Lecture 5:
Parallel Programming Basics (part II)
Programming for Performance (part I)

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

Assignments

- **Assignment 1 due tonight @ midnight**
  - No grace days

- **Assignments 2, 3, 4**
  - Can work in pairs (optional)
  - 3 total grace days per group

- **Final project**
  - No grace days

Recall: shared address space solver

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

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

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

      BARRIER(bar1, nprocs);
   endwhile
end procedure
```

Why are there so many barriers?

Shared address space solver: one barrier

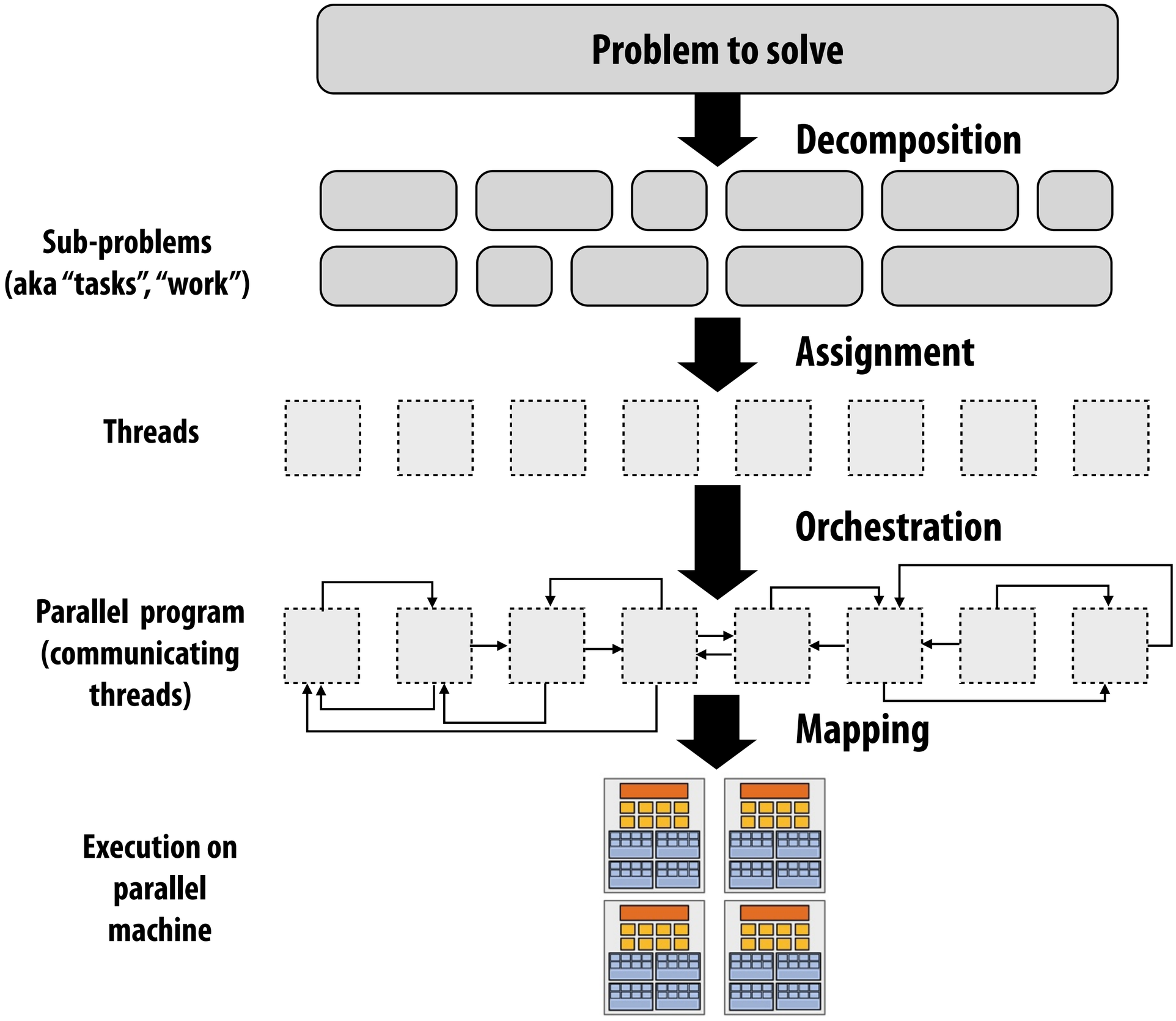
```
float diff[3];  // global diff
float mydiff;   // thread local variable
int index = 0;  // thread local variable

LOCKDEC(diff_lock);
BARDEC(bar);

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

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

Idea:

Remove dependencies by using different diff variables in successive loop iterations

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

Steps in creating a parallel program



Solver implementation in two programming models

- **Data-parallel programming model**
  - Synchronization:
    - Single logical thread of control, but iterations of `forall` loop can be parallelized (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

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

Solver implementation in two programming models

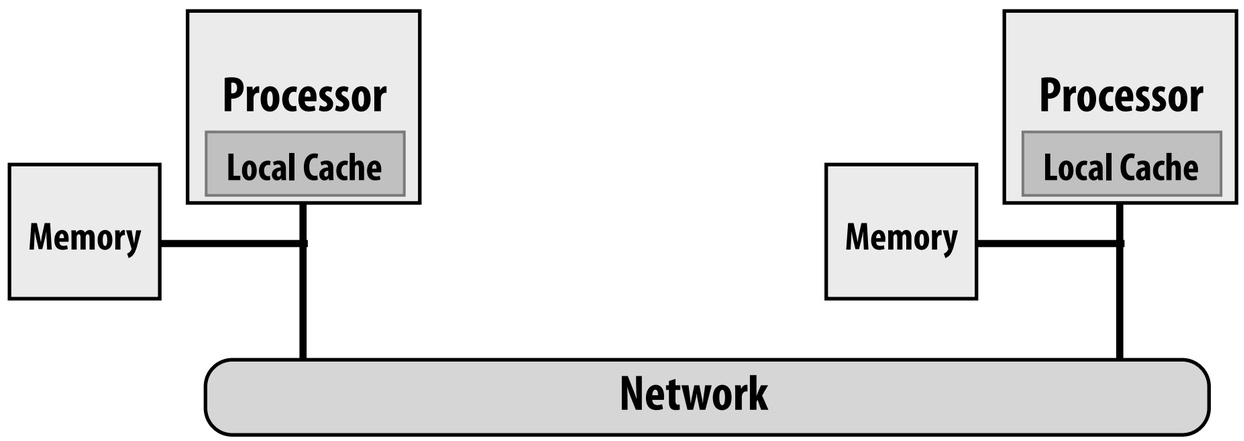
- **Data-parallel programming model**
  - Synchronization:
    - Single logical thread of control, but iterations of `forall` loop can be parallelized (barrier at end of outer `forall` loop body)
  - Communication
    - Implicit in loads and stores (like shared address space)
    - Special built-in primitives: 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

Today: message passing model

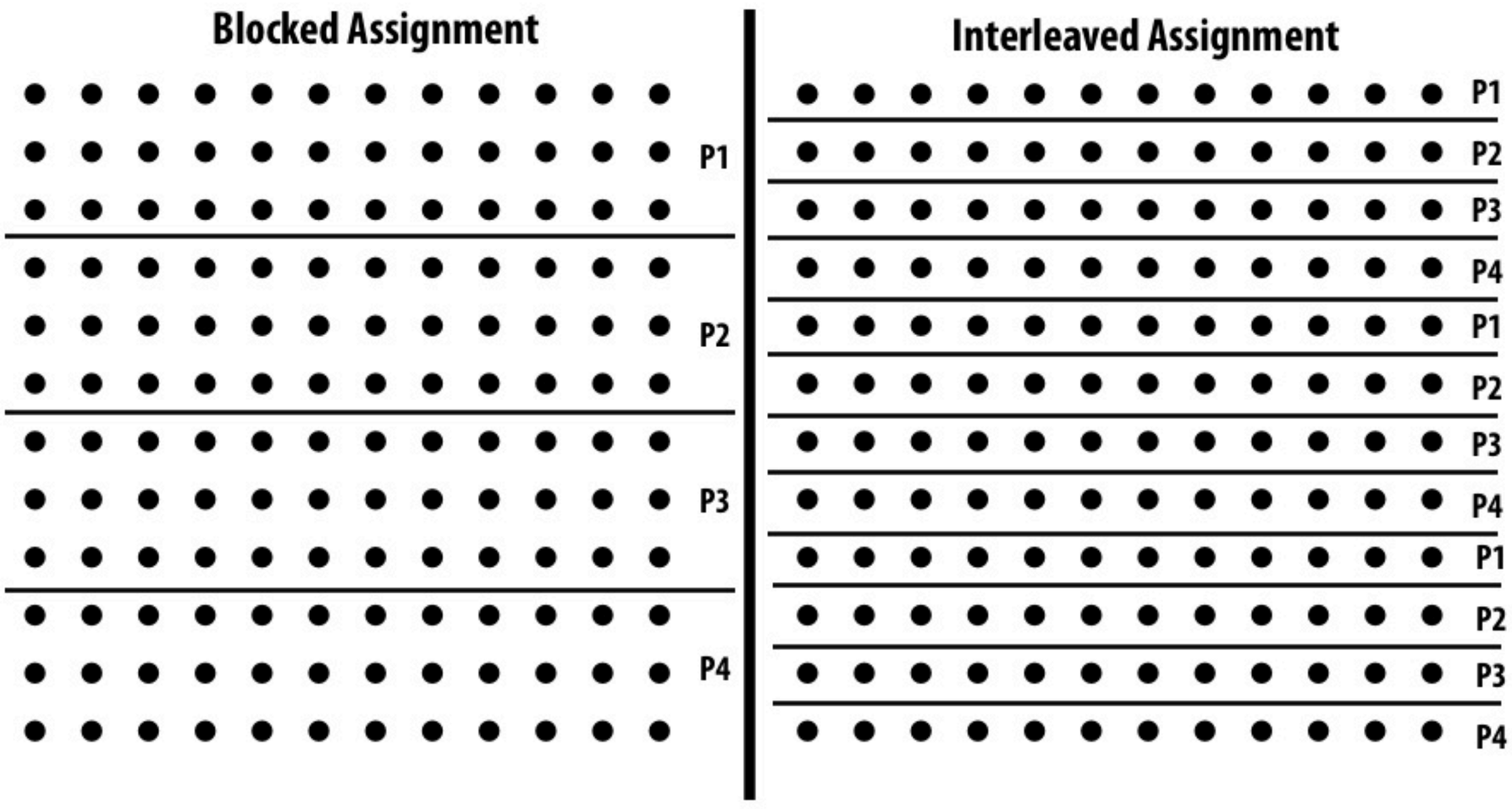
- No shared address space abstraction (i.e., no shared variables)
- Each thread has it's own address space
- Threads communicate & synchronize by sending/receiving messages

