Lecture 6:

Programming for Performance, Part 1: Work Distribution

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

The Heavy

Colleen

(Great Vengeance and Furious Fire)

"Colleen? Ha, that wasn't about a girl. We wrote that one about the dangers of premature program optimization. It burns everyone, and it's certainly burned me."

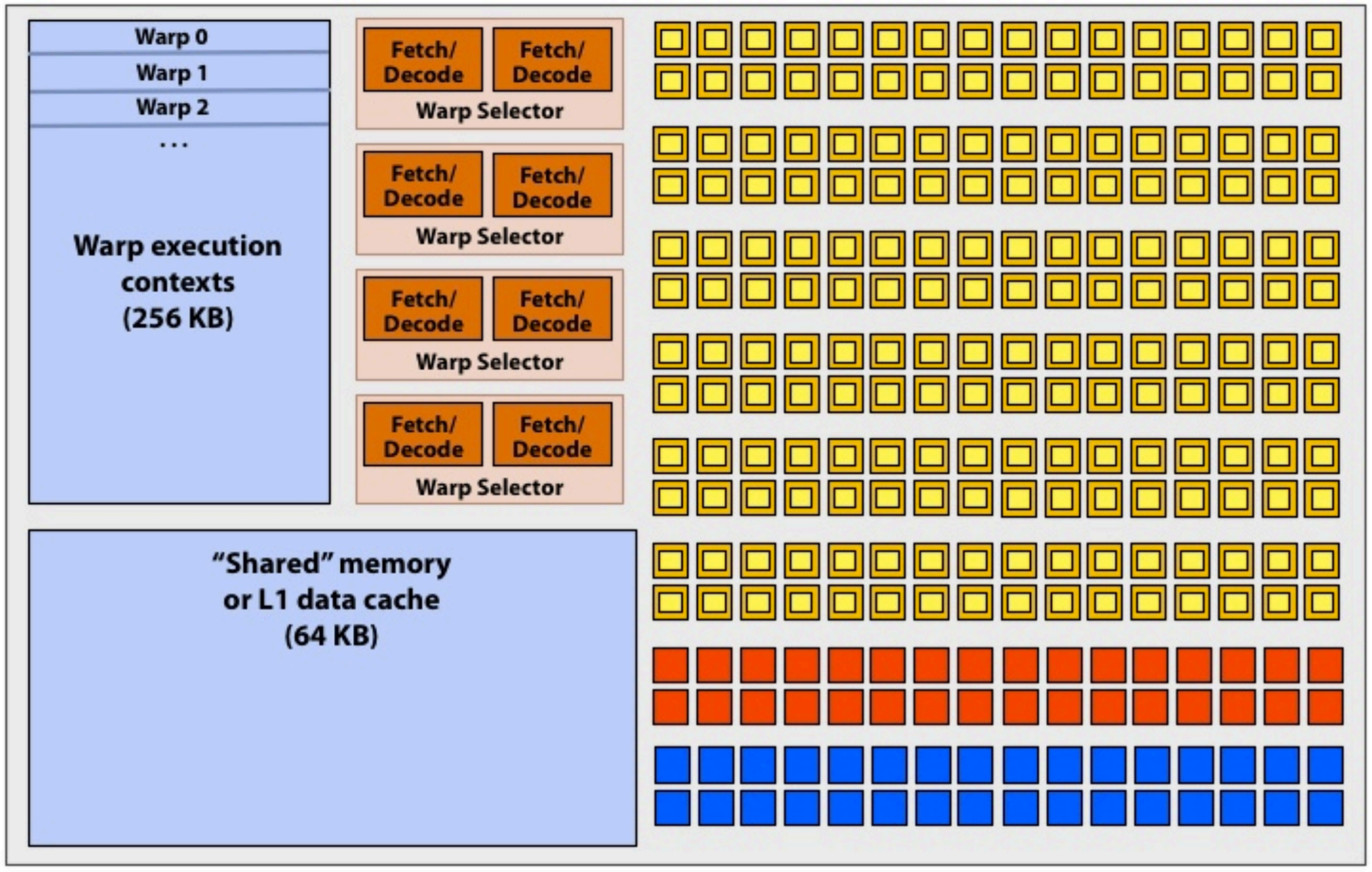
- Kelvin Swaby

Today

- Review mapping of CUDA programming to GPUs
- Solver example in the message passing model
- Begin discussing techniques for optimizing parallel programs

Finishing up a few CUDA concepts...

Review: executing warps on GTX 680



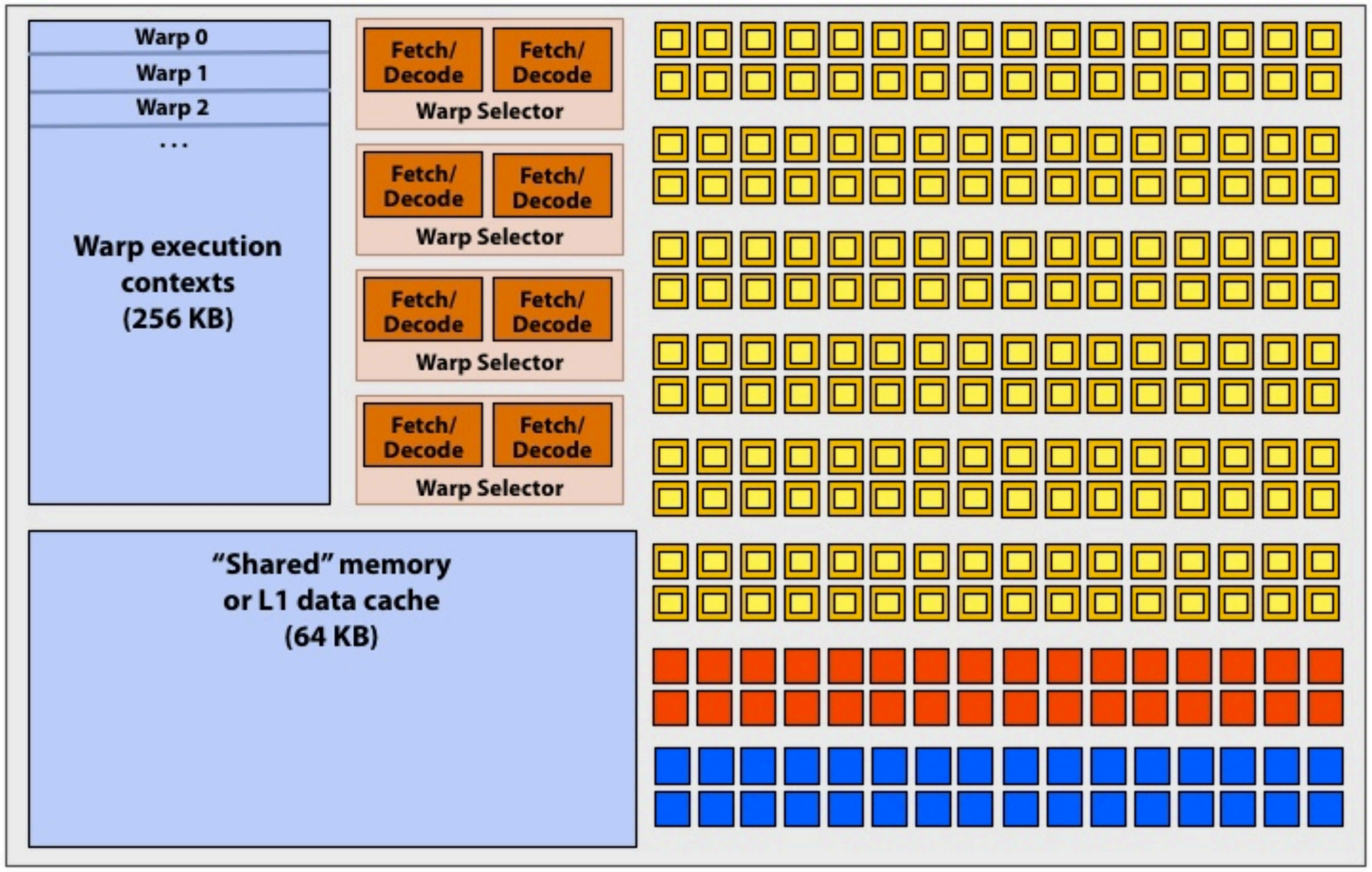
= SIMD functional unit, control shared across 32 units (1 MUL-ADD per clock)

= "special" SIMD functional unit, control shared across 32 units (operations like sin/cos)

= SIMD load/store unit (handles warp loads/stores, gathers/scatters)

- SMX core operation each clock:
  - Select up to four runnable warps from up to 64 resident on core (thread-level parallelism)
  - Select up to two runnable instructions per warp (instruction-level parallelism)
  - Execute instructions on available groups of SIMD ALUs, special-function ALUs, or LD/ST units

Review: scheduling threads in a thread block




```
#define THREADS_PER_BLK 256

__global__ void convolve(int N,
                         float* input, float* output)
{
  __shared__ float support[THREADS_PER_BLK+2];
   int index = blockIdx.x * blockDim.x + threadIdx.x;

   support[threadIdx.x] = input[index];
   if (threadIdx.x < 2) {
      support[THREADS_PER_BLK+threadIdx.x]
        = input[index+THREADS_PER_BLK];
   }

   __syncthreads();

   float result = 0.0f;  // thread-local variable
   for (int i=0; i<3; i++)
     result += support[threadIdx.x + i];

   output[index] = result;
}
```

Imagine a thread block with 256 CUDA threads

Only 4 warps worth of parallel execution in HW

Why not just have a pool of four "worker" warps?

(e.g., run 4x32=128 threads in block to completion, then run next 128 threads in block)

CUDA kernels may create dependencies between threads in a block

Simplest example is __syncthreads()

Threads in a block <u>cannot</u> be executed by the system in any order when dependencies exist.

CUDA semantics: threads in a block ARE running concurrently. If a thread in a block is runnable it will eventually be run! (no deadlock)

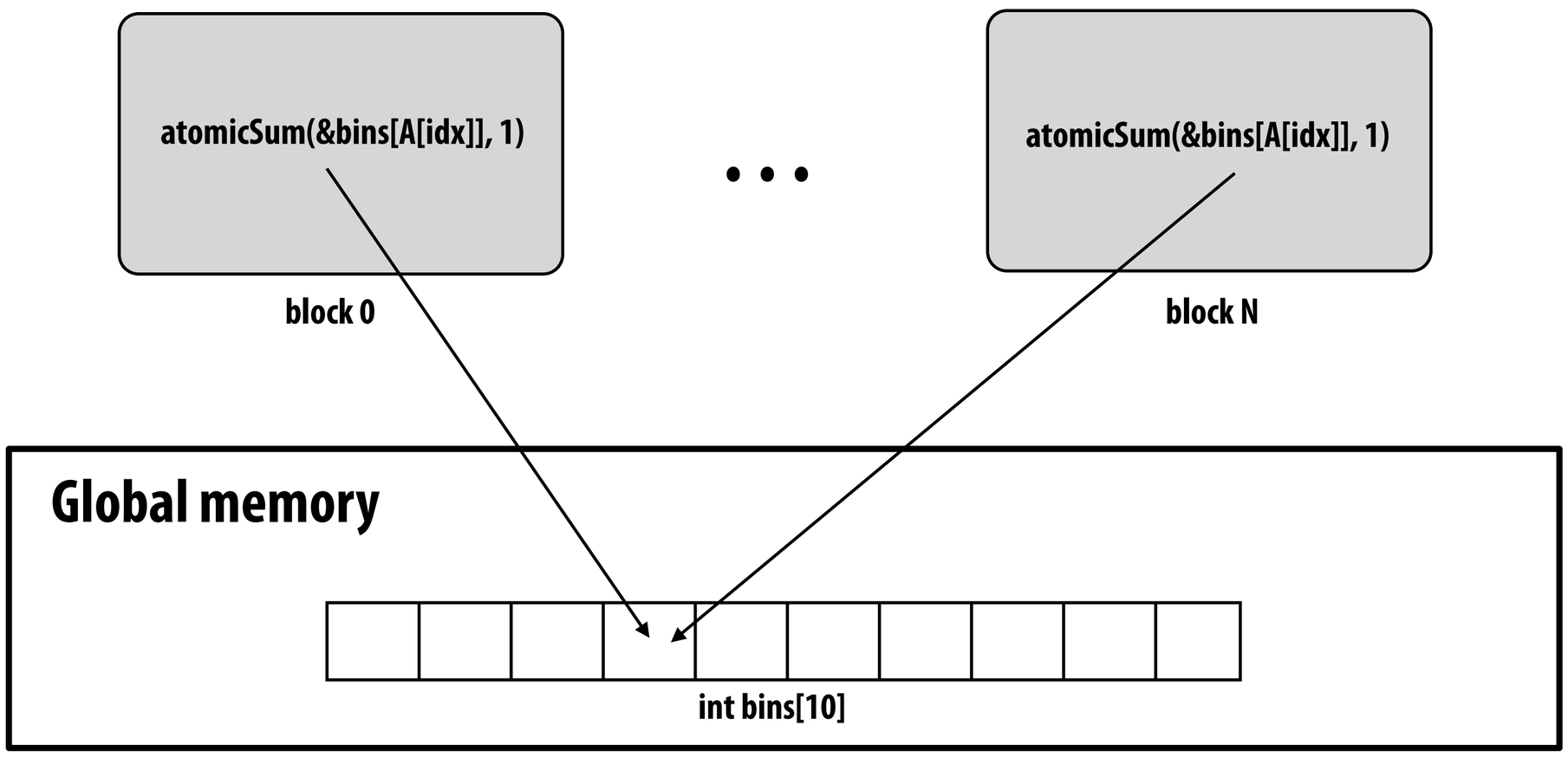
CUDA execution semantics

- **Thread blocks can be scheduled in any order by the system**
  - System assumes no dependencies
  - A lot like ISPC tasks, right?
- **Threads in a block DO run concurrently**
  - When block begin execution, all threads are running concurrently (these semantics impose a scheduling constraint on the system)
  - A CUDA thread block <u>is itself</u> an SPMD program (like an ISPC gang of program instances)
  - Threads in thread-block are concurrent, cooperating "workers"
- **CUDA implementation:**
  - A Kepler GPU warp has performance characteristics akin to an ISPC gang of instances (but unlike an ISPC gang, the warp concept is not CUDA programming model concept *)
  - All warps in a thread block are scheduled onto the same core, allowing for high-BW/low latency communication through shared memory variables
  - When all threads in block complete, block resources (shared memory allocations, warp execution contexts) become available for next block

