Parallel and Distributed Computing (B4B36PDV)

Matěj Kafka, Michal Jakob

kafkamat@fel.cvut.cz

https://pdv.pages.fel.cvut.cz

Coding session 1 (leftovers from lecture 3)

OPENMP CANCELLATION AND NESTED PARALLELISM

How to solve problems faster?

First, solve them serially.

1. Define a problem.
2. Think about it.
3. Implement a serial solution.
4. Is it fast enough? You're finished. *
5. Is it easy to make it faster? Go to step 2. *
6. Think about it.
7. Is it hard to parallelize? Go to step 2. *

...

*All conditions are quite fuzzy.

How to solve parallel problems faster?

Only then try to parallelize.

...

8. Decompose the problem into parallelizable parts.
9. Implement a parallel solution on the CPU.
10. Is it fast enough? You're finished. *
11. Can you easily improve the solution? Go to step 8. *
    - Can you make the decomposition more granular?
    - Can you use a different decomposition?
12. Is it possible to use SIMD? Use a GPU, go to step 8. *
13. Buy better hardware or get better at optimization.

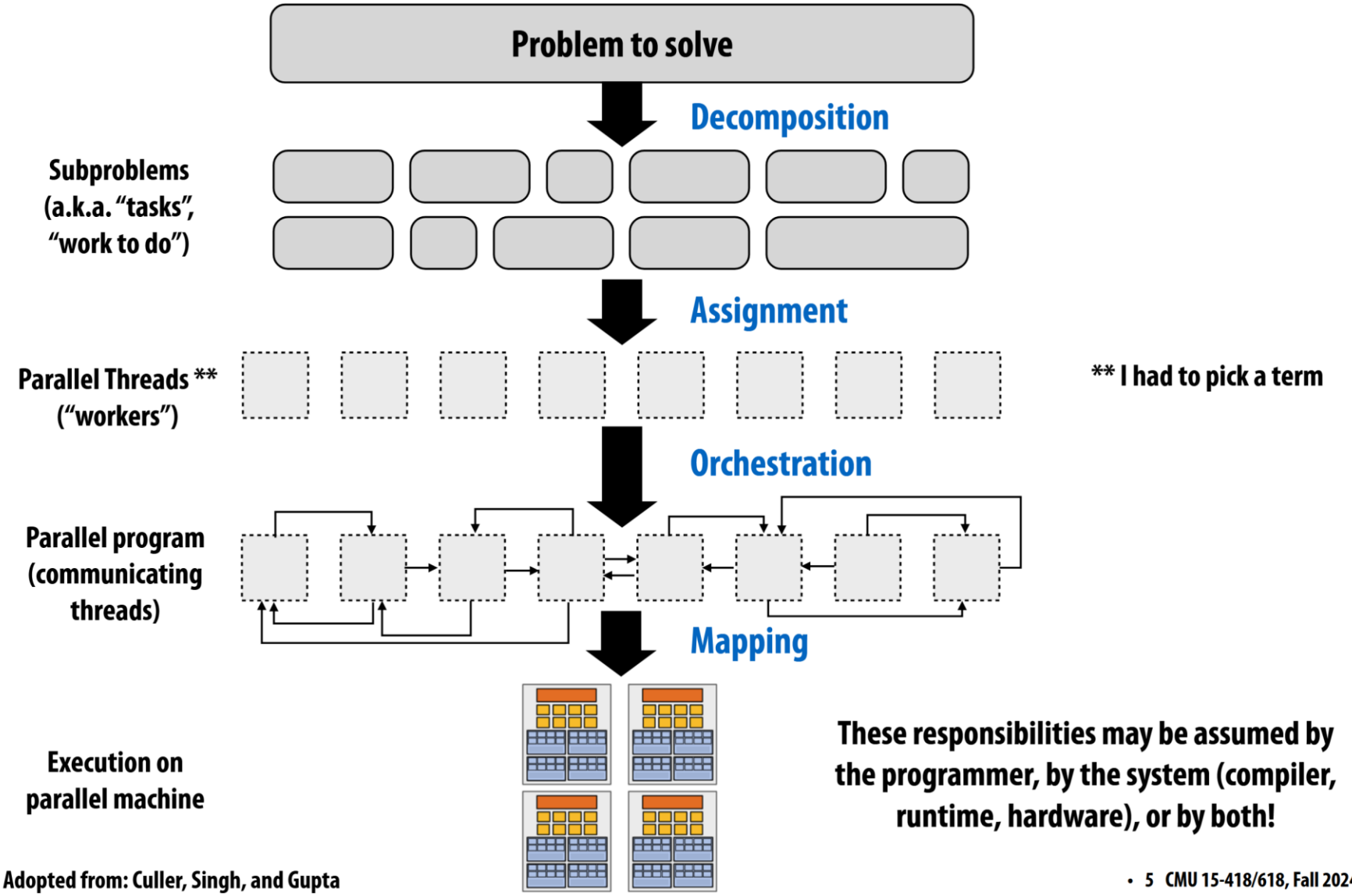
*All conditions are quite fuzzy.

"Decompose the problem into parallelizable parts."

DESIGNING PARALLEL ALGORITHMS

Designing a parallel algorithm

https://www.cs.cmu.edu/~15418/lectures/06_progbasics.pdf



Adopted from: Culler, Singh, and Gupta

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

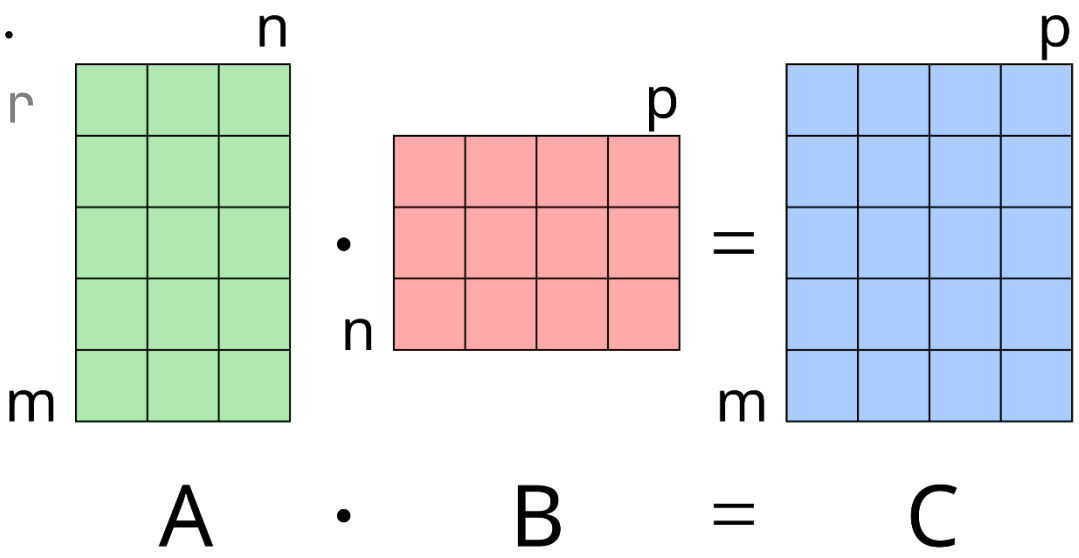
Main idea: create *at least* enough tasks to keep all execution units on a machine busy

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

Data decomposition

Split input/output data between threads.

- Partition the problem based on input / intermediate / output data.
- Applicable if we have the whole input up-front, and the operation is a somewhat straightforward mapping from the input to the output.
- Very common, typically the serial solution involves a loop over all input data.
  - `pragma omp for`

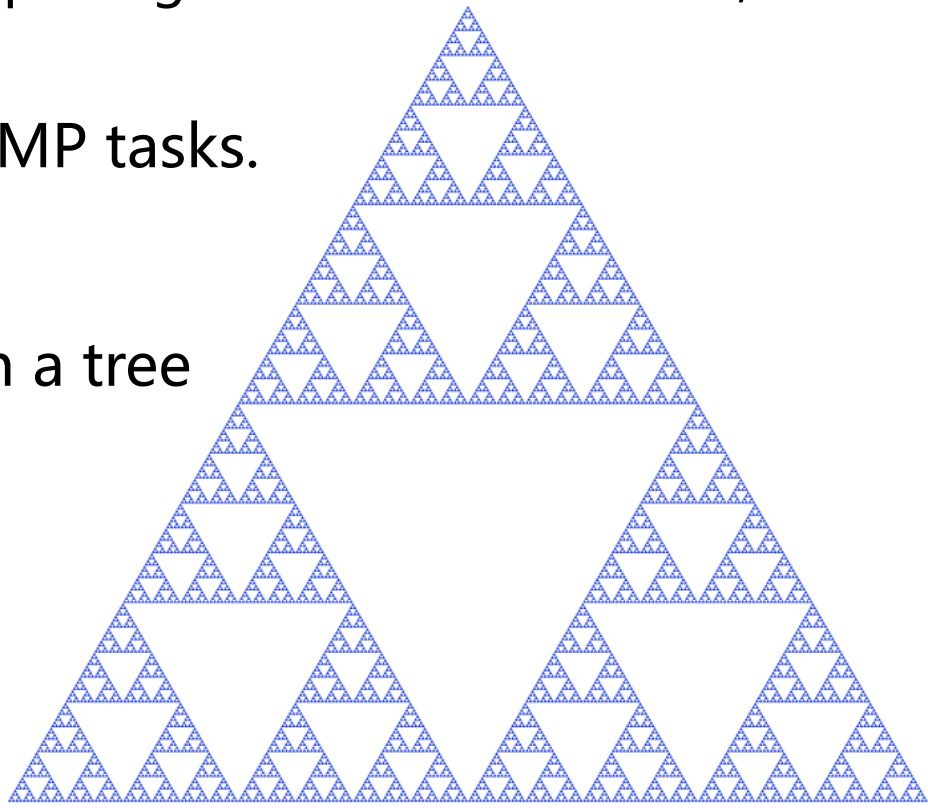


https://commons.wikimedia.org/wiki/File:Matrix_multiplication_qtl1.svg

Recursive decomposition

Recursively break the problem down into tasks.

