

( how to be l33t )

Lecture 5:

# ~~Performance Optimization~~ Part 1: Work Distribution and Scheduling

---

Parallel Computing  
Stanford CS149, Fall 2023

# Programming for high performance

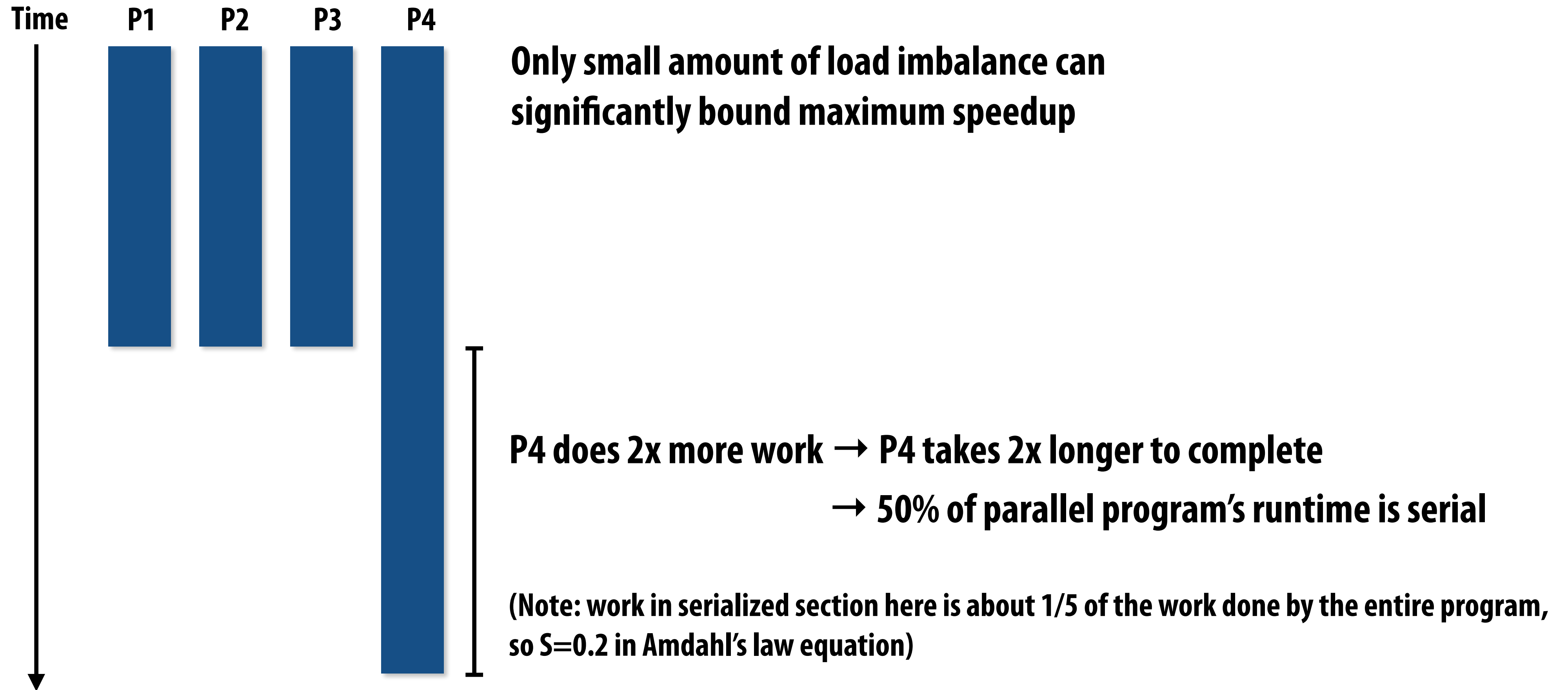
- **Optimizing the performance of parallel programs is an iterative process of refining choices for decomposition, assignment, and orchestration...**
- **Key goals (that are at odds with each other)**
  - **Balance workload onto available execution resources**
  - **Reduce communication (to avoid stalls)**
  - **Reduce extra work (overhead) performed to increase parallelism, manage assignment, reduce communication, etc.**
- **We are going to talk about a rich space of techniques**

# Programming for high performance

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

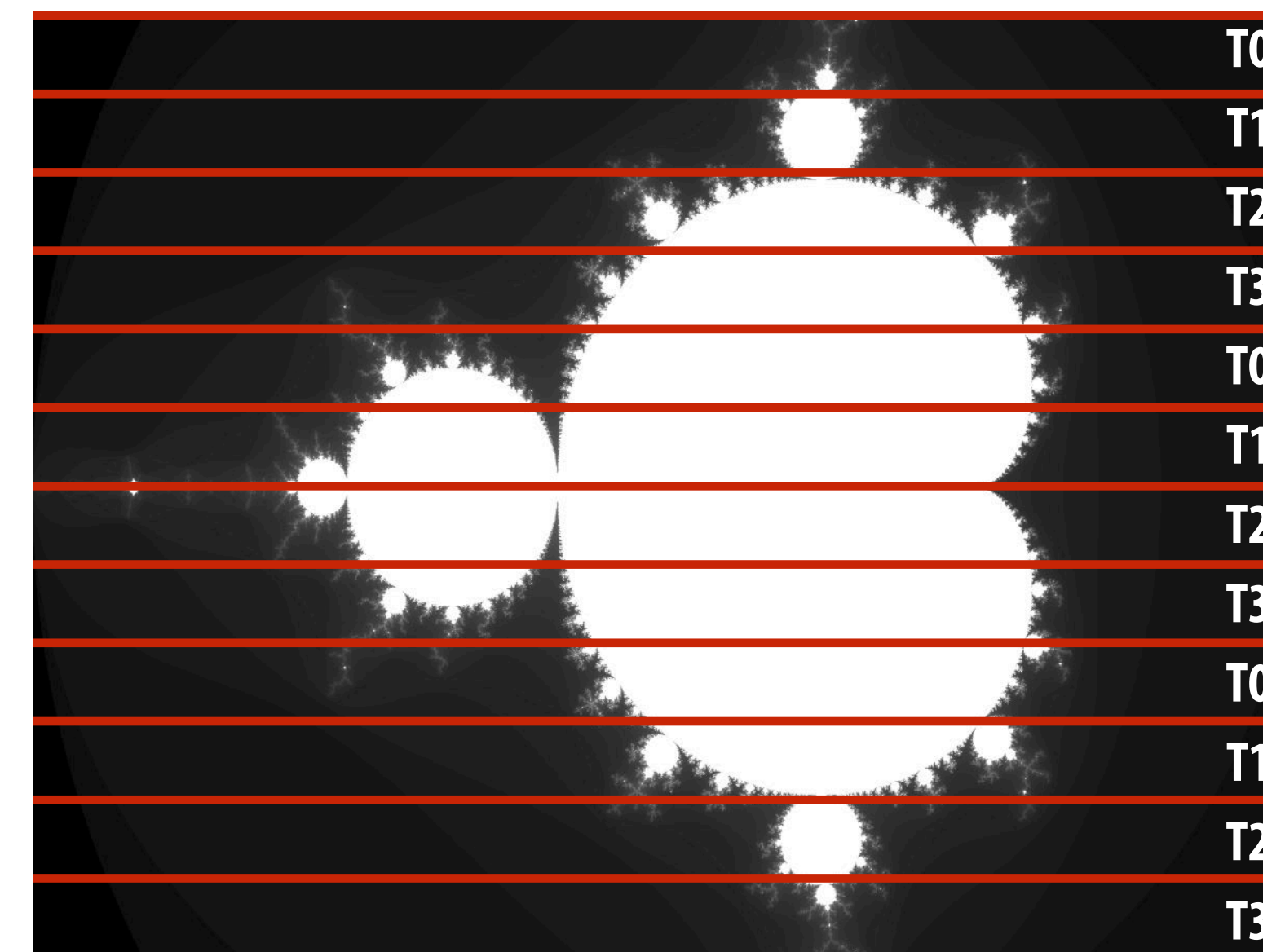
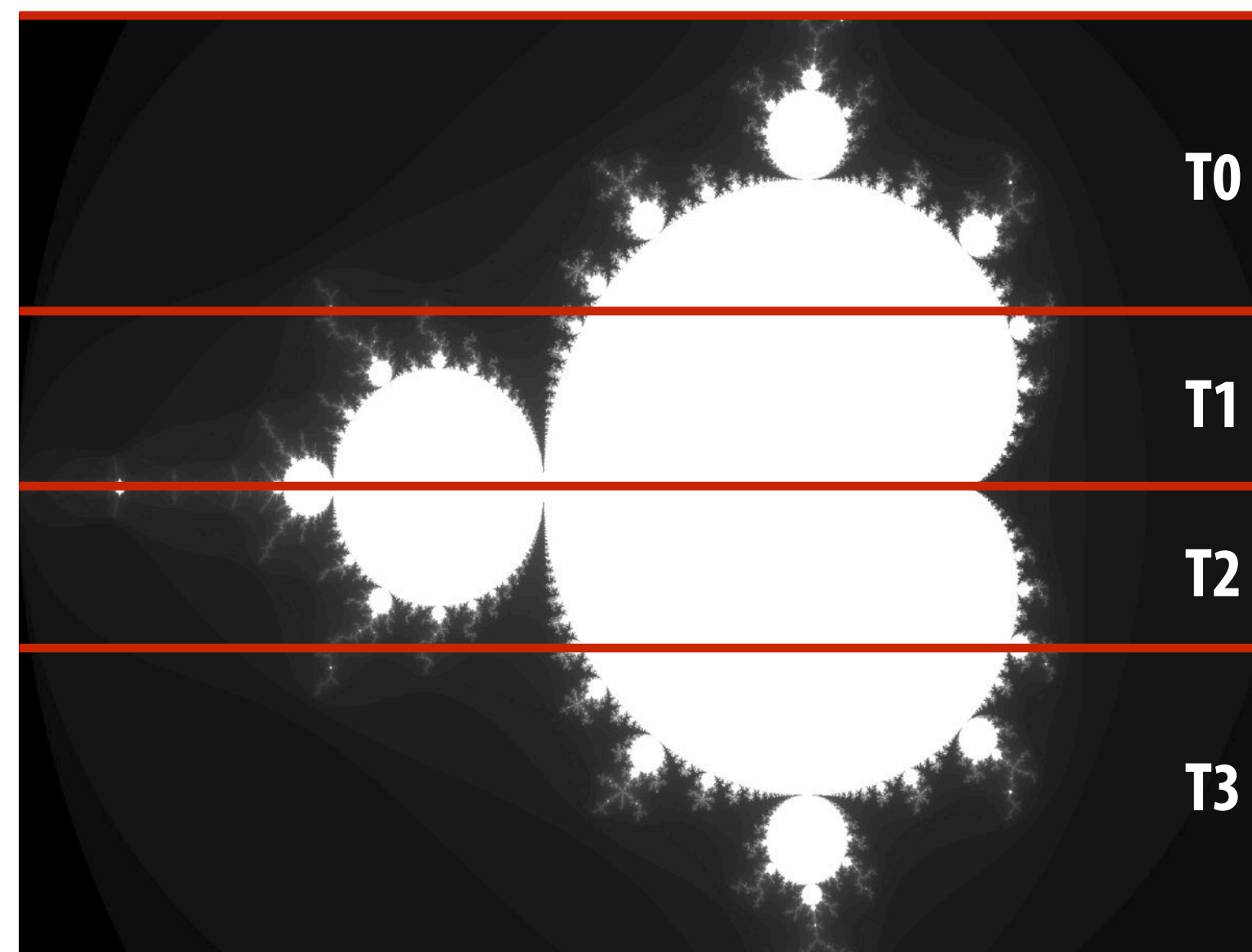
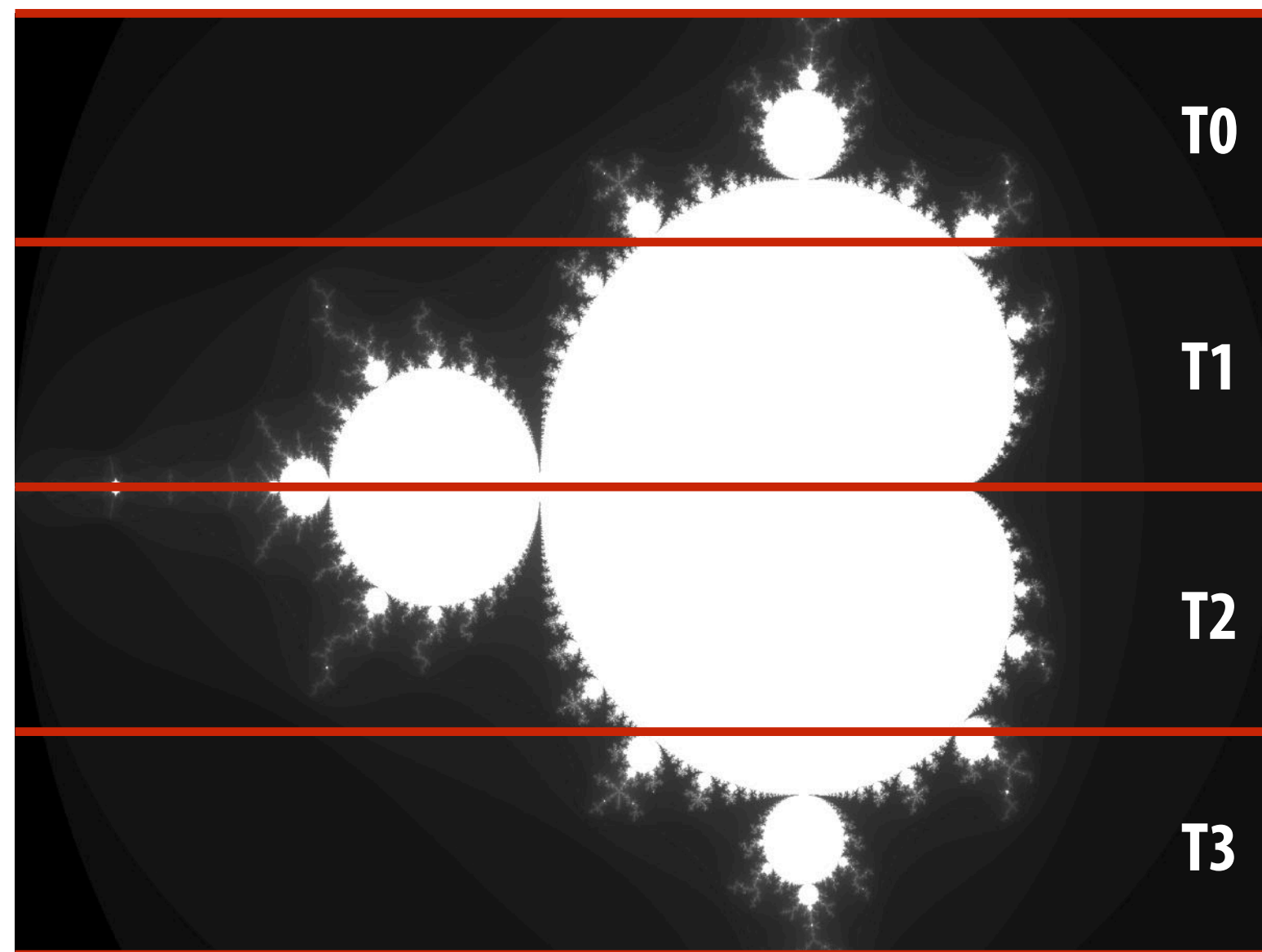
# Balancing the workload

**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)**



# Static assignment

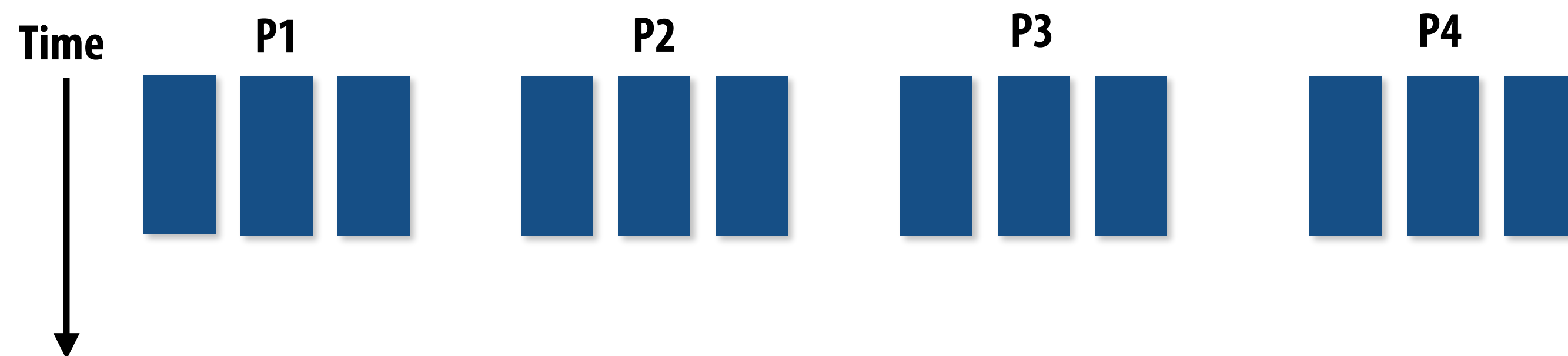
- Assignment of work to threads does not depend on dynamic behavior
  - Assignment not necessarily set at compile-time (we call it static is the assignment is determined when the amount of work and number of workers is known: assignment may depend on runtime parameters such as input data size, number of threads, etc.)
- Recall Programming Assignment 1, program 1: assign equal number of grid cells to each thread
  - Students explored different static assignments of work to workers



- Good aspects of static assignment: simple, essentially zero runtime overhead to perform assignment (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 predictable, allowing the programmer to work out a good assignment in advance
- Simplest example: it is known up front that all work has the same cost



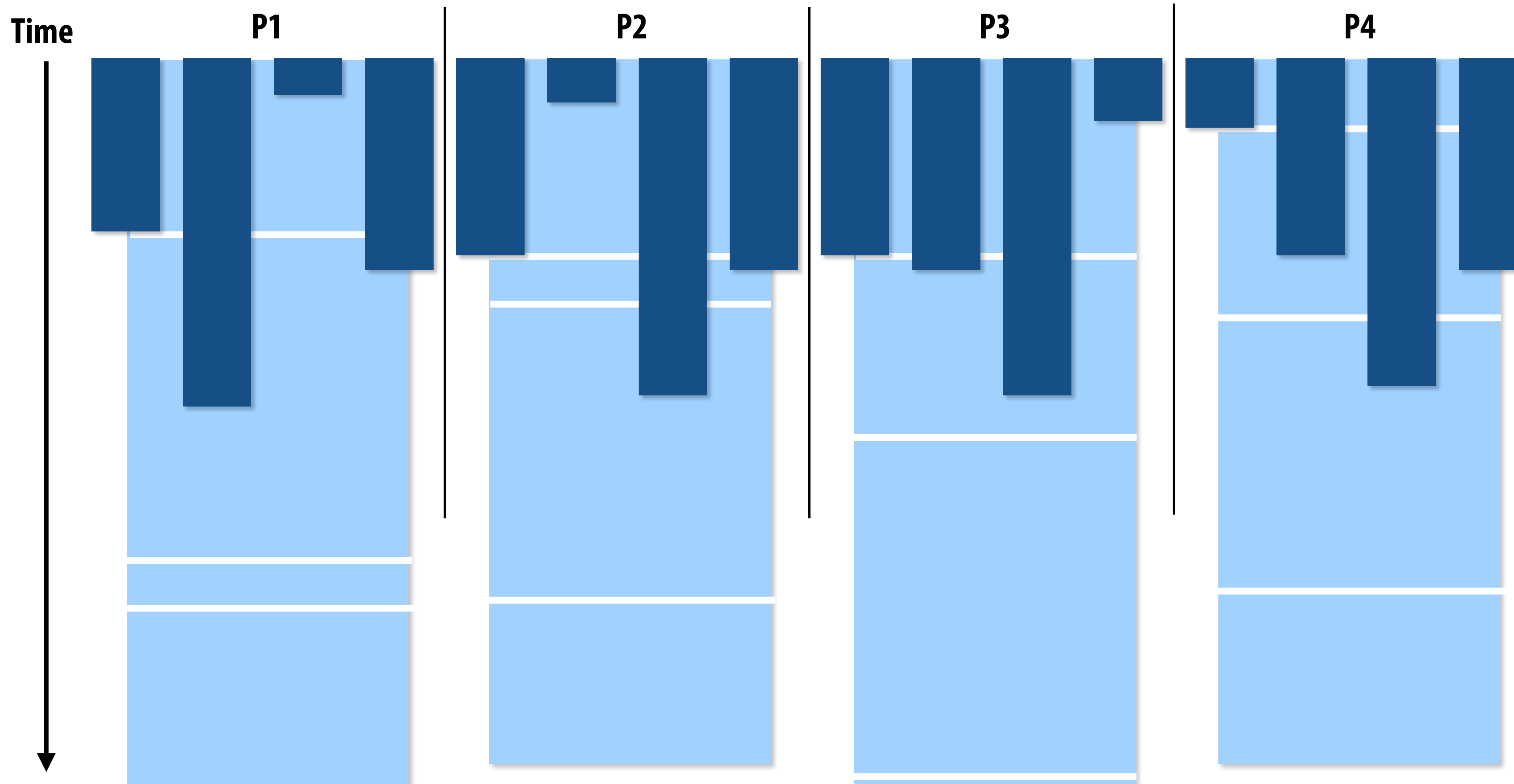
**In the example above:**

**There are 12 tasks, and it is known that each have the same cost.**

**Static assignment: statically assign three tasks to each of the four processors.**

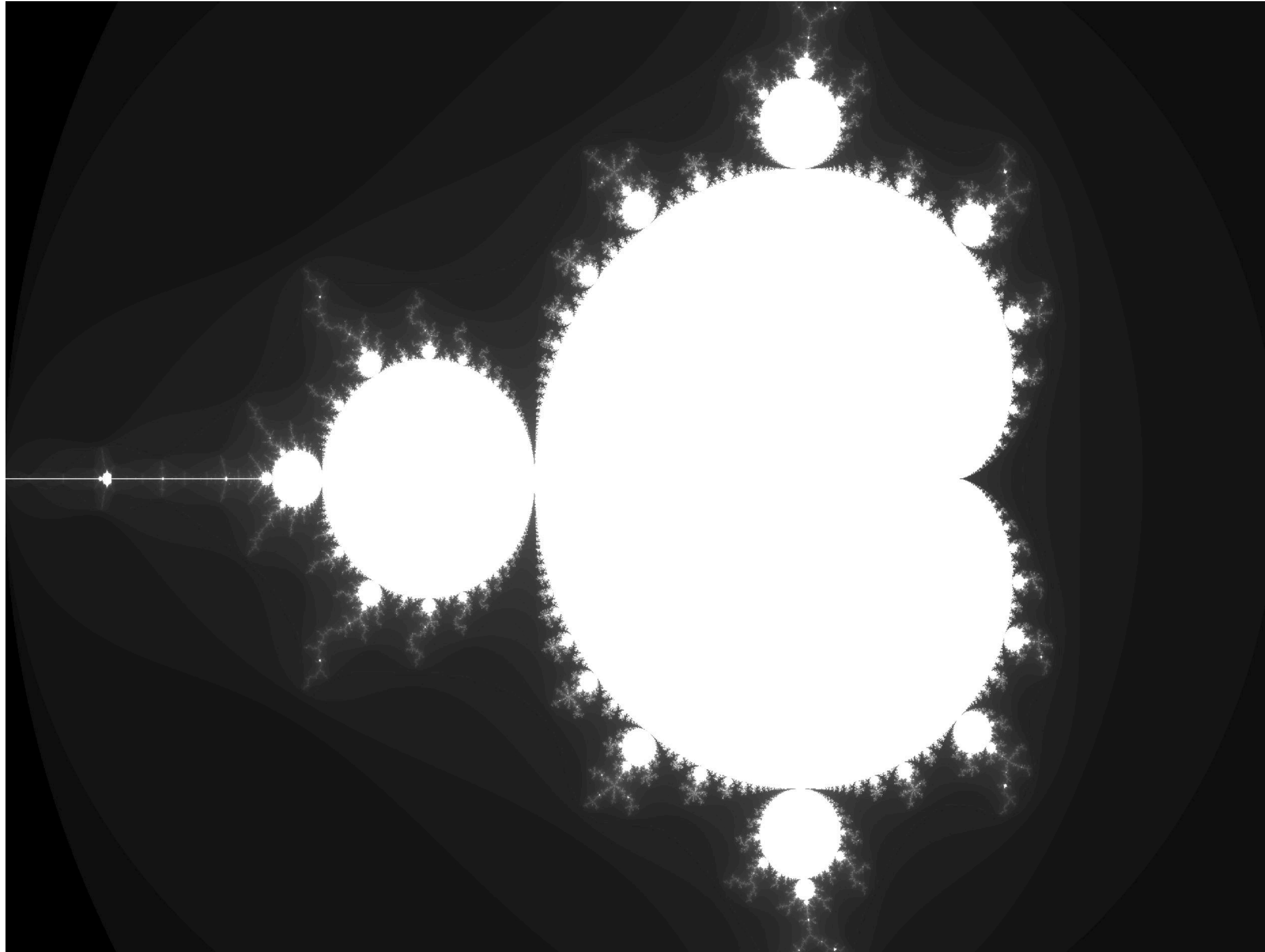
# When is static assignment applicable?

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



**Jobs have unequal, but known cost: assign equal number of tasks to processors to ensure good load balance (on average)**

# Example from programming assignment 1 (prog 1)

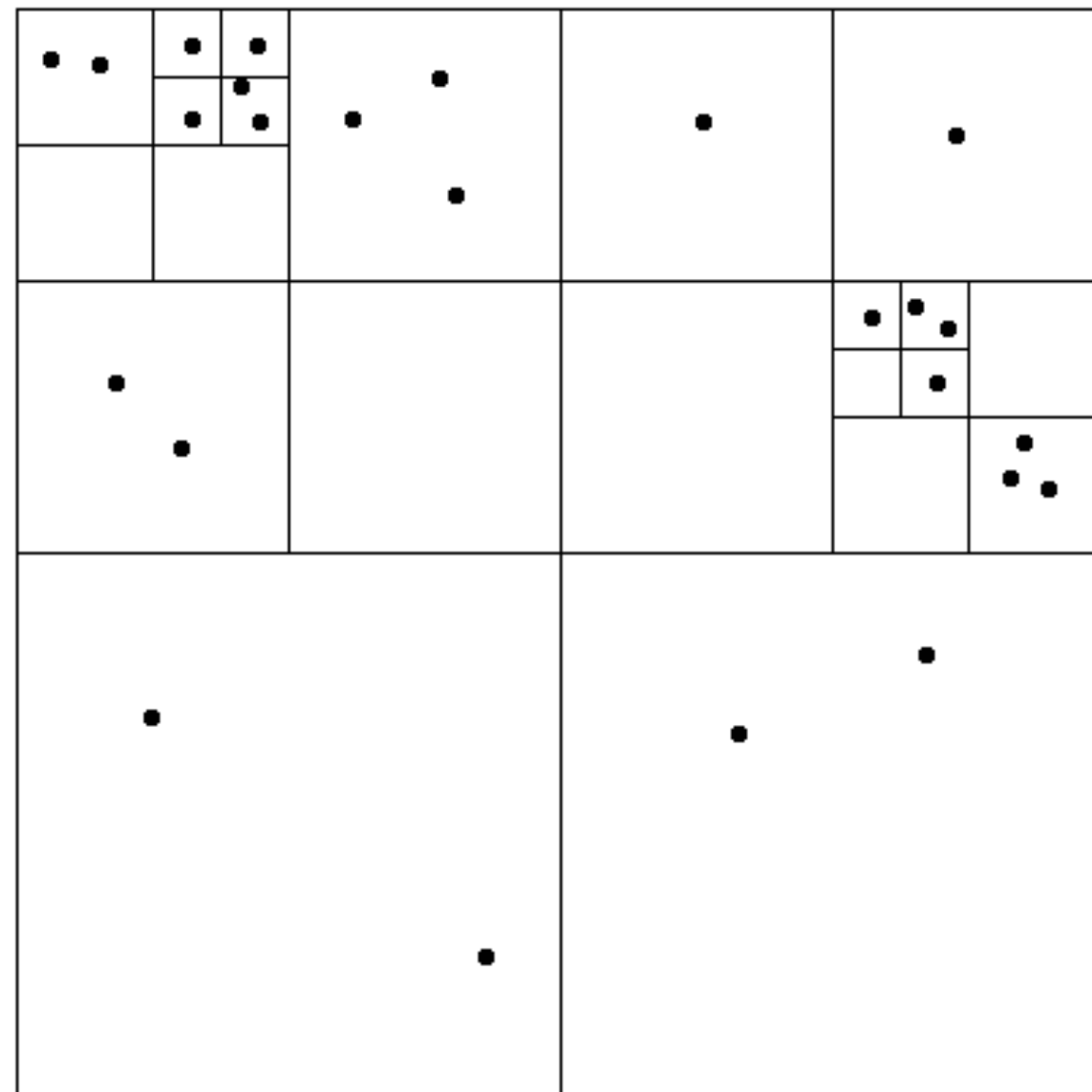


- Why was a static row-interleaved assignment a good solution to this problem?