* Exceptions to this statement include intra-warp builtin operations like swizzle and vote

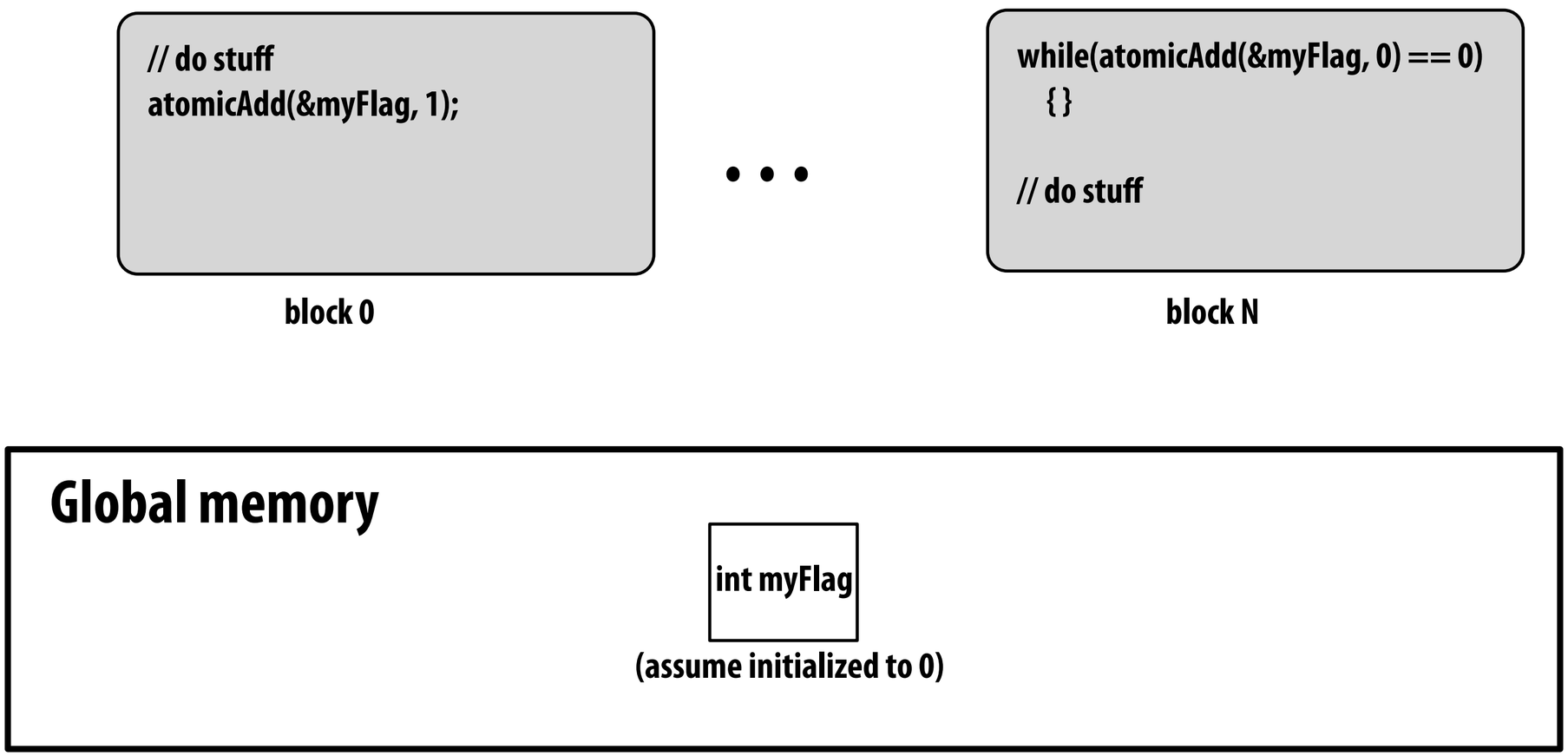
Implications of CUDA global memory atomics

- Last class I pointed out that GPUs schedule CUDA thread blocks in any order.
- CUDA threads can atomically update shared variables in global memory
  - Example: build a histogram of values in an array
- Observe how this use of atomics does not impact implementation's ability to schedule blocks in any order (I'm using atomics for mutual exclusion, and nothing more)



Implications of CUDA atomics

- But what about this?
- Consider a single core GPU with execution resources for one block per core
  - What are the possible outcomes of different schedules?



"Persistent thread" technique for CUDA programming

```
#define THREADS_PER_BLK 128
#define BLOCKS_PER_CHIP 15 * 12   // specific to a certain GTX 480 GPU

__device__ int workCounter = 0;  // global mem variable

__global__ void convolve(int N, float* input, float* output) {
  __shared__ int startingIndex;
  __shared__ float support[THREADS_PER_BLK+2];  // shared across block

  while (1) {

     if (threadIdx.x == 0)
        startingIndex = atomicInc(workCounter, THREADS_PER_BLK);
     __syncthreads();
     if (startingIndex >= N)
        break;

     int index = startingIndex + threadIdx.x; // thread local
     support[threadIdx.x] = input[index];
     if (threadIdx.x < 2)
        support[THREADS_PER_BLK+threadIdx.x] = input[index+THREADS_PER_BLK];

     __syncthreads();

     float result = 0.0f;  // thread-local variable
     for (int i=0; i<3; i++)
       result += support[threadIdx.x + i];
     output[index] = result;

      __syncthreads();
   }
}

// host code ////////////////////////////////////////////////////////
int N = 1024 * 1024;
cudaMalloc(&devInput, N+2);  // allocate array in device memory
cudaMalloc(&devOutput, N);   // allocate array in device memory
// properly initialize contents of devInput here ...

convolve<<<BLOCKS_PER_CHIP, THREADS_PER_BLK>>>(N, devInput, devOutput);
```

Some developers write CUDA code that makes assumptions about number of cores in the underlying GPU's implementation:

Programmer launches exactly as many thread-blocks as will fill the GPU

(Exploit knowledge of implementation: that GPU will in fact run all blocks concurrently)

Work assignment to blocks is implemented entirely by the application

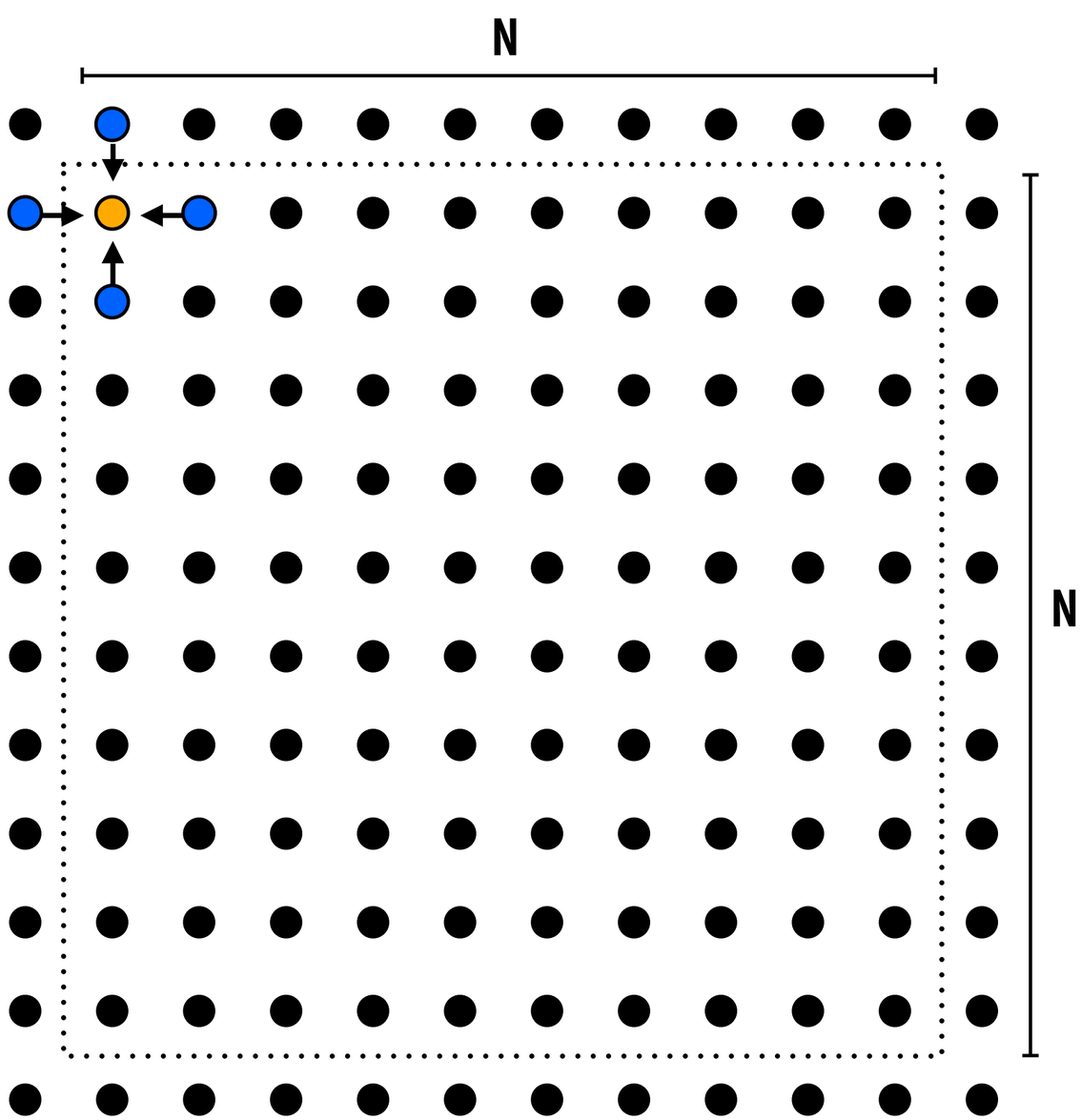
(circumvents GPU thread block scheduler, and intended CUDA thread block semantics)

Now programmer's mental model is that *all* threads are concurrently running on the machine at once.

Finally... let's finish up the solver discussion from lecture 4...