- We know all parts of the problem up front but cannot easily decompose it directly.
- Divide and conquer.
  - Create tasks by recursively splitting the work into smaller, **mostly independent** parts.
- Typically a good fit for OpenMP tasks.
- Example: Quicksort
- Example: Most algorithms on a tree

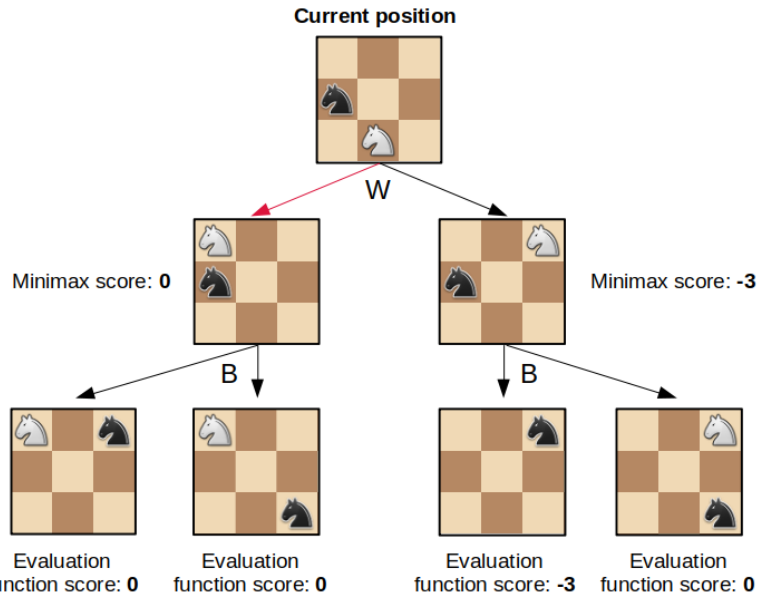


https://en.wikipedia.org/wiki/File:Sierpinski_triangle.svg

Explorative decomposition

Dynamically create tasks while exploring the state space.

- We discover parts of the problem space as we explore it.
- We typically do not have a good estimate for the size and structure of each discovered subproblem.
- Example: Many discrete optimalization problems
- Example: Game playing (e.g. Reversi from RPH)



https://www.codertime.org/minimax-chess-engine-programming-r/

Decomposition

Who is responsible for performing decomposition?

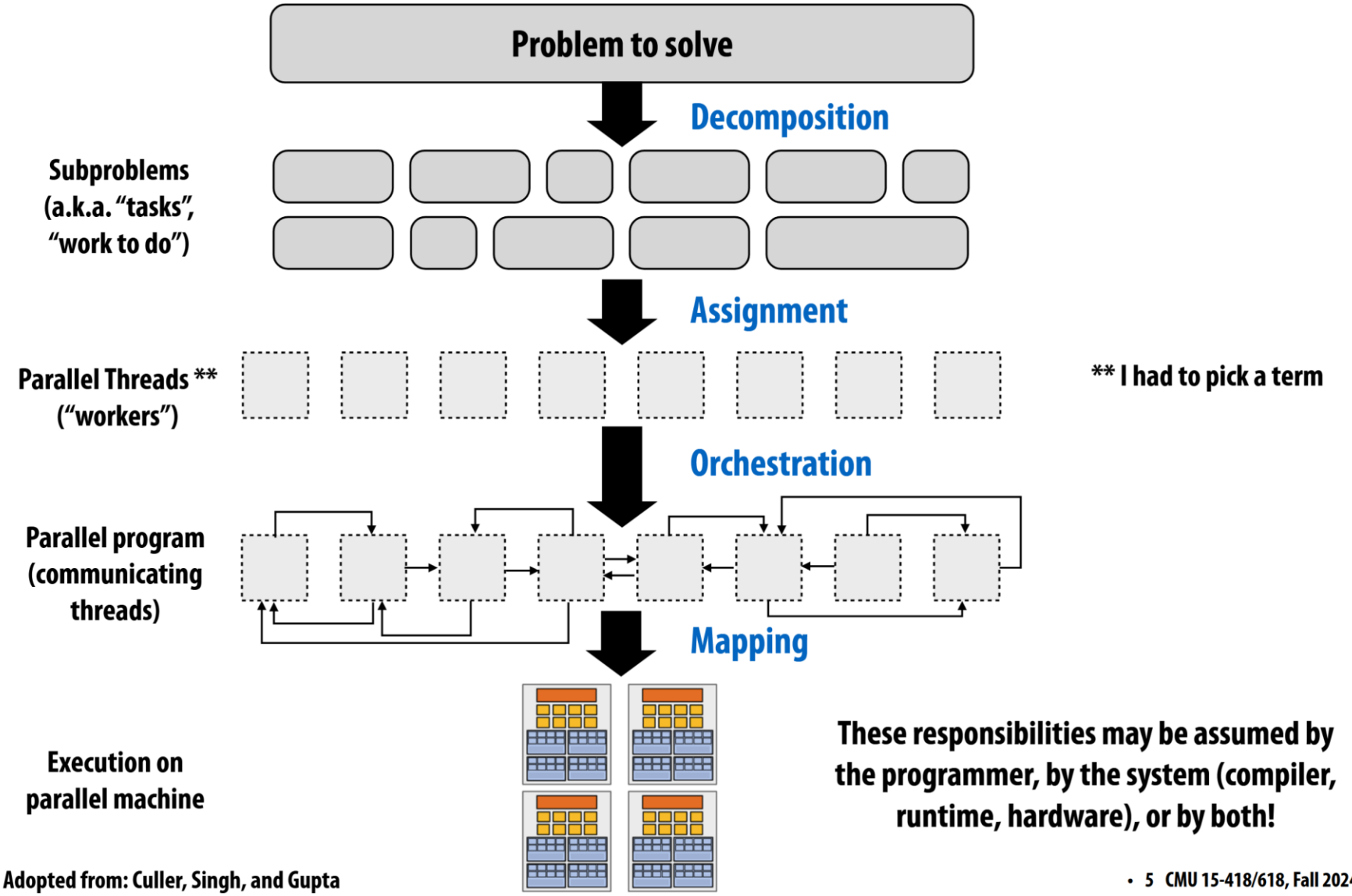
- In most cases: the programmer

Automatic decomposition of sequential programs continues to be a challenging research problem (very difficult in general case)

- Compiler must analyze program, identify dependencies
  - What if dependencies are data dependent (not known at compile time)?
- Researchers have had modest success with simple loop nests
- The "magic parallelizing compiler" for complex, general-purpose code has not yet been achieved

Assignment

Which thread executes which task?

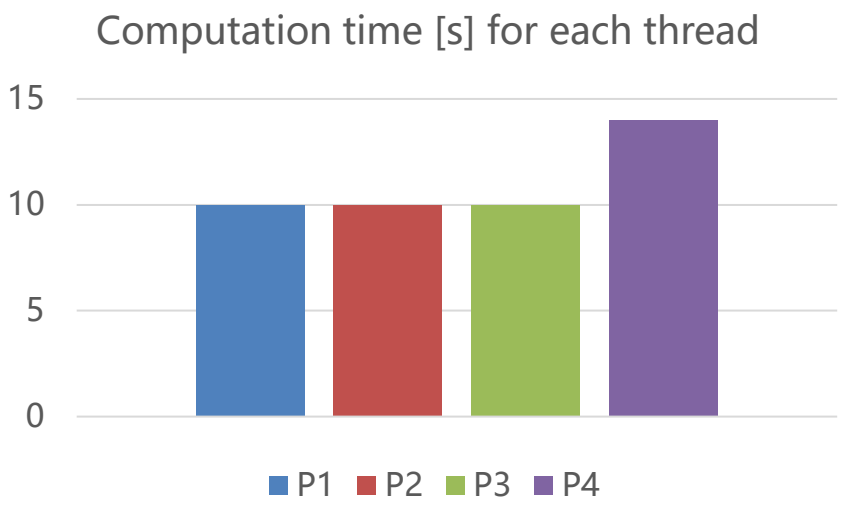


Adopted from: Culler, Singh, and Gupta

Assignment

Which thread executes which task?

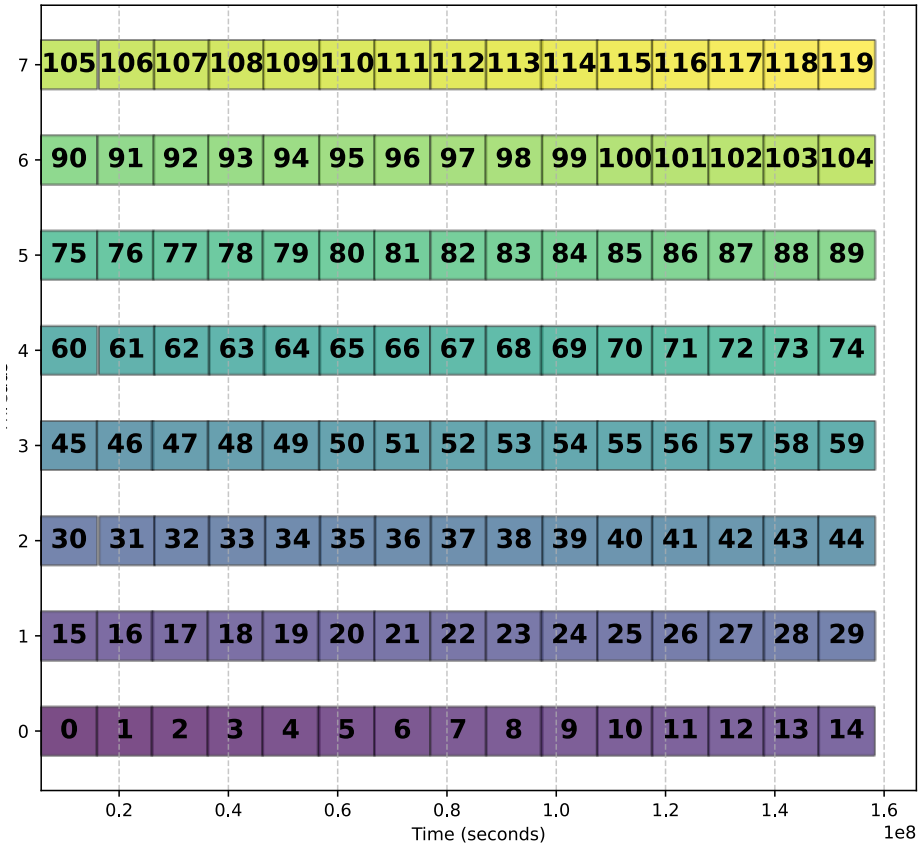
- Assign each task to a thread.
- Assignment will significantly affect runtime performance.
  - Synchronization/dependencies between tasks on the same thread is typically free.
  - Not the case for tasks on different threads.
- You already know about OpenMP loop scheduling.
- Goals:
  - Balance workload.
  - Minimize communication.
  - Minimize duplication.

