Parallel Programming

ISPC abstractions

```
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?

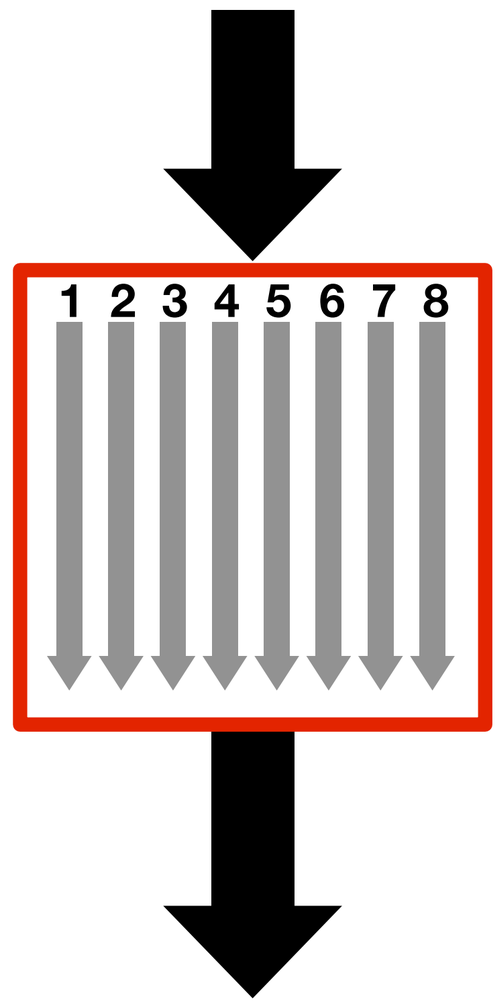
Program instances (that run in parallel) were created when the sinx() ispc function was called

