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 (overhead) performed to increase parallelism, manage assignment, reduce communication, etc.

- **We are going to talk about a rich space of techniques**

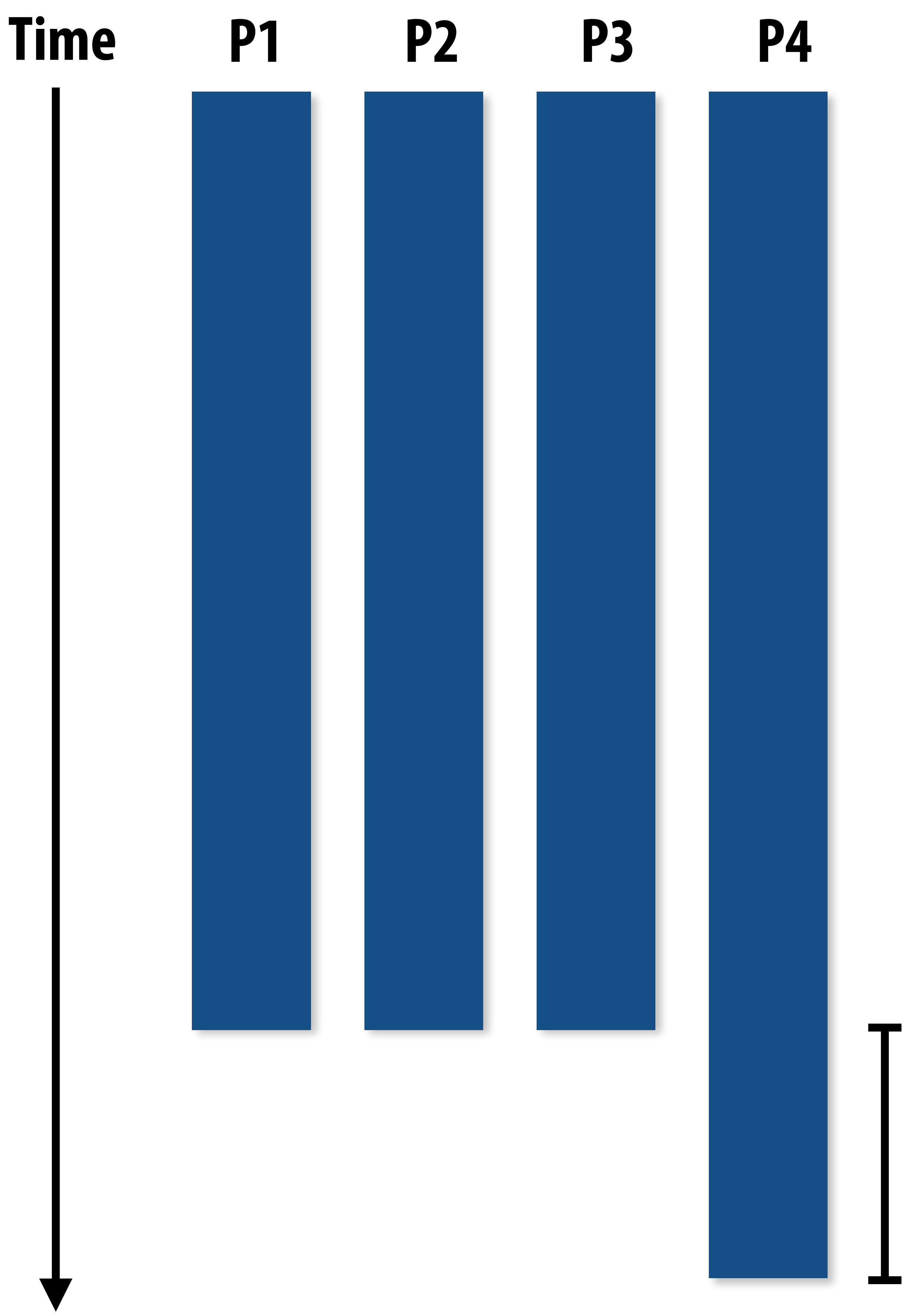
TIP #1: Always implement the simplest solution first, then measure performance to determine if you need to do better.

"My solution 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 creates and 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)