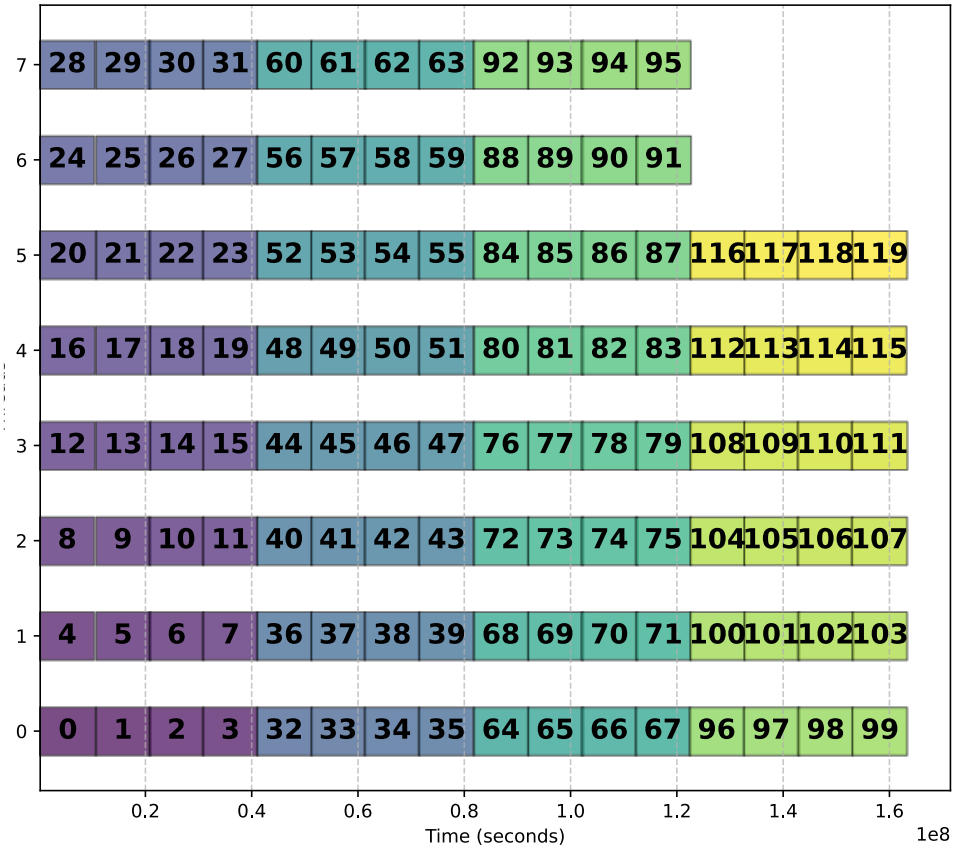


Static assignment

Ahead-of-time, no load balancing.

- Each thread is assigned some tasks at the beginning of the computation (may still depend on runtime parameters).
- Should have almost zero runtime overhead.





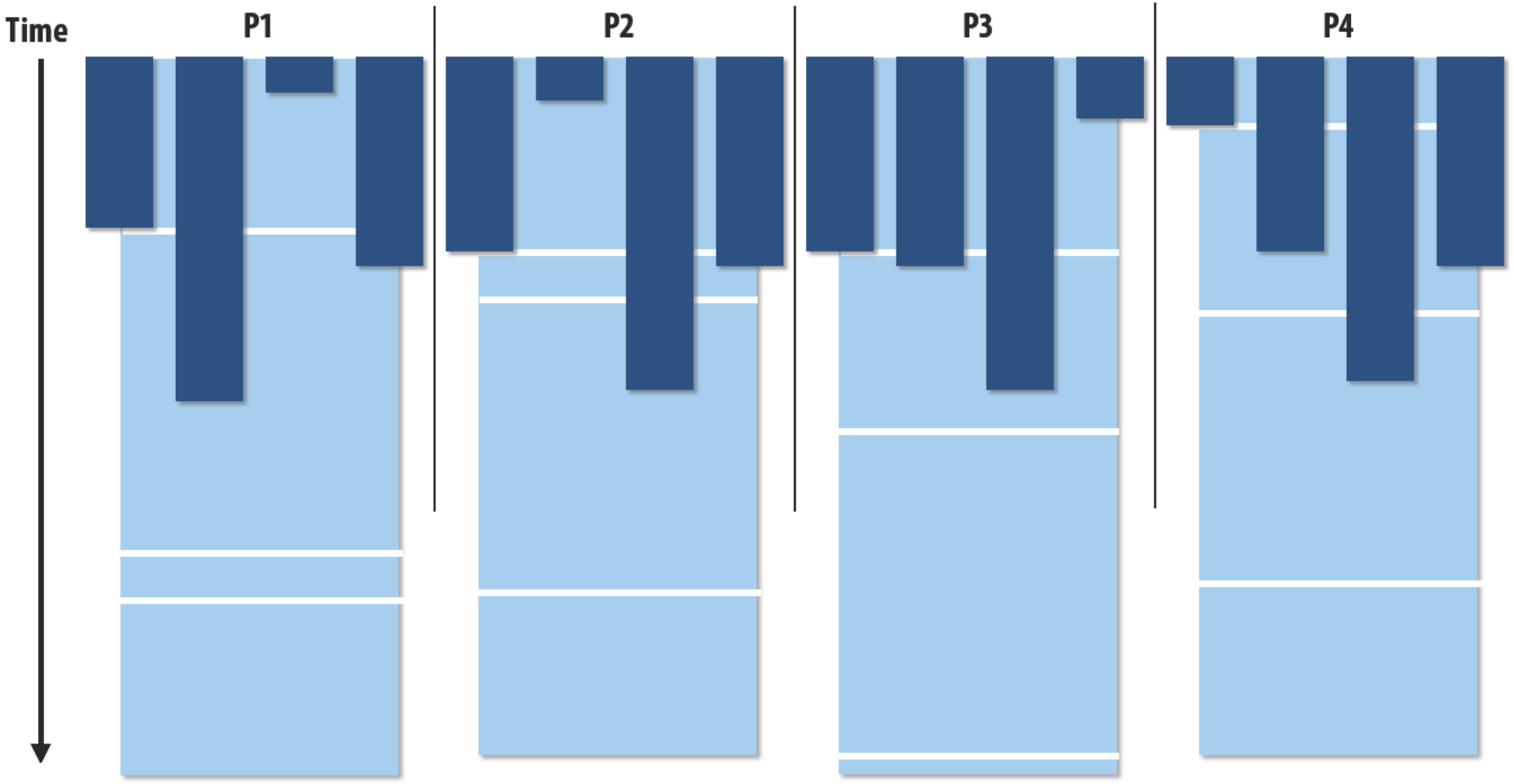
When is static assignment applicable?

- When the cost (execution time) of work and the amount of work is predictable (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?

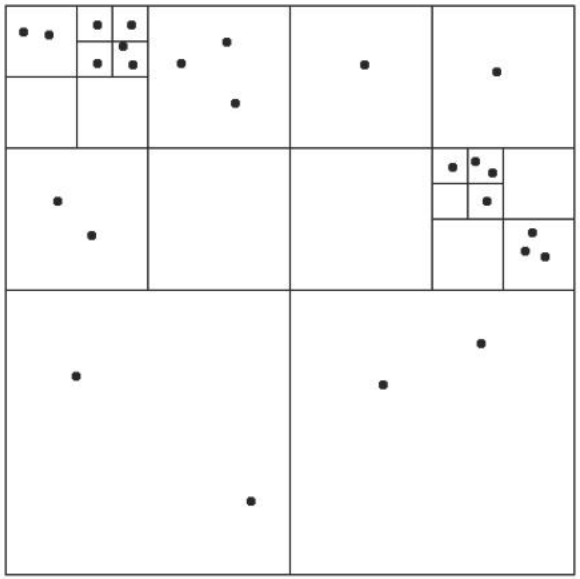
- When work is predictable, but not all jobs have same cost (see example below)
- When statistics about execution time are known (e.g., same cost on average)



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 itself 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)

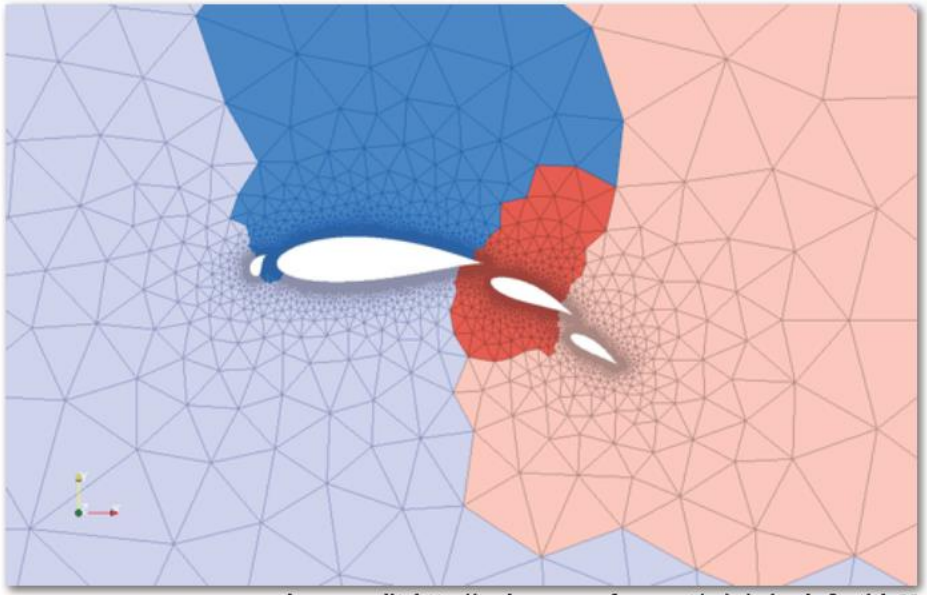


Image credit: http://typhon.sourceforge.net/spip/spip.php?article22

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);