```
#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);
```



Sequential execution (C code)

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

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

Sequential execution (C code)

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

Decomposition

Problem to solve

Decomposition

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

Assignment

Parallel Threads ** (“workers”)

Orchestration

Parallel program (communicating threads)

Mapping

Execution on parallel machine

**I had to pick a term

Creating a parallel program

- Thought process:
  1. Identify work that can be performed in parallel
  2. Partition work (and also data associated with the work)
  3. Manage data access, communication, and synchronization
- A common goal is maximizing speedup *
  For a fixed computation:

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

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

Problem decomposition

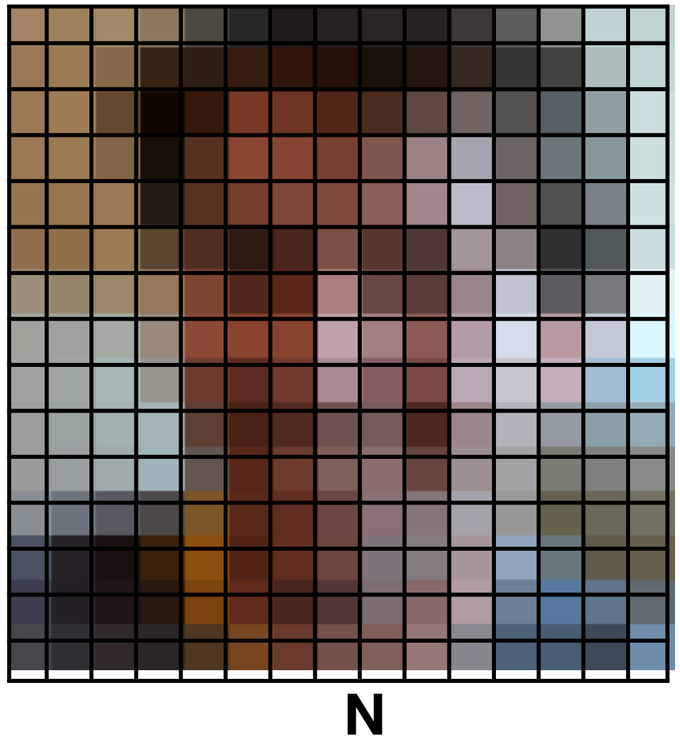
- Break up problem into tasks that can be carried out in parallel

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

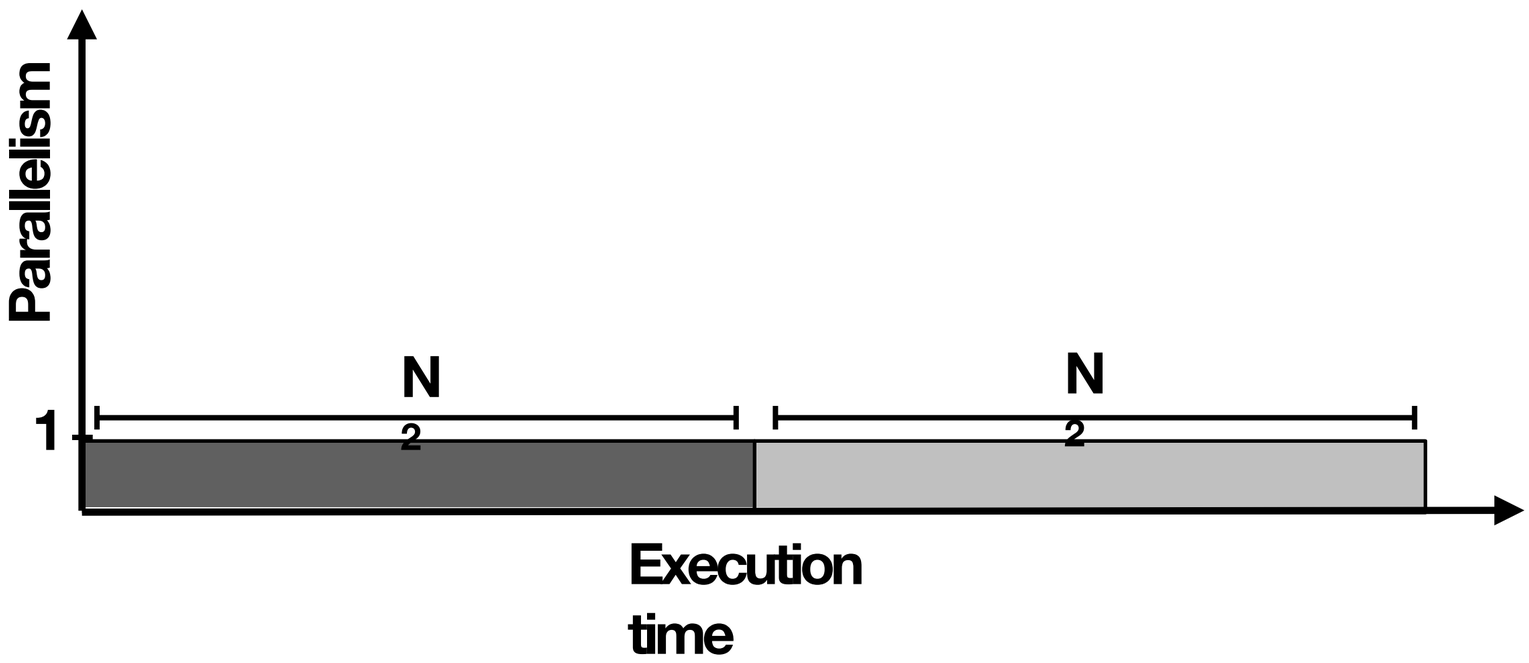
Key challenge of
decomposition:
identifying dependencies
(or... a lack of dependencies)

Asimple example

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

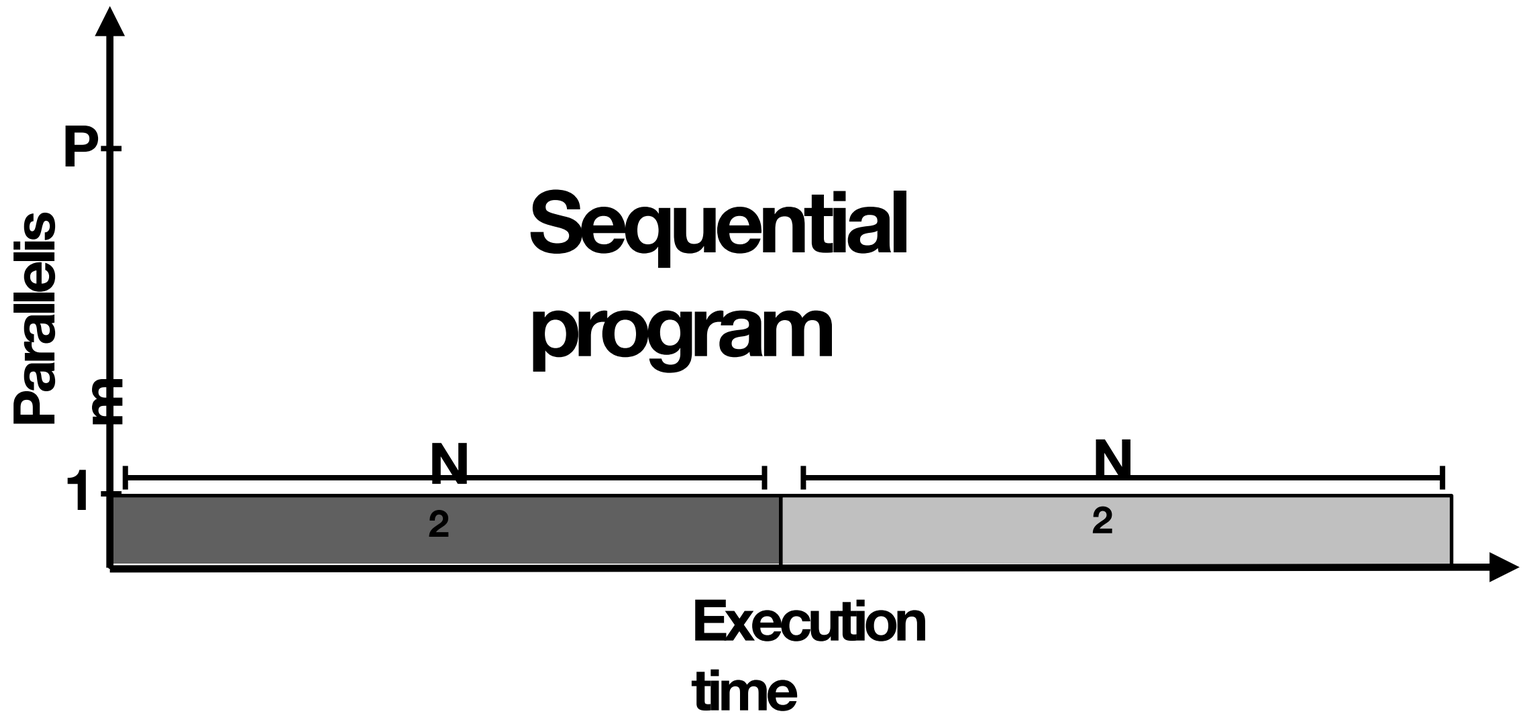