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



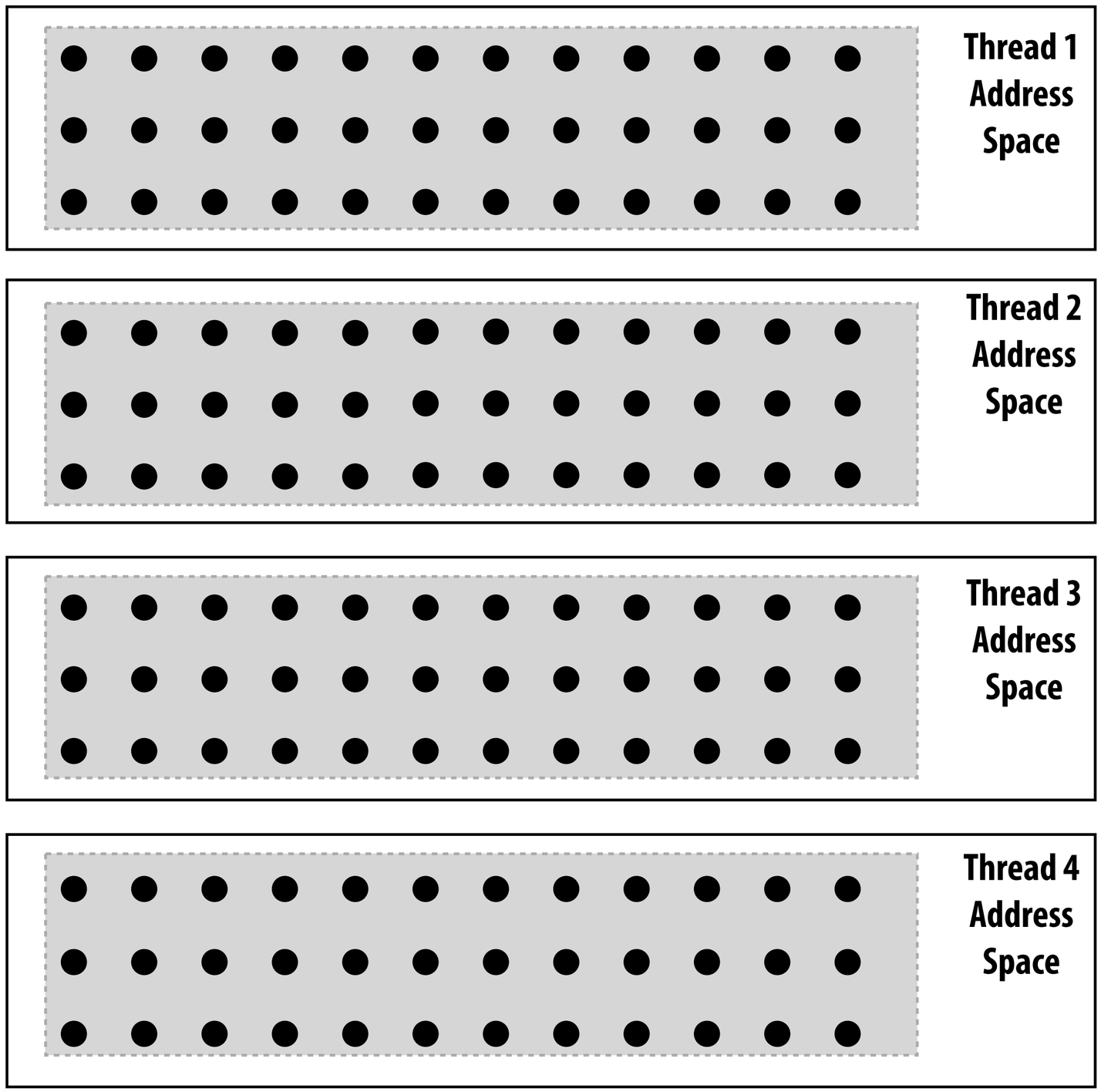
Last time: assignment in a shared address space

- 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)



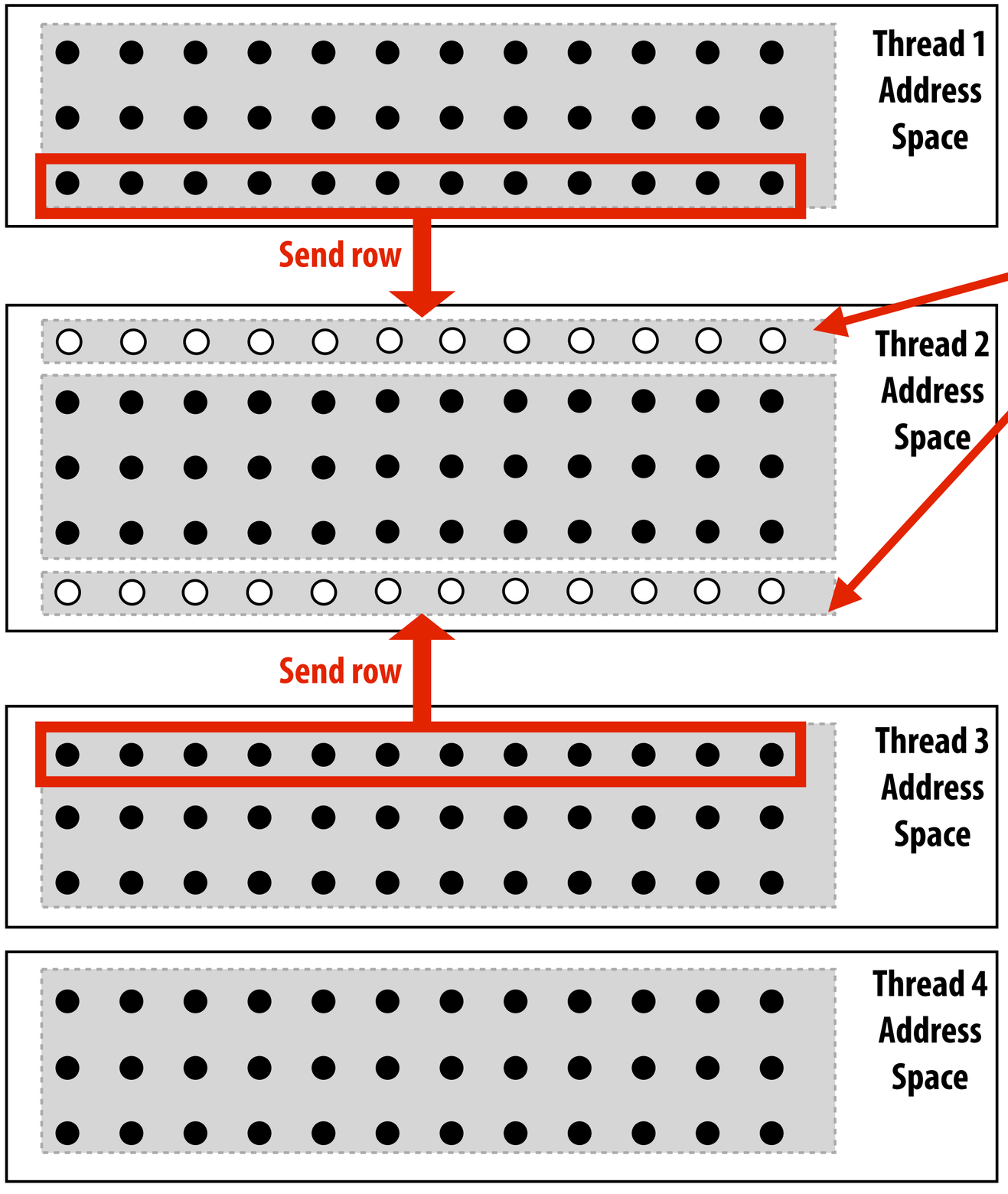
Message passing model

- 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

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

```
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
```

Notes on message passing example

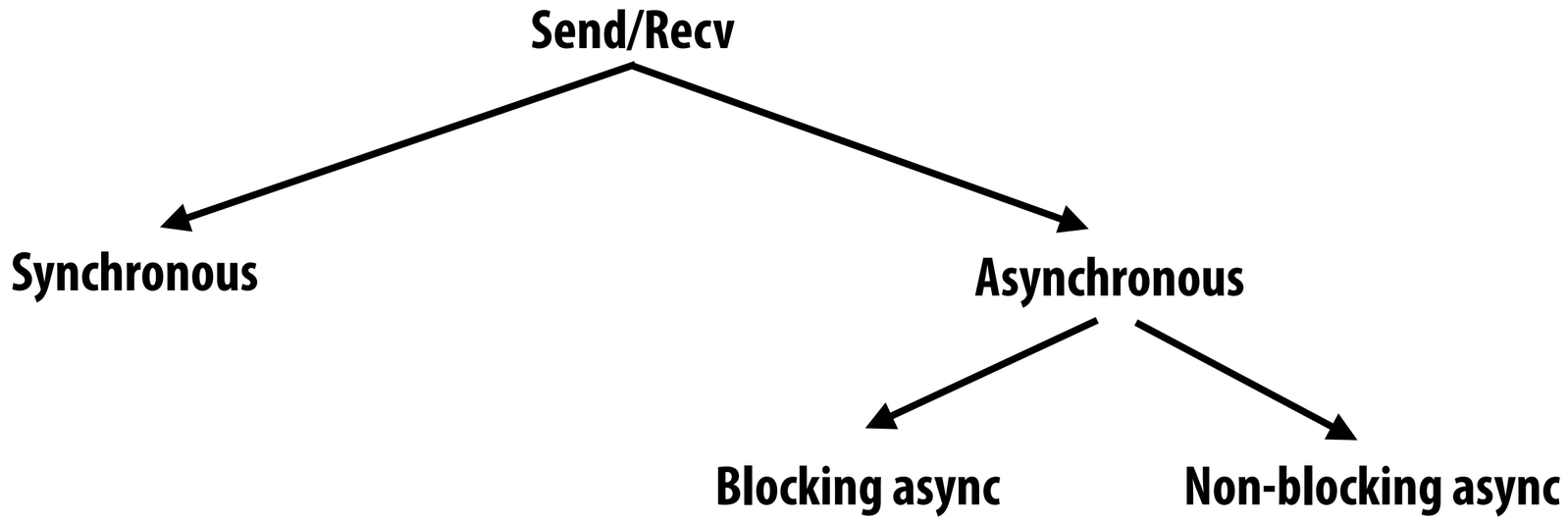
- **Computation**
  - Array indexing is relative to local address space (not global grid coordinates)
- **Communication:**
  - Performed through messages
  - En masse, not element at a time. Why?
- **Synchronization:**
  - Performed through sends and receives
  - Think of how to implement mutual exclusion, barriers, flags using messages