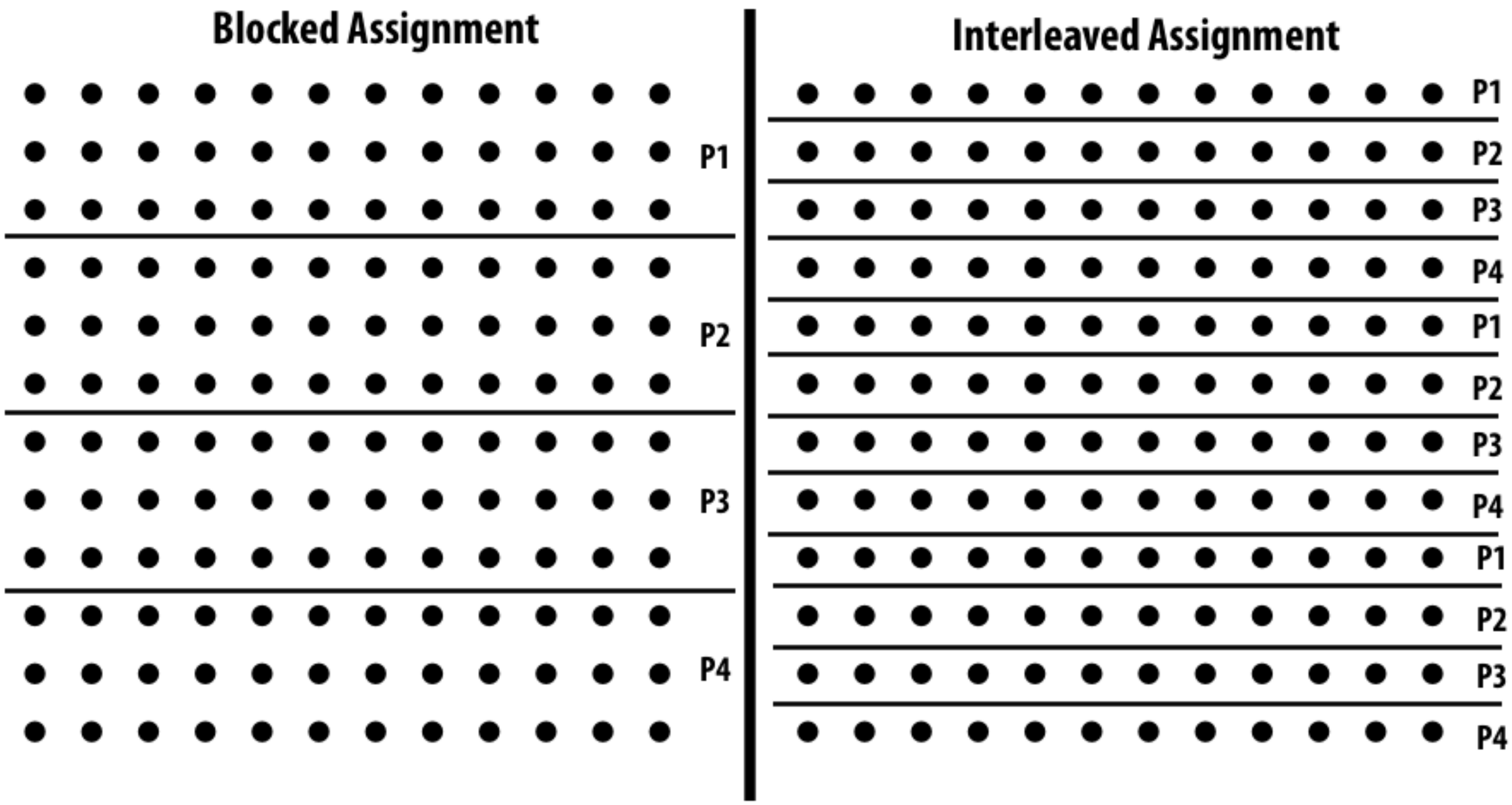
Recall Amdahl's Law:
Only small amount of load imbalance can significantly bound maximum speedup

P4 does 20% more work → P4 takes 20% longer to complete
→ 20% of parallel program's runtime is serial execution

(work in serialized section here is about 5% of the work of the whole program: S=.05 in Amdahl's law equation)

Static assignment

- Assignment of work to threads is pre-determined
  - Not necessarily determined at 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 (work) to each thread (worker)
  - We discussed two static assignments of work to workers (blocked and interleaved)



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

When is static assignment applicable?

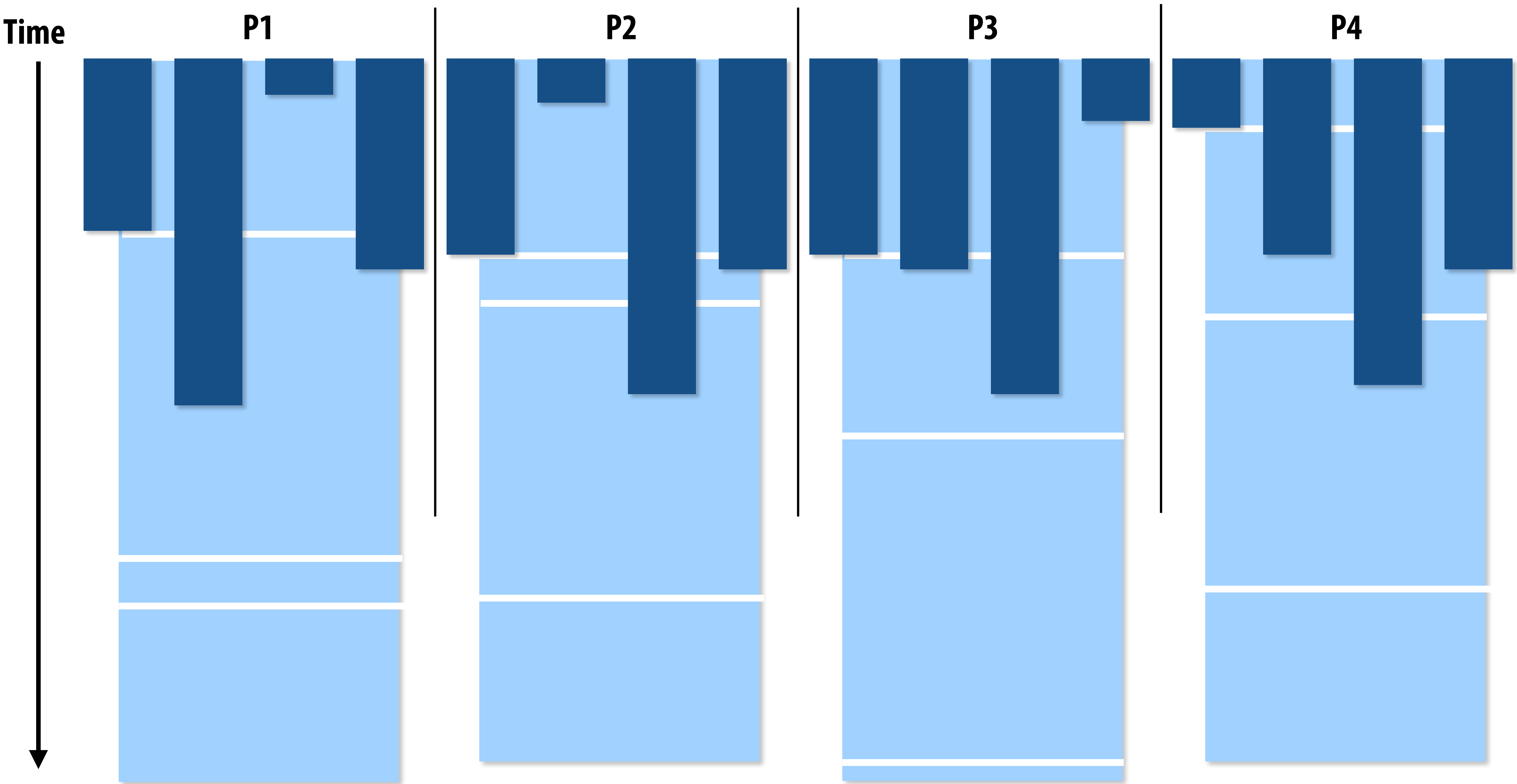
- When the cost (execution time) of work and the amount of work is <u>predictable</u> (so the programmer can work out a good assignment in advance)