- Sequential implementation of program
  - Both steps take ~ N^2 time, so total time is ~ $2N^2$



First attempt at parallelism (P processors)

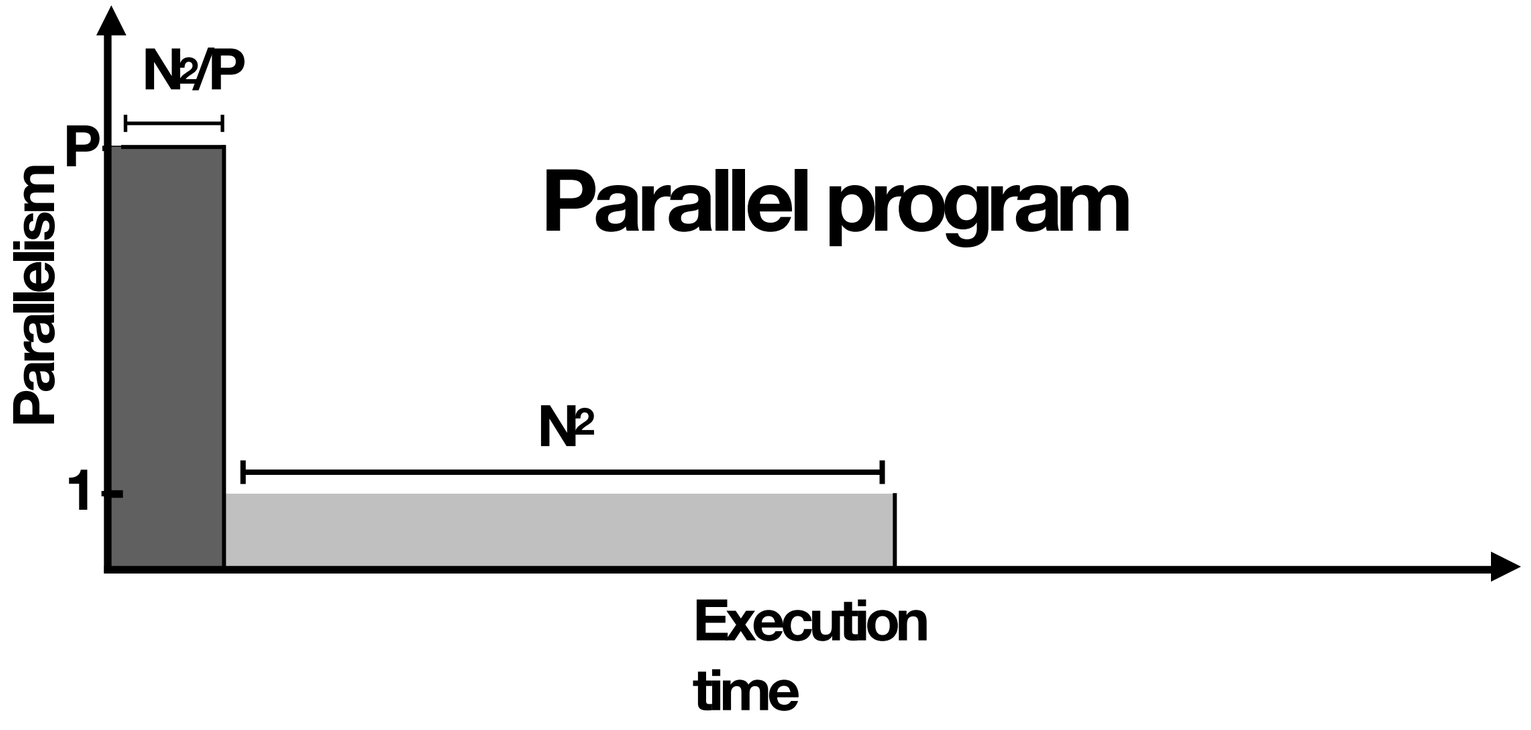
- **Strategy:**
  - Step 1: execute in parallel
    - time for phase 1: N^2/P
  - Step 2: execute serially
    - time for phase 2: N^2



- **Overall performance:**

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

Speedup ≤ 2

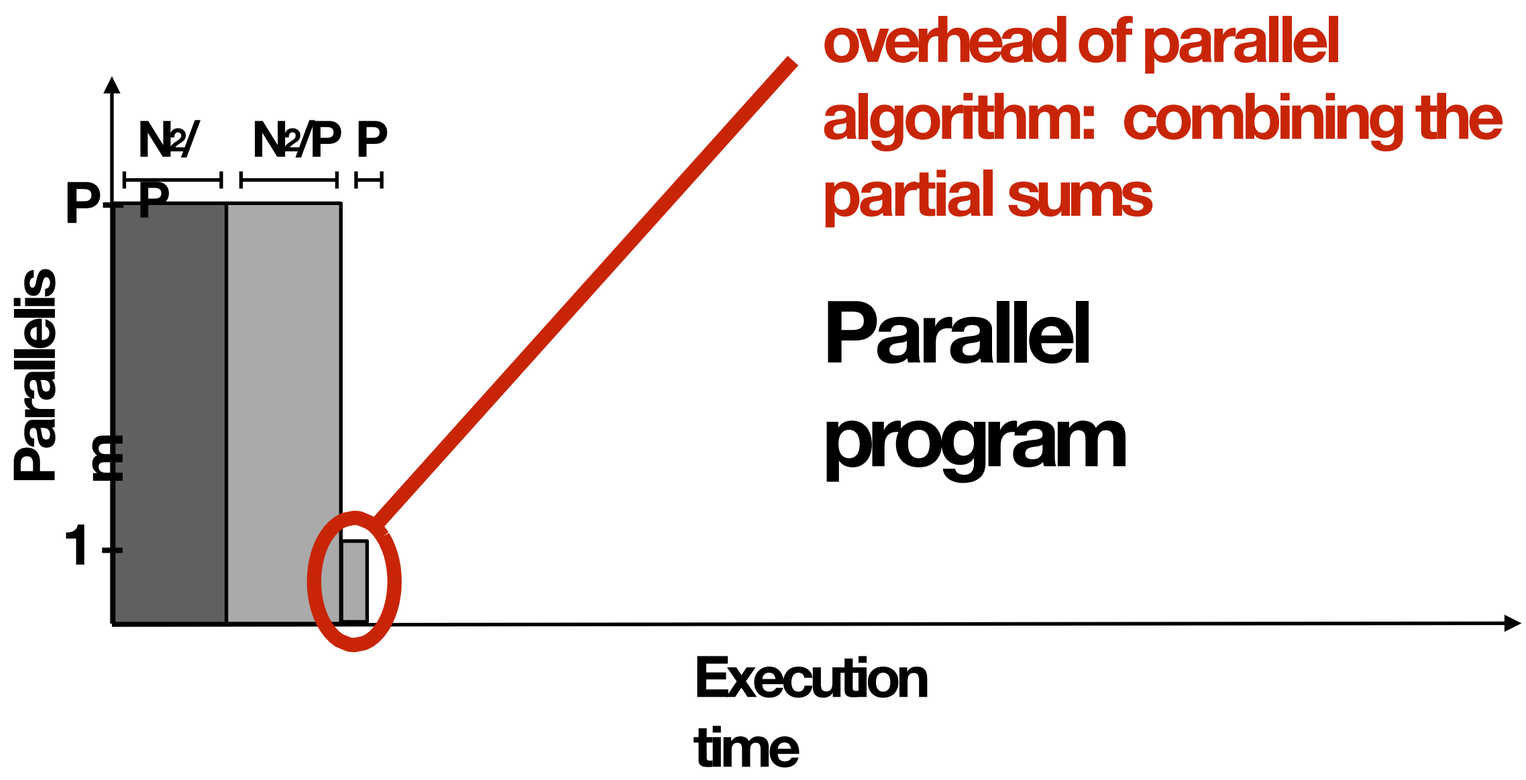


Parallelizing step 2

- 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

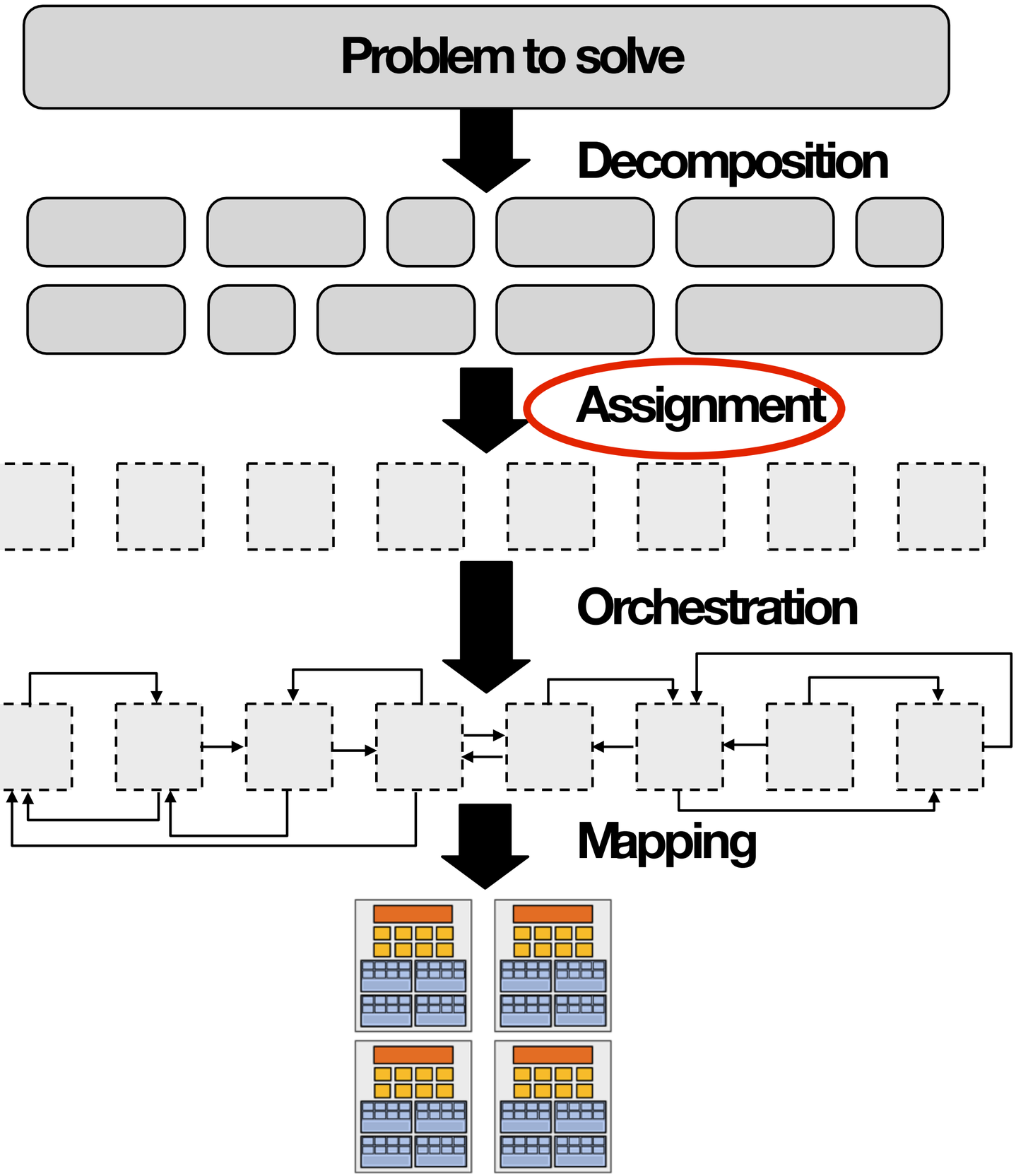


Decomposition

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

- Automatic decomposition of sequential programs continues to be a challenging research problem (very difficult in general case)
  - Compiler must analyze program, identify dependencies
    - What if dependencies are data dependent (not known at compile time)?
  - Researchers have had modest success with simple loop nests
  - The “magic parallelizing compiler” for complex, general-purpose code has not yet been achieved

Assignment



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

Assignment

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

- Goals: achieve good workload balance, reduce communication costs

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

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

ISPC Demos