# “Semi-static” assignment

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



## Particle simulation:

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

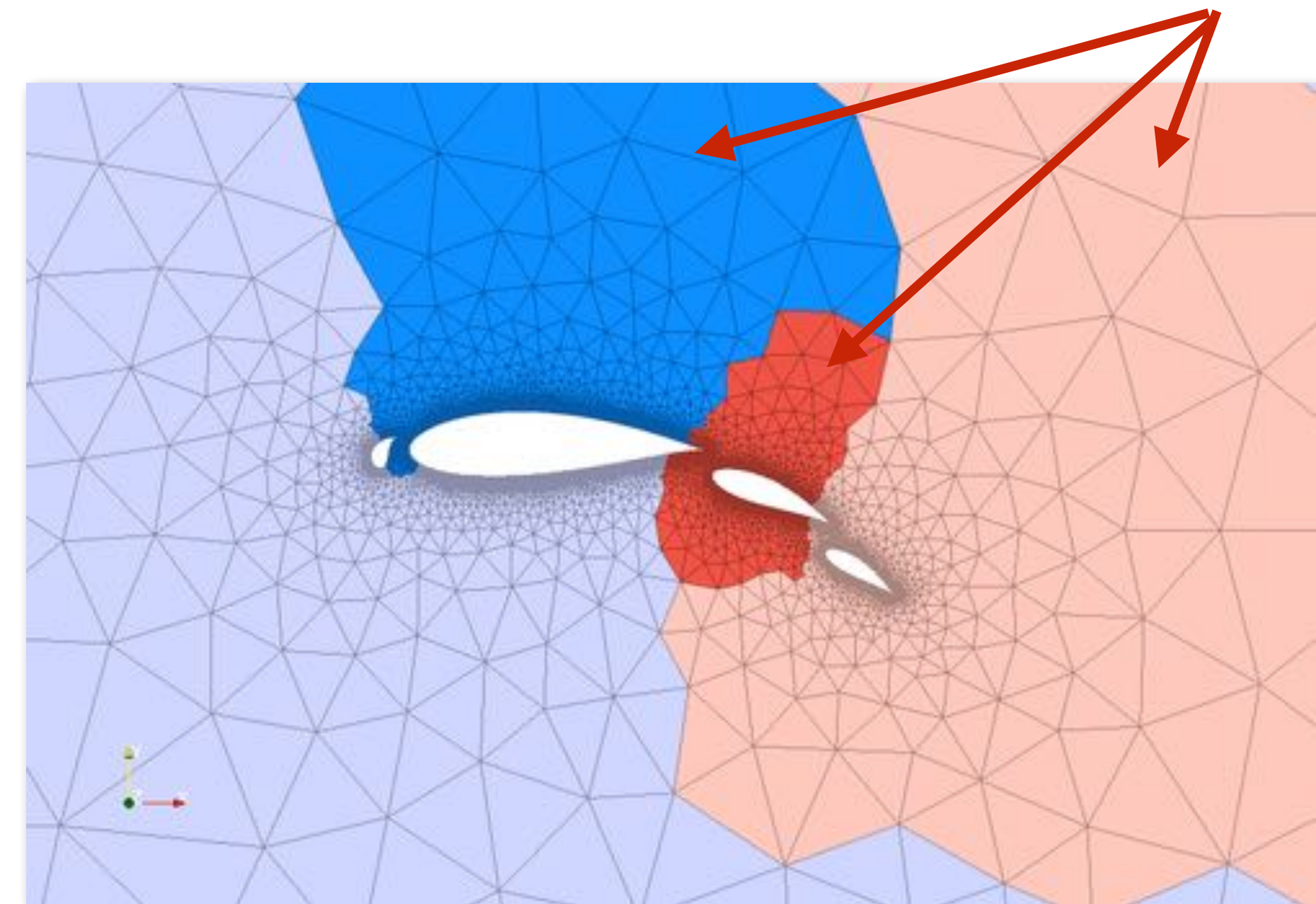


Image credit: <http://typhon.sourceforge.net/spip/spip.php?article22>

## 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 unknown or unpredictable.)

Sequential program  
(independent loop iterations)

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

// assume elements of x initialized 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)


```
int N = 1024;

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

// assume elements of x are initialized here

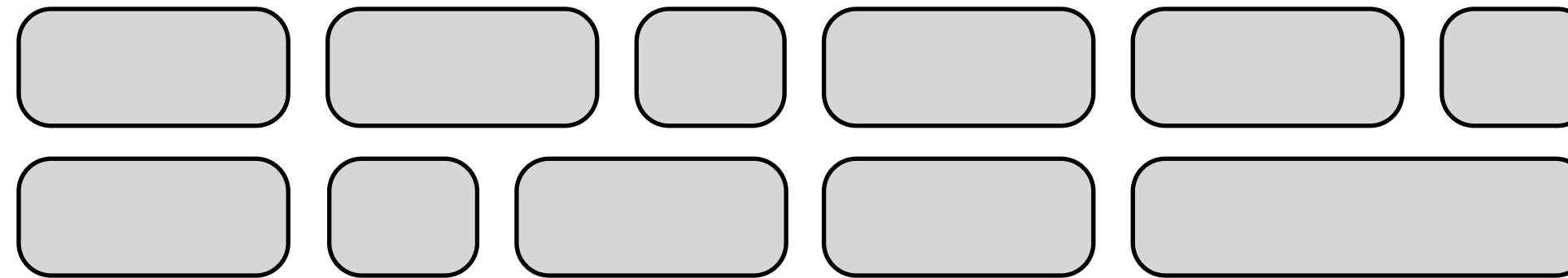
LOCK counter_lock;
int counter = 0;    // shared variable

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

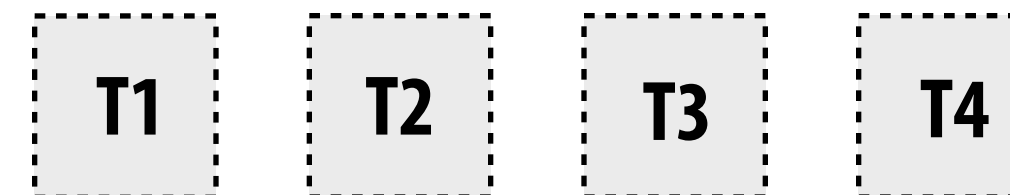
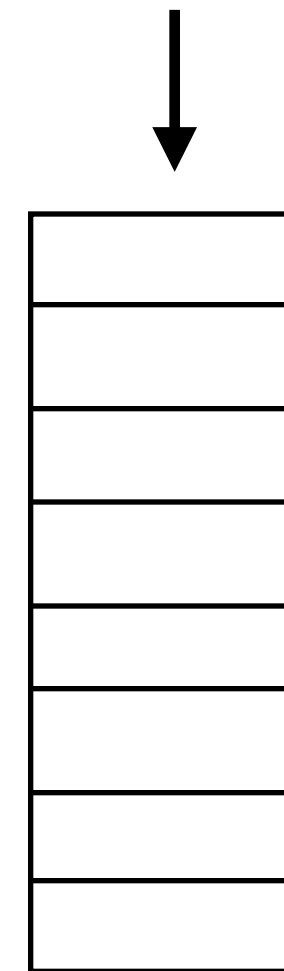


# Dynamic assignment using a work queue

**Sub-problems**  
(a.k.a. “tasks”, “work”)

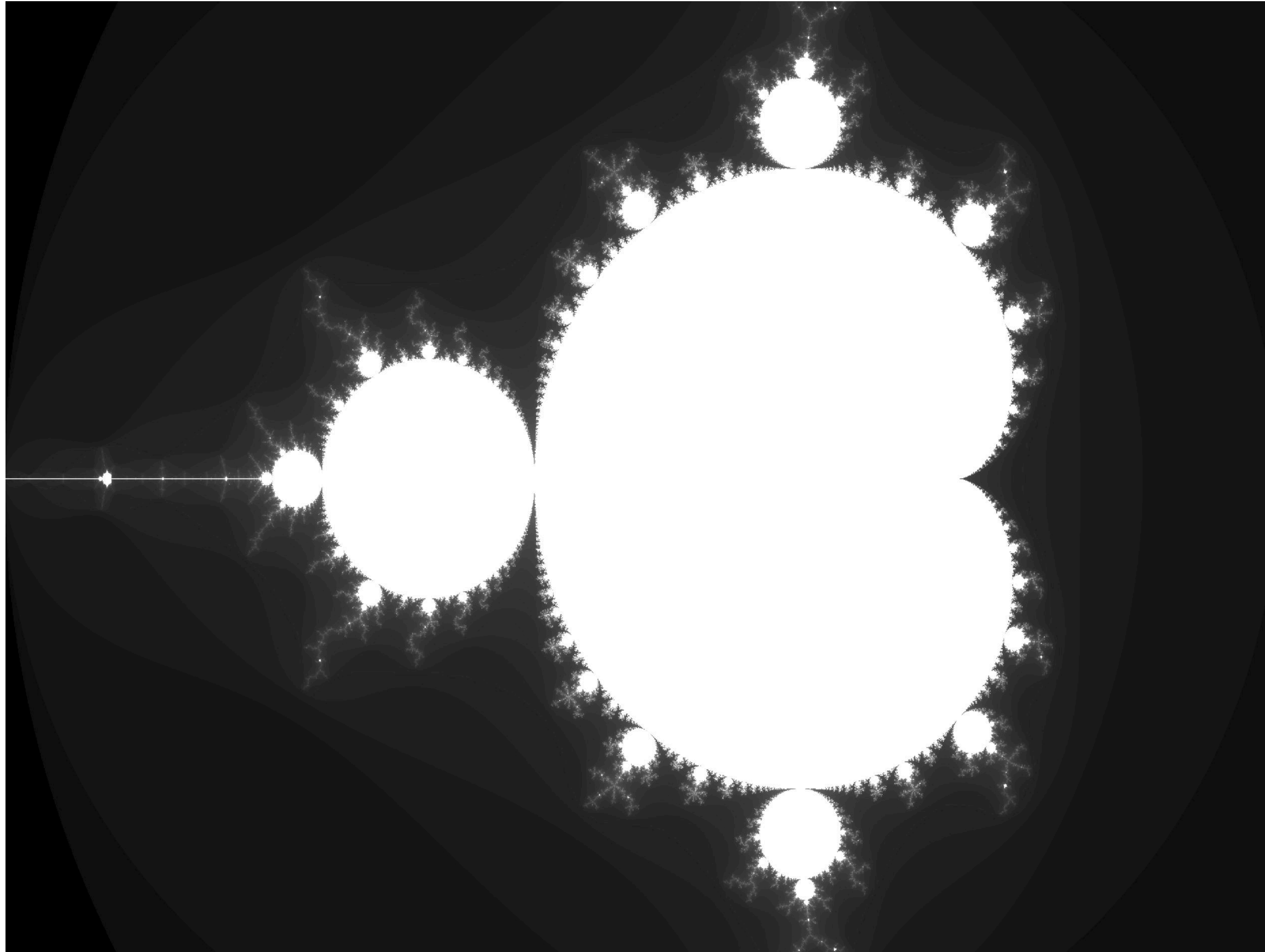


**Shared work queue: a list of work to do**  
(for now, let’s assume each piece of work is independent)



**Worker threads:**  
Pull data from shared work queue  
Push new work to queue as it is created

# Example from programming assignment 1 (prog 3)



**Why did breaking the problem up  
into many ISPC tasks improve  
performance?**

# What constitutes a piece of work?

What is a potential performance 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];

// assume elements of x are initialized here

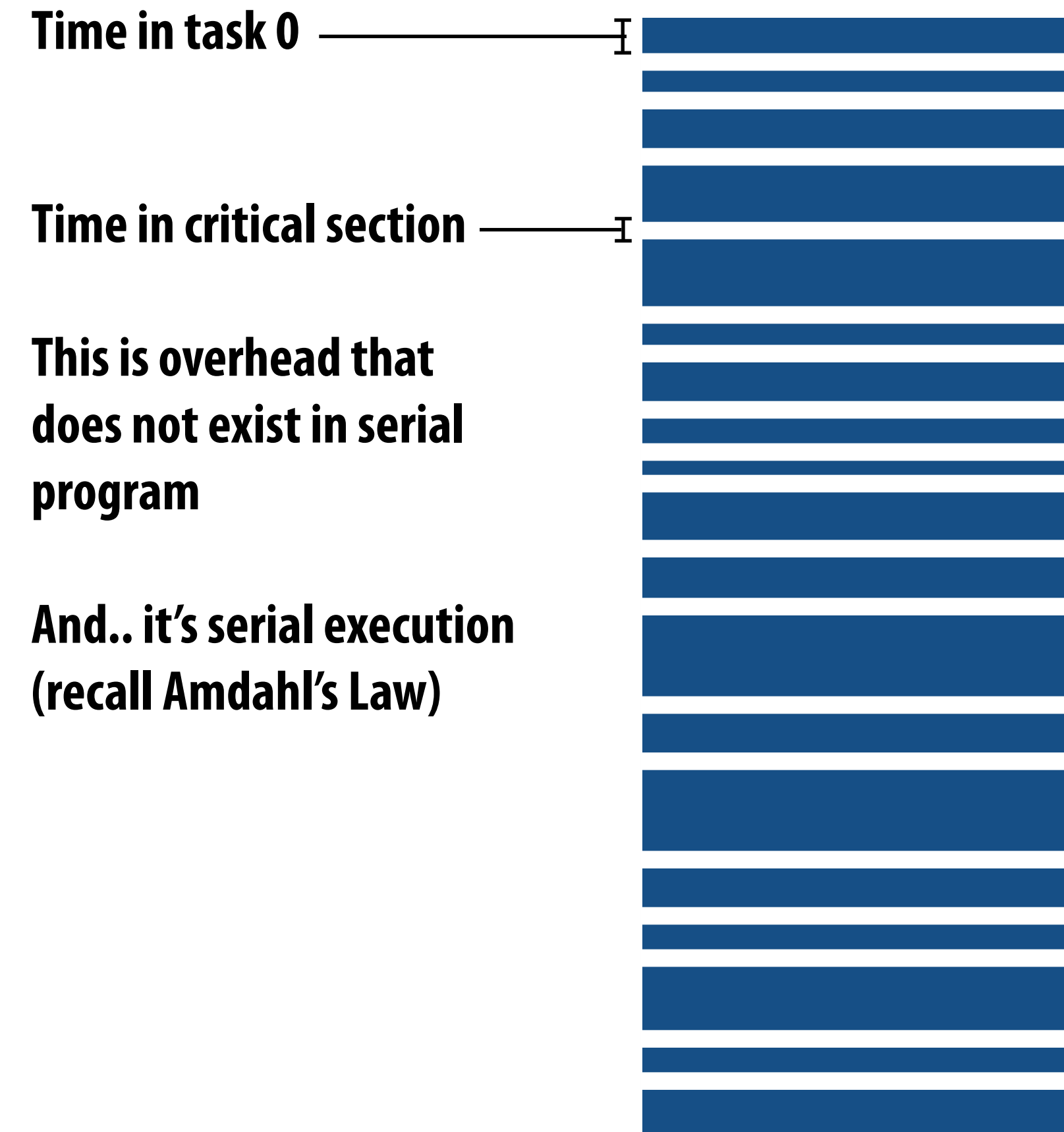
LOCK counter_lock;
int counter = 0;

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

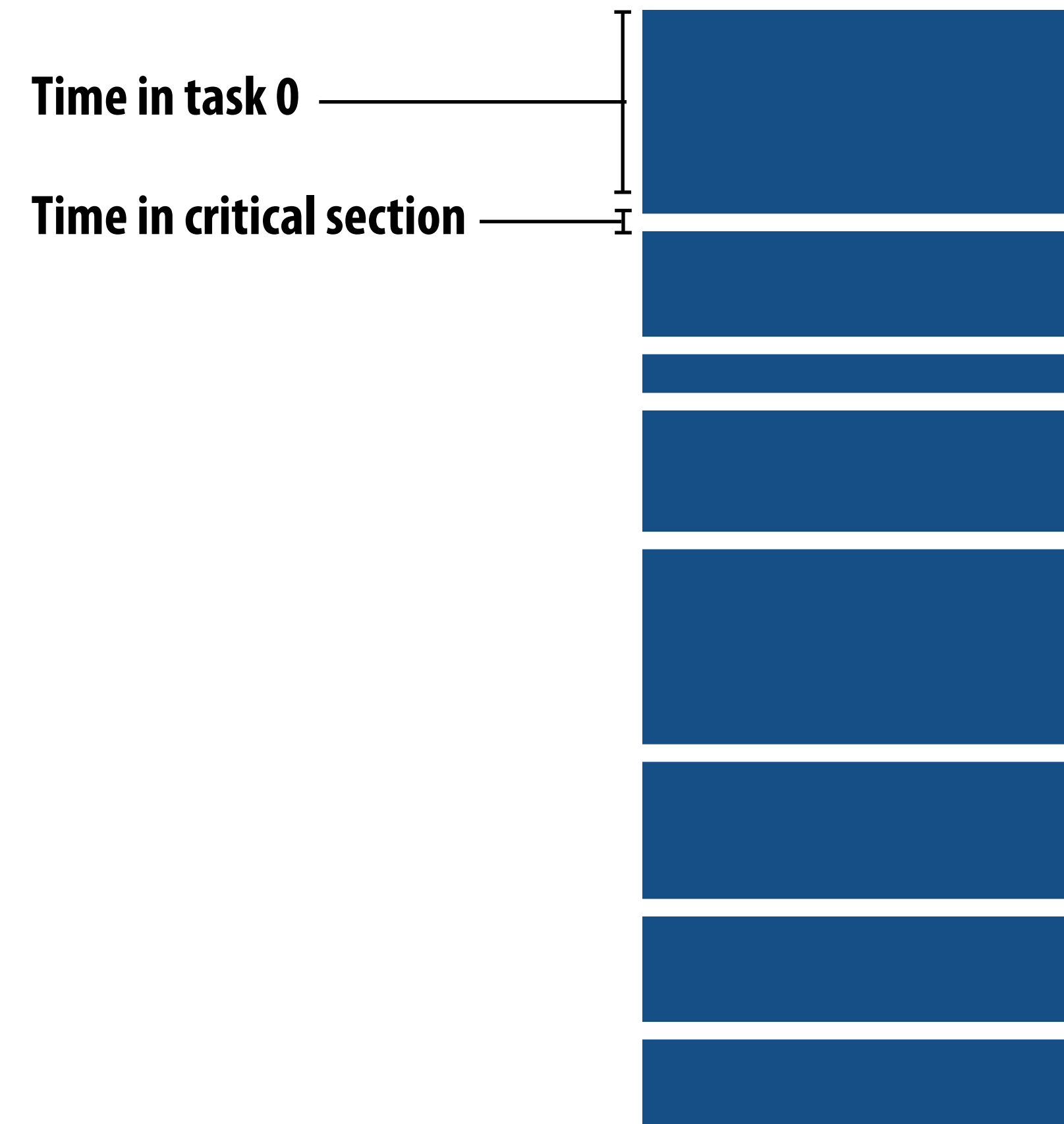
```
const int N = 1024;
const int GRANULARITY = 10;

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

// assume elements of x are initialized 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_primalty(x[i]);
}
```



**Coarse granularity partitioning: 1 “task” = 10 elements**  
**Decreased synchronization cost**  
**(Critical section entered 10 times less)**

# Choosing task size

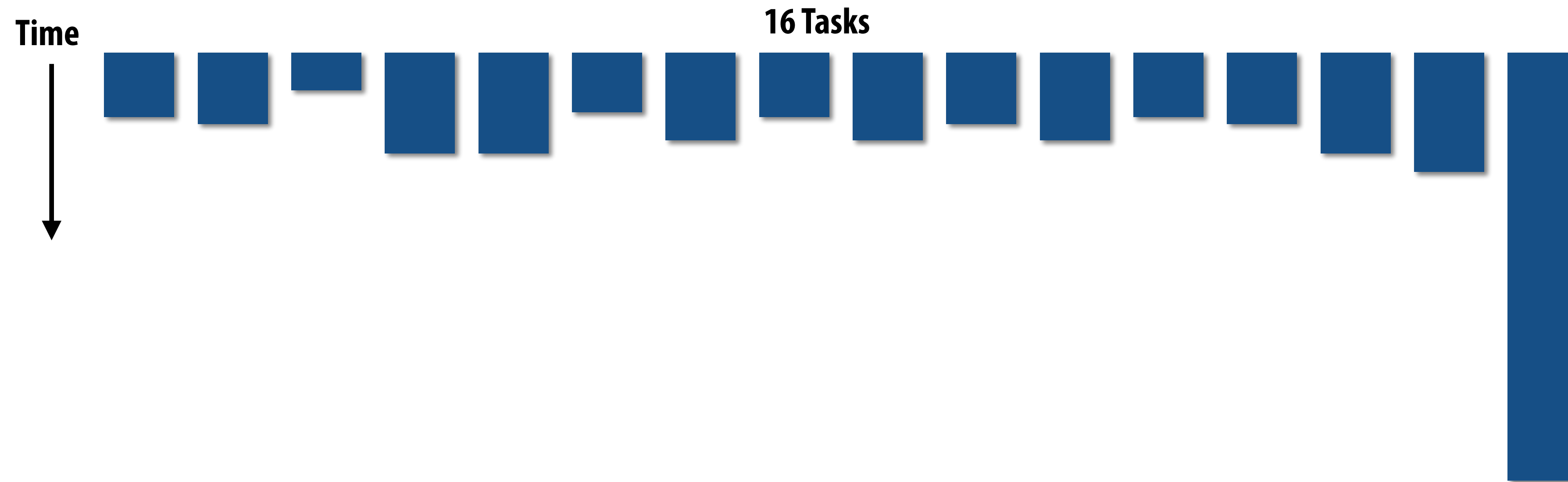
- **Useful to have many more tasks\* than processors**  
(many small tasks enables good workload balance via dynamic assignment)
  - Motivates small granularity tasks
- **But want as few tasks as possible to minimize overhead of managing the assignment**
  - Motivates large granularity tasks
- **Ideal 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. I'm not specifically referring to ISPC tasks

# Smarter task scheduling

Consider dynamic scheduling via a shared work queue

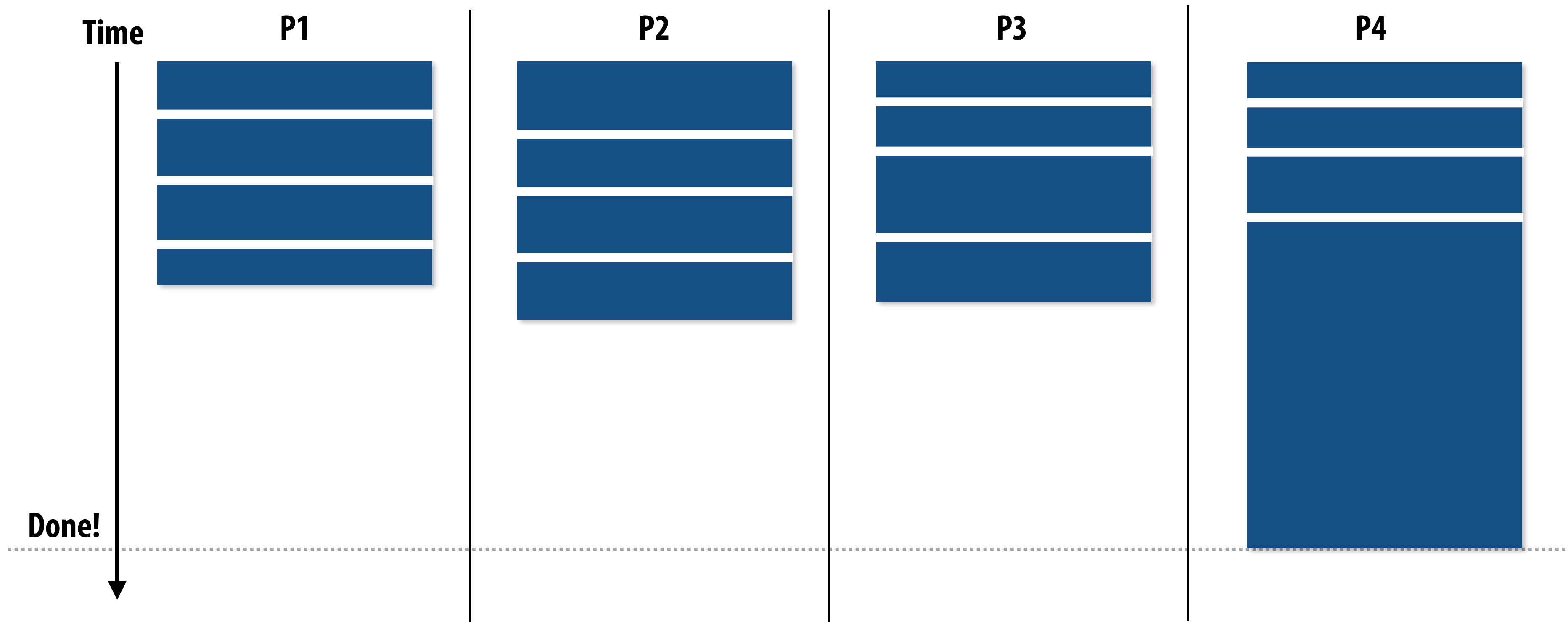
What happens if the system assigns these tasks to workers in left-to-right order?





# Smarter task scheduling

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



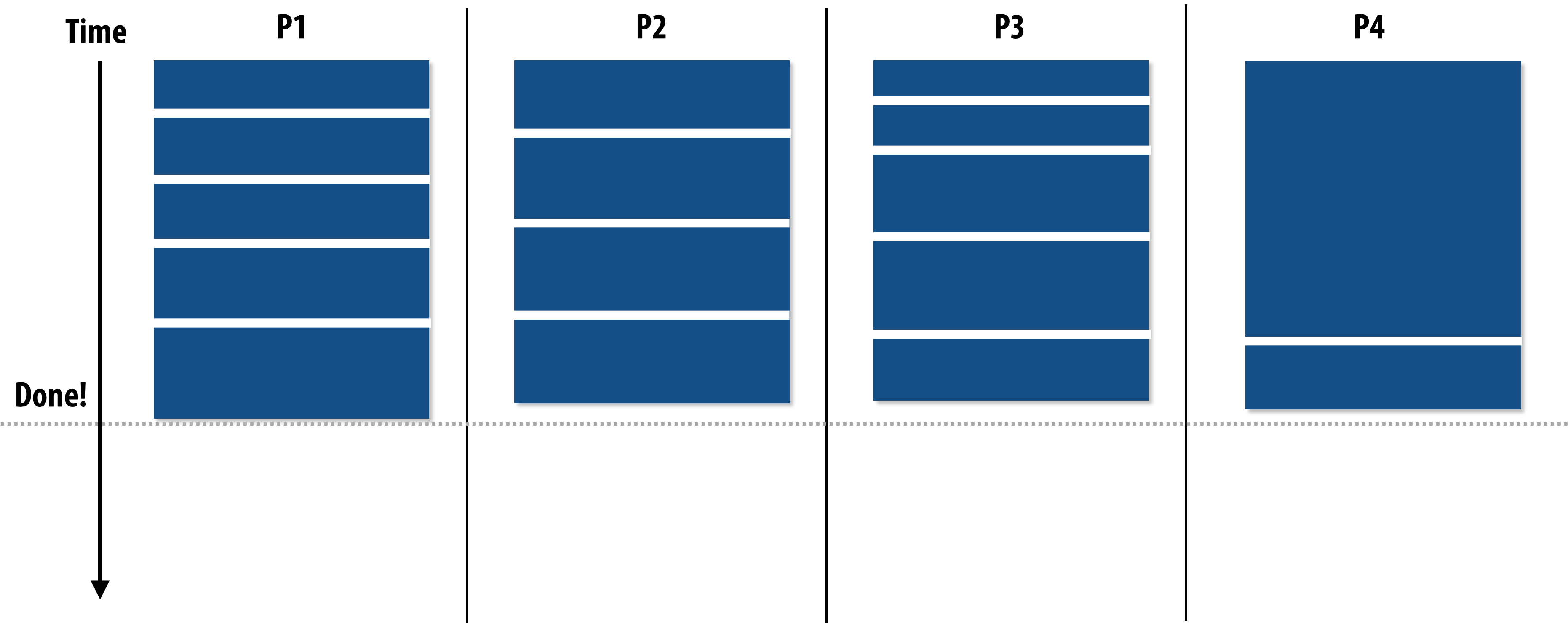
One possible solution to imbalance problem:

Divide work into a larger number of smaller tasks

- Hopefully this makes the “long pole” 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)

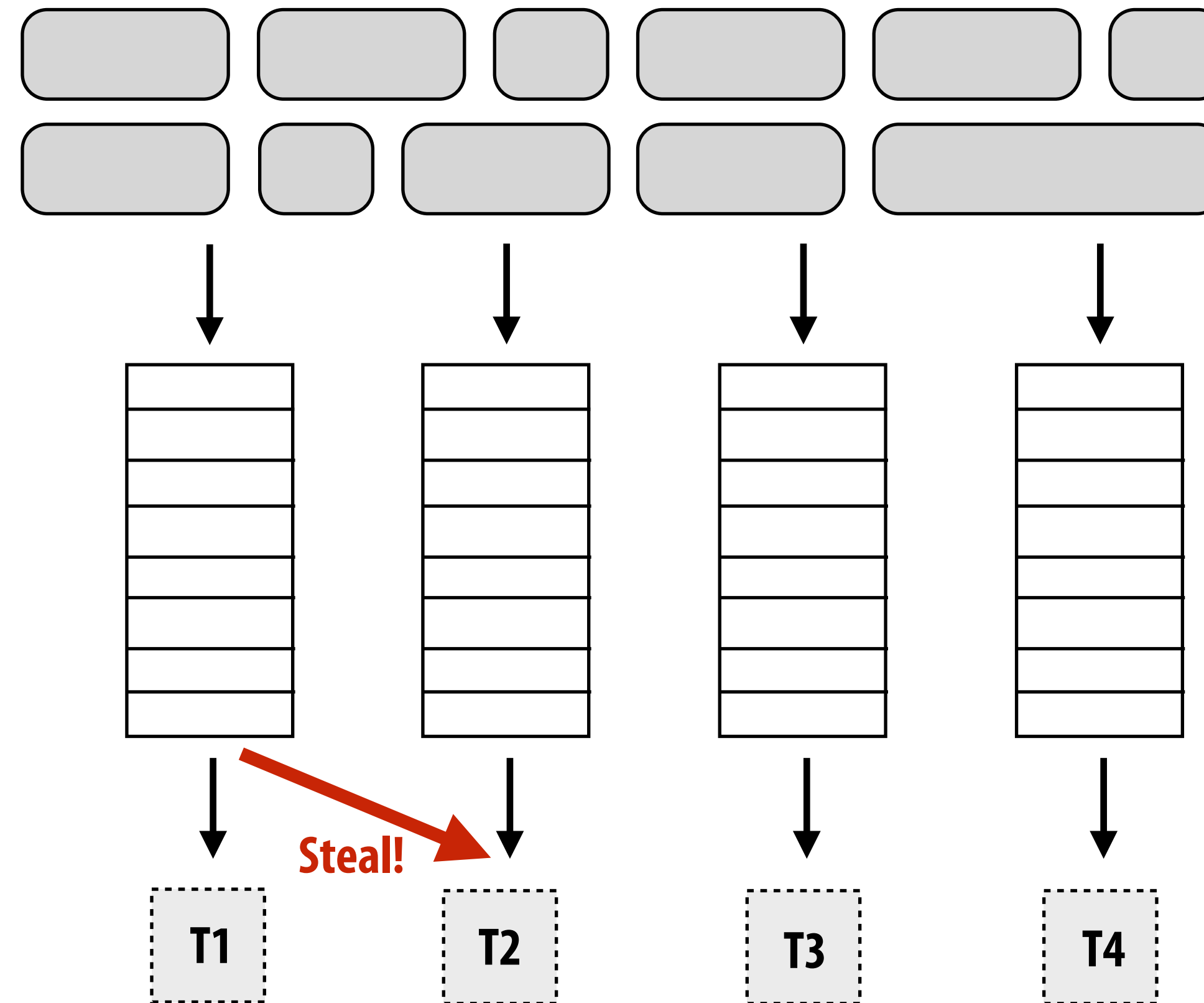
# Decreasing synchronization overhead using distributed queues

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

**Subproblems**  
(a.k.a. “tasks”, “work to do”)

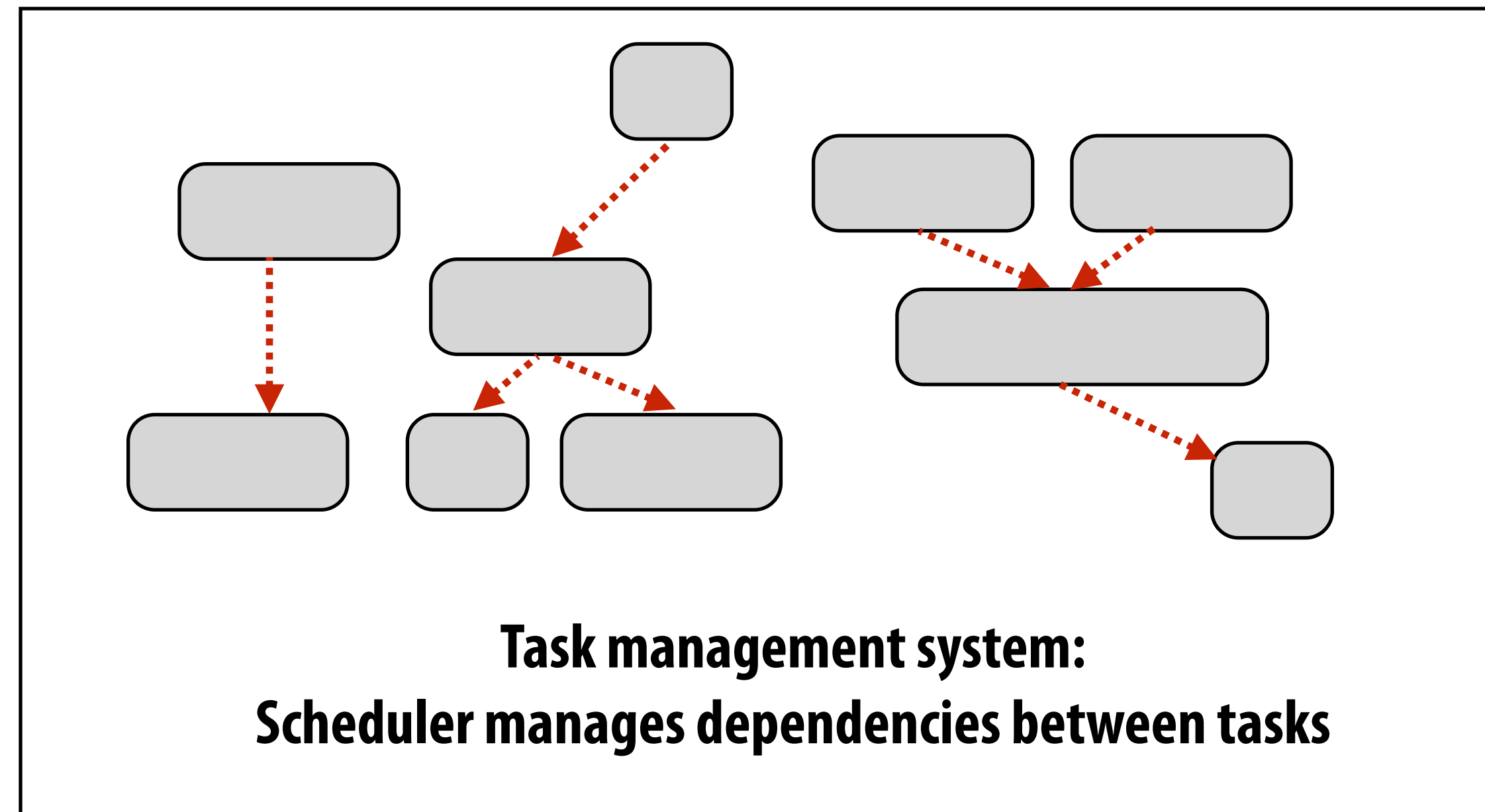
**Set of work queues**  
(In general, one per worker thread)

**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**



# Work in task queues need not be independent

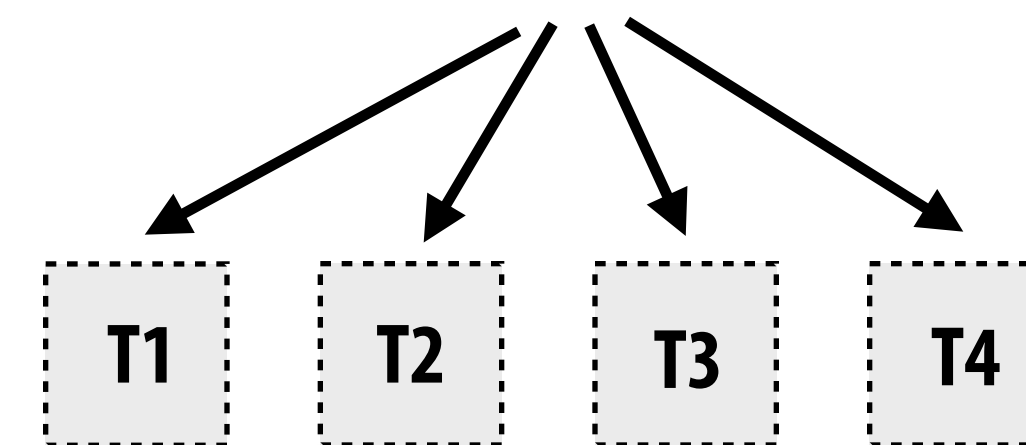
 = application-specified dependency



A task cannot be assigned to worker thread until all its task dependencies are satisfied

Workers can submit new tasks (with optional explicit dependencies) to task system