- Simplest example: it is known up front that all work has the same cost

In the example above:
There are 12 tasks, and it is known that each have the same cost.
Assignment solution: statically assign three tasks to each of the four processors.

When is static assignment applicable?

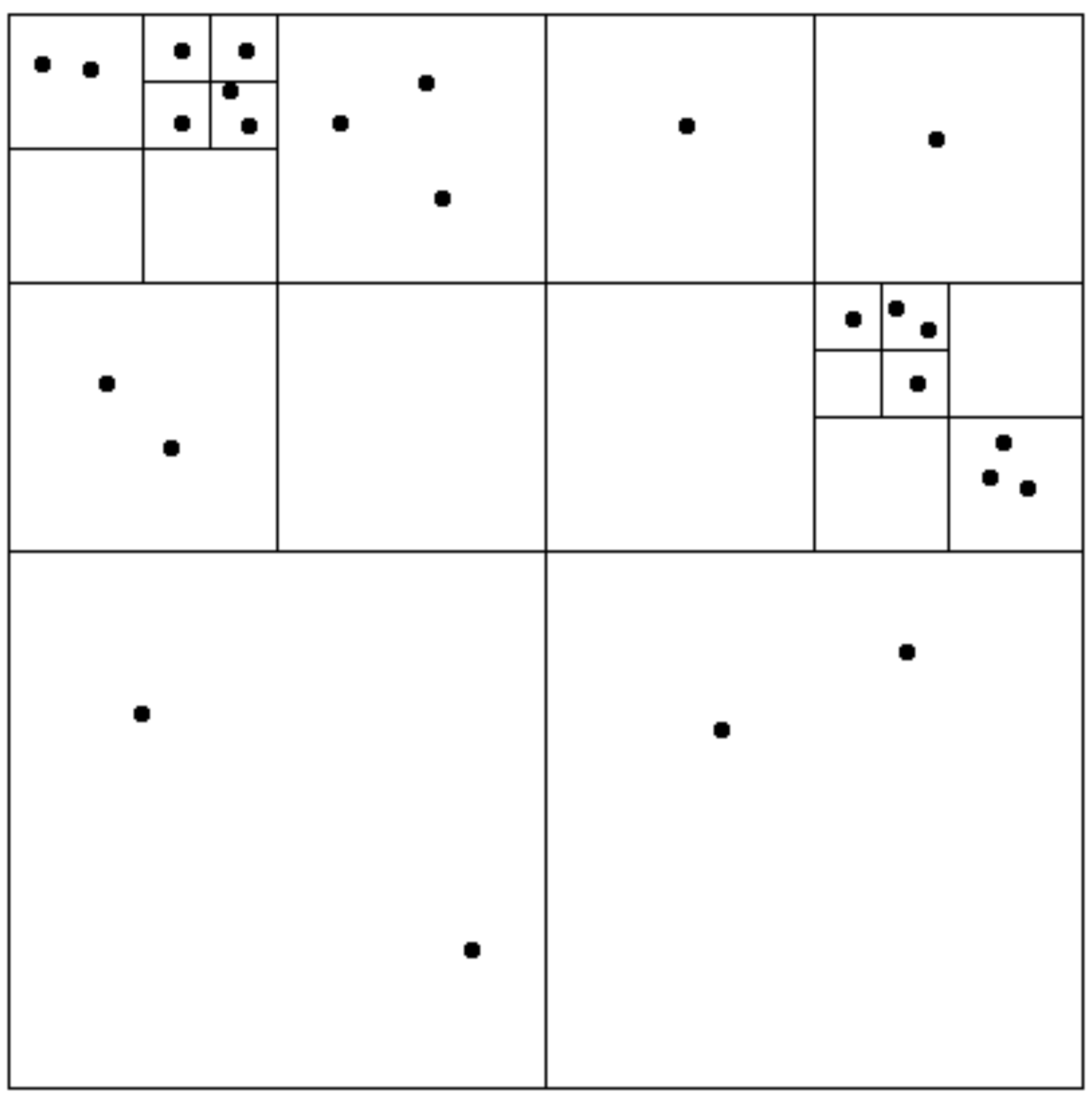
- When work is predictable, but not all jobs have same cost (see example below)
- When statistics about execution time are known (e.g., same cost on average)