```
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;
   }
}
```

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;
   }
}
```

Loop assignment

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)

Orchestration

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

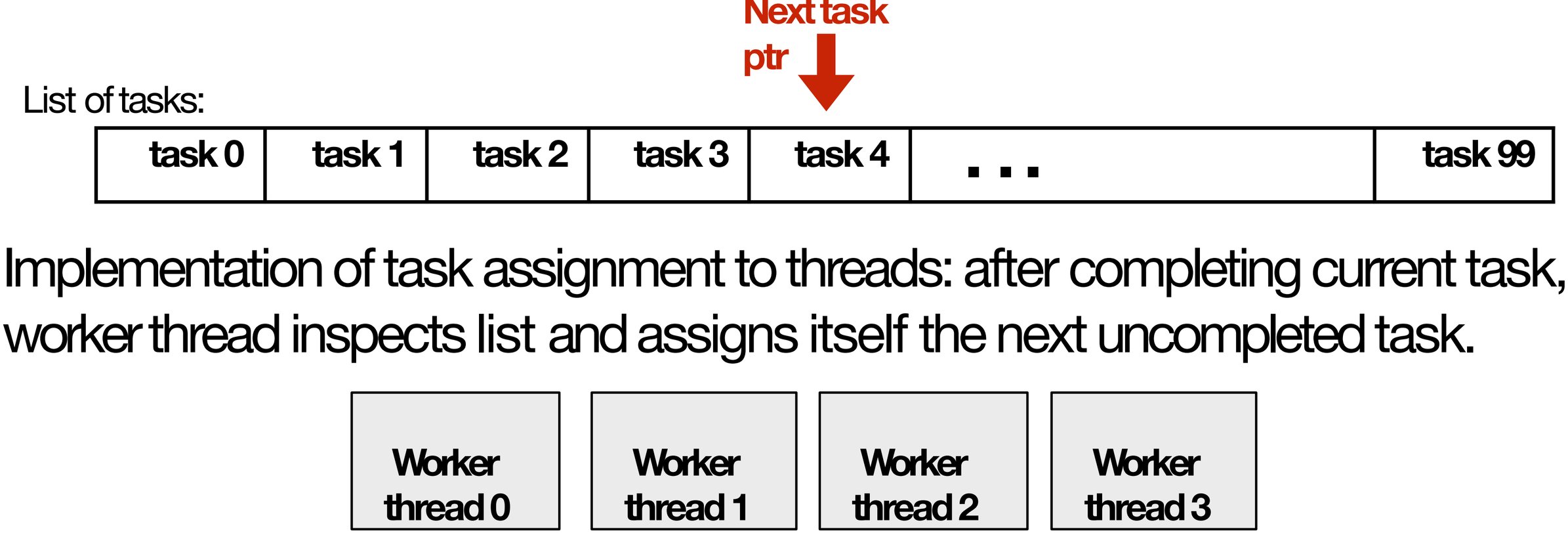
- Goals: reduce costs of communication/sync, preserve locality of data reference, reduce overhead, etc.

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

Dynamic assignment using ISPC tasks

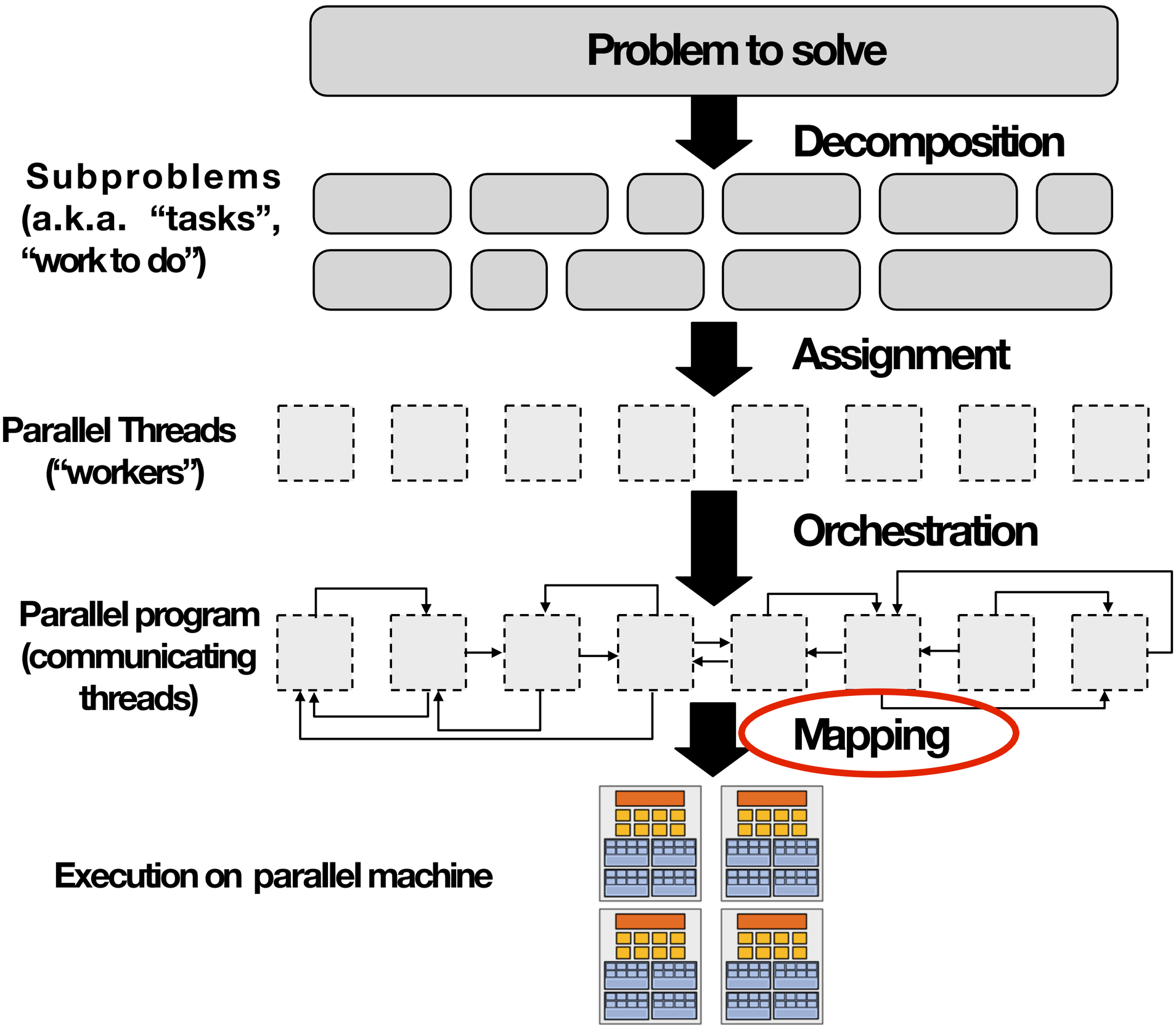
```
void foo(uniform float*
         input,  uniform
         float* output,
         uniform int N)
{
  // create a bunch of tasks
  launch[100] my_ispc_task(input, output,
  N);
}
```

ISPC runtime assigns tasks to worker threads



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

Mapping


Problem to solve
Decomposition
Subproblems (a.k.a. "tasks", "work to do")
Assignment
Parallel Threads ("workers")
Orchestration
Parallel program (communicating threads)
Mapping
Execution on parallel machine