Recall basic 2D grid solver

Solver example: described in terms of data parallelism and SPMD programming models




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

Review: data-parallel solver implementation

- **Synchronization:**
  - `forall` loop iterations are independent (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

```
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;
  }
}
```

Example from: Culler, Singh, and Gupta

Solver implementation in two programming models

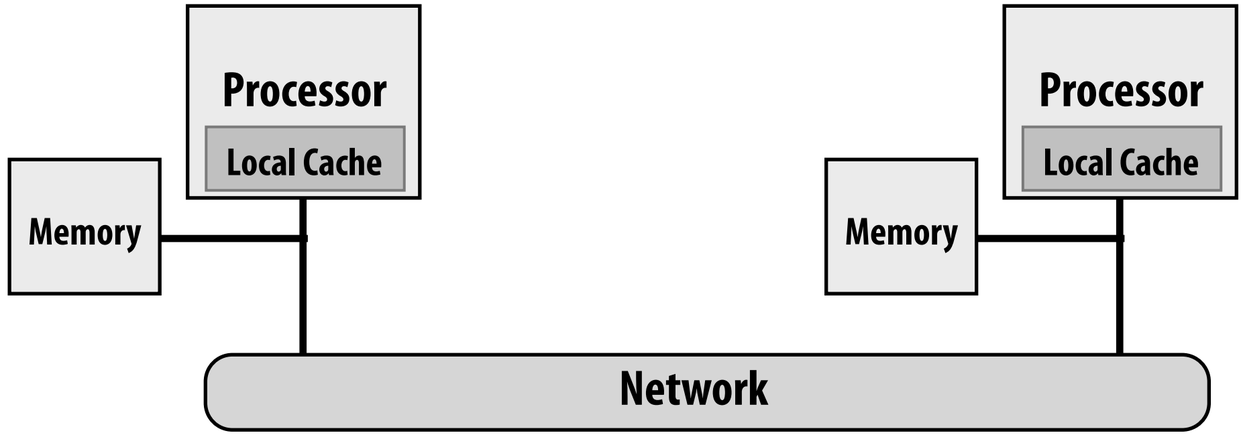
- **Data-parallel programming model**
  - Synchronization:
    - `forall` loop iterations are independent (can be parallelized)
    - Implicit barrier at end of outer `forall` loop body
  - Communication
    - Implicit in loads and stores (like shared address space)
    - Special built-in primitives: e.g., reduce

- **Shared address space**
  - Synchronization:
    - Locks (for mutual exclusion) and barriers (to separate phases of computation) are used to express dependencies
  - Communication
    - Implicit in loads/stores to shared variables

Today: message passing model

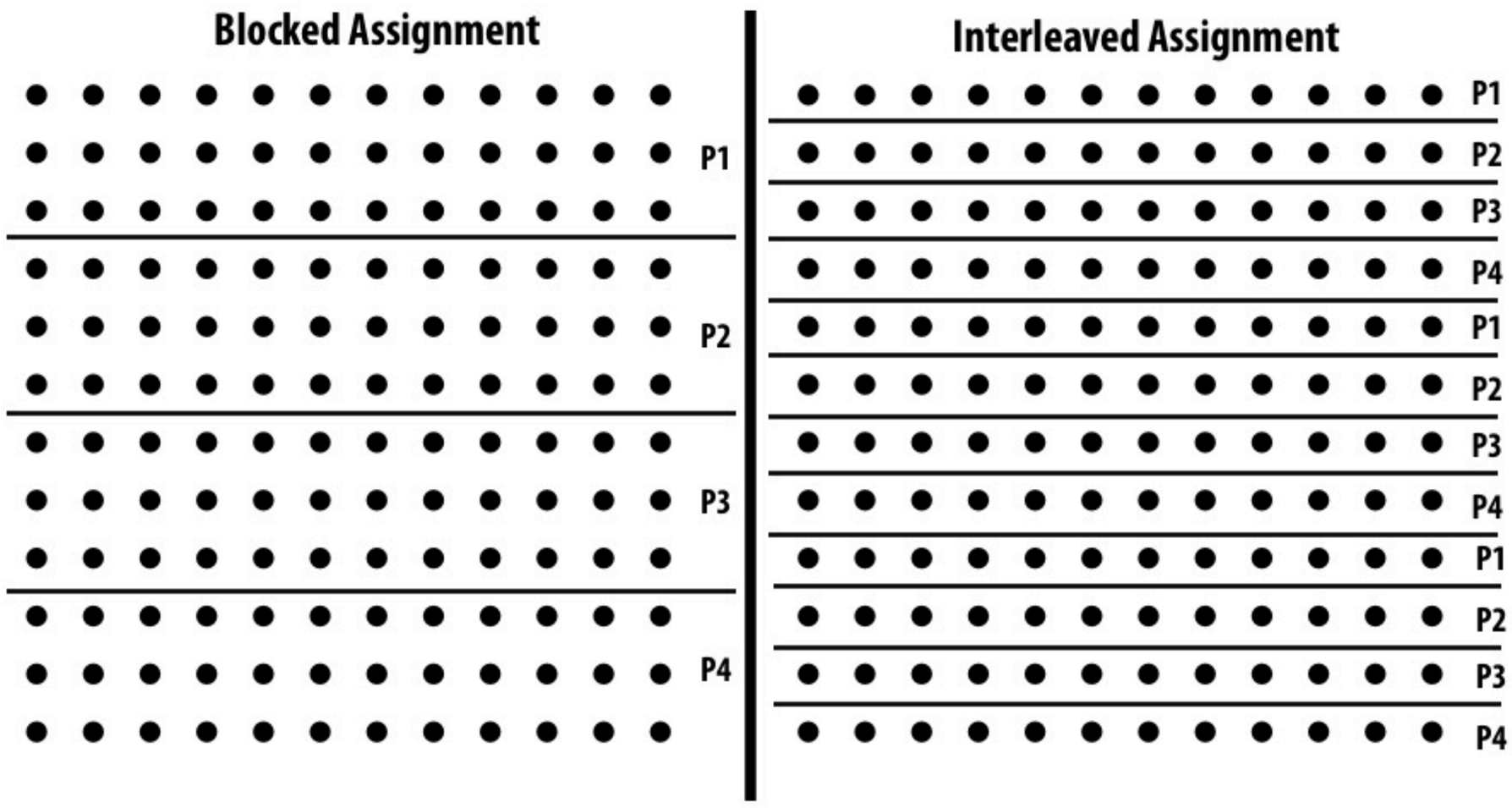
- No shared address space abstraction (i.e., no shared variables)
- Each thread has its own address space
- Threads communicate & synchronize by sending/receiving messages

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



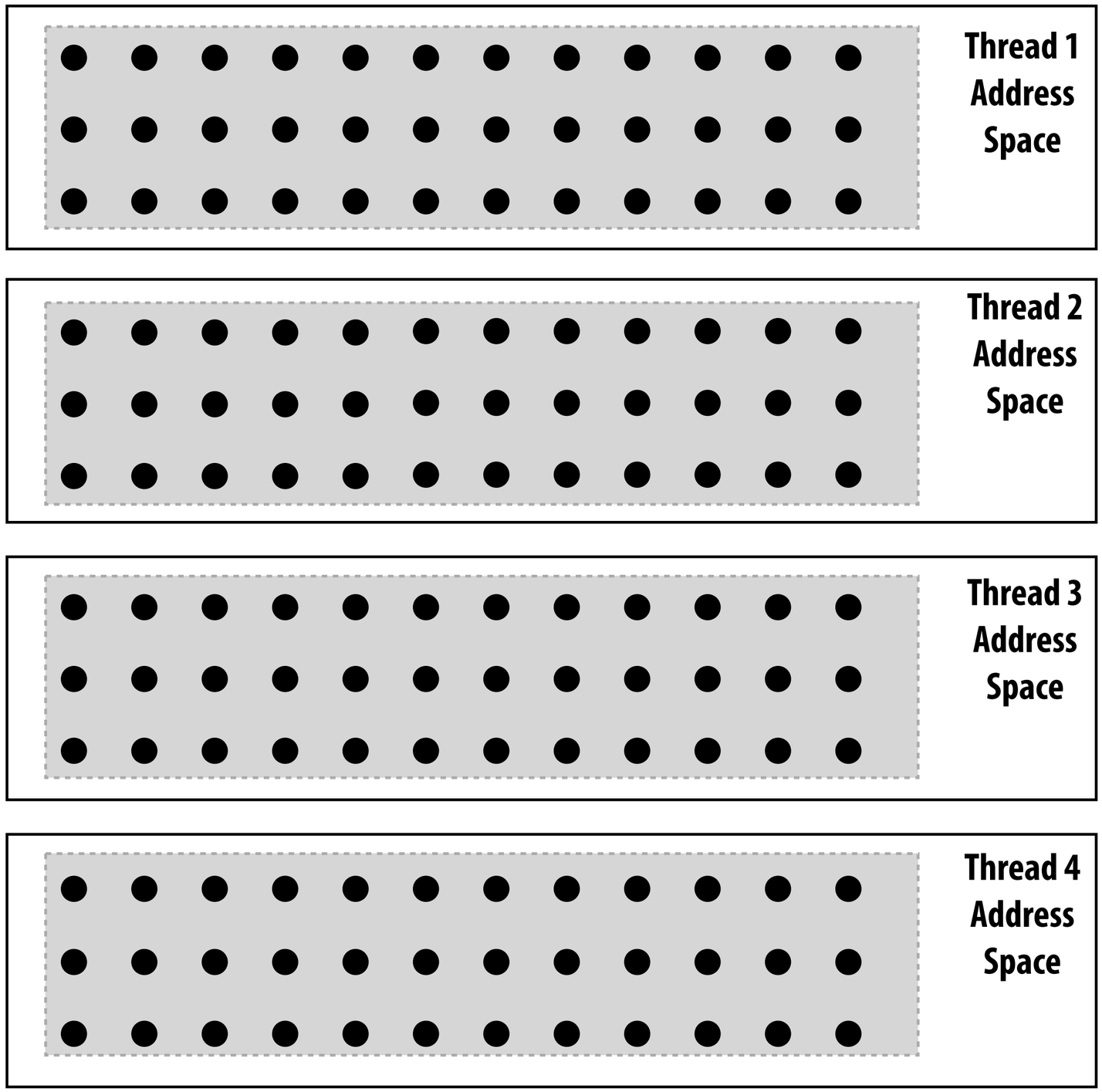
Review: assignment in a shared address space

- Grid data resided in a single array in shared address space
  - Array was accessible to all threads
- Each thread manipulated the region it was assigned to process
  - Assignment decisions impacted performance
  - Different assignments could yield different amounts of communication



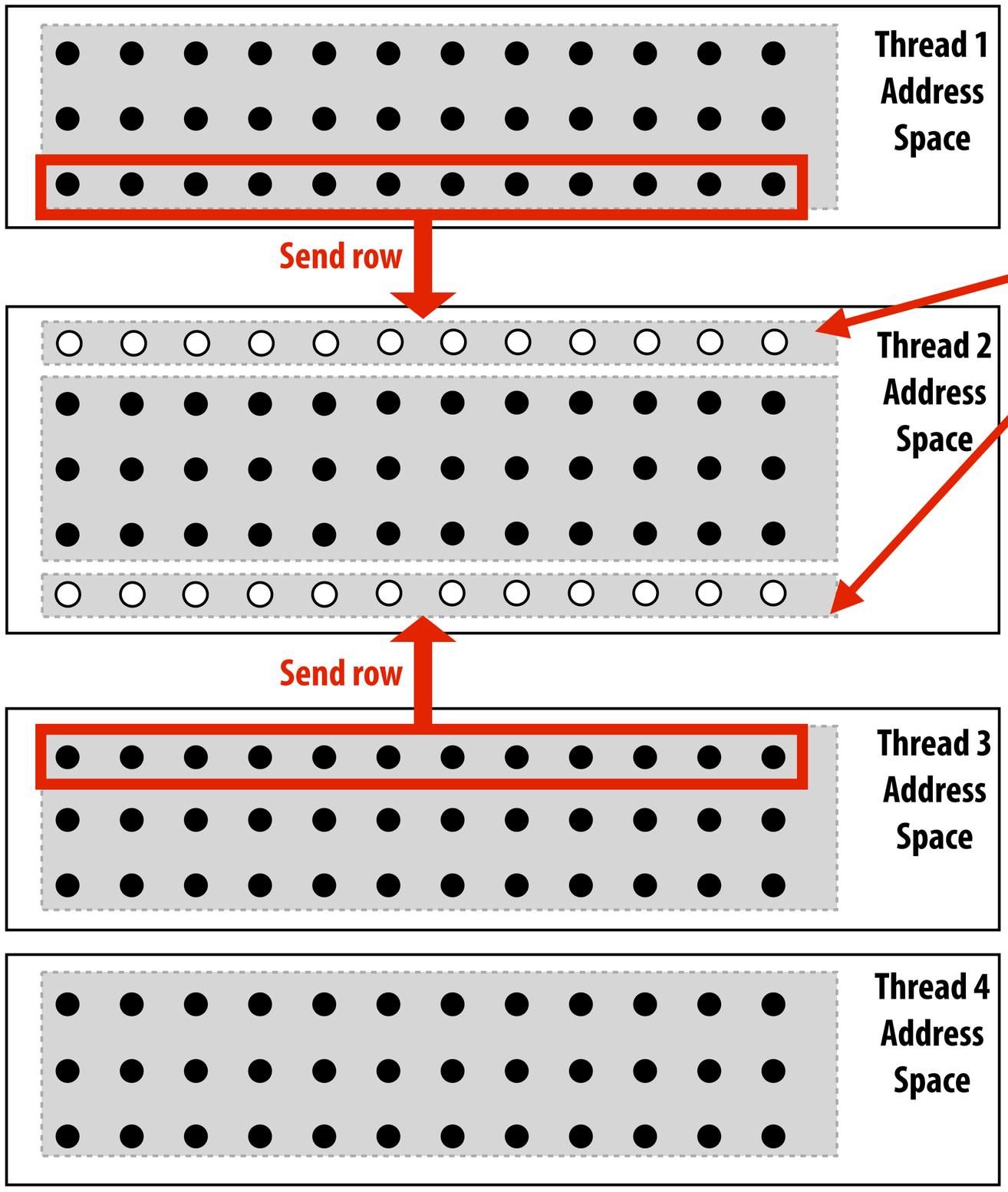
Message passing model

- 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. It's common to say that information in ghost cells is "owned" by other threads.

```
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 in message sends and receives

```
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 evaluates termination predicate sends result back to all other threads