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

"Semi-static" assignment

- Cost of work is predictable for near-term future
  - Idea: recent past good predictor of near future
- Application periodically profiles application and re-adjusts assignment
  - Assignment is "static" for the interval between re-adjustments



Particle simulation:

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

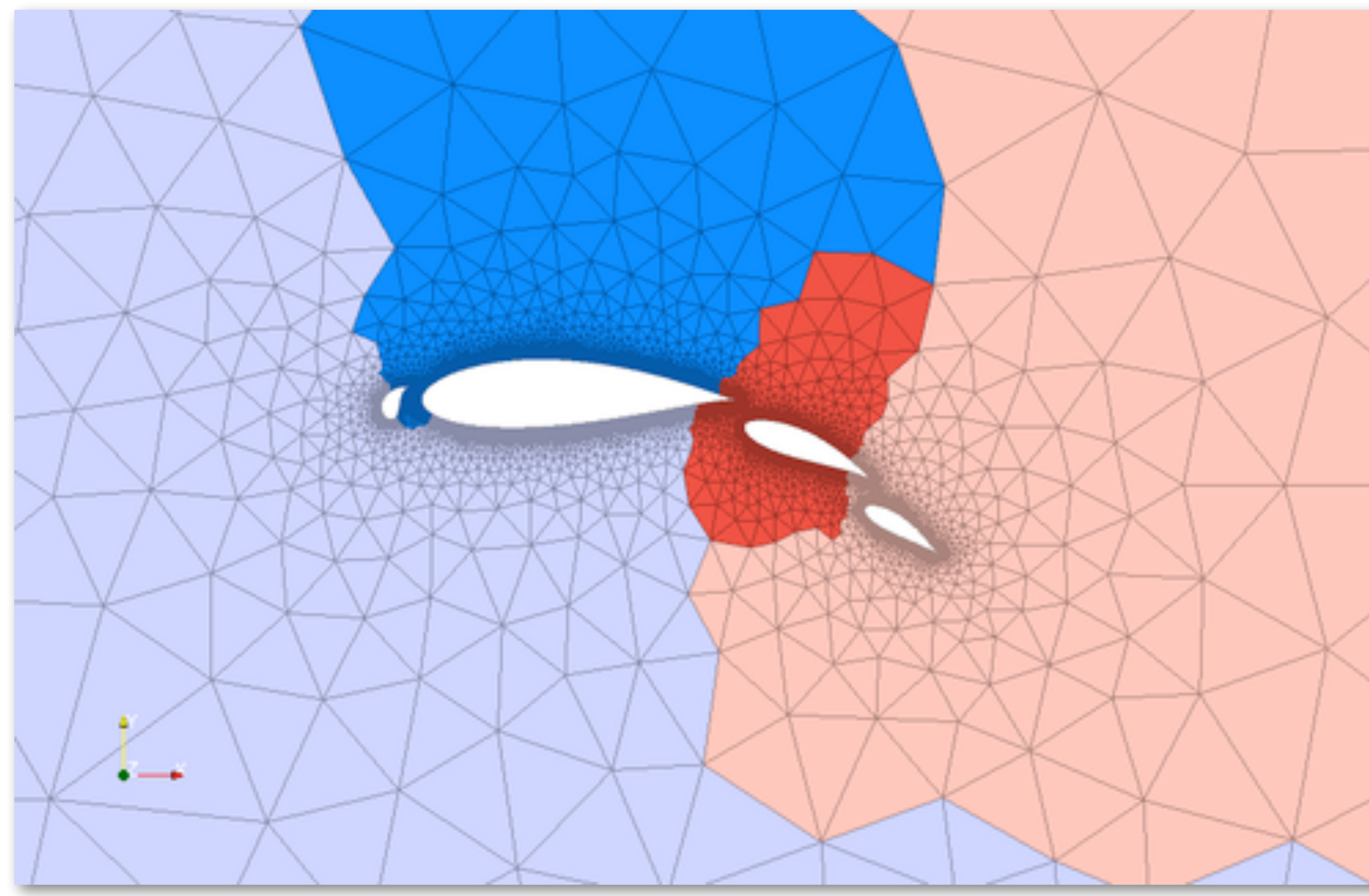


Image credit: http://typhon.sourceforge.net/spip/spip.php?article22

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

Program determines assignment dynamically 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 here