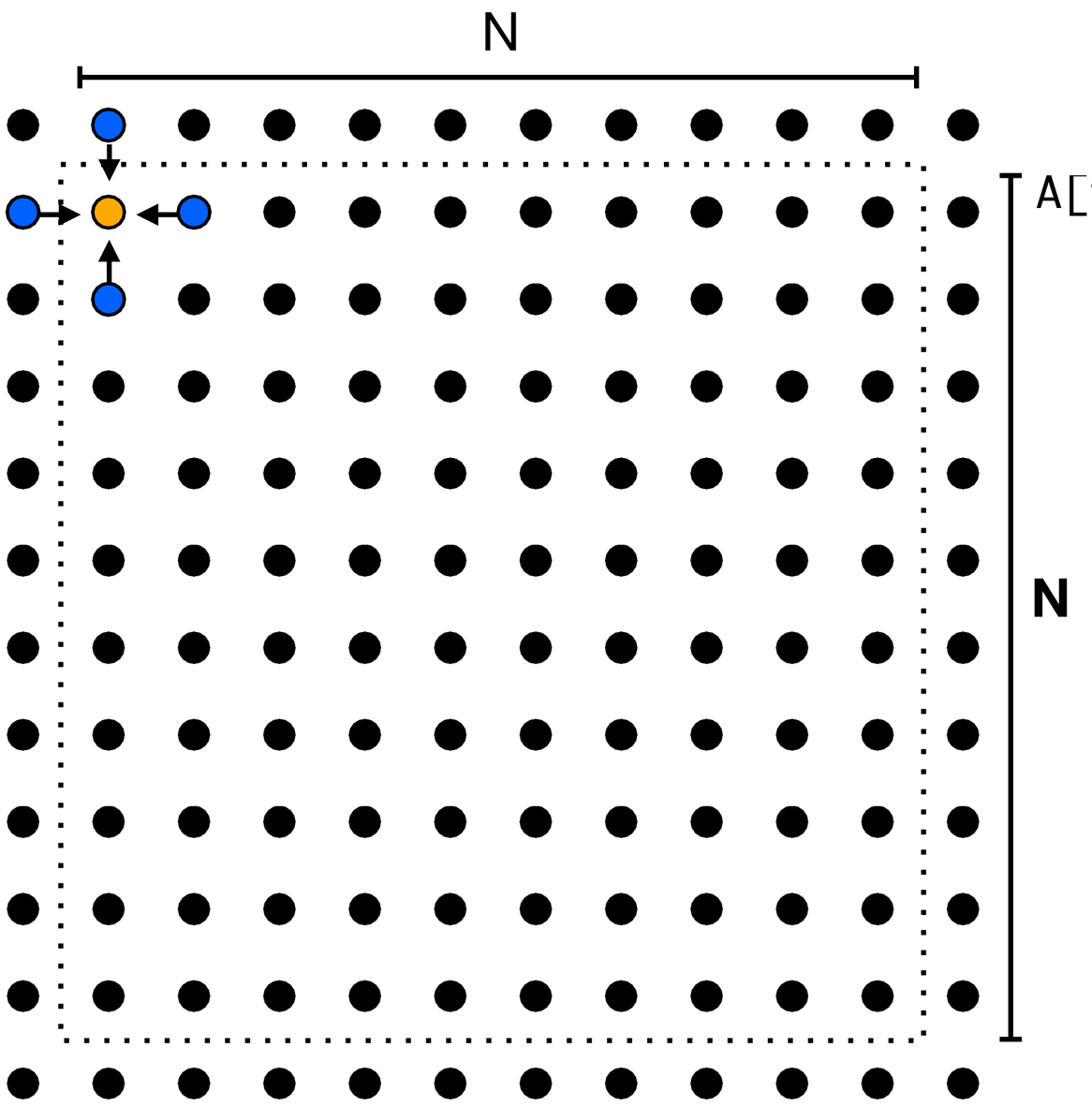
Mapping to hardware

- Mapping “threads” (“workers”) to hardware 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 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

A parallel programming example

A2D-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 example from: Culler, Singh, and Gupta

Grid solver algorithm

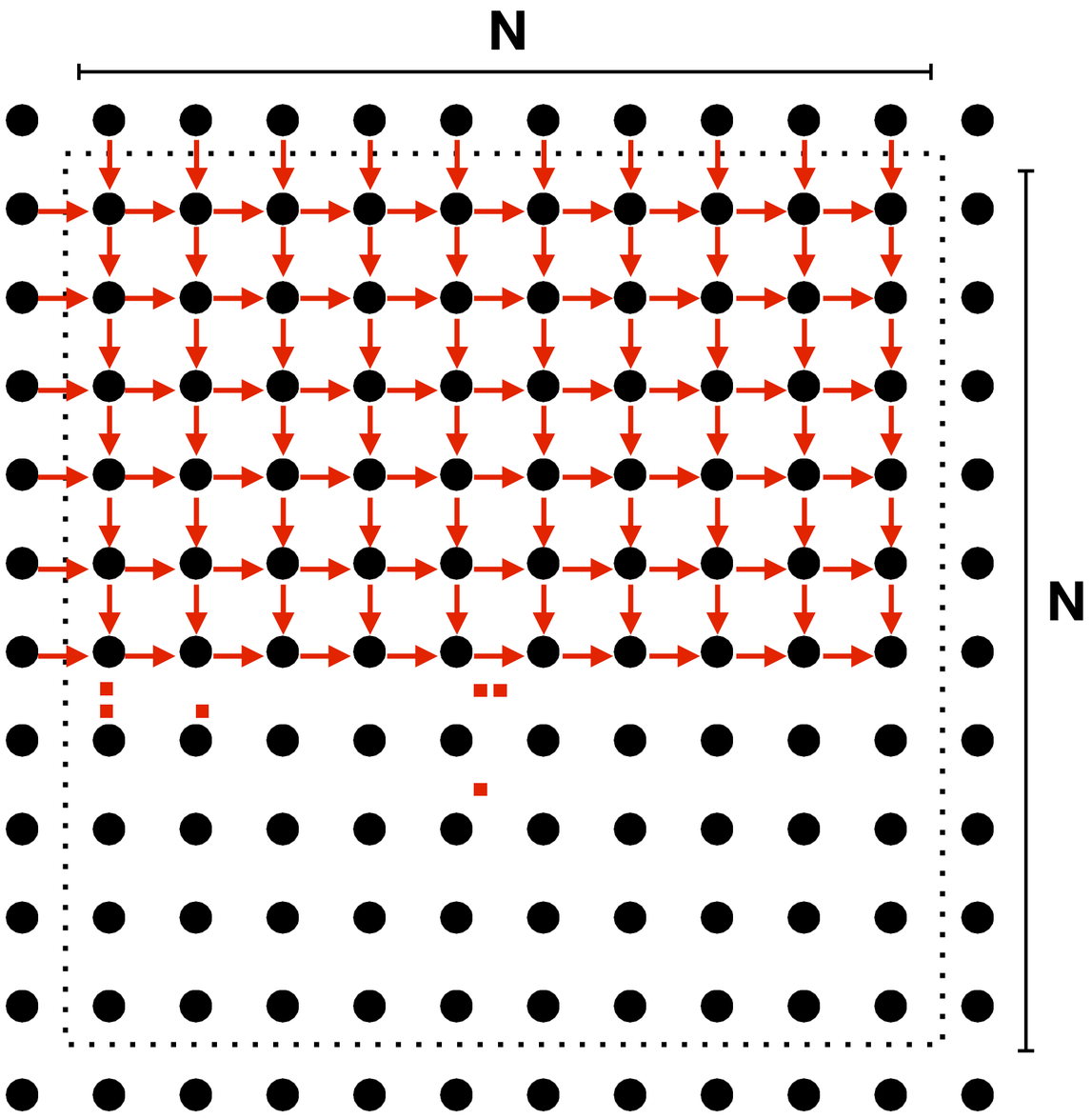
C-like pseudocode for sequential algorithm is provided below

```
const int
n;  float*                    // assume allocated for grid of N+2 x N+2
A;                            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) <                    // quit if
      TOLERANCE)  done =                converged
  }   true;
}
```

