Lecture 4:

Parallel Programming Basics

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2014

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?

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

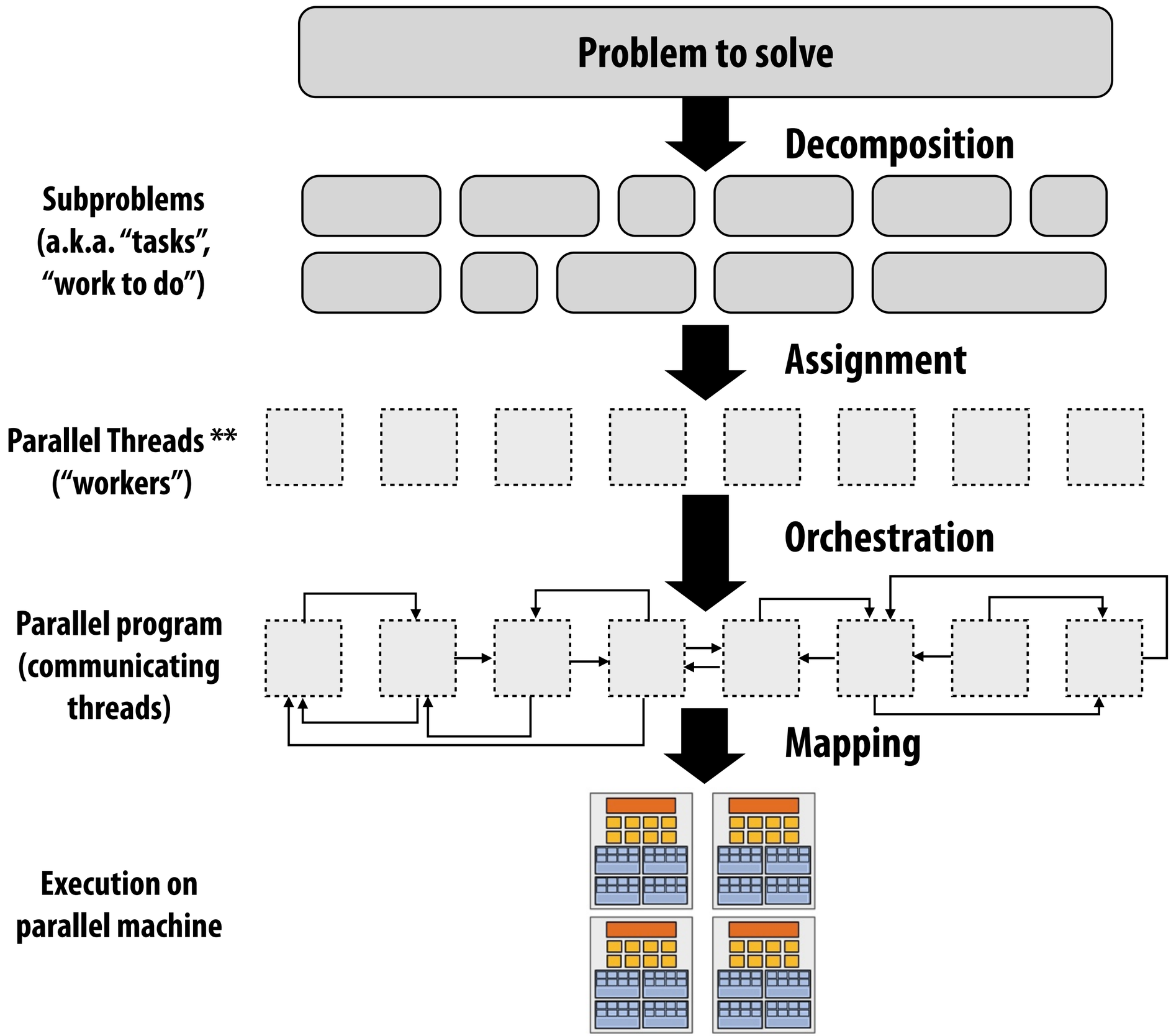
- **Recall one of our main goals is speedup ***

  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

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 at least enough tasks to keep all execution units on a machine busy

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

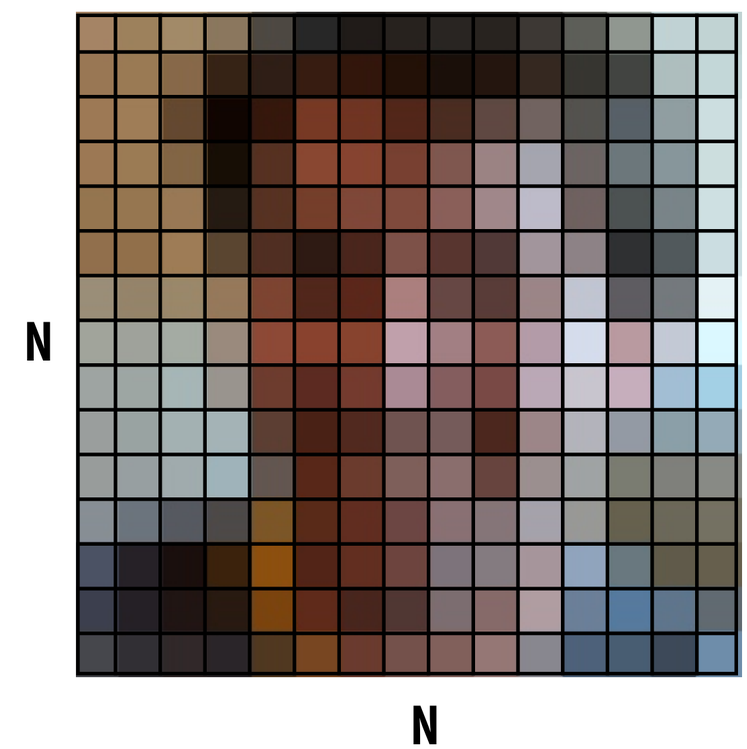
Amdahl's Law: dependencies limit maximum speedup due to parallelism

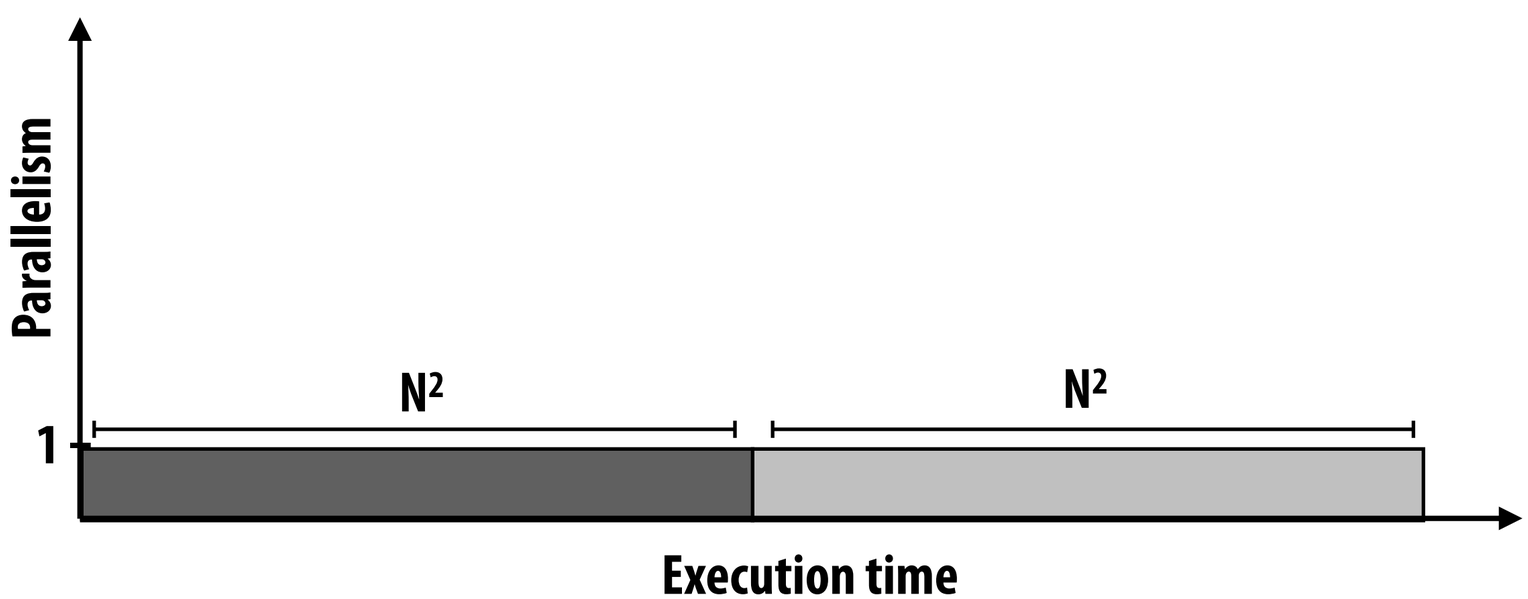
- You run your favorite sequential program...
- Let S = the fraction of sequential execution that is inherently sequential (dependencies prevent parallel execution)
- Then maximum speedup due to parallel execution $\leq {}^{1}/_{S}$

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

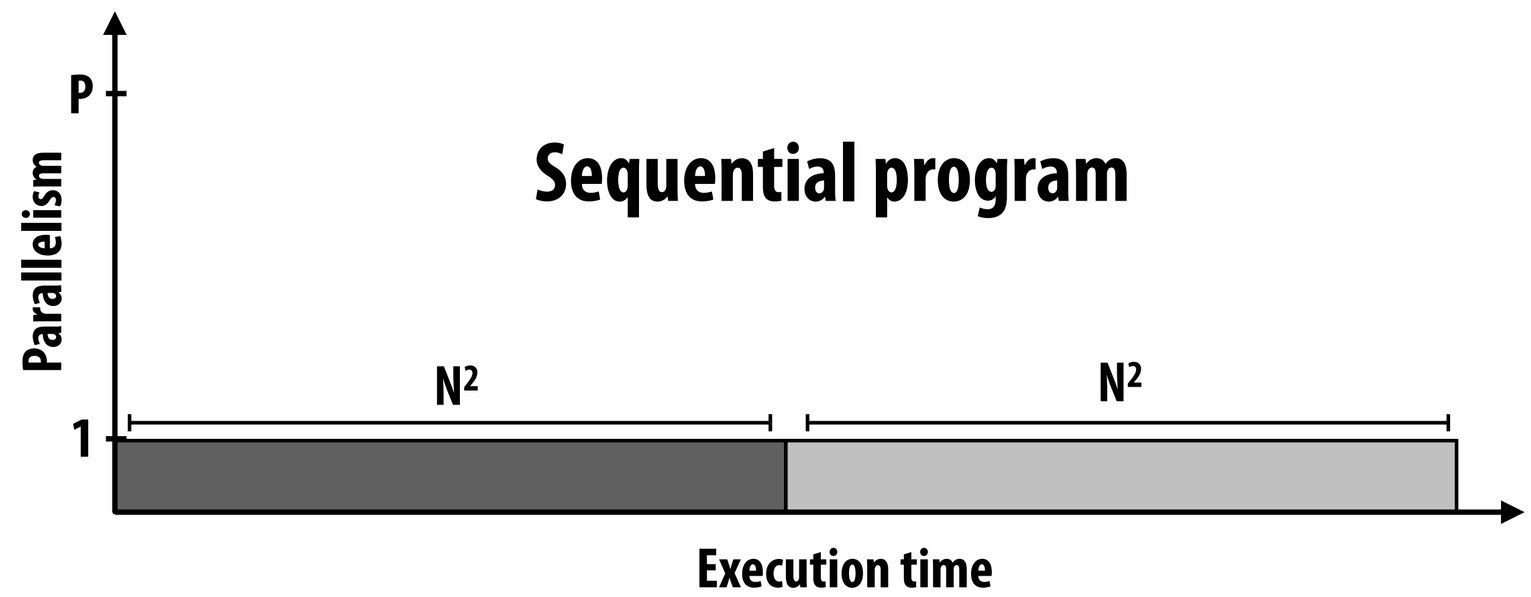
- **Sequential implementation of program**
  - Both steps take ~ N^2 time, so total time is ~ $2N^2$





First attempt at parallelism (P processors)