Example pseudocode from: Culler, Singh, and Gupta

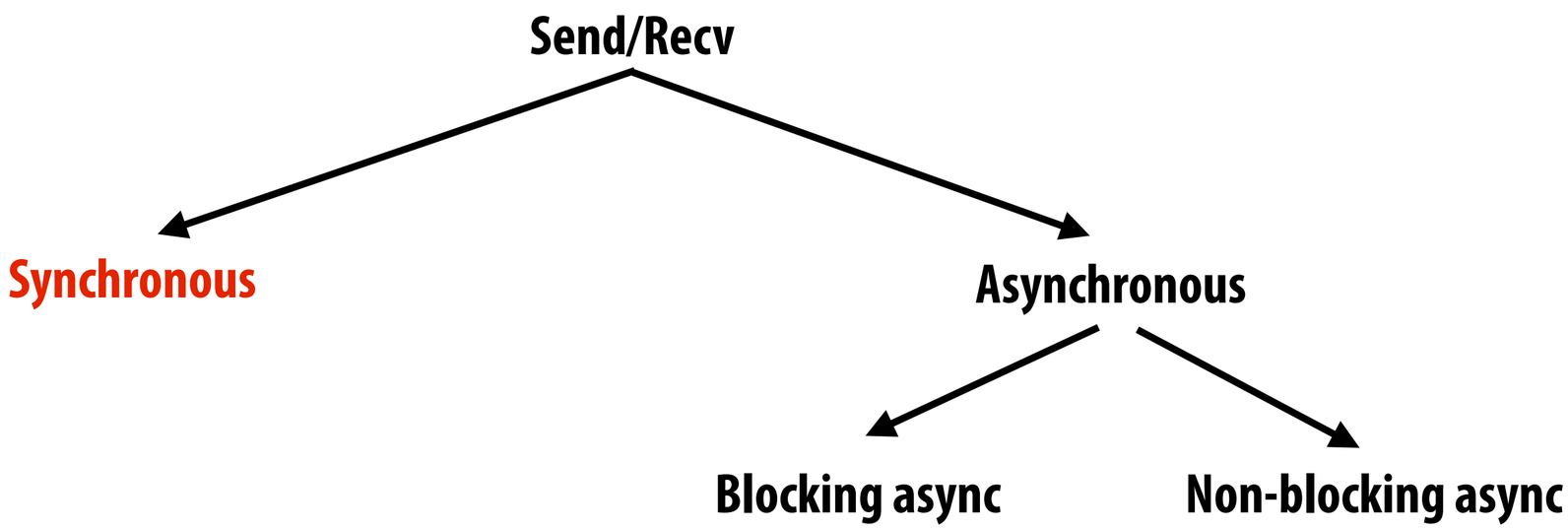
Notes on message passing example

- **Computation**
  - Array indexing is relative to local address space (not global grid coordinates)
- **Communication:**
  - Performed through messages
  - Communicate entire rows at a time (not individual elements)
- **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)**

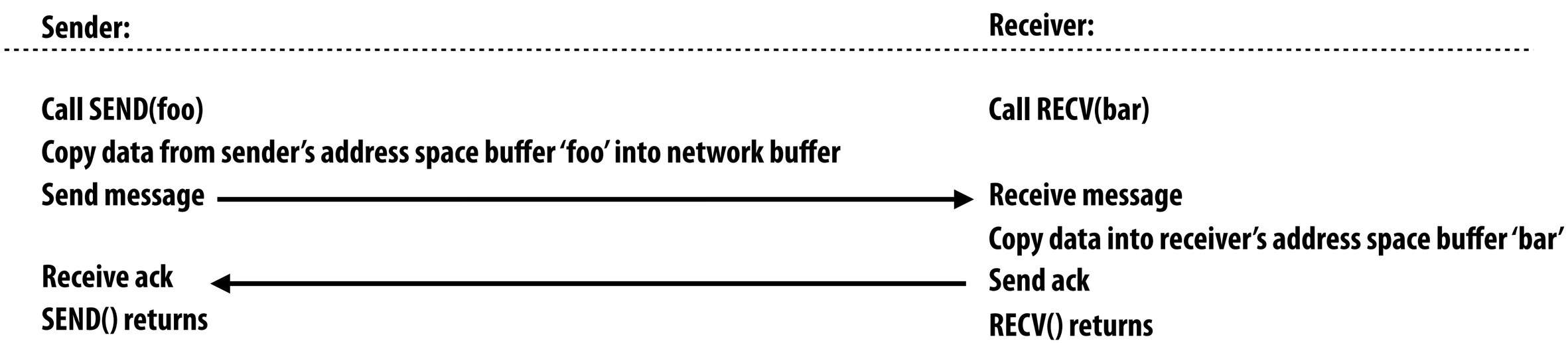
```
REDUCE(0,mydiff,sizeof(float),ADD);
if (pid == 0) then
   if (mydiff/(n*n) < TOL) then done = 1;
endif
   BROADCAST(0,done,sizeof(int),DONE);
```

Alternative solution using reduce/broadcast constructs

Variants of send and receive messages



- **Synchronous:**
  - SEND: call returns when sender receives acknowledgement message data resides in address space of receiver
  - RECV: call returns when data from message copied into address space of receiver and acknowledgement sent back to sender



As implemented on the prior 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

```
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
```

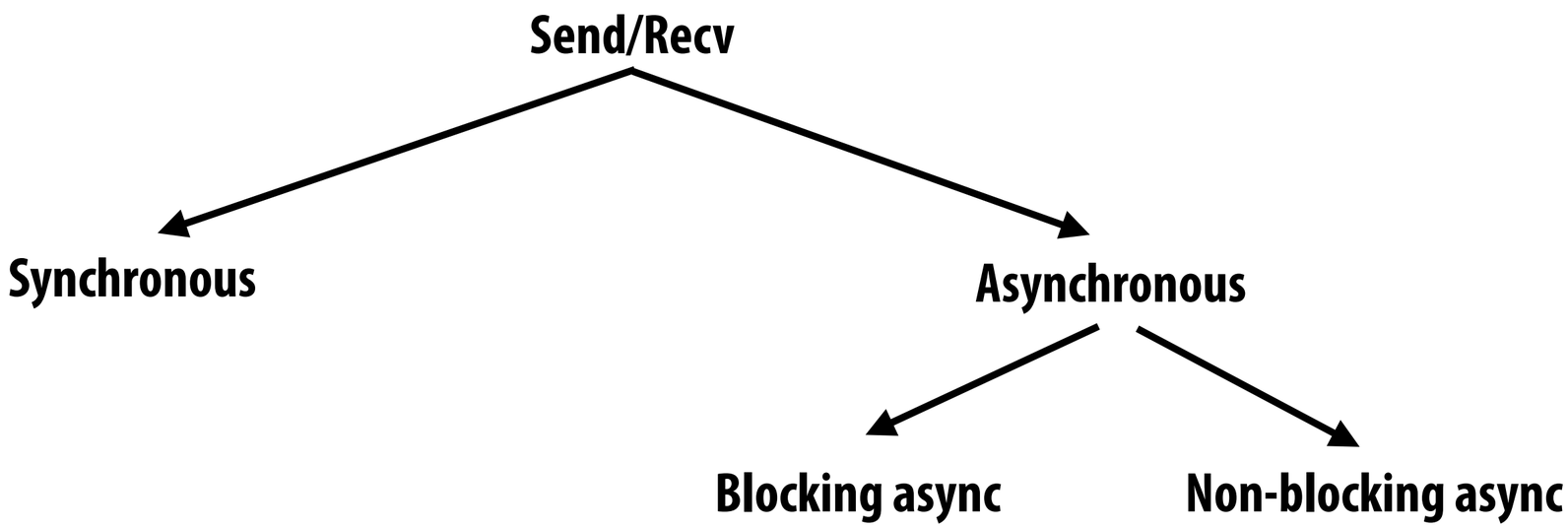
Send and receive ghost rows

Perform computation

All threads send local mydiff to thread 0

Thread 0 evaluates termination predicate sends result back to all other threads

Variants of send and receive messages

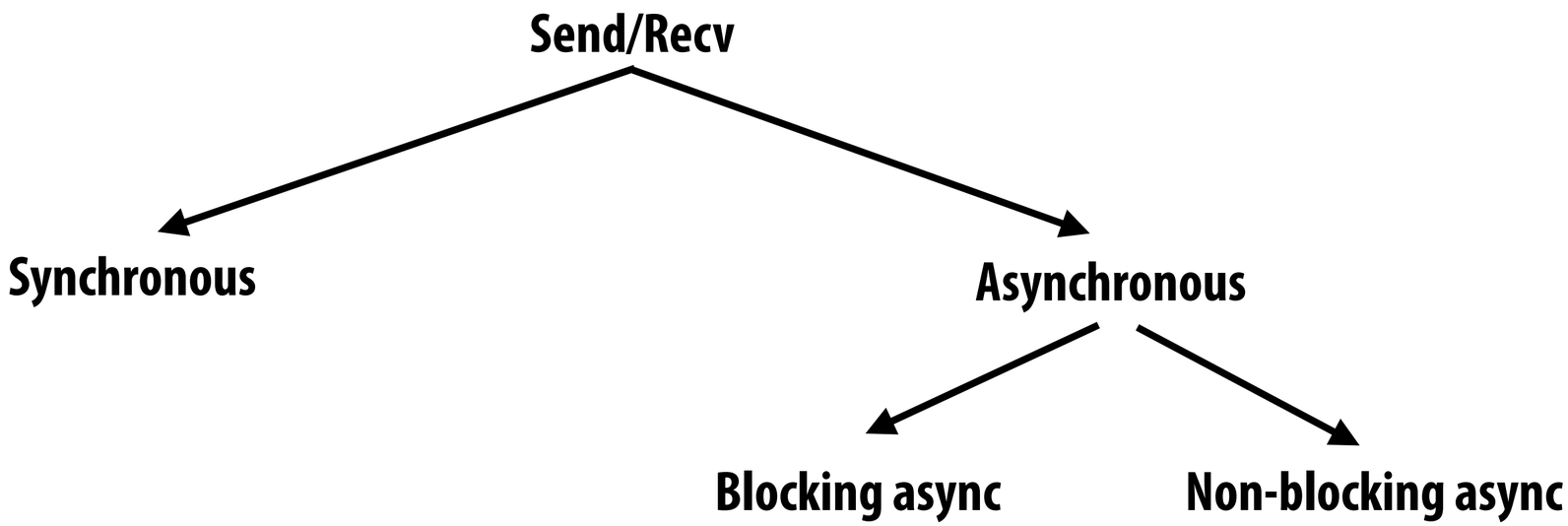


- **Blocking async:**
  - SEND: call copies data from address space into system buffers, then returns
    - Does not guarantee message has been received (or even sent)
  - RECV: call returns when data copied into address space, but no ack sent

Sender:	Receiver:
Call SEND(foo)	Call RECV(bar)
Copy data from sender's address space buffer 'foo' into network buffer	
SEND(foo) returns, calling thread continues execution	
Send message →	Receive message
	Copy data into receiver's address space buffer
	RECV(bar) returns

RED TEXT = executes concurrently with application thread

Variants of send and receive messages

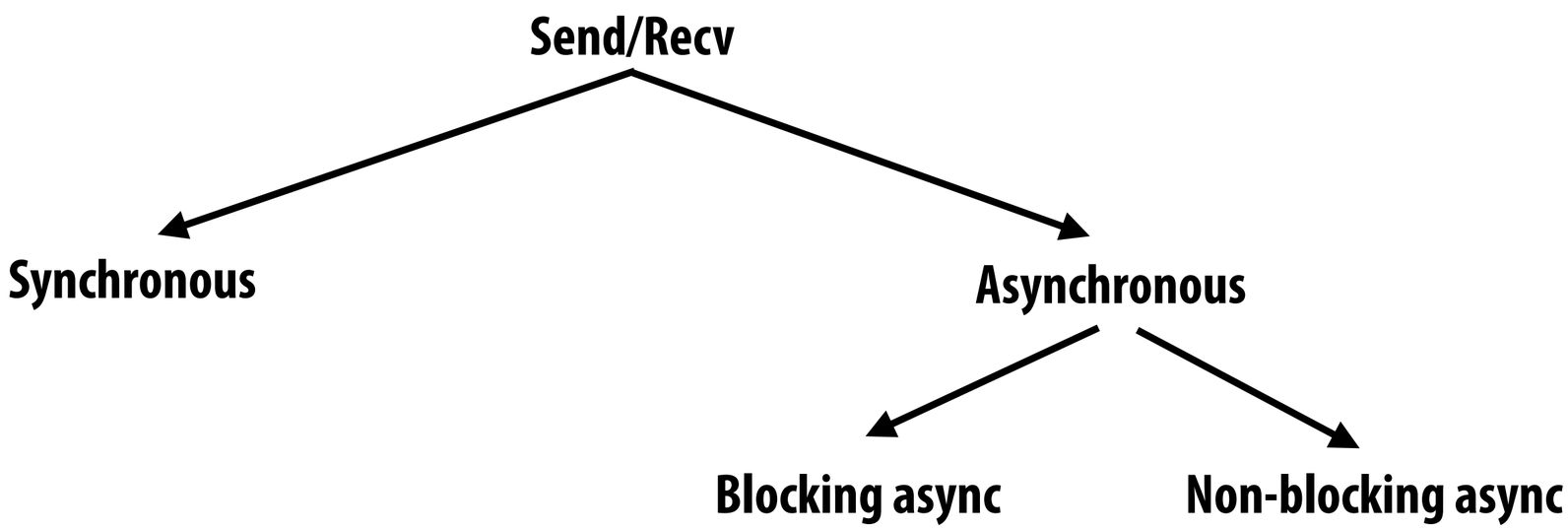


- **Non-blocking async: ("non-blocking")**
  - SEND: call returns immediately. Buffer provided to SEND cannot be modified by calling thread since message processing occurs concurrently with thread execution
  - RECV: call posts intent to receive, returns immediately.
  - Use SENDPROBE, RECVPROBE to determine actual send/receipt status