Step 1: identify dependencies (problem decomposition phase)

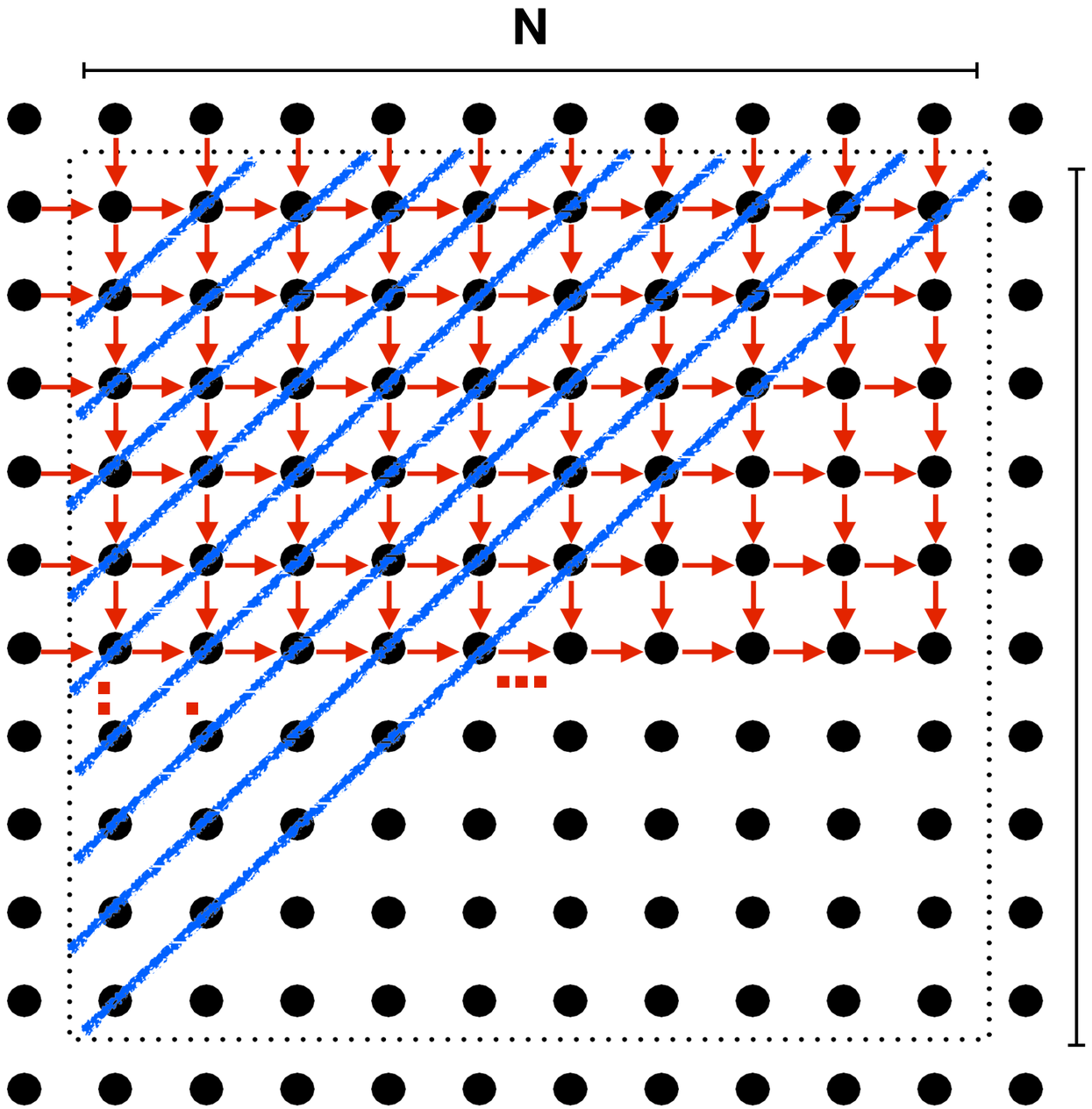


Each row element depends on element to left.

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

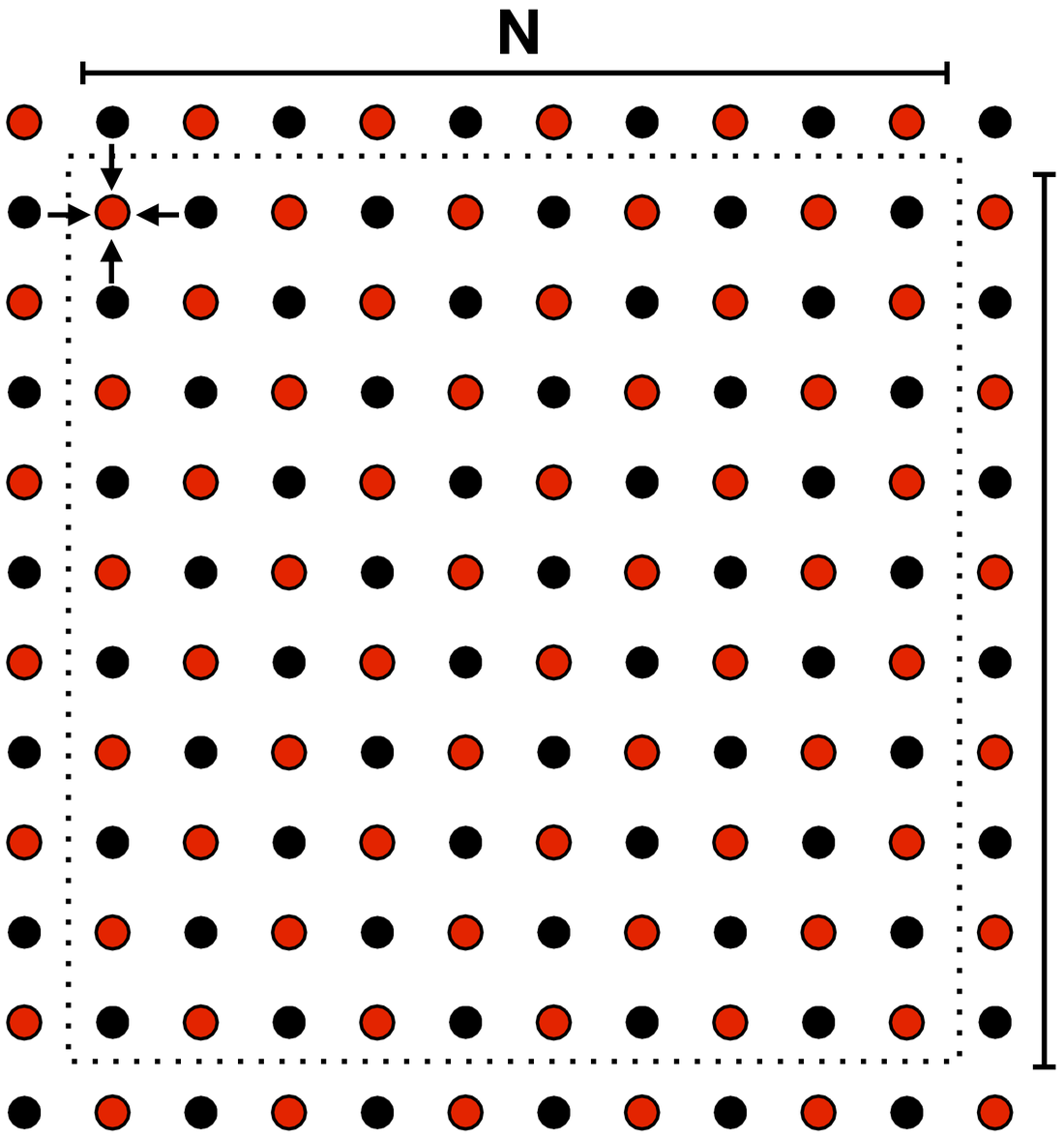
Not much parallelism at beginning and end of computation.