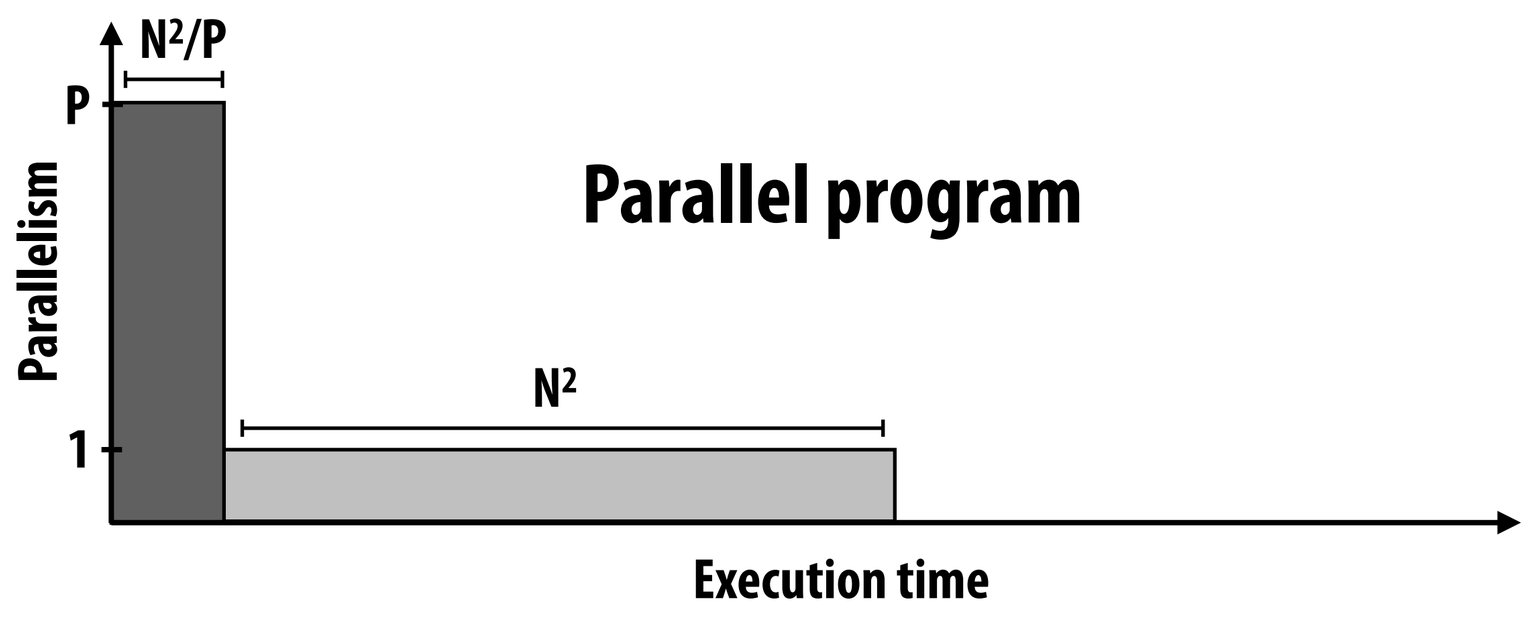
- **Strategy:**
  - Step 1: execute in parallel
    - time for phase 1: N^2/P
  - Step 2: execute serially
    - time for phase 2: N^2



- **Overall performance:**

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

Speedup ≤ 2

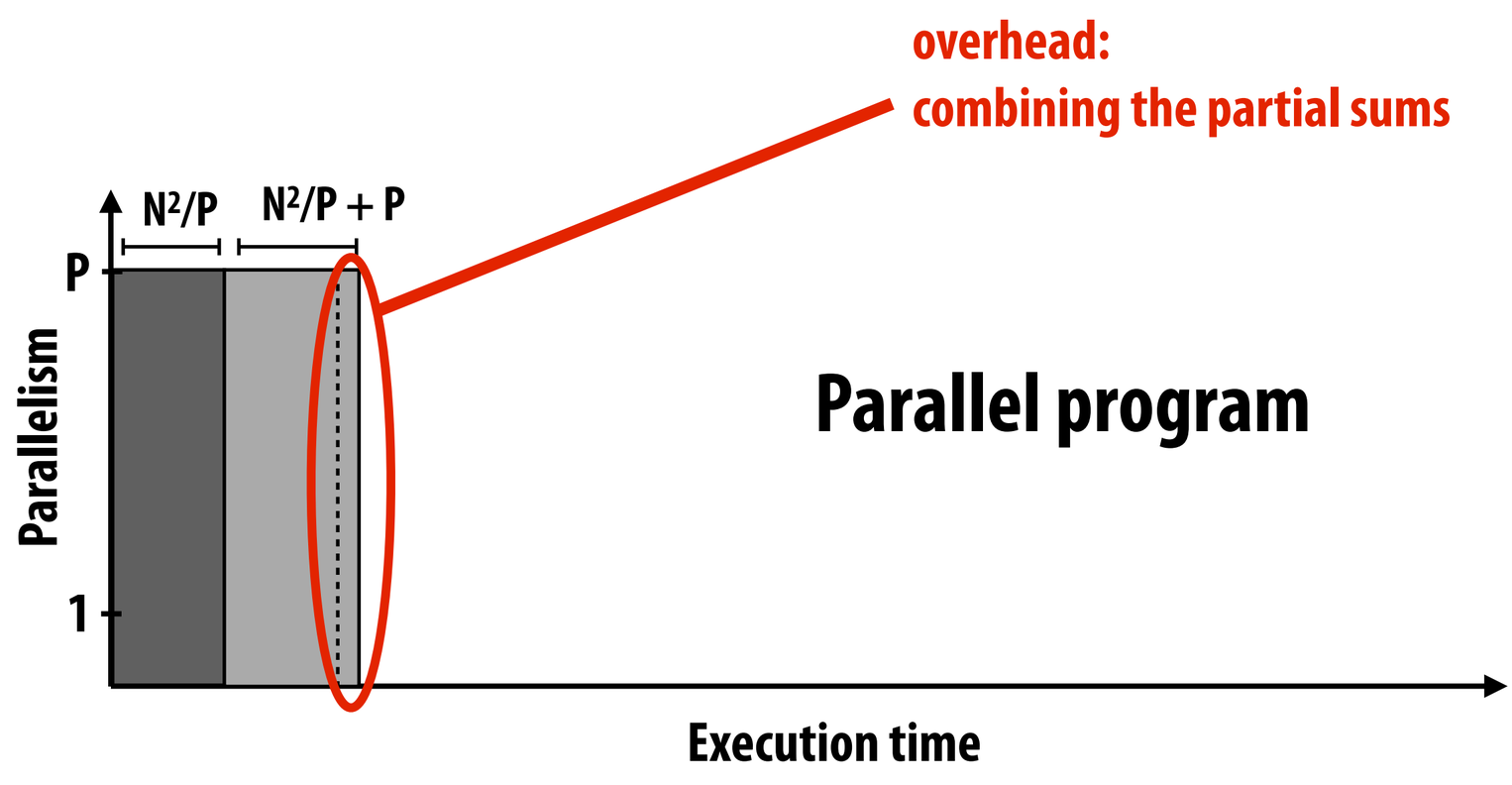


Parallelizing step 2

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

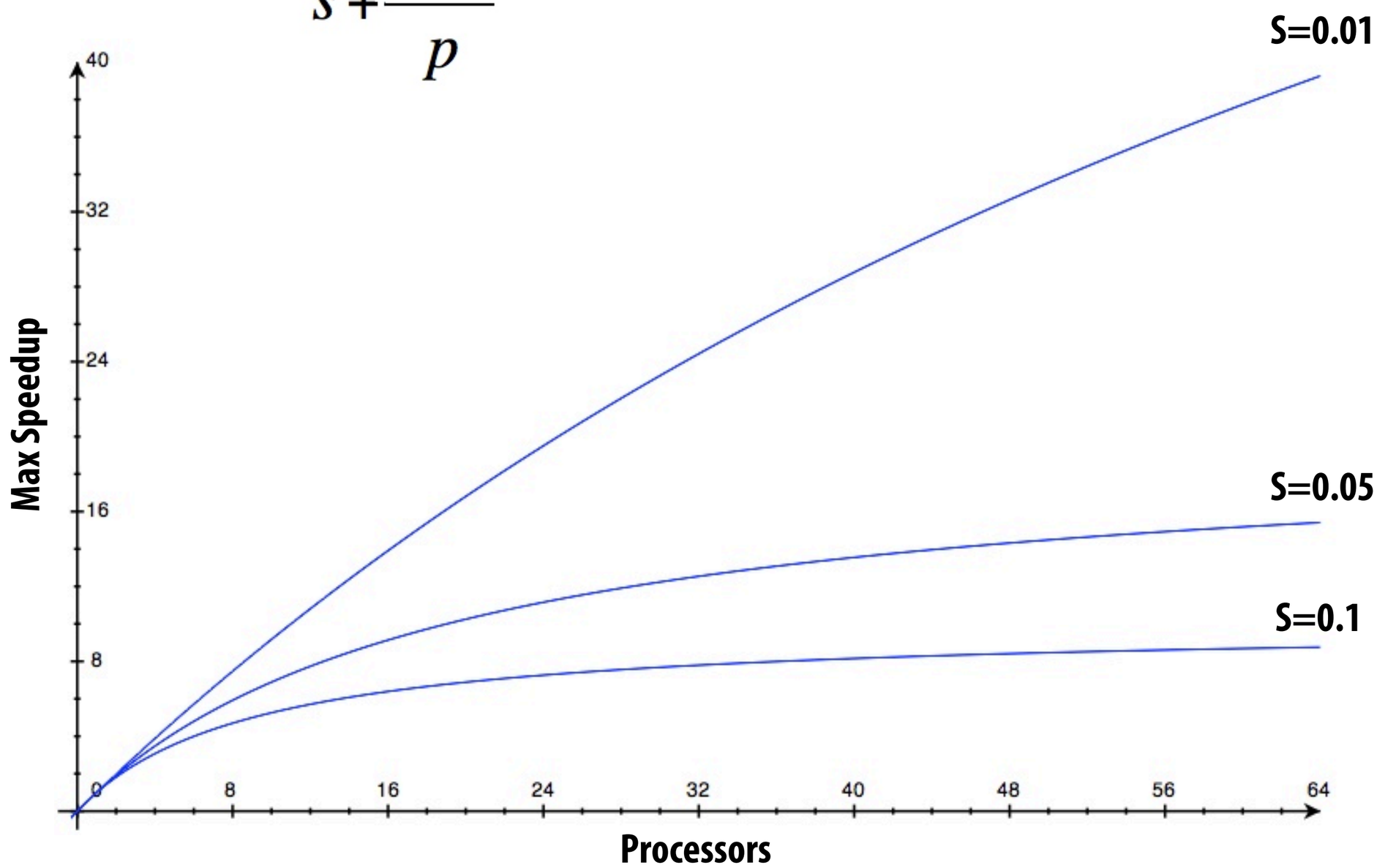
Note: speedup → P when N >> P



Amdahl's law

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

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

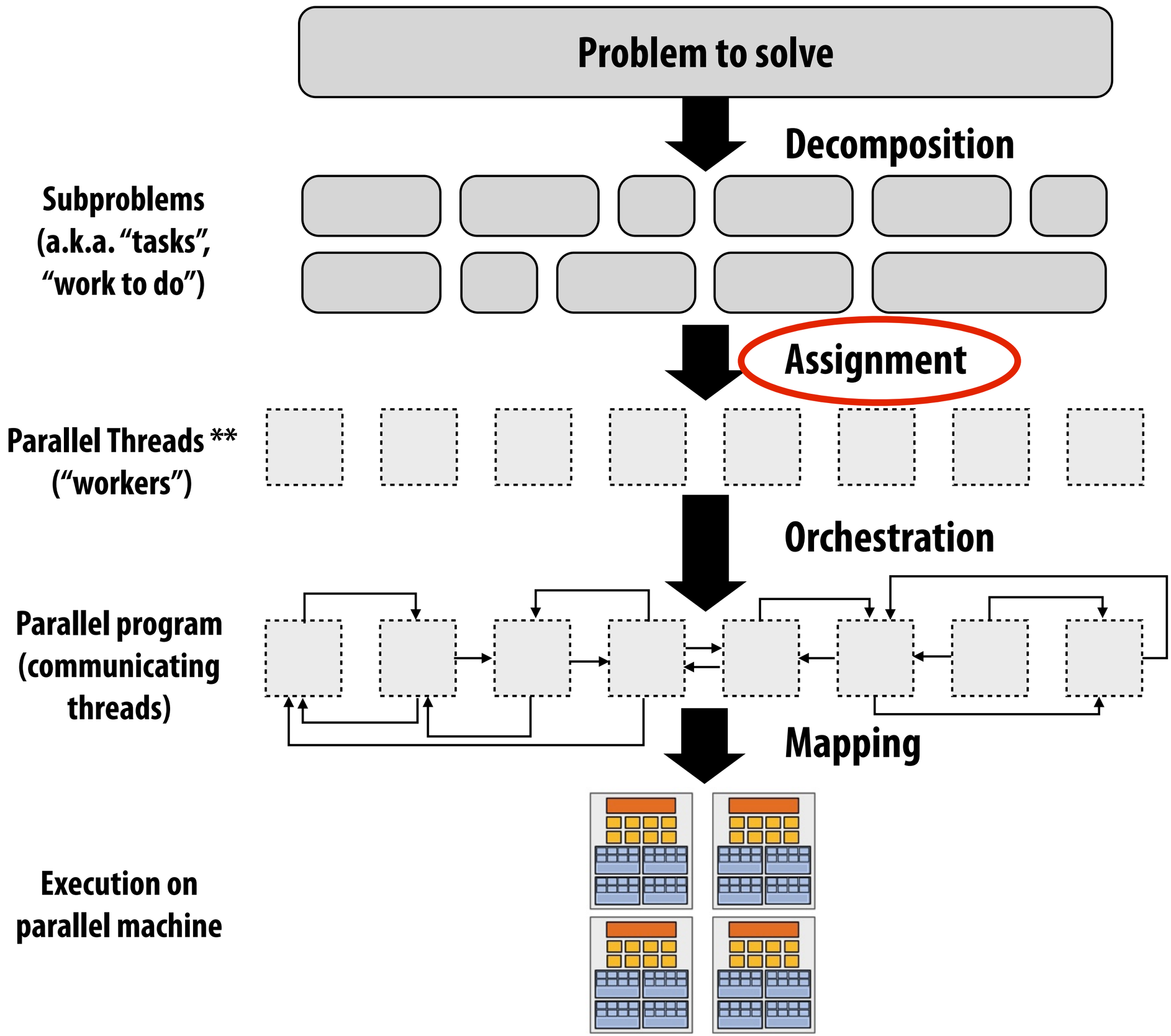


Decomposition

- **Who is responsible for performing decomposition?**
  - In 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, identify dependencies
    - What if dependencies are data-dependent (not known at compile time)?
  - Researchers have had 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**