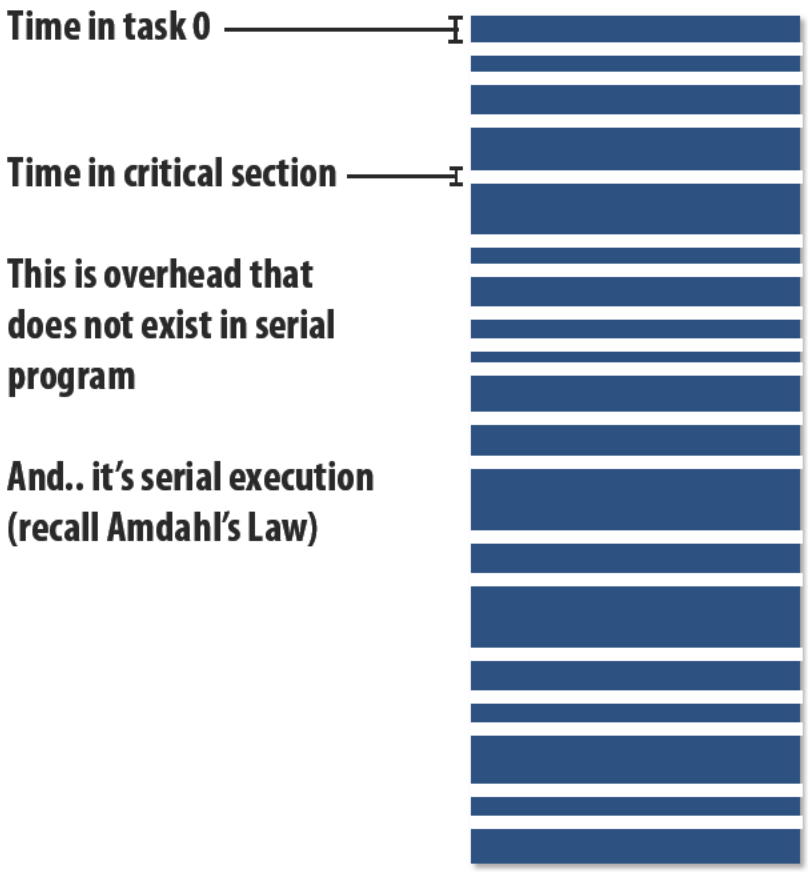
What constitutes a piece of work?

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

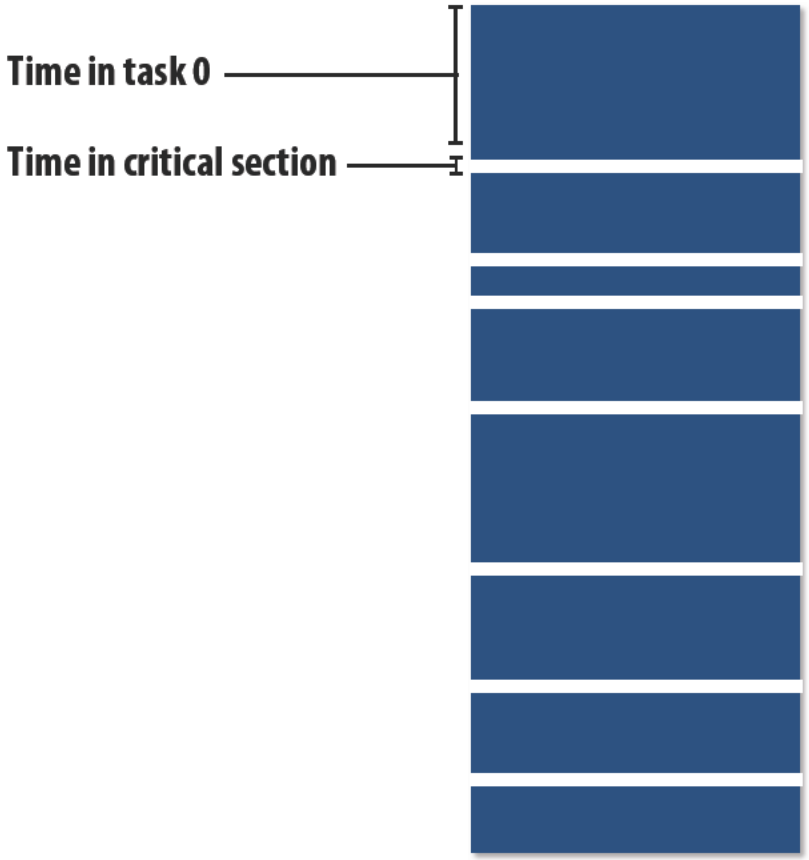
Likely good workload balance (many small tasks)
Potential for high synchronization cost
(serialization at critical section)