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

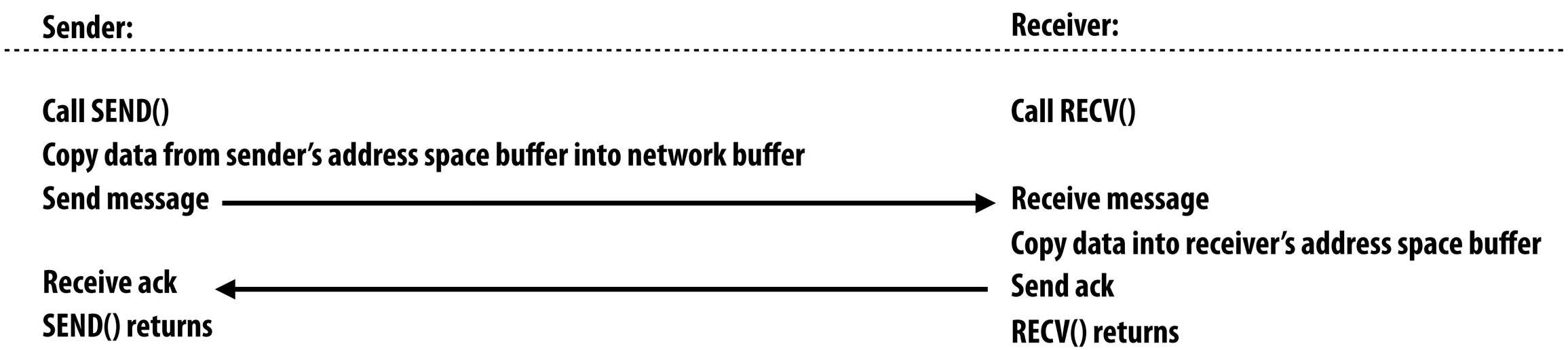
```
REDUCE(0,mydiff,sizeof(float),ADD);
if (pid == 0) then
   if (mydiff/(n*n) < TOL) then done = 1;
endif
   BROADCAST(0,done,sizeof(int),DONE);
```

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 ack that this is the case)**
  - **RECV: call returns when data from message copied into address space of receiver and ack 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

Send and receive ghost rows

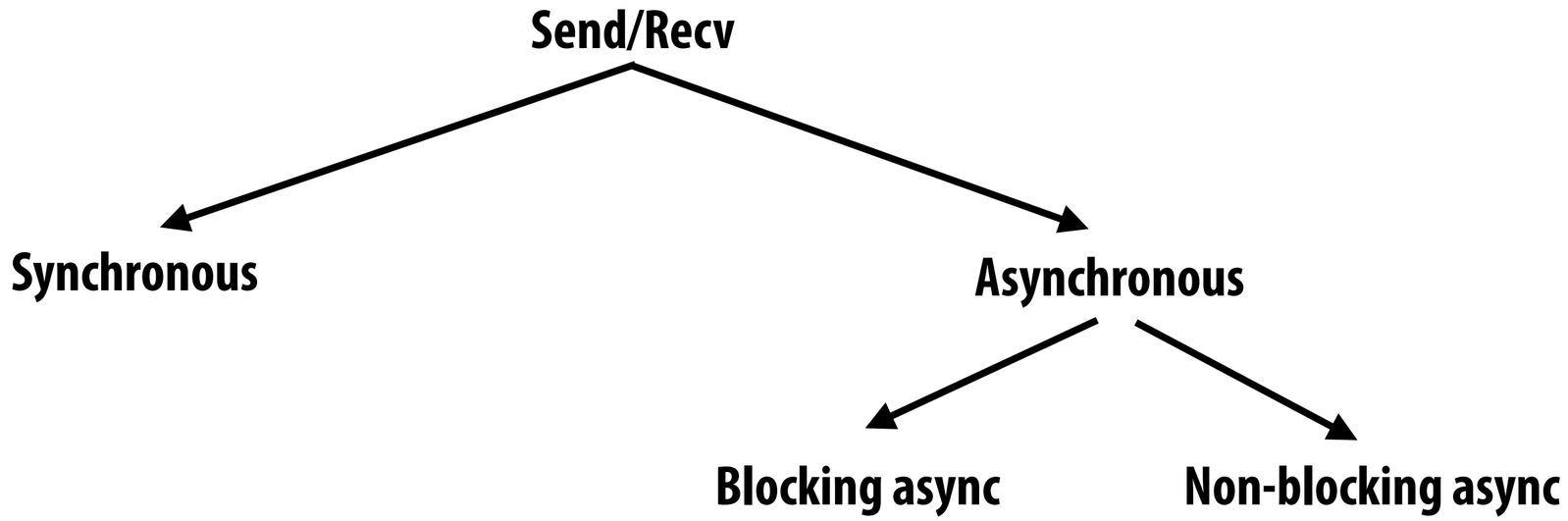
Perform computation

All threads send local mydiff to thread 0

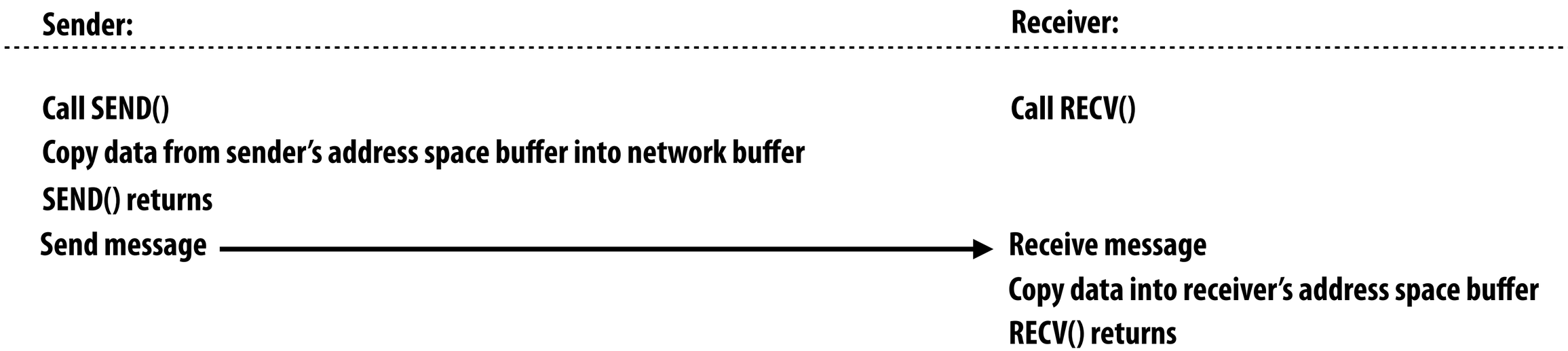
Thread 0 computes termination, predicate sends result back to all other threads

```
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
```

