

**Lecture 4:**

# **Parallel Programming Basics**

---

**Parallel Computer Architecture and Programming**  
**CMU 15-418/15-618, Fall 2019**

# Review: 3 parallel programming models

## Shared address space

- Communication is unstructured, implicit in loads and stores
- Natural way of programming, but can shoot yourself in the foot easily
  - Program might be correct, but not perform well

## Message passing

- Structure all communication as messages
- Often harder to get first correct program than shared address space
- Structure often helpful in getting to first correct, scalable program

## Data parallel

- Structure computation as a big “map” over a collection
- Assumes a shared address space from which to load inputs/store results, but model severely limits communication between iterations of the map (goal: preserve independent processing of iterations)
- Modern embodiments encourage, but don’t enforce, this structure

# Modern practice: mixed programming

Use **shared address space** programming **within a multi-core node** of a cluster, use **message passing** **between nodes**

- Very, very common in practice
- Use convenience of shared address space where it can be implemented efficiently (within a node), require explicit communication elsewhere

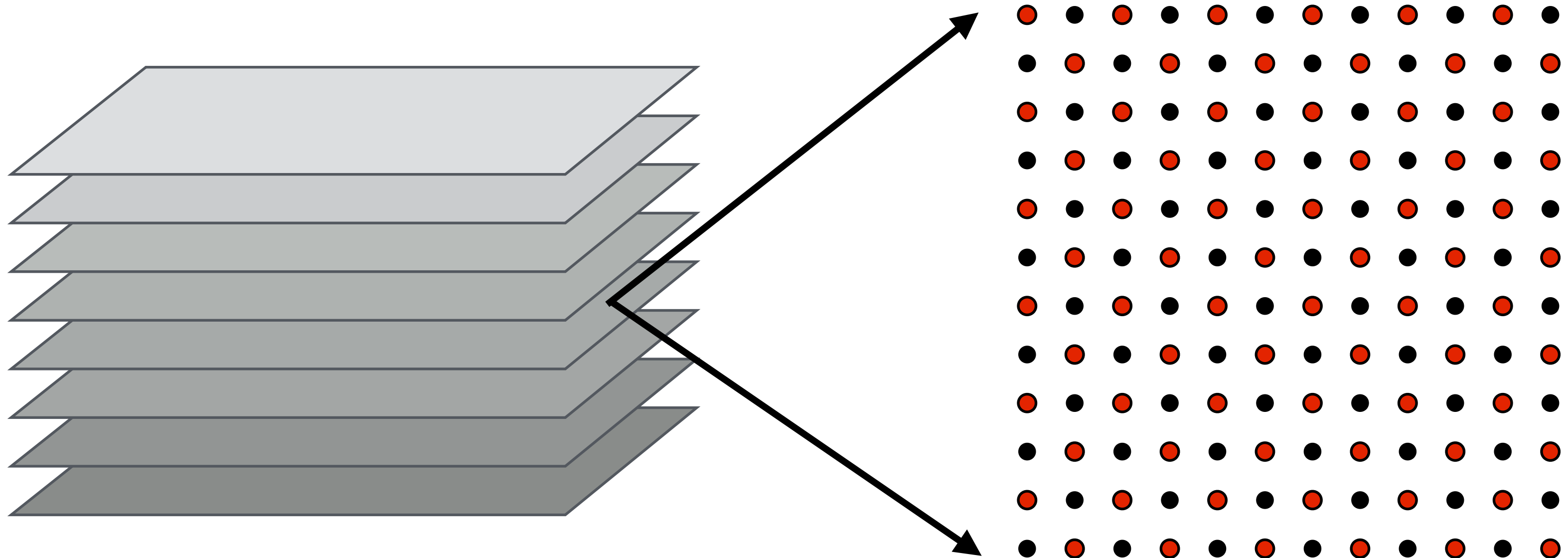
**Data-parallel-ish programming models support shared-memory style synchronization primitives in kernels**

- Permit limited forms of inter-iteration communication (e.g., CUDA, OpenCL)

**In a future lecture... CUDA/OpenCL use data-parallel model to scale to many cores, but adopt shared-address space model allowing threads running on the same core to communicate.**

# Examples of applications to parallelize

# Simulating of ocean currents



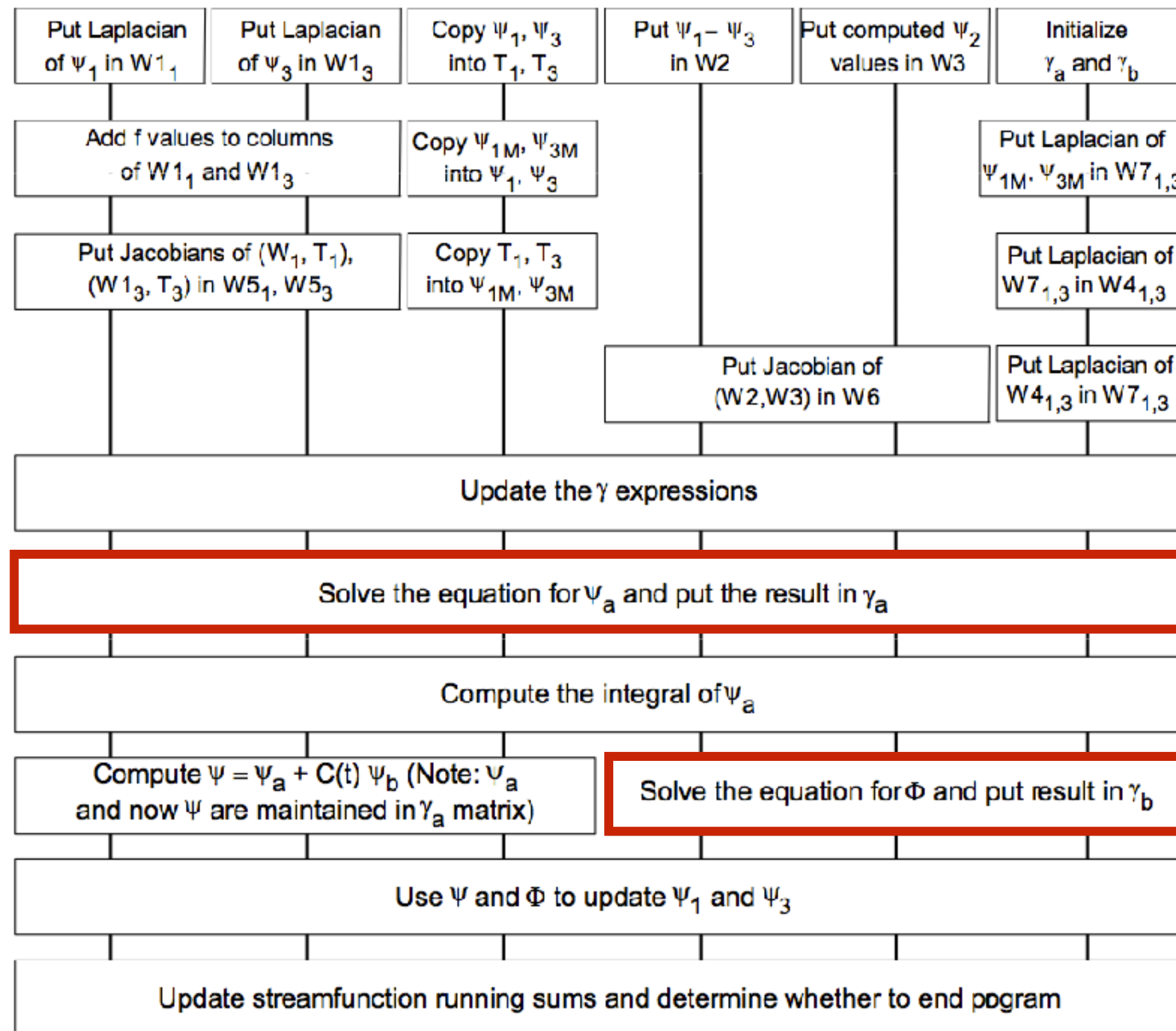
**Discretize 3D ocean volume into slices represented as 2D grids**

**Discretize time evolution of ocean:  $\Delta t$**

**High accuracy simulation requires small  $\Delta t$  and high resolution grids**

# Where are the dependencies?

## Dependencies in one time step of ocean simulation



Boxes correspond to computations on grids

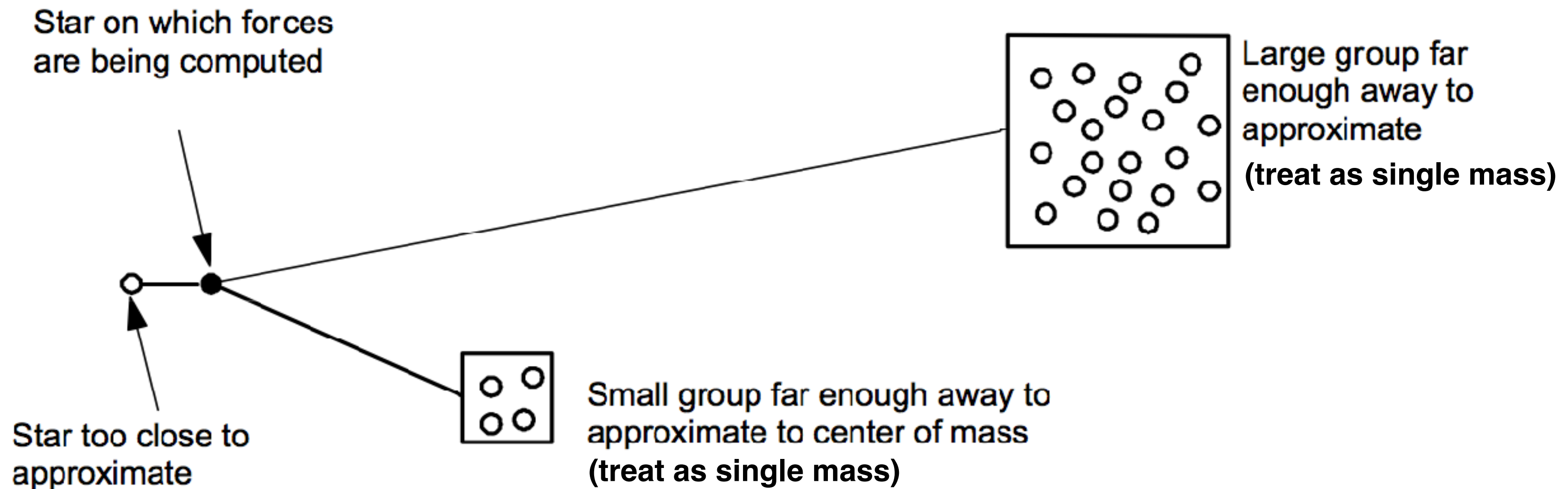
Lines express dependencies between computations on grids

The "grid solver" example corresponds to these parts of the application

**Parallelism within a grid (data-parallelism) and across operations on the different grids. The implementation only leverages data-parallelism (for simplicity)**

