*outer loop over all diagonal elements*/
      mydiff = diff = 0;                       /*set global diff to 0 (okay for all to do it)*/
   BARRIER(bar1, nprocs);
      for i ← mymin to mymax do                /*for each of my rows*/
            for j ← 1 to n do                  /*for all nonborder elements in that row*/
            temp = A[i,j];
            A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
               A[i,j+1] + A[i+1,j]);
            mydiff += abs(A[i,j] - temp);
         endfor
      endfor
      LOCK(diff_lock);
      diff += mydiff;
      UNLOCK(diff_lock);
      BARRIER(bar1, nprocs);
      if (diff/(n*n) < TOL) then done = 1;     /*check convergence; all get
                                               same answer*/
      BARRIER(bar1, nprocs);
   endwhile
end procedure
```

Value of pid is different for each SPMD instance: use value to compute region of grid to work on

partial sum

Why are there so many barriers?

Shared address space solver: one barrier

```
float diff[3];  // global diff
float mydiff;   // thread local variable
int index = 0;  // thread local variable

LOCKDEC(diff_lock);
BARDEC(bar);

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

while (!done) {
   mydiff = 0.0f;
   //
   // perform computation (accumulate locally into mydiff)
   //
   lock(diff_lock);
   diff[index] += mydiff;    // atomically update global diff
   unlock(diff_lock);
   diff[(index+1) % 3] = 0.0f;
   barrier(nprocs, bar);
   if (diff[index]/(n*n) < TOL)
      break;
   index = (index + 1) % 3;
}
```

Idea:

Remove dependencies by using different diff variables in successive loop iterations

Trade off footprint for reduced synchronization! (common parallel programming technique)

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	T0 stores 1 to diff

- Need set of three instructions to be atomic

Mechanisms for atomicity

- Lock/Unlock mutex variable around critical section

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

- Some languages have first-class support

```
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

More on specifying dependencies

- **Barriers: simple, but conservative (coarse granularity)**
  - **Everything done up until now must finish, then before next phase**
- **Specifying specific dependencies can increase performance (by revealing more parallelism)**
  - **Example: two threads. One produces a result, the other consumes it.**

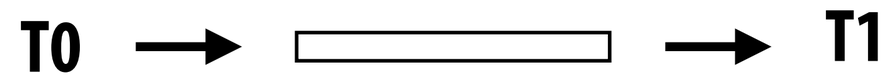
T0

```
// 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: 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

Summary

- **Amdahl's Law**
  - Overall speedup limited by amount of serial execution in code

- **Steps in creating a parallel program**
  - Decomposition, assignment, orchestrating, mapping
  - We'll talk a lot about making good decisions in each of these phases in coming lectures (in practice, very inter-related)

- **Focus today: identifying dependencies**

- **Focus soon: identifying locality**