Sender:	Receiver:
Call SEND(foo)	Call RECV(bar)
SEND(foo) returns handle h1	RECV(bar) returns handle h2
Copy data from 'foo' into network buffer	
Send message →	Receive message
	Messaging library copies data into 'bar'
Call SENDPROBE(h1) // if message sent, now safe for thread to modify 'foo'	Call RECVPROBE(h2)
	// if received, now safe for thread
	// to access 'bar'

RED TEXT = executes concurrently with application thread

Variants of send and receive messages



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:
  - `forall` loop iterations are independent (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:
  - Locks used to ensure mutual exclusion
  - Barriers used to express coarse dependencies (e.g., between phases of computation)
- Communication
  - Implicit in loads/stores to shared variables

3. Message passing model

- Synchronization:
  - Implemented via messages
  - Mutual exclusion exists by default: no shared data structures
- Communication:
  - Explicit communication via send/recv needed for parallel program correctness
  - Bulk communication for efficiency: e.g., communicate entire rows, not single elements
  - Several variants of send/recv, each has different semantics

Optimizing parallel program performance

(how to be l33t)

Programming for high performance

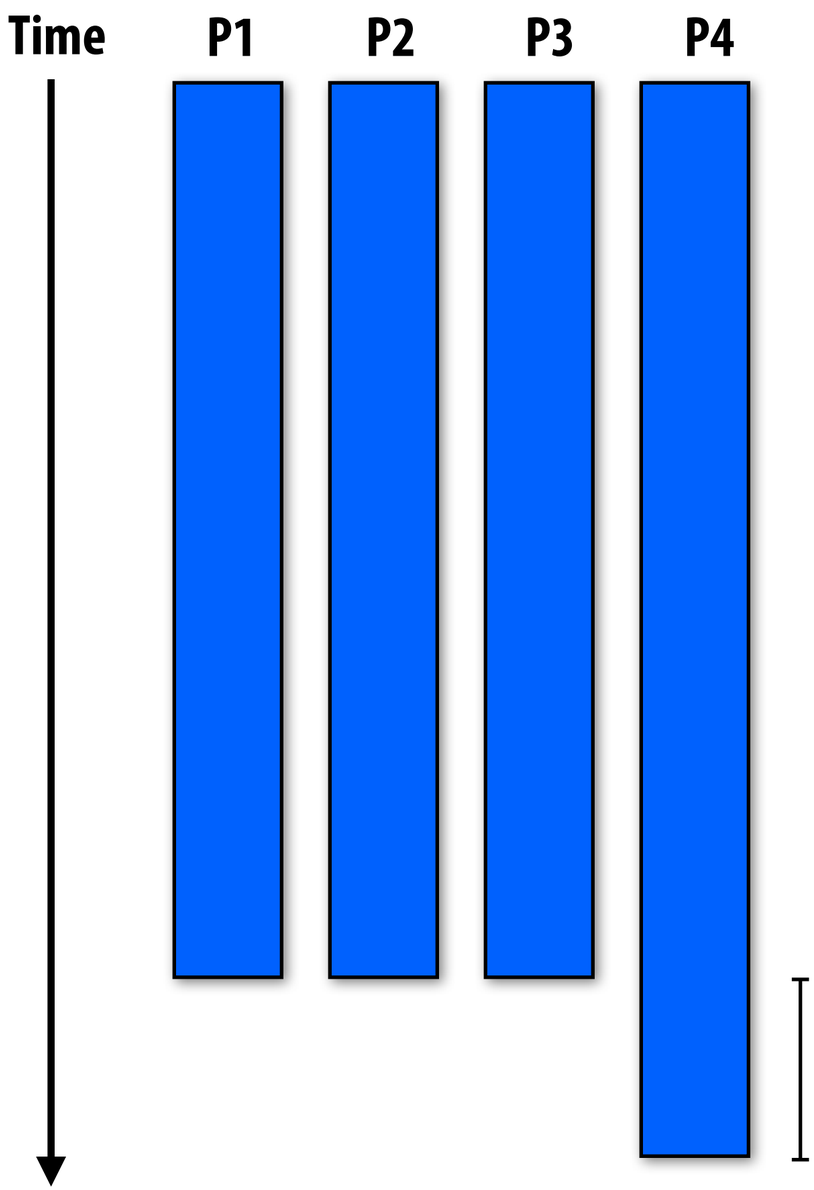
- Optimizing the performance of parallel programs is an iterative process of refining choices for decomposition, assignment, and orchestration...

- Key goals (that are at odds with each other)
  - Balance workload onto available execution resources
  - Reduce communication (to avoid stalls)
  - Reduce extra work performed to increase parallelism, manage assignment, etc.

- We are going to talk about a rich space of techniques
  - TIP #1: Always do the simplest thing first, then measure/analyze
  - "It scales" = your code scales as much as you need it to (if you anticipate only running low core count machines, it may be unnecessary to implement a complex approach that created hundreds or thousands of pieces of independent work)

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

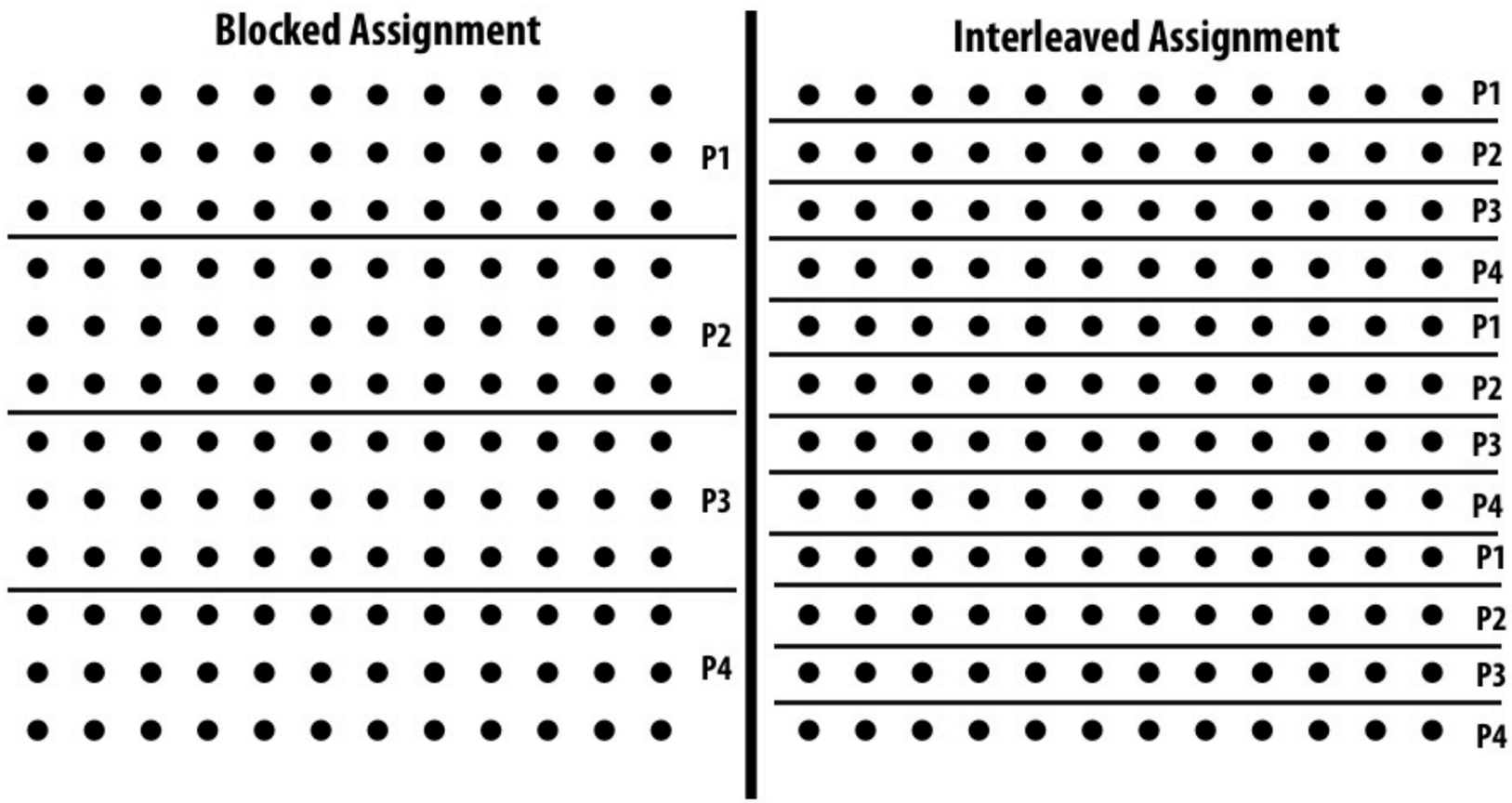
P4 does 20% more work → P4 takes 20% longer to complete

→ 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

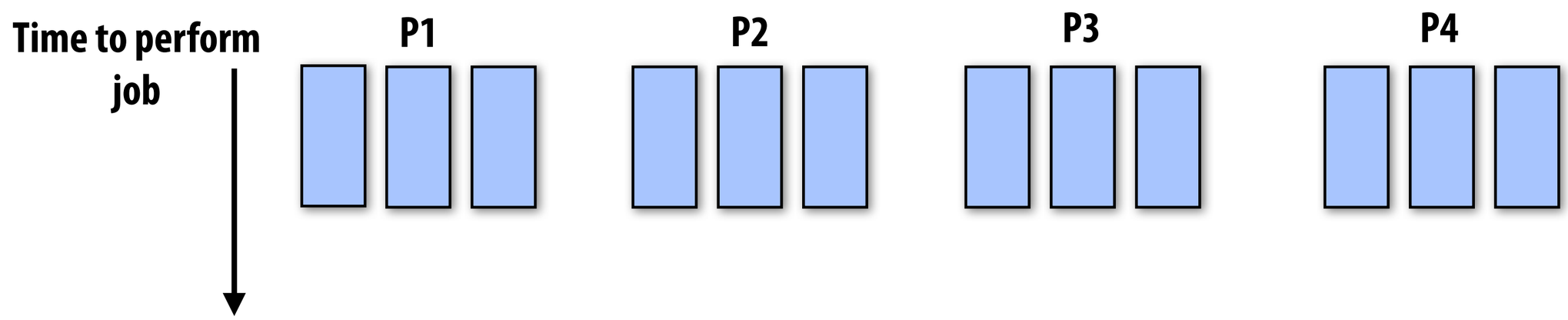
- Assignment of work to threads is pre-determined
  - Not necessarily compile-time (assignment algorithm may depend on runtime parameters such as input data size, number of threads, etc.)
- Recall solver example: assign equal number of grid cells to each thread
  - We discussed blocked and interleaved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



- Good properties: simple, essentially zero runtime overhead
  (in this example: extra work to implement assignment is a little bit of indexing math)

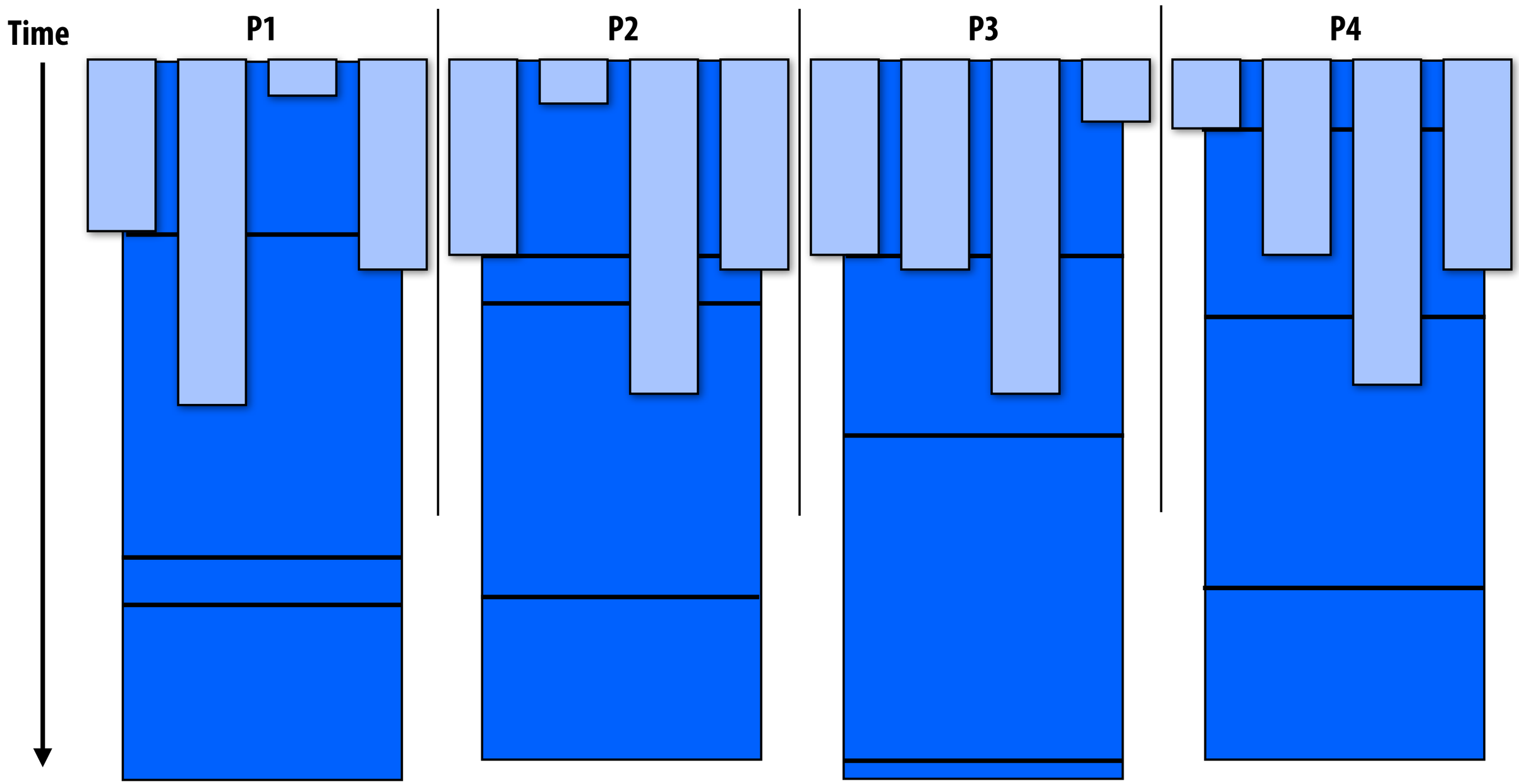
Static assignment

- When is static assignment applicable?
- When the cost (execution time) of work and the amount of work is <u>predictable</u>
- Simplest example: it is known that all work has the same cost