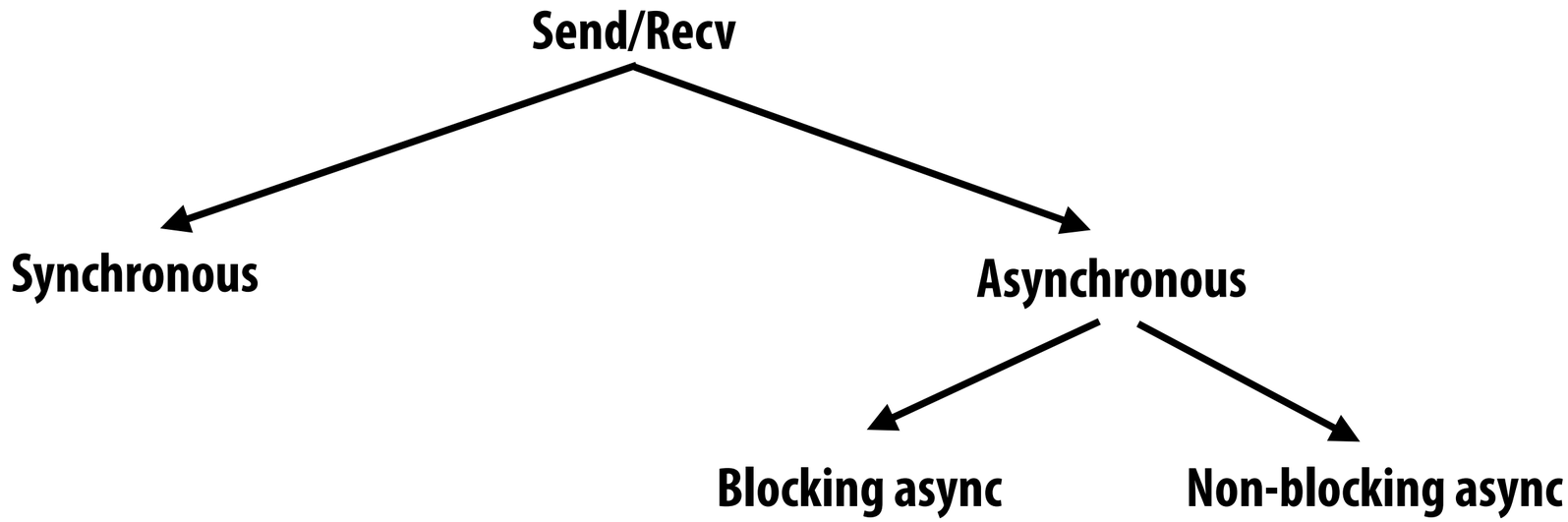
Send and receive variants



- **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:**

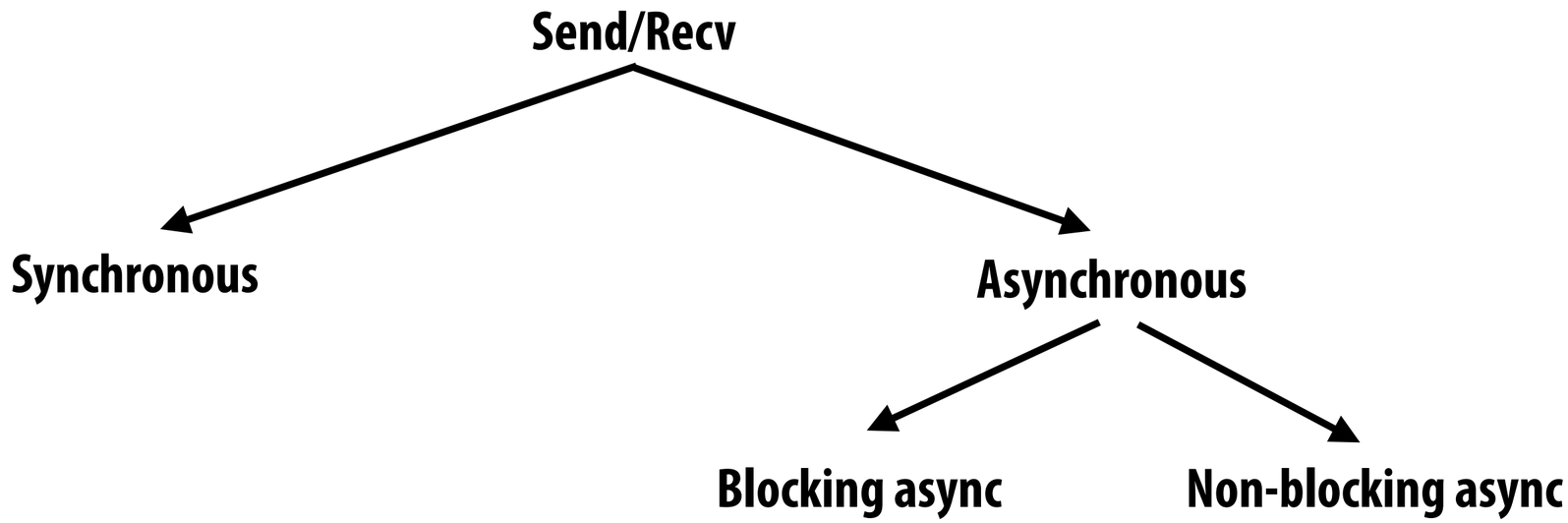
Call SEND(local_buf)
SEND() returns

Copy data from local_buf into network buffer
Send message ⟶ **Receive message**
Copy data into recv_local_buf
Call RECV(recv_local_buf)
RECV() returns

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 (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

Optimizing parallel program performance

(how to be l33t)

(CMU 15-418, Spring 2012)

Programming for performance

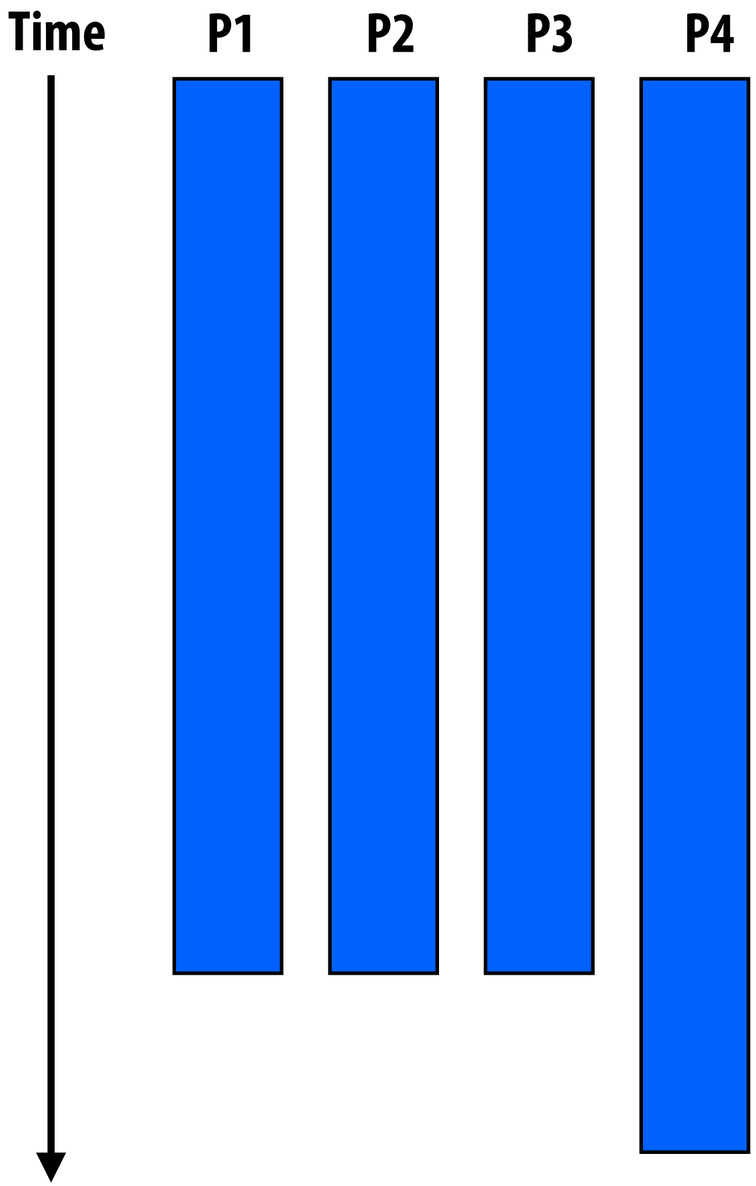
- Optimizing the performance of parallel programs is an iterative process of refining choices for decomposition, assignment, and orchestration...

- Key goals (that are at odds with each other)
  - Balance workload onto available execution resources
  - Reduce communication (to avoid stalls)
  - Reduce extra work done to determine/manage assignment

- We are going to talk about a rich space of techniques
  - TIP #1: Always do the simple thing first, then measure/analyze
  - "It scales" = it scales as much as you need it too

Balancing the workload

Ideally all processors are computing all the time during program execution
(they are computing simultaneously, and they finish their portion of the work at the same time)



Recall Amdahl's Law:
Only small amount of load imbalance can significantly bound maximum speedup

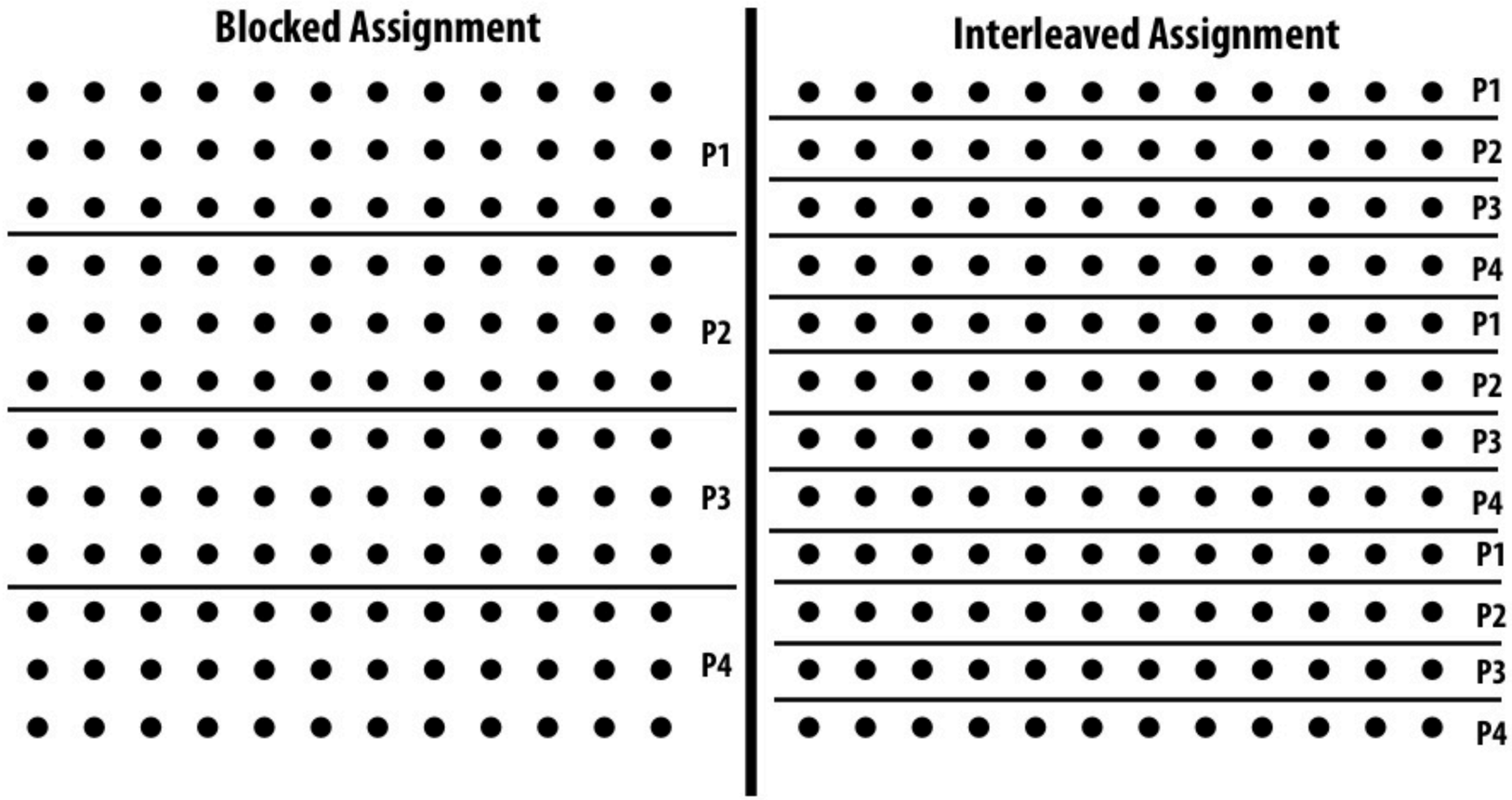
P4 does 20% more work → P4 takes 20% longer to complete

→ 20% of parallel program runtime is essentially serial execution

(clarification: work in serialized section here is about 5% of a sequential program's execution time: S=.05 in Amdahl's law eqn)