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

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

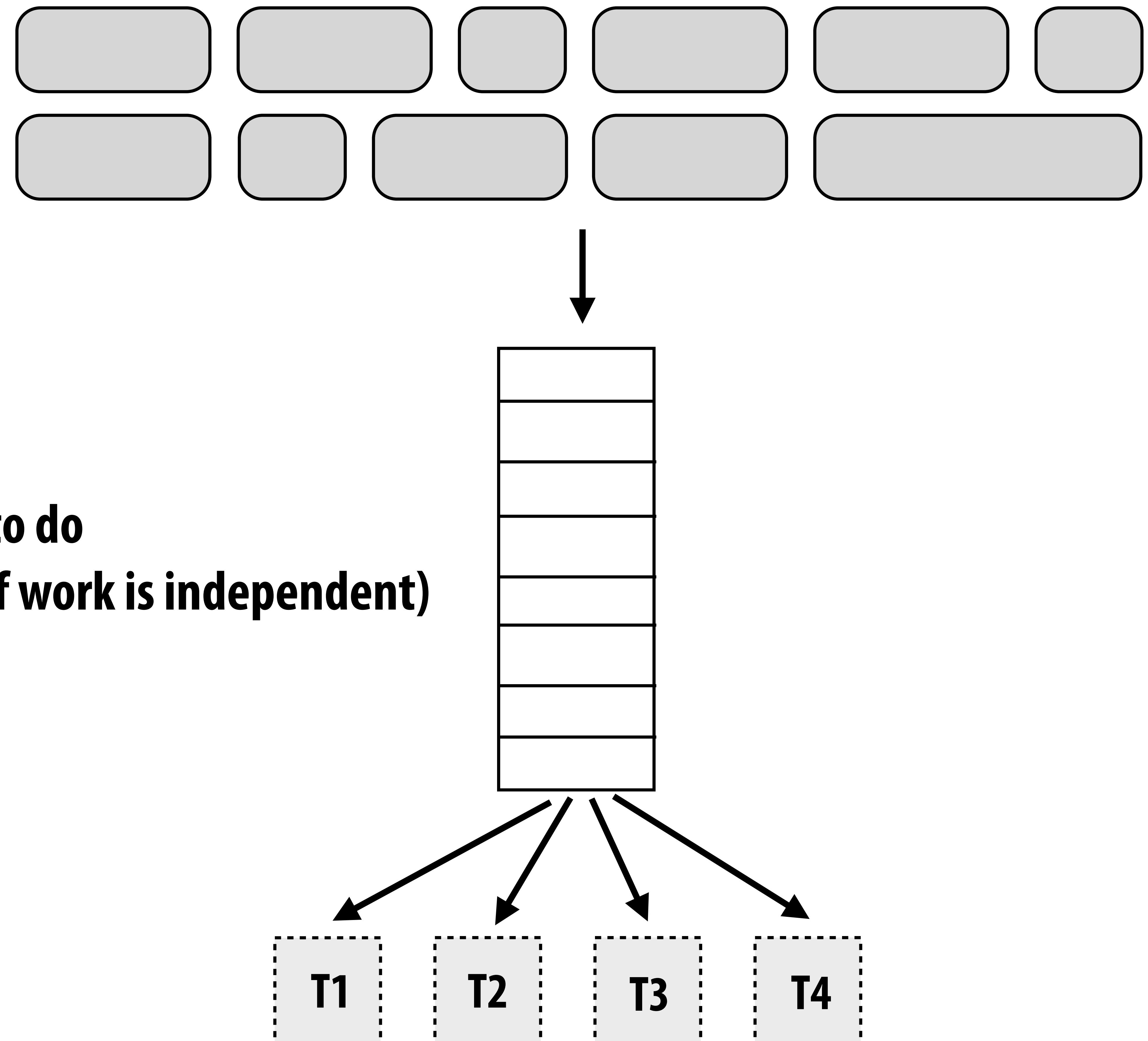
```
int N = 1024;
// assume allocations are only executed by 1 thread
int* x = new int[N];
bool* is_prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;    // shared variable

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 a work queue

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 is created

What constitutes a piece of work?

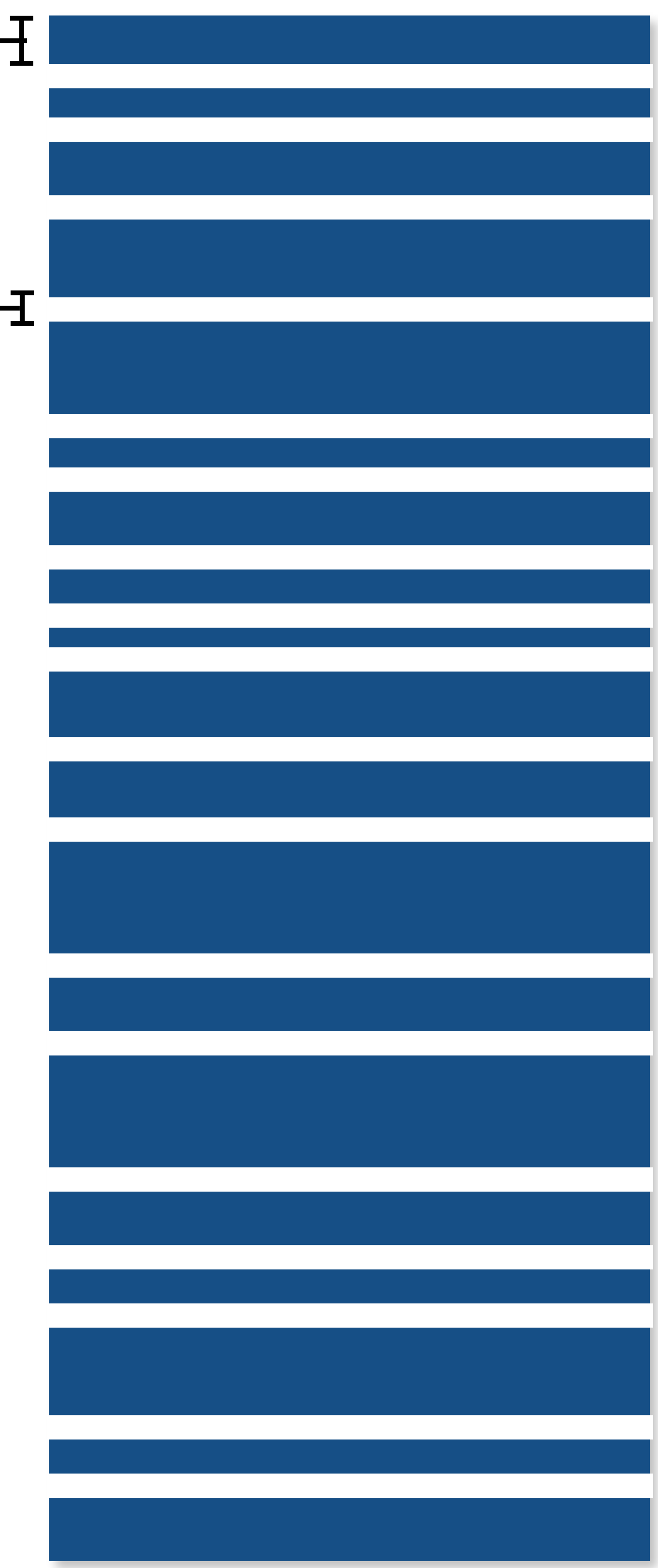
What is a potential problem with this implementation?