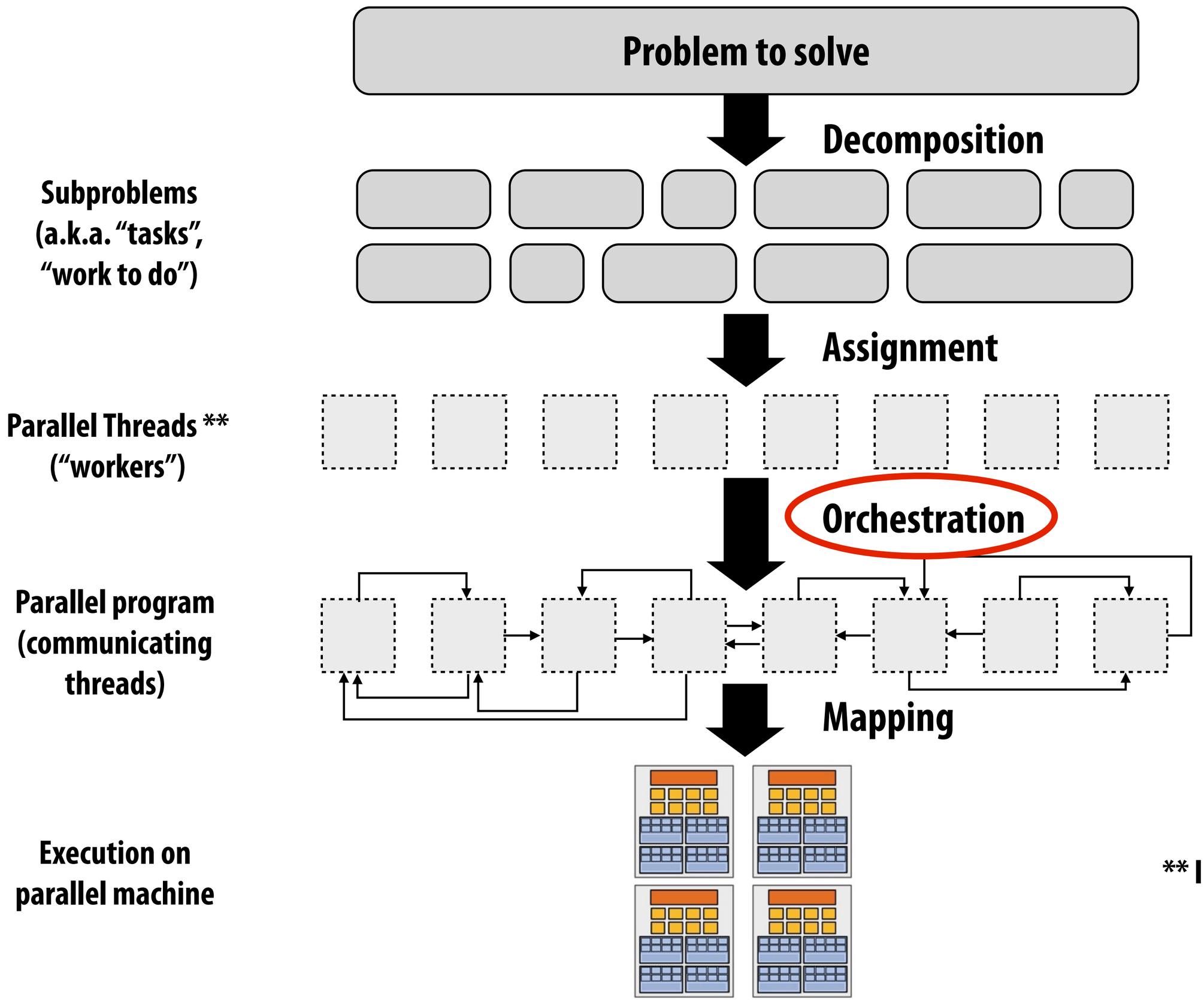
```
export void sinx(
   uniform int N,
   uniform int terms,
   uniform float* x,
   uniform float* result)
{
   foreach (i = 0 ... N)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      uniform int denom = 6;  // 3!
      uniform int sign = -1;

      for (uniform int j=1; j<=terms; j++)
      {
         value += sign * numer / denom
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         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



** I had to pick a term

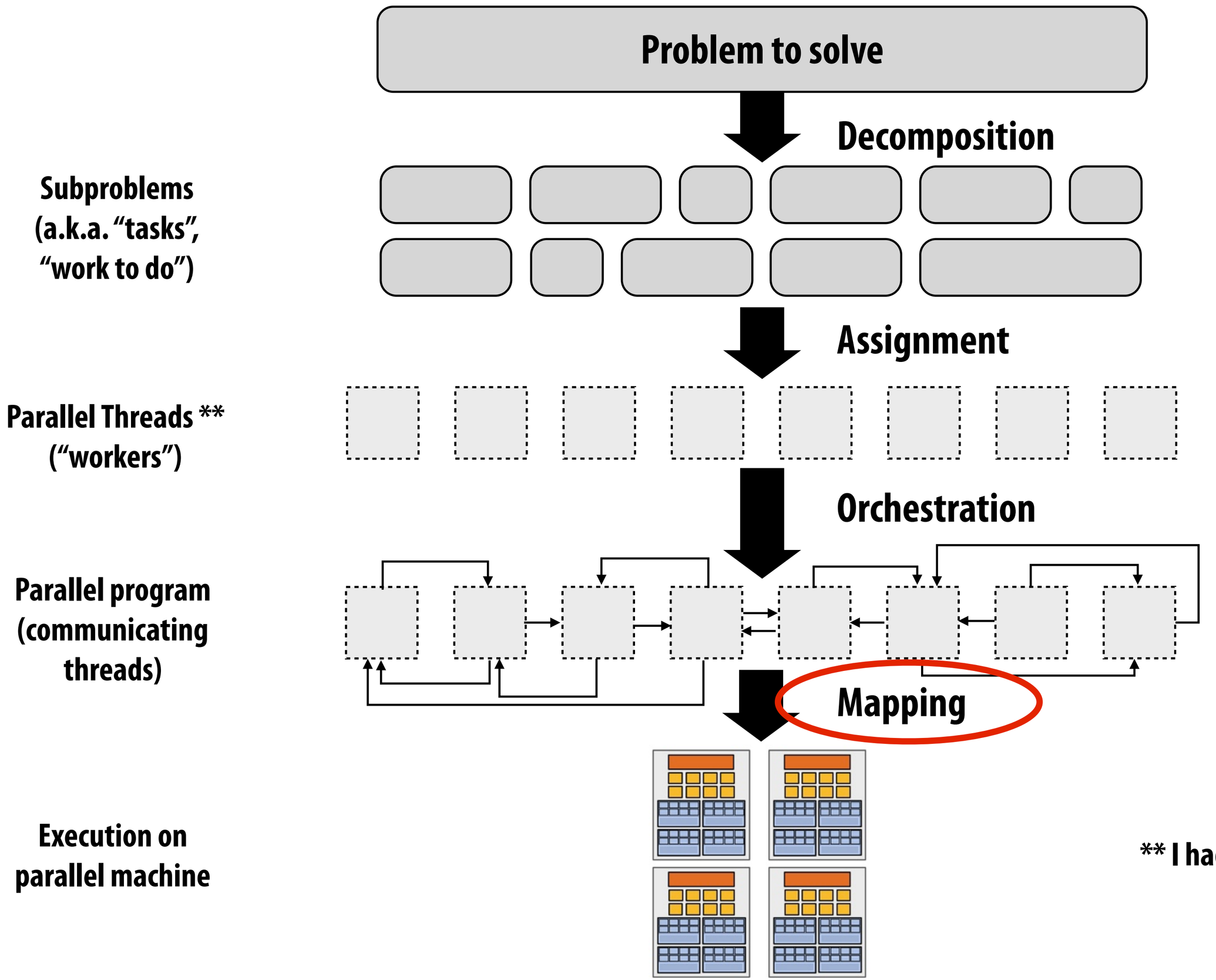
Orchestration

- **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

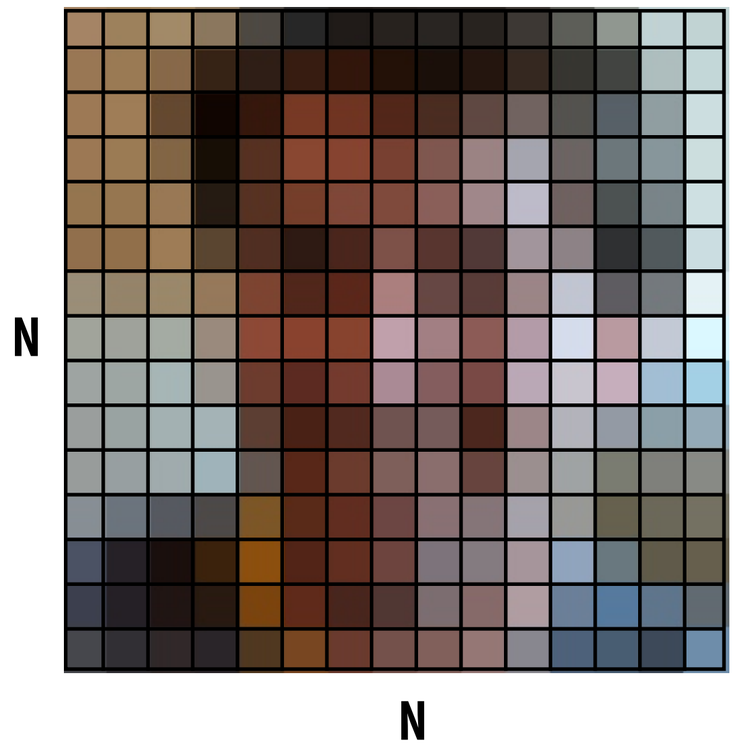
Mapping

- 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 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>.

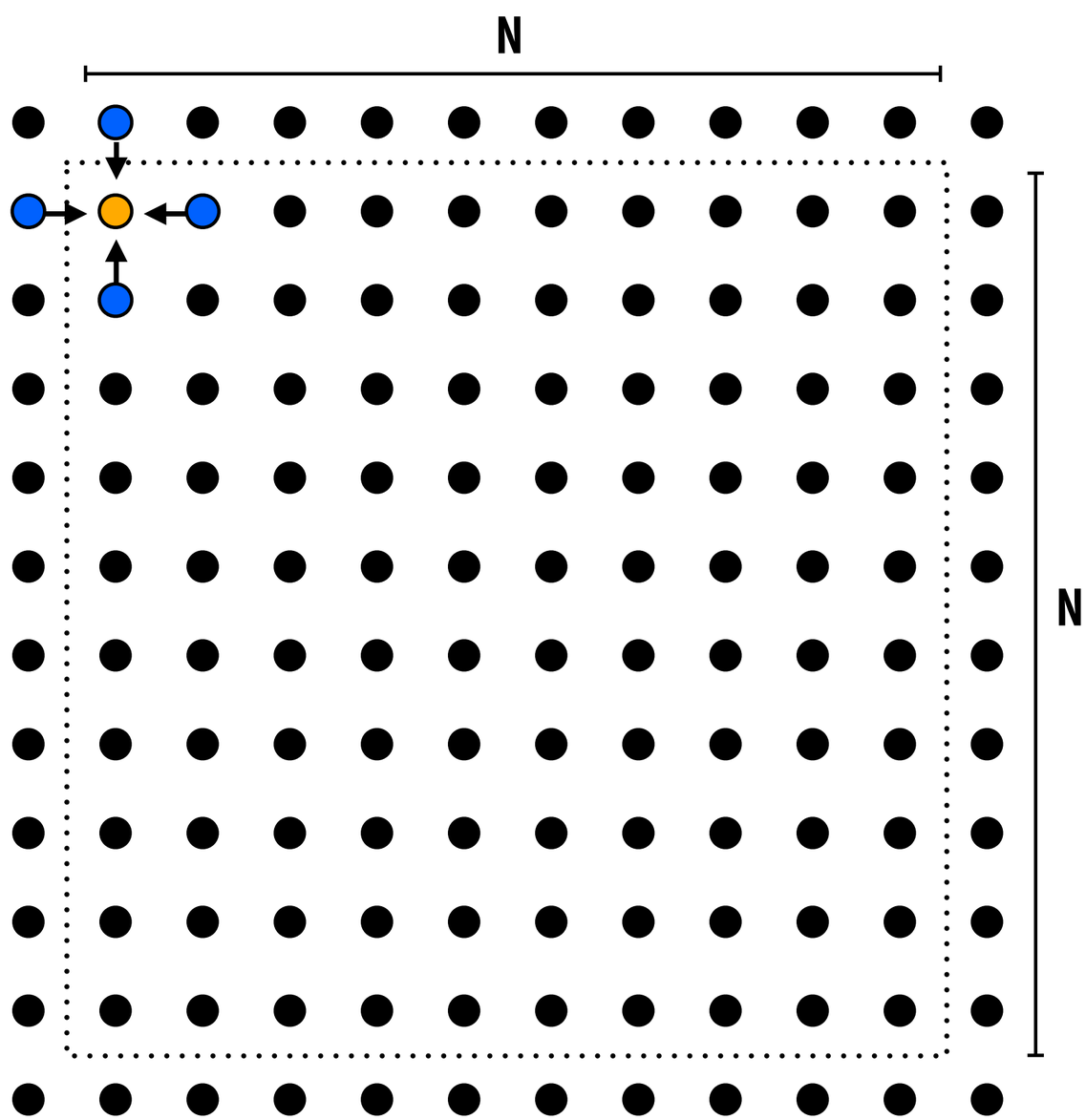
Often, it's equally valid to think of <u>partitioning data</u>. (computations go with the data)

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 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]);
```