# Galaxy evolution

## Barnes-Hut algorithm

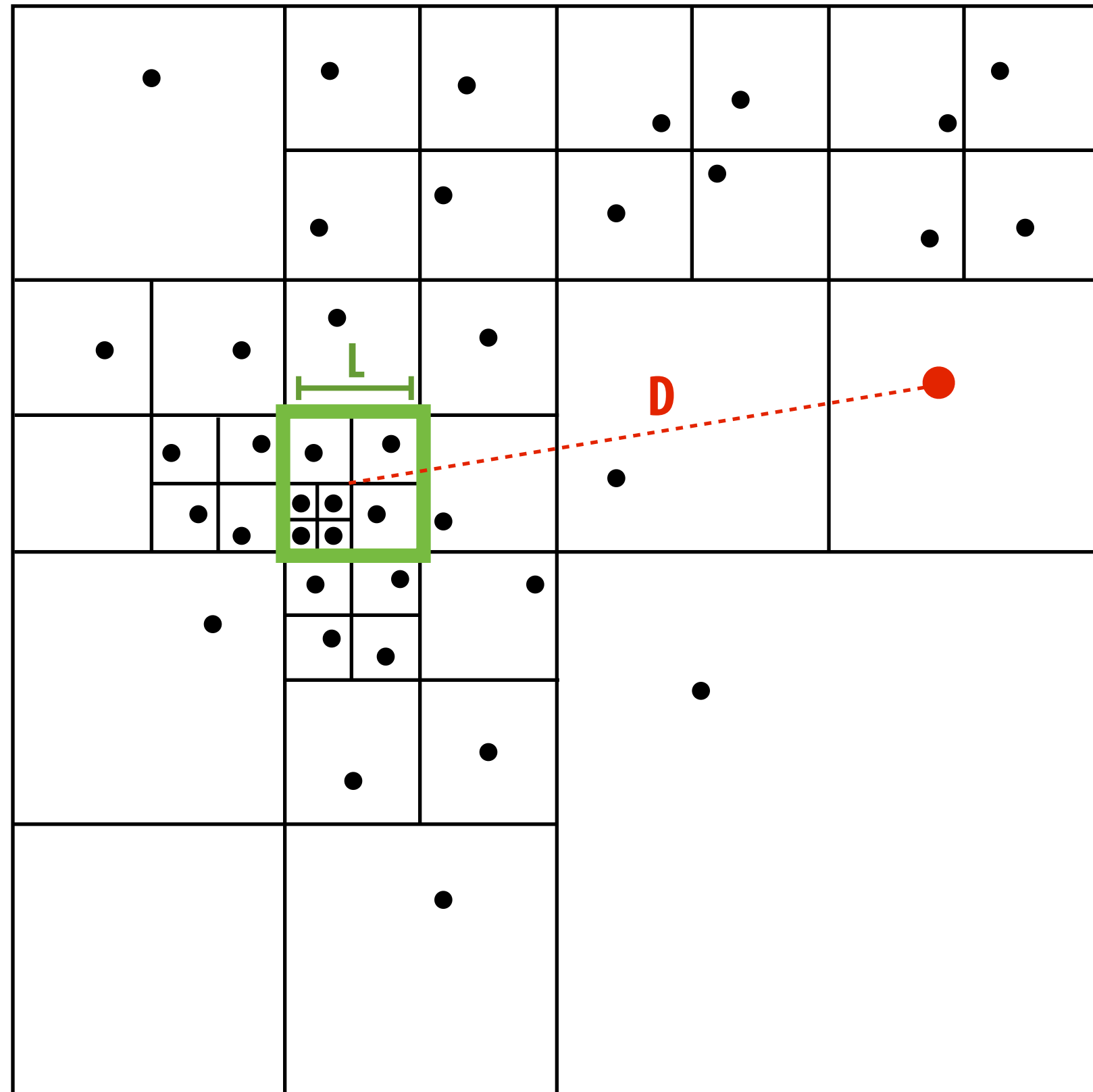


**Represent galaxy as a collection of  $N$  particles (think: particle = star)**

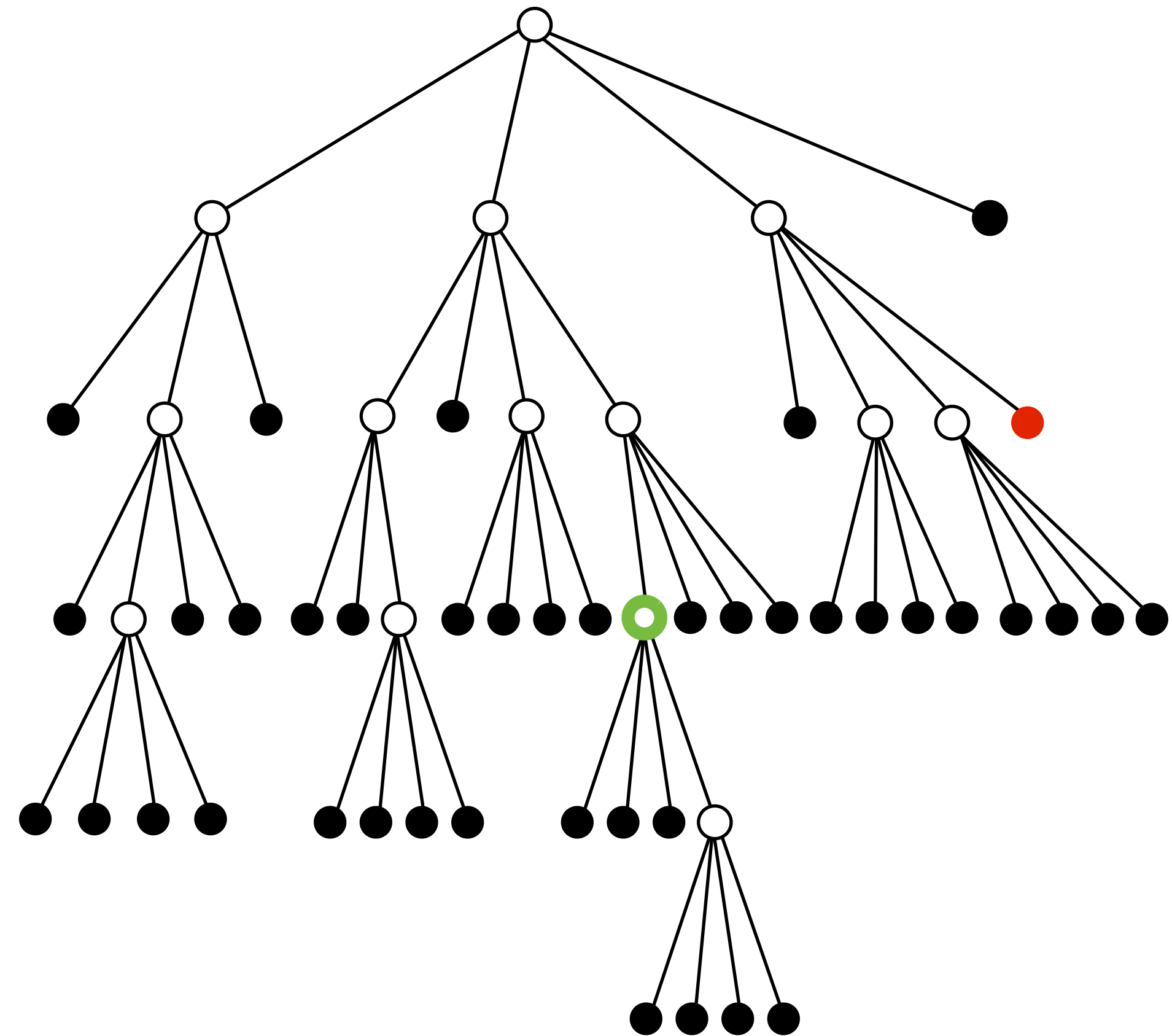
**Compute forces on each particle due to gravity**

- Naive algorithm is  $O(N^2)$  — all particles interact with all others (gravity has infinite extent)
- Magnitude of gravitational force falls off with distance (so algorithms approximate forces from far away stars to gain performance)
- Result is an  $O(N \lg N)$  algorithm for computing gravitational forces between all stars

# Barnes-Hut tree



Spatial Domain



Quad-Tree Representation of Bodies

**Leaf nodes are star particles**

**Interior nodes store center of mass + aggregate mass of all child bodies**

**To compute forces on each body, traverse tree... accumulating forces from all other bodies**

**– Compute forces using aggregate interior node if  $L/D < \Theta$ , else descend to children**

**Expected number of nodes touched  $\sim \lg N / \Theta^2$**



# 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

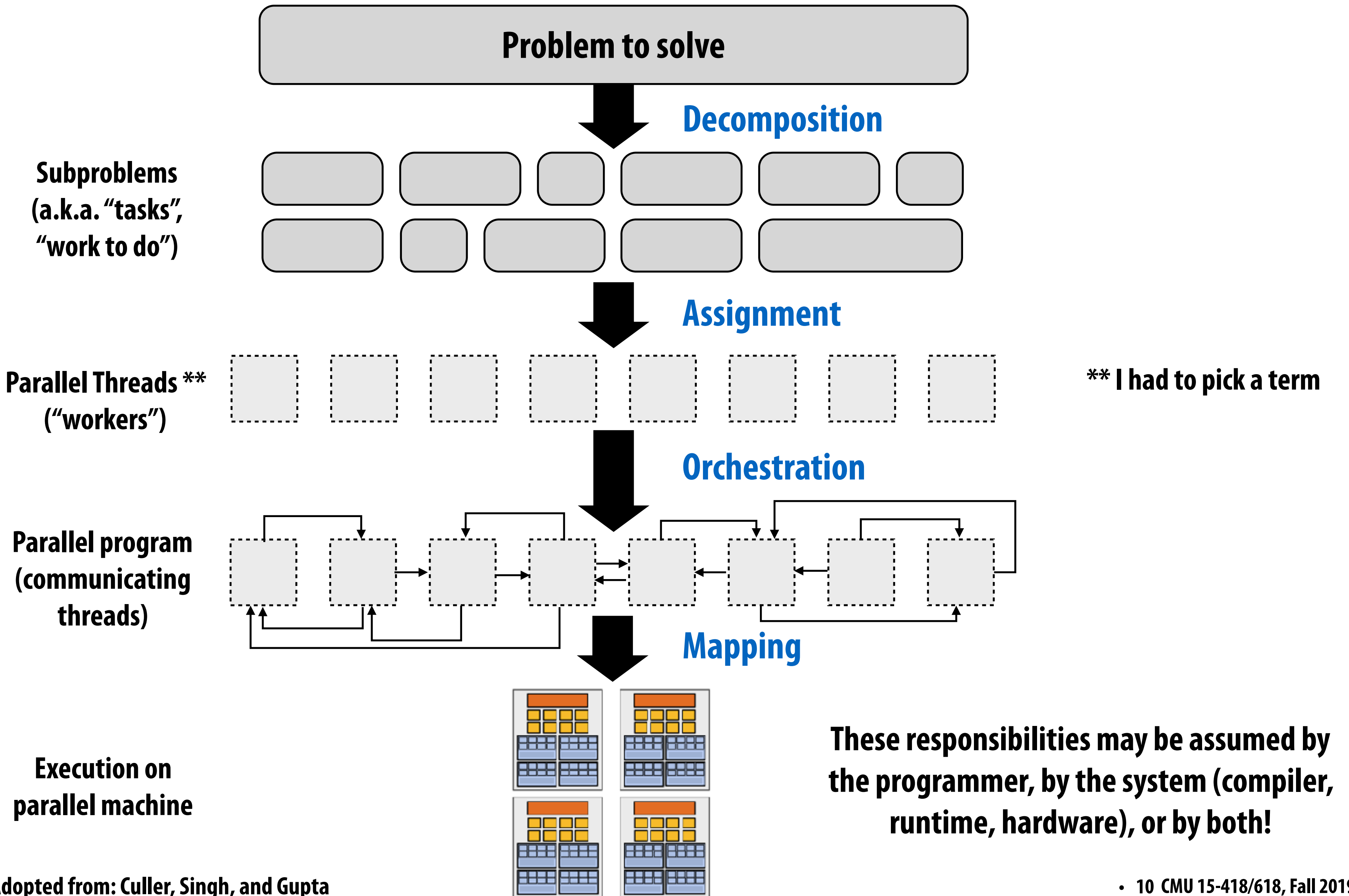
Recall one of our main goals is speedup \*

For a fixed computation:

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

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

# Creating a parallel program



# 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 *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  $\leq 1/S$

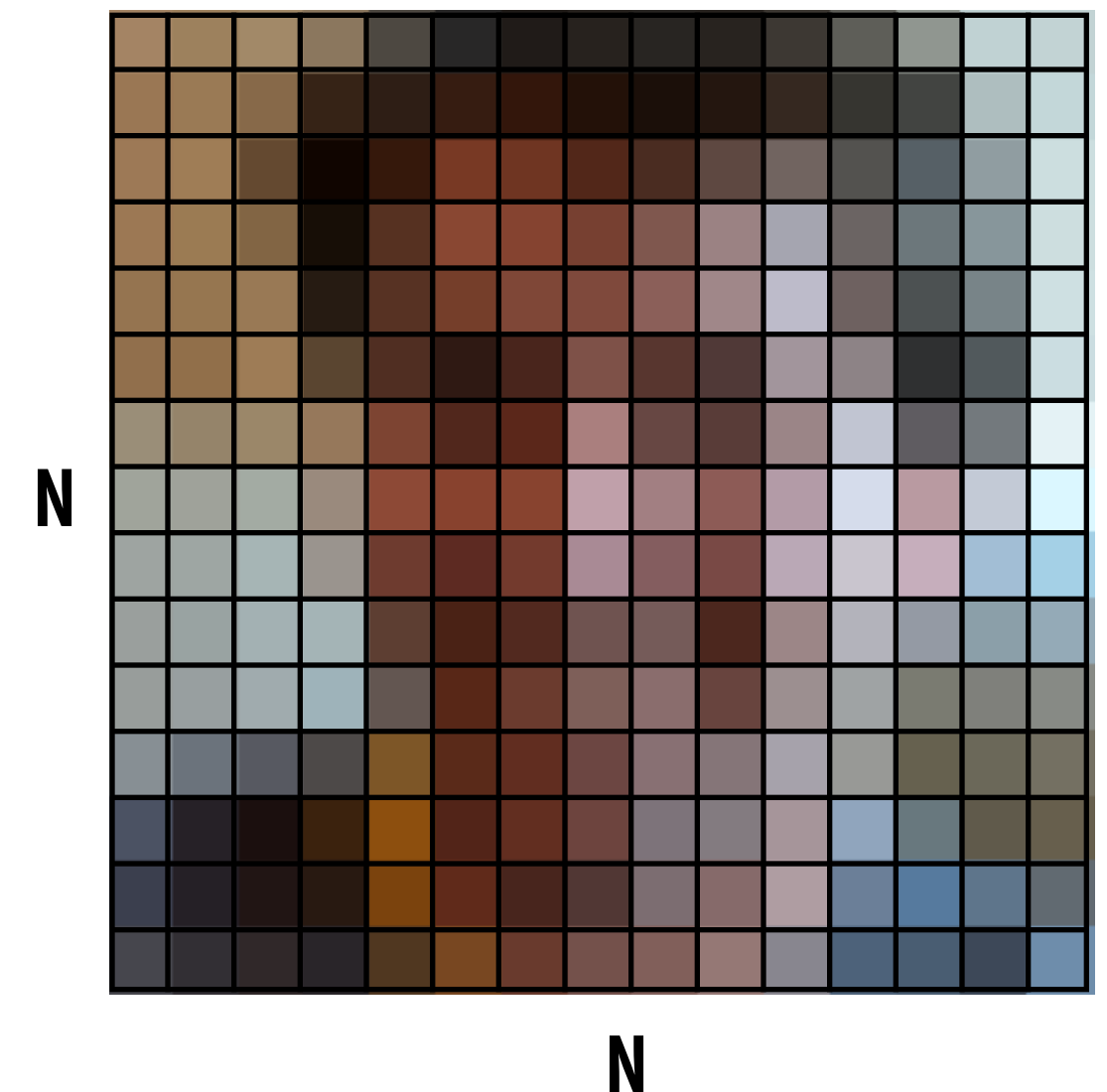
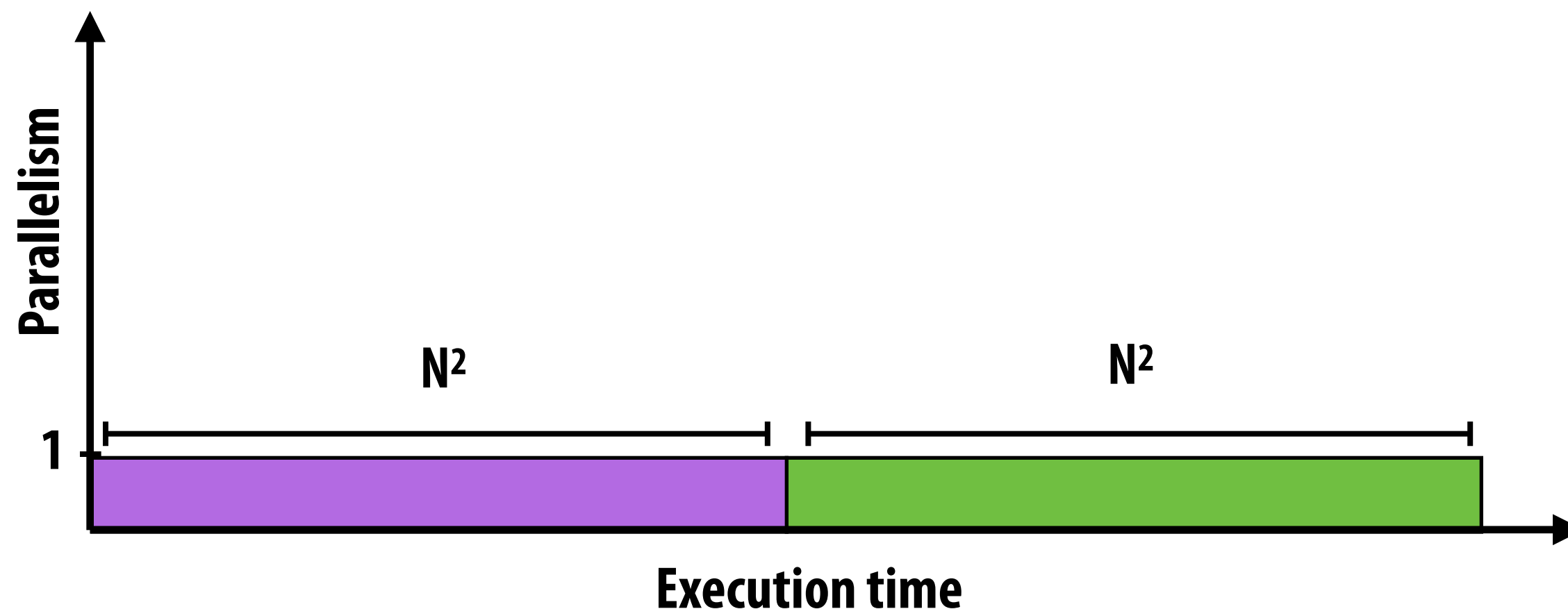
# A simple example

Consider a two-step computation on a  $N \times N$  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  $\sim N^2$  time, so total time is  $\sim 2N^2$