```
foo_handle = enqueue_task(foo);           // enqueue task foo (independent of all prior tasks)
bar_handle = enqueue_task(bar, foo_handle); // enqueue task bar, cannot run until foo is complete
```



# 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)

# Scheduling fork-join parallelism

# Common parallel programming patterns

## Data parallelism:

Perform same sequence of operations on many data elements

// ISPC foreach

```
foreach (i=0 ... N) {  
    B[i] = foo(A[i]);  
}
```

// ISPC bulk task launch

```
launch[numTasks] myFooTask(A, B);
```

// using higher-order function 'map'

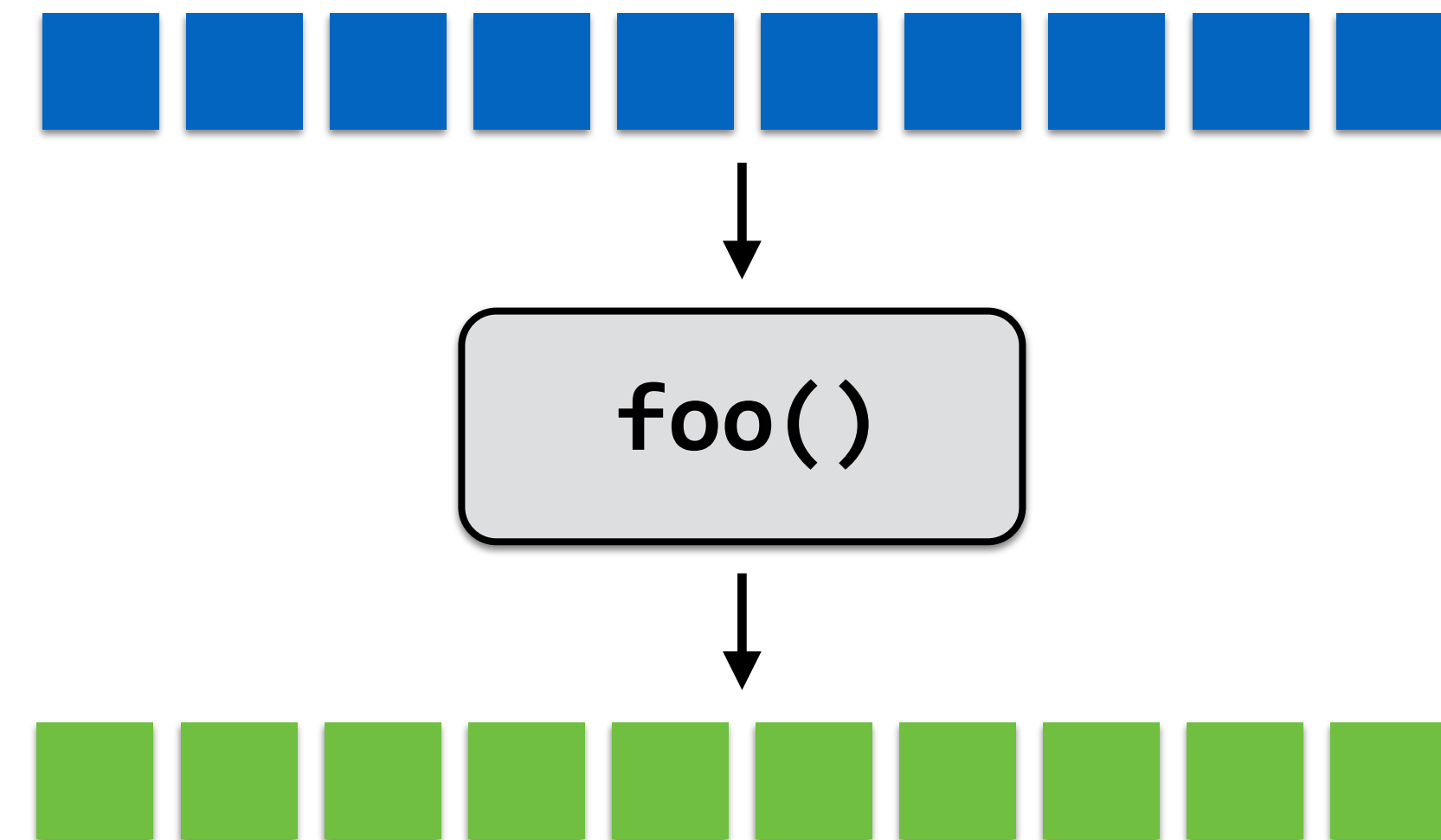
```
map(foo, A, B);
```

// openMP parallel for

```
#pragma omp parallel for  
for (int i=0; i<N; i++) {  
    B[i] = foo(A[i]);  
}
```

// bulk CUDA thread launch (GPU programming, in a future lecture)

```
foo<<<numBlocks, threadsPerBlock>>>(A, B);
```



# Common parallel programming patterns

**Explicit management of parallelism with threads:**

**Create one thread per execution unit (or per amount of desired concurrency)**

**- Example below: C code with C++ threads**

```
float* A;  
float* B;
```

```
// initialize arrays A and B here
```

```
void myFunction(float* A, float* B { ... }
```

```
std::thread thread[NUM_HW_EXEC_CONTEXTS];
```

```
for (int i=0; i<NUM_HW_EXEC_CONTEXTS; i++) {  
    thread[i] = std::thread(myFunction, A, B);  
}
```

```
for (int i=0; i<num_cores; i++) {  
    thread[i].join();  
}
```



# Consider divide-and-conquer algorithms

## Quick sort:

```
// sort elements from 'begin' up to (but not including) 'end'
```

```
void quick_sort(int* begin, int* end) {
```

```
    if (begin >= end-1)
```

```
        return;
```

```
    else {
```

```
        // choose partition key and partition elements
```

```
        // by key, return position of key as 'middle'
```

```
        int* middle = partition(begin, end);
```

```
        quick_sort(begin, middle);
```

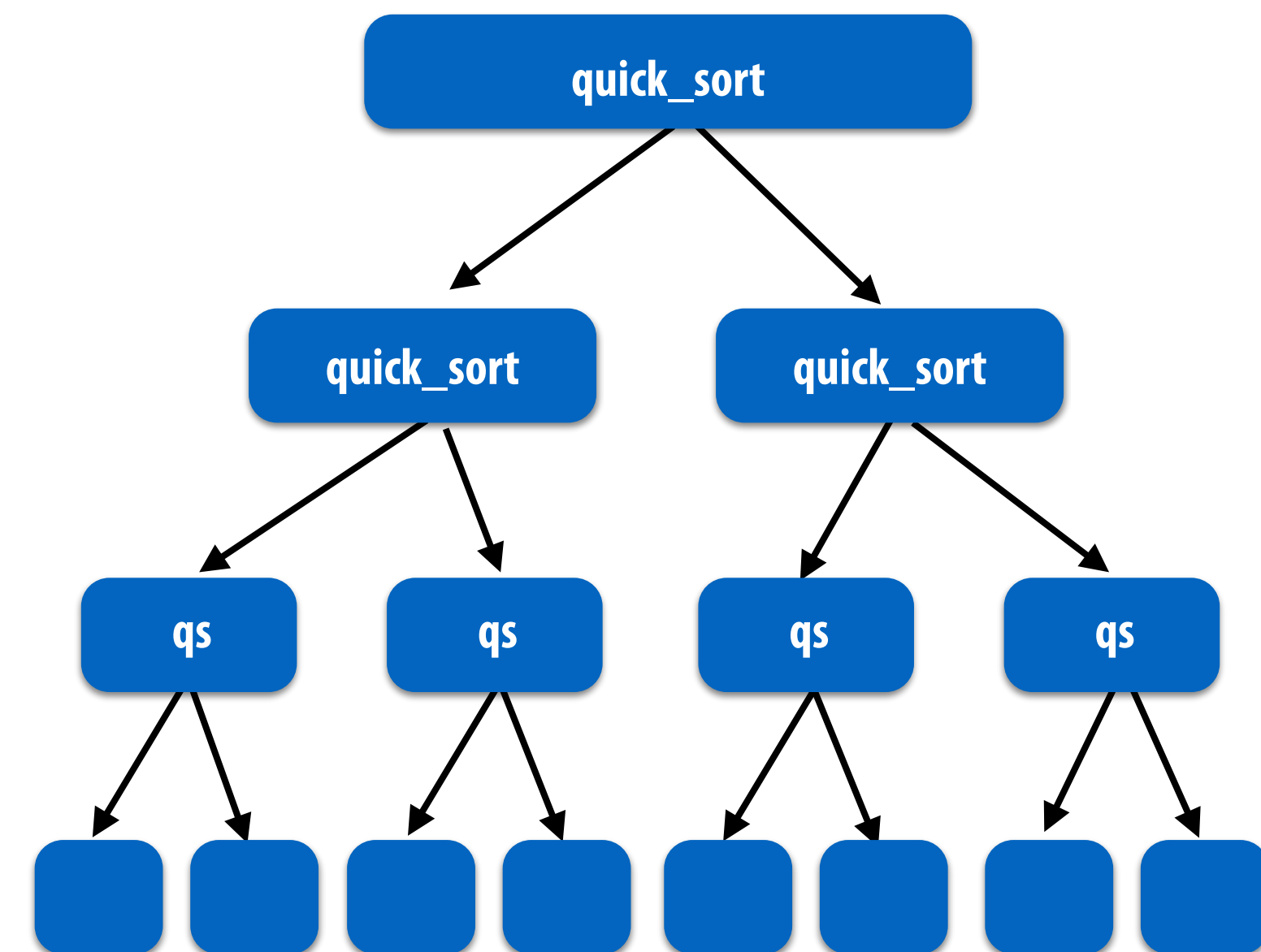
```
        quick_sort(middle+1, last);
```

```
    }
```

```
}
```

**independent work!**

## Dependencies



# Fork-join pattern

- Natural way to express the independent work that is inherent in divide-and-conquer algorithms
- This lecture's code examples will be in Cilk Plus
  - C++ language extension
  - Originally developed at MIT, now adapted as open standard (in GCC, Intel ICC)

`cilk_spawn foo(args);` ← **“fork” (create new logical thread of control)**

**Semantics:** invoke `foo`, but unlike standard function call, caller may continue executing asynchronously with execution of `foo`.

`cilk_sync;` ← **“join”**

**Semantics:** returns when all calls spawned by current function have completed. (“sync up” with the spawned calls)

**Note:** there is an implicit `cilk_sync` at the end of every function that contains a `cilk_spawn` (implication: when a Cilk function returns, all work associated with that function is complete)

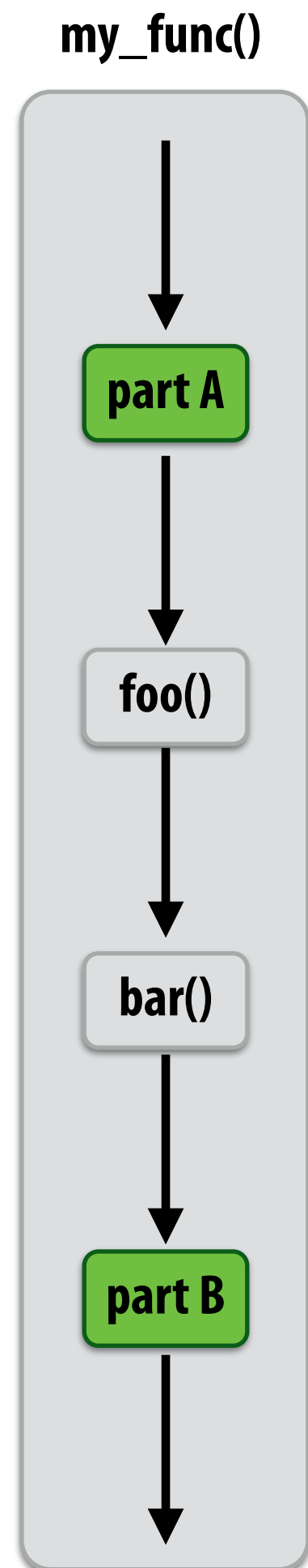
# Call-return of a function in C\*

```
void my_func() {  
    // calling function (part A)  
  
    foo();  
    bar();  
  
    // calling function (part B)  
  
}
```

**Semantics of a function call:**

**Control moves to the function that is called  
(Thread executes instructions for the function)**

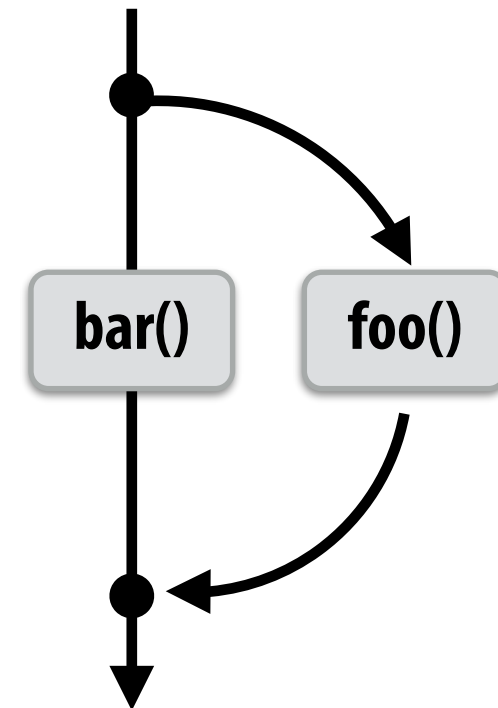
**When function returns, control returns back to caller  
(thread resumes executing instructions from the caller)**



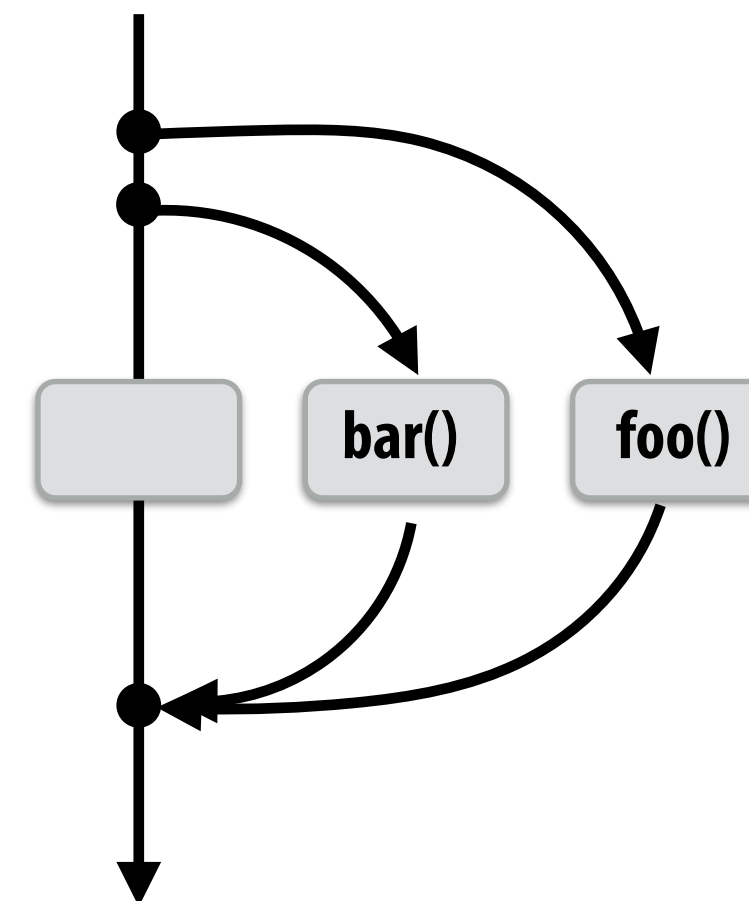
\* And many other languages

# Basic Cilk Plus examples

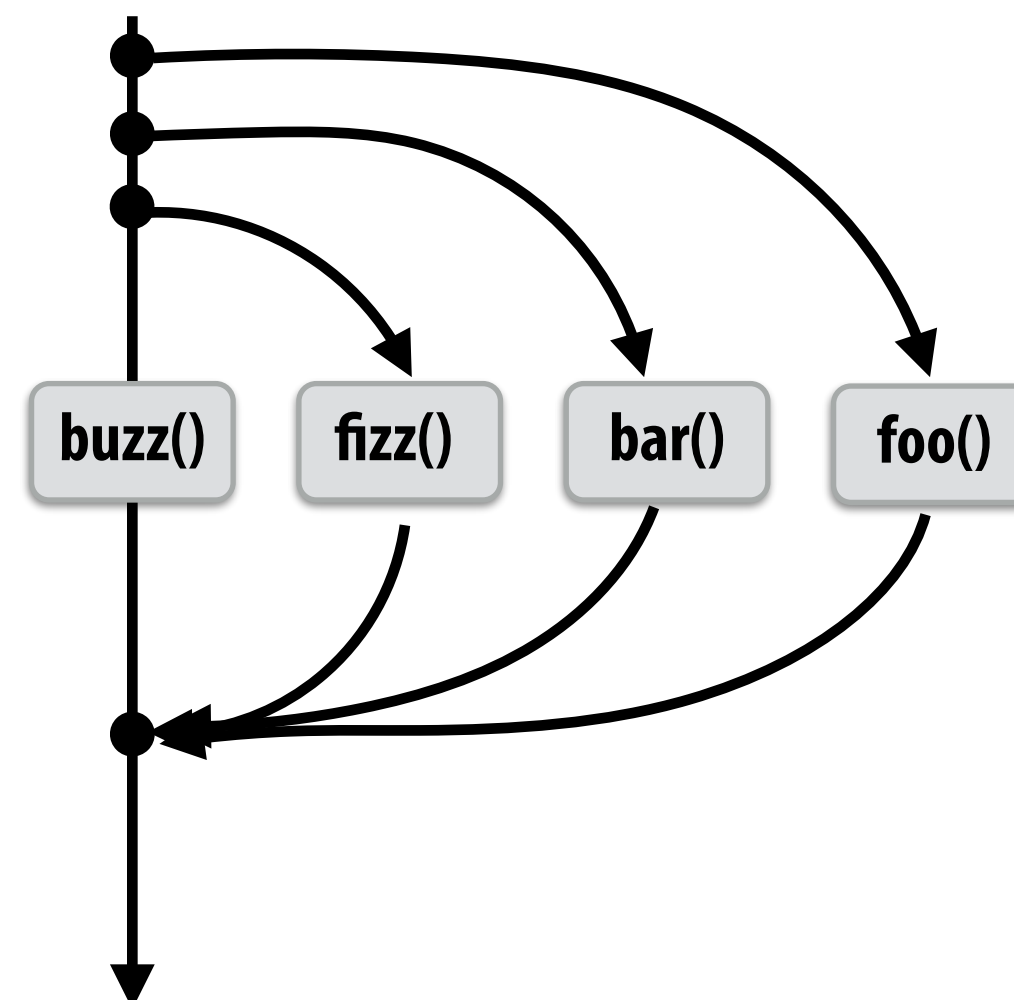
```
// foo() and bar() may run in parallel
cilk_spawn foo();
bar();
cilk_sync;
```



```
// foo() and bar() may run in parallel
cilk_spawn foo();
cilk_spawn bar();
cilk_sync;
```



```
// foo, bar, fizz, buzz, may run in parallel
cilk_spawn foo();
cilk_spawn bar();
cilk_spawn fizz();
buzz();
cilk_sync;
```



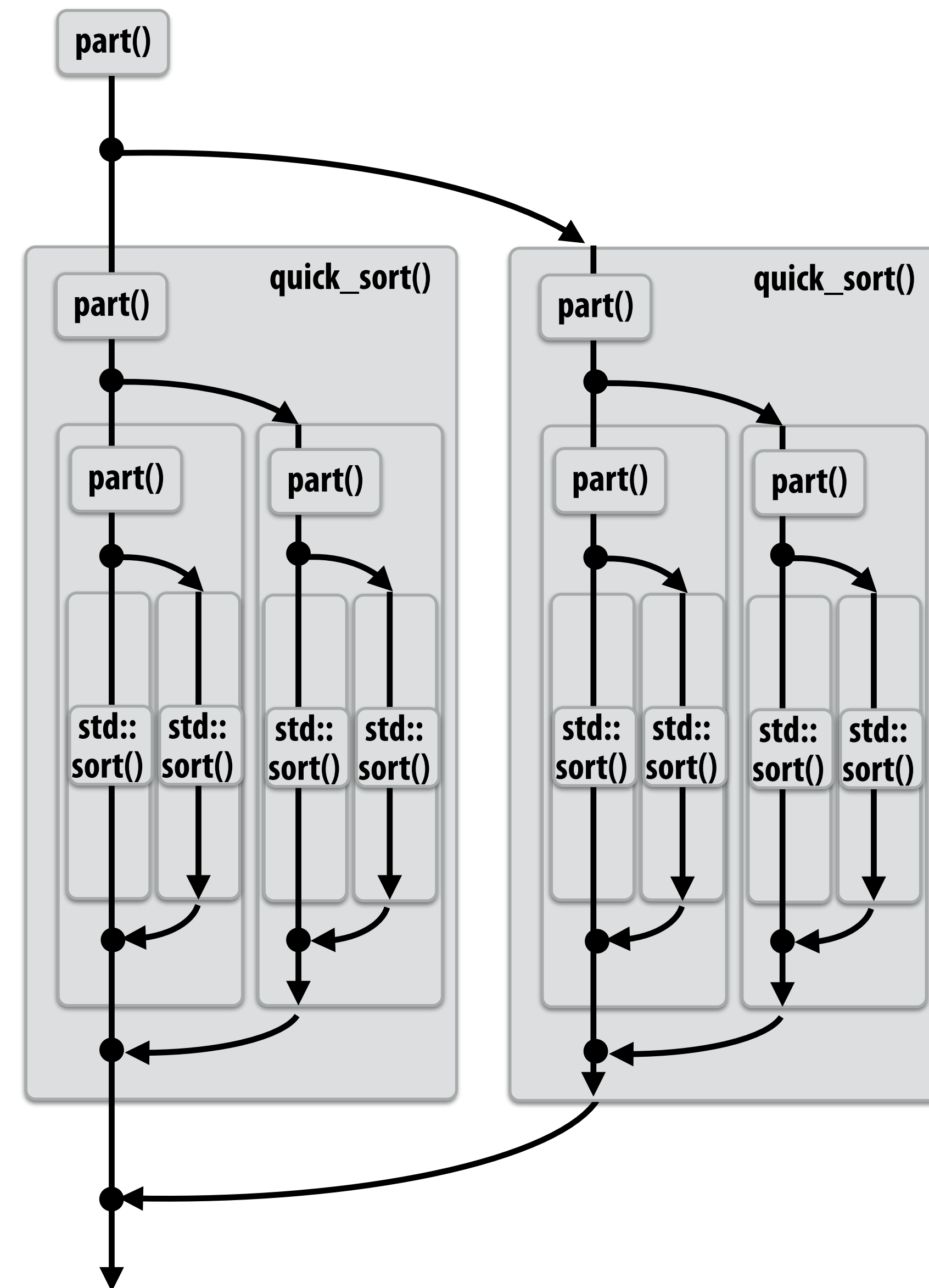
# Abstraction vs. implementation

- Notice that the `cilk_spawn` abstraction does not specify how or when spawned calls are scheduled to execute
  - Only that they may be run concurrently with caller (and with all other calls spawned by the caller)
  - Question: Is an implementation of Cilk correct if it implements `cilk_spawn foo()` the same way as it implements a normal function call to `foo()`?
- But `cilk_sync` does serve as a constraint on scheduling
  - All spawned calls must complete before `cilk_sync` returns

# Parallel quicksort in Cilk Plus