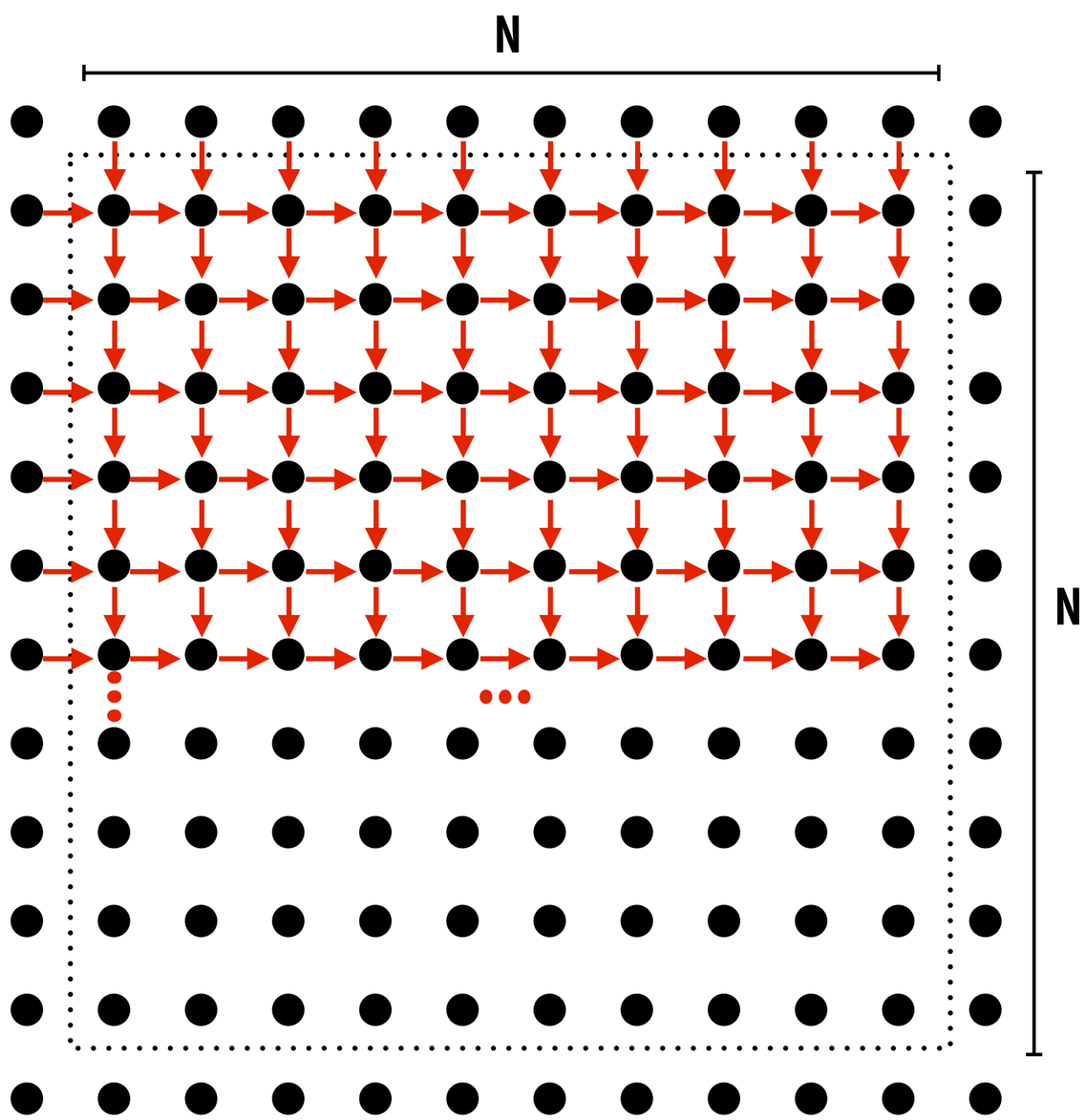
Example from: Culler, Singh, and Gupta

Grid solver algorithm

(generic pseudocode for sequential algorithm is provided below)