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[j] = test_primality(x[j]);
}
```



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

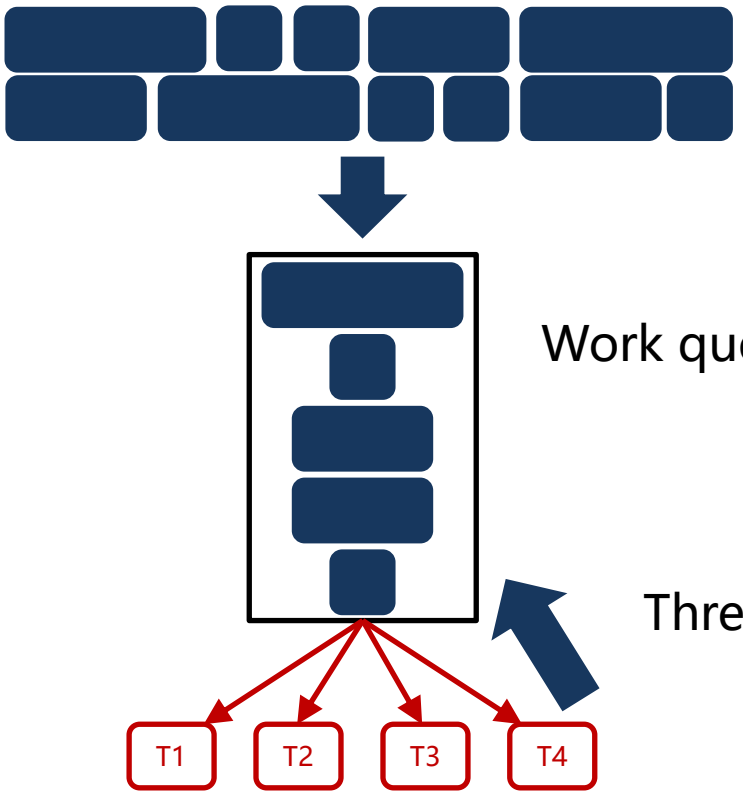
- **But want as few tasks as possible to minimize overhead of managing the assignment**
  - Motivates large granularity tasks

- **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.**

Dynamic assignment using a work queue

Generalization of loop index assignment.



Set of starting tasks

Work queue, shared between threads.

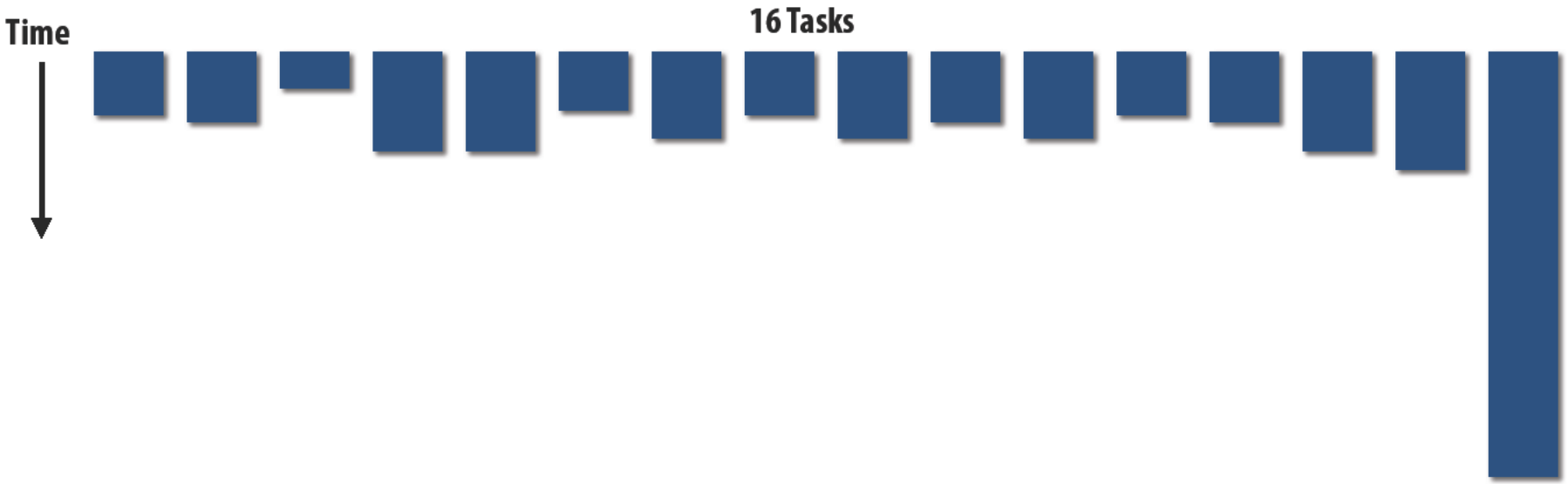
Threads can add new tasks.

Threads in a thread pool accept new tasks and execute them.

Smarter task scheduling

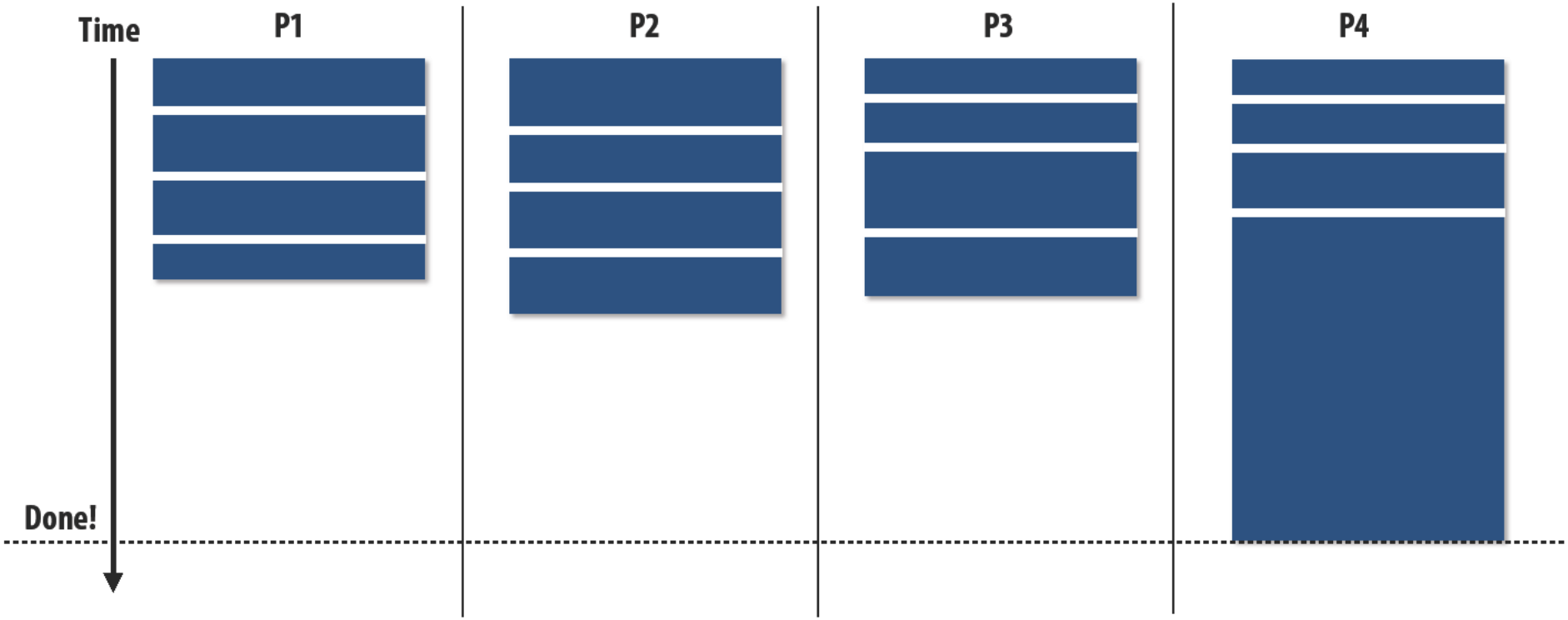
Consider dynamic scheduling via a shared work queue

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!



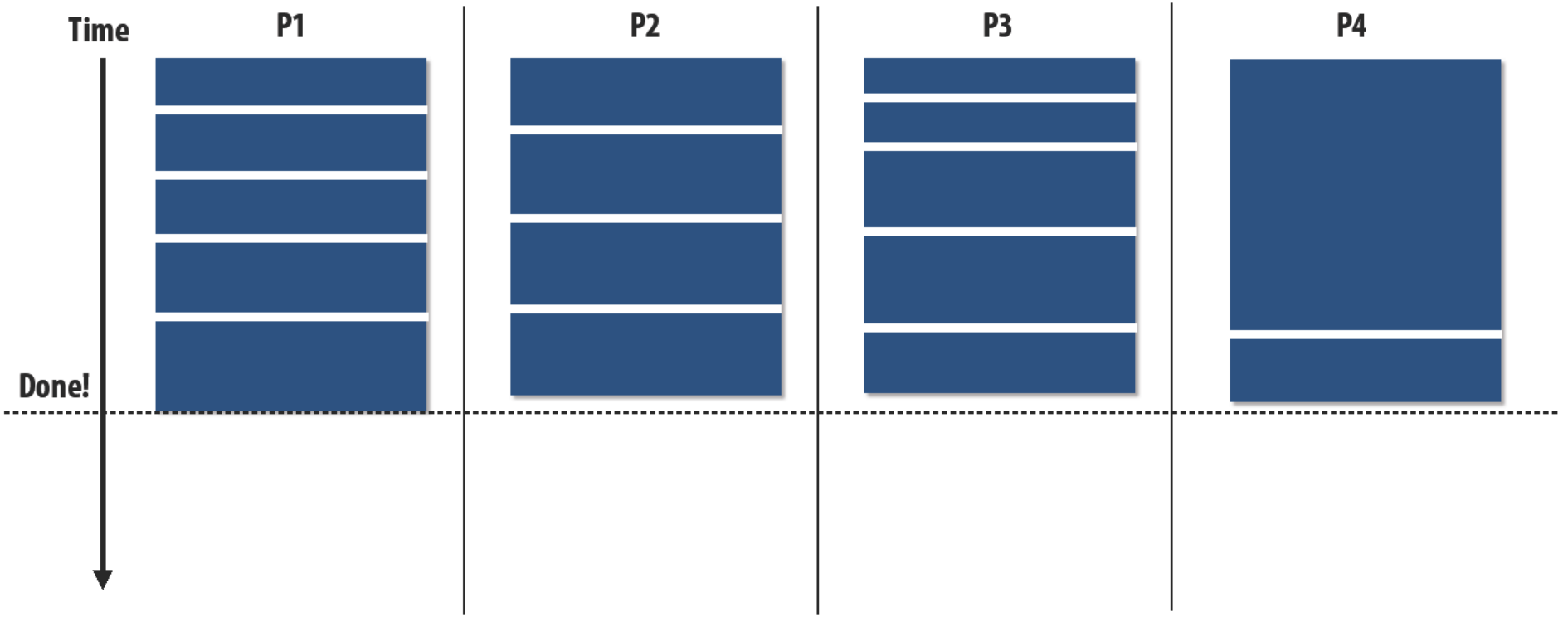
One possible solution to imbalance problem:

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)

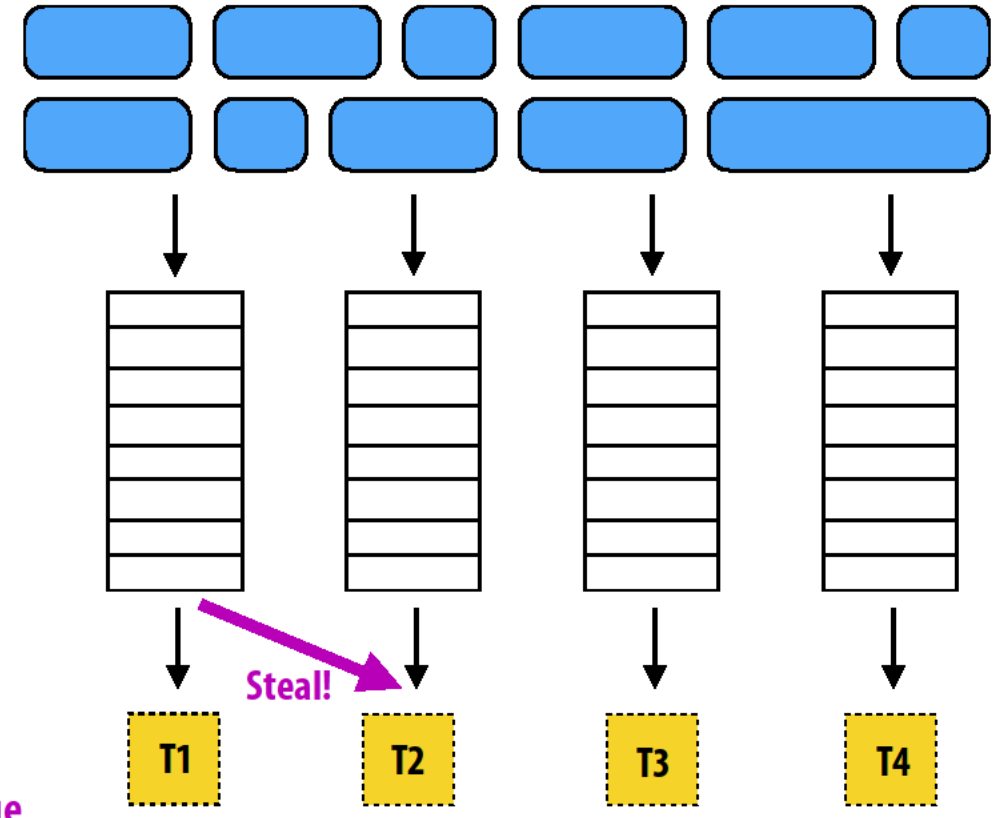
Decreasing synchronization overhead using a distributed set of queues

(avoid need for all workers to synchronize on single work queue)

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

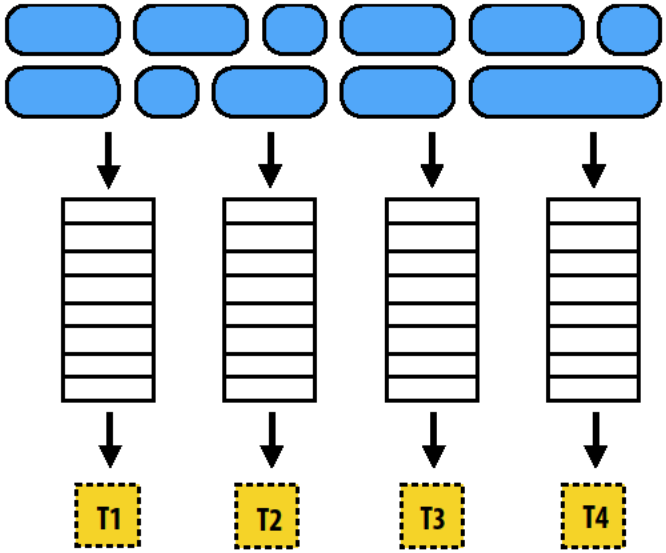
Set of work queues
(In general, one per worker thread)

Worker threads:
Pull data from OWN work queue
Push new work to OWN work queue
When local work 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 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)



Summary

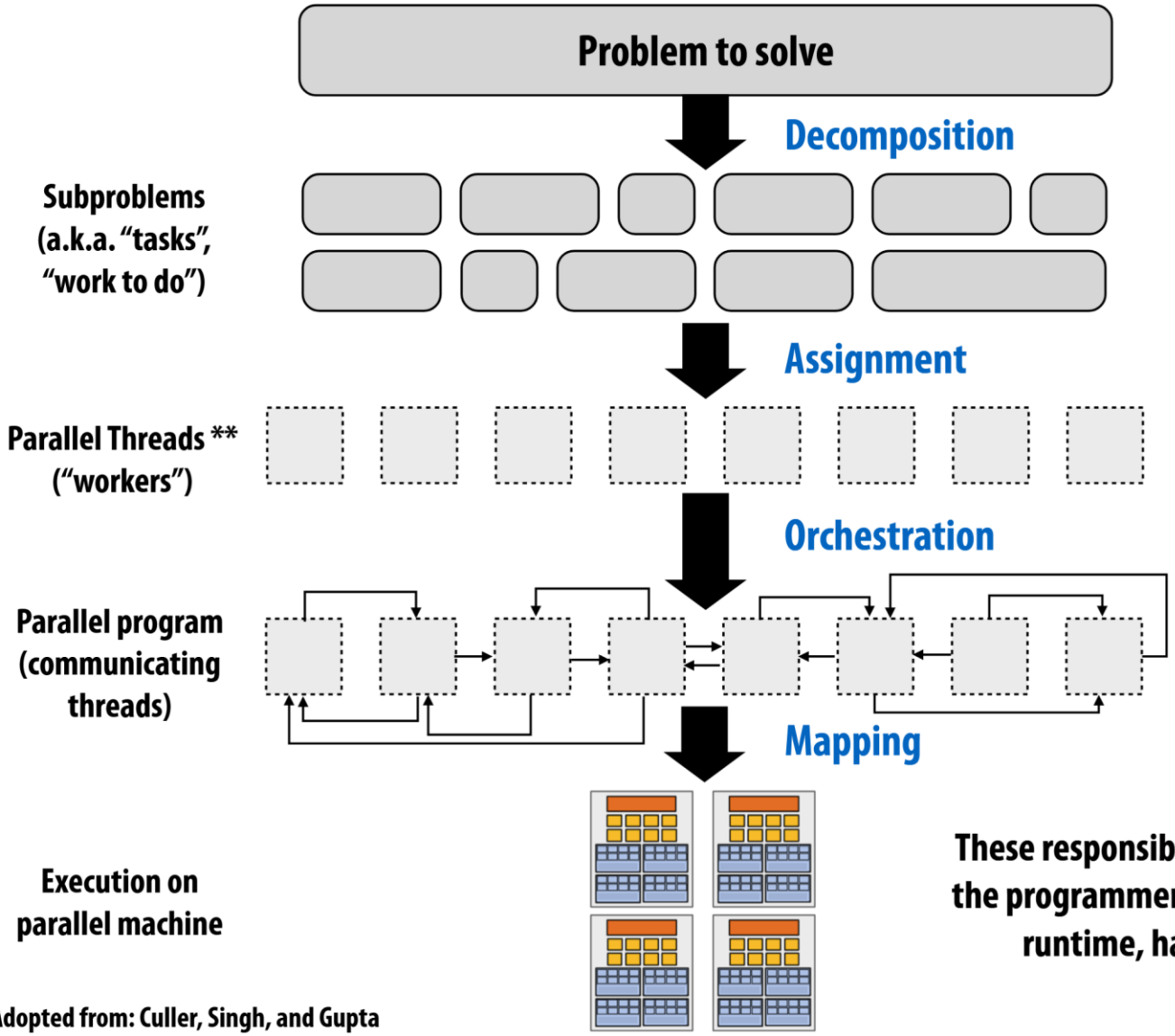
- **Challenge: achieving good workload balance**
  - Want all processors working all the time (otherwise, resources are idle!)
  - But want low-cost solution for achieving this balance
    - Minimize computational overhead (e.g., scheduling/assignment logic)