# First attempt at parallelism (P processors)

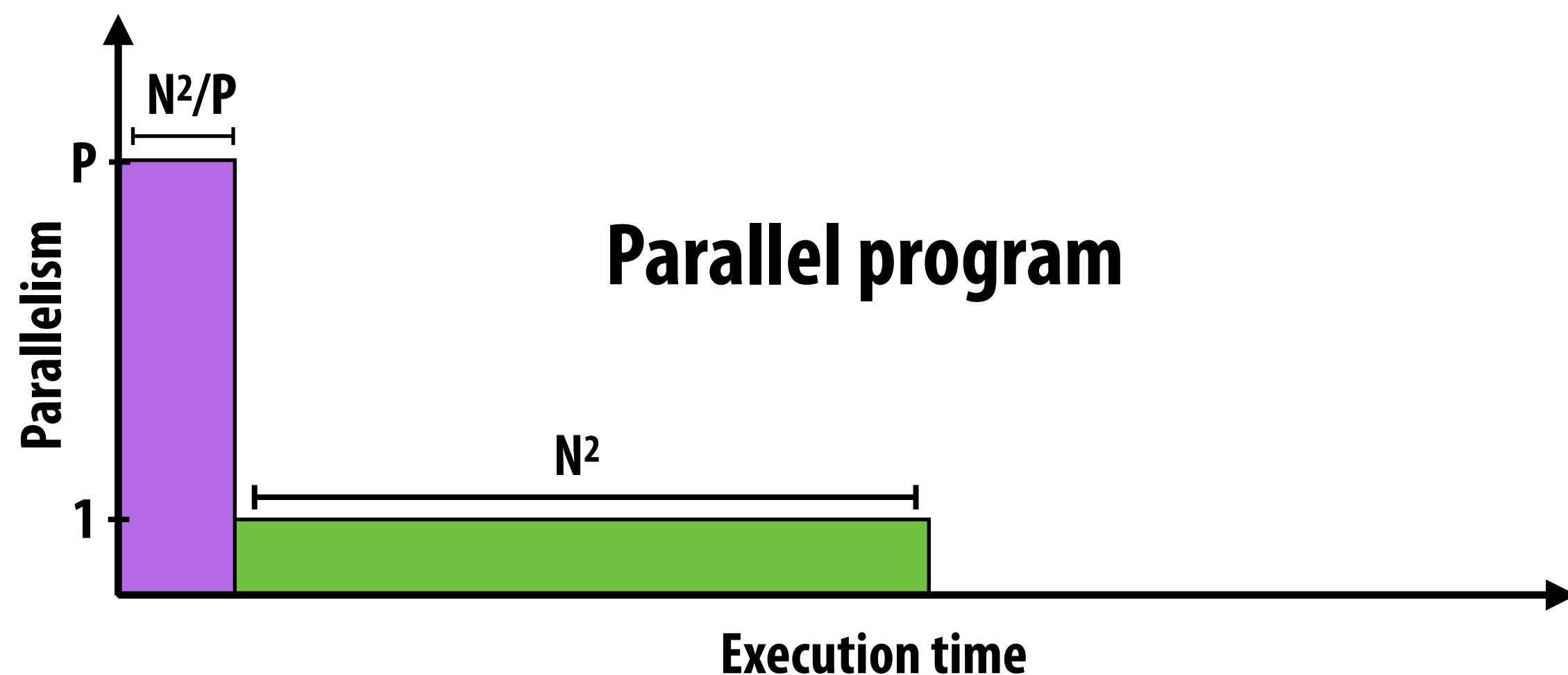
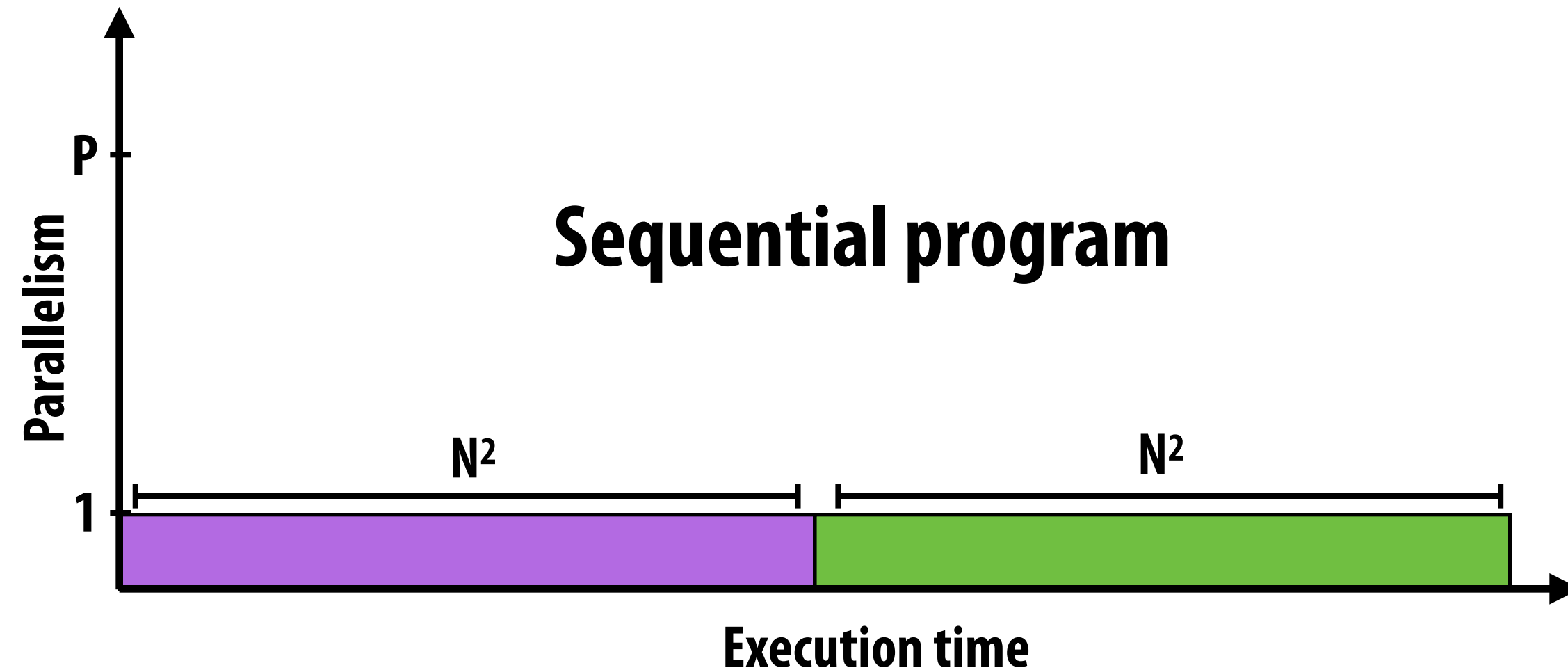
## Strategy:

- **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:

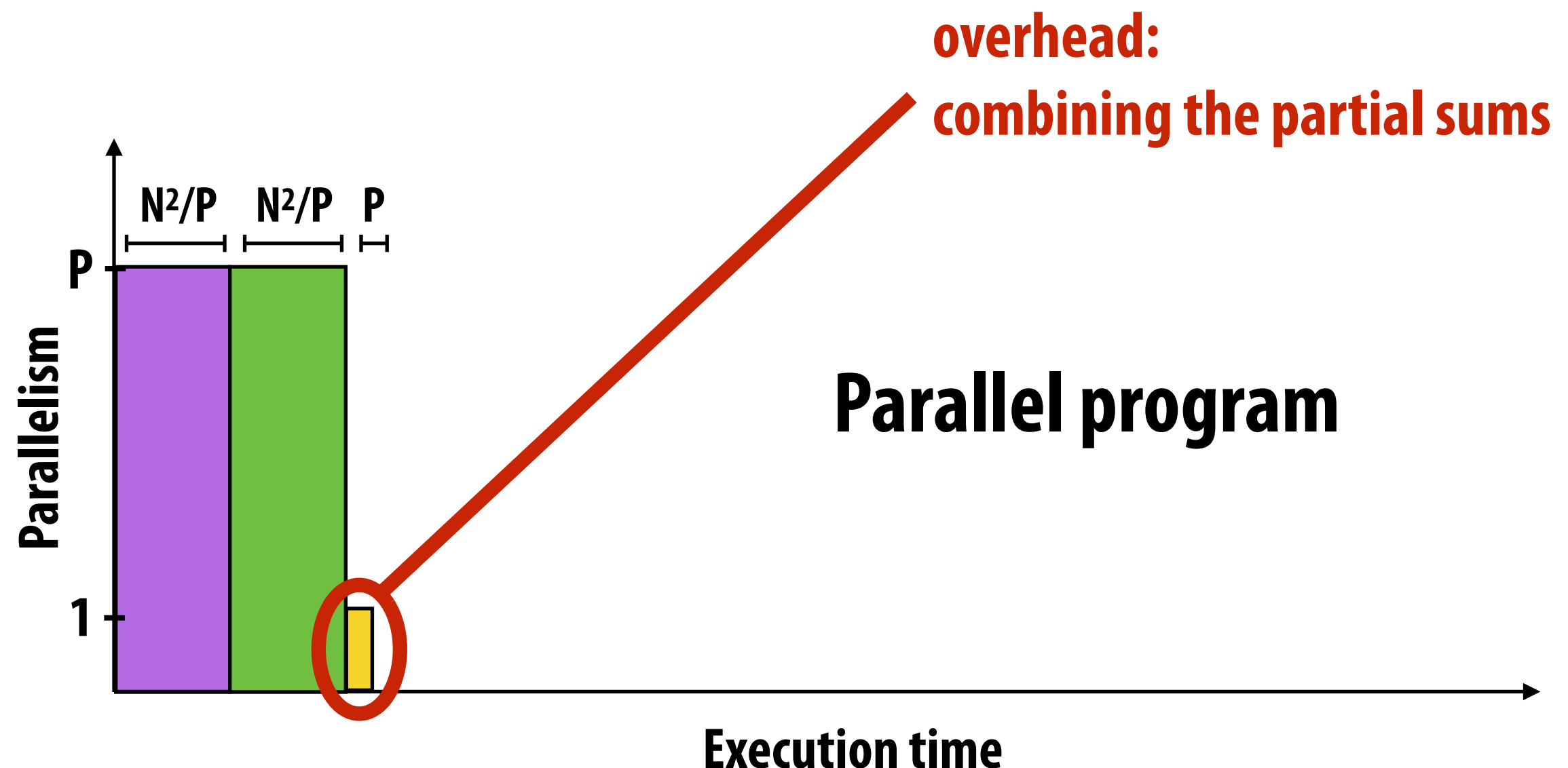
- **Step 1**: execute in parallel
  - time for phase 1:  $N^2/P$
- **Step 2**: compute **partial sums in parallel**, combine results serially
  - time for phase 2:  $N^2/P + P$

## Overall performance:

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

## Note:

speedup  $\rightarrow P$  when  $N \gg P$

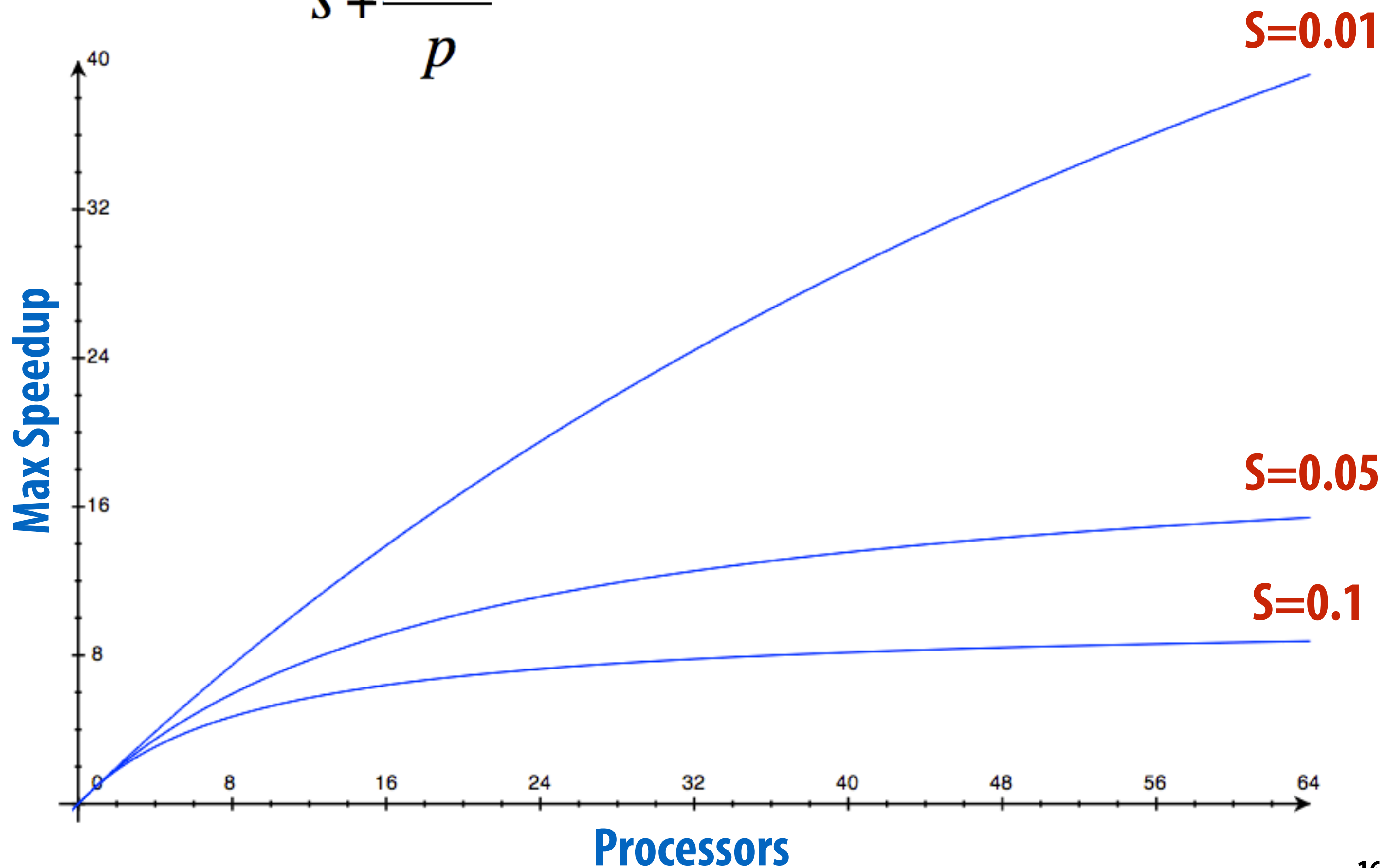


# Amdahl's law

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

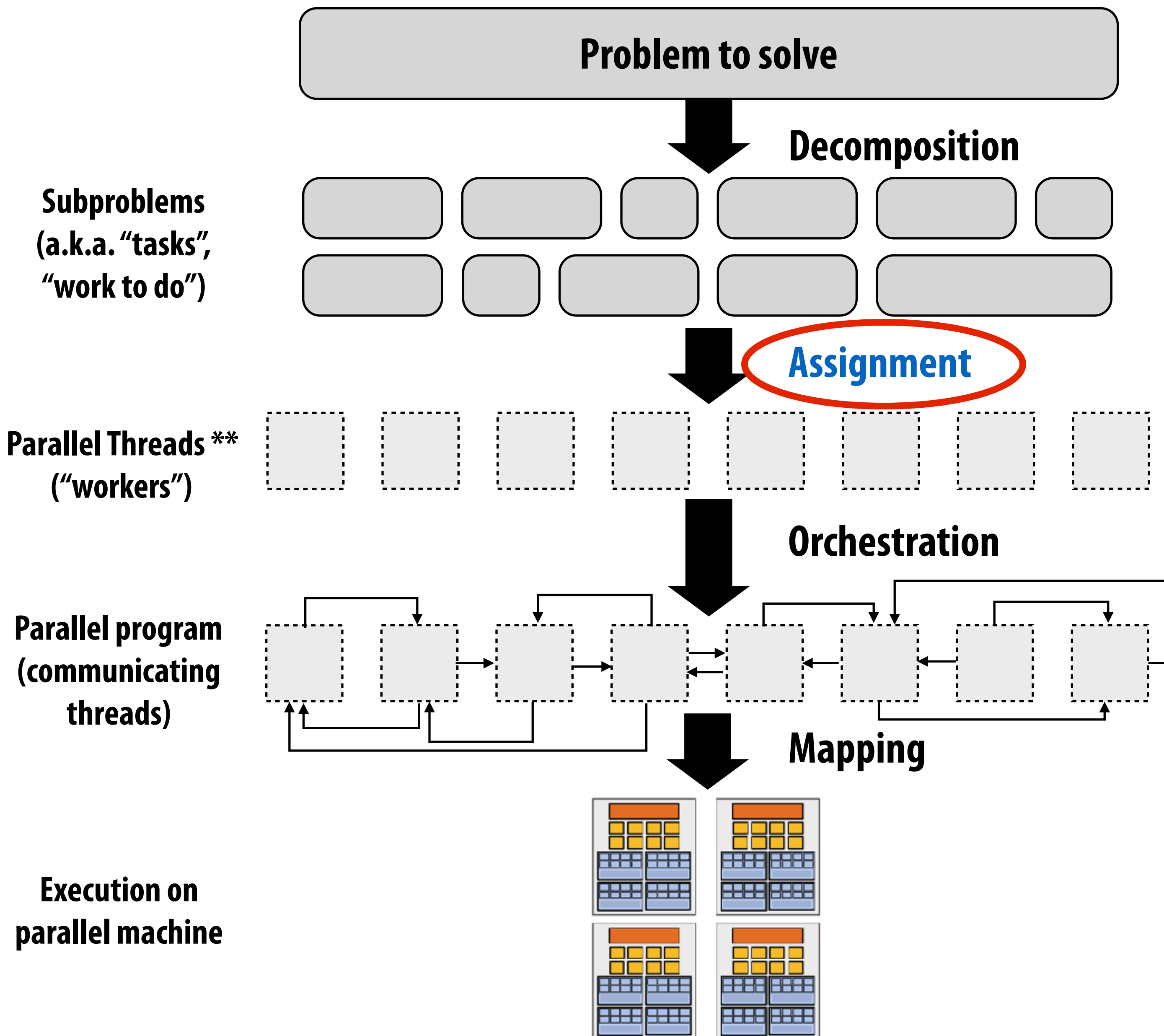
**Who is responsible for performing decomposition?**