```
const int N = 1024;
// assume allocations are only executed by 1 thread
float* x = new float[N];
bool* prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;

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: 1 "task" = 1 element

Time in task 0

Time in critical section



This is overhead that does not exist in serial program

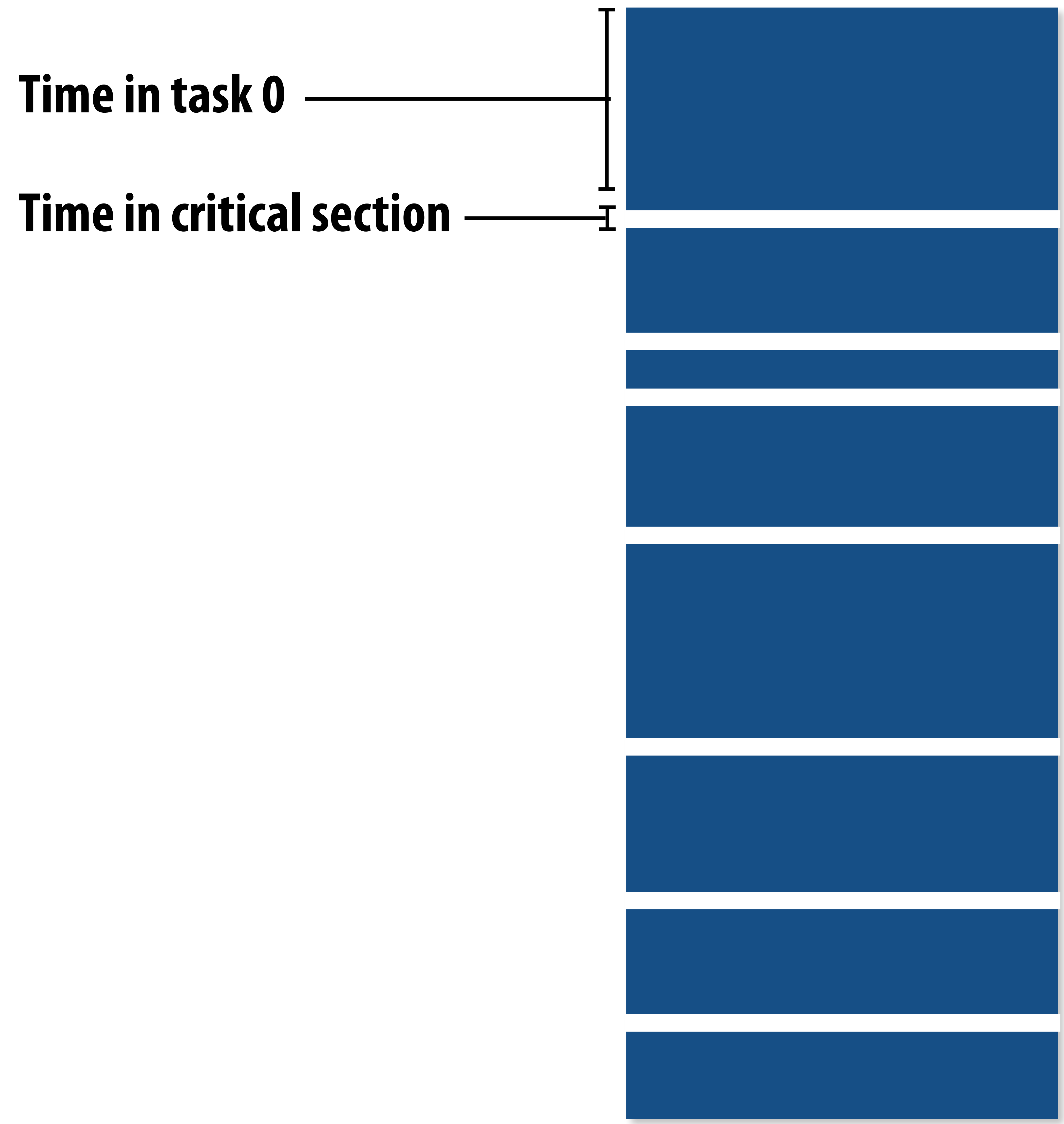
And.. it's serial execution (recall Amdahl's Law)