Static assignment

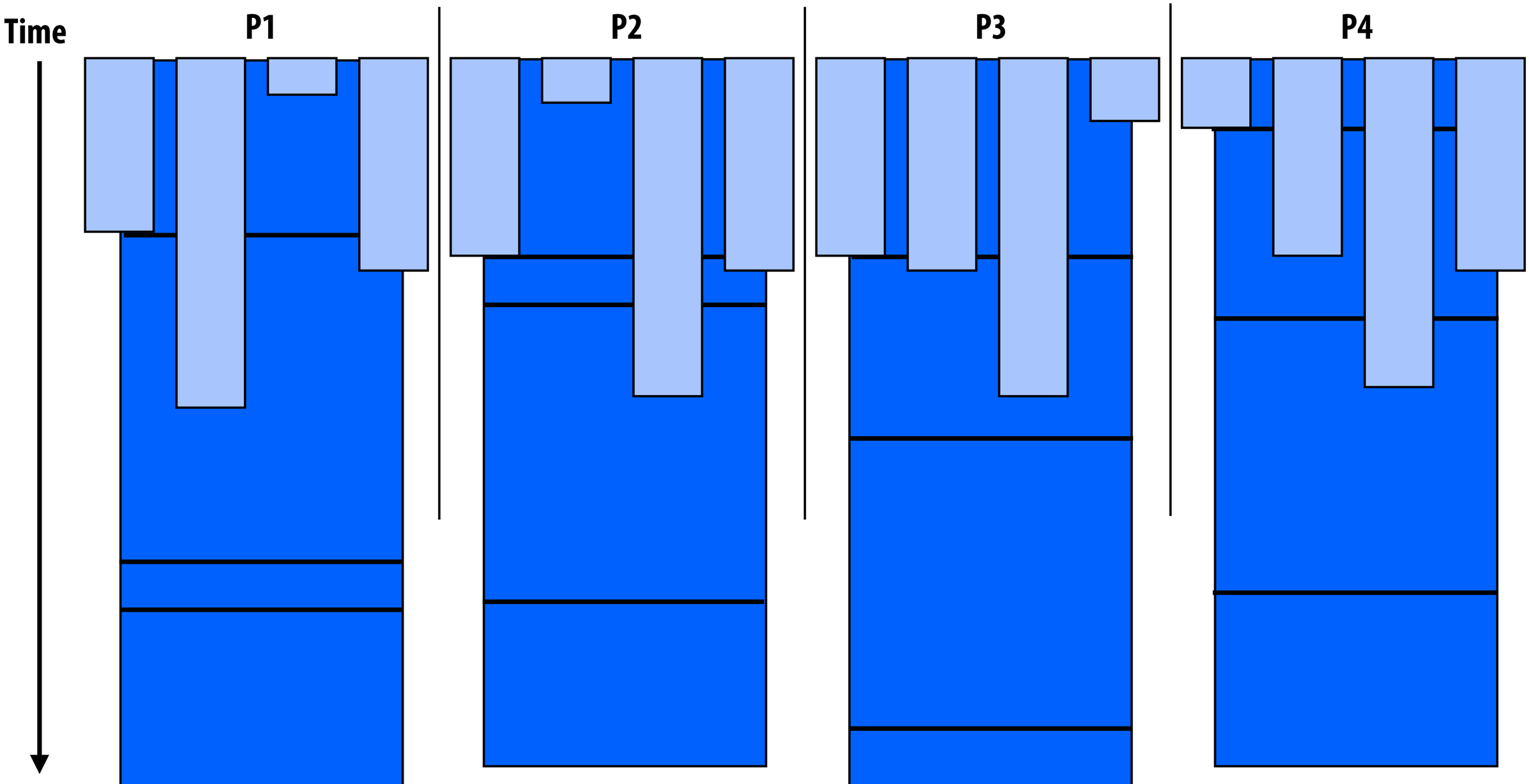
- Assignment of work to threads is pre-determined
  - Not necessarily compile-time (assignment algorithm may depend on runtime parameters such as input data size, number of threads, etc.)
- Recall solver example: assign equal number of grid cells to each thread
  - We discussed blocked and interleaved static assignments



- Good properties: simple, low runtime overhead
  (here: extra work to implement mapping is a little bit of indexing math)

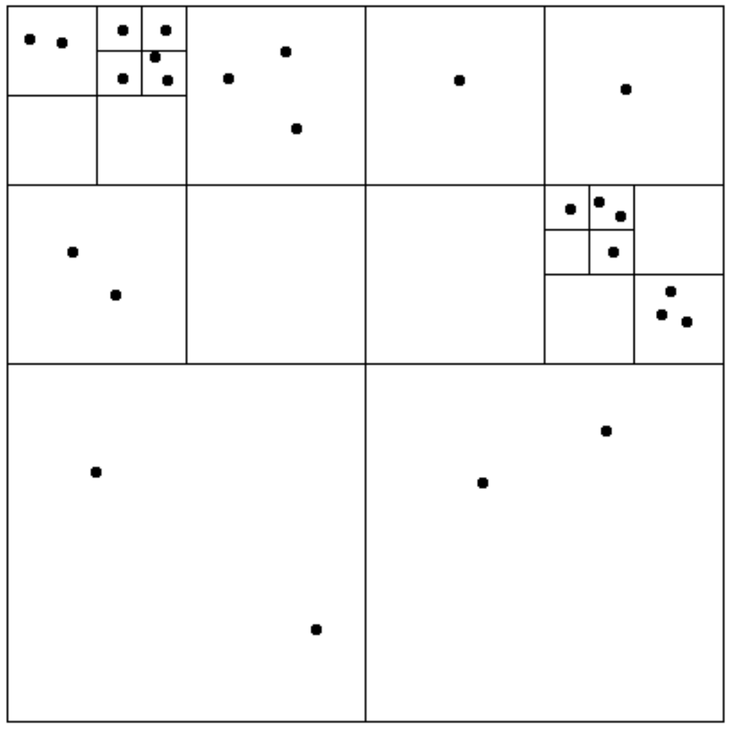
Static assignment

- When is static assignment applicable?
  - When cost (execution time) of work is predictable
    - Simplest example: it known that all work is the same cost
    - When statistics about execution time are known (e.g., same on average)



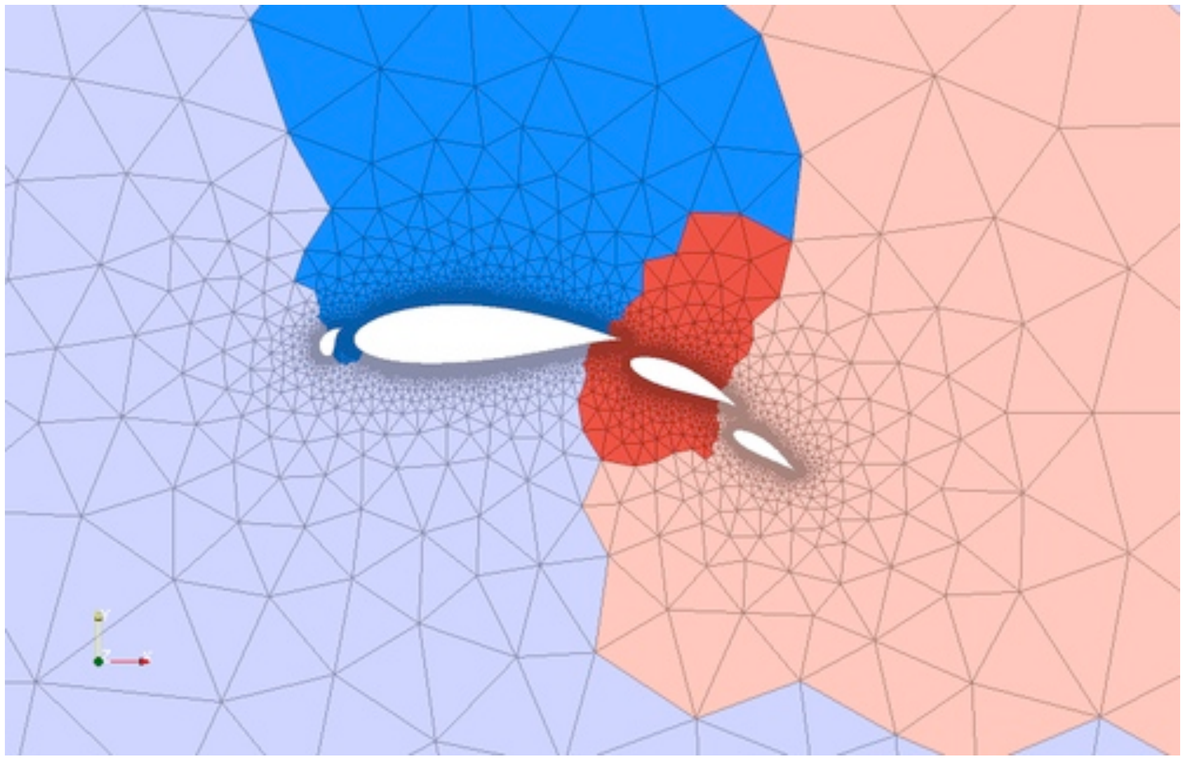
Semi-static assignment

- Cost of work predictable over near-term horizon
  - Recent past good predictor of near future
- Periodically profile application and re-adjust assignment
  - Assignment is static during interval between re-adjustment



Particle simulation:

Redistribute particles as they move over course of simulation (if motion is slow, redistribution need not occur often)



Adaptive mesh:

Refine mesh as object moves or flow over object changes

Dynamic assignment

- Program logic adapts at runtime to ensure well distributed load (execution time of tasks is unpredictable)

Sequential program
(independent loop iterations)

```
int N = 1024;
int* x = new int[N];
bool* prime = new bool[N];

// initialize elements of x

for (int i=0; i<N; i++)
{
    // unknown execution time
    prime[i] = test_primality(x[i]);
}
```

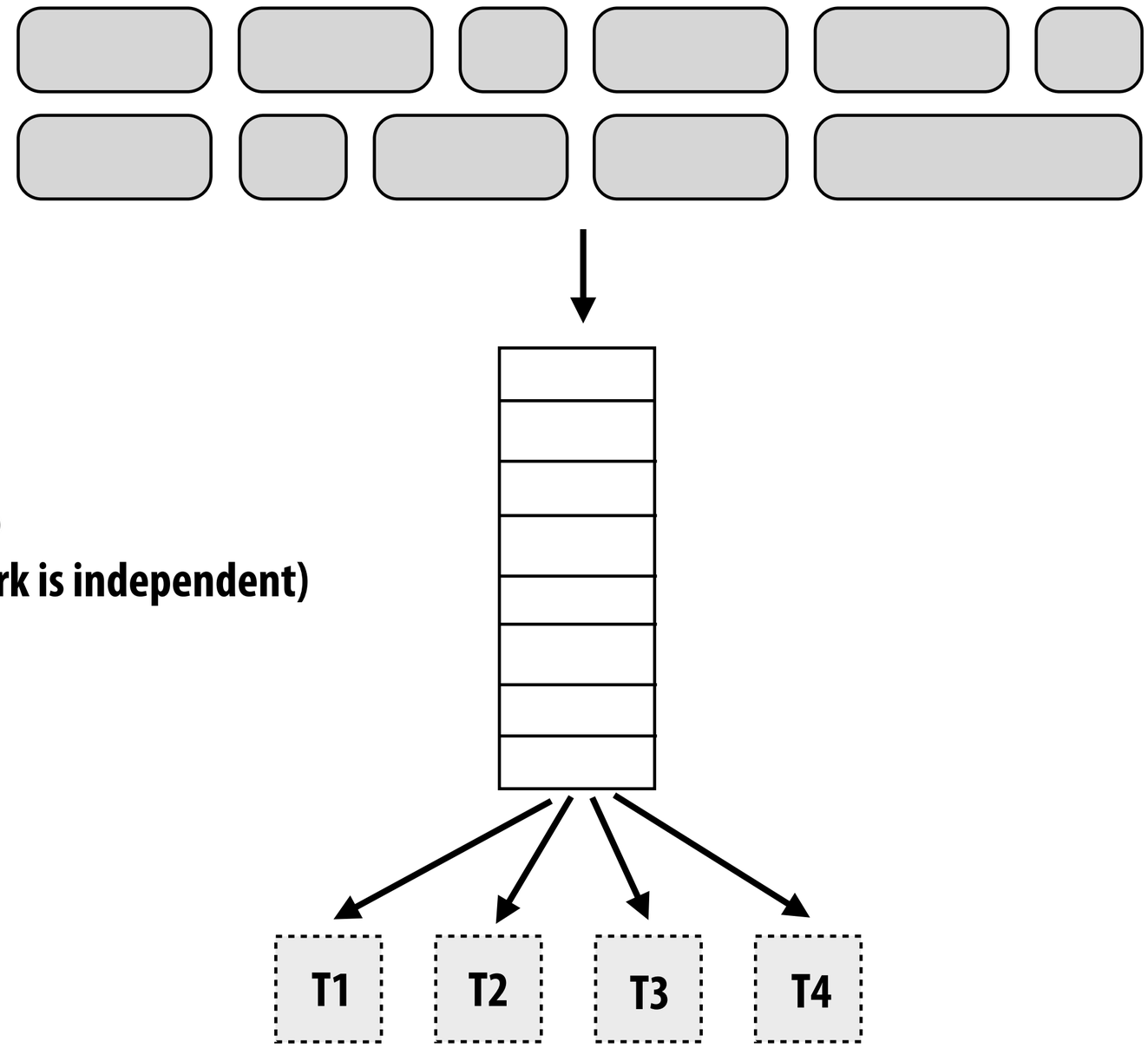
Parallel program
(SPMD execution, shared address space model)

```
LOCK counter_lock;
int counter = 0;    // shared variable (assume
                    // initialization to 0;

int N = 1024;
int* x = new int[N];
bool* prime = new bool[N];

// initialize elements of x

while (1) {
  int i;
  lock(counter_lock);
  i = counter++;
  unlock(counter_lock);
  if (i >= N)
     break;
  prime[i] = test_primality(x[i]);
}
```