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

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

procedure Solve (A)                     /*solve the equation system*/
   float **A;                           /*A is an (n + 2)-by-(n + 2) array*/
begin
   int i, j, done = 0;
   float diff = 0, temp;
   while (!done) do                     /*outermost loop over sweeps*/
      diff = 0;                         /*initialize maximum difference to 0*/
      for i ← 1 to n do                 /*sweep over nonborder points of grid*/
         for j ← 1 to n do
            temp = A[i,j];              /*save old value of element*/
            A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
               A[i,j+1] + A[i+1,j]); /*compute average*/
            diff += abs(A[i,j] - temp);
         end for
      end for
      if (diff/(n*n) < TOL) then done = 1;
   end while
end procedure
```

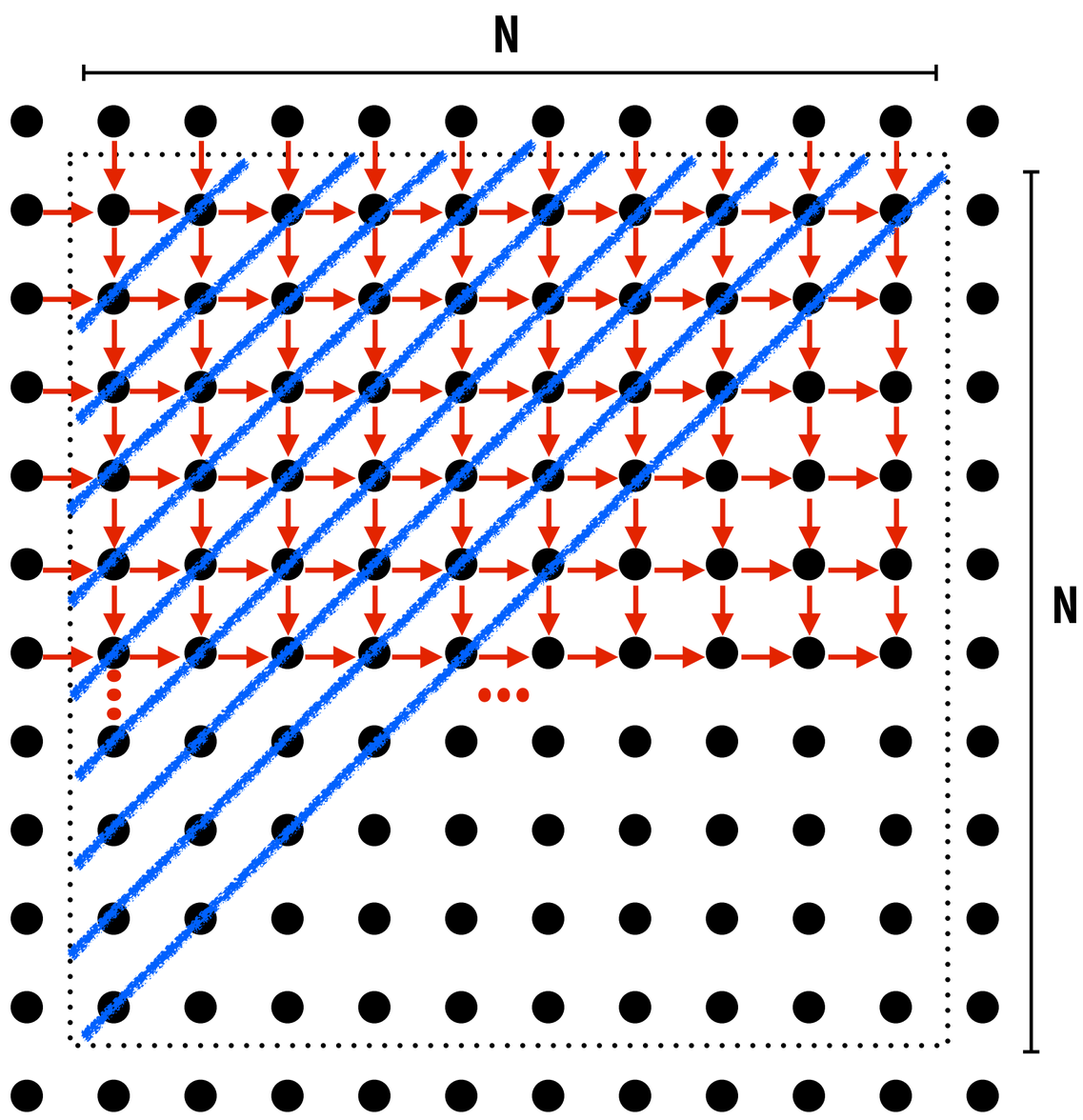
Step 1: identify dependencies (problem decomposition phase)



Each row element depends on element to left.

Each column depends on previous column.

Step 1: identify dependencies (problem decomposition phase)



There is independent work 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

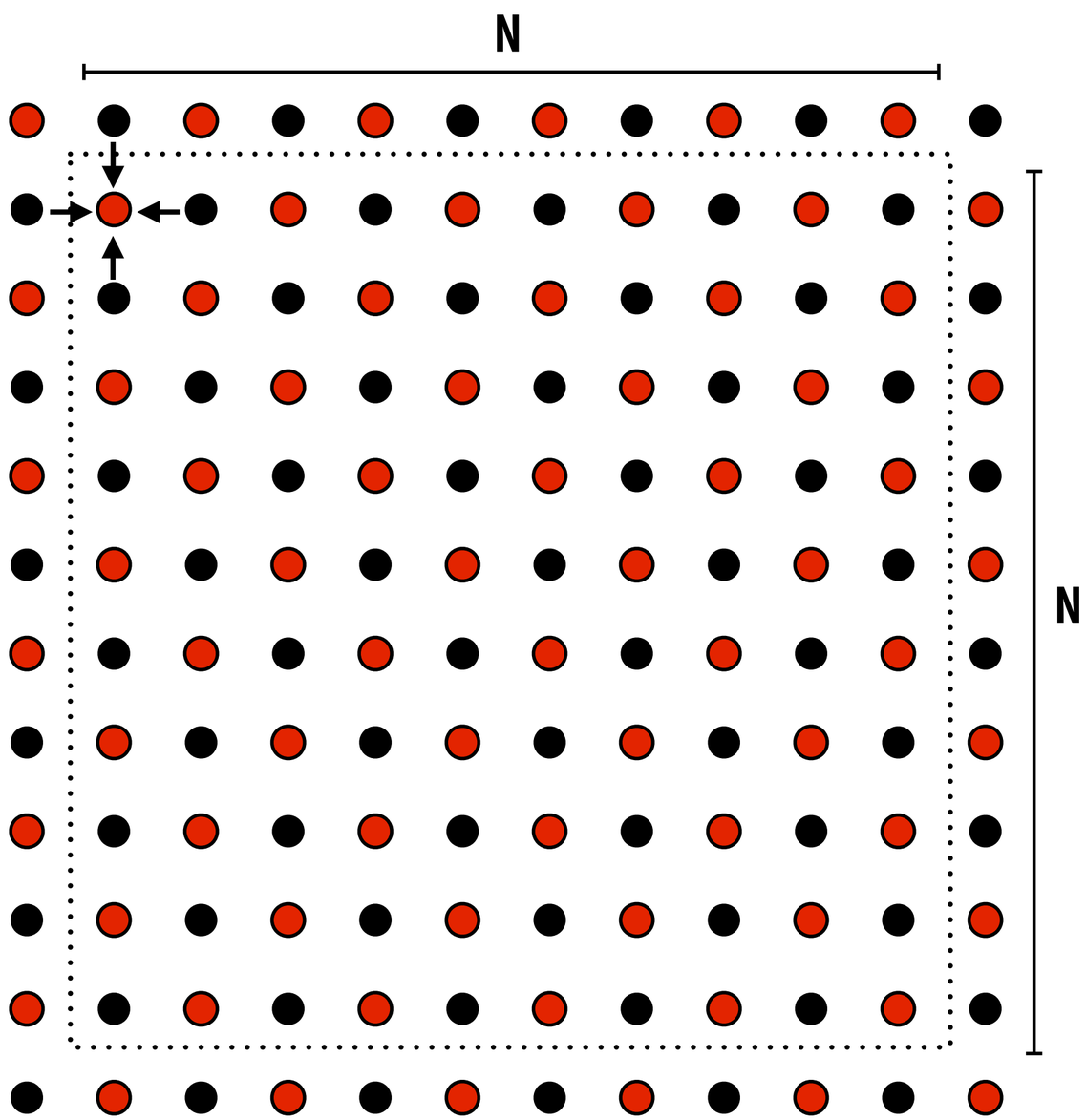
Frequent synchronization (each diagonal)

Key idea: change algorithm to one that is more easily parallelized

- Change the 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

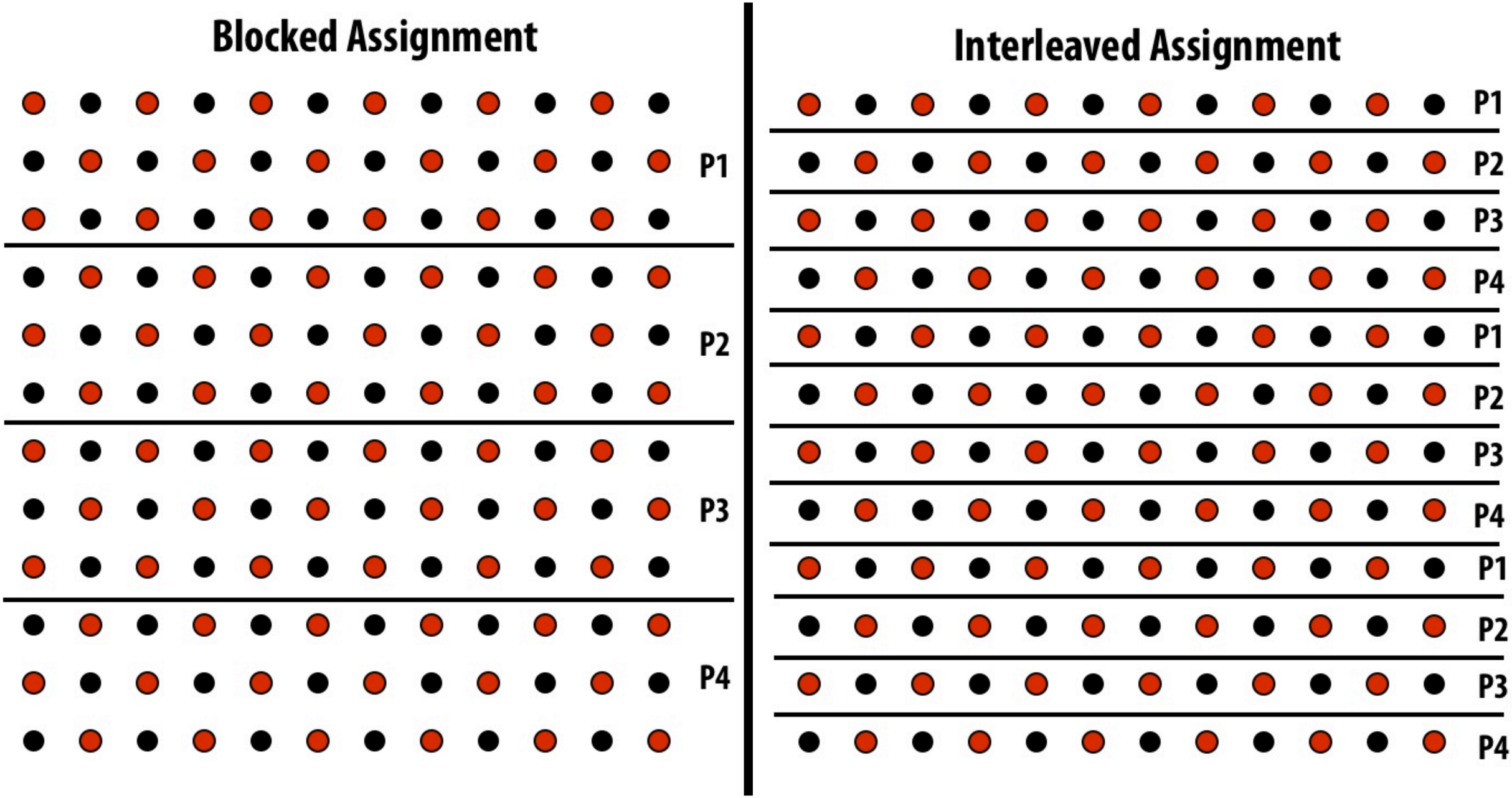


Update all red cells in parallel

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

Repeat until convergence

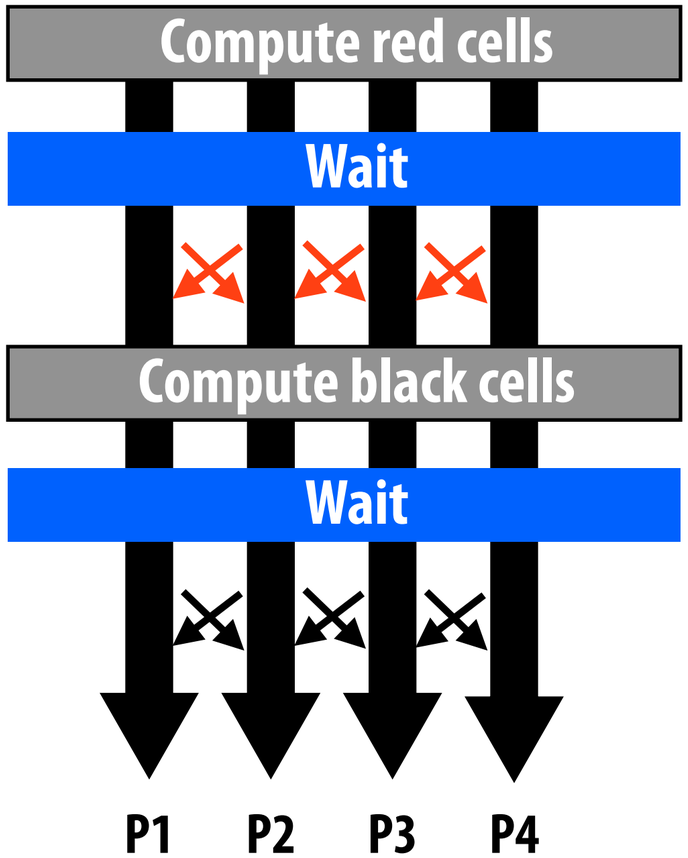
Assignment



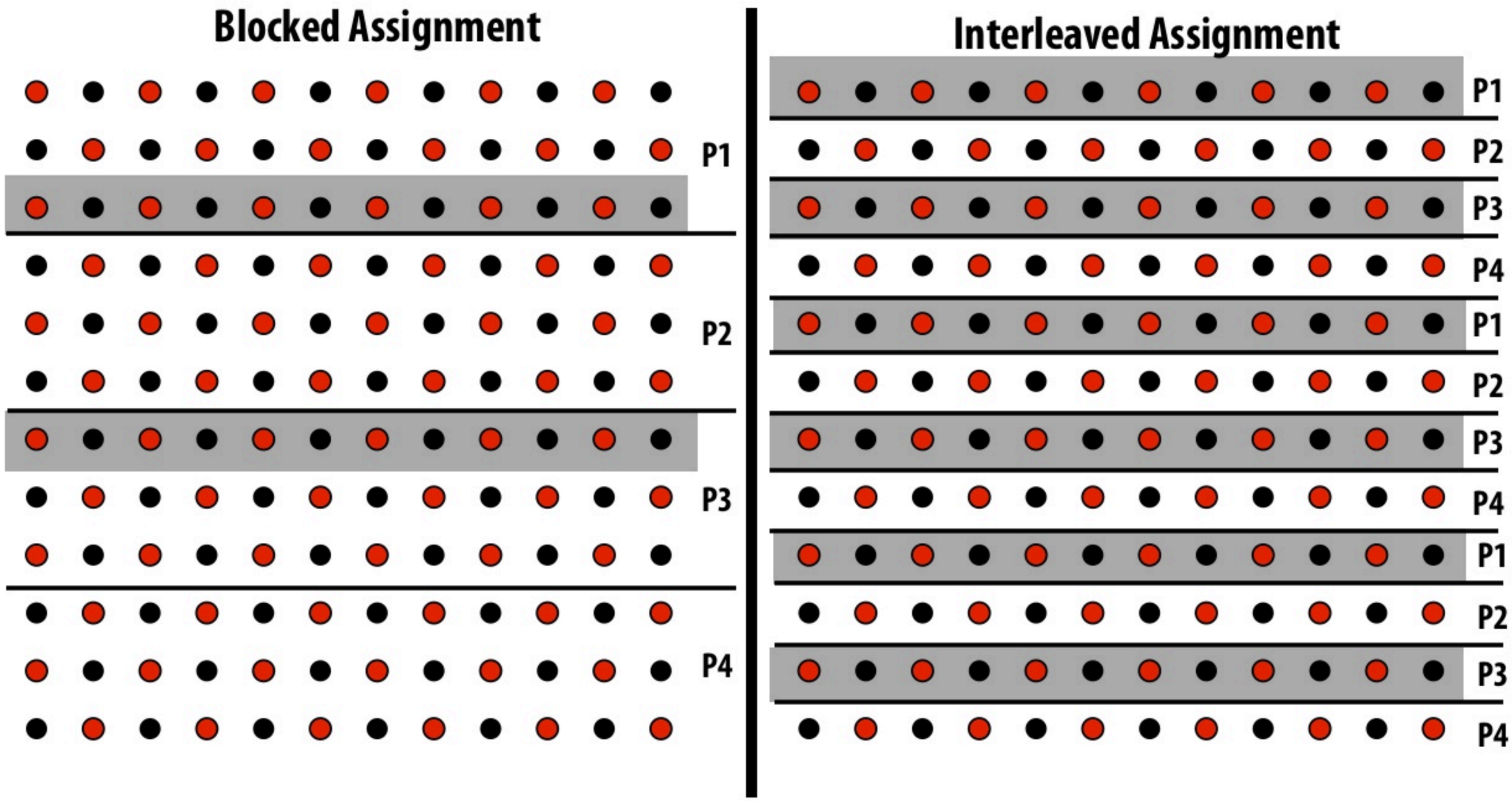
Which is better? Does it matter?

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 pseudocode: just showing red-cell update

```
int n;                          // grid size
bool done = false;
float diff = 0.0;

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

void solve(float** A) {

   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

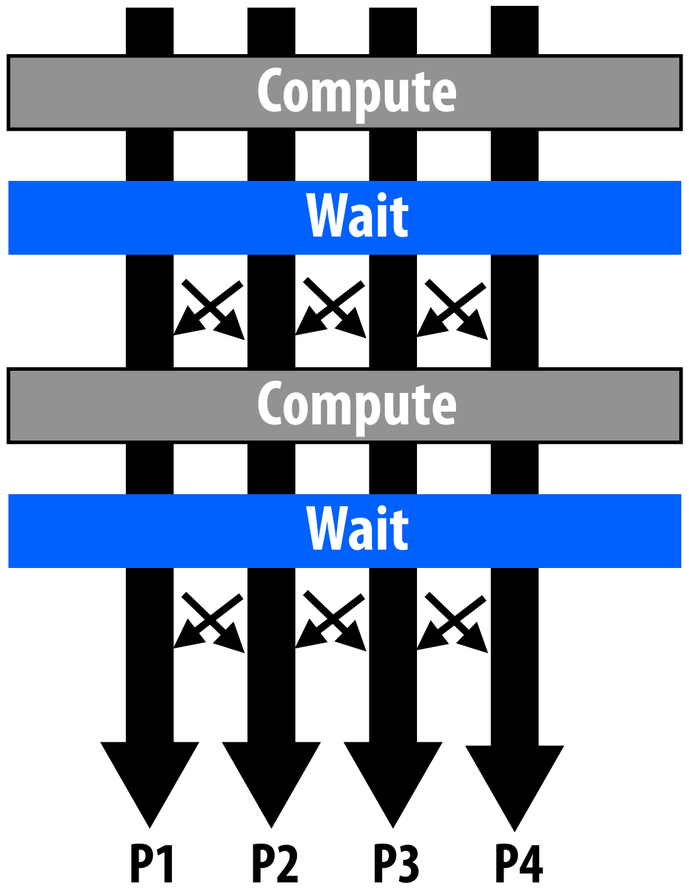
Orchestration: handled by system
(End of for_all block is implicit wait for all workers before returning to sequential control)