    - Minimize synchronization costs

- **Static assignment vs. dynamic assignment**
  - Really, it is 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**

Orchestration

Synchronization, ordering, data structures.



Adopted from: Culler, Singh, and Gupta

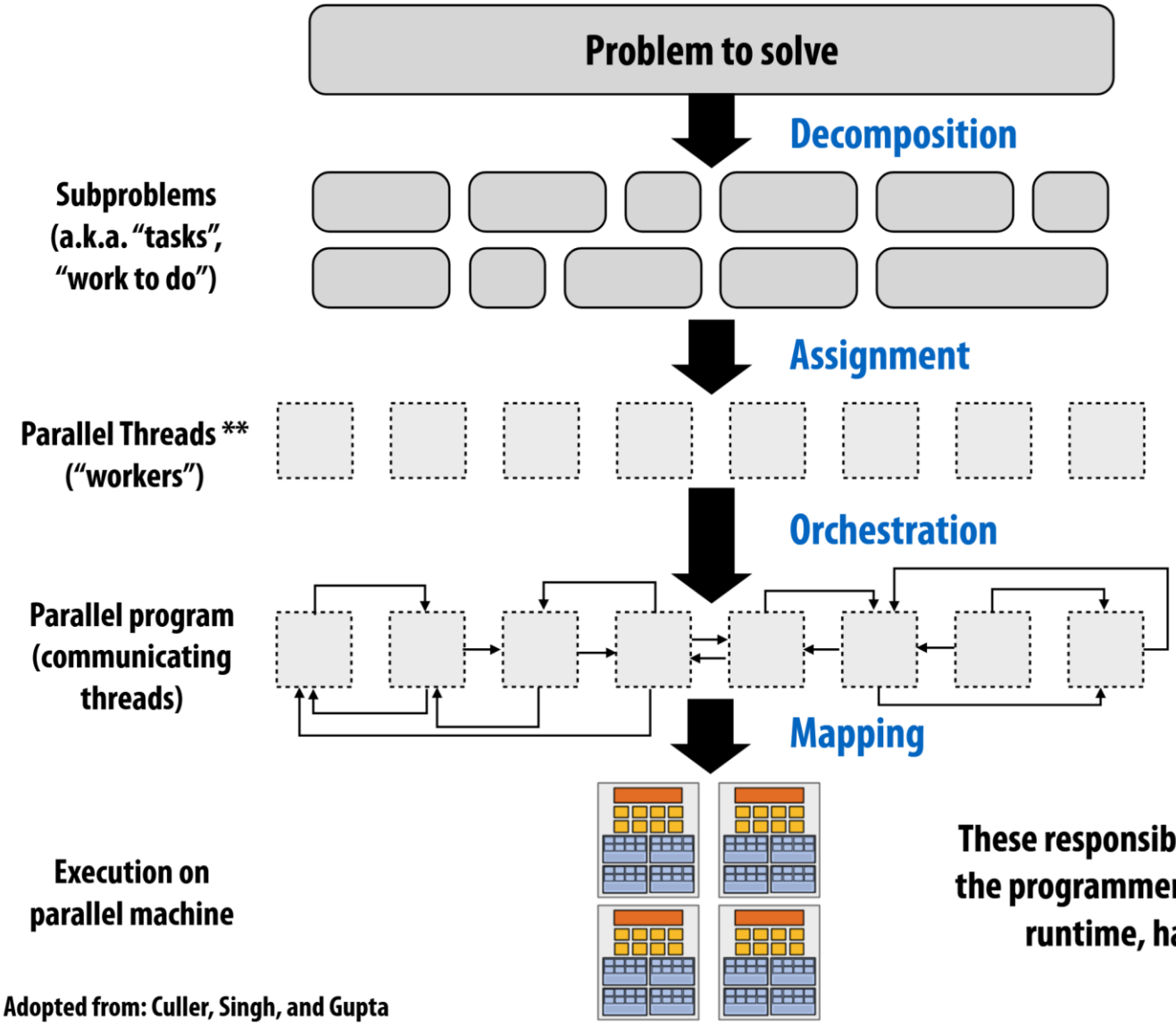
Orchestration

Synchronization, ordering, data structures.

- Add communication between tasks to ensure:
  - Safe access to shared resources.
  - Correct dependencies (e.g. one task needs data from another task).
- Strive to minimize communication and waiting time.
- Dependent on our hardware.
- Available tools and approaches already covered by previous lectures.

Mapping to hardware

Executing an abstract program on real hardware.



Adopted from: Culler, Singh, and Gupta

Mapping to hardware

Mapping "threads" ("workers") to hardware execution units

Example 1: mapping by the operating system

- e.g., map pthread to HW execution context on a CPU core

Example 2: mapping by the compiler

- Map ISPC program instances to vector instruction lanes

Example 3: mapping by the hardware

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

Some interesting mapping decisions:

- Place related threads (cooperating threads) on the same processor (maximize locality, data sharing, minimize costs of comm/sync)
- Place unrelated threads on the same processor (one might be bandwidth limited and another might be compute limited) to use machine more efficiently

Slightly less trivial than the ones so far…

PRACTICAL EXAMPLES

Divide and conquer

Quicksort

- Basic, but very common sorting algorithm.
- In each level of recursion, select a pivot, reorder numbers so that the pivot is in the correct position, then recurse into each half.