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 connverges to within 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

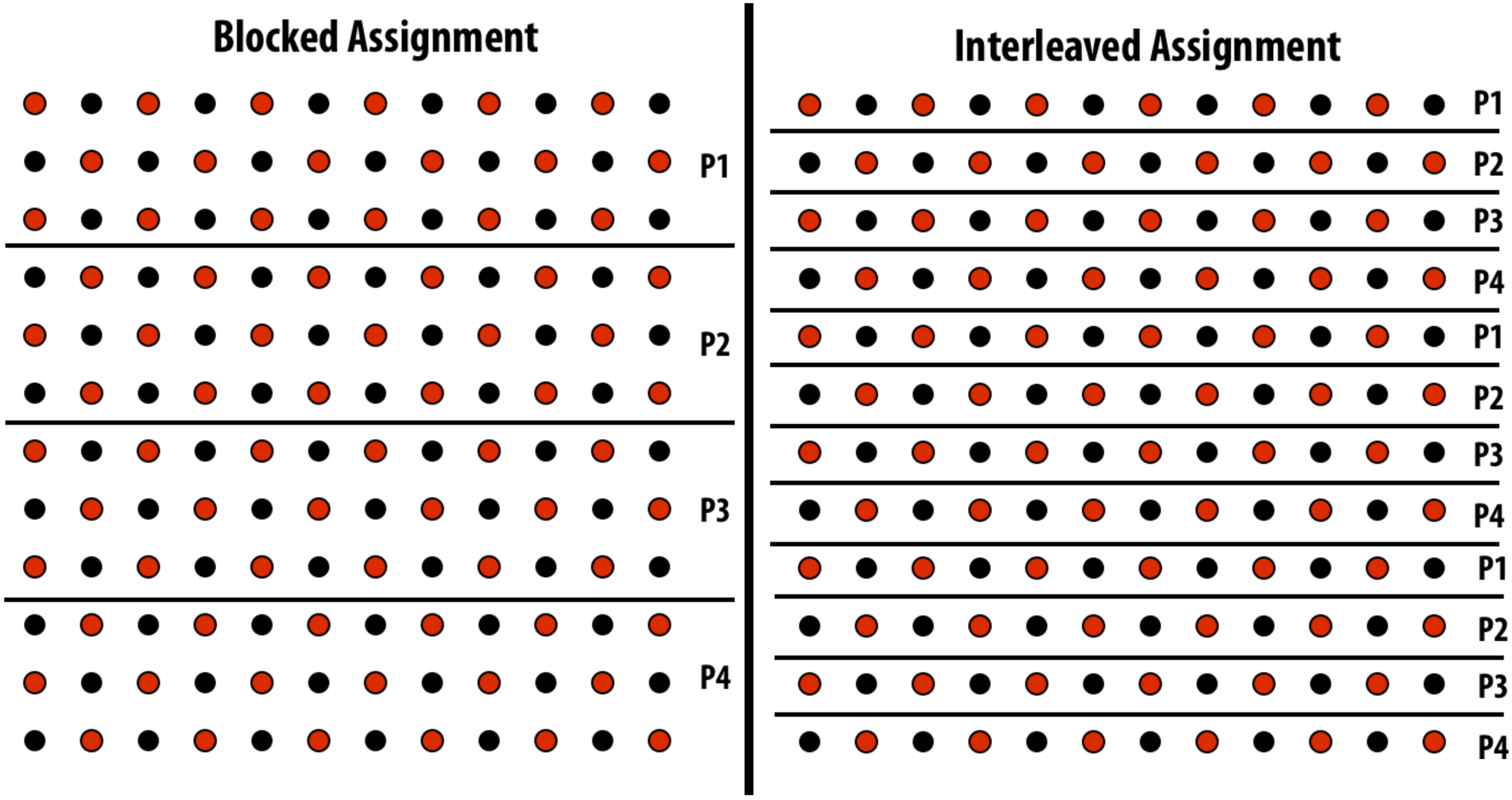


Update all red cells in parallel

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

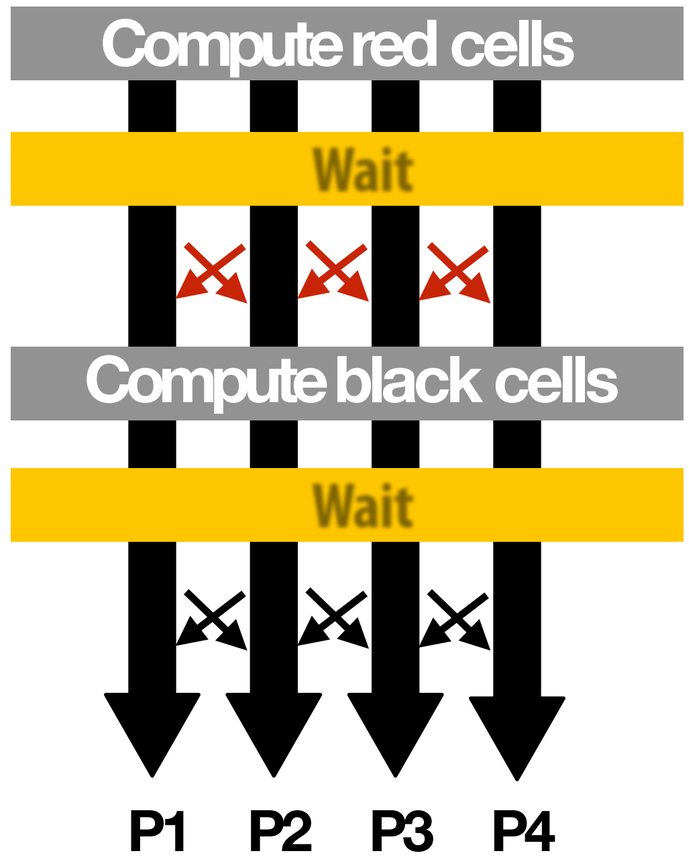


Question: Which is better? Does it matter?