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

Shared address space solver

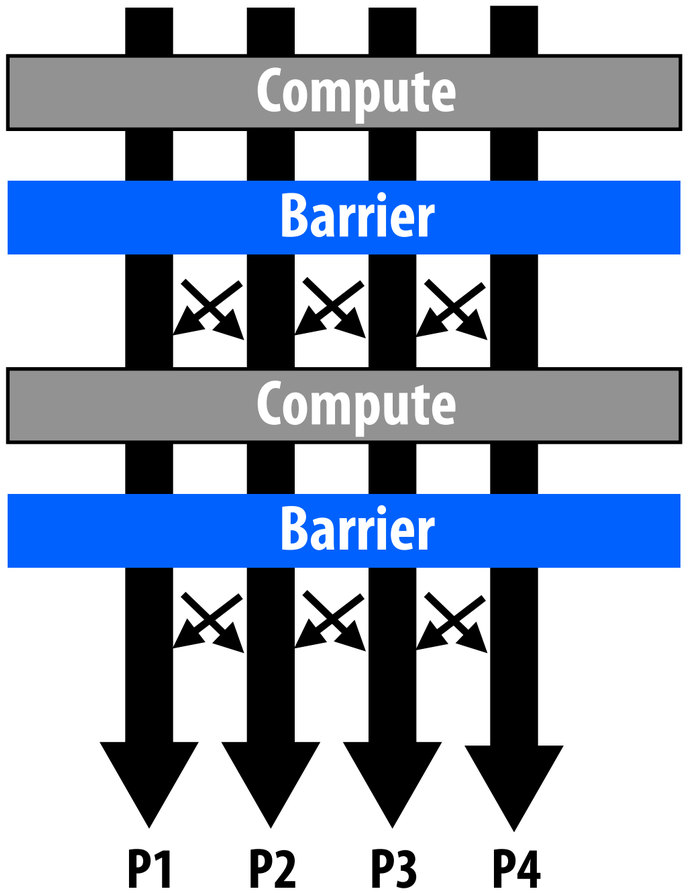
SPMD execution model

- 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)

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

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

each thread computes rows it is responsible for

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**

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 = 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);
   }
}
```

partial sum

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

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

Why are there so many barriers?

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 reduced synchronization! (common parallel programming technique)

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.

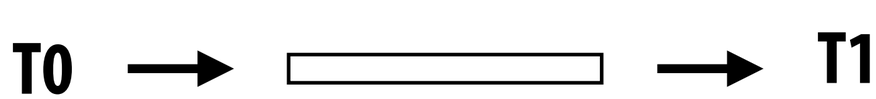
T0

```
// 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, orchestration, mapping
  - We'll talk a lot about making good decisions in each of these phases in coming lectures (in practice, they are very inter-related)

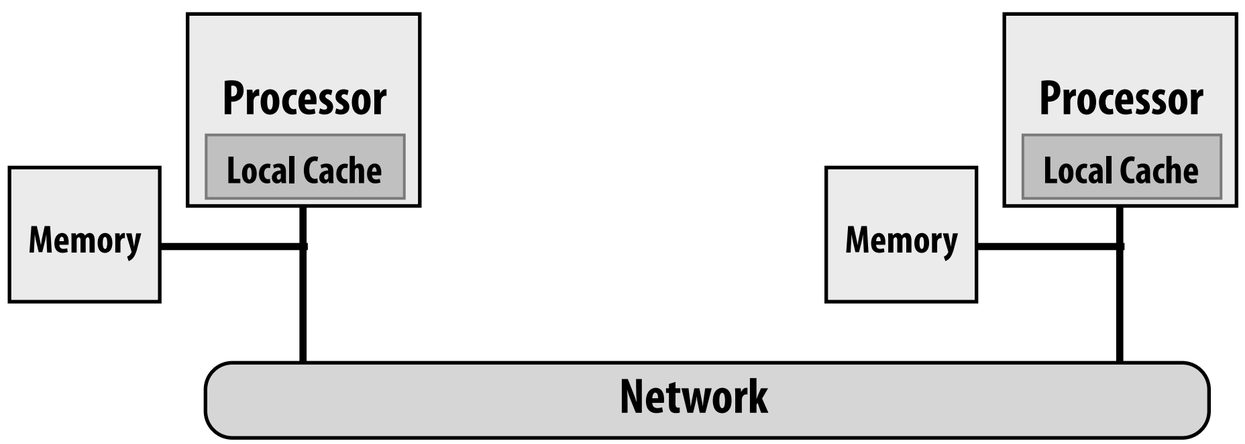
- **Focus today: identifying dependencies**

- **Focus soon: identifying locality**

Message passing model

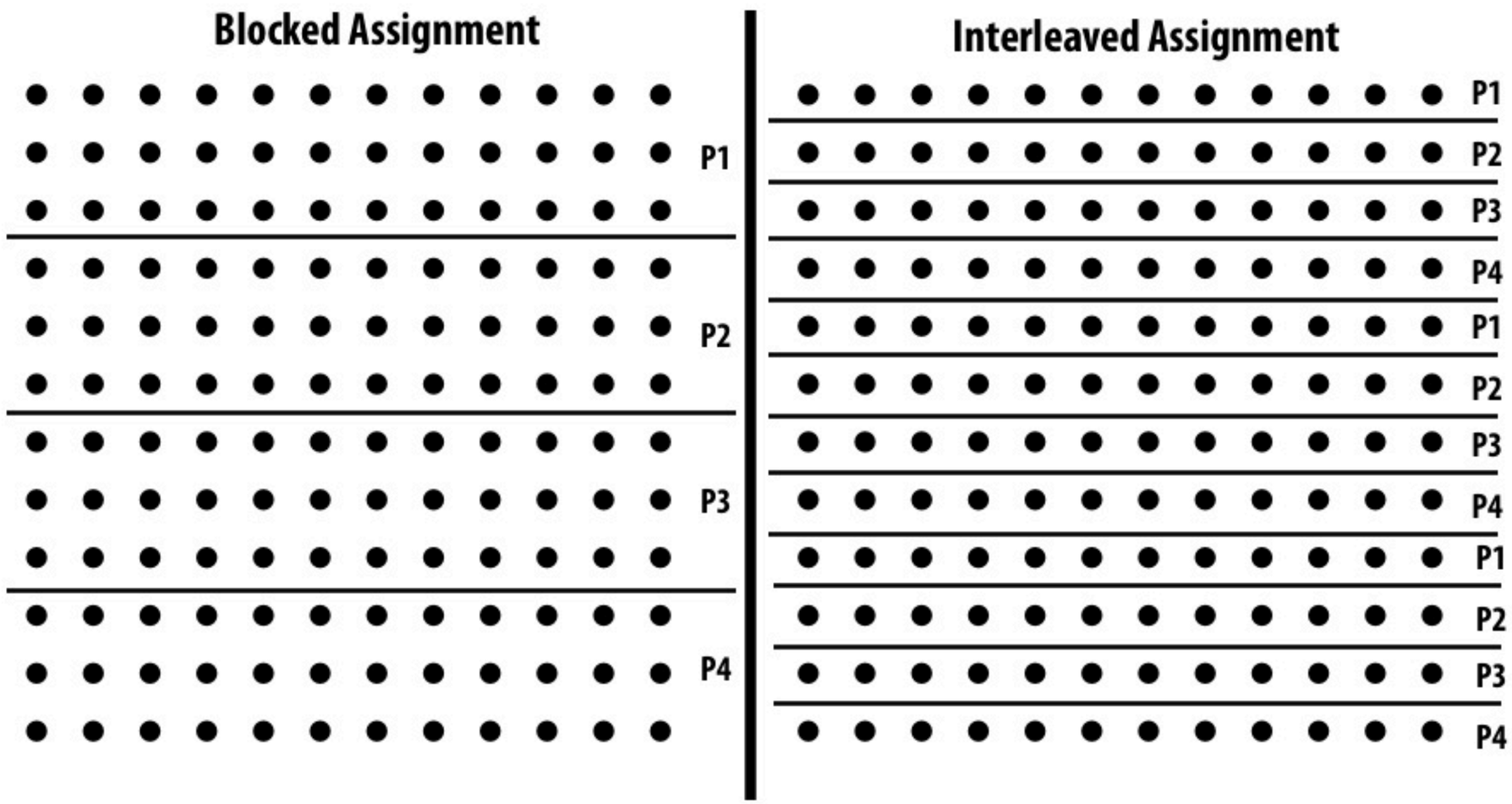
- No shared address space abstraction (i.e., no shared variables)
- Each thread has its own address space
- Threads communicate & synchronize by sending/receiving messages

One possible message passing implementation: cluster of workstations (recall lecture 3)



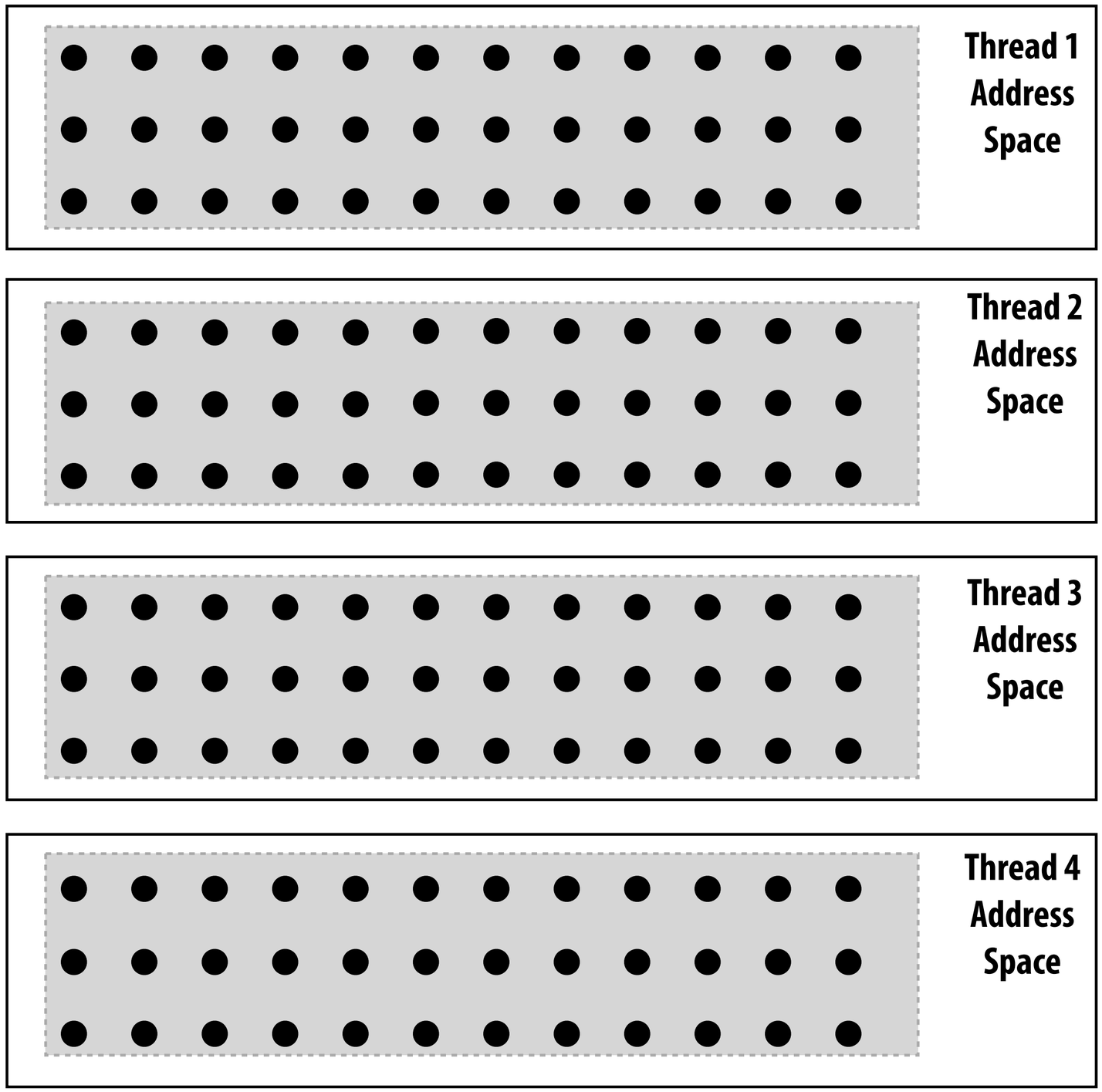
Recall: assignment in a shared address space

- Grid data resided in a single array in shared address space (array was accessible to all threads)
- Assignment partitioned elements to processors to divide up the computation
  - Performance differences
  - Different assignments may yield different amounts of communication due to implementation details (e.g., caching)