- 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 \*\*

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

- Think of “tasks” as things to do
- Think of 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(thread_args->N, thread_args->terms, thread_args->x, thread_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);  
}
```

**ISPC runtime assign tasks to worker threads**

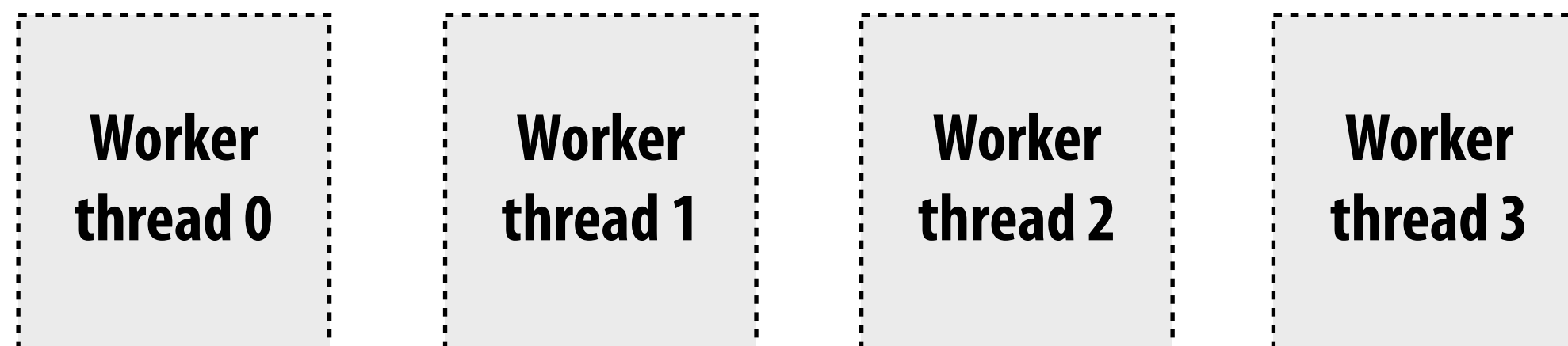
Next task ptr



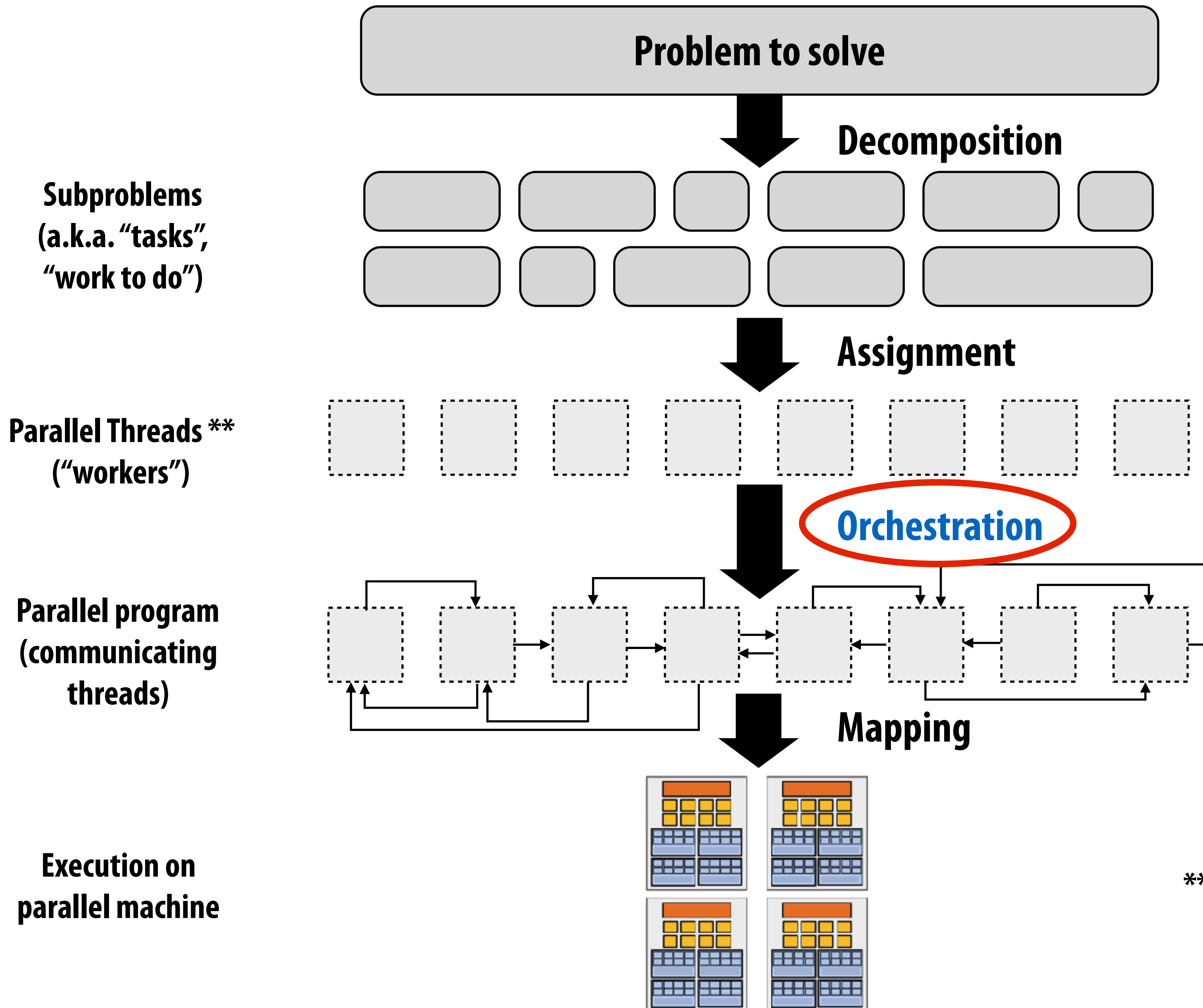
List of tasks:



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



# Orchestration



# Orchestration

## Involves:

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

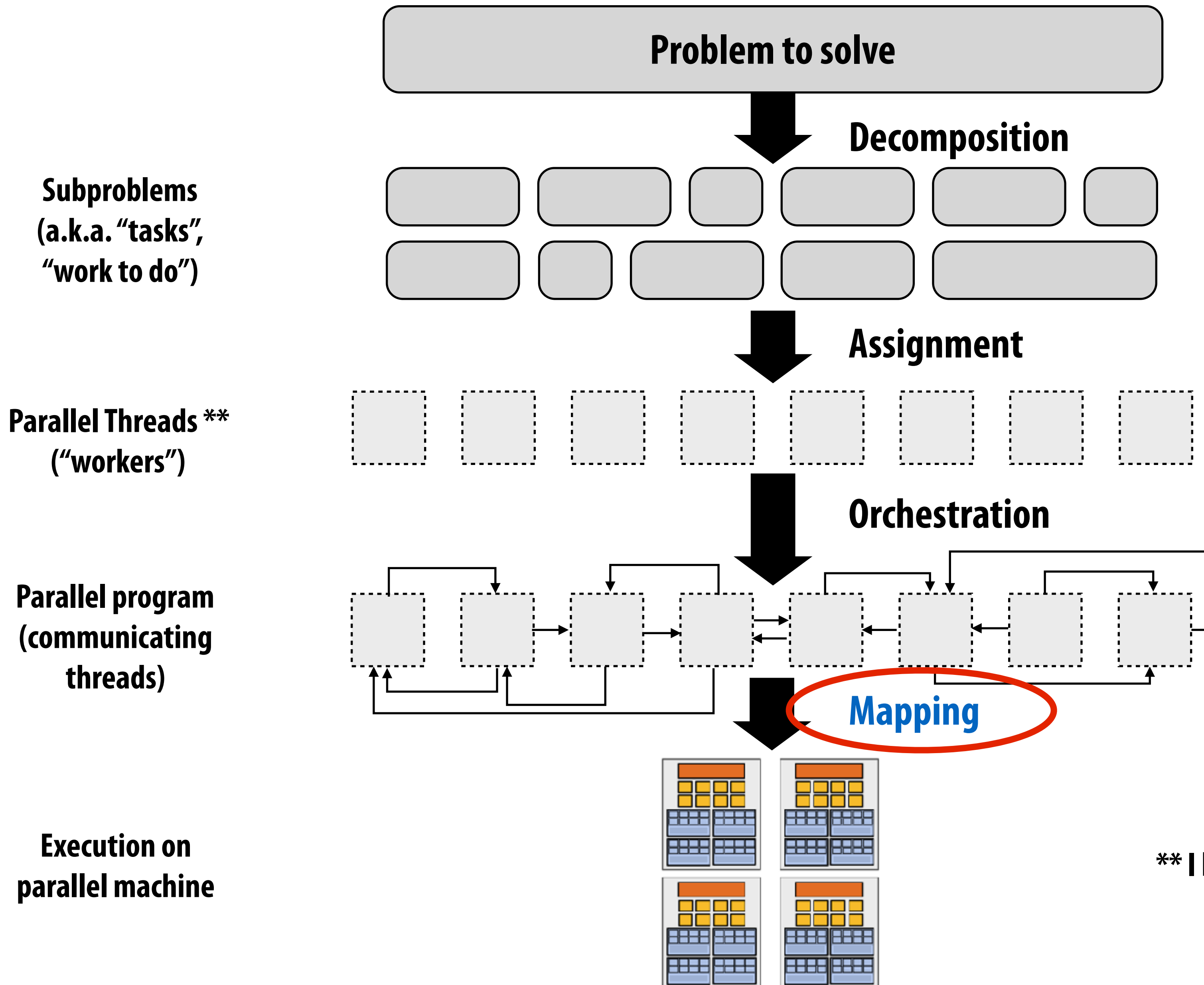
**Goals:** reduce **costs of communication/sync**, preserve **locality** of data reference, reduce **overhead**, etc.

## Machine details impact many of these decisions

- If synchronization is expensive, might use it more sparsely



# Mapping to hardware



# 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

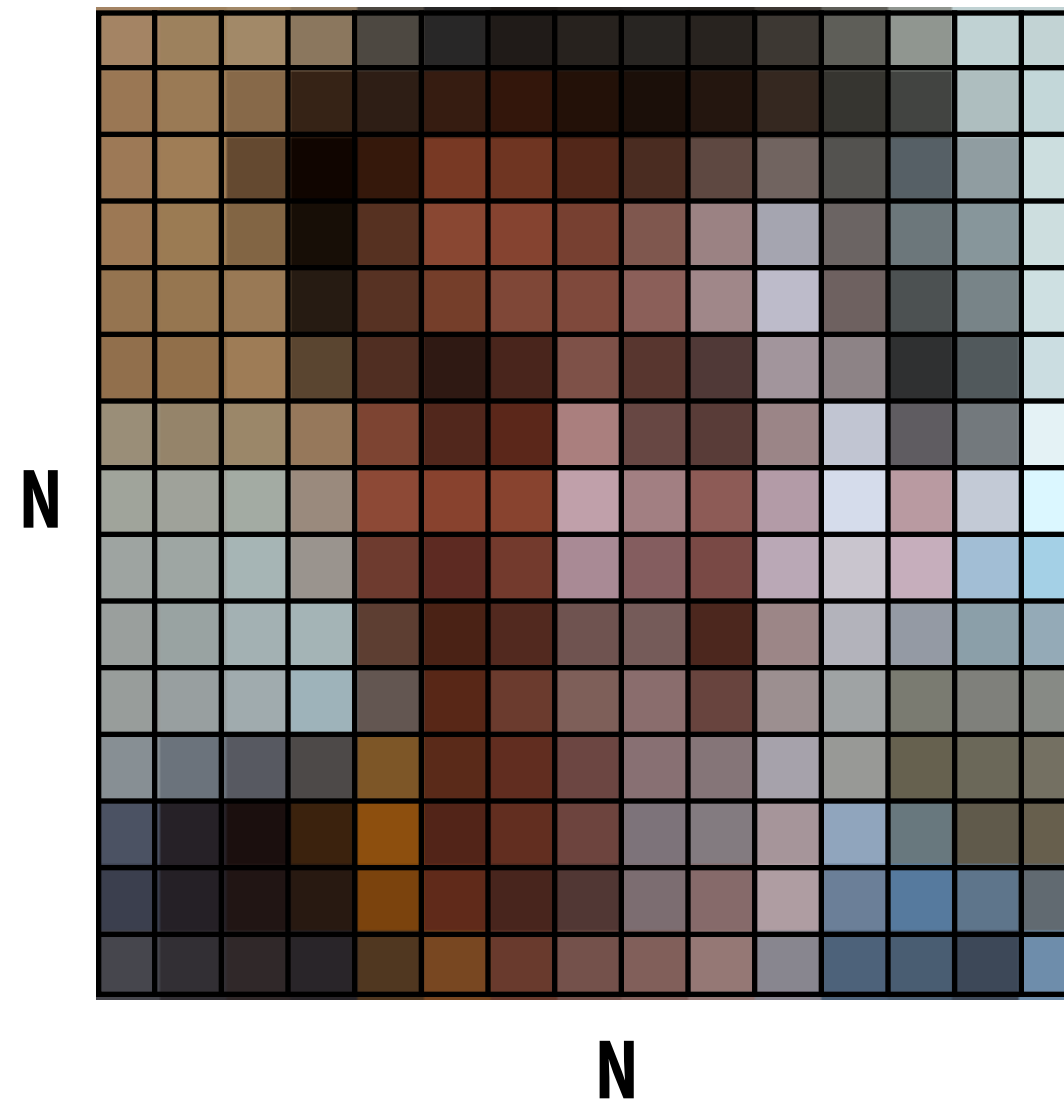
### Example 3: mapping by the **hardware**

- Map CUDA thread blocks to GPU cores (future lecture)

## Some interesting mapping decisions:

- Place **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.**

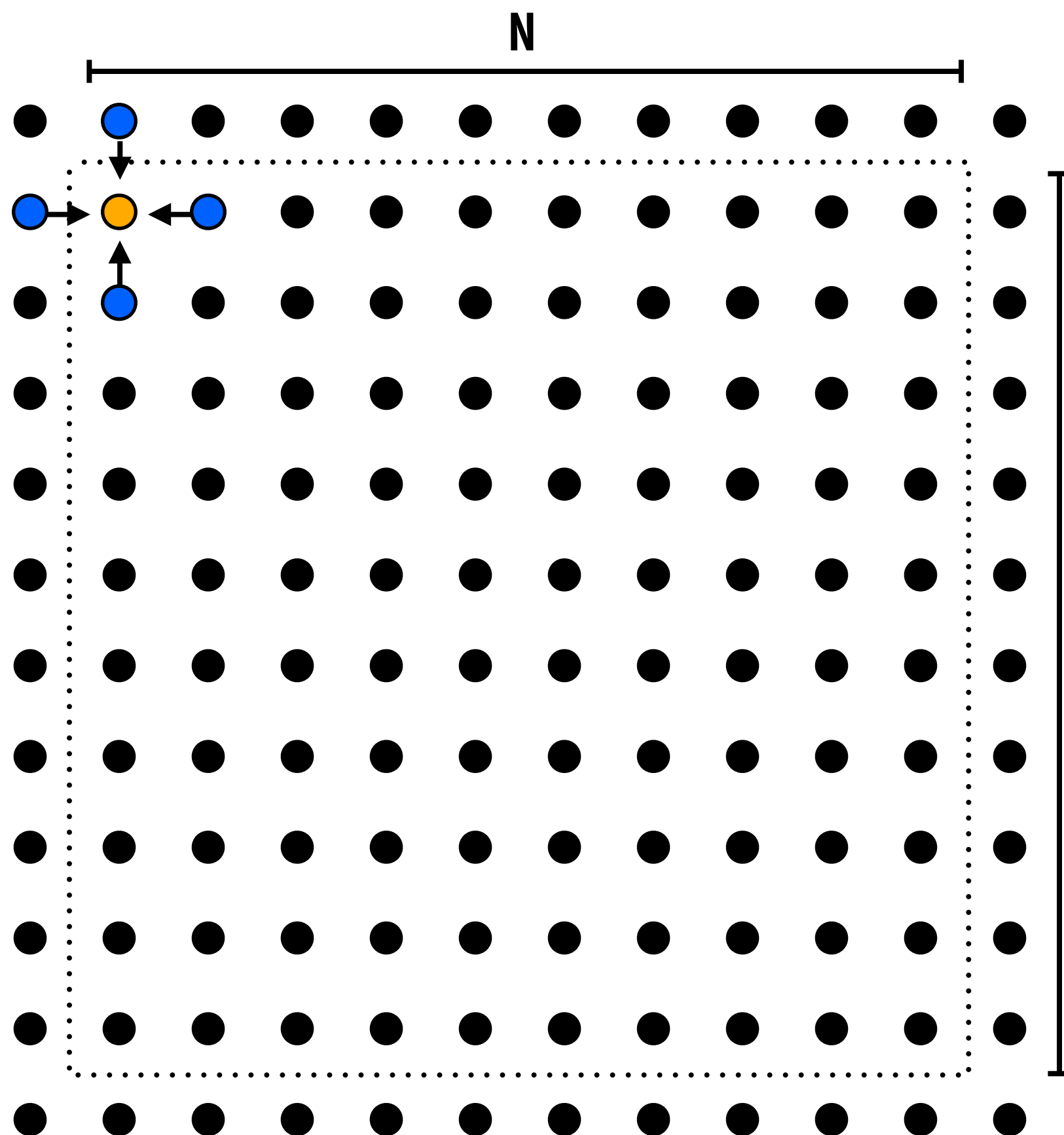
# **A parallel programming example**

# A 2D-grid based solver

Solve partial differential equation (PDE) on  $(N+2) \times (N+2)$  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

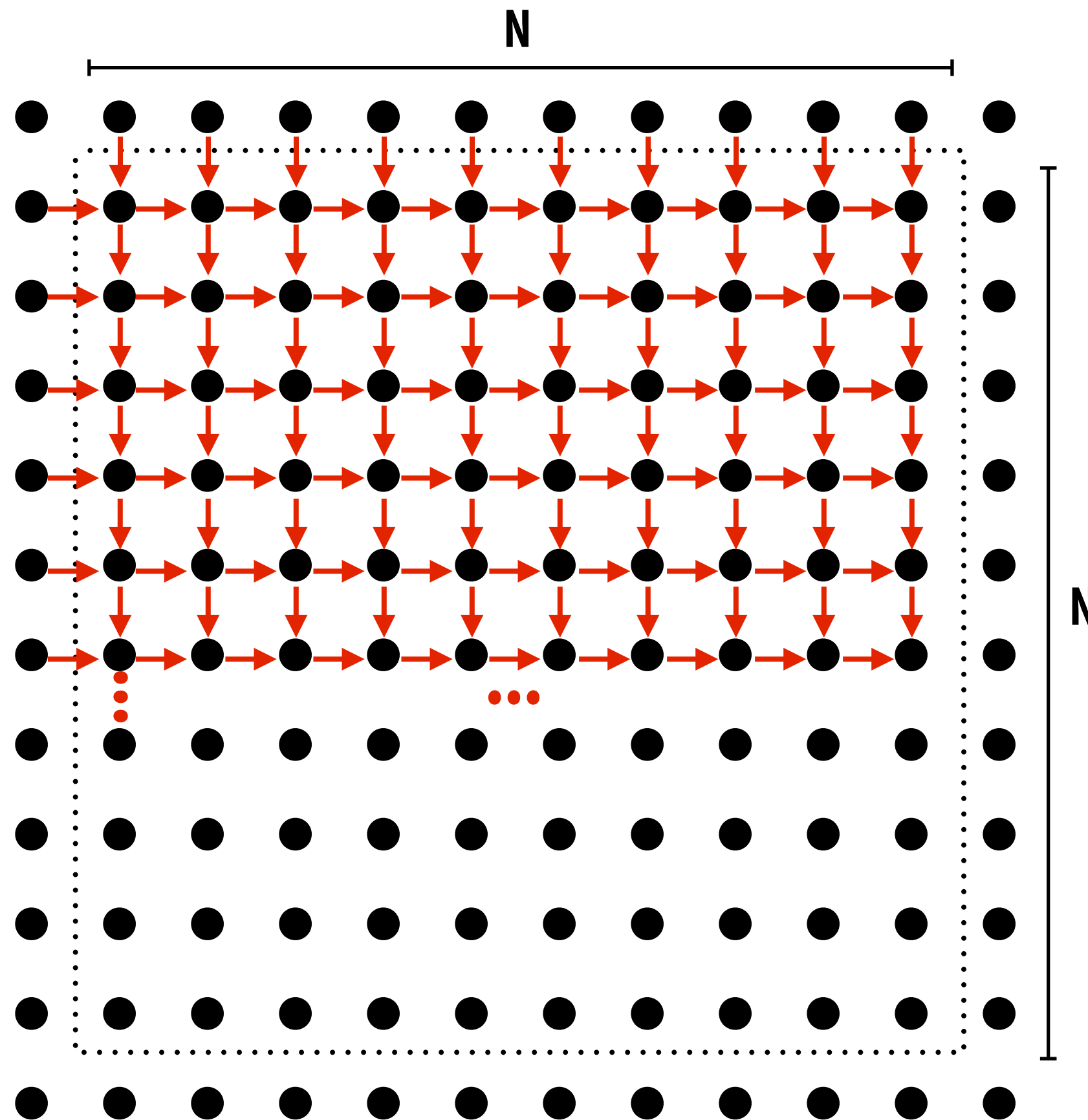
void solve(float* A) {

    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;
    }
}
```