```
void quick_sort(int* begin, int* end) {  
    if (begin >= end - PARALLEL_CUTOFF)  
        std::sort(begin, end);  
    else {  
        int* middle = partition(begin, end);  
        cilk_spawn quick_sort(begin, middle);  
        quick_sort(middle+1, last);  
    }  
}
```

Sort sequentially if problem size is sufficiently small (overhead of spawn trumps benefits of potential parallelization)



# Writing fork-join programs

- **Main idea: expose independent work (potential parallelism) to the system using `cilk_spawn`**
- **Recall parallel programming rules of thumb**
  - **Want at least as much work as parallel execution capability (e.g., program should probably spawn at least as much work as needed to fill all the machine's processing resources)**
  - **Want more independent work than execution capability to allow for good workload balance of all the work onto the cores**
    - **"parallel slack" = ratio of independent work to machine's parallel execution capability (in practice: ~8 is a good ratio)**
  - **But not too much independent work so that granularity of work is too small (too much slack incurs overhead of managing fine-grained work)**

# Scheduling fork-join programs

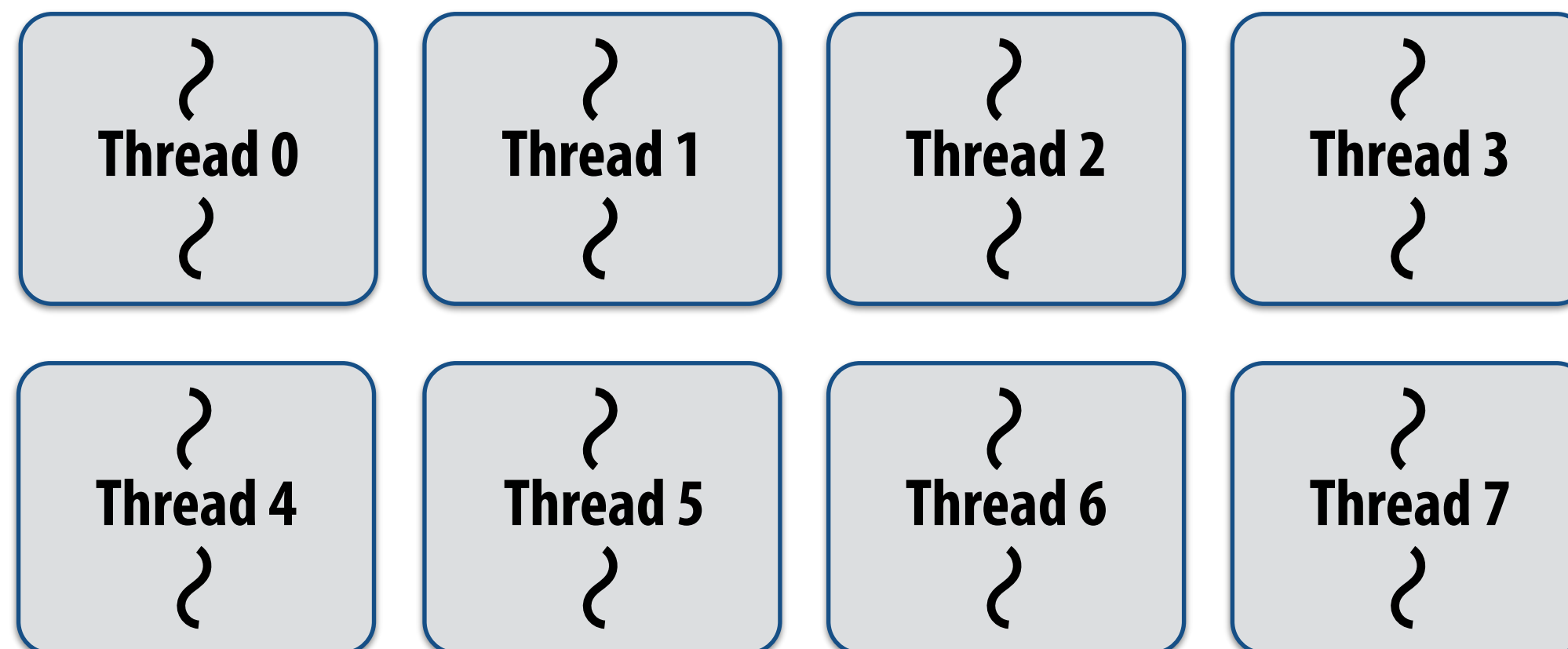
- **Consider very simple scheduler:**
  - Launch pthread for each `cilk_spawn` using `pthread_create`
  - Translate `cilk_sync` into appropriate `pthread_join` calls
- **Potential performance problems?**
  - Heavyweight spawn operation
  - Many more concurrently running threads than cores
    - Context switching overhead
    - Larger working set than necessary, less cache locality

**Note: now we are going to talk about the implementation of Cilk**



# Pool of worker threads

- **The Cilk Plus runtime maintains pool of worker threads**
  - **Think: all threads are created at application launch \***
  - **Exactly as many worker threads as execution contexts in the machine**



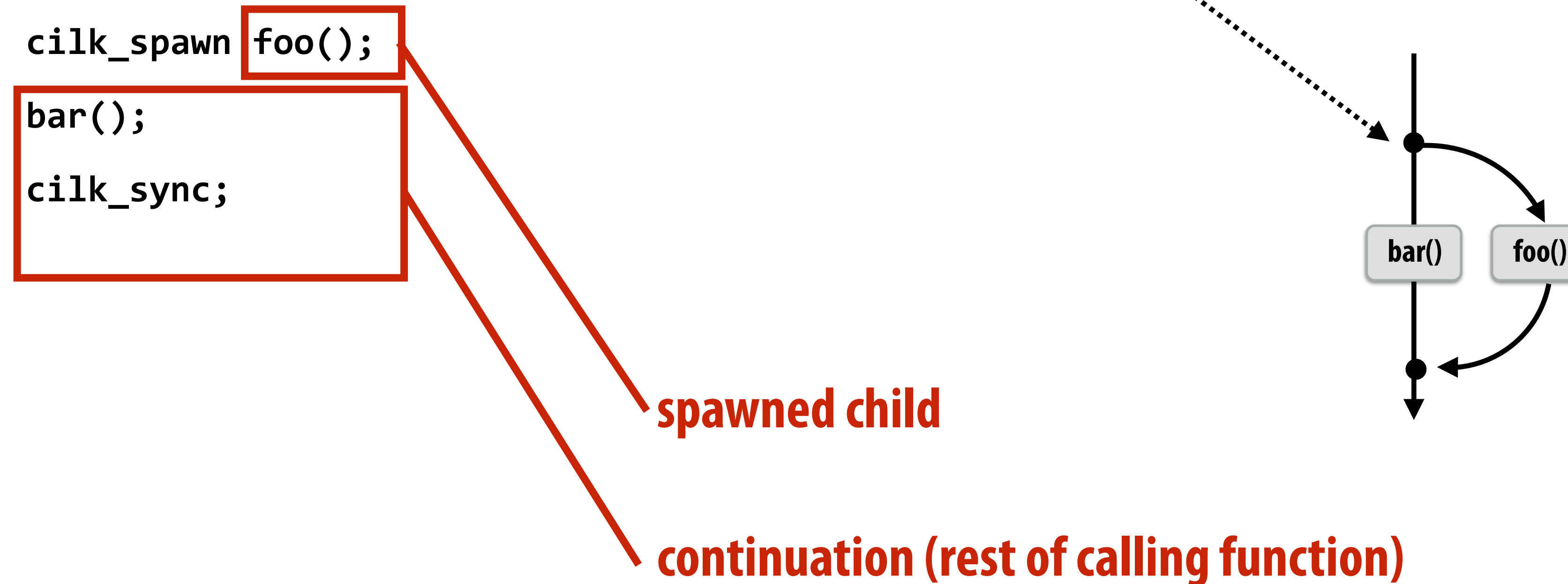
**Example: Eight thread worker pool for my quad-core laptop with Hyper-Threading**

```
while (work_exists()) {  
    work = get_new_work();  
    work.run();  
}
```

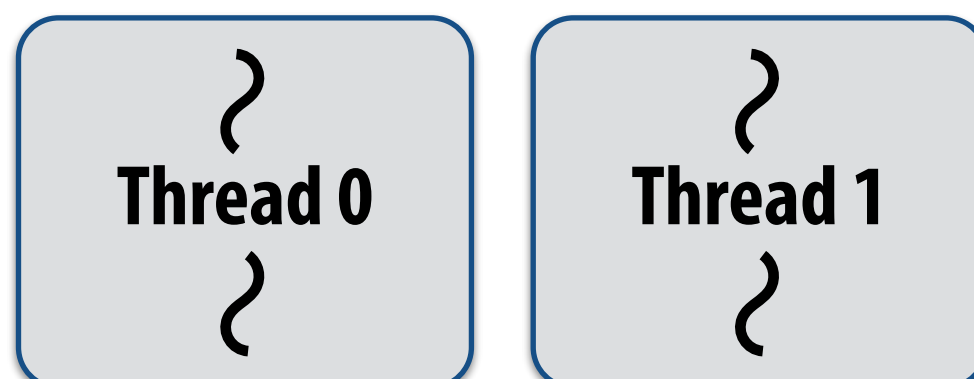
**\* It's perfectly fine to think about it this way, but in reality, runtimes tend to be lazy and initialize worker threads on the first Cilk spawn. (This is a common implementation strategy, ISPC does the same with worker threads that run ISPC tasks.)**

# Consider execution of the following code

Specifically, consider execution from the point `foo()` is spawned



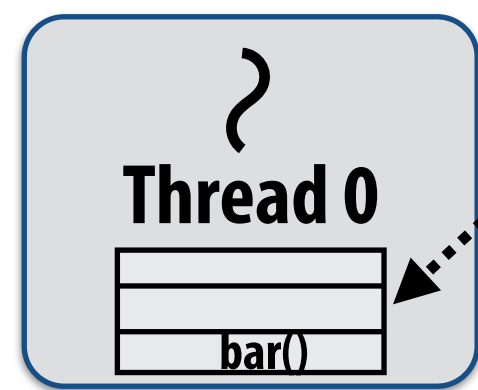
**Assignment question: what threads should `foo()` and `bar()` be executed by?**



# First, consider a serial implementation

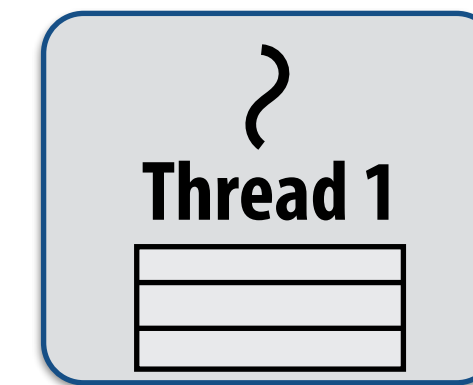
Run child first... via a regular function call

- Thread runs `foo()`, then returns from `foo()`, then runs `bar()`
- Continuation is implicit in the thread's stack



Traditional thread call stack  
(indicates `bar()` will be run next  
after return from `foo()`)

Executing `foo()`...

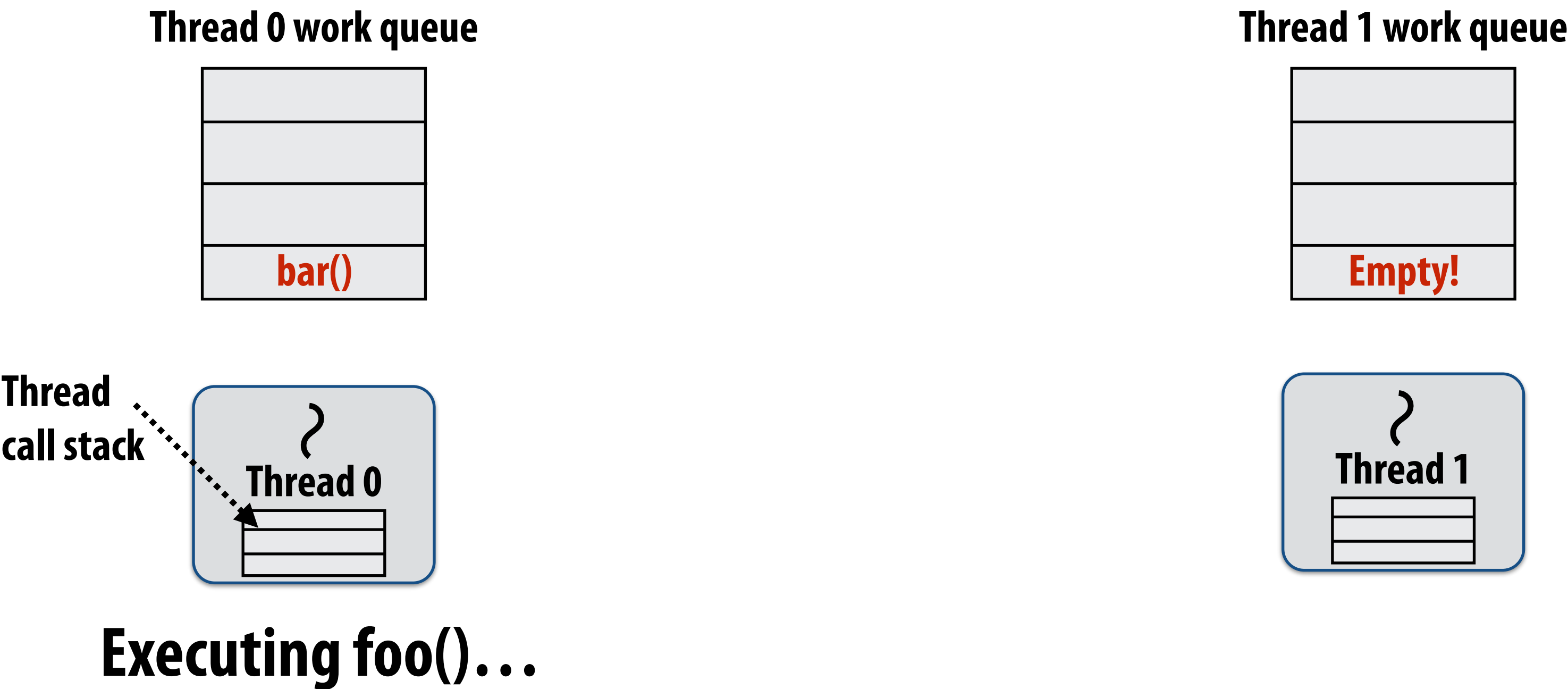


What if, while executing `foo()`,  
thread 1 goes idle...

Thread 1 could be performing `bar()`  
at this time!

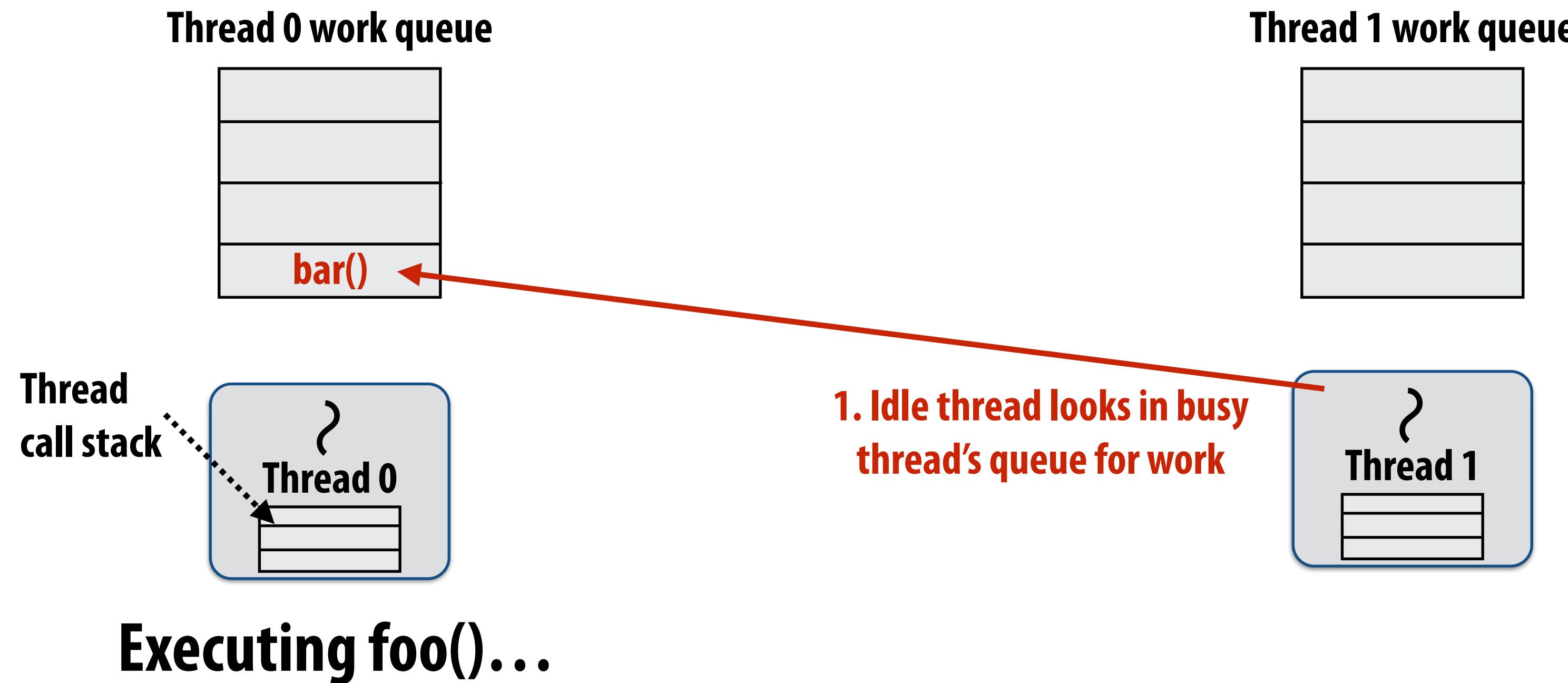
# Per-thread work queues store “work to do”

Upon reaching `cilk_spawn foo()`, thread places continuation in its work queue, and begins executing `foo()`.



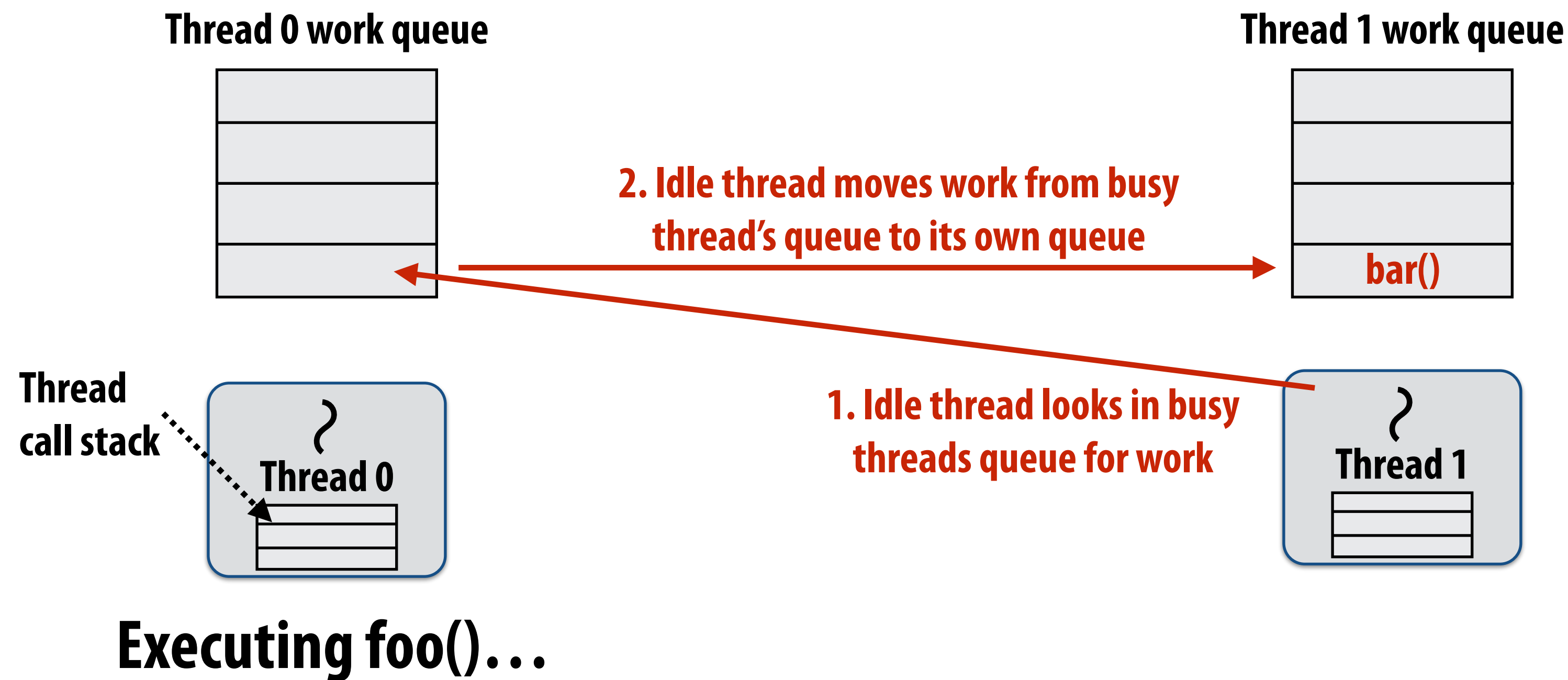
# Idle threads “steal” work from busy threads

If thread 1 goes idle (a.k.a. there is no work in its own queue), then it looks in thread 0’s queue for work to do.



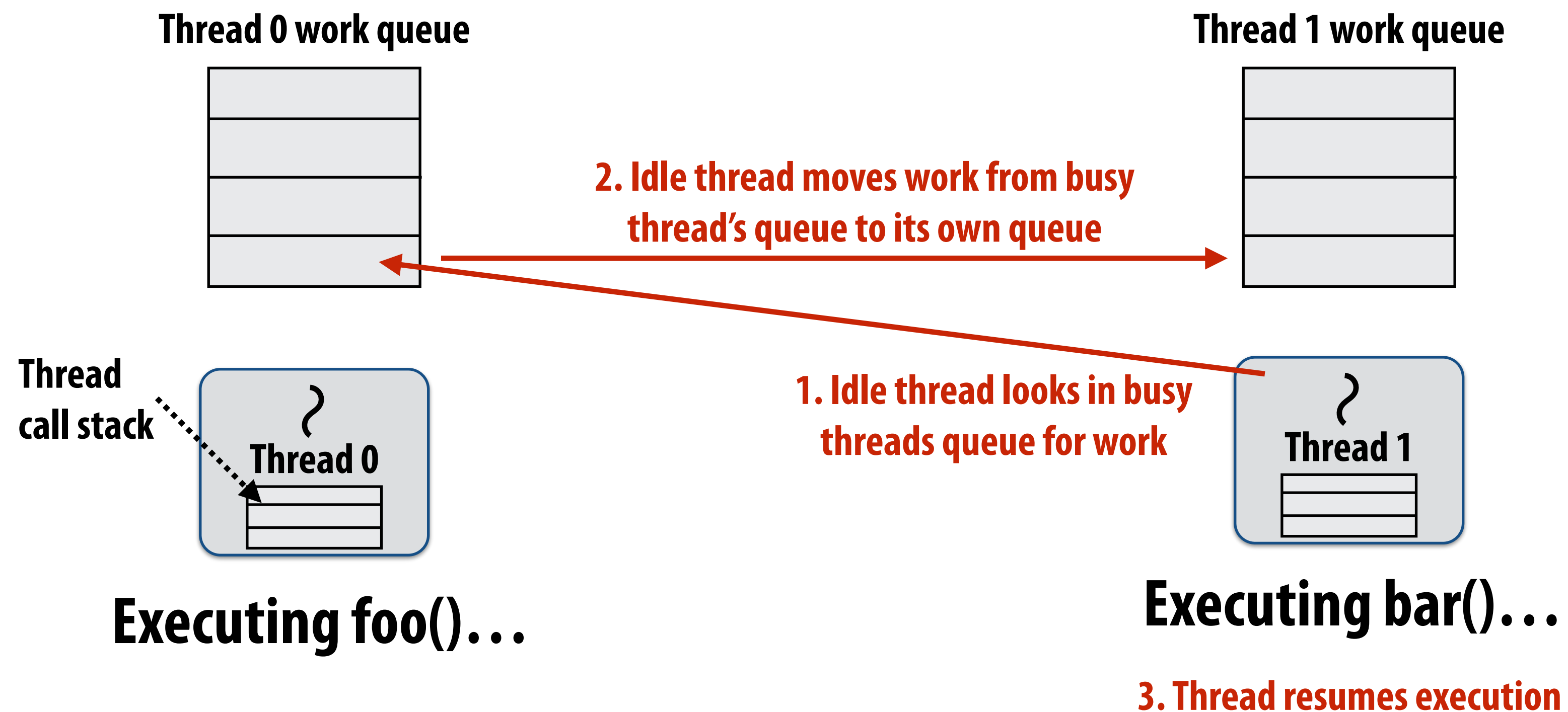
# Idle threads “steal” work from busy threads

If thread 1 goes idle (a.k.a. there is no work in its own queue), then it looks in thread 0’s queue for work to do.

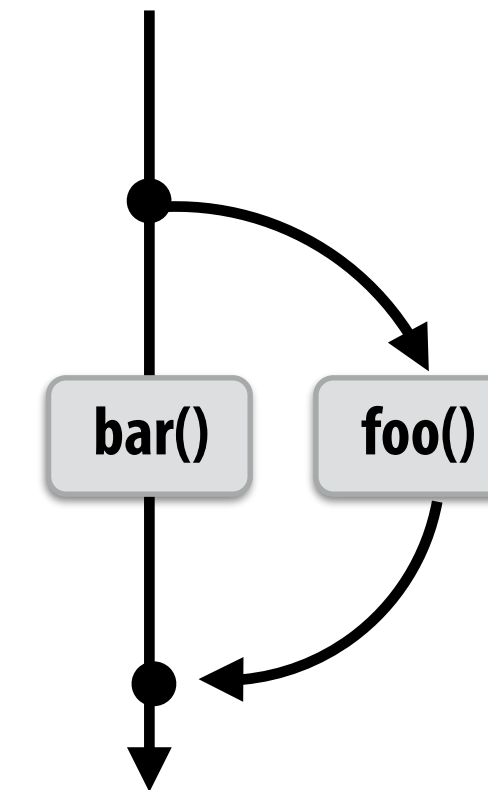
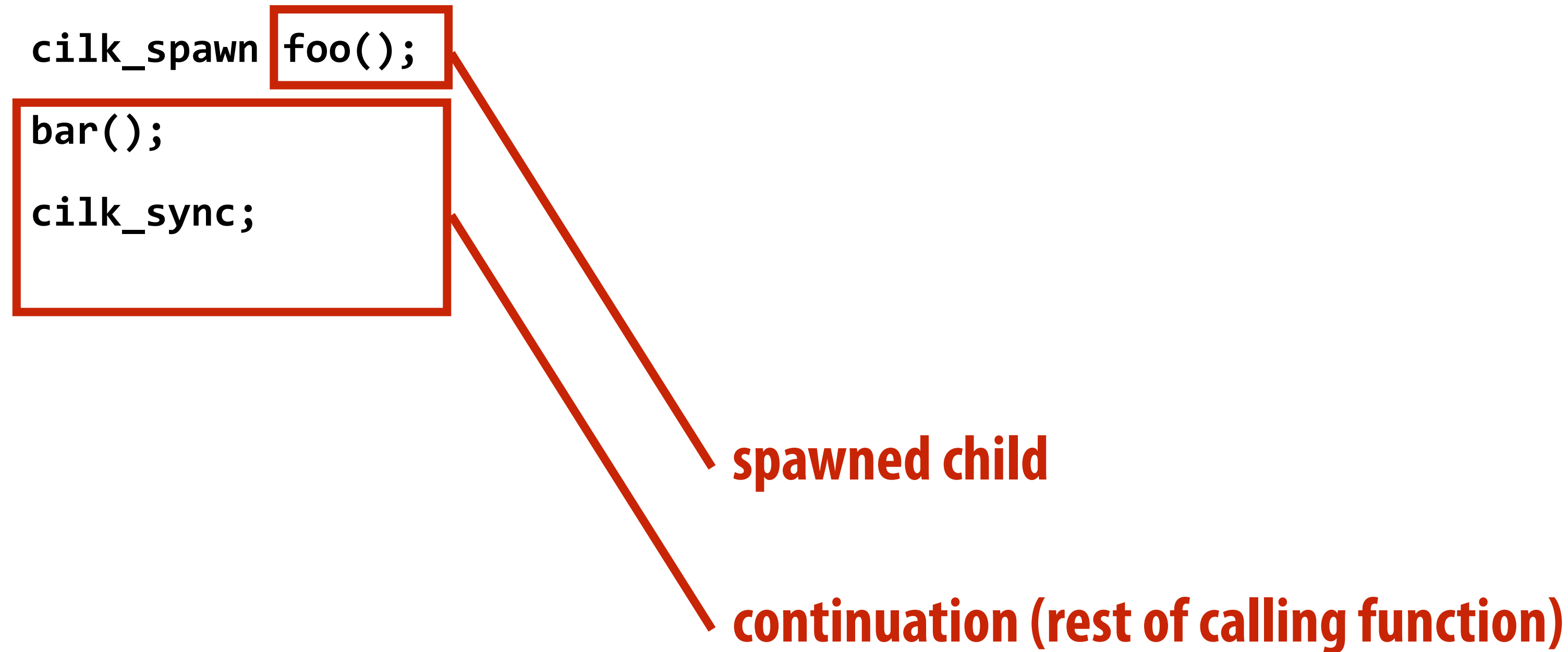


# Idle threads “steal” work from busy threads

If thread 1 goes idle (a.k.a. there is no work in its own queue), then it looks in thread 0’s queue for work to do.



# At spawn, should calling thread run the child or the continuation?



## Run continuation first: queue child for later execution

- Child is made available for stealing by other threads (“child stealing”)

## Run child first: enqueue continuation for later execution

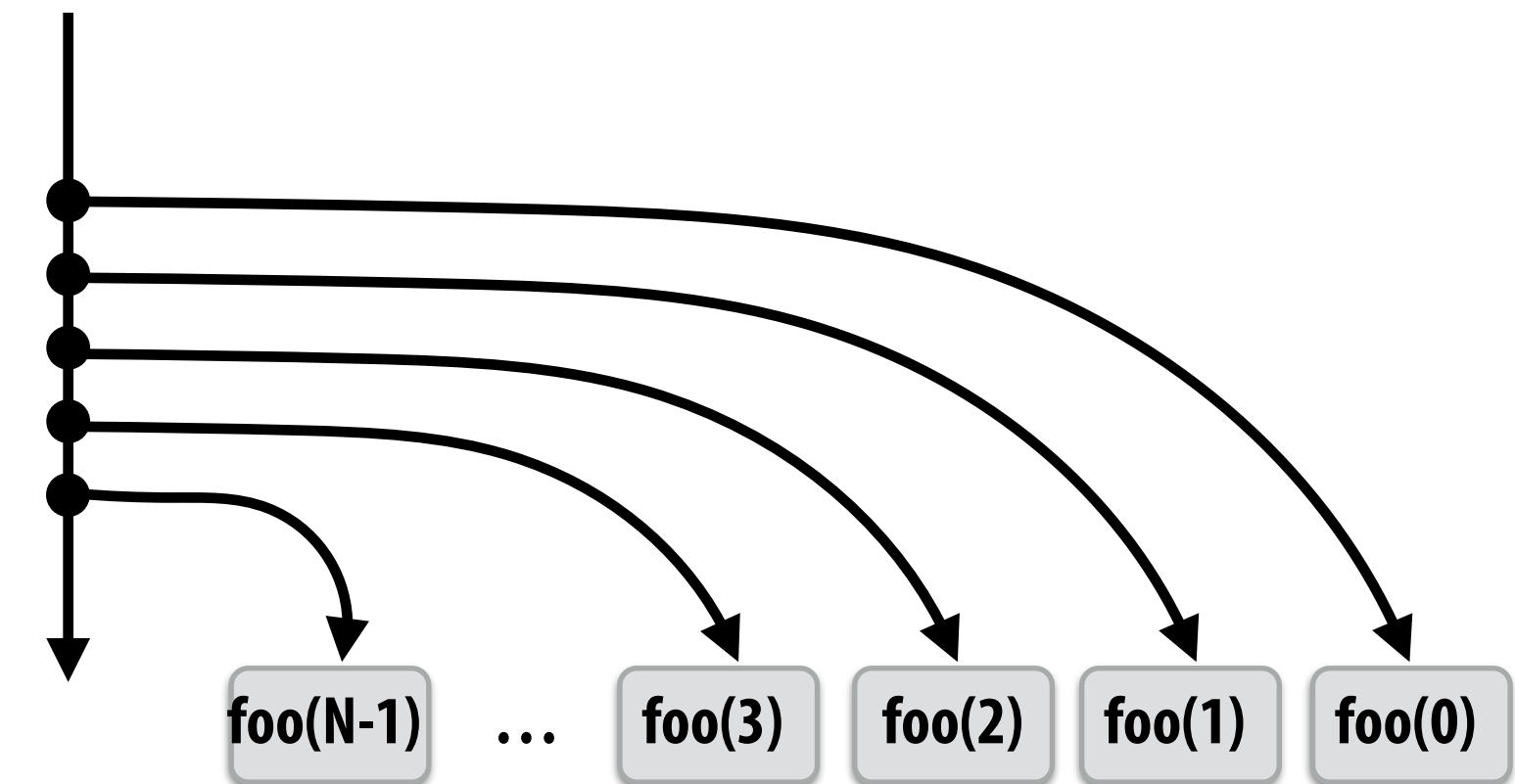
- Continuation is made available for stealing by other threads (“continuation stealing”)

## Which implementation do we choose?



# Consider thread executing the following code

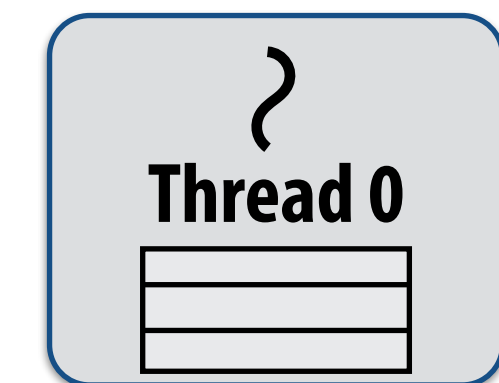
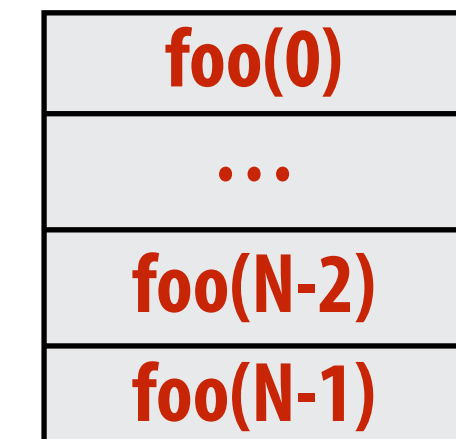
```
for (int i=0; i<N; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync;
```



## ■ Run continuation first (“child stealing”)

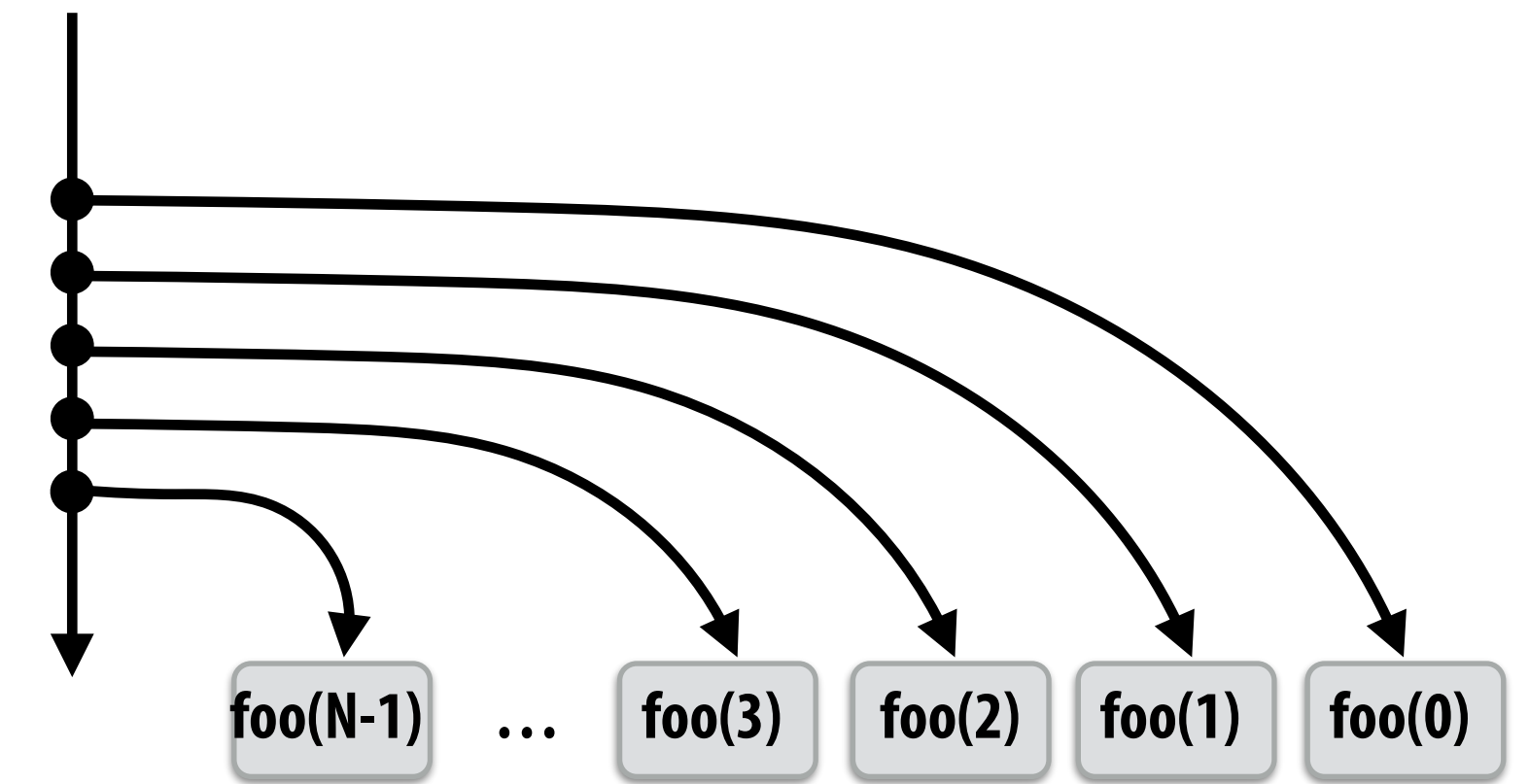
- Caller thread spawns work for all iterations before executing any of it
- Think: breadth-first traversal of call graph.  $O(N)$  space for spawned work (maximum space)
- If no stealing, execution order is very different than that of program with `cilk_spawn` removed

Thread 0 work queue



# Consider thread executing the following code

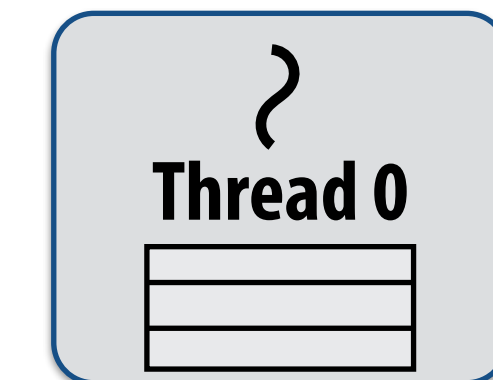
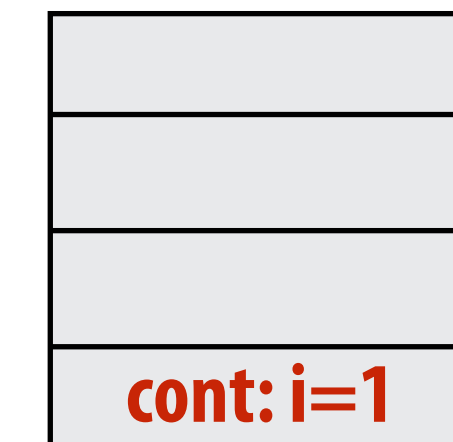
```
for (int i=0; i<N; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync;
```



## ■ Run child first (“continuation stealing”)

- Caller thread only creates one item to steal (continuation that represents all remaining iterations)
- If no stealing occurs, thread continually pops continuation from work queue, enqueues new continuation (with updated value of `i`)
- Order of execution is the same as for program with `spawn` removed.
- Think: depth-first traversal of call graph

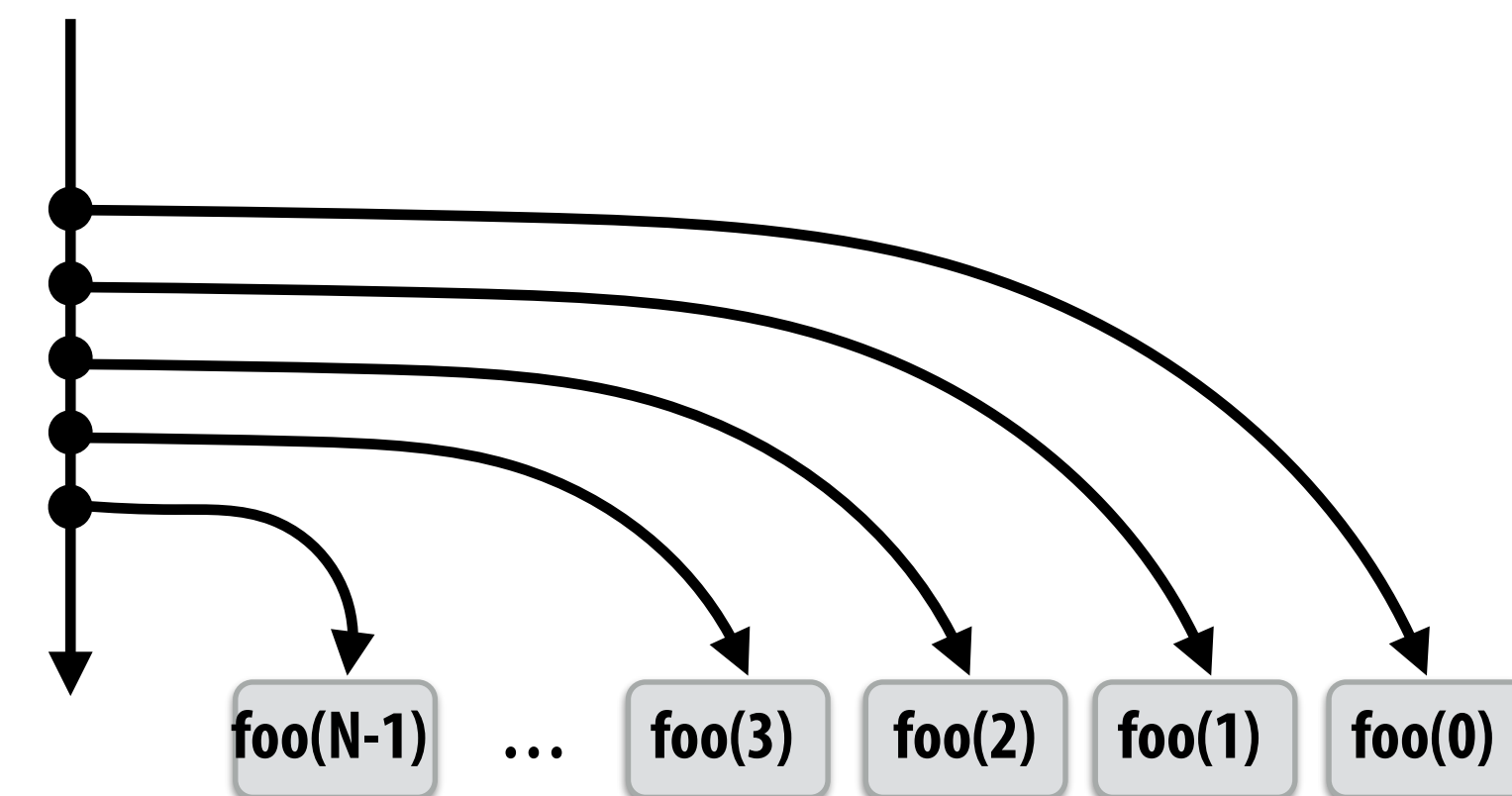
Thread 0 work queue



Executing `foo(0)`...

# Consider thread executing the following code

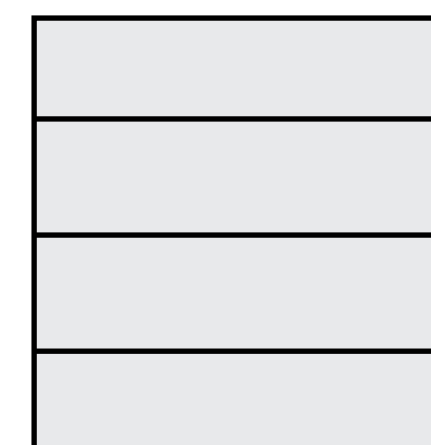
```
for (int i=0; i<N; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync;
```



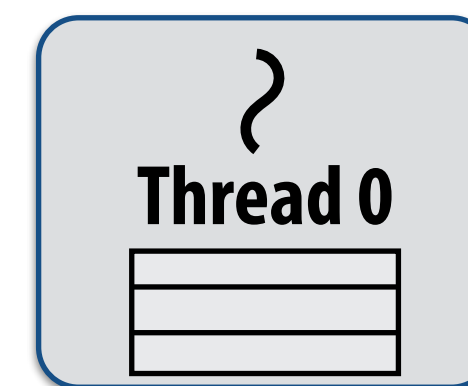
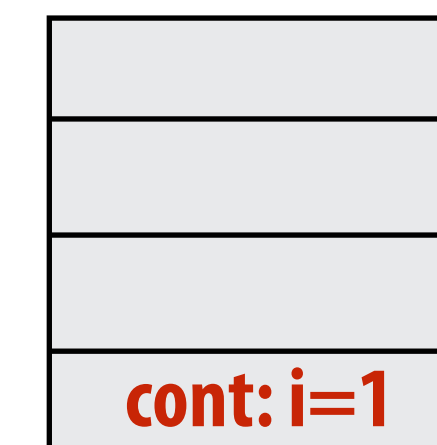
## ■ Run child first (“continuation stealing”)

- Enqueues continuation with `i` advanced by 1
- If continuation is stolen, stealing thread spawns and executes next iteration
- Can prove that work queue storage for system with  $T$  threads is no more than  $T$  times that of stack storage for single threaded execution

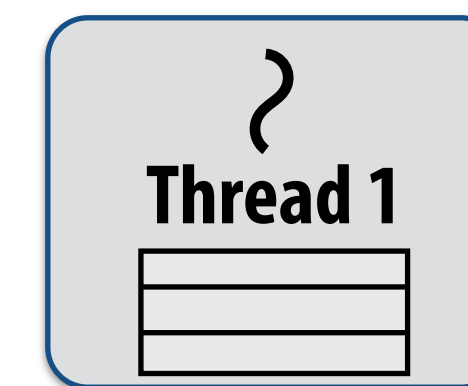
Thread 0 work queue



Thread 1 work queue



Executing `foo(0)`...



Executing `foo(1)`...

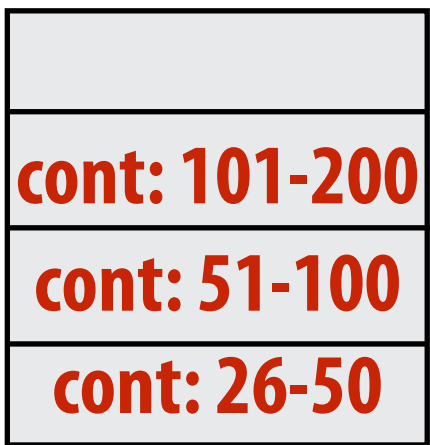
# Scheduling quicksort: assume 200 elements