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

So... IS IT a problem?

Increasing task granularity

```
const int N = 1024;
const int GRANULARITY = 10;
// assume allocations are only executed by 1 thread

float* x = new float[N];
bool* prime = new bool[N];

// initialize elements of x here

LOCK counter_lock;
int counter = 0;

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)

Choosing task size

- **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**

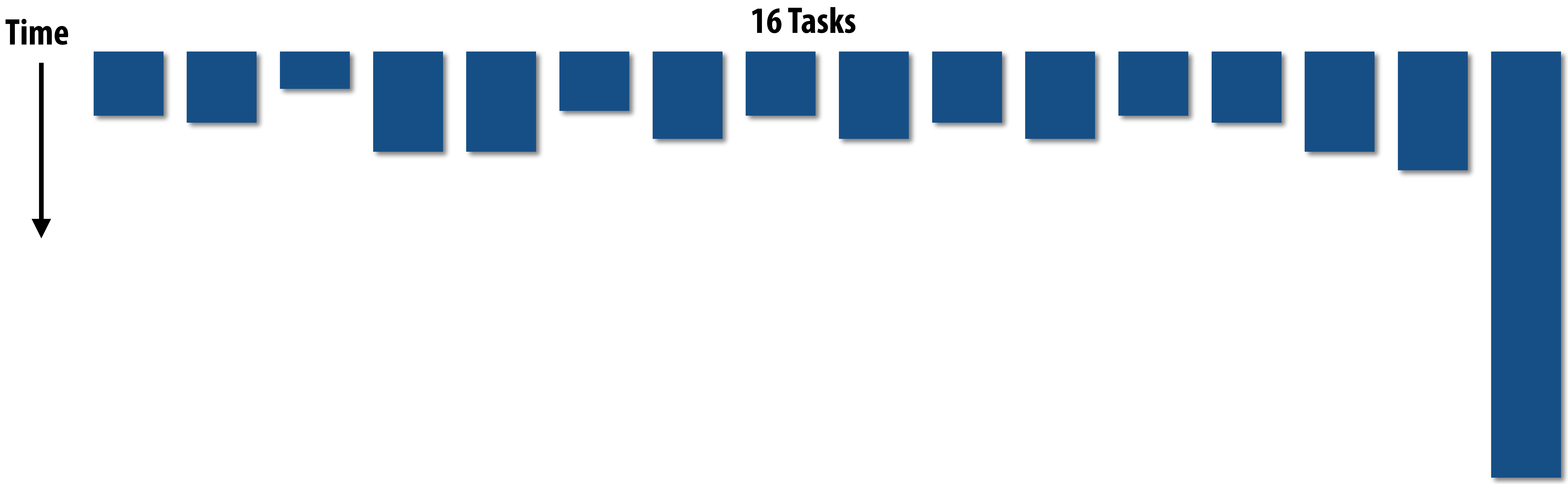