```
template<typename It>
void quicksort(It begin, It end) {
    if (end - begin <= 1) {
        return;
    }

    // use first value as pivot
    auto pivot_it = partition(begin, end, begin);

    quicksort(begin, pivot_it);
    quicksort(pivot_it + 1, end);
}
```

Divide and conquer

Quicksort

```
template<typename It>
void quicksort(It begin, It end, size_t task_count) {
    if (end - begin <= 1) return;

    if (task_count == 1) {
        quicksort_seq(begin, end);
        return;
    }

    // use first value as pivot
    auto pivot_it = partition(begin, end, begin);

    #pragma omp task
    quicksort(begin, pivot_it, task_count / 2);

    // run this branch in the current task
    quicksort(pivot_it + 1, end, task_count / 2);
}
```

Cutoff -> switch to serial version.

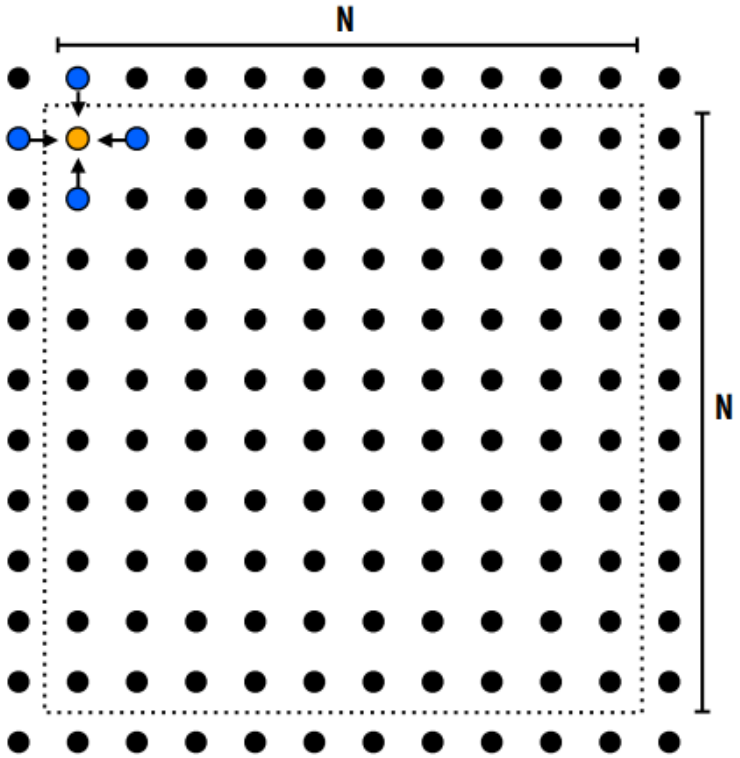
Create a task for one branch.

Avoid task overhead.

Heat diffusion simulation

Let's make it a bit harder.

- Quicksort parallelization was easy, as we had no dependencies.
- What if we have dependencies?



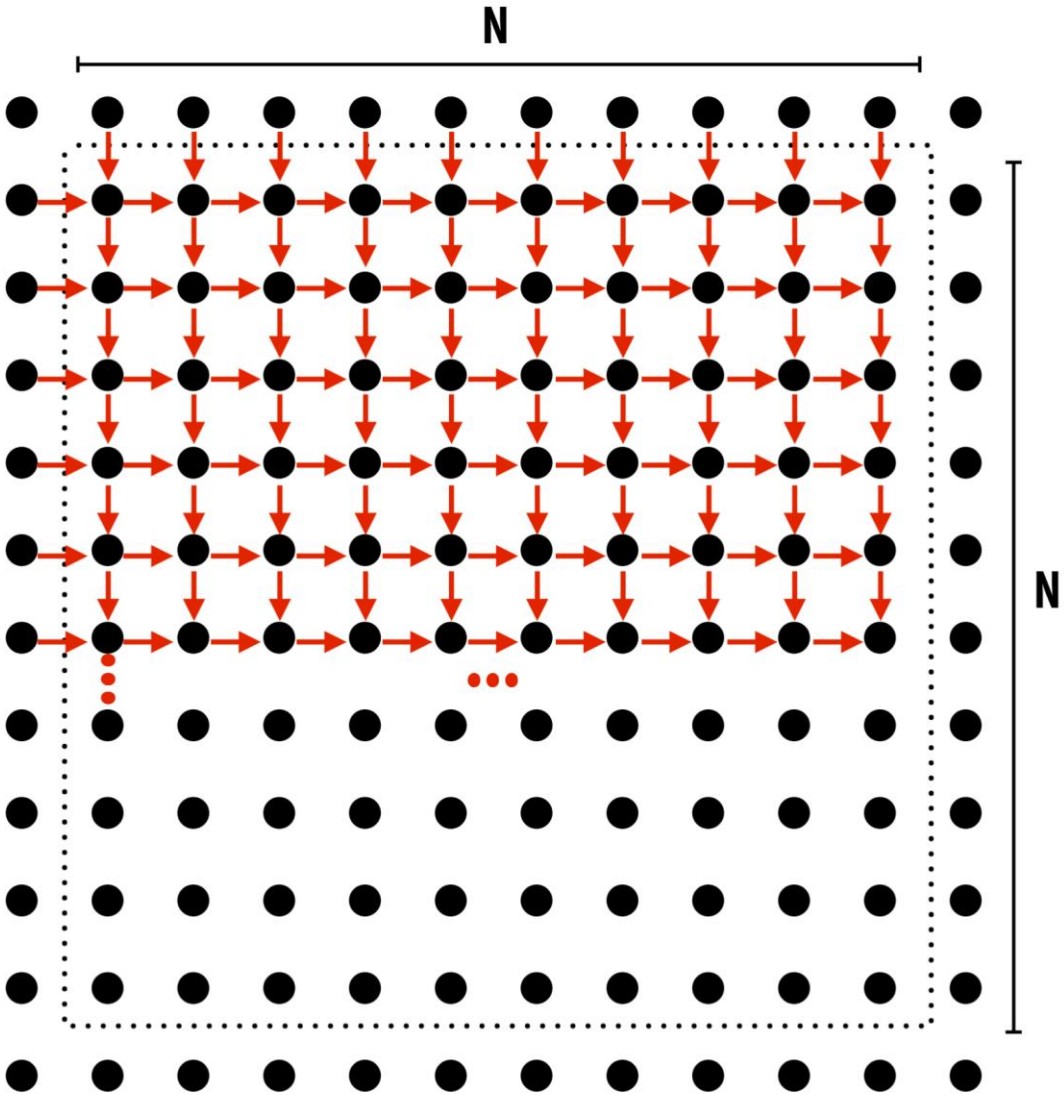
Problem: We have a 2D space containing heat sources and sinks with fixed temperature. Find the stable temperature of each point.

- Simplification: Discretize to 2D grid.
- $A[i,j] = 0.2 * (A[i-1,j] + A[i,j-1] + A[i,j] + A[i+1,j] + A[i,j+1])$
- Iteratively compute heat transfer for each point until the system stabilizes, going row by row.
- Note that some of the surrounding points have already been updated, some not.

Adapted from CMU 15-418/618, at https://www.cs.cmu.edu/~15418/.

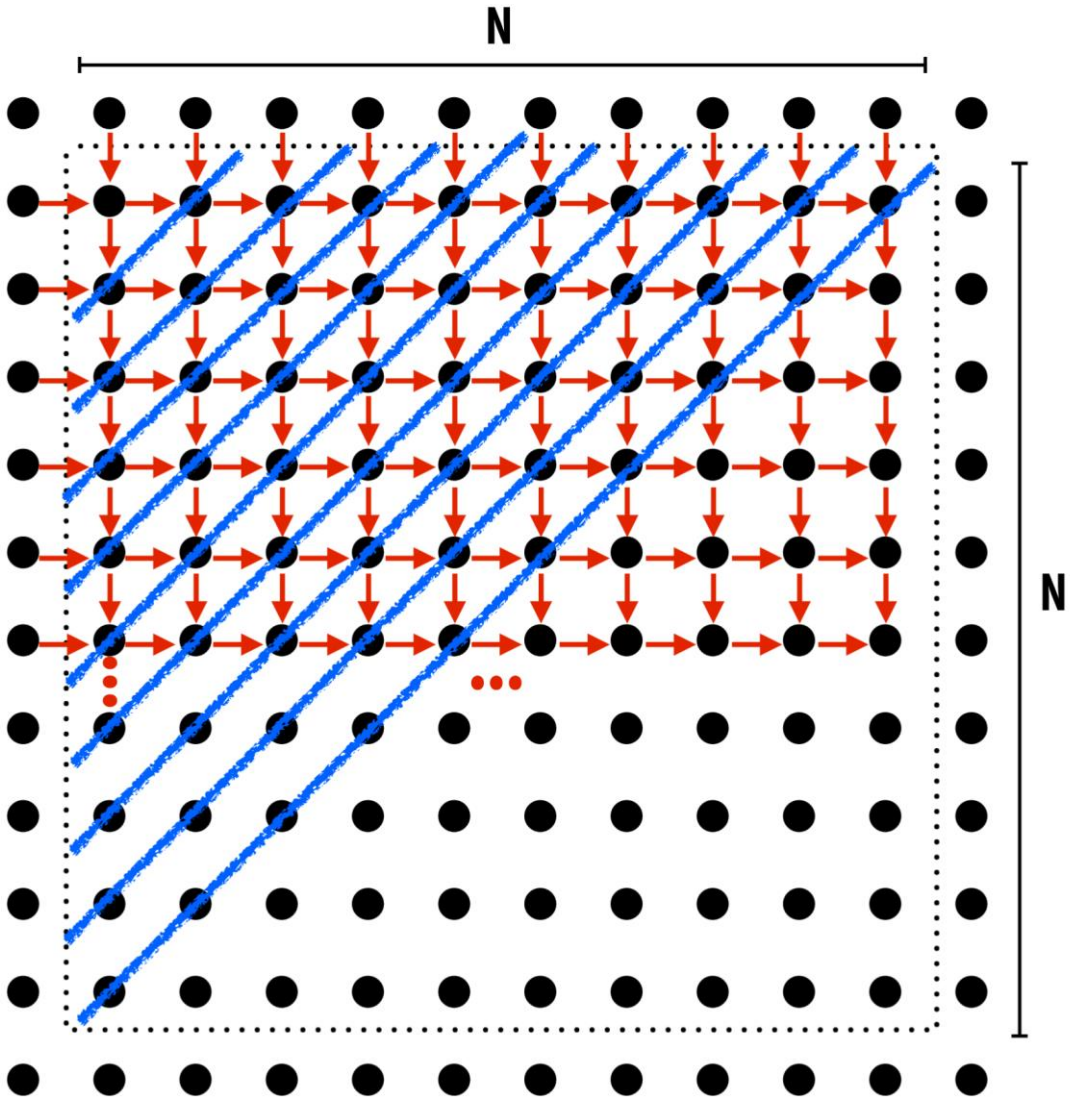
Heat diffusion simulation

What are the dependencies in one iteration?



Heat diffusion simulation

What can we execute in parallel?

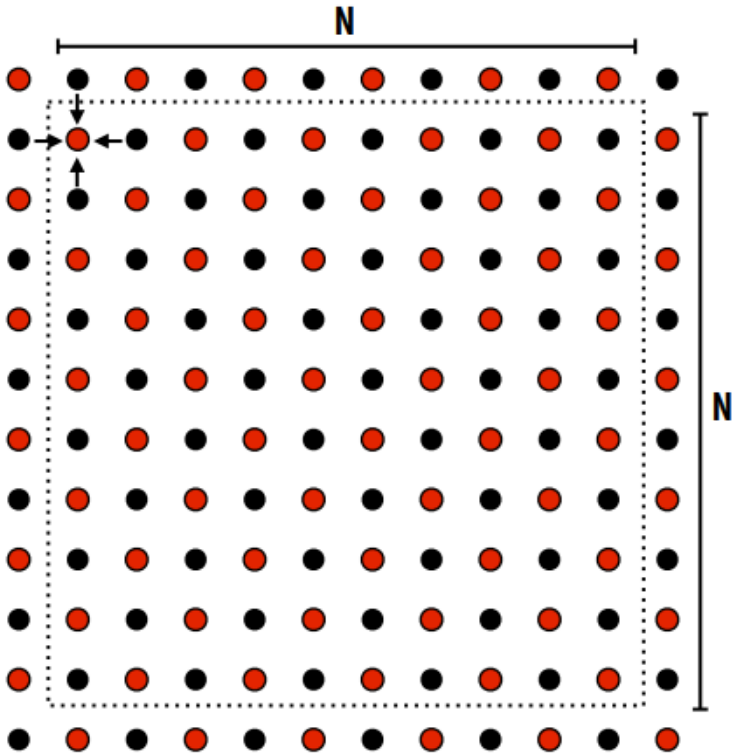


Heat diffusion simulation

Reformulate the problem.

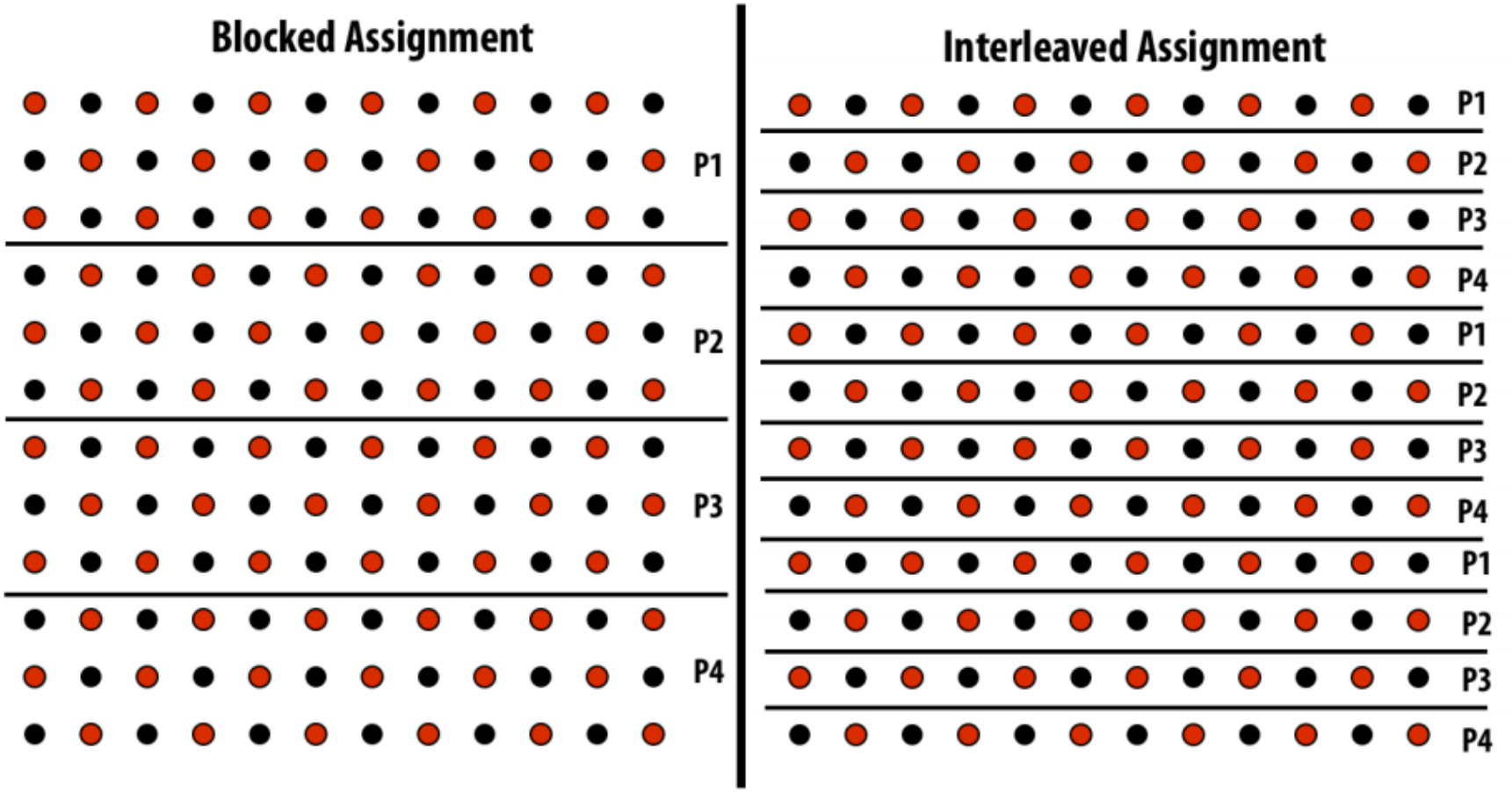
- Parallelization on diagonals needs too much synchronization.
- Idea: Could we change the algorithm to still reach the same result, but make it easier to parallelize?

- We could change iteration order to first update odd nodes, then the even ones.
- Is that correct?
  - = Does it give the same steady state?



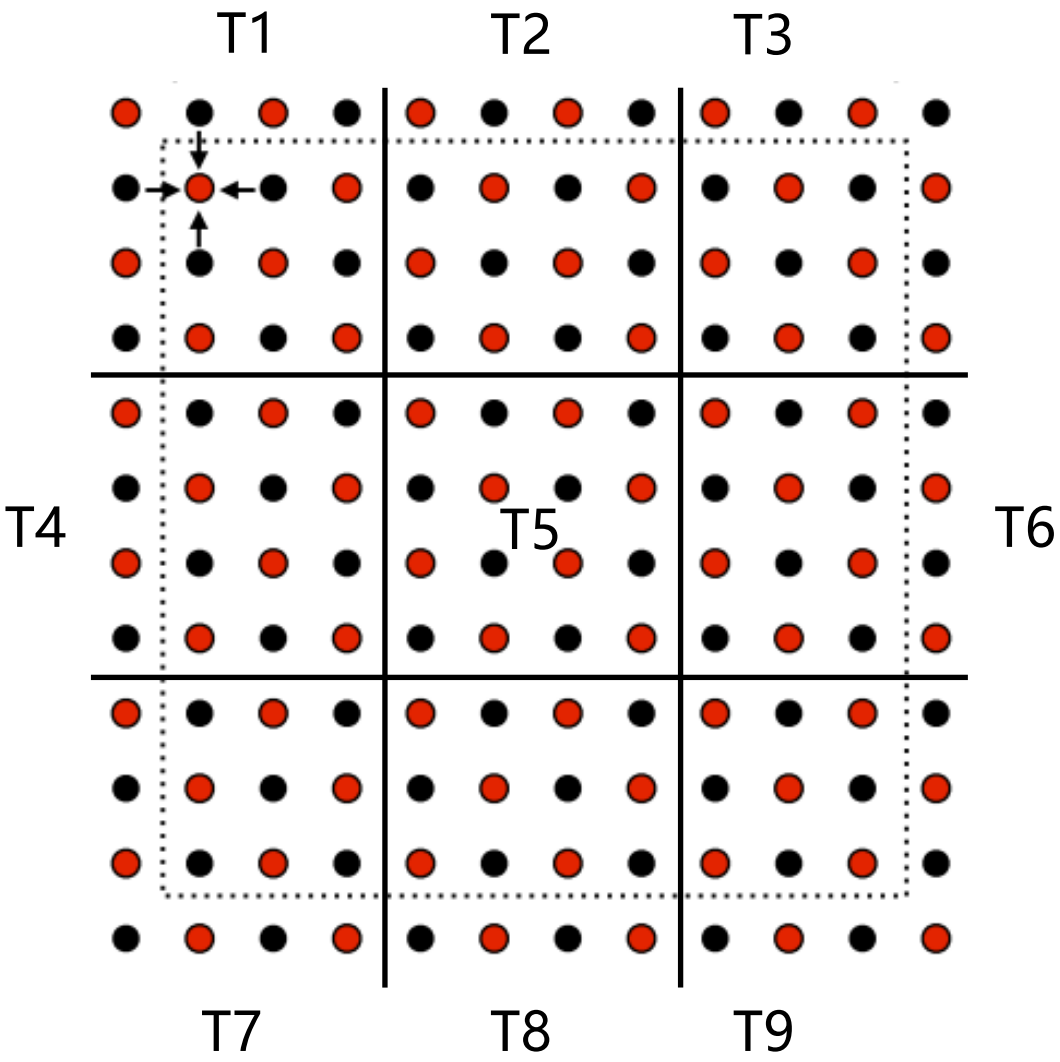
Heat diffusion simulation

How to assign areas?



Heat diffusion simulation

2D assignment



Sieve of Eratosthenes

More dependencies.