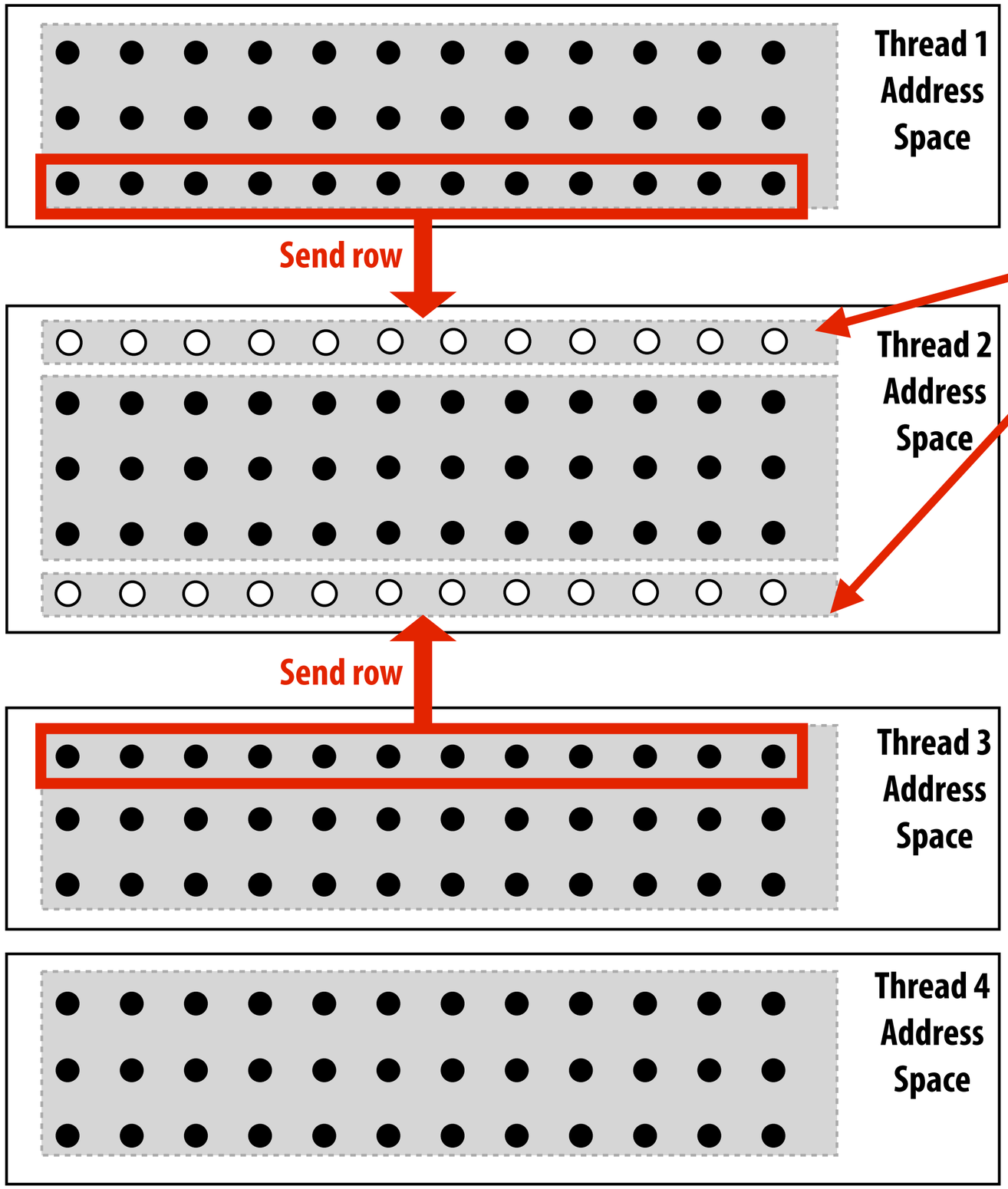
Message passing model

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



Replication required to perform computation

Required for correctness



Example:

Thread 1 and 3 send row to thread 2 (otherwise thread 2 cannot update its local cells)

"Ghost cells":

Grid cells replicated from remote address space.

```
Thread 2 logic:

cell_t ghost_row_top[N+2]; // ghost row storage
cell_t ghost_row_bot[N+2]; // ghost row storage

int bytes = sizeof(cell_t) * (N+2);
recv(ghost_row_top, bytes, pid-1, TOP_MSG_ID);
recv(ghost_row_bot, bytes, pid+1, BOT_MSG_ID);

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

Message passing solver

Note similar structure to shared address space solver, but now communication is explicit

```
int pid, n, b;                          /*process id, matrix dimension and number of
                                         processors to be used*/
float **myA;
main()
begin
    read(n);   read(nprocs);            /*read input matrix size and number of processes*/
    CREATE (nprocs-1, Solve);
    Solve();                            /*main process becomes a worker too*/
    WAIT_FOR_END (nprocs-1);            /*wait for all child processes created to terminate*/
end main

procedure Solve()
begin
    int i,j, pid, n' = n/nprocs, done = 0;
    float temp, tempdiff, mydiff = 0;   /*private variables*/
  myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);
                                        /*my assigned rows of A*/
initialize(myA);                        /*initialize my rows of A, in an unspecified way*/

while (!done) do
    mydiff = 0;                         /*set local diff to 0*/
    if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
    if (pid != nprocs-1) then
      SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
    if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);
    if (pid != nprocs-1) then
      RECEIVE(&myA[n'+1,0],n*sizeof(float), pid+1,ROW);
                                        /*border rows of neighbors have now been copied
                                         into myA[0,*] and myA[n'+1,*]*/
    for i ← 1 to n' do                  /*for each of my (nonghost) rows*/
      for j ← 1 to n do                 /*for all nonborder elements in that row*/
        temp = myA[i,j];
        myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
          myA[i,j+1] + myA[i+1,j]);
        mydiff += abs(myA[i,j] - temp);
      endfor
    endfor
                                        /*communicate local diff values and determine if
                                         done; can be replaced by reduction and broadcast*/
      if (pid != 0) then                /*process 0 holds global total diff*/
        SEND(mydiff,sizeof(float),0,DIFF);
        RECEIVE(done,sizeof(int),0,DONE);
      else                              /*pid 0 does this*/
        for i ← 1 to nprocs-1 do        /*for each other process*/
          RECEIVE(tempdiff,sizeof(float),*,DIFF);
        mydiff += tempdiff;             /*accumulate into total*/
      endfor
      if (mydiff/(n*n) < TOL) then      done = 1;
        for i ← 1 to nprocs-1 do        /*for each other process*/
          SEND(done,sizeof(int),i,DONE);
        endfor
    endif
  endwhile
end procedure
```

Send and receive ghost rows

Perform computation

All threads send local mydiff to thread 0

Thread 0 computes termination, predicate sends result back to all other threads

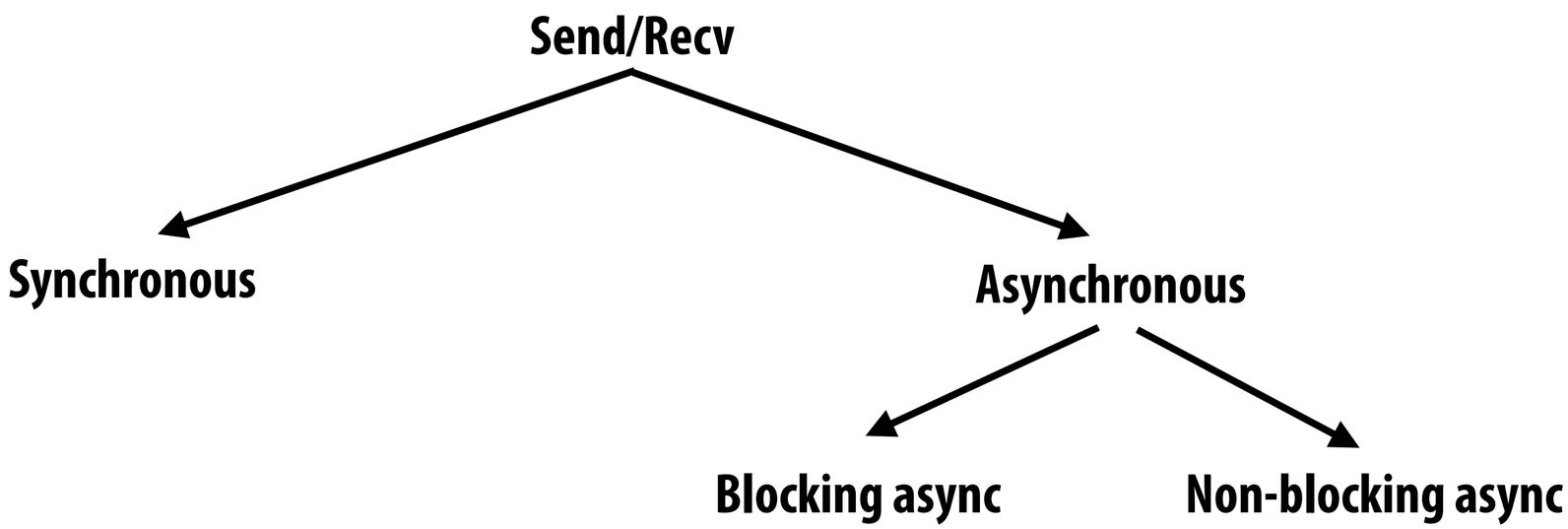
Notes on message passing example

- **Computation**
  - Array indexing is relative to local address space (not global grid coordinates)
- **Communication:**
  - Performed through messages
  - En masse, not element at a time. Why?
- **Synchronization:**
  - Performed through sends and receives
  - Think of how to implement mutual exclusion, barriers, flags using messages
- **For convenience: message passing libraries often include higher-level primitives (implemented using send and receive)**

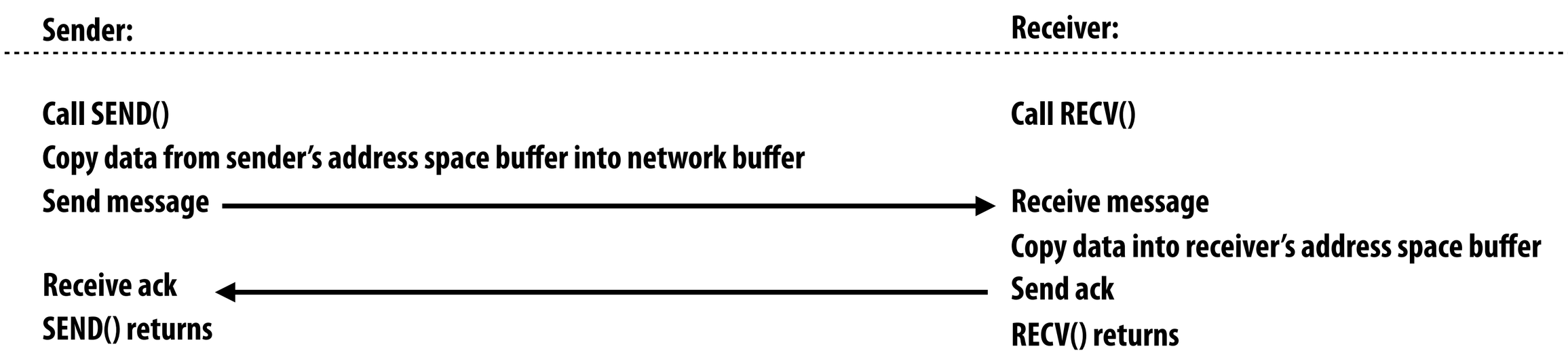
```
REDUCE(0,mydiff,sizeof(float),ADD);
if (pid == 0) then
   if (mydiff/(n*n) < TOL) then done = 1;
endif
   BROADCAST(0,done,sizeof(int),DONE);
```

Alternative solution using reduce/broadcast constructs

Send and receive variants



- **Synchronous:**
  - SEND: call returns when message data resides in address space of receiver (and sender has received acknowledgement that this is the case)
  - RECV: call returns when data from message copied into address space of receiver and acknowledgement sent)



As implemented on previous slide, if our message passing solver uses blocking send/recv it would deadlock!

Why?

How can we fix it?

(while still using blocking send/recv)

Message passing solver

This code will deadlock. Why?