Static assignment

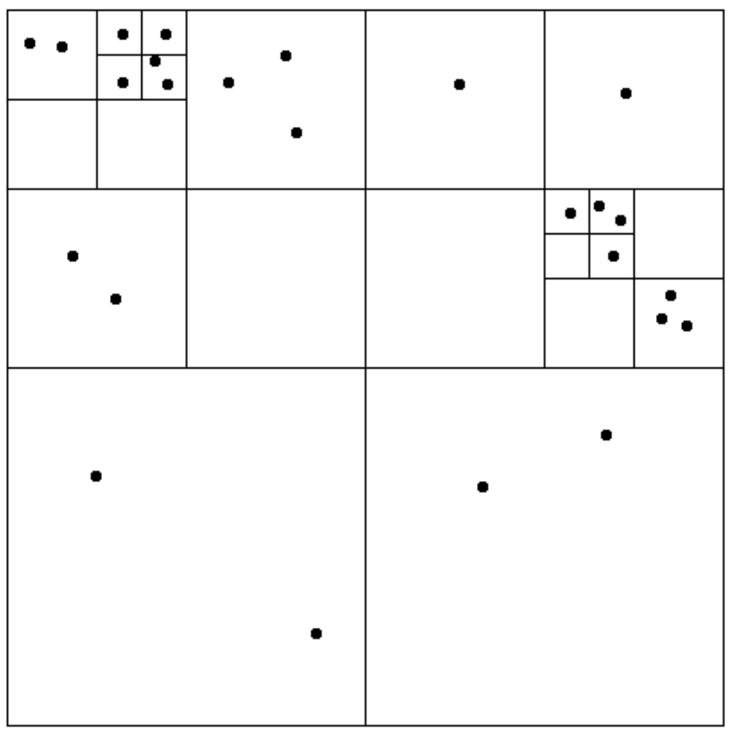
- **When is static assignment applicable?**
  - Example 2: predictable, but not all jobs have same cost (see example below)
  - Example 3: When statistics about execution time are known (e.g., same cost on average)



Jobs have unequal, but known cost: assign to processors to ensure overall good load balance

"Semi-static" assignment

- Cost of work predictable for near-term future
  - 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)

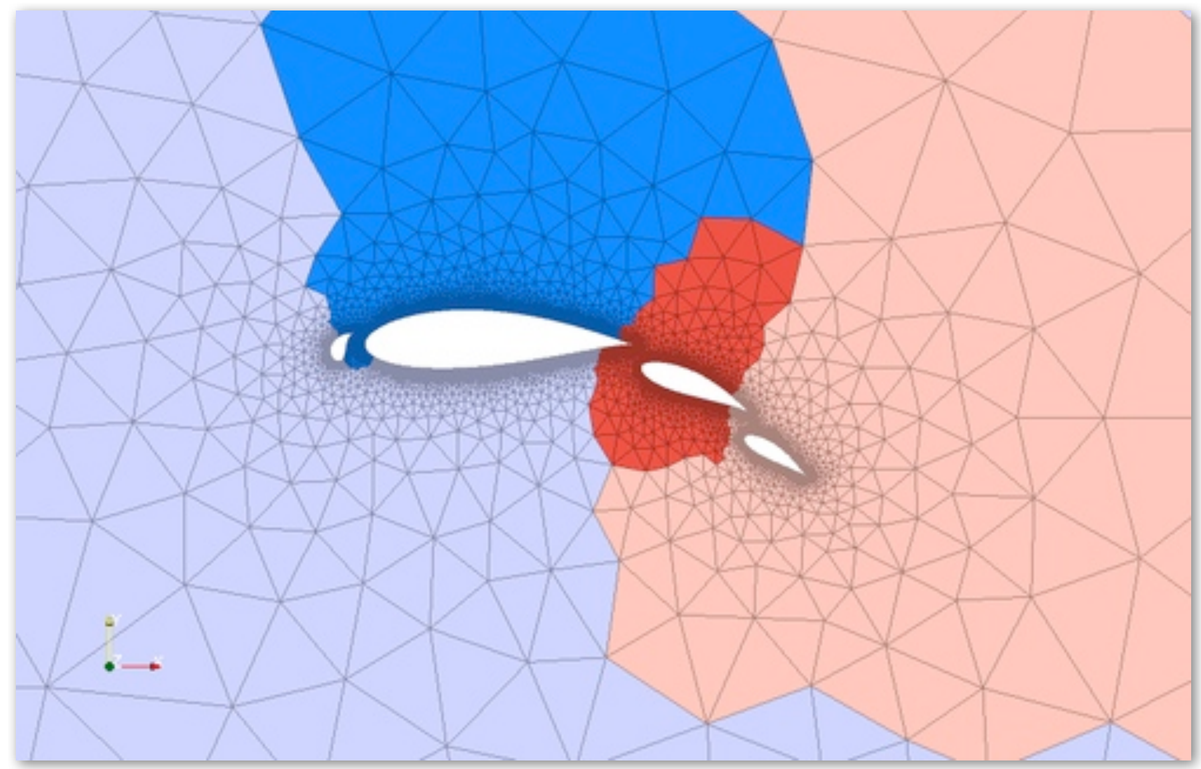


Image credit: http://typhon.sourceforge.net/spip/spip.php?article22

Adaptive mesh:

Mesh is changed as object moves or flow over object changes, but changes occur slowly (color indicates assignment of parts of mesh to processors)

Dynamic assignment

- **Assignment is determined at runtime to ensure a well distributed load. (The execution time of tasks, or the total number 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
    is_prime[i] = test_primality(x[i]);
}
```

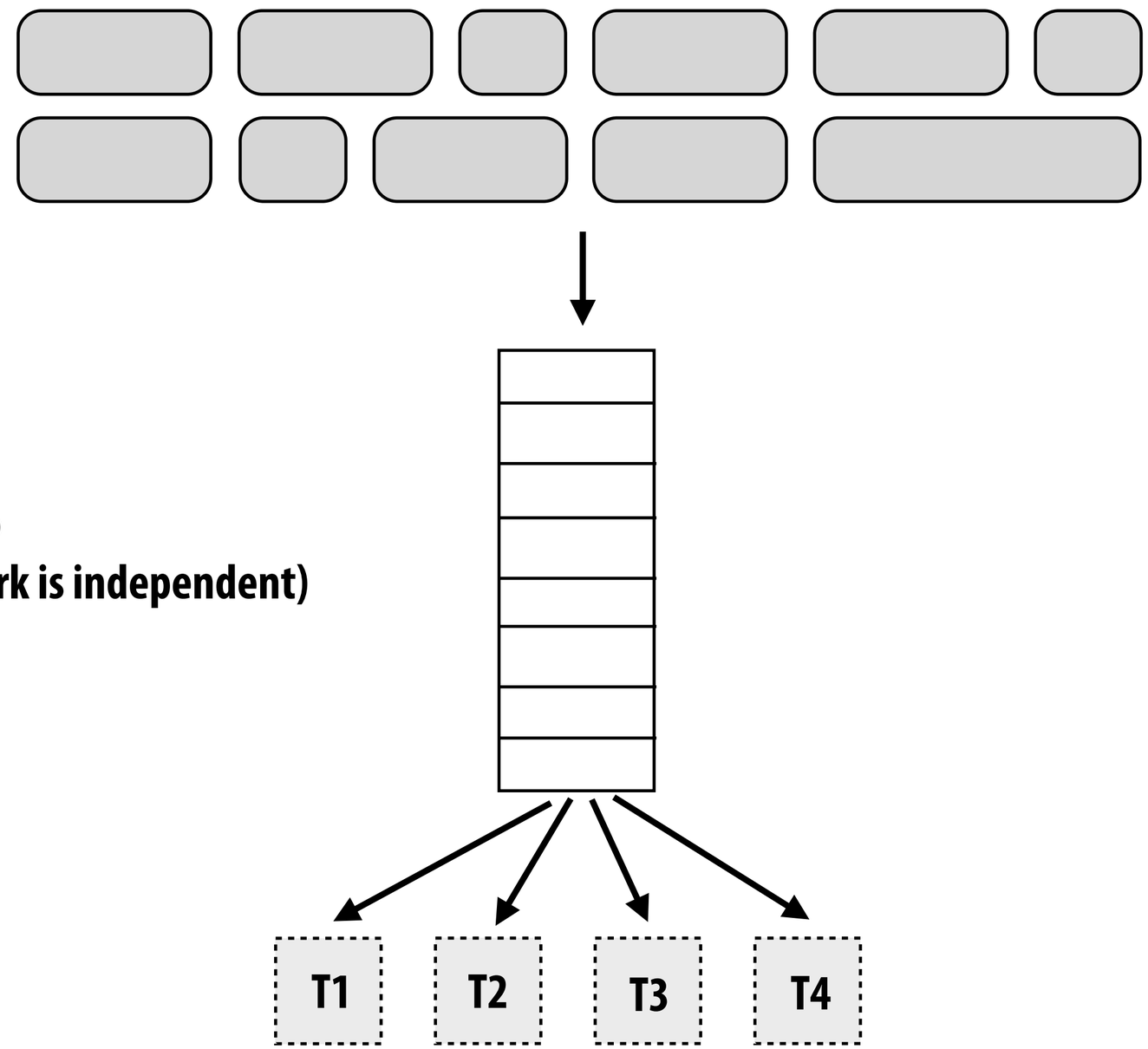
Parallel program
(SPMD execution of multiple threads, shared address space model)

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

// initialize elements of x

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

atomic_incr(counter);

Dynamic assignment using work queues

Sub-problems
(a.k.a. "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 shared work queue
Push new work to queue as it's created

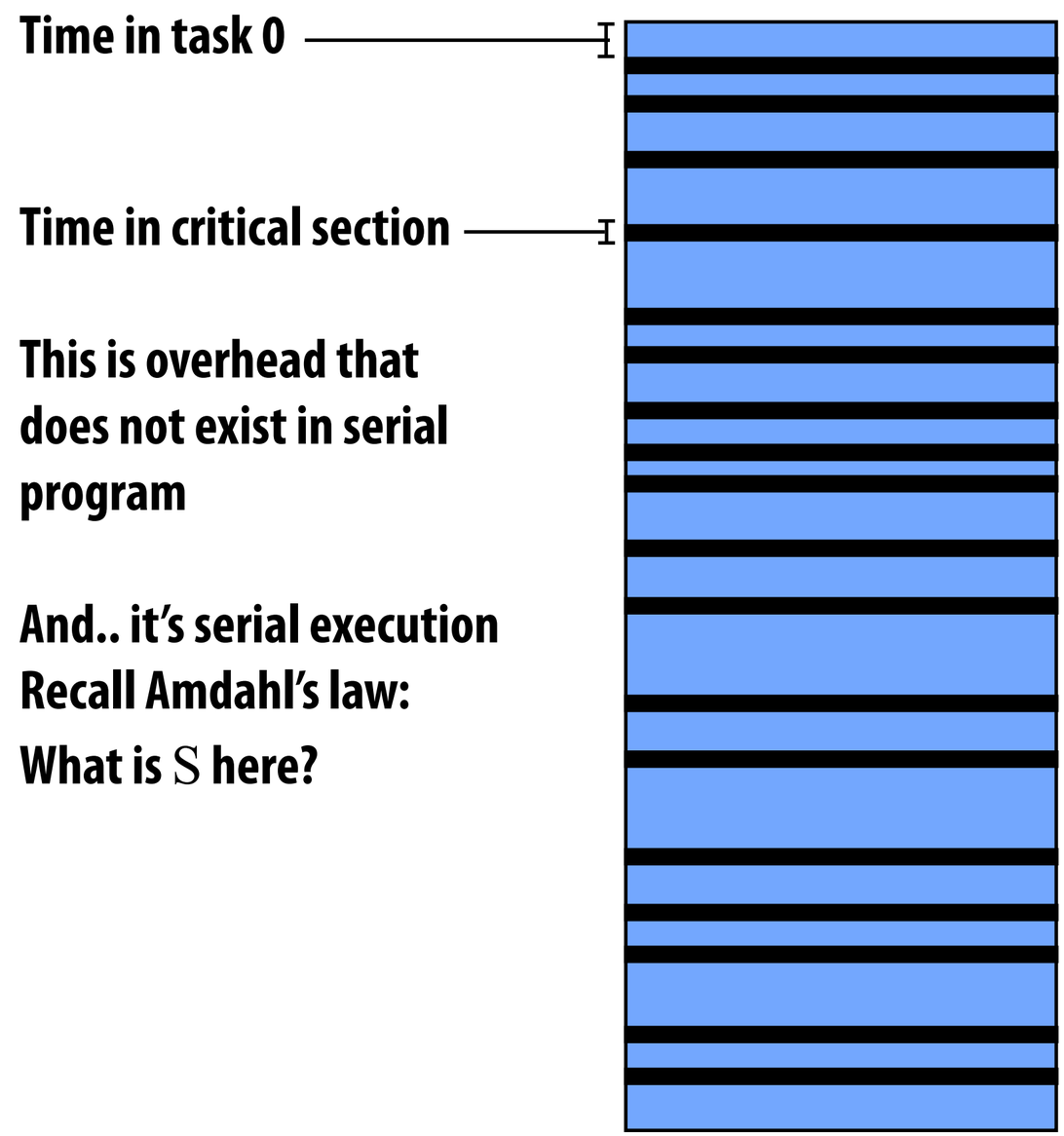
What constitutes a piece of work?

- **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;
  is_prime[i] = test_primality(x[i]);
}
```