```
void quick_sort(int* begin, int* end) {
    if (begin >= end - PARALLEL_CUTOFF)
        std::sort(begin, end);
    else {
        int* middle = partition(begin, end);
        cilk_spawn quick_sort(begin, middle);
        quick_sort(middle+1, last);
    }
}
```

**What work in the queue should other threads steal?**  
**(e.g., steal from top or bottom)**



Thread 0 work queue



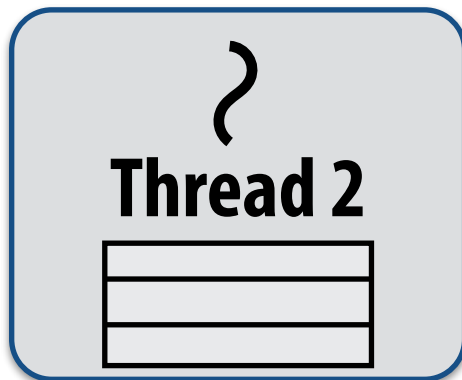
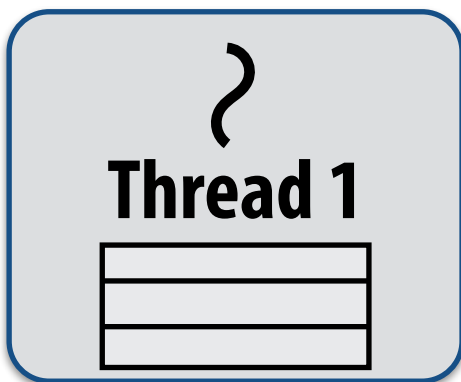
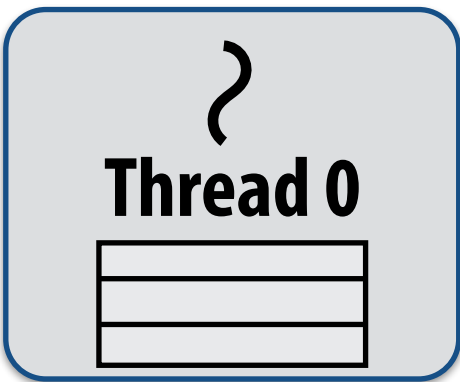
Thread 1 work queue



Thread 2 work queue



...

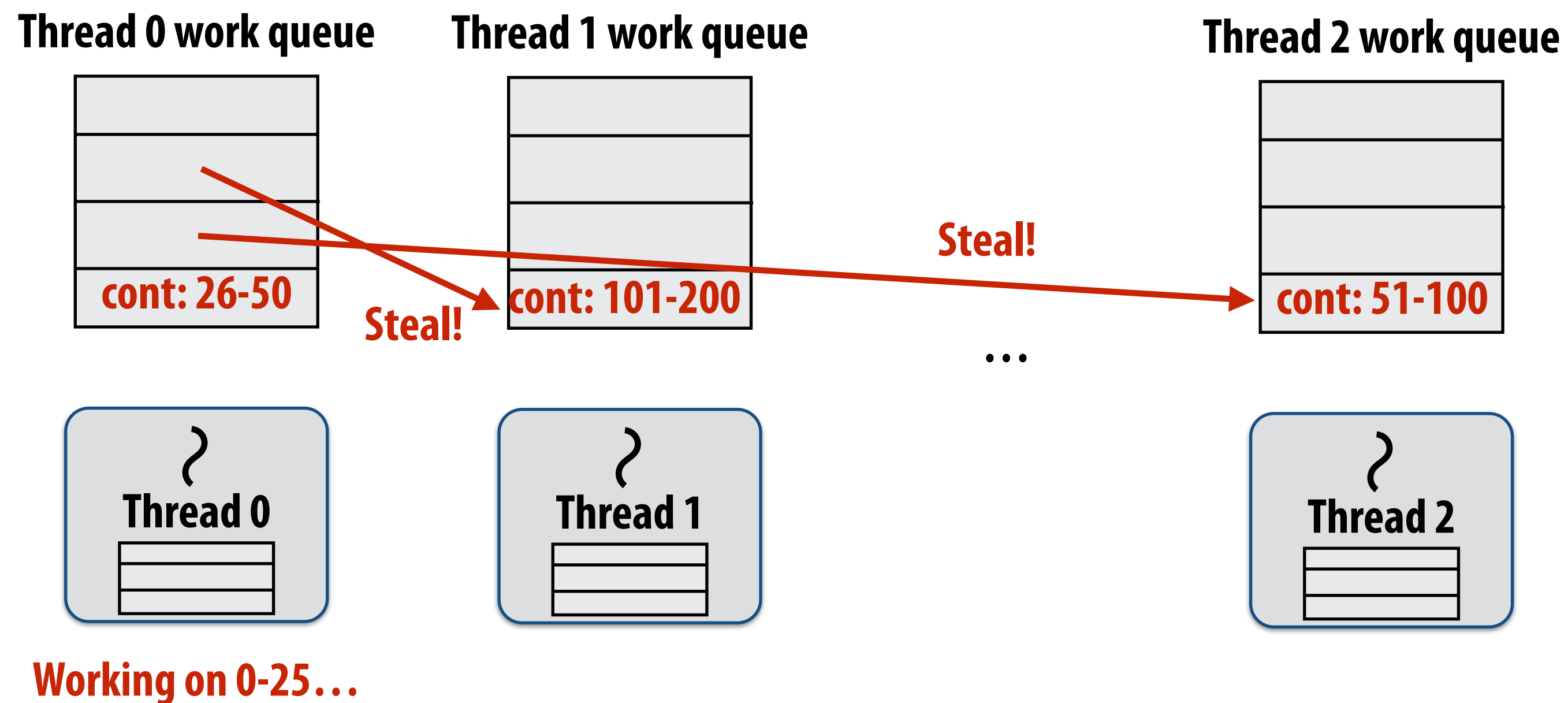


**Working on 0-25...**

# Implementing work stealing: dequeue per worker

Work queue implemented as a dequeue (double ended queue)

- Local thread pushes/pops from the “tail” (bottom)
- Remote threads steal from “head” (top)

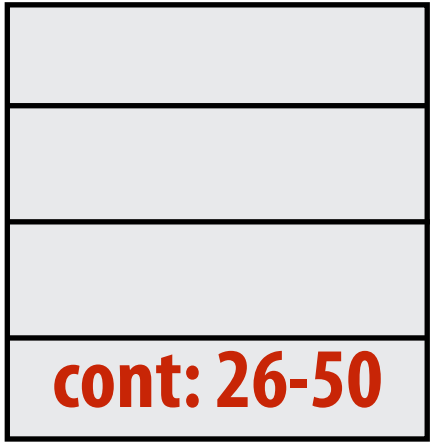


# Implementing work stealing: dequeue per worker

Work queue implemented as a dequeue (double ended queue)

- Local thread pushes/pops from the “tail” (bottom)
- Remote threads steal from “head” (top)

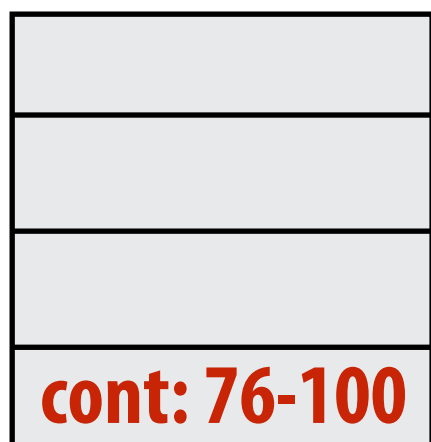
Thread 0 work queue



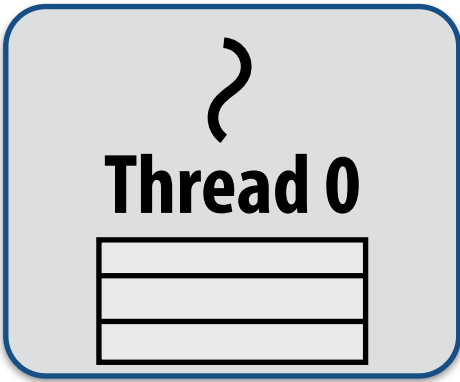
Thread 1 work queue



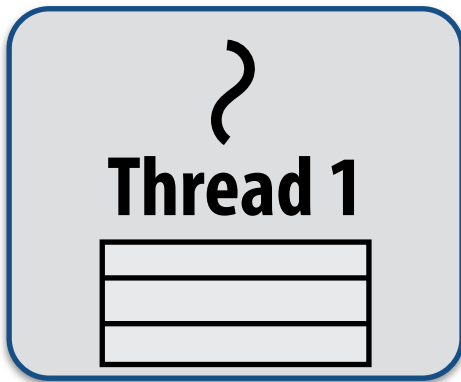
Thread 2 work queue



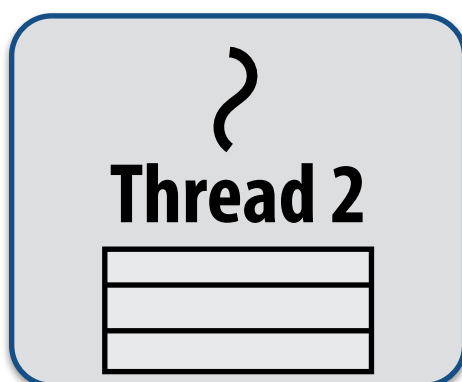
...



Working on 0-25...



Working on 101-150...



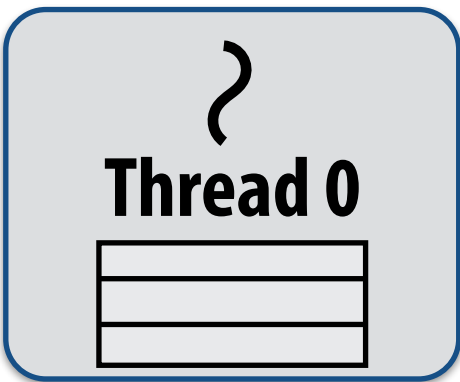
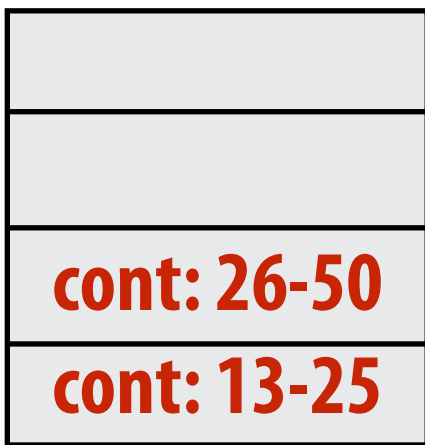
Working on 51-75...

# Implementing work stealing: dequeue per worker

## Work queue implemented as a dequeue (double ended queue)

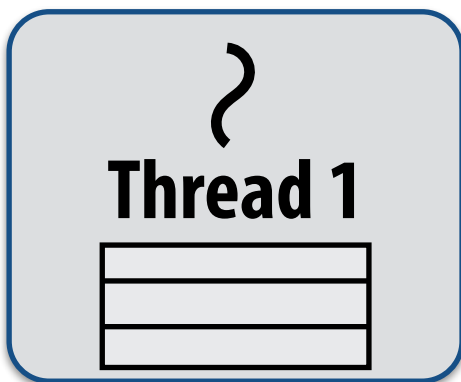
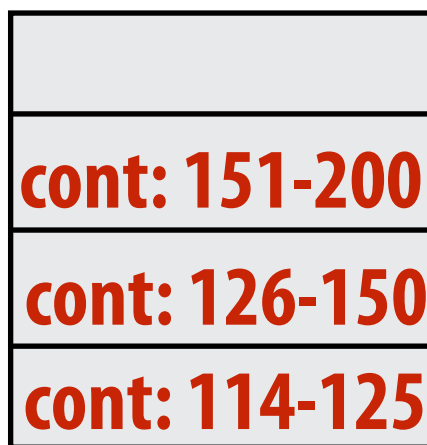
- Local thread pushes/pops from the “tail” (bottom)
- Remote threads steal from “head” (top)

Thread 0 work queue



Working on 0-12...

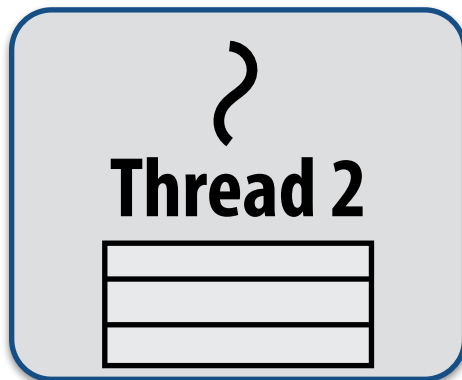
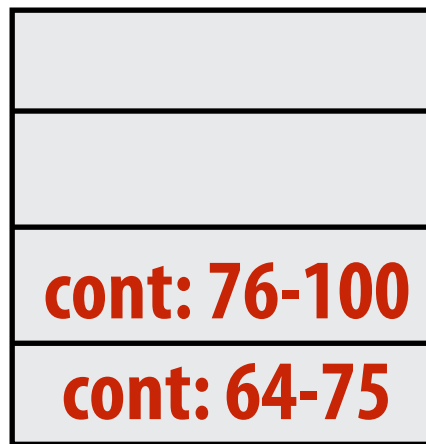
Thread 1 work queue



Working on 101-113...

...

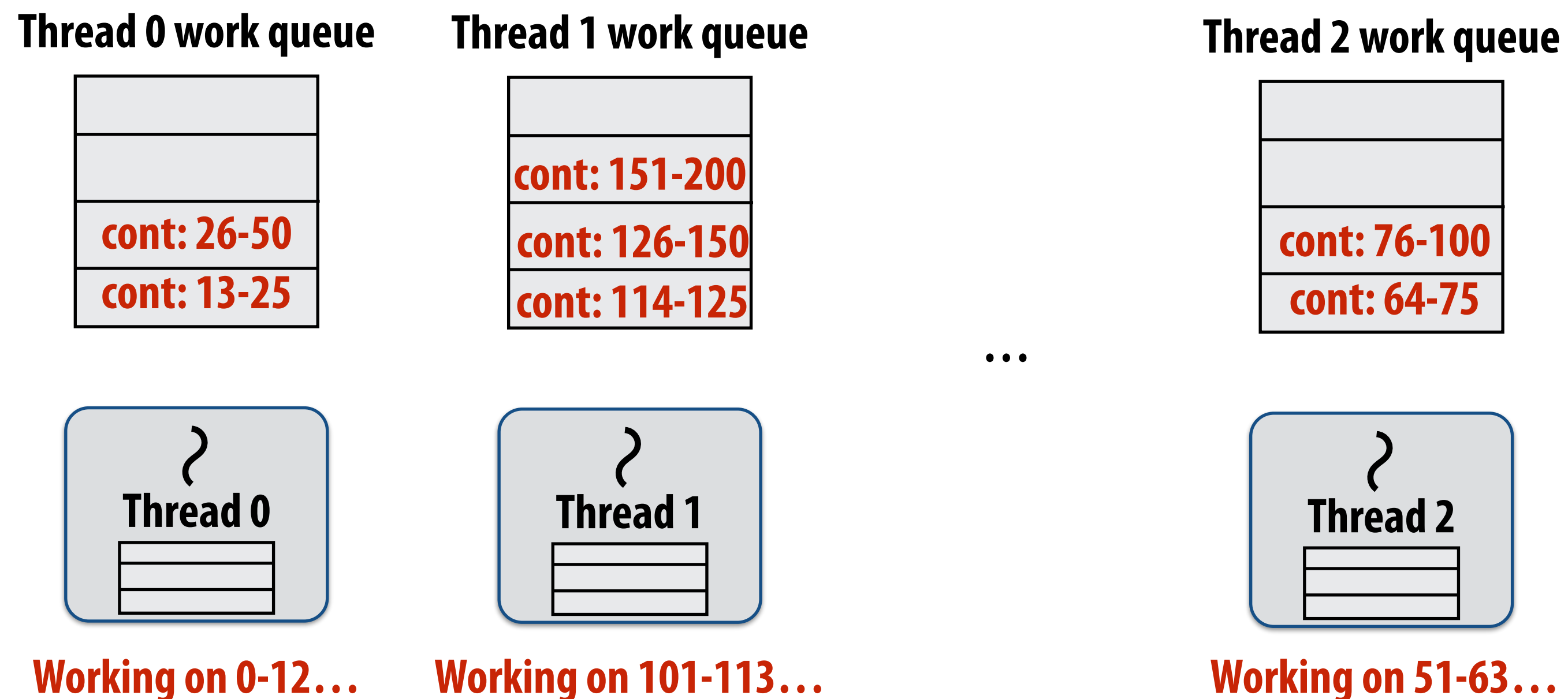
Thread 2 work queue



Working on 51-63...

# Implementing work stealing: choice of victim

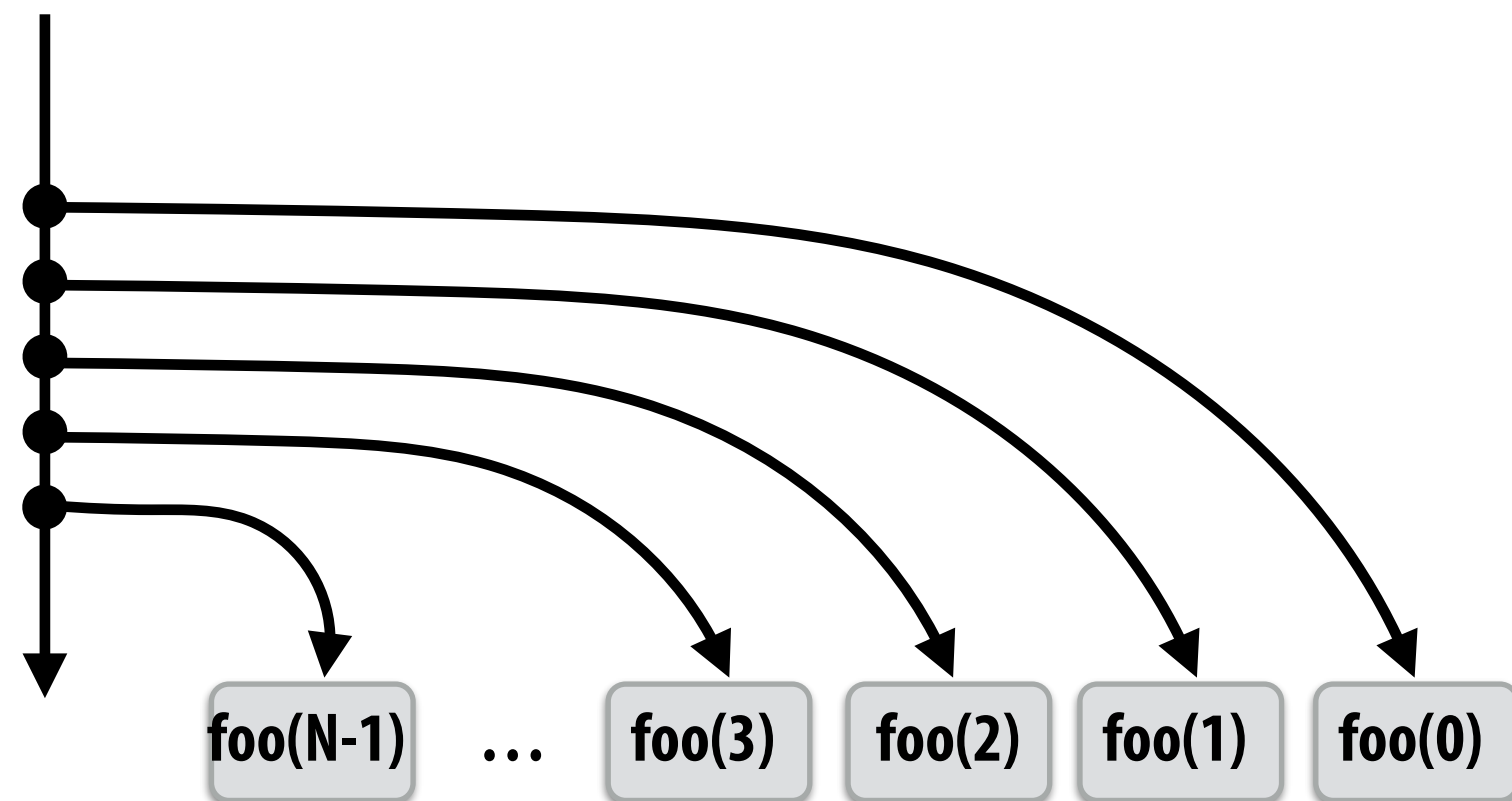
- Idle threads randomly choose a thread to attempt to steal from
- Steal work from top of dequeue:
  - Steals largest amount of work (reduce number of steals)
  - Maximum locality in work each thread performs (when combined with run child first scheme)
  - Stealing thread and local thread do not contend for same elements of dequeue (efficient lock-free implementations of dequeue exist)





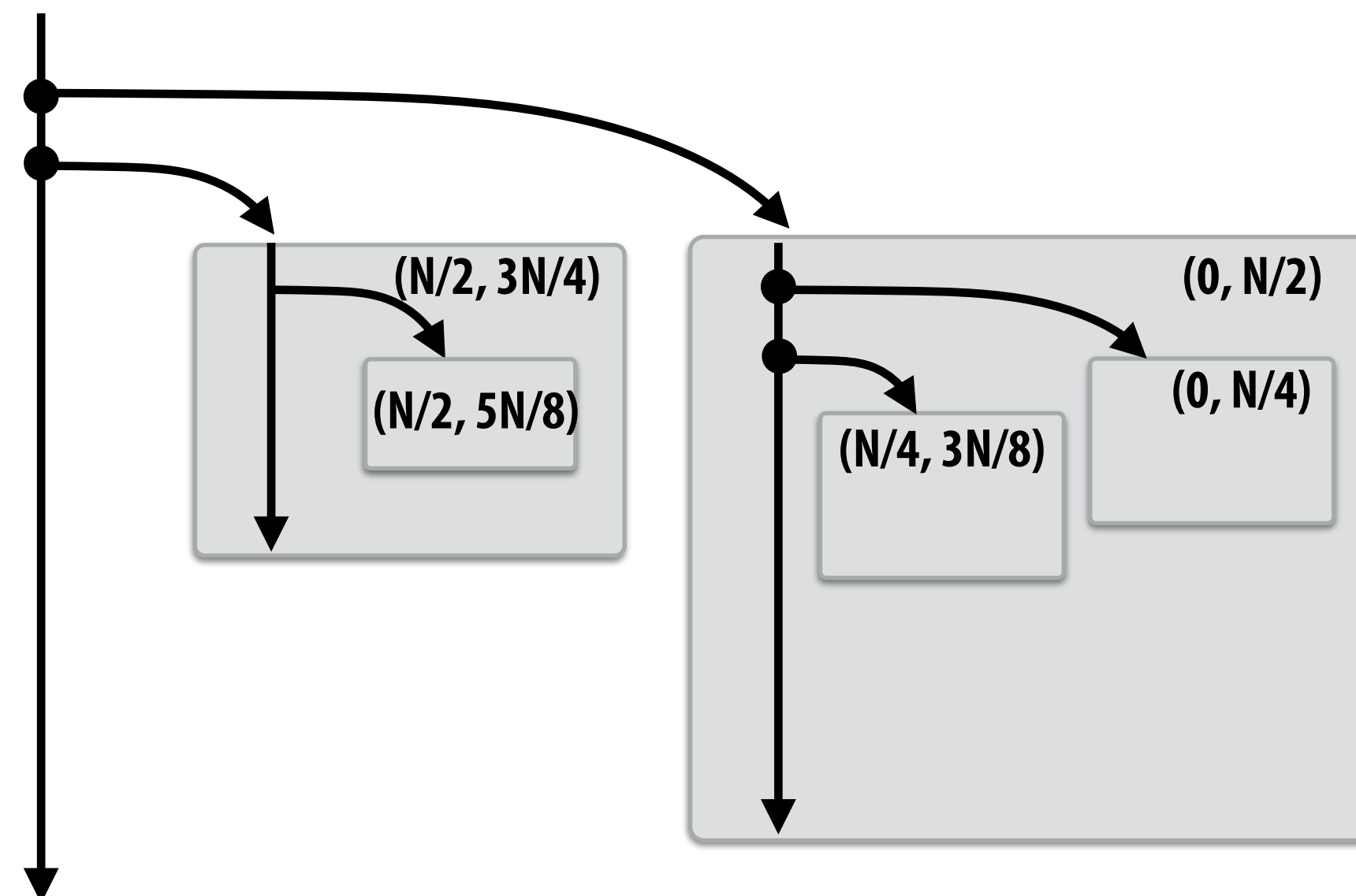
# Child-first work stealing scheduler anticipates divide-and-conquer parallelism