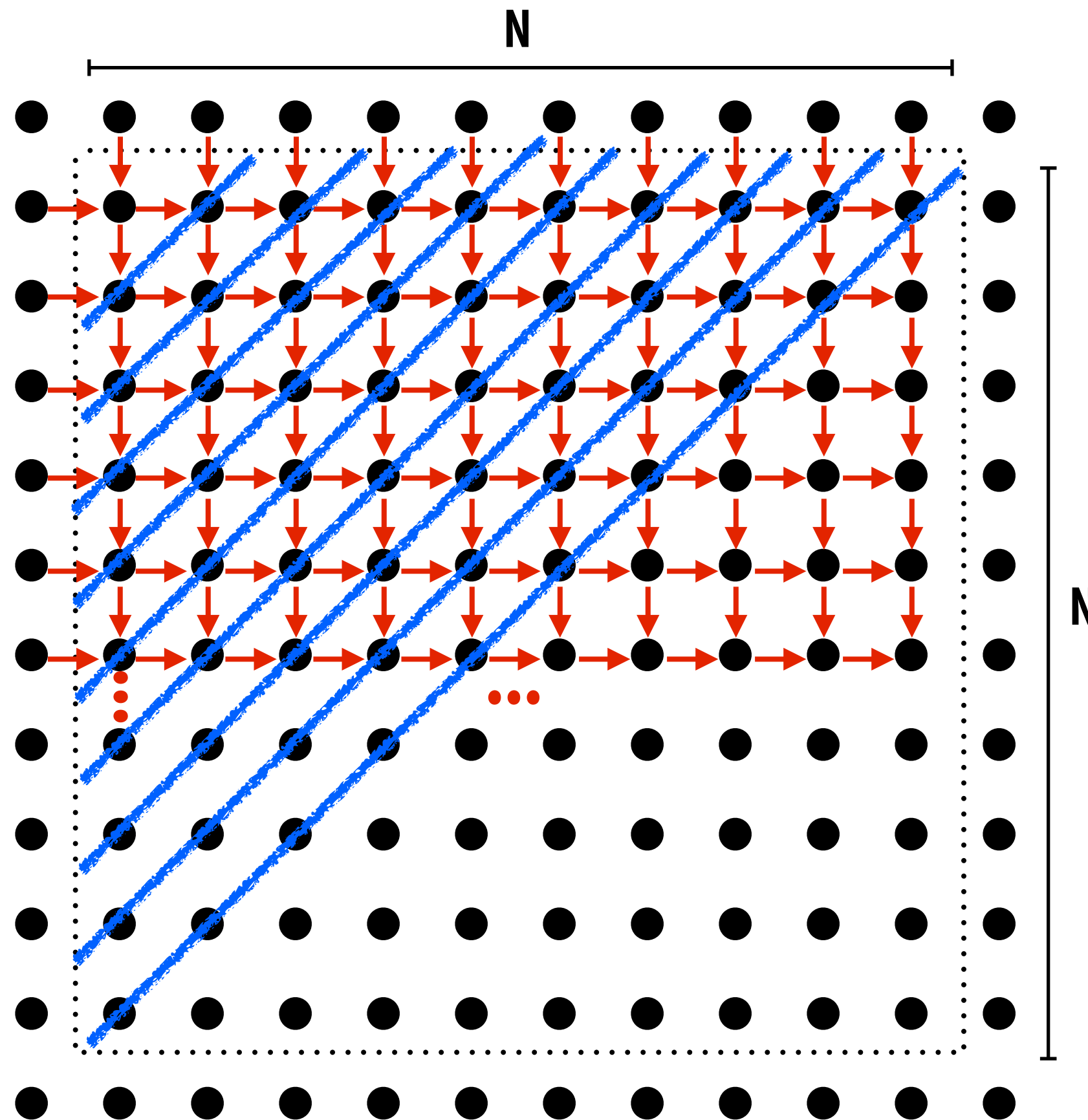
# Step 1: identify dependencies (problem decomposition phase)



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: independent work is hard to exploit**  
**Not much parallelism at beginning and end of computation.**