```
1. int pid, n, b;                          /*process id, matrix dimension and number of
                                             processors to be used*/
2. float **myA;
3. main()
4. begin
5.     read(n);   read(nprocs);            /*read input matrix size and number of processes*/
8a.    CREATE (nprocs-1, Solve);
8b.    Solve();                            /*main process becomes a worker too*/
8c.    WAIT_FOR_END (nprocs-1);            /*wait for all child processes created to terminate*/
9. end main

10.  procedure Solve()
11.  begin
13.    int i,j, pid, n' = n/nprocs, done = 0;
14.    float temp, tempdiff, mydiff = 0;    /*private variables*/
6.   myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);
                                           /*my assigned rows of A*/
7. initialize(myA);                        /*initialize my rows of A, in an unspecified way*/

15.  while (!done) do
16.    mydiff = 0;                         /*set local diff to 0*/
16a.   if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
16b.   if (pid != nprocs-1) then
         SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
16c.   if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);
16d.   if (pid != nprocs-1) then
         RECEIVE(&myA[n'+1,0],n*sizeof(float), pid+1,ROW);
                                           /*border rows of neighbors have now been copied
                                             into myA[0,*] and myA[n'+1,*]*/
17.    for i ← 1 to n' do                  /*for each of my (nonghost) rows*/
18.      for j ← 1 to n do                 /*for all nonborder elements in that row*/
19.        temp = myA[i,j];
20.        myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21.          myA[i,j+1] + myA[i+1,j]);
22.        mydiff += abs(myA[i,j] - temp);
23.      endfor
24.    endfor
                                           /*communicate local diff values and determine if
                                             done; can be replaced by reduction and broadcast*/
25a.     if (pid != 0) then                /*process 0 holds global total diff*/
25b.       SEND(mydiff,sizeof(float),0,DIFF);
25c.       RECEIVE(done,sizeof(int),0,DONE);
25d.     else                              /*pid 0 does this*/
25e.       for i ← 1 to nprocs-1 do        /*for each other process*/
25f.         RECEIVE(tempdiff,sizeof(float),*,DIFF);
25g.       mydiff += tempdiff;             /*accumulate into total*/
25h.     endfor
25i      if (mydiff/(n*n) < TOL) then      done = 1;
25j.       for i ← 1 to nprocs-1 do        /*for each other process*/
25k.         SEND(done,sizeof(int),i,DONE);
25l.       endfor
25m.   endif
26.  endwhile
27.  end procedure
```

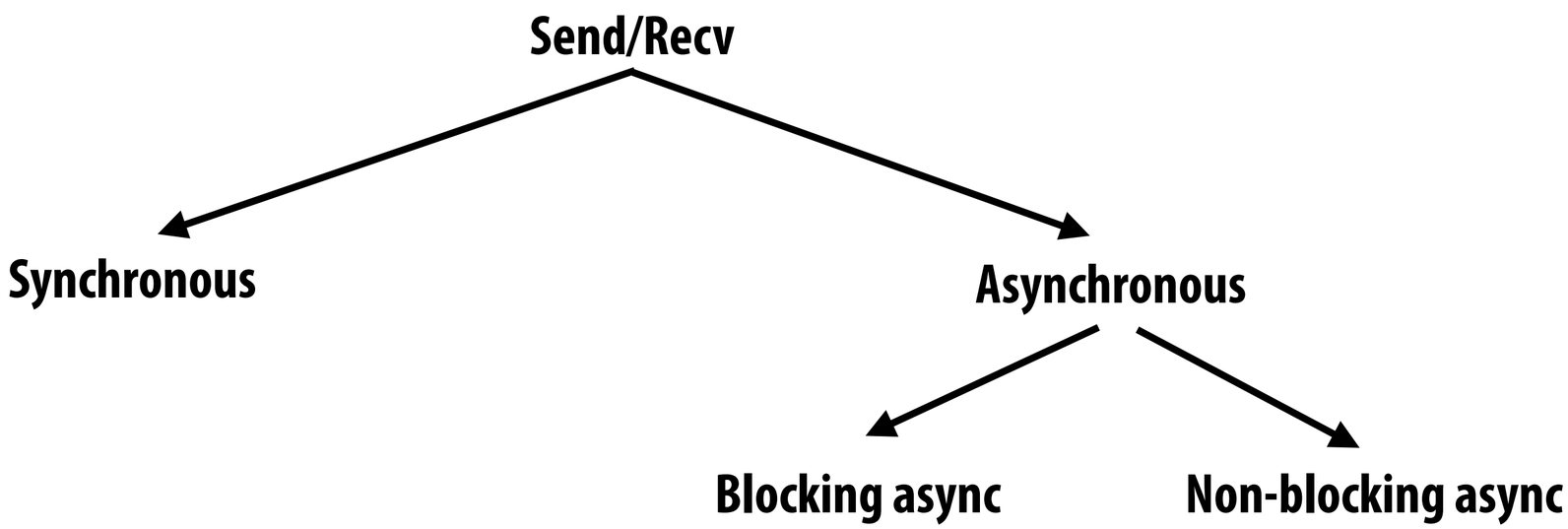
Send and receive ghost rows

Perform computation

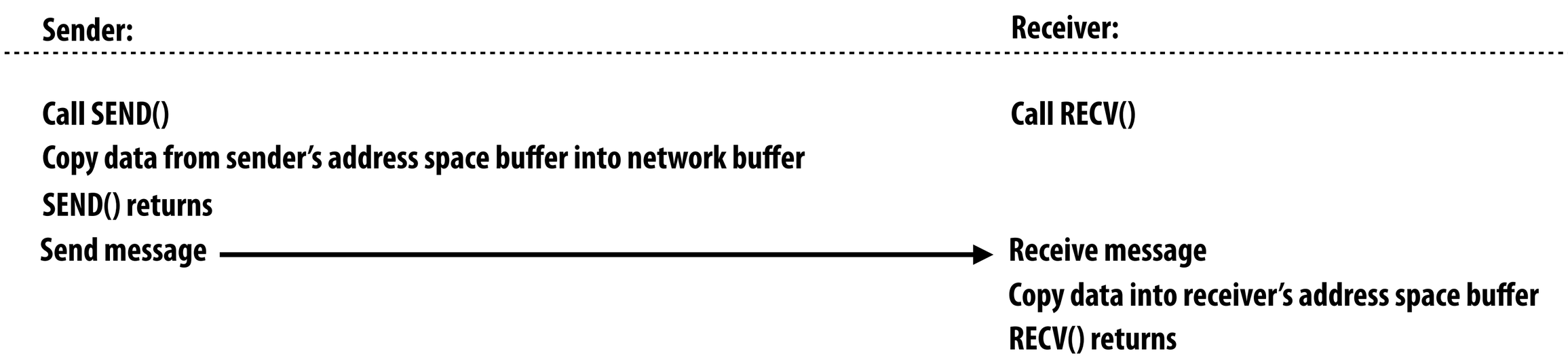
All threads send local mydiff to thread 0

Thread 0 computes termination, predicate sends result back to all other threads

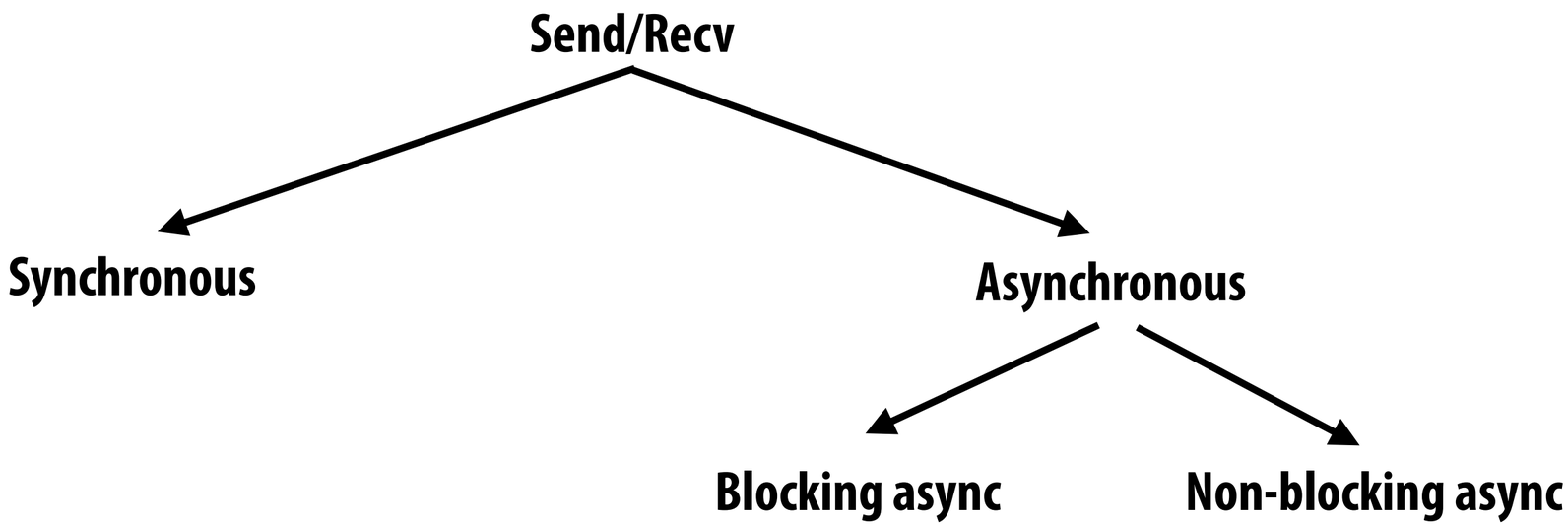
Send and receive variants



- **Async blocking:**
  - SEND: call copies data from address space into system buffers, then returns
    - Does not guarantee message has been received (or even sent)
  - RECV: call returns when data copied into address space, but no ack sent



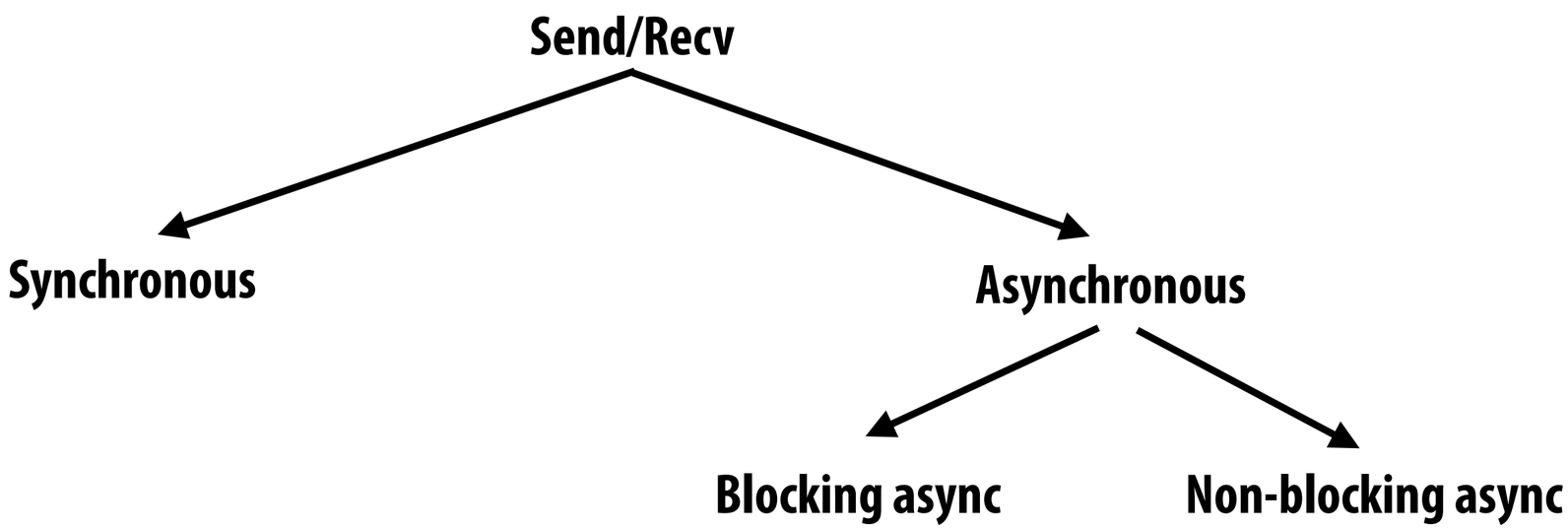
Send and receive variants



- **Async non-blocking: ("non-blocking")**
  - SEND: call returns immediately. Buffer provided to SEND cannot be touched by called through since message processing occurs concurrently
  - RECV: call posts intent to receive, returns immediately
  - Use SENDPROBE, RECVPROBE to determine actual send/receipt status

Sender:	Receiver:
Call SEND(local_buf)	Call RECV(recv_local_buf)
SEND() returns	RECV() returns
Copy data from local_buf into network buffer	
Send message →	Receive message
	Copy data into recv_local_buf
Call SENDPROBE // if sent, now safe for thread to modify local_buf	Call RECVPROBE
	// if received, now safe for thread
	// to access recv_local_buf

Send and receive variants



The variants of send/recv provide different levels of programming complexity / opportunity to optimize performance

Solver implementation in THREE programming models

1. Data-parallel 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

2. Shared address space model

- Synchronization:
  - Mutual exclusion required for shared variables
  - Barriers used to express dependencies (between phases of computation)
- Communication
  - Implicit in loads/stores to shared variables

3. Message passing model

- Synchronization:
  - Implemented via messages
  - Mutual exclusion by default: no shared data structures
- Communication:
  - Explicit communication via send/recv needed for parallel program correctness
  - Bulk communication: communicated entire rows, not single elements
  - Several variants on send/recv semantics

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**