```
for (int i=0; i<N; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync;
```



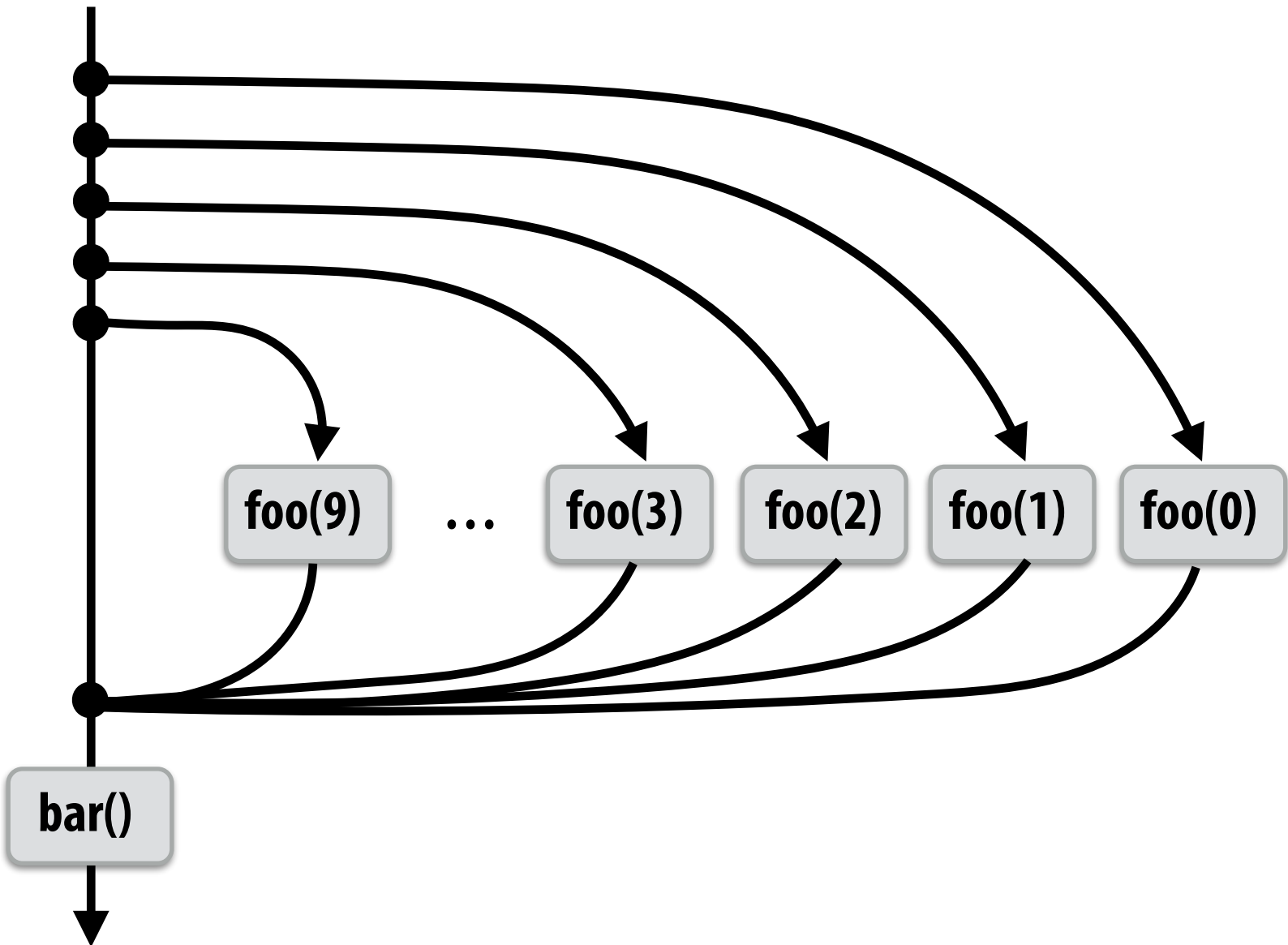
Code at right generates work in parallel, (code at left does not), so it more quickly fills up parallel machine

```
void recursive_for(int start, int end) {  
    while (start <= end - GRANULARITY) {  
        int mid = (end - start) / 2;  
        cilk_spawn recursive_for(start, mid);  
        start = mid;  
    }  
  
    for (int i=start; i<end; i++)  
        foo(i);  
}  
  
recursive_for(0, N);
```

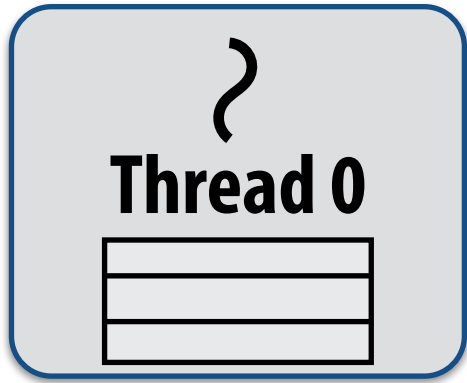
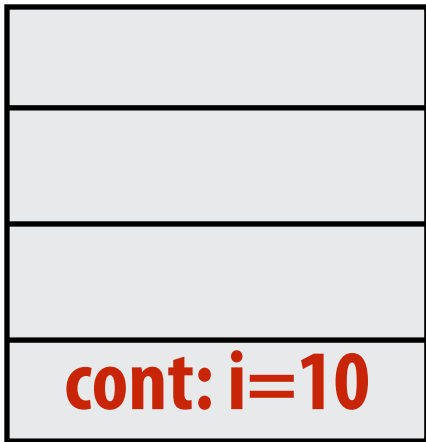


# Implementing sync

```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync;  
bar();
```

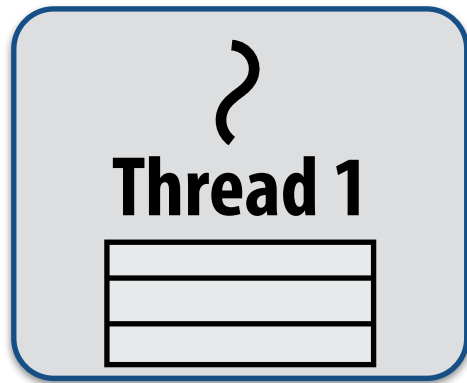


Thread 0 work queue



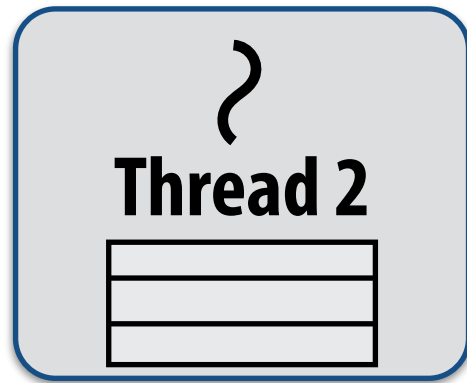
Working on foo(9)...

Thread 1 work queue



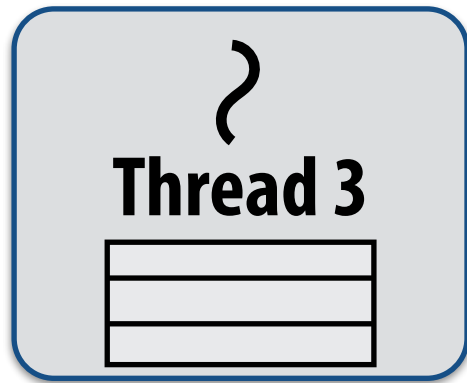
Working on foo(7)...

Thread 2 work queue



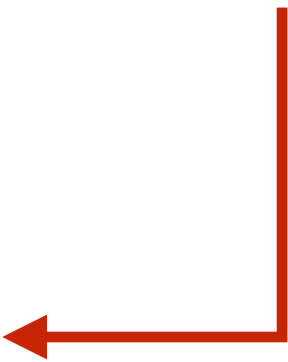
Working on foo(8)...

Thread 3 work queue



Working on foo(6)...

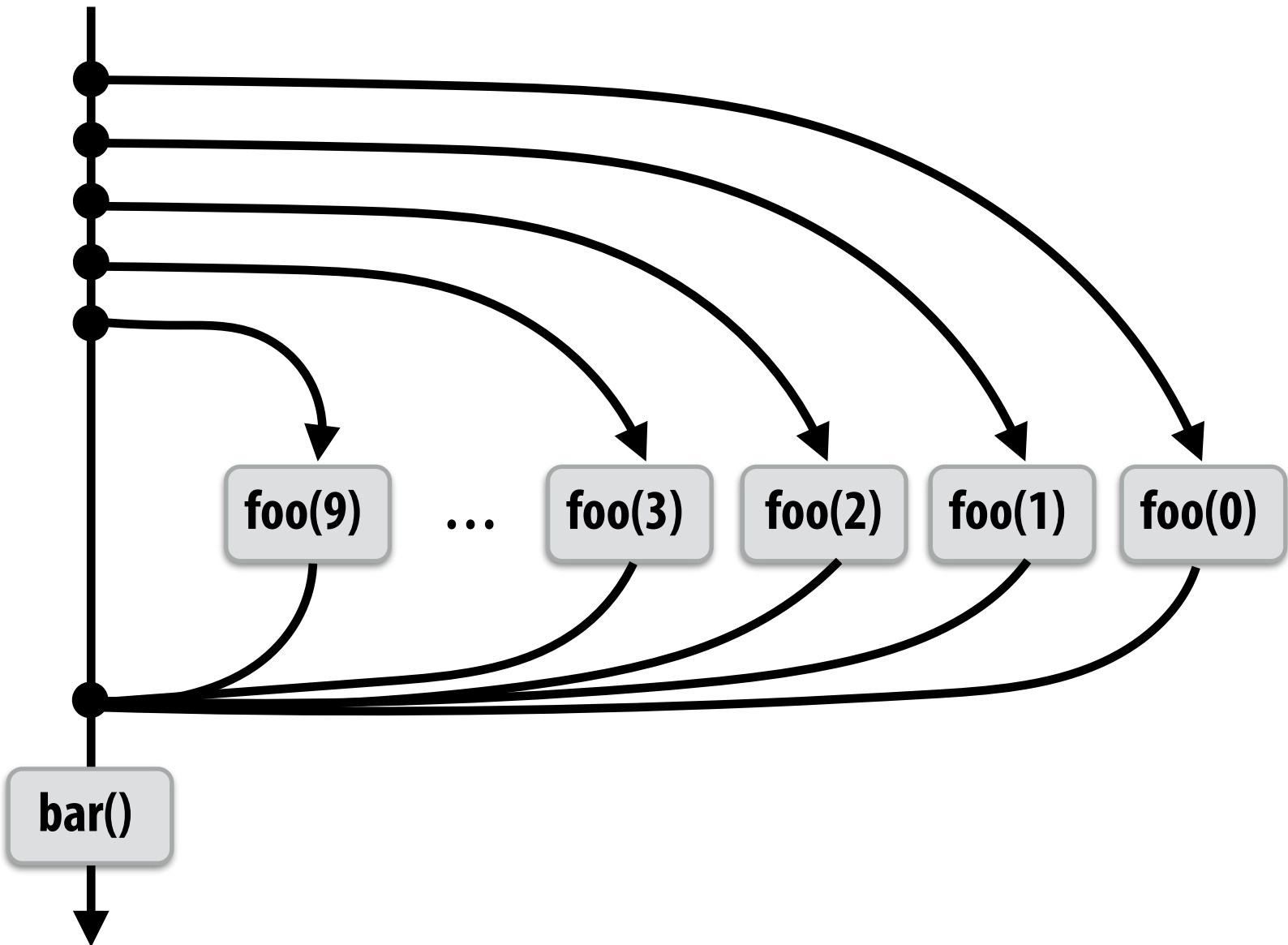
State of worker threads  
when all work from loop  
is nearly complete



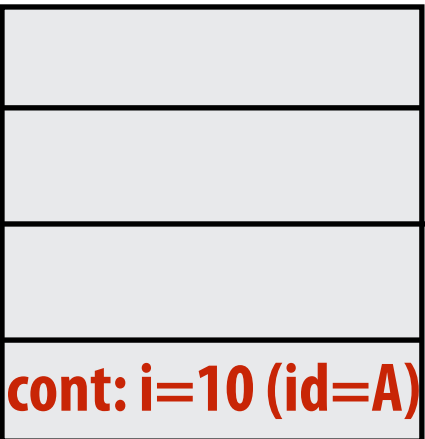
# Implementing sync: no stealing case

**block (id: A)**

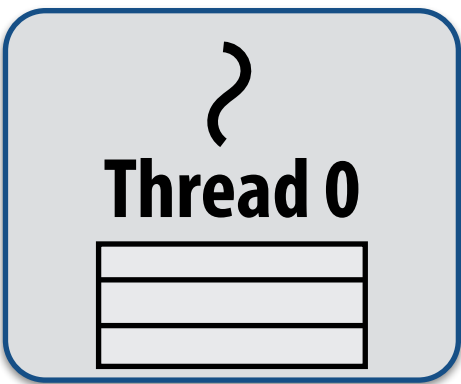
```
[ for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();  
]
```



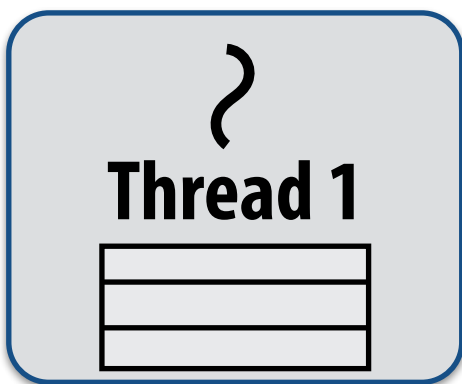
Thread 0 work queue



Thread 1 work queue



**Working on foo(9), id=A...**



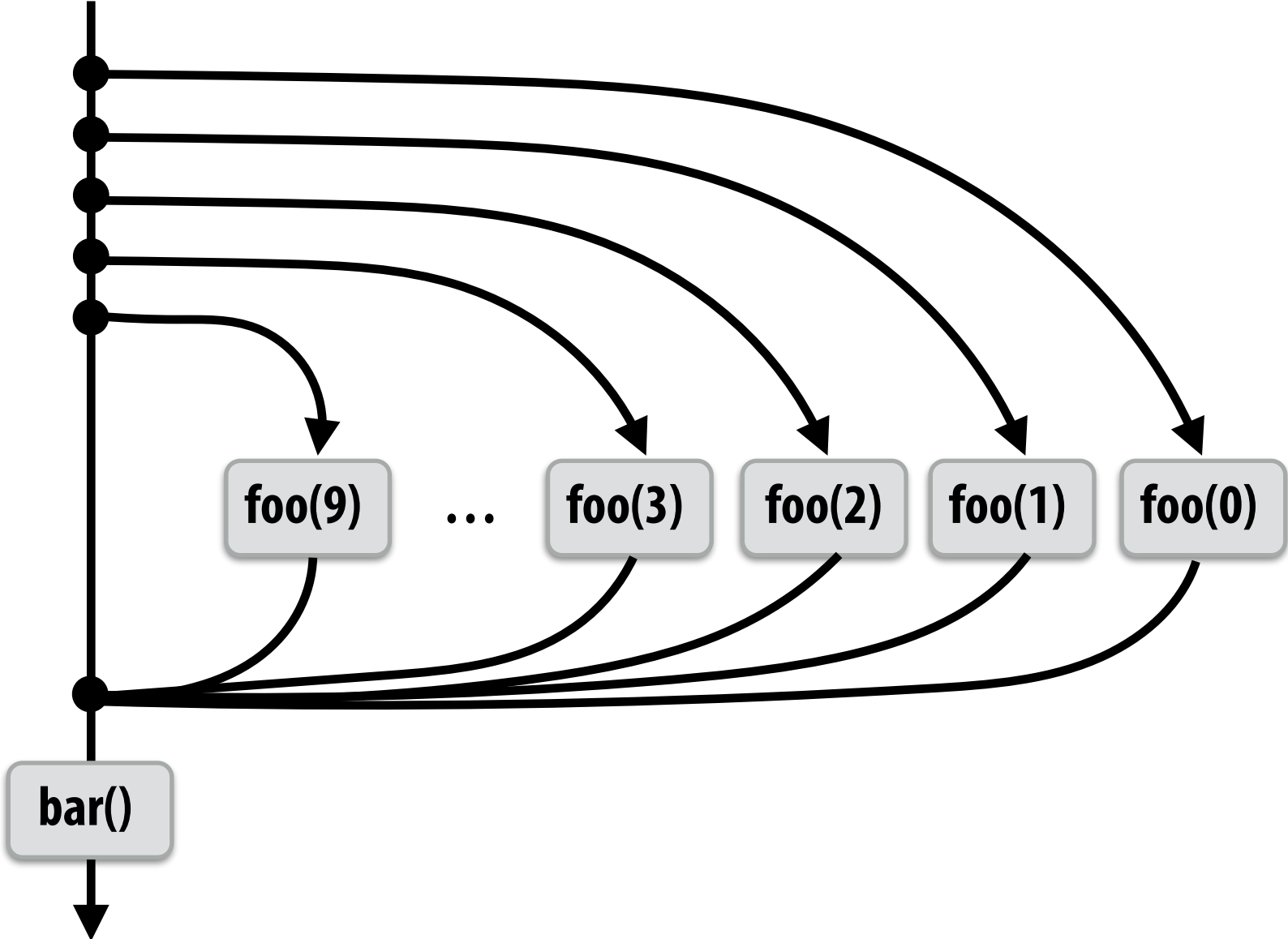
**If no work has been stolen by other threads, then there's nothing to do at the sync point.**

**cilk\_sync is a no-op.**

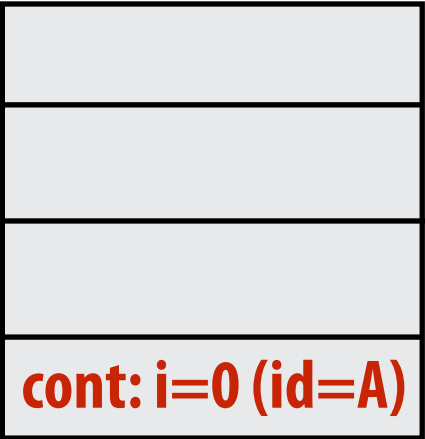
# Implementing sync: stealing case

**block (id: A)**

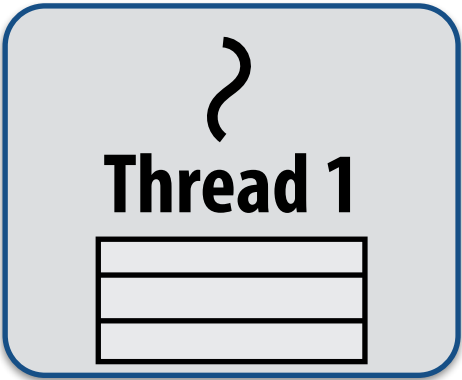
```
[ for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();  
]
```



Thread 0 work queue



Thread 1 work queue



**Working on foo(0), id=A...**

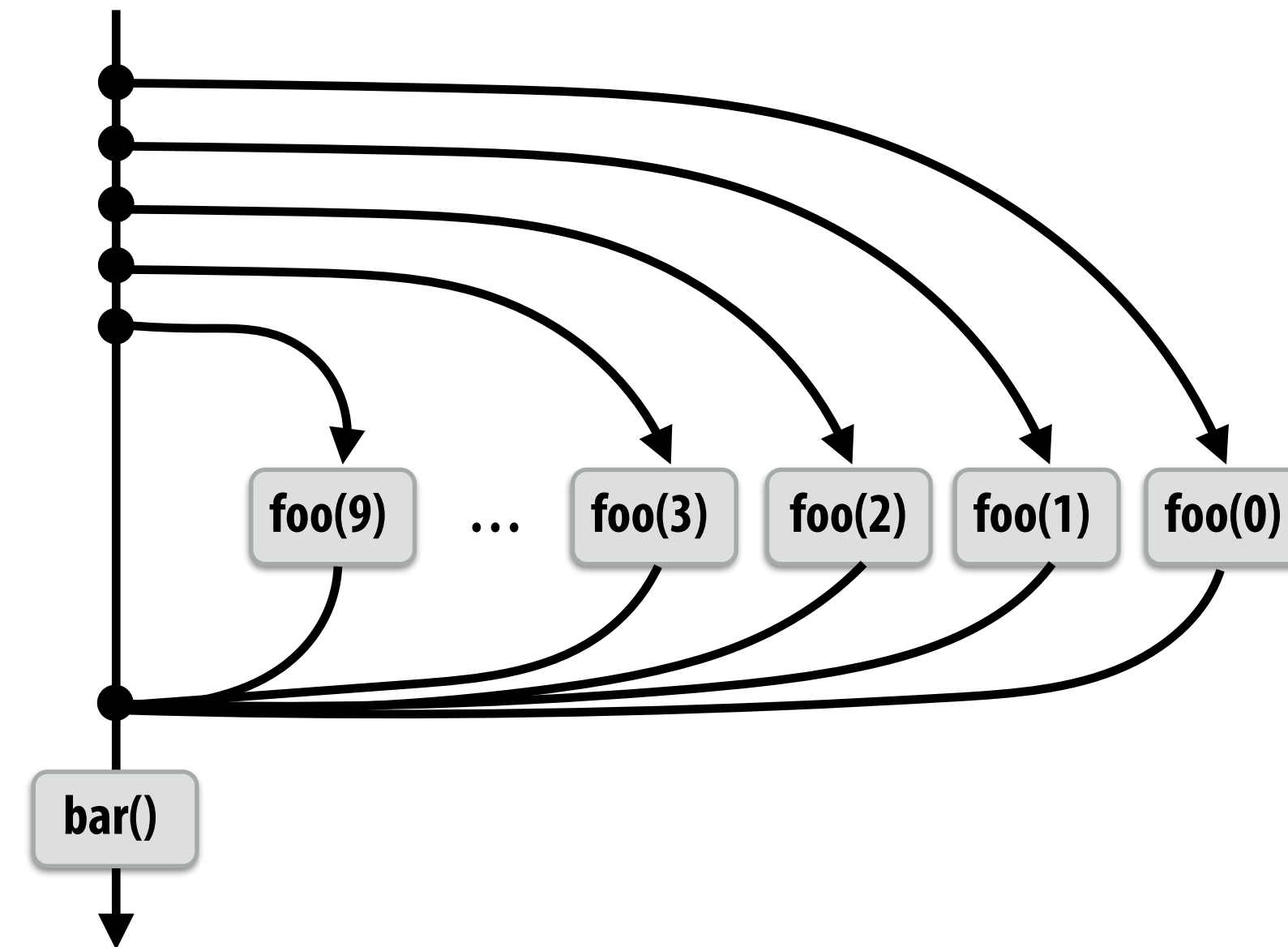
# Implementing sync: stealing case

**block (id: A)**