**Frequent synchronization (after completing each diagonal)**

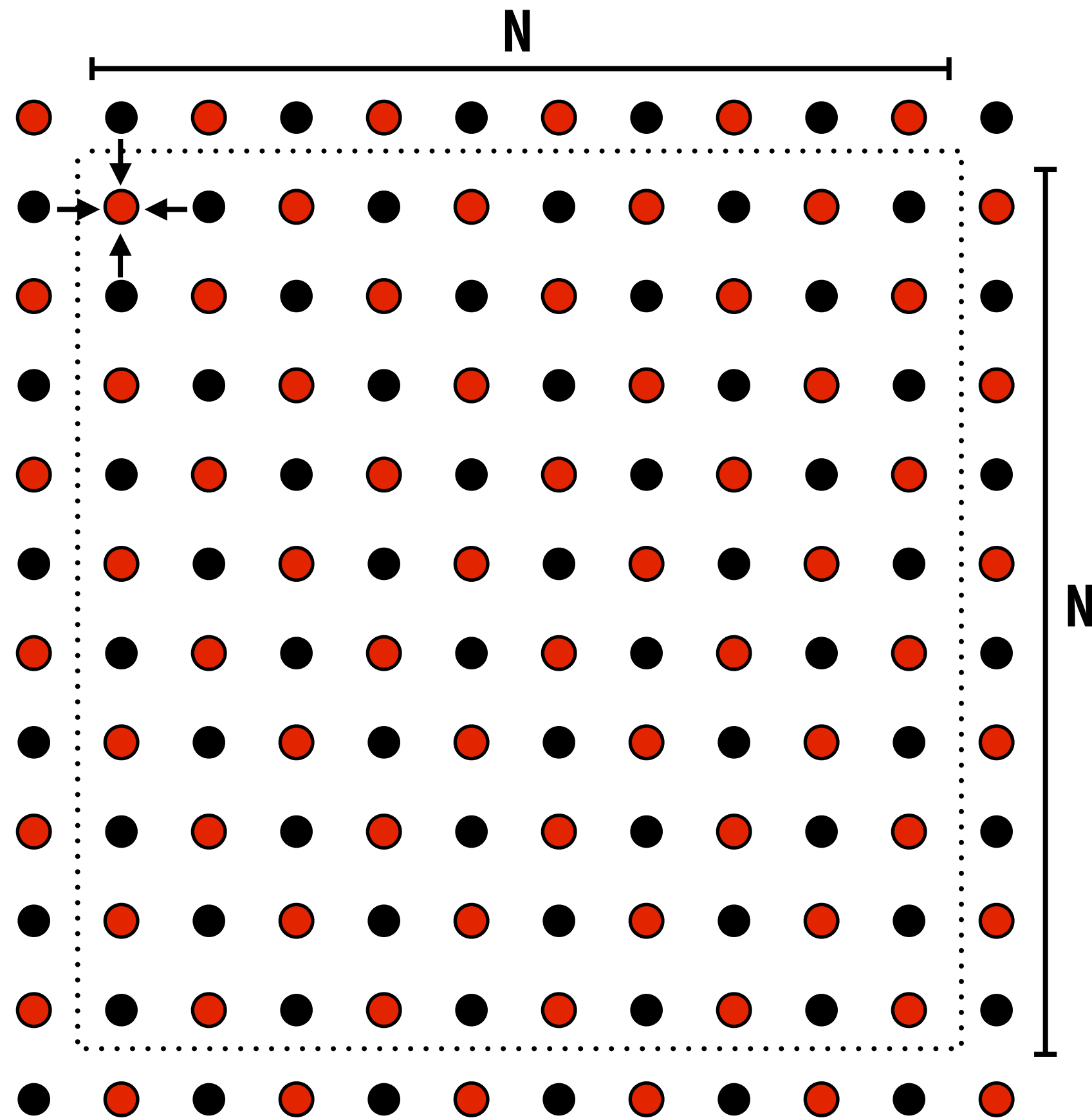


# Let's make life easier on ourselves

**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



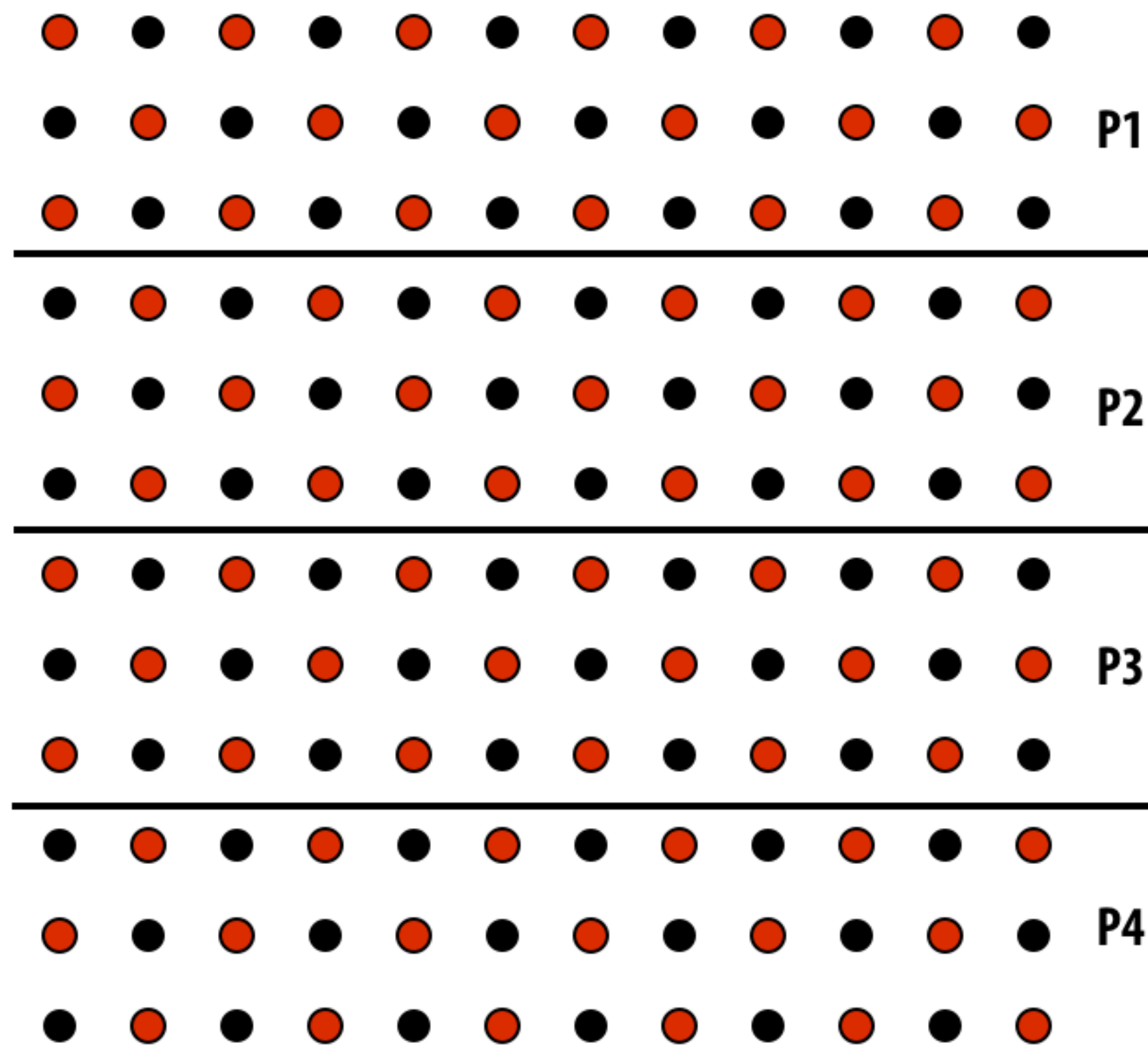
**Update all red cells in parallel**

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

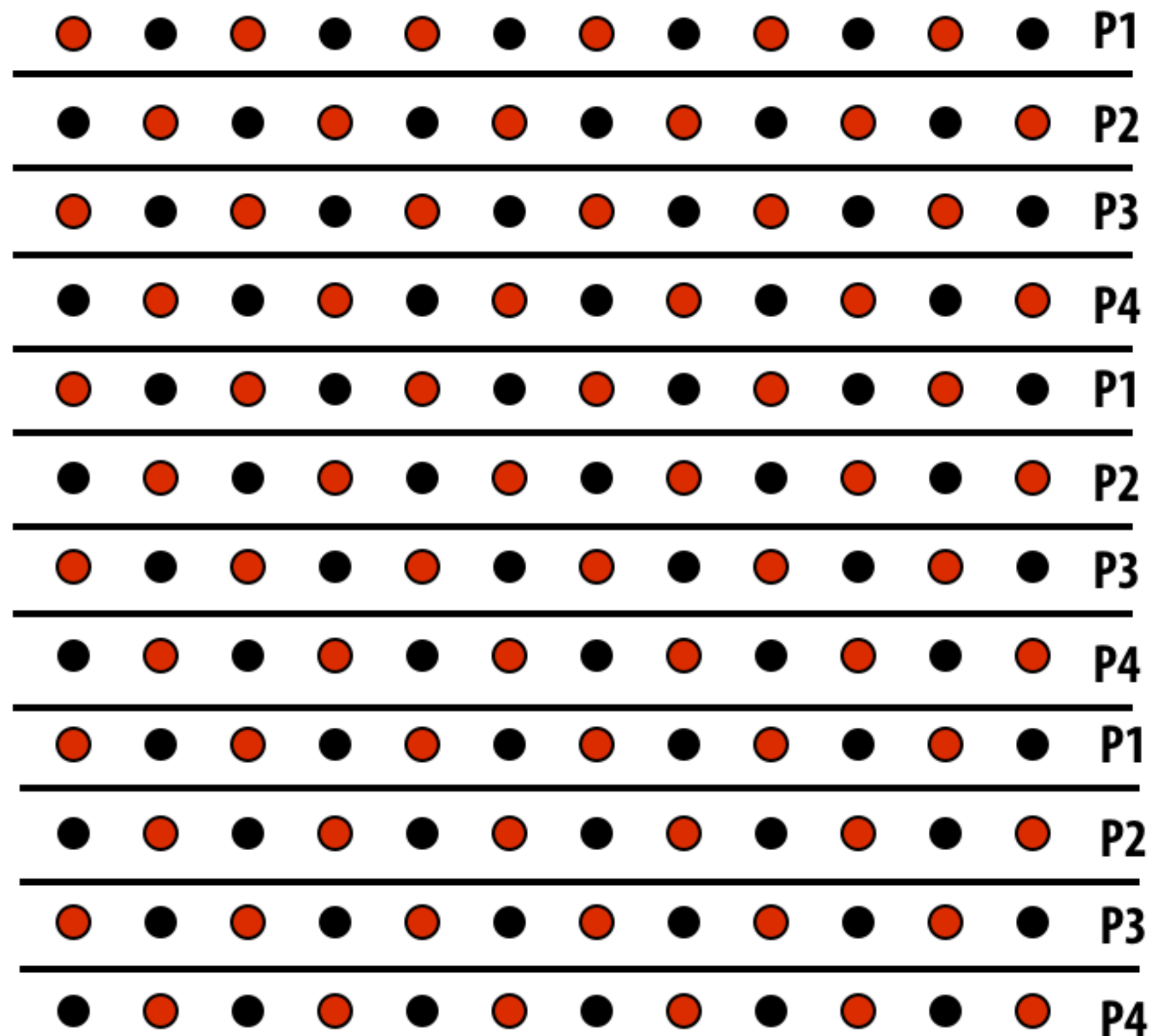
**Repeat until convergence**

# Possible assignments of work to processors

## Blocked Assignment



## Interleaved Assignment

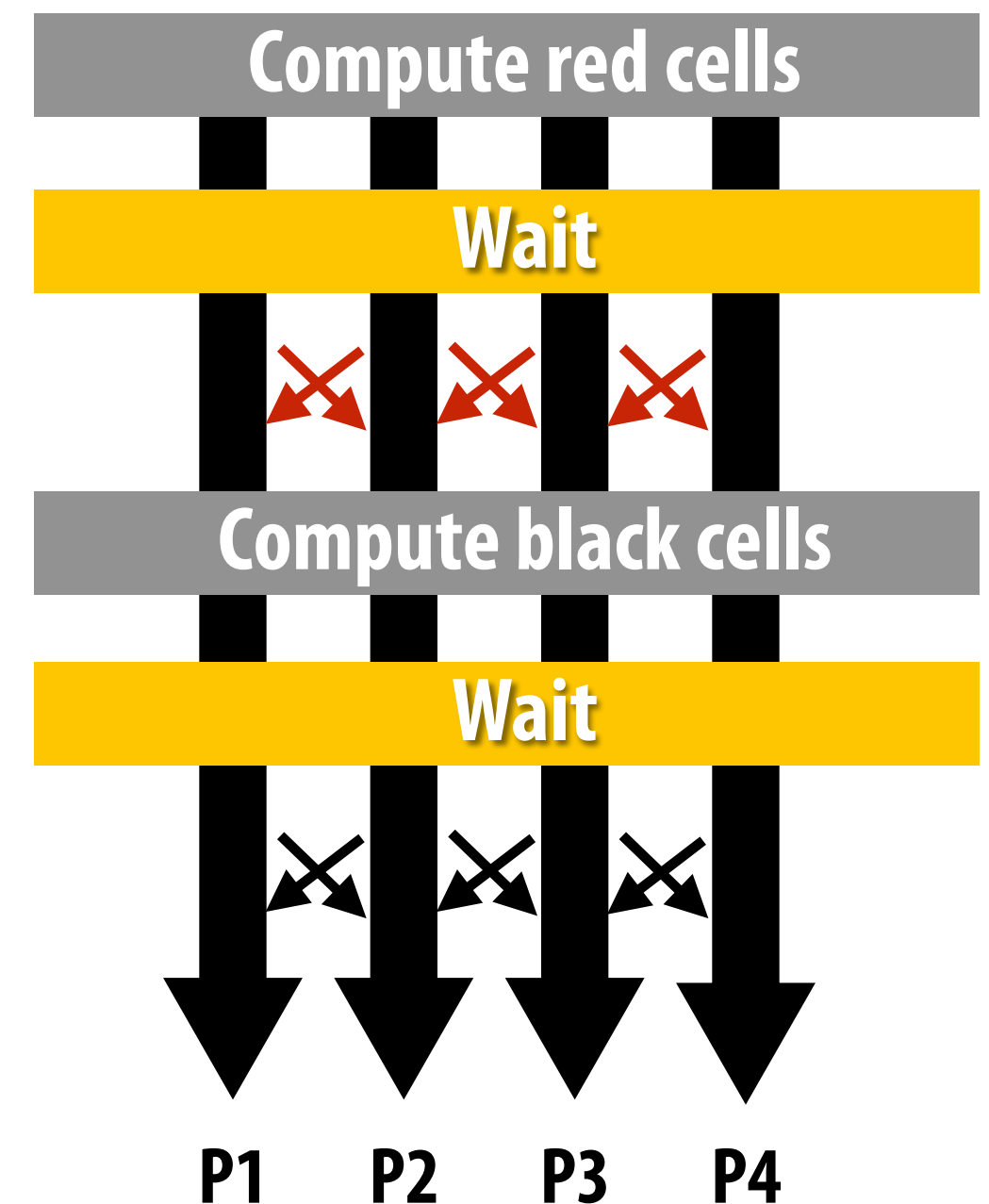


**Question: Which is better? Does it matter?**

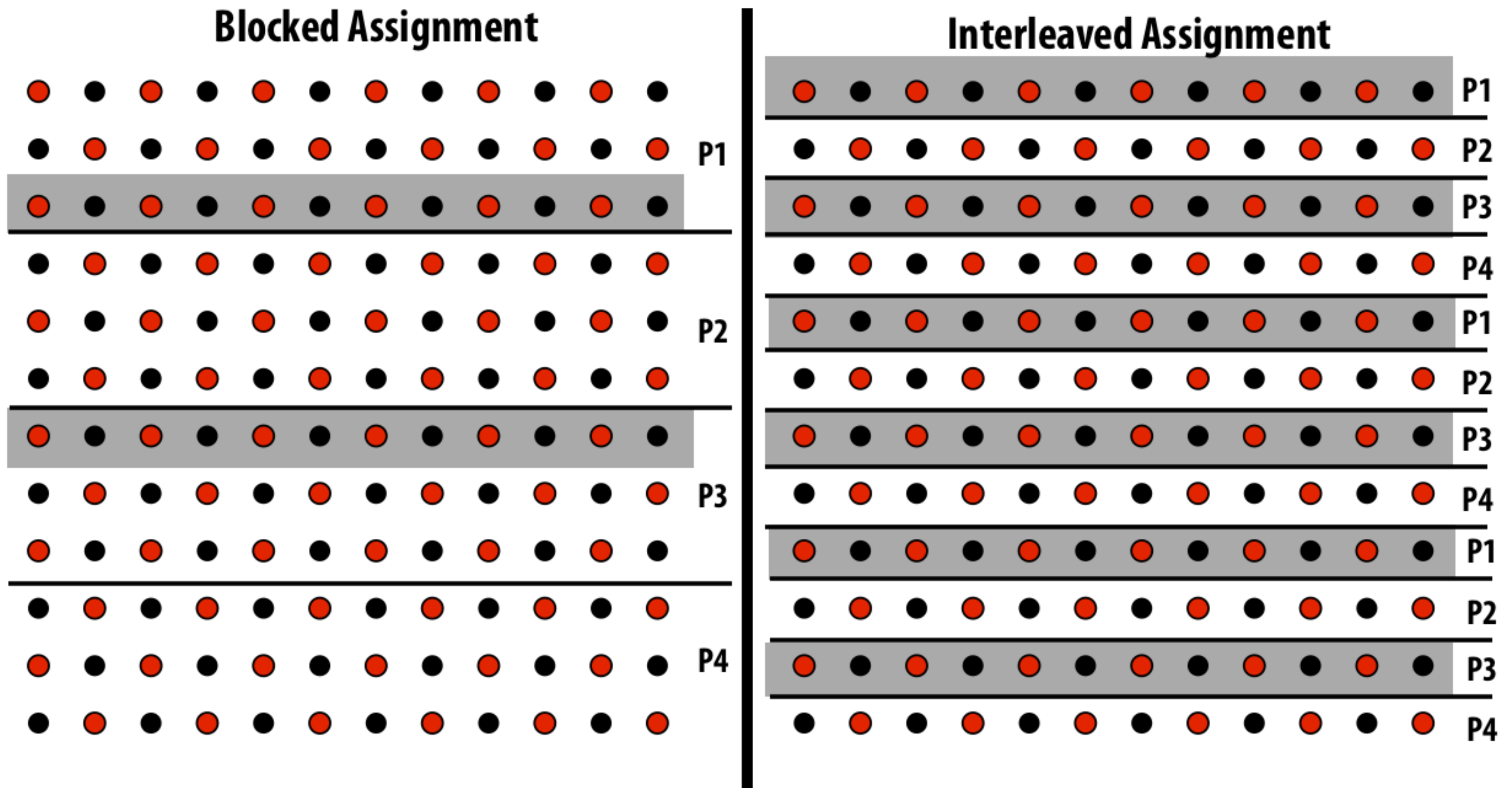
**Answer: it depends on the system this program is running on**

# 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;
```

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

Assignment: ???