atomic_incr(counter);

Dynamic assignment using work queues

Sub-problems
(aka "tasks", "work")



Shared work queue: a list of work to do
(for now, let's assume each piece of work is independent)

Worker threads:
Pull data from work queue
Push new work to queue as it's created

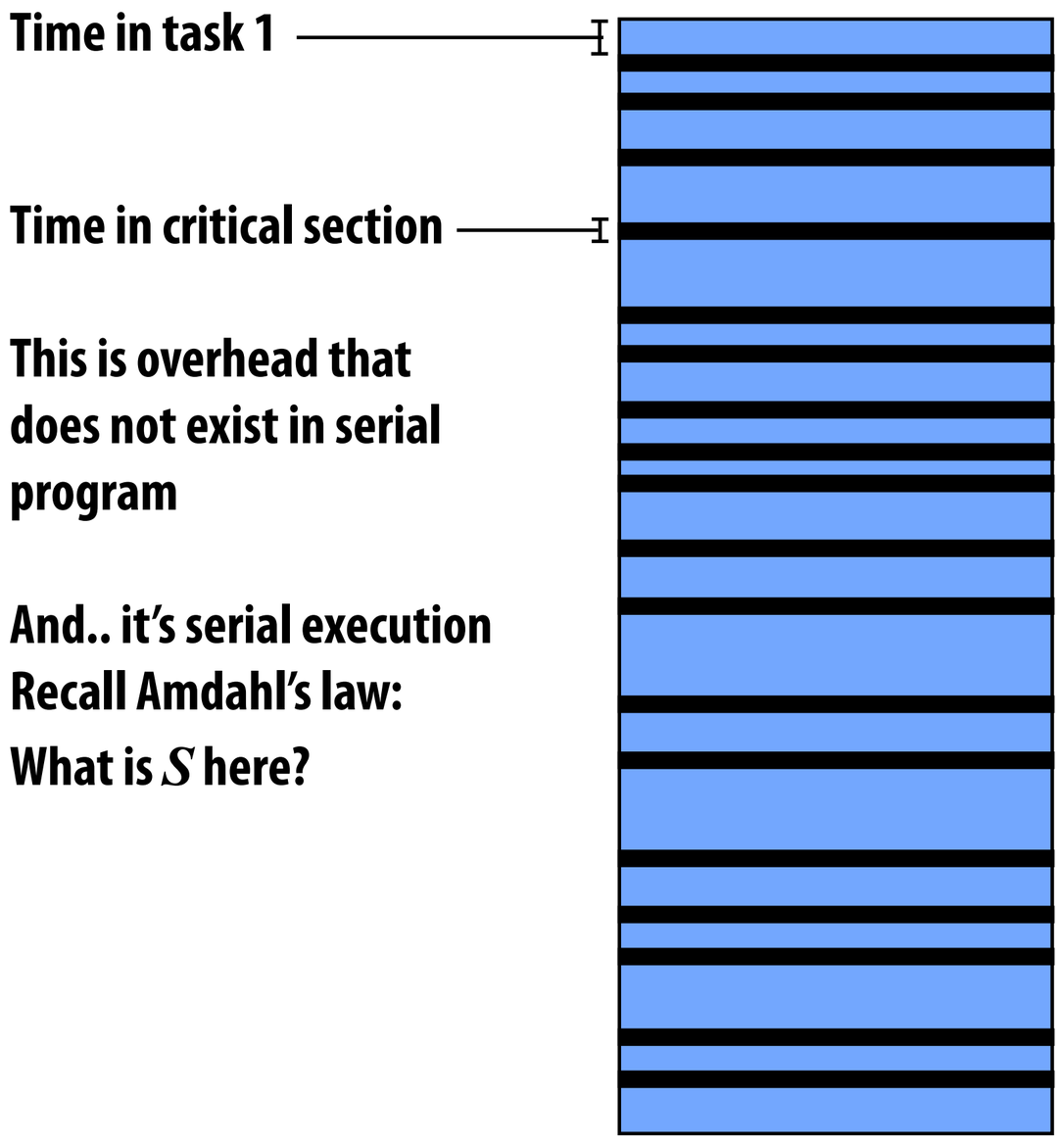
What constitutes a piece of work?

- What is a potential problem with this implementation?

```
LOCK counter_lock;
int counter = 0;    // shared variable (assume
                    // initialization to 0;
const int N = 1024;
float* x = new float[N];
bool* prime = new bool[N];

// initialize elements of x

while (1) {
  int i;
  lock(counter_lock);
  i = counter++;
  unlock(counter_lock);
  if (i >= N)
     break;
  prime[i] = test_primality(x[i]);
}
```

Fine granularity partitioning:
1 task = 1 element

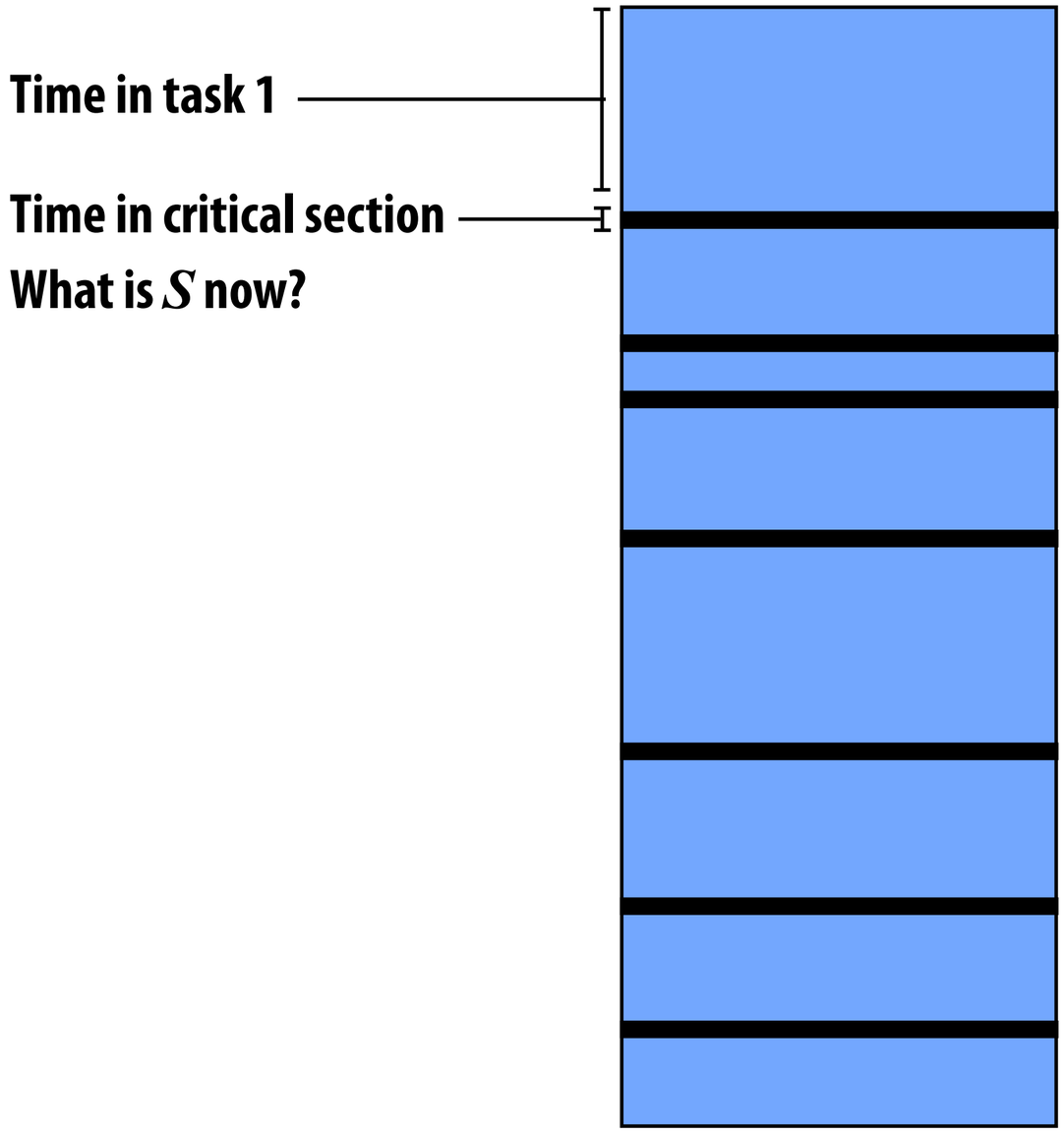
Likely good workload balance (many small tasks)
Potential for high synchronization cost
(serialization at critical section)



So... IS this a problem?