```
std::vector<uint64_t> find_primes_under(uint64_t max_n) {
    auto found_primes = std::vector<uint64_t>();
    auto sieve = std::vector<bool>(max_n, true);

    for (size_t n = 2; n <= std::sqrt(max_n); n++) {
        if (sieve[n]) {
            found_primes.push_back(n);
            for (size_t mult = n; mult < max_n; mult += n) {
                sieve[mult] = false;
            }
        }
    }

    for (size_t n = std::sqrt(max_n); n < max_n; n++) {
        if (sieve[n]) {
            found_primes.push_back(n);
        }
    }
    return found_primes;
}
```

Sieve of Eratosthenes

How to parallelize?

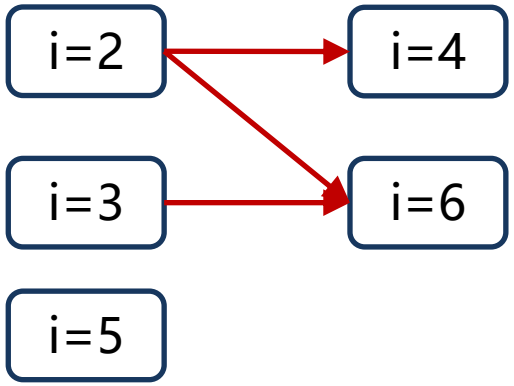
- We could parallelize the outer loop.
  - LOT of duplicated work, since we don't yet know if the number is prime when pruning its multiples.
  - Likely results in frequent false sharing.
  - But the result is correct.
- We could parallelize the inner loop.
  - Quite obviously safe, with minimal sharing.
  - Somewhat high overhead from barriers.

- Can we do something smarter?
- What are the actual dependencies?

Sieve of Eratosthenes

What are the dependencies?

- We can start pruning multiples of X when we know that X is prime.
- We know that X is prime when we already pruned all its possible divisors.
- Huh…



Sieve of Eratosthenes

What are the dependencies?

- Do we know anything about the structure of primes?
- Yes: If we're pruning multiples of `n`, we know that everything unpruned under `2*n` is a prime.

- We can safely parallelize batches between powers of 2.
- 2 -> 4 -> 8 -> 16 -> 32 -> 64 -> 128 -> ...

- How to parallelize?
  1. Split primes between threads.
     - Each thread picks a prime and prunes its multiples.
  2. Split subranges of the sieve between threads.
     - Each thread prunes a range of numbers for all primes.

Sieve of Eratosthenes

What are the dependencies?

- Can we make the batches even larger?
- Yes: If we reached `sqrt(n)` and did not prune `n`, we know it's a prime!

- We can jump by exponentiation, not just multiplication!
- 2 -> 4 -> 16 -> 256 -> 65535 -> ...