```
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;
```

```
    }
```

```
}
```

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)**

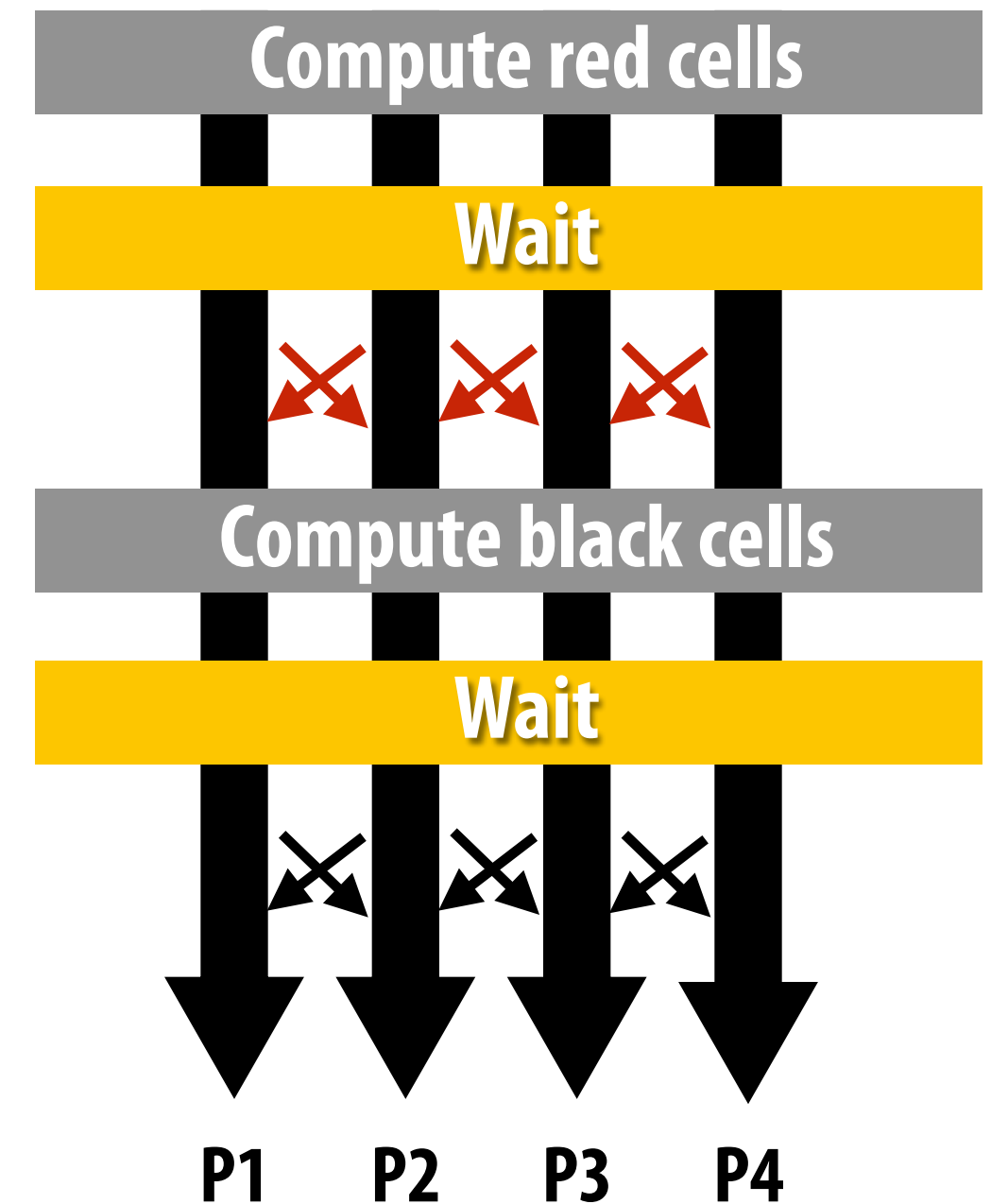
## **expression of solver**



# 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);
```

```
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)
        done = true;
    barrier(myBarrier, NUM_PROCESSORS);
}
```

```
// check convergence, all threads get same answer
```

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

# 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
<code>r1 ← diff</code>	T0 reads value 0
	T1 reads value 0
<code>r1 ← r1 + r2</code>	T0 sets value of its <code>r1</code> to 1
	T1 sets value of its <code>r1</code> to 1
<code>diff ← r1</code>	T0 stores 1 to <code>diff</code>
	T1 stores 1 to <code>diff</code>

- Need this set of three instructions to be atomic

# Mechanisms for preserving 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);
```

# 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?**

# 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**

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

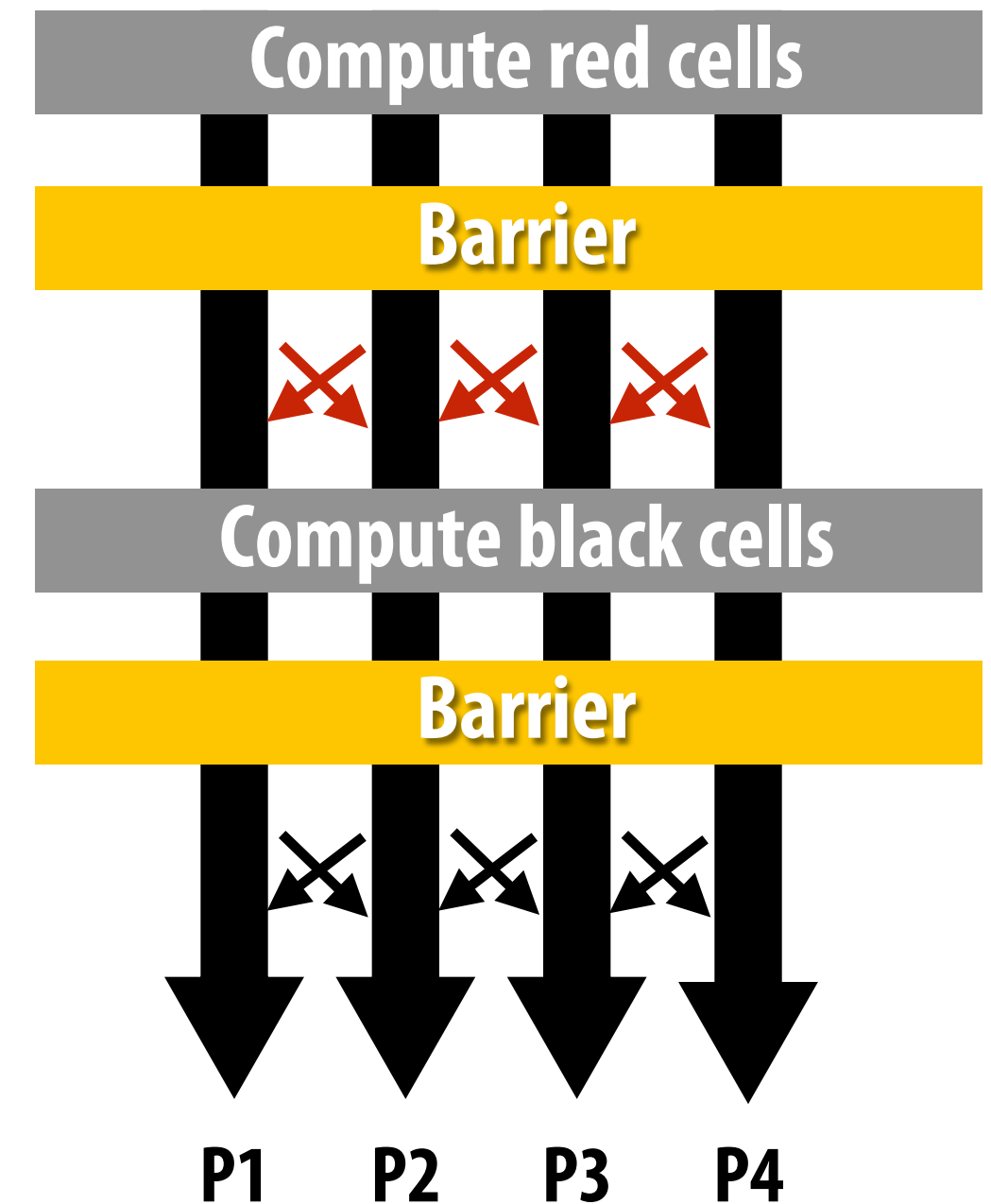
# Barrier synchronization primitive

`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)

```
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)
                done = true;
            barrier(myBarrier, NUM_PROCESSORS);
        }
    }
}
```

Why are there three barriers?

// check convergence, all threads get same answer



# 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;
// do more work here...
```

**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 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 (between phases of computation)
- **Communication**
  - Implicit in loads/stores to shared variables

# **Message-passing expression of solver**

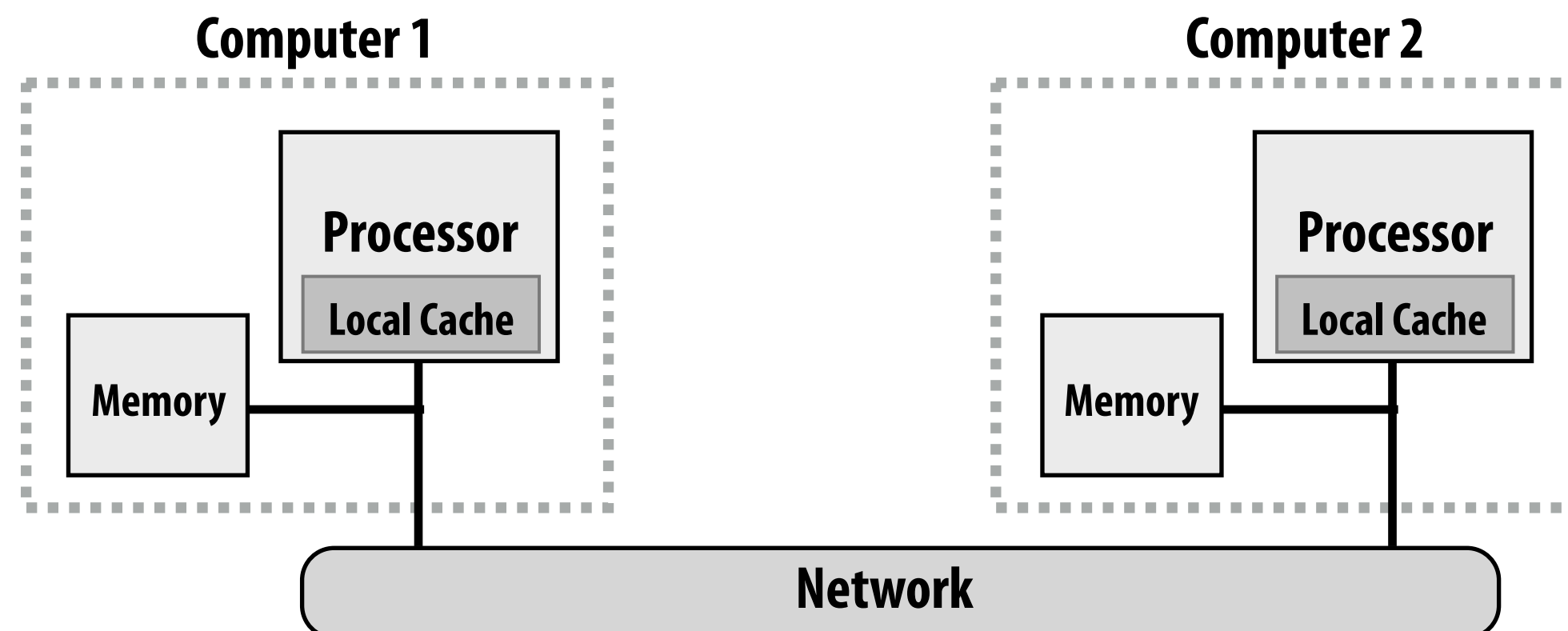
# Let's think about expressing a parallel grid

Each thread has its own address space

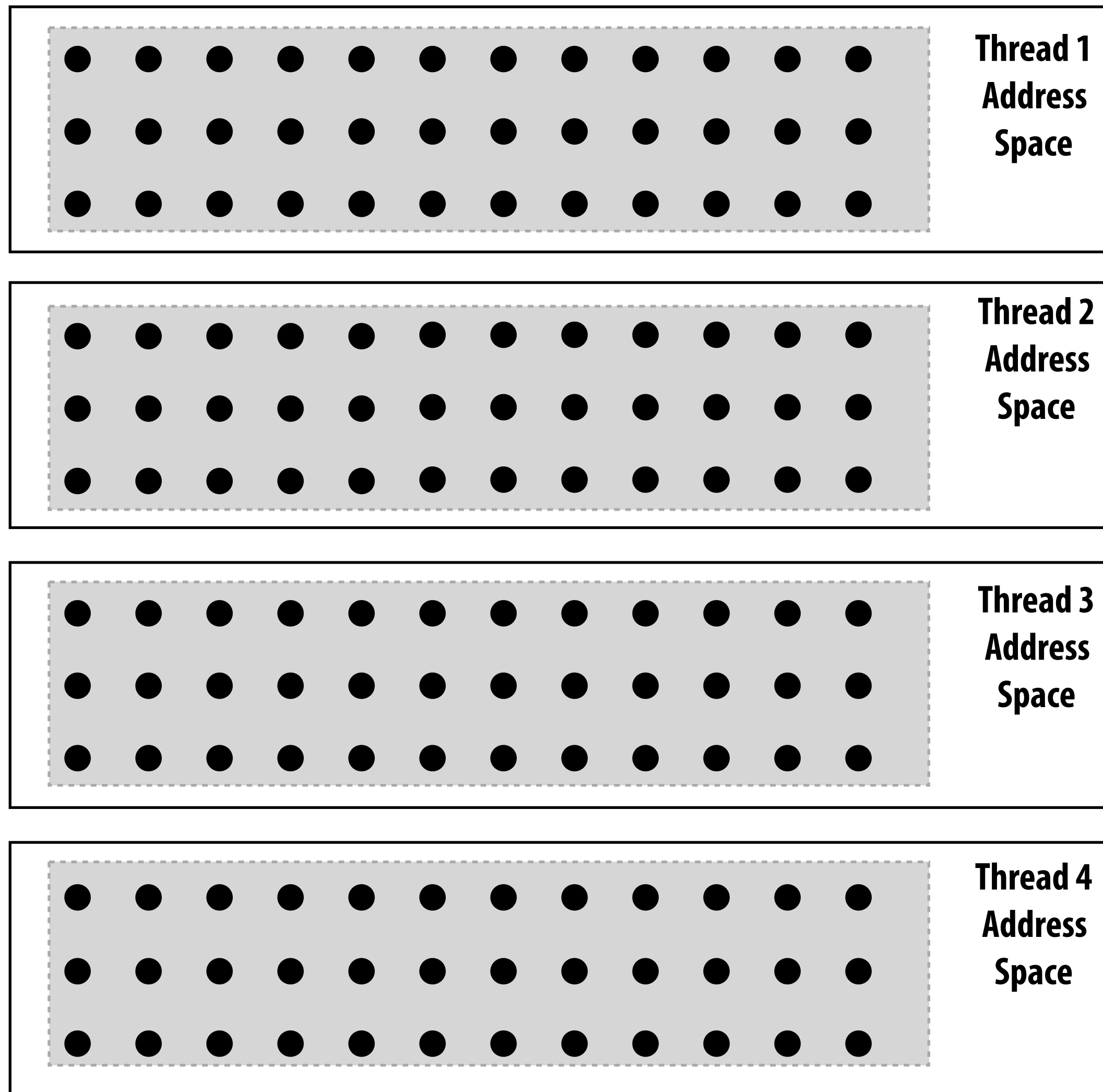
– No shared address space abstraction (i.e., no shared variables)

Threads communicate and synchronize by sending/receiving messages

**One possible message passing machine configuration: a cluster of two workstations (you could make this cluster yourself using the machines in the GHC labs)**