Increasing task granularity

```
LOCK counter_lock;
int counter = 0;    // shared variable (assume
                    // initialization to 0;
const int N = 1024;
const int GRANULARITY = 10;
float* x = new float[N];
bool* prime = new bool[N];

// initialize elements of x

while (1) {
  int i;
  lock(counter_lock);
  i = counter;
  counter += GRANULARITY;
  unlock(counter_lock);
  if (i >= N)
     break;
  int end = min(i + GRANULARITY, N);
  for (int j=i; j<end; j++)
     prime[i] = test_primality(x[i]);
```



What is S now?

Coarse granularity partitioning:
1 task = 10 elements

Decreased synchronization cost
(Critical section entered 10 times less)

So... have we done better?

Rule of thumb

- **Want many more tasks than processors**
  (many small tasks enable a partitioning that achieves good workload balance)
  - Motivates small granularity tasks

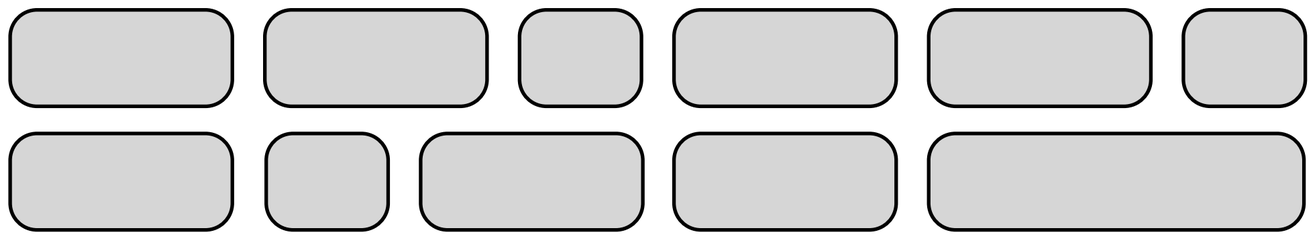
- **But want as few tasks as possible to minimize overhead of managing the assignment**
  - Motivates large granularity tasks

- **Ideal granularity depends on many factors**
  (must know your workload, and your machine)

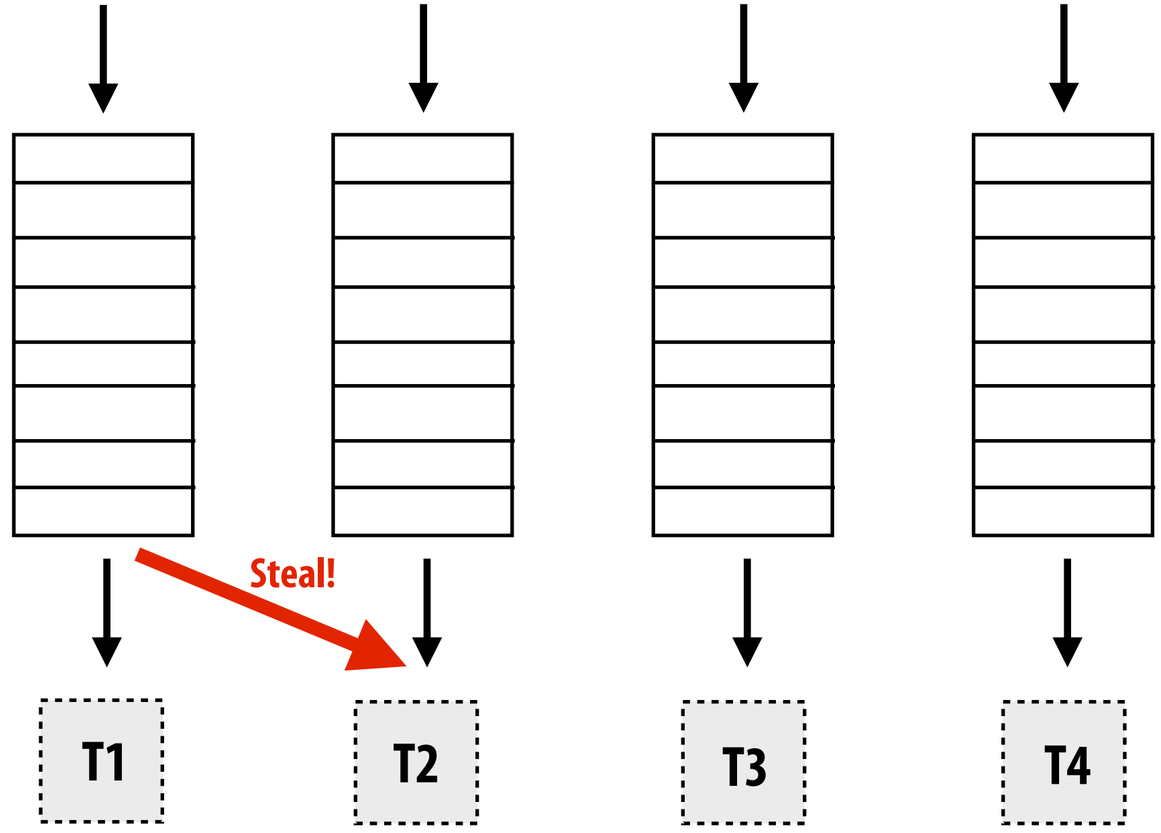
Decreasing synchronization overhead

- **Distributed work queues**
  - Replicate data to remove synchronization
  - Reoccurring theme: recall barrier example at the start of the lecture

Subproblems
(aka "tasks", "work")



Set of work queues
(In general, one per thread)



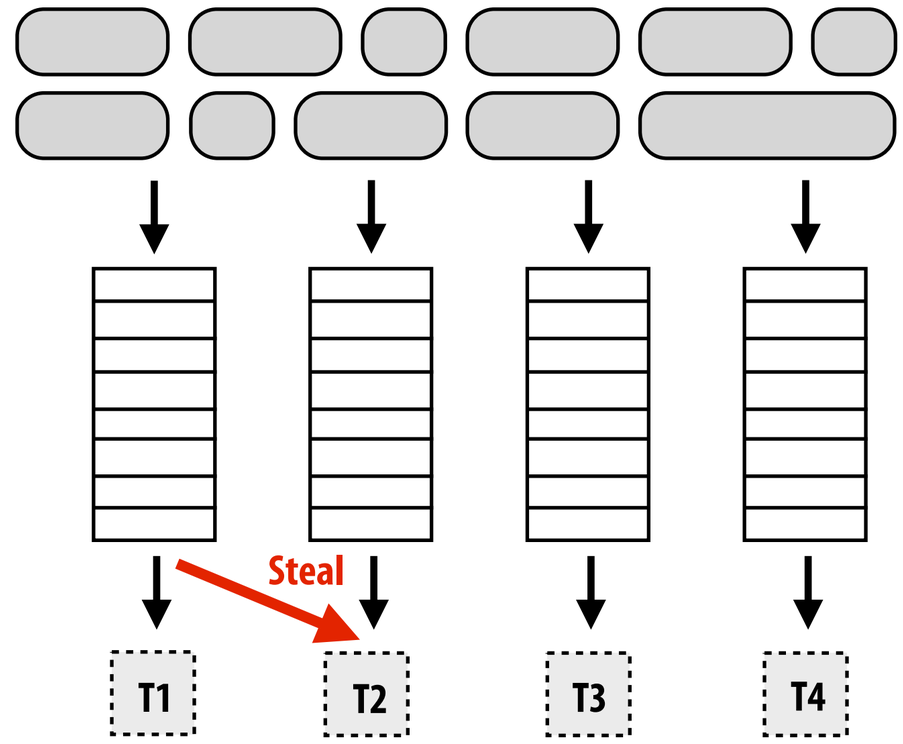
Worker threads:
Pull data from OWN work queue
Push work to OWN work to queue
When idle...
STEAL work from another work queue

Distributed work queues

- Costly synchronization/communication occurs during stealing
  - But not every time a thread takes on new work
  - Stealing occurs only when necessary to ensure good load balance

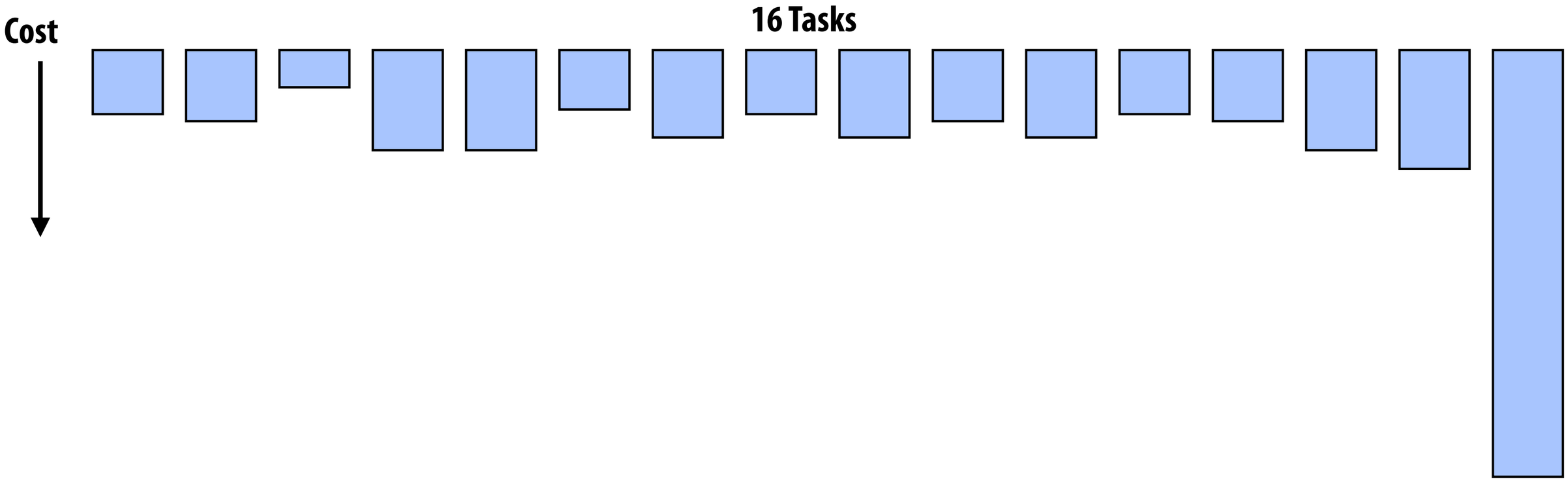
- Leads to increased locality
  - Common case: threads work on tasks they create (producer-consumer locality)

- Implementation challenges
  - Who to steal from?
  - How much to steal?
  - How to detect program termination?
  - Ensuring local queue access is fast (while preserving mutual exclusion)