- **Ideal granularity depends on many factors**
  (Common theme in this course: must know your workload, and your machine)

*** I had to pick a term for a piece of work, a sub-problem, etc.**

Smarter task scheduling

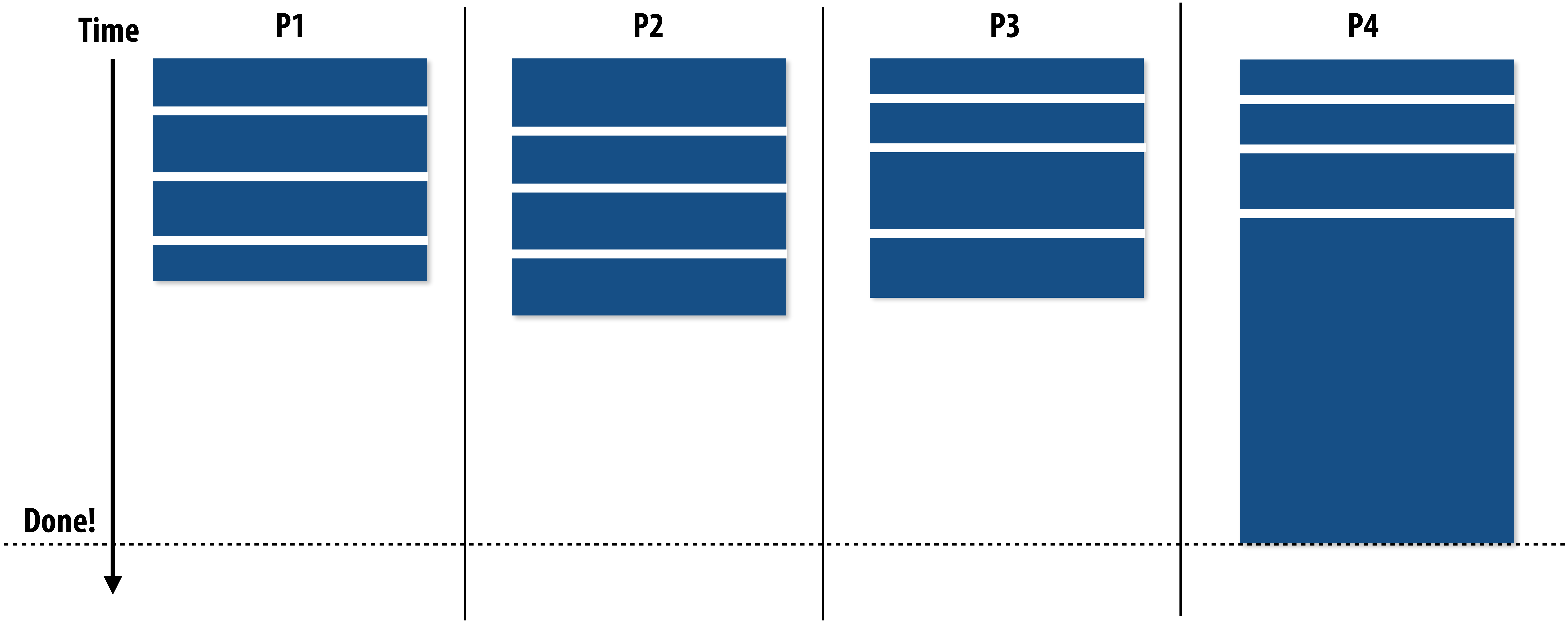
Consider dynamic scheduling via a shared work queue

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!



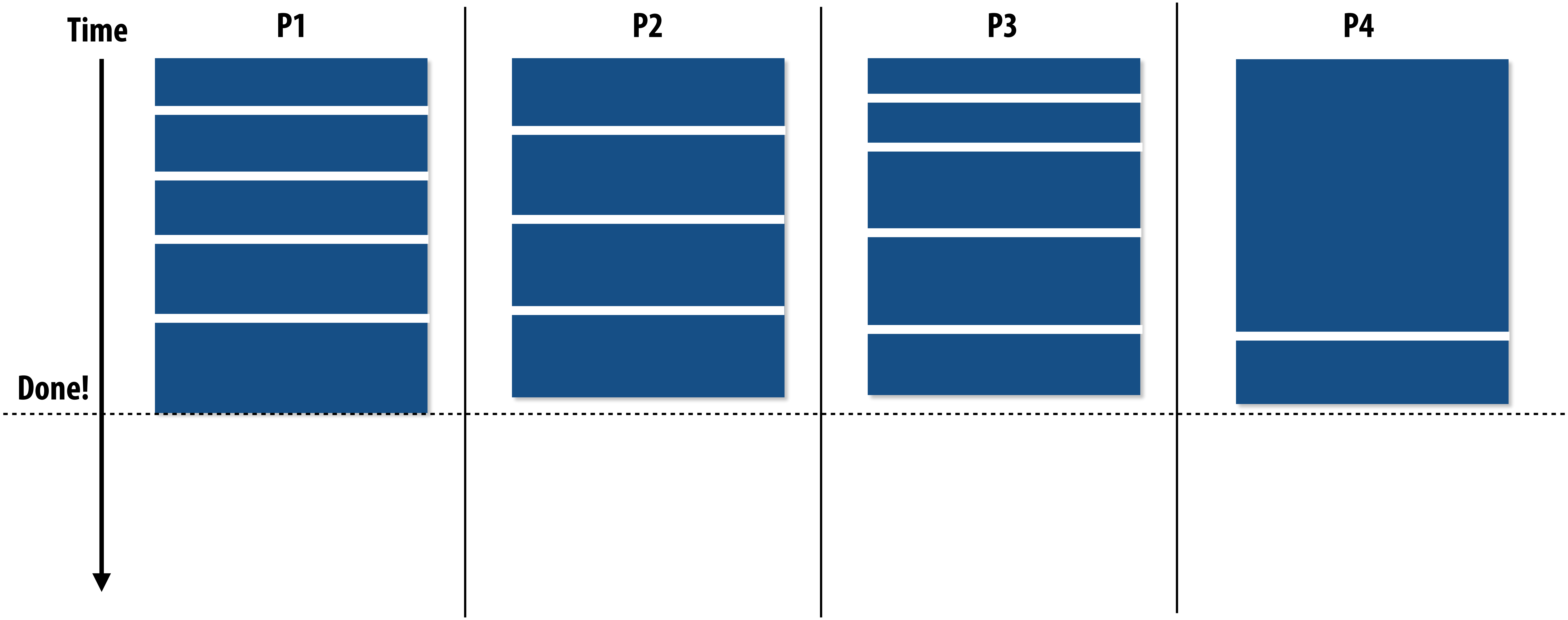
One possible solution to imbalance problem:

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)