Fine granularity partitioning:
Here: 1 "task" = 1 element

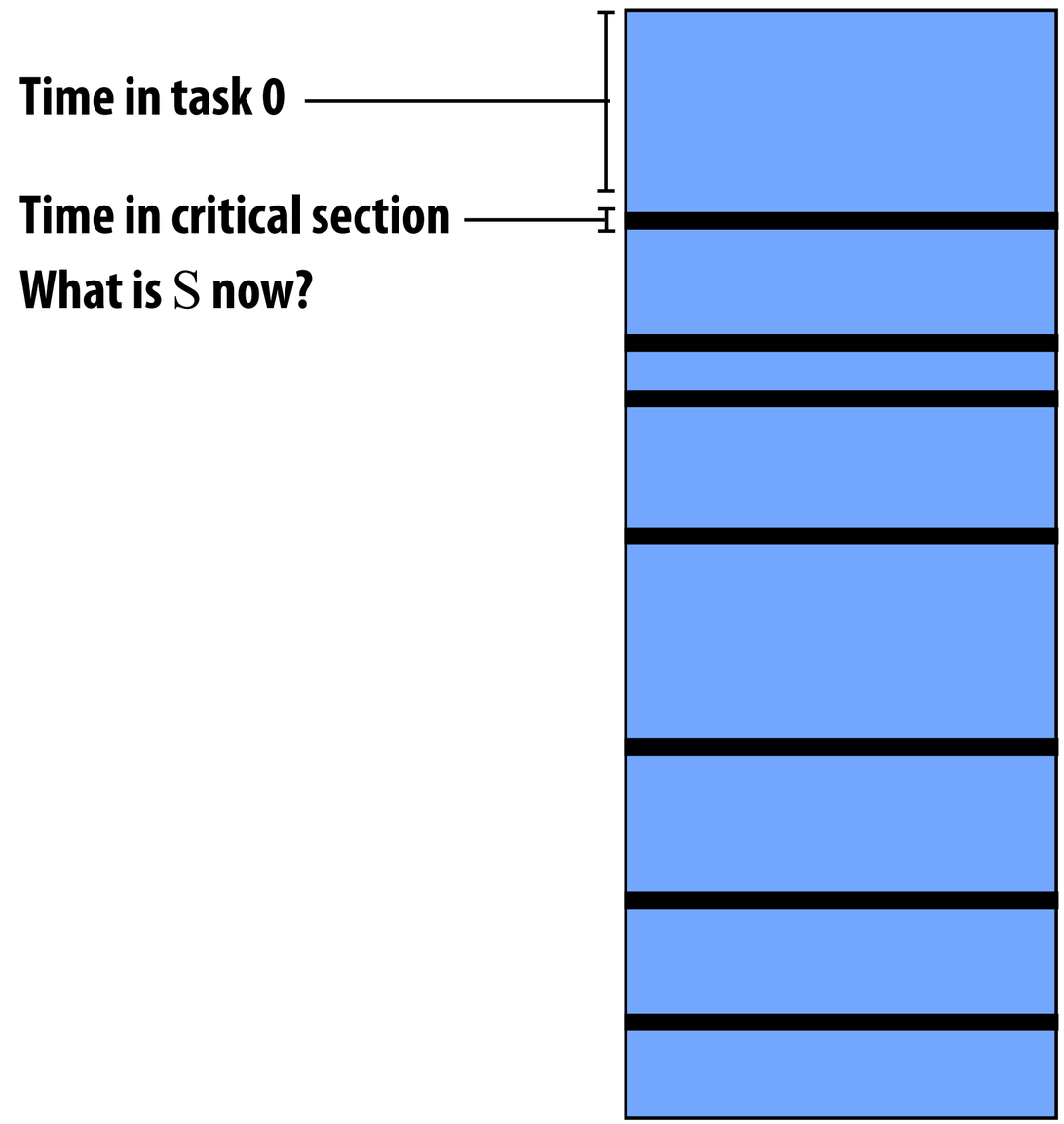
Likely <u>good</u> workload balance (many small tasks)
Potential for <u>high</u> synchronization cost
(serialization at critical section)