```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}
```

**cilk\_sync; Sync for all calls spawned within block A**

```
bar();
```

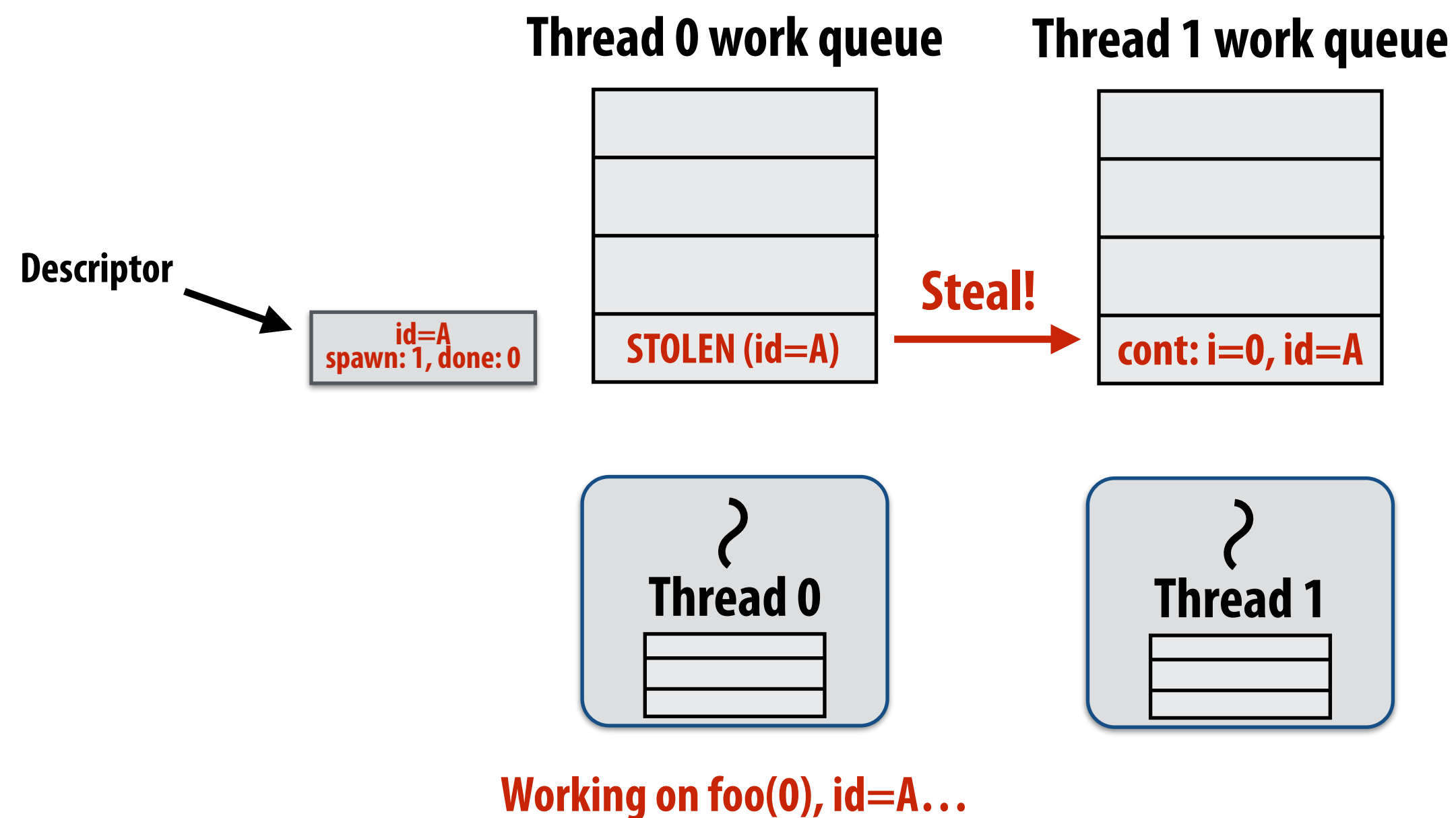


**Idle thread 1 steals from busy thread 0**

**Note: descriptor for block A created**

The descriptor tracks the number of outstanding spawns for the block, and the number of those spawns that have completed.

The 1 spawn tracked by the descriptor corresponds to foo(0) being run by thread 0. (Since the continuation is now owned by thread 1 after the steal.)



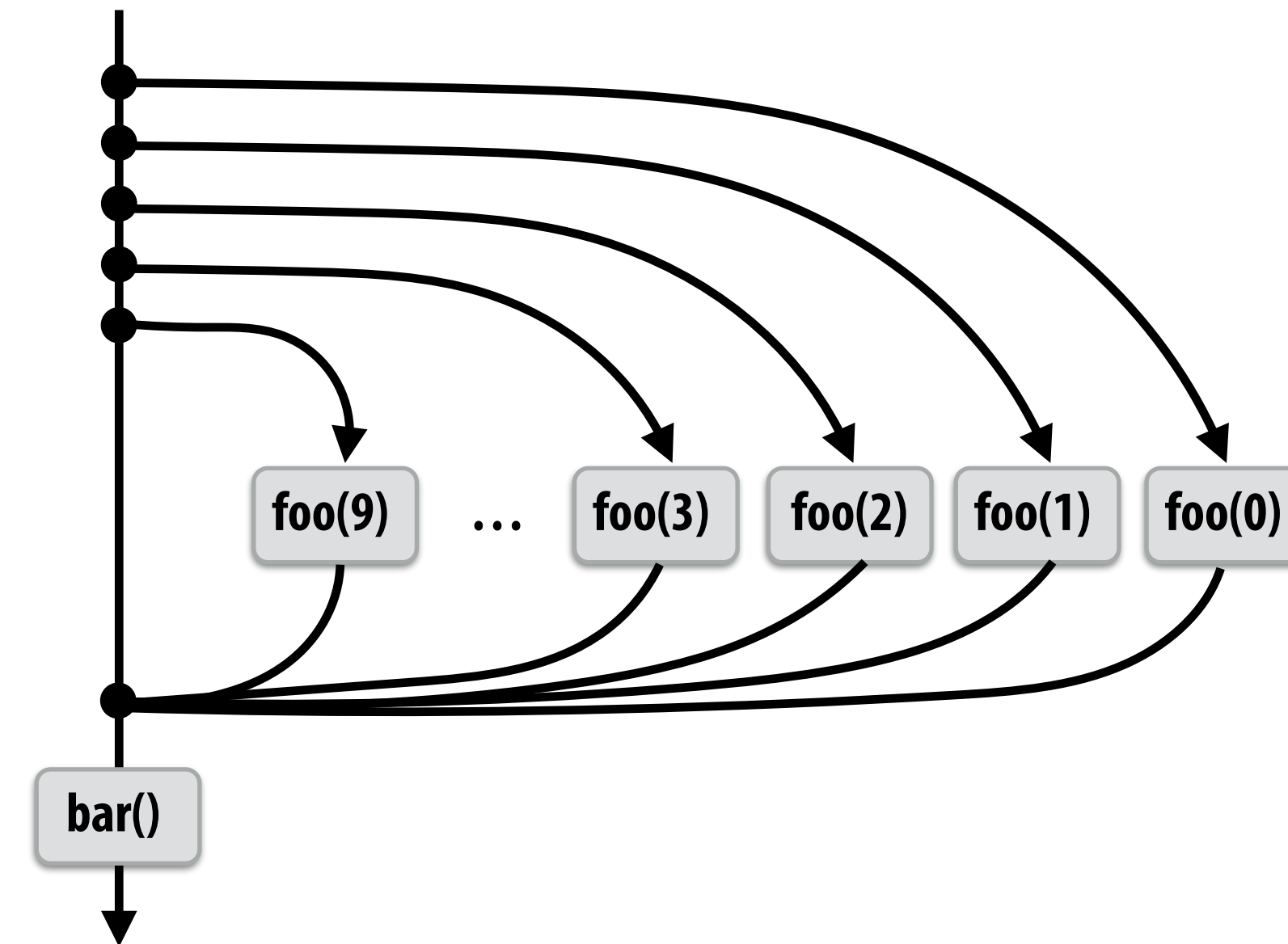
# Implementing sync: stealing case

**block (id: A)**

```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}
```

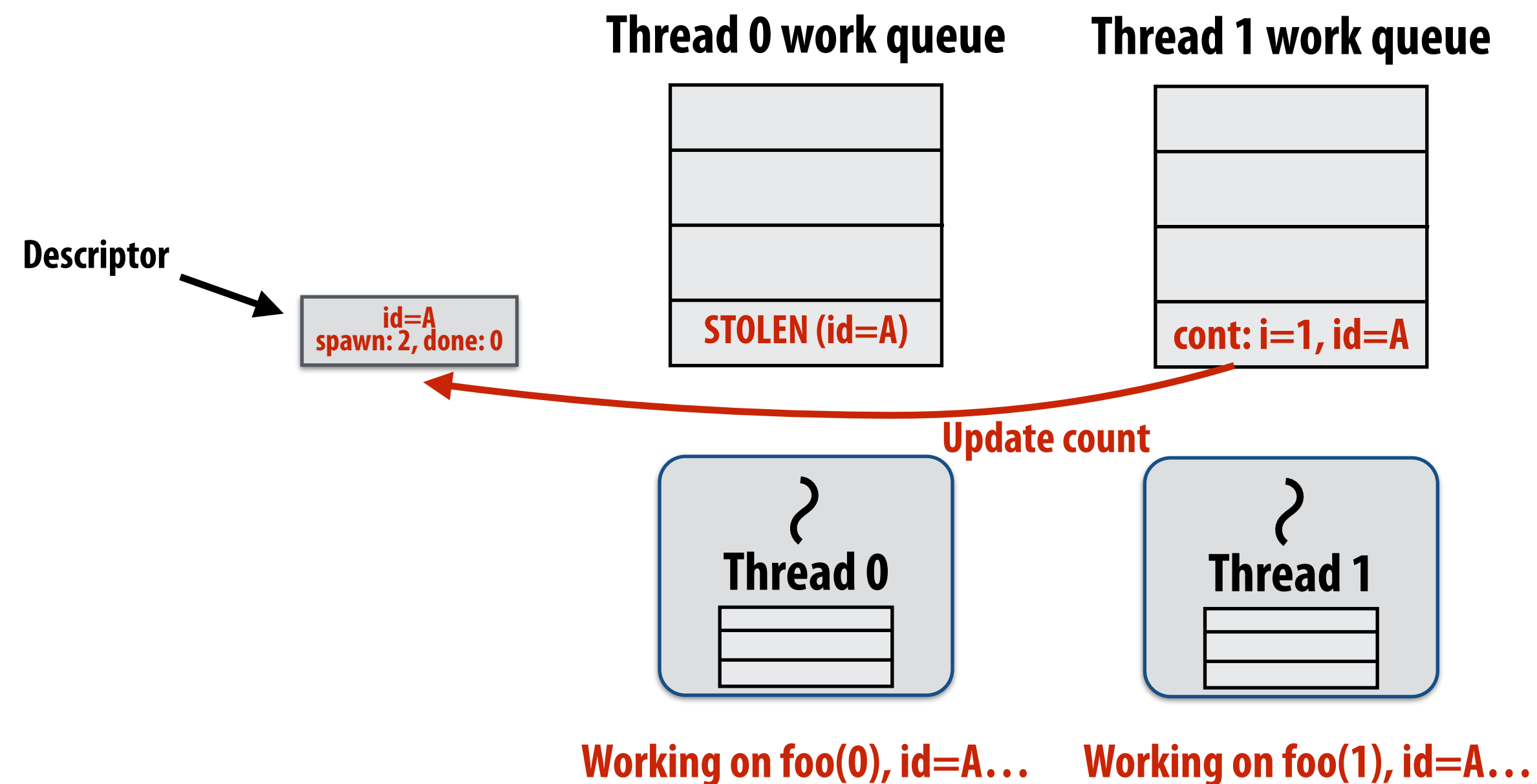
**cilk\_sync; Sync for all calls spawned within block A**

```
bar();
```



**Thread 1 is now running foo(1)**

**Note: spawn count is now 2**



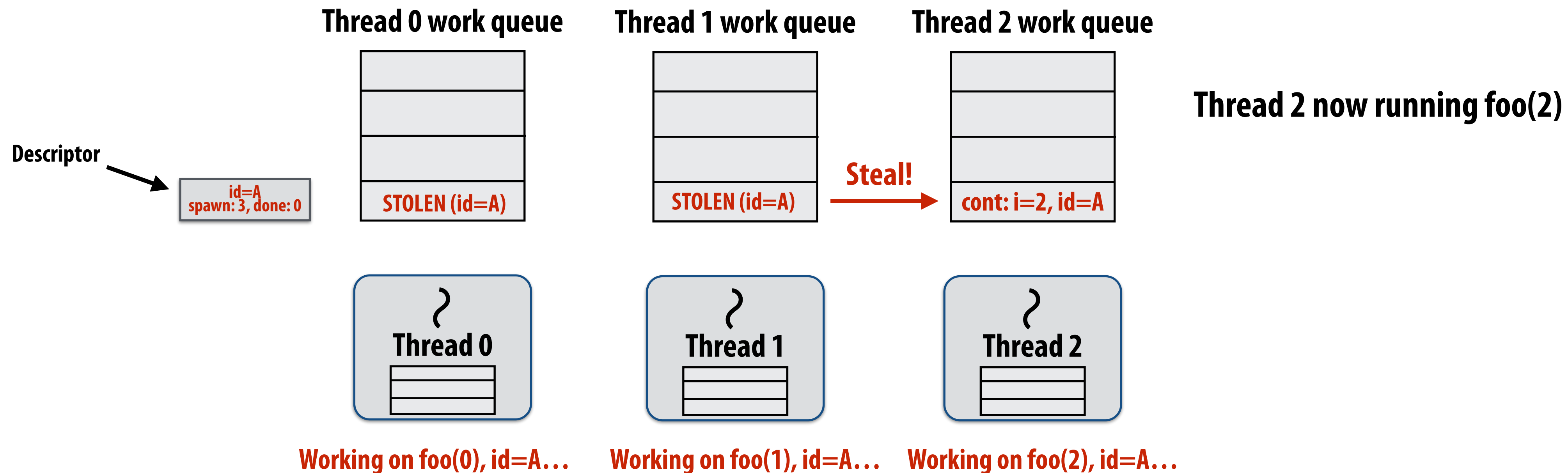
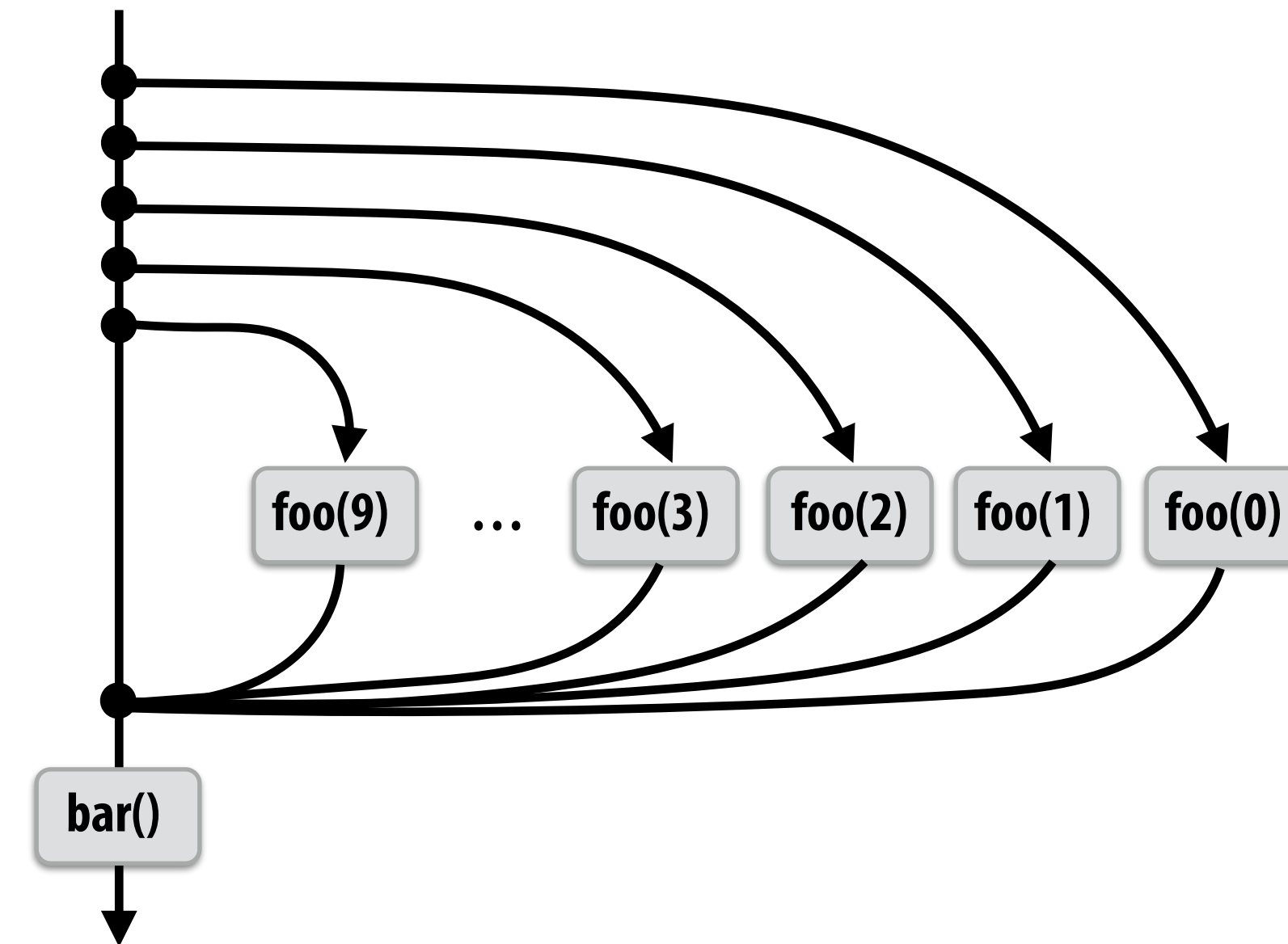
# Implementing sync: stealing case

**block (id: A)**

```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}
```

**cilk\_sync;** Sync for all calls spawned within block A

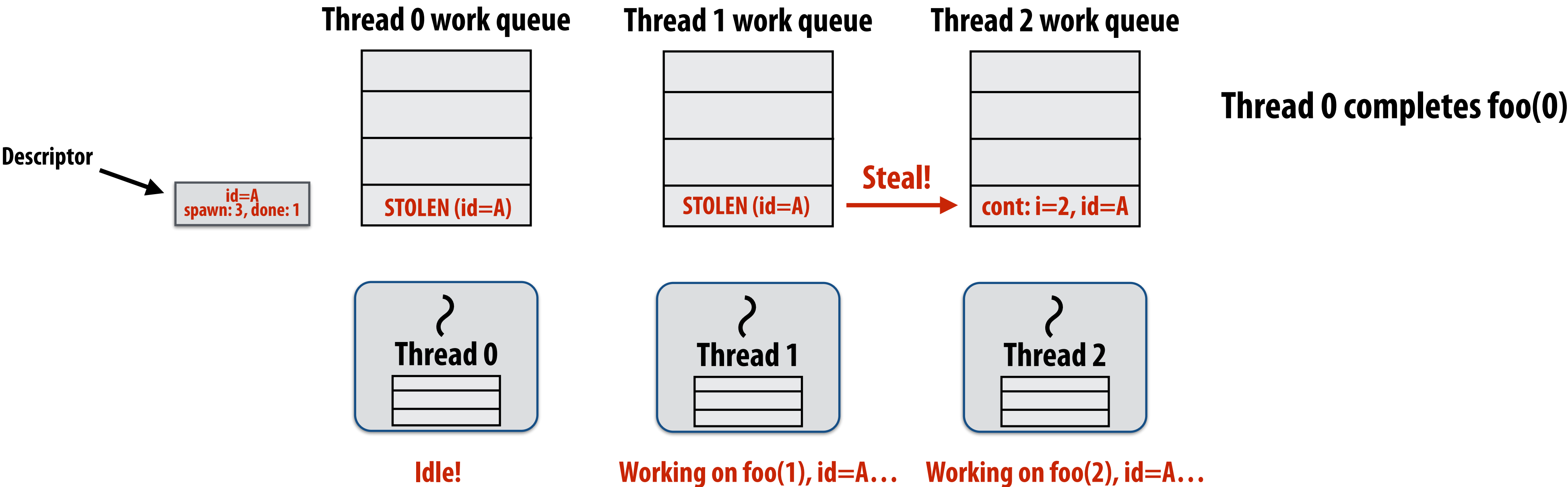
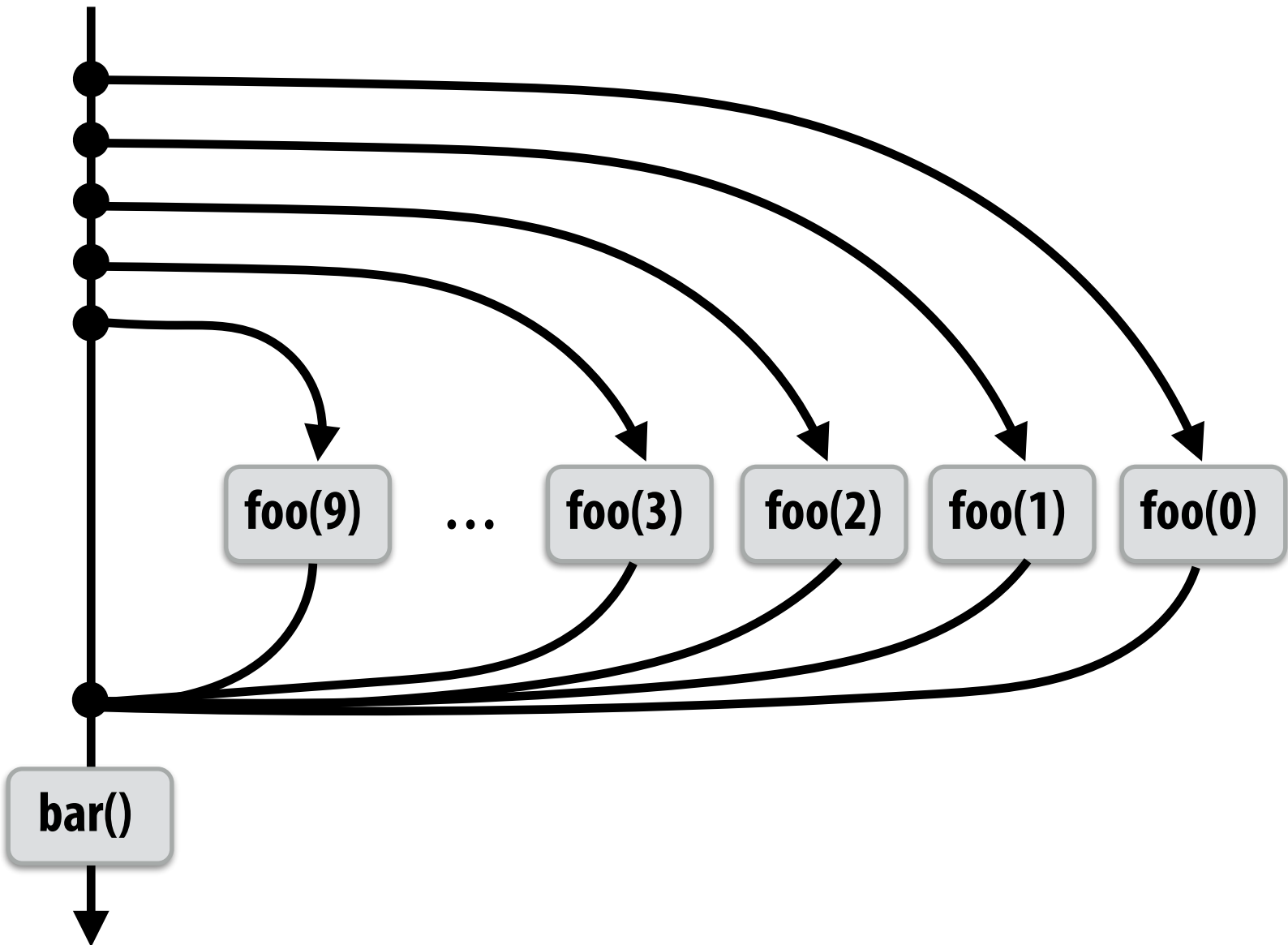
```
bar();
```



# Implementing sync: stealing case

**block (id: A)**