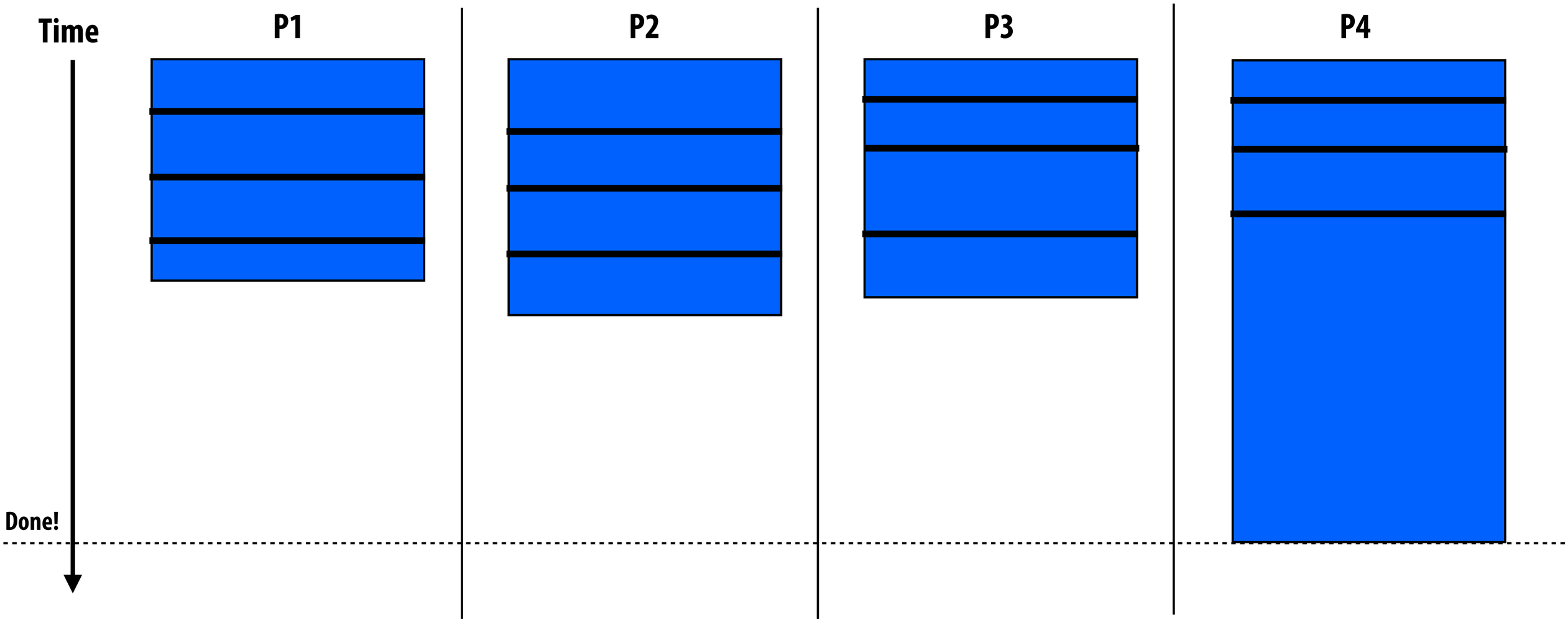
Task scheduling

What happens if scheduler runs the long task last?



Task scheduling

What happens if scheduler runs the long task last?



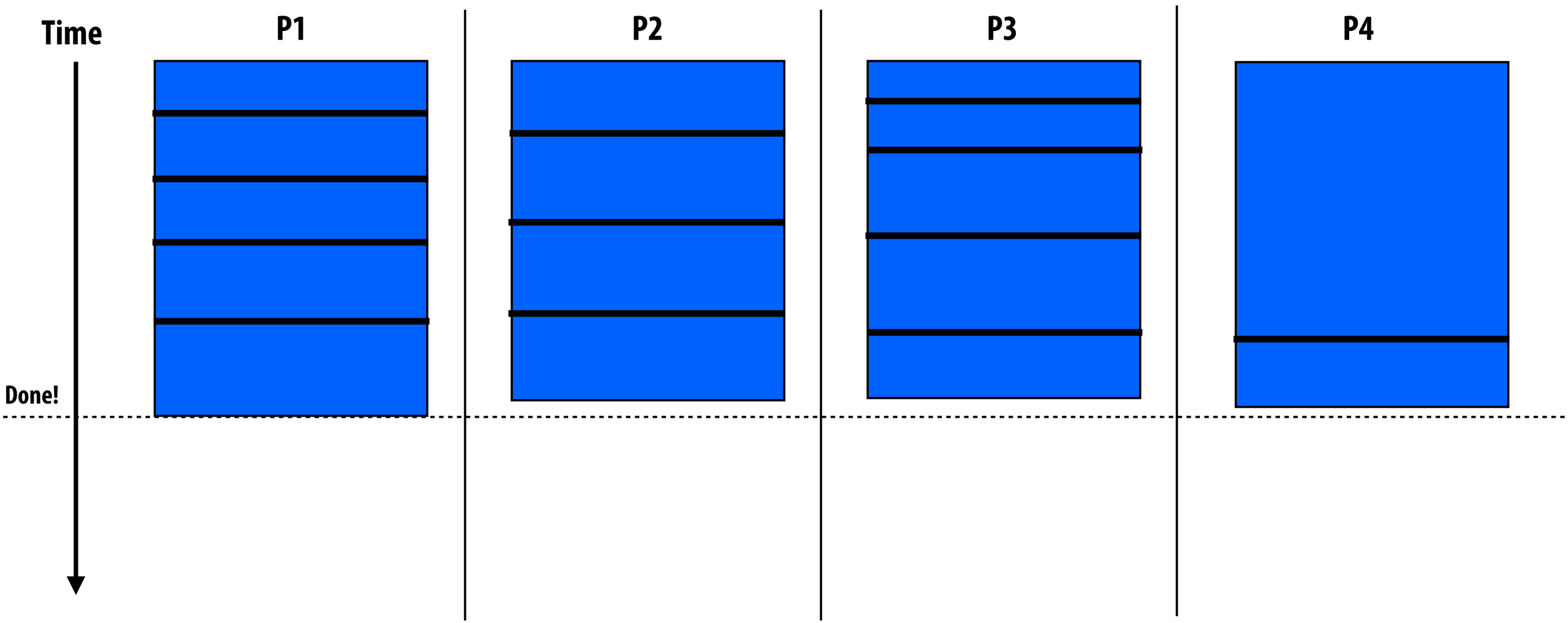
One possible solution to imbalance problem:

Divide work into a larger number of smaller tasks

- **"Long pole" gets shorter relative to overall execution time**
- **May increase synchronization overhead**
- **May not be possible (perhaps long task is fundamentally sequential)**

Task scheduling

Schedule long task first to reduce "slop" at end of computation



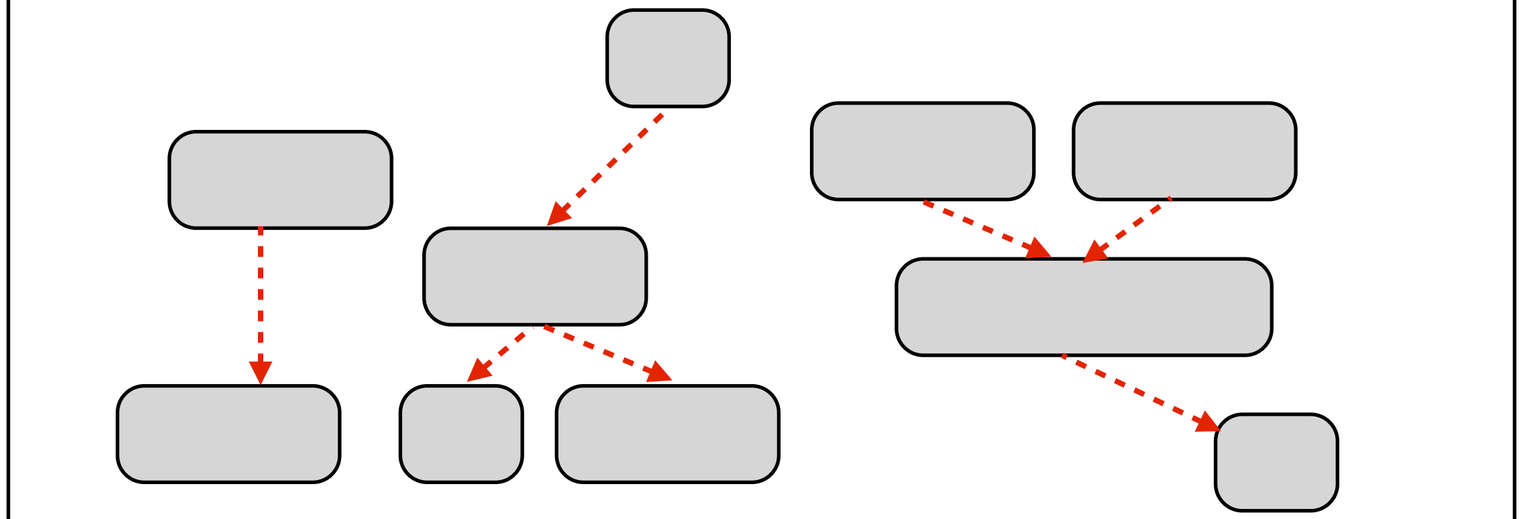
Another solution: better scheduling

Schedule long task first

- Thread performing long task performs fewer tasks
- Requires some knowledge of workload (some predictability of cost)

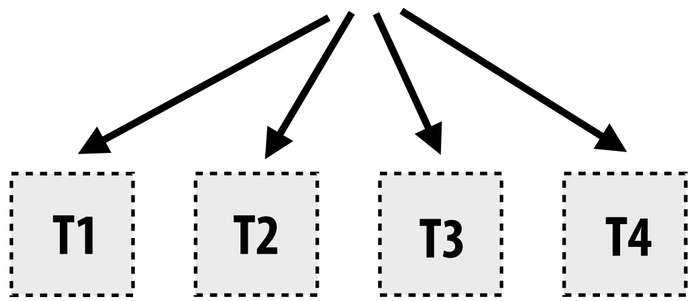
Work in task queues need not be independent

= dependency



Task management system:
Scheduler manages dependencies

Worker threads:
Assigned tasks only when dependencies are satisfied
Can submit new tasks (with optional explicit dependencies) to task system



Summary

- Challenge: achieving good workload balance
  - Want all processors working at all times
  - But want low cost to achieve this balance
    - Minimize computational overhead (e.g., scheduling logic)
    - Minimize synchronization costs

- Static assignment vs. dynamic assignment (really, it's a continuum)
  - Use up front knowledge about workload as much as possible to reduce task management/synchronization costs (in the limit, fully static)

- Issues discussed span decomposition, assignment, and orchestration