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++)
     is_prime[i] = test_primality(x[i]);
}
```



Coarse granularity partitioning:
1 "task" = 10 elements

Decreased synchronization cost
(Critical section entered 10 times less)

So... have we done better?

Rule of thumb

- **Useful to have many more tasks* than processors**
  (many small tasks enables good workload balance via dynamic assignment)
  - Motivates small granularity tasks

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

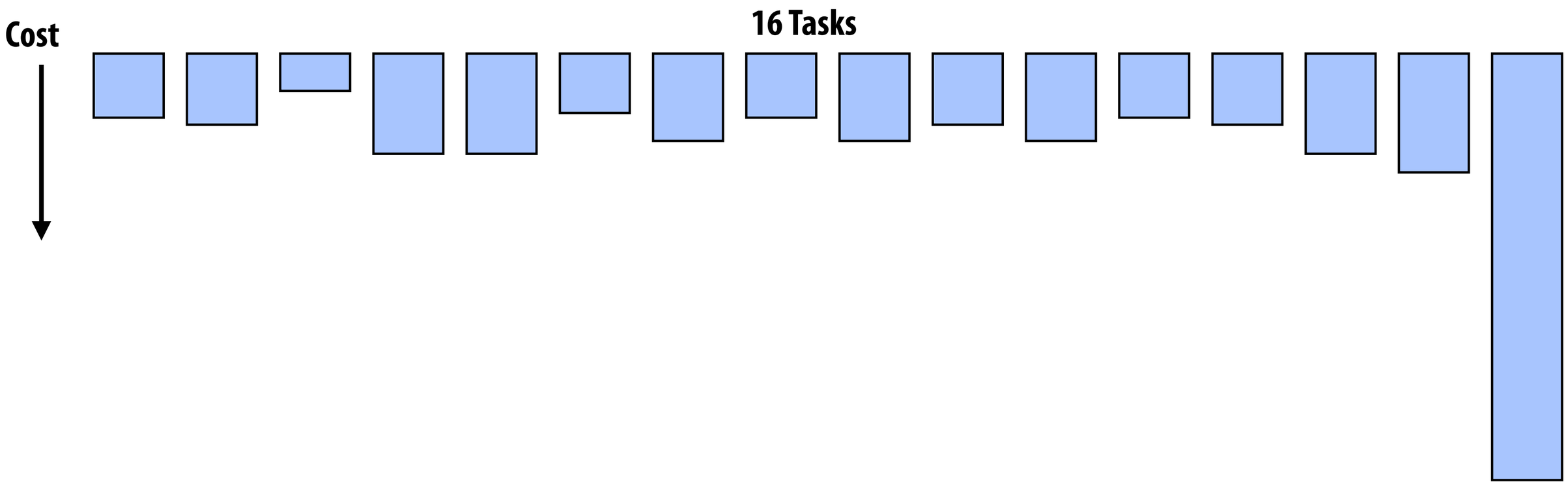
- **Ideal granularity depends on many factors**
  (Common theme in this course: must know your workload, and your machine)

* I had to pick a term. Here I'm using "task" generally: it's a piece of work, a sub-problem, etc.

Smarter task scheduling

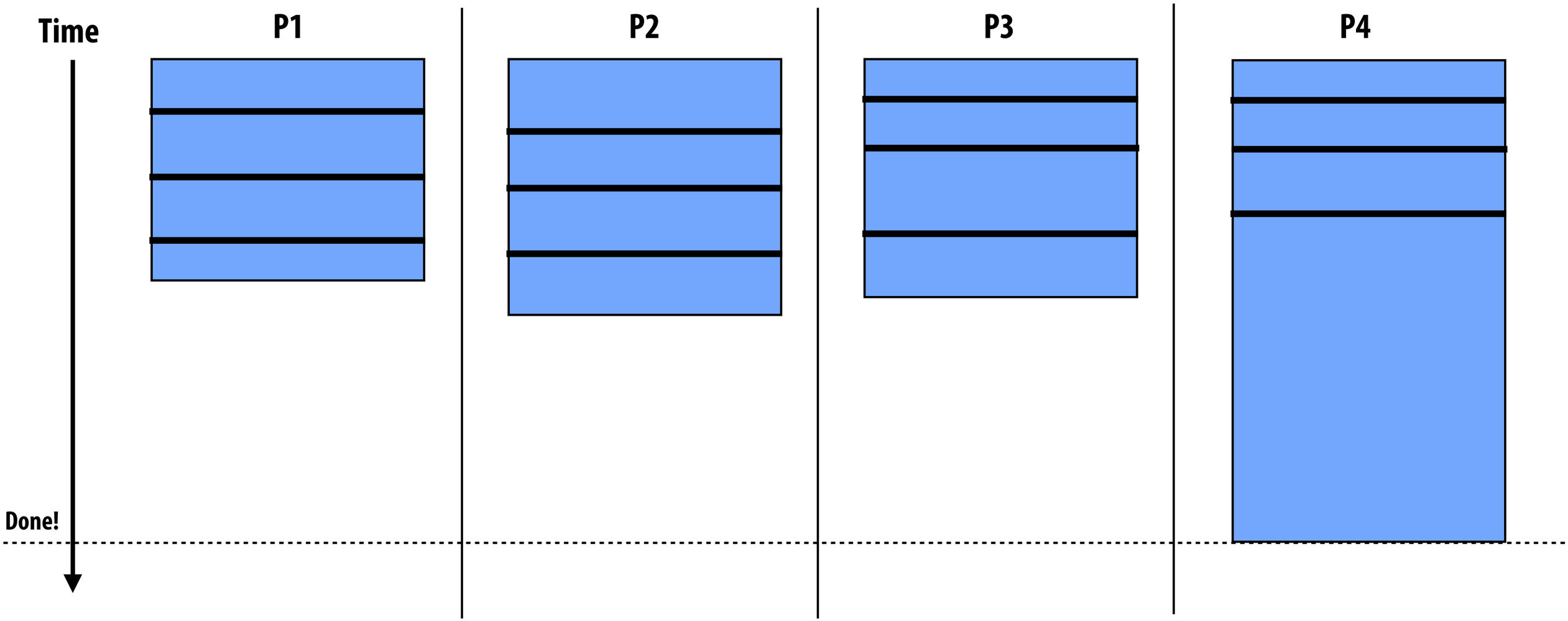
Consider dynamic scheduling via a shared work queue

What happens if the system assigns these tasks to workers in left-to-right order?



Smarter task scheduling

What happens if scheduler runs the long task last? Potential for load imbalance!



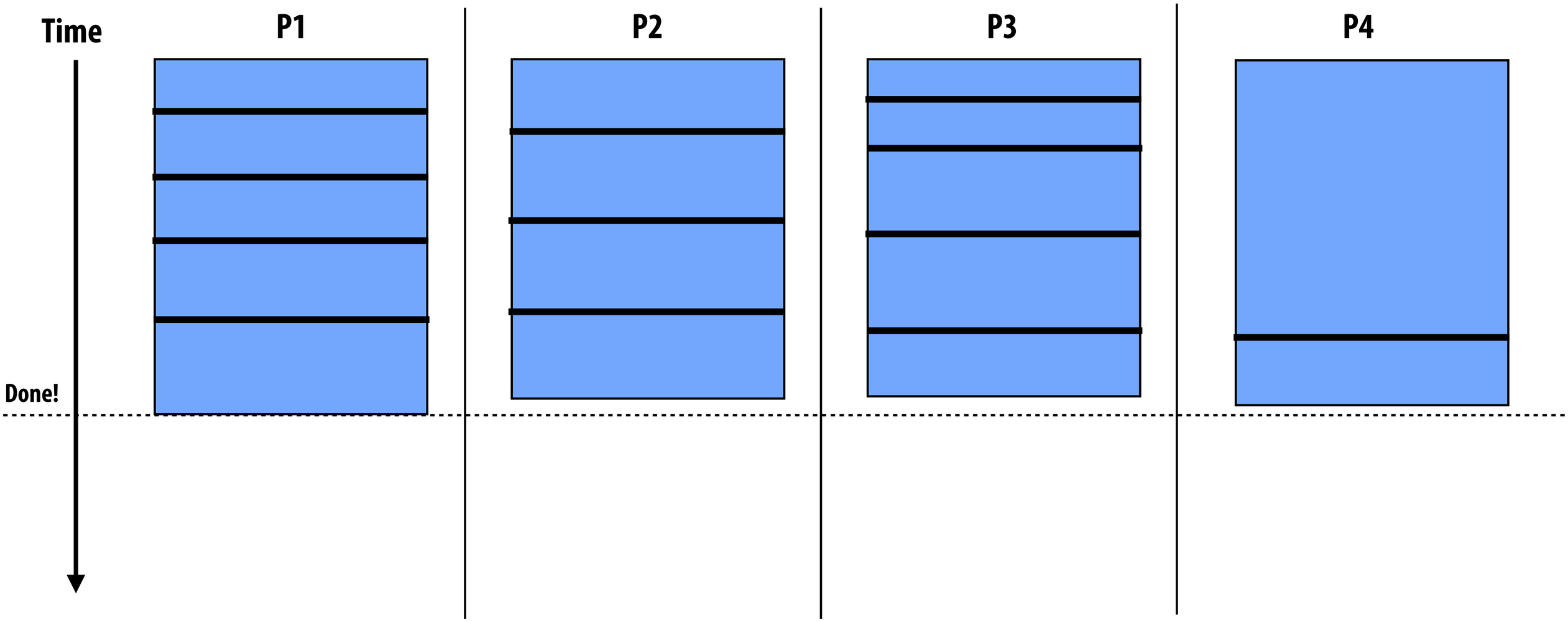
One possible solution to imbalance problem:

Divide work into a larger number of smaller tasks

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

Smarter task scheduling

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



Another solution: smarter scheduling

Schedule long tasks first

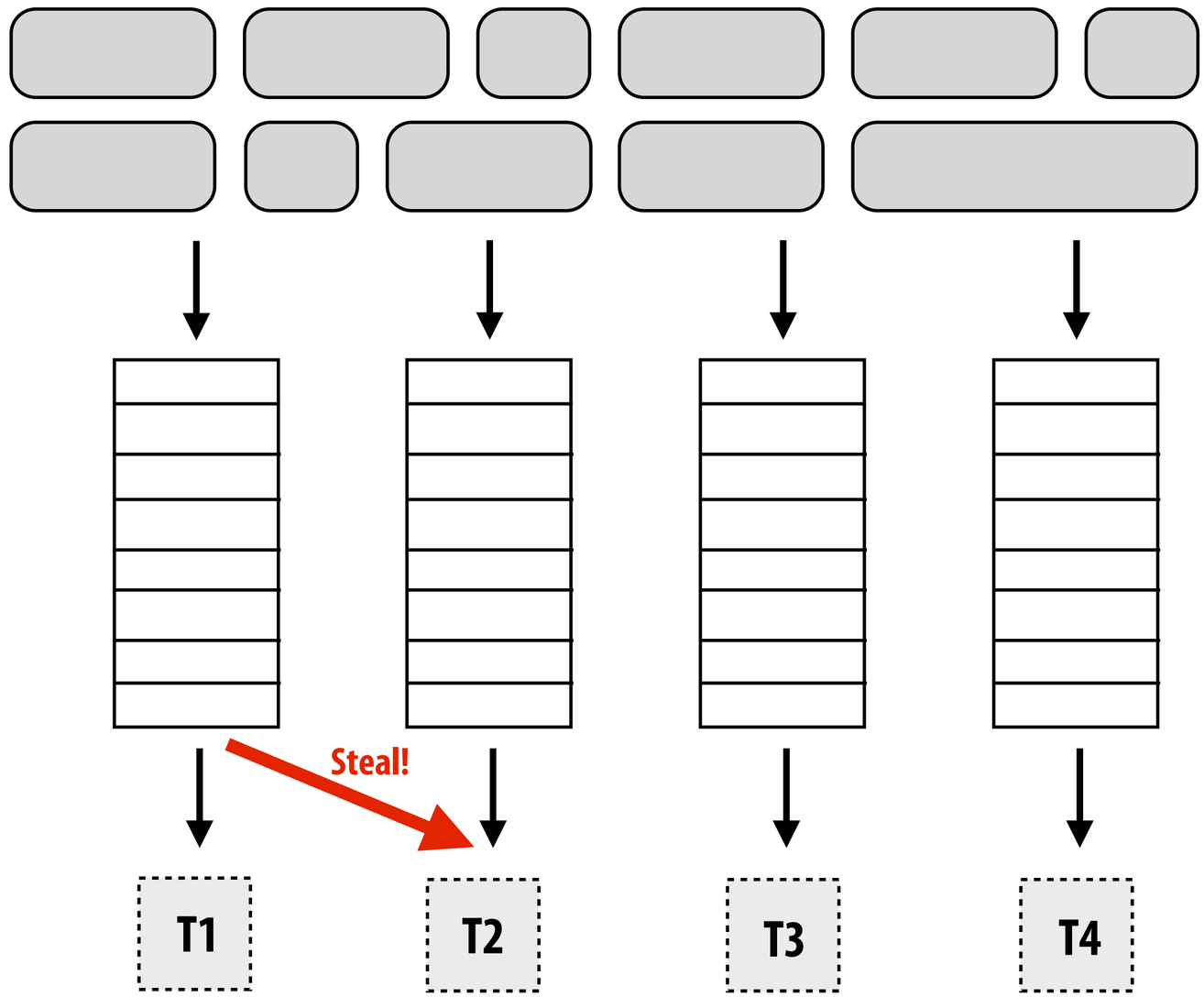
- Thread performing long task performs fewer overall tasks, but approximately the same amount of work as the other threads.
- Requires some knowledge of workload (some predictability of cost)

Decreasing synchronization overhead

- **Distributed work queues**
  - Replicate data to remove synchronization

Subproblems
(a.k.a. "tasks", "work to do")

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



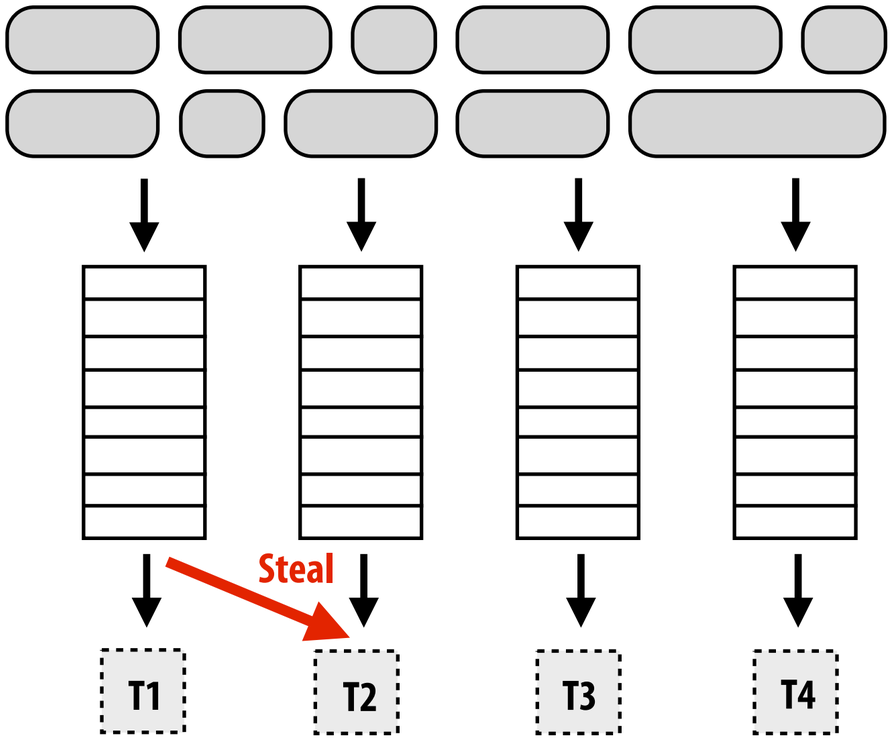
Worker threads:
Pull data from OWN work queue
Push new work to OWN work to queue
When localwork queue is empty...
STEAL work from another work queue

Distributed work queues

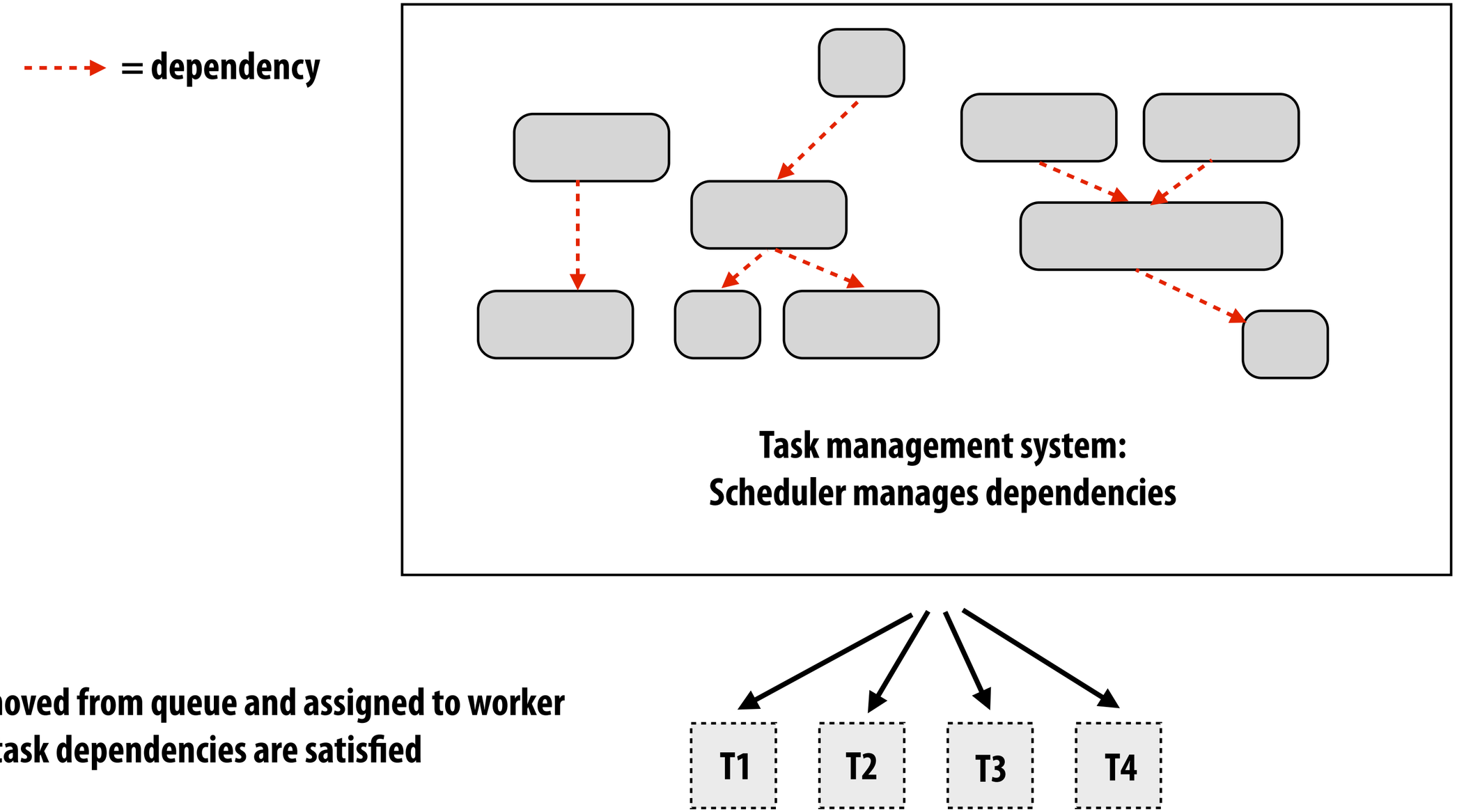
- **Costly synchronization/communication occurs during stealing**
  - But not every time a thread takes on new work
  - Stealing occurs <u>only when necessary</u> to ensure good load balance

- **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)



Work in task queues need not be independent



A task is not removed from queue and assigned to worker thread until all task dependencies are satisfied

Workers 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/assignment logic)
    - Minimize synchronization costs

- **Static assignment vs. dynamic assignment**
  - Really, it's not an either/or decision, there's a continuum of choices
  - Use up-front knowledge about workload as much as possible to reduce load imbalance and task management/synchronization costs (in the limit, if the system knows everything, use fully static assignment)

- **Issues discussed today span decomposition, assignment, and orchestration**