```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```

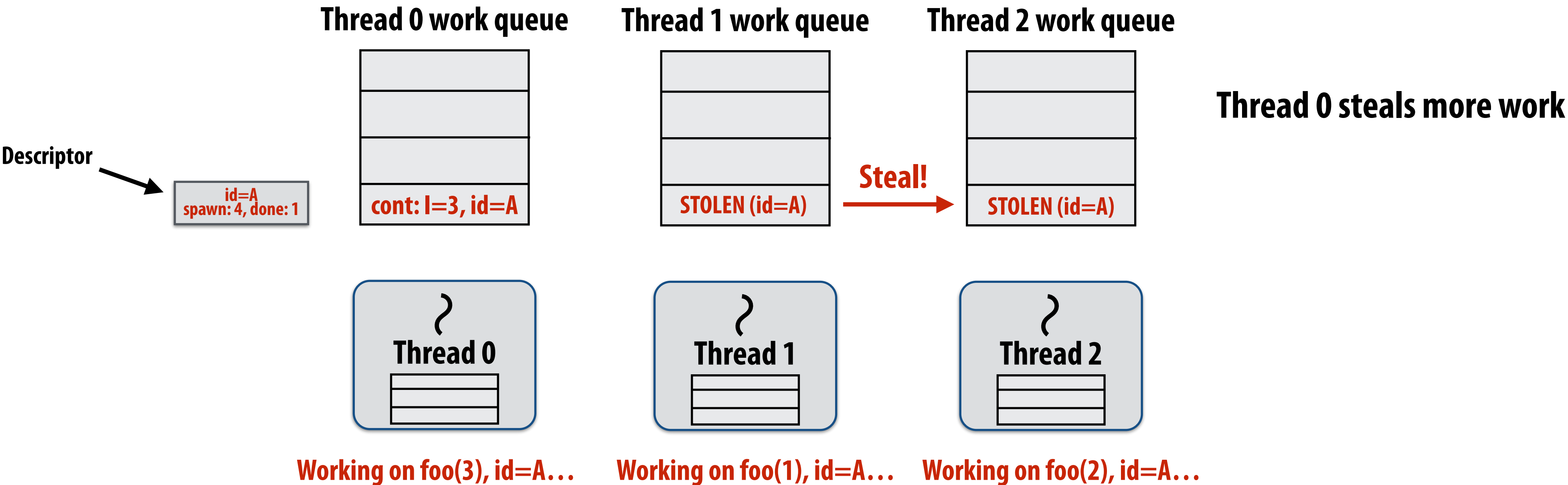
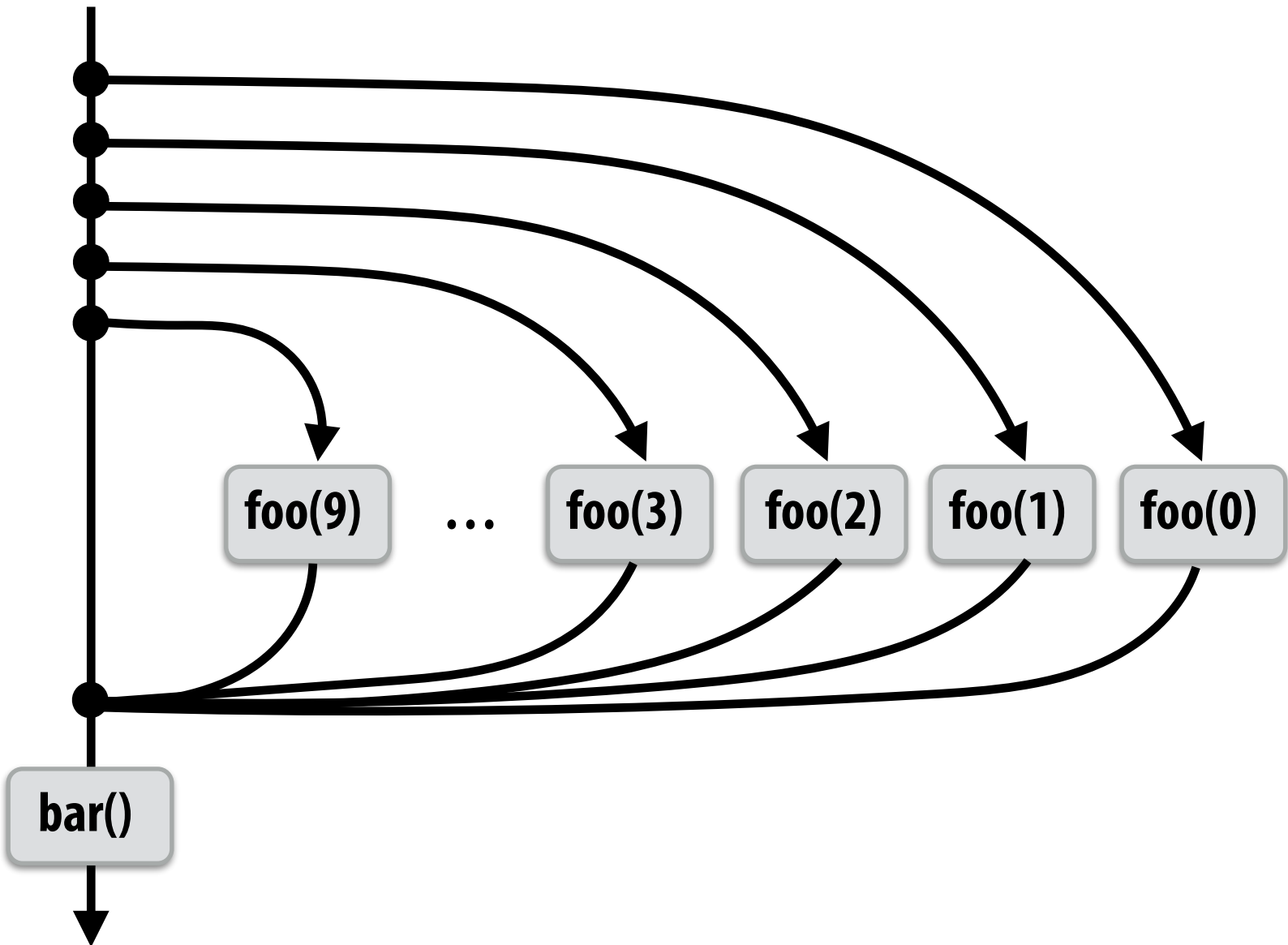




# Implementing sync: stealing case

**block (id: A)**

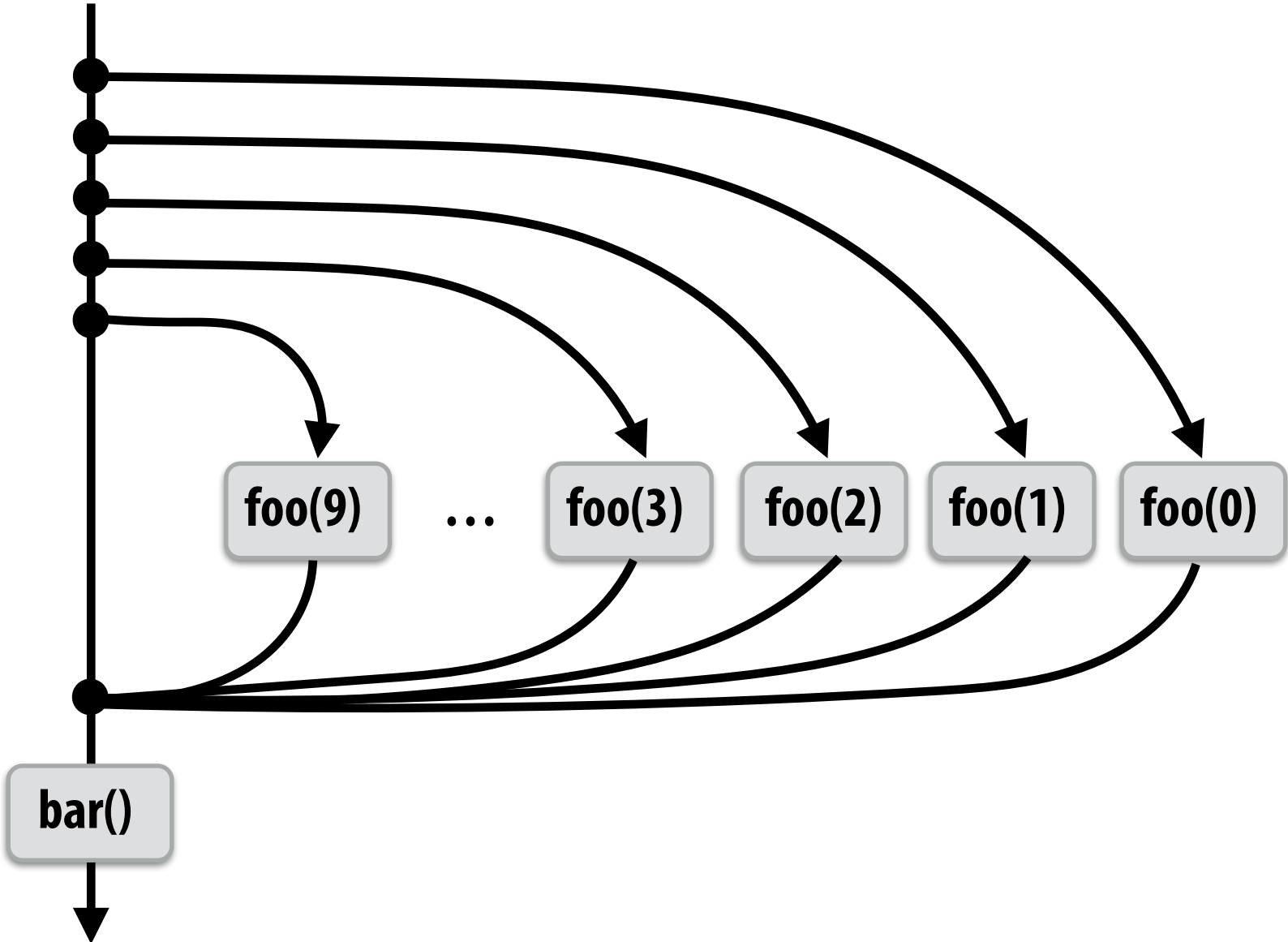
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



# Implementing sync: stealing case

**block (id: A)**

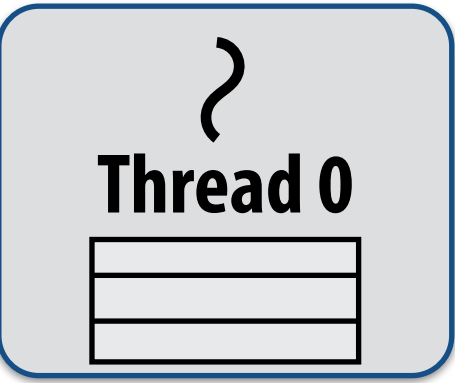
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



Descriptor

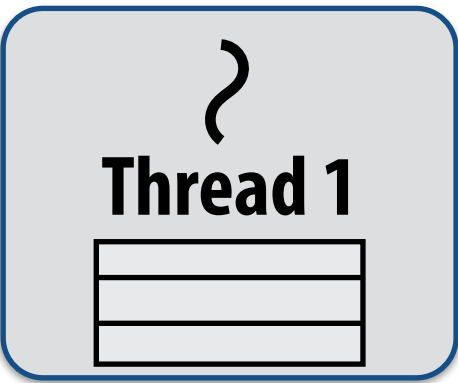
id=A  
spawn: 10, done: 9

Thread 0 work queue



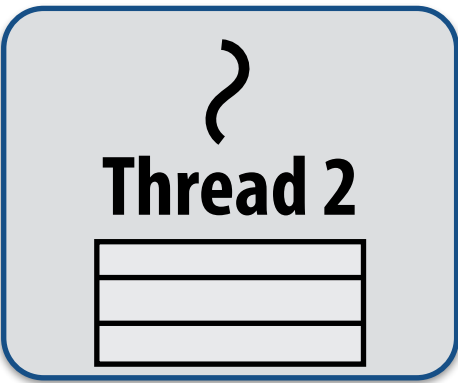
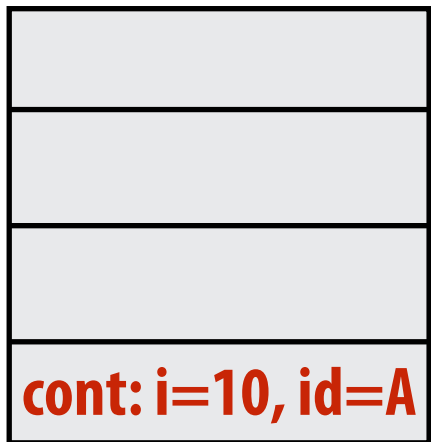
Idle!

Thread 1 work queue



Idle!

Thread 2 work queue



Working on foo(9), id=A...

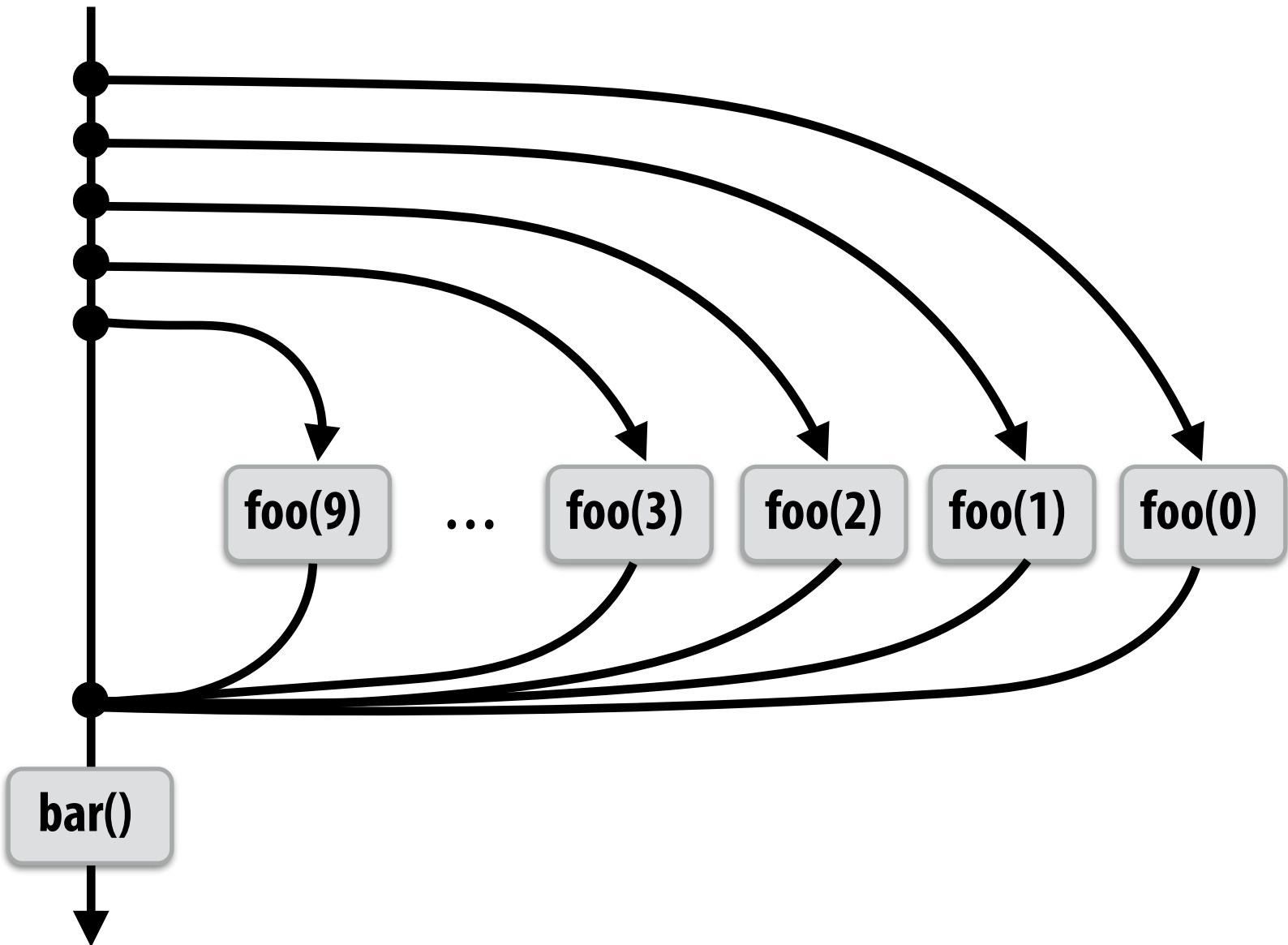
Computation nearing end...

Only foo(9) remains to be completed.

# Implementing sync: stealing case

**block (id: A)**

```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



Descriptor

**id=A**  
**spawn: 10, done: 10**

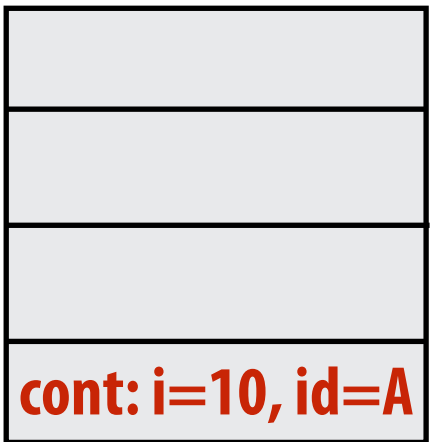
Thread 0 work queue



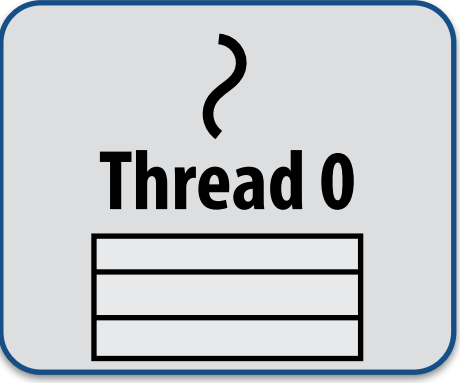
Thread 1 work queue



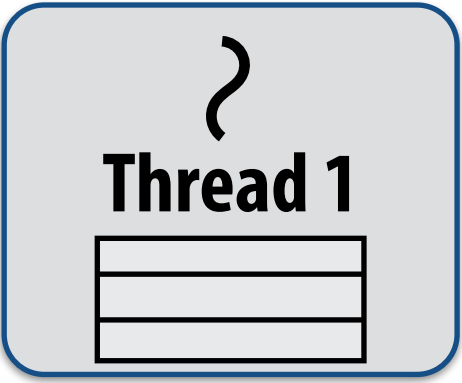
Thread 2 work queue



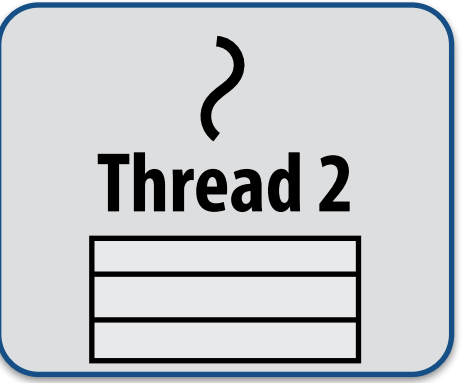
**Last spawn completes.**



**Idle!**



**Idle!**



**Idle!**

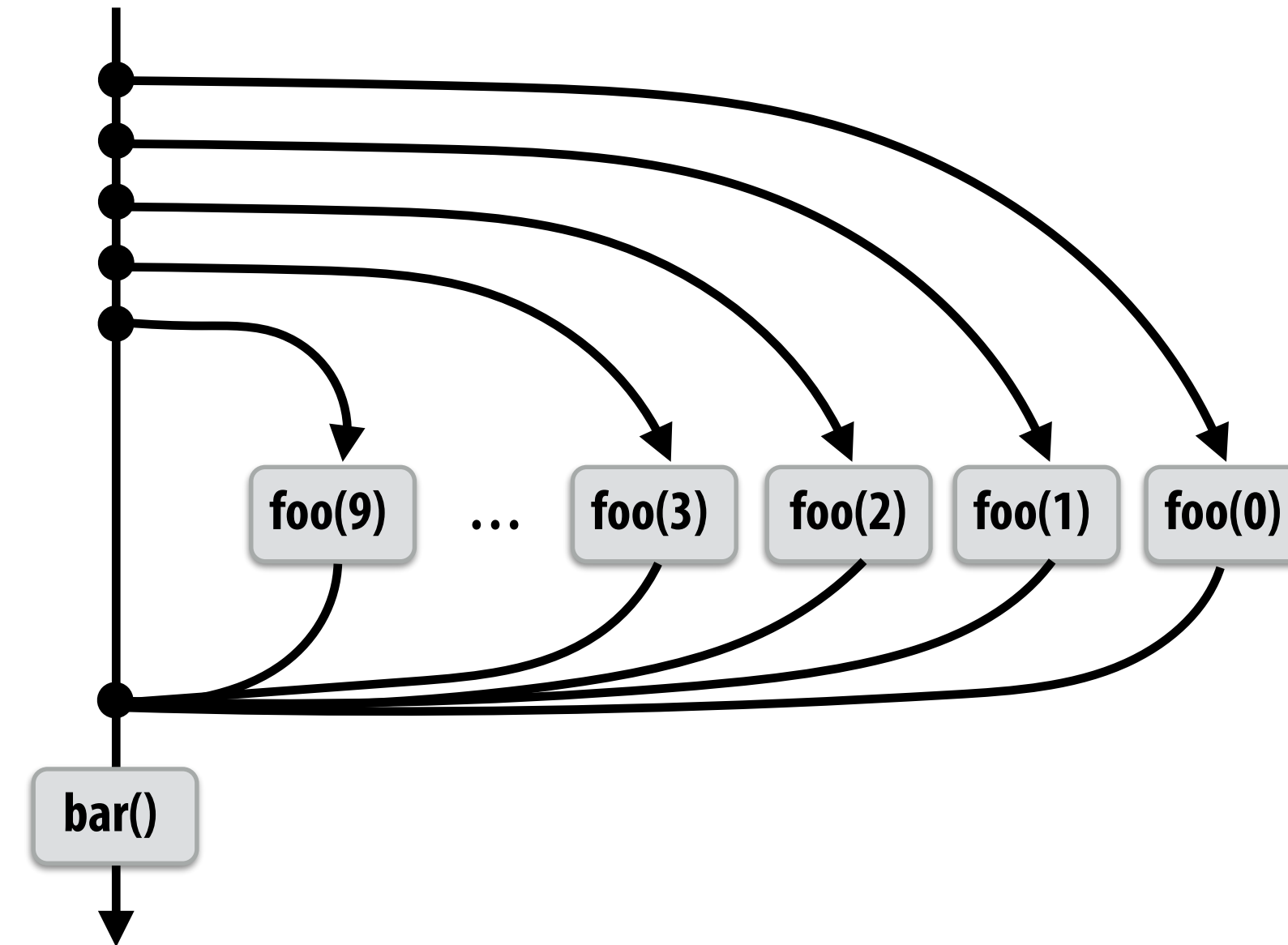
# Implementing sync: stealing case

**block (id: A)**

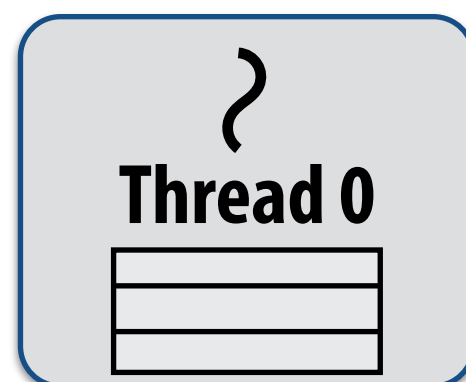
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}
```

**cilk\_sync; Sync for all calls spawned within block A**

**bar();**

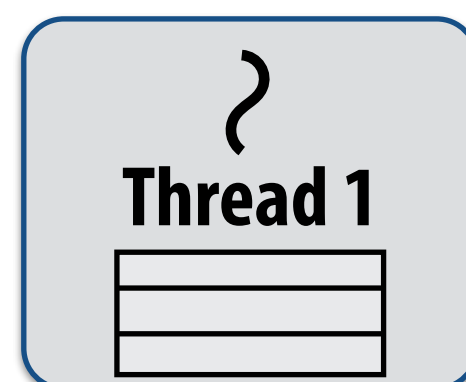
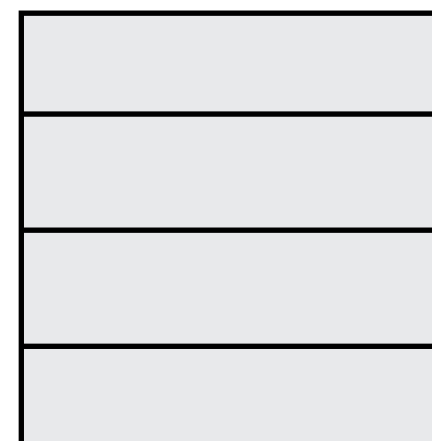


**Thread 0 work queue**



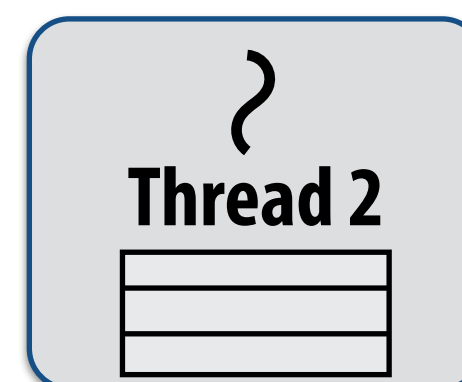
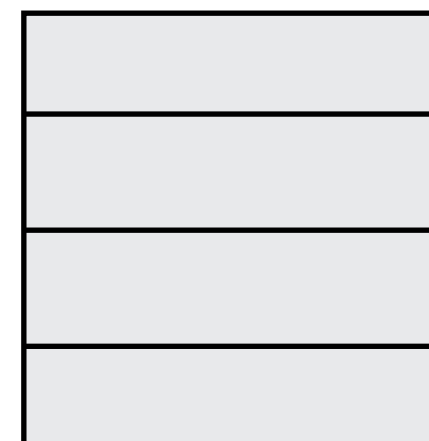
**Idle!**

**Thread 1 work queue**



**Idle!**

**Thread 2 work queue**



**Working on bar()...**

**Thread 2 now resumes continuation  
and executes bar()  
Note block A descriptor is now free.**

# Cilk uses greedy join scheduling

## ■ Greedy join scheduling policy

- All threads always attempt to steal if there is nothing to do
- Threads only go idle if there is no work to steal in the system
- Worker thread that initiated spawn may not be thread that executes logic after `cilk_sync`

## ■ Remember:

- Overhead of bookkeeping steals and managing sync points only occurs when steals occur
- If large pieces of work are stolen, this should occur infrequently
  - Most of the time, threads are pushing/popping local work from their local deque

# Cilk summary

- **Fork-join parallelism: a natural way to express divide-and-conquer algorithms**
  - Discussed Cilk Plus, but many other systems also have fork/join primitives (e.g., OpenMP)
- **Cilk Plus runtime implements spawn/sync abstraction with a locality-aware work stealing scheduler**
  - Always run spawned child (continuation stealing)
  - Greedy behavior at join (threads do not wait at join, immediately look for other work to steal)