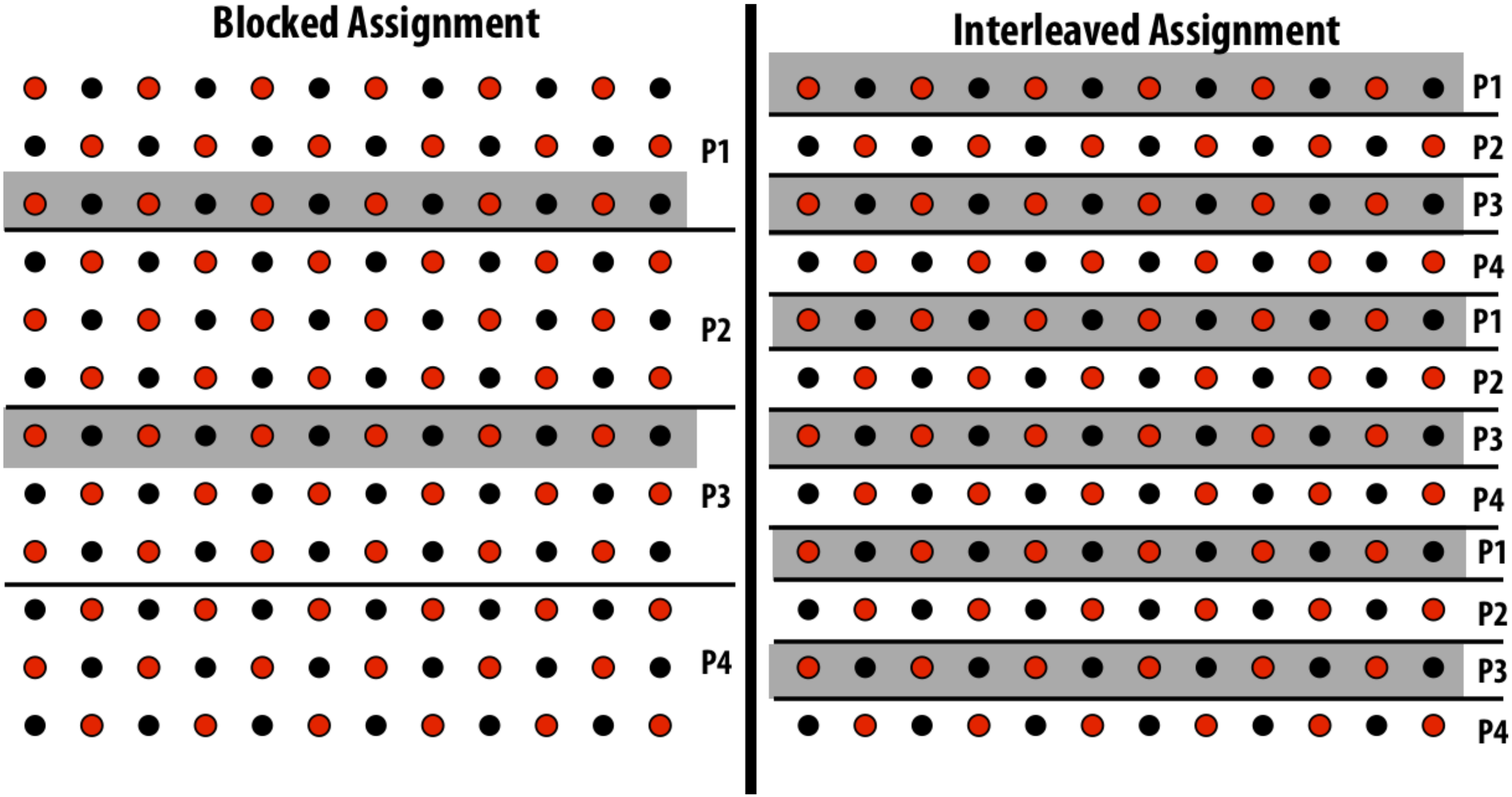
Answer: it depends on the system this program is running on

Consider dependencies (data flow)

1. Perform red 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



= data that must be sent to P2 each iteration

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];
          reduceAdd(diff, abs(A[i,j] - prev));
      }

      if (diff/(n*n) < TOLERANCE)
          done = true;
    }
}
```

Assignment: ???

Decomposition: updating individual grid elements constitute

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

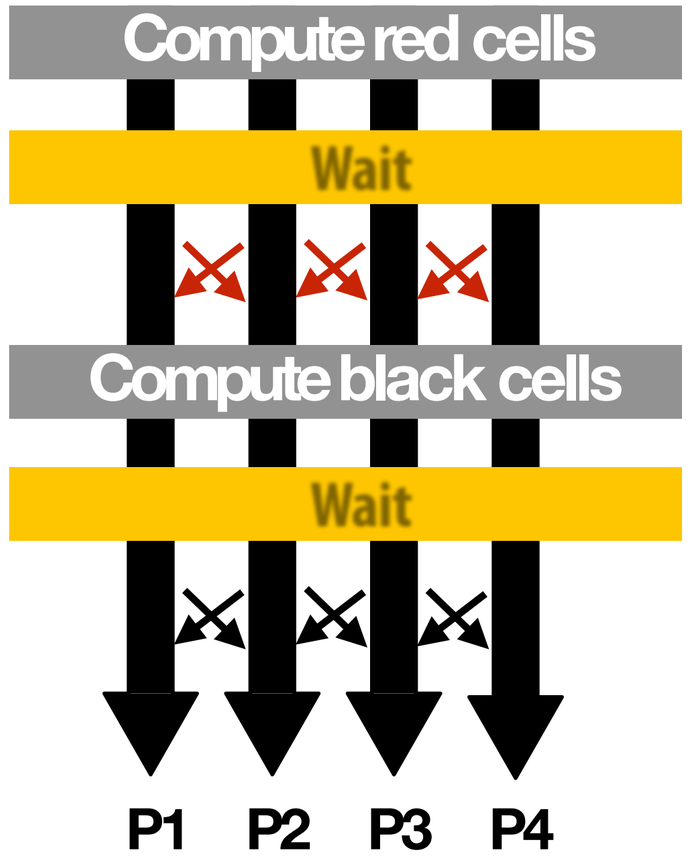
Shared address space expression of solver

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) {

   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

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) {

   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);
   }
```

Improve performance by accumulating into partial sum locally, then complete global reduction at the end of the iteration.

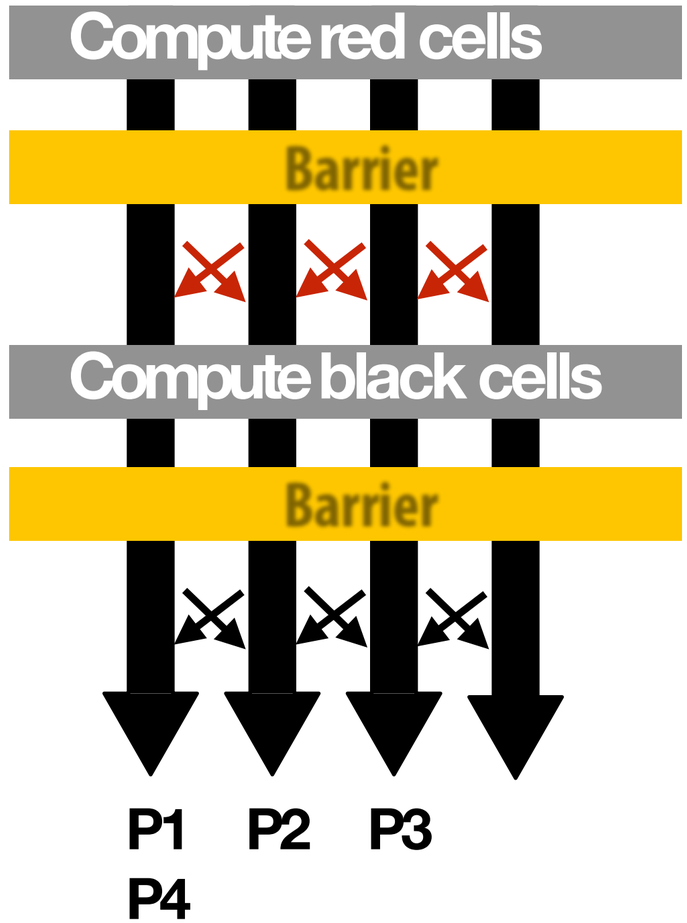
Compute partial sum per worker

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

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) {

   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);
   }

}
```

Why are there three barriers?

Remove barriers and tradeoff space

```
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;
}
```

variables in successive loop iterations

Solver implementation in two programming models

Data-parallel programming model

- Synchronization:
  - Single logical thread of control, but iterations of `forall` loop may 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
- Communication
  - Implicit in loads/stores to shared variables

Summary

- 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)