# Message passing model: each thread operates in its own address space

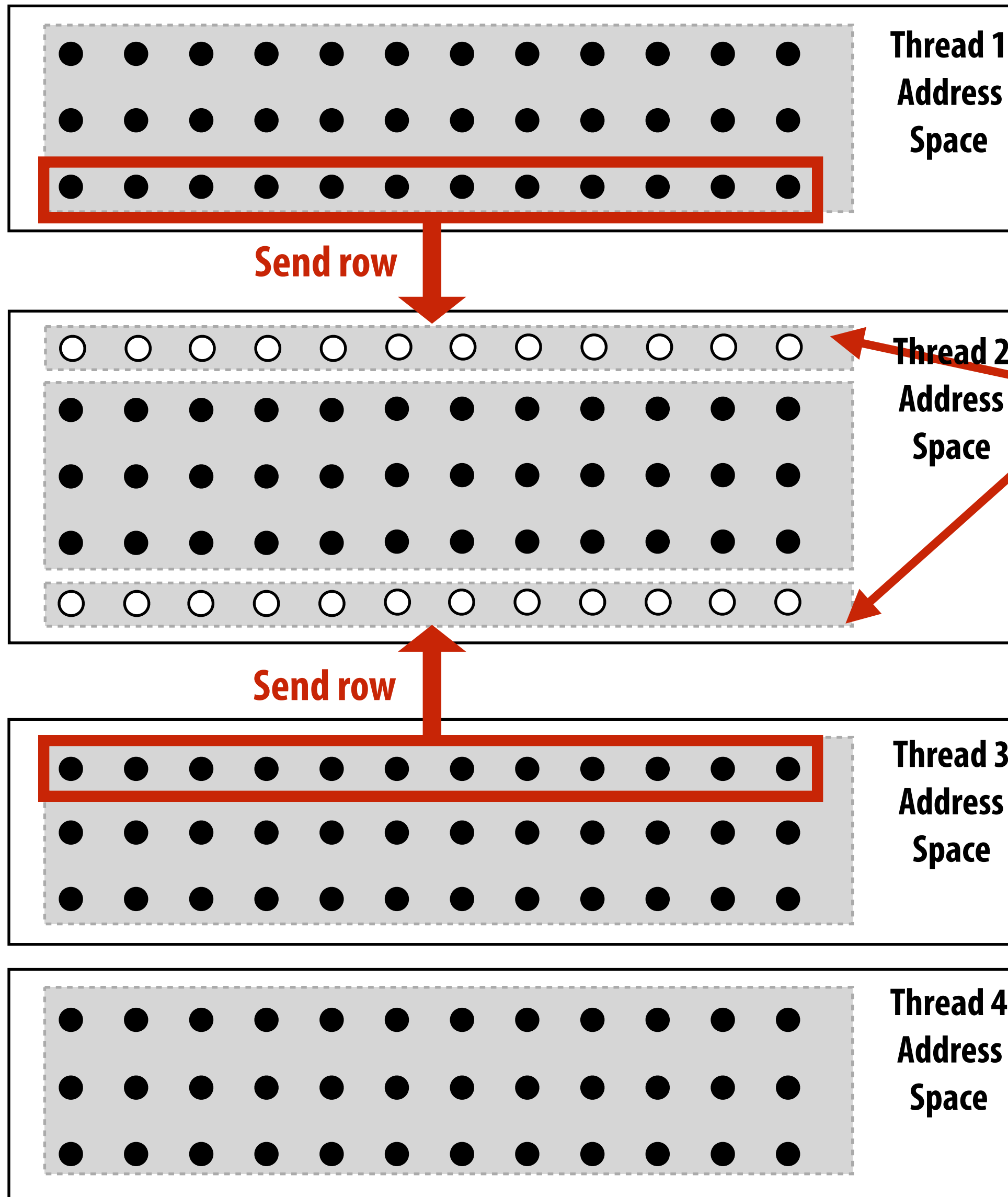


**In this figure: four threads**

**The grid data is partitioned into four allocations, each residing in one of the four unique thread address spaces (four per-thread private arrays)**

# Data replication is now required to correctly execute the program

## Grid data stored in four separate address spaces (four private arrays)



### Example:

After red cell processing is complete, thread 1 and thread 3 send row of data to thread 2 (thread 2 requires up-to-date red cell information to update black cells in the next phase)

"Ghost cells" are grid cells replicated from a remote address space. It's common to say that information in ghost cells is "owned" by other threads.

### Thread 2 logic:

```
float* local_data = allocate(N+2, rows_per_thread+2);

int tid = get_thread_id();
int bytes = sizeof(float) * (N+2);

// receive ghost row cells (white dots)
recv(&local_data[0,0], bytes, tid-1);
recv(&local_data[rows_per_thread+1,0], bytes, tid+1);

// Thread 2 now has data necessary to perform
// future computation
```

# Message passing solver

Similar structure to shared address space solver, but now communication is explicit in message sends and receives

```
int N;
int tid = get_thread_id();
int rows_per_thread = N / get_num_threads();

float* localA = allocate(rows_per_thread+2, N+2);

// assume localA is initialized with starting values
// assume MSG_ID_ROW, MSG_ID_DONE, MSG_ID_DIFF are constants used as msg ids

//////////

void solve() {
    bool done = false;
    while (!done) {

        float my_diff = 0.0f;

        if (tid != 0)
            send(&localA[1,0], sizeof(float)*(N+2), tid-1, MSG_ID_ROW);
        if (tid != get_num_threads()-1)
            send(&localA[rows_per_thread,0], sizeof(float)*(N+2), tid+1, MSG_ID_ROW);

        if (tid != 0)
            recv(&localA[0,0], sizeof(float)*(N+2), tid-1, MSG_ID_ROW);
        if (tid != get_num_threads()-1)
            recv(&localA[rows_per_thread+1,0], sizeof(float)*(N+2), tid+1, MSG_ID_ROW);

        for (int i=1; i<rows_per_thread+1; i++) {
            for (int j=1; j<n+1; j++) {
                float prev = localA[i,j];
                localA[i,j] = 0.2 * (localA[i-1,j] + localA[i,j] + localA[i+1,j] +
                                   localA[i,j-1] + localA[i,j+1]);
                my_diff += fabs(localA[i,j] - prev);
            }
        }

        if (tid != 0) {
            send(&mydiff, sizeof(float), 0, MSG_ID_DIFF);
            recv(&done, sizeof(bool), 0, MSG_ID_DONE);
        } else {
            float remote_diff;
            for (int i=1; i<get_num_threads()-1; i++) {
                recv(&remote_diff, sizeof(float), i, MSG_ID_DIFF);
                my_diff += remote_diff;
            }
            if (my_diff/(N*N) < TOLERANCE)
                done = true;
            for (int i=1; i<get_num_threads()-1; i++)
                send(&done, sizeof(bool), i, MSG_ID_DONE);
        }
    }
}
```

Send and receive ghost rows to “neighbor threads”

Perform computation

(just like in shared address space version of solver)

All threads send local my\_diff to thread 0

Thread 0 computes global diff, evaluates termination predicate and sends result back to all other threads



# Notes on message passing example

## Computation

- Array indexing is relative to **local address space** (not global grid coordinates)

## Communication:

- Performed by sending and receiving messages
- Bulk transfer: **communicate entire rows at a time** (not individual elements)

## Synchronization:

- Performed by **sending and receiving messages**
- Think of how to implement mutual exclusion, barriers, flags using messages

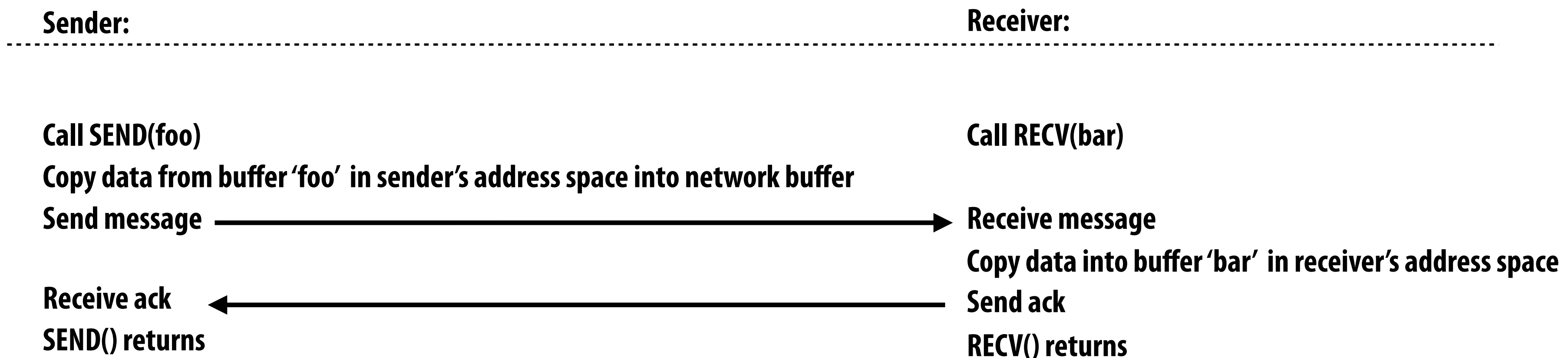
**For convenience, message passing libraries often include higher-level primitives (implemented via send and receive)**

```
reduce_add(0, &my_diff, sizeof(float));           // add up all my_diffs, return result to thread 0
if (pid == 0 && my_diff/(N*N) < TOLERANCE)
    done = true;
broadcast(0, &done, sizeof(bool), MSG_DONE);     // thread 0 sends done to all threads
```

# Synchronous (blocking) send and receive

**send():** call returns when sender receives acknowledgement that message data resides in address space of receiver

**recv():** call returns when data from received message is copied into address space of receiver and acknowledgement sent back to sender



**As implemented on the prior slide, there is a big problem with our message passing solver if it uses synchronous send/rcv!**

**Why?**

**How can we fix it?**

**(while still using synchronous send/rcv)**

# Message passing solver

```

int N;
int tid = get_thread_id();
int rows_per_thread = N / get_num_threads();

float* localA = allocate(rows_per_thread+2, N+2);

// assume localA is initialized with starting values
// assume MSG_ID_ROW, MSG_ID_DONE, MSG_ID_DIFF are constants used as msg ids
////////////////////////////////////

void solve() {
    bool done = false;
    while (!done) {

        float my_diff = 0.0f;

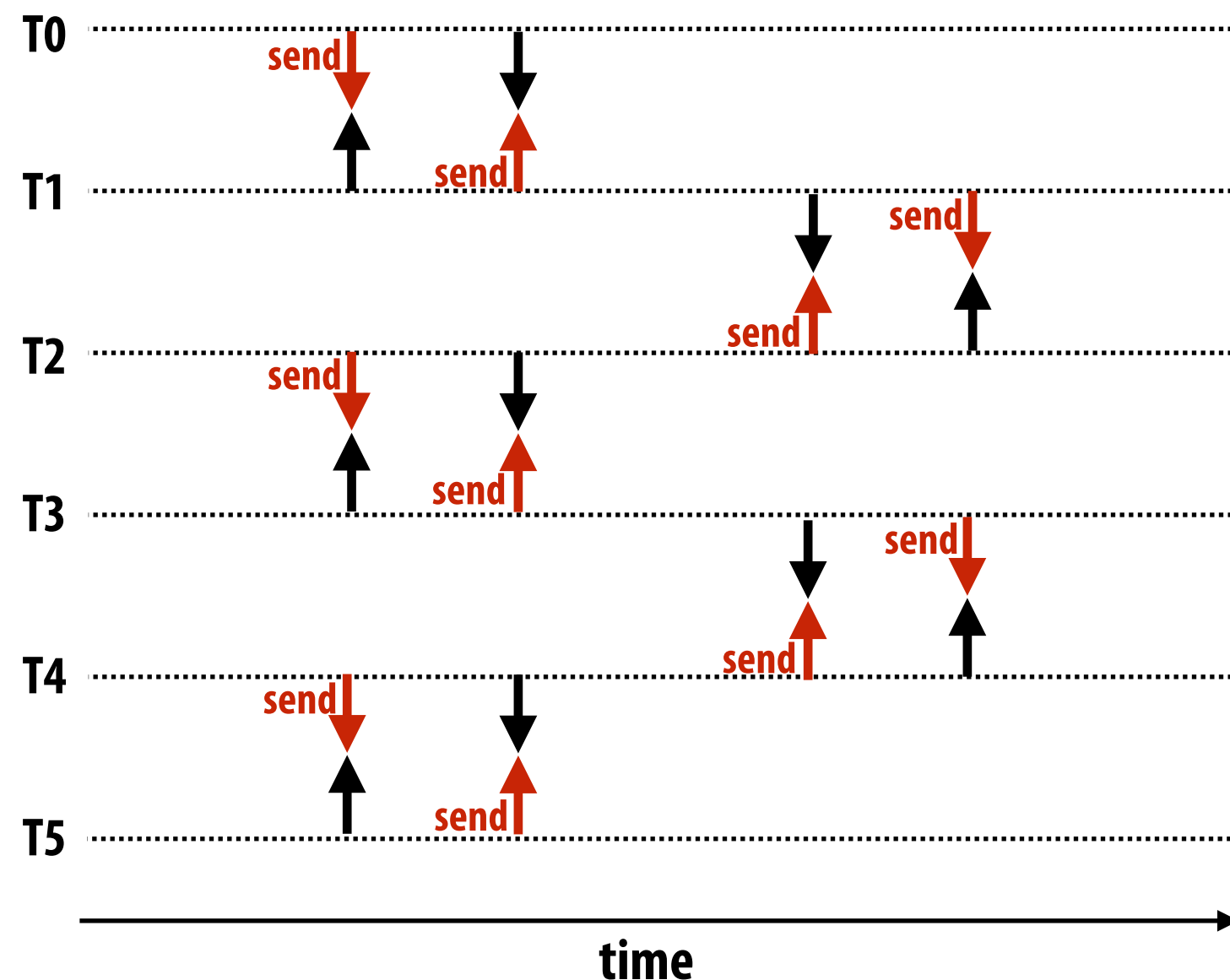
        if (tid % 2 == 0) {
            sendDown(); rcvDown();
            sendUp();   rcvUp();
        } else {
            rcvUp();   sendUp();
            rcvDown(); sendDown();
        }

        for (int i=1; i<rows_per_thread-1; i++) {
            for (int j=1; j<n+1; j++) {
                float prev = localA[i,j];
                localA[i,j] = 0.2 * (localA[i-1,j] + localA[i,j] + localA[i+1,j] +
                                     localA[i,j-1] + localA[i,j+1]);
                my_diff += fabs(localA[i,j] - prev);
            }
        }

        if (tid != 0) {
            send(&mydiff, sizeof(float), 0, MSG_ID_DIFF);
            rcv(&done, sizeof(bool), 0, MSG_ID_DONE);
        } else {
            float remote_diff;
            for (int i=1; i<get_num_threads()-1; i++) {
                rcv(&remote_diff, sizeof(float), i, MSG_ID_DIFF);
                my_diff += remote_diff;
            }
            if (my_diff/(N*N) < TOLERANCE)
                done = true;
            if (int i=1; i<gen_num_threads()-1; i++)
                send(&done, sizeof(bool), i, MSD_ID_DONE);
        }
    }
}

```

Send and receive ghost rows to "neighbor threads"  
 Even-numbered threads send, then receive  
 Odd-numbered thread rcv, then send



# Non-blocking asynchronous send/recv

## send(): call returns immediately

- Buffer provided to send() cannot be modified by calling thread since message processing occurs concurrently with thread execution
- Calling thread can perform other work while waiting for message to be sent

## recv(): posts intent to receive in the future, returns immediately

- Use checksend(), checkrecv() to determine actual status of send/receipt
- Calling thread can perform other work while waiting for message to be received

Sender:

Call SEND(foo)  
SEND returns handle h1

Copy data from 'foo' into network buffer  
Send message

Call CHECKSEND(h1) // if message sent, now safe for thread to modify 'foo'

Receiver:

Call RECV(bar)  
RECV(bar) returns handle h2

Receive message  
Messaging library copies data into 'bar'

Call CHECKRECV(h2)  
// if received, now safe for thread  
// to access 'bar'

RED TEXT = executes concurrently with application thread

# 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