

# Work Distribution and Scheduling

---

# 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.

(Example: 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

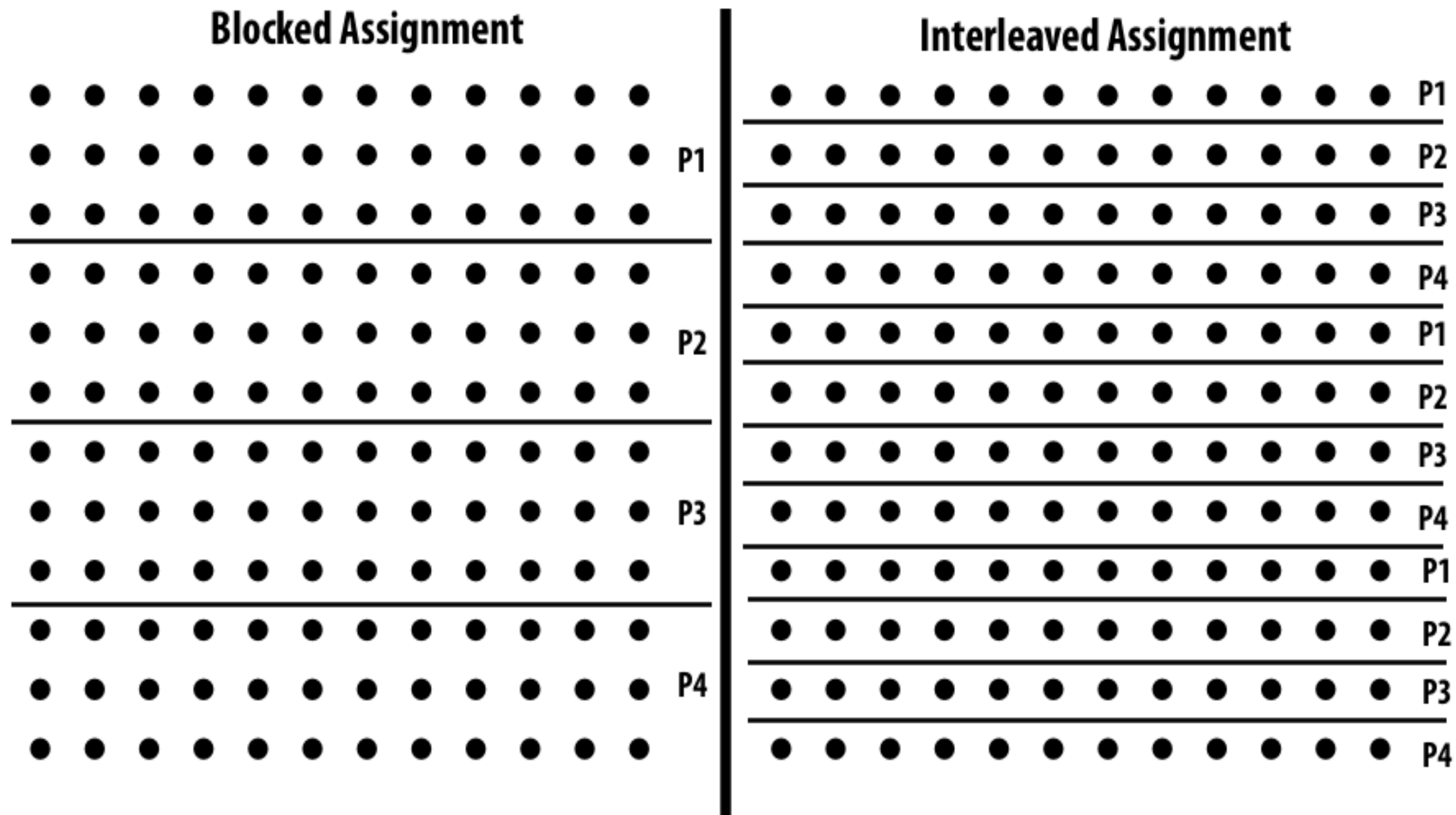
P4 does 20% more work → P4 takes 20%  
longer to complete

→ 20% of parallel  
program's runtime  
is serial execution

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

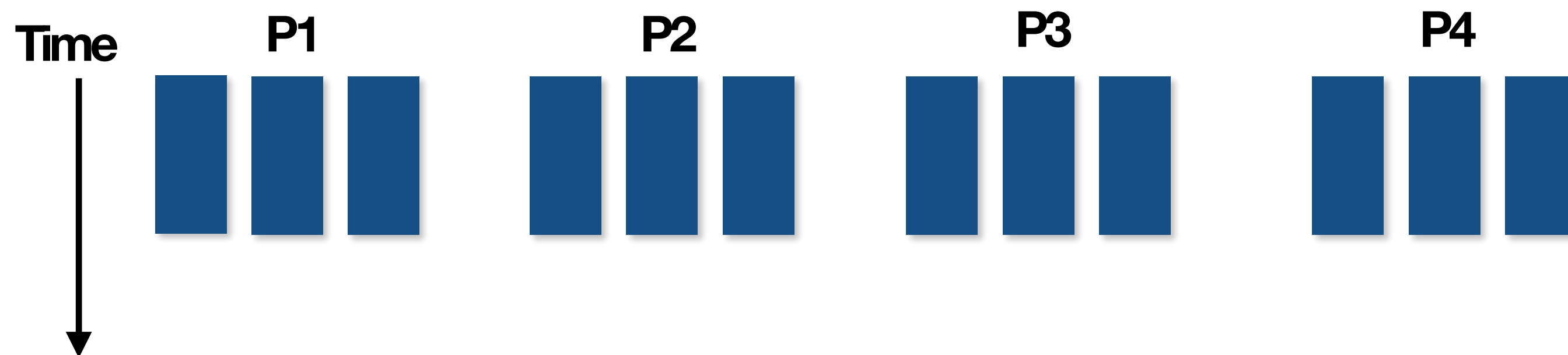
# Static assignment

- Assignment of work to threads is written into the program
  - Not necessarily determined at compile-time (assignment algorithm may depend on runtime parameters such as input, threads)
- Recall solver example: assign equal number of grid cells (work) to each thread (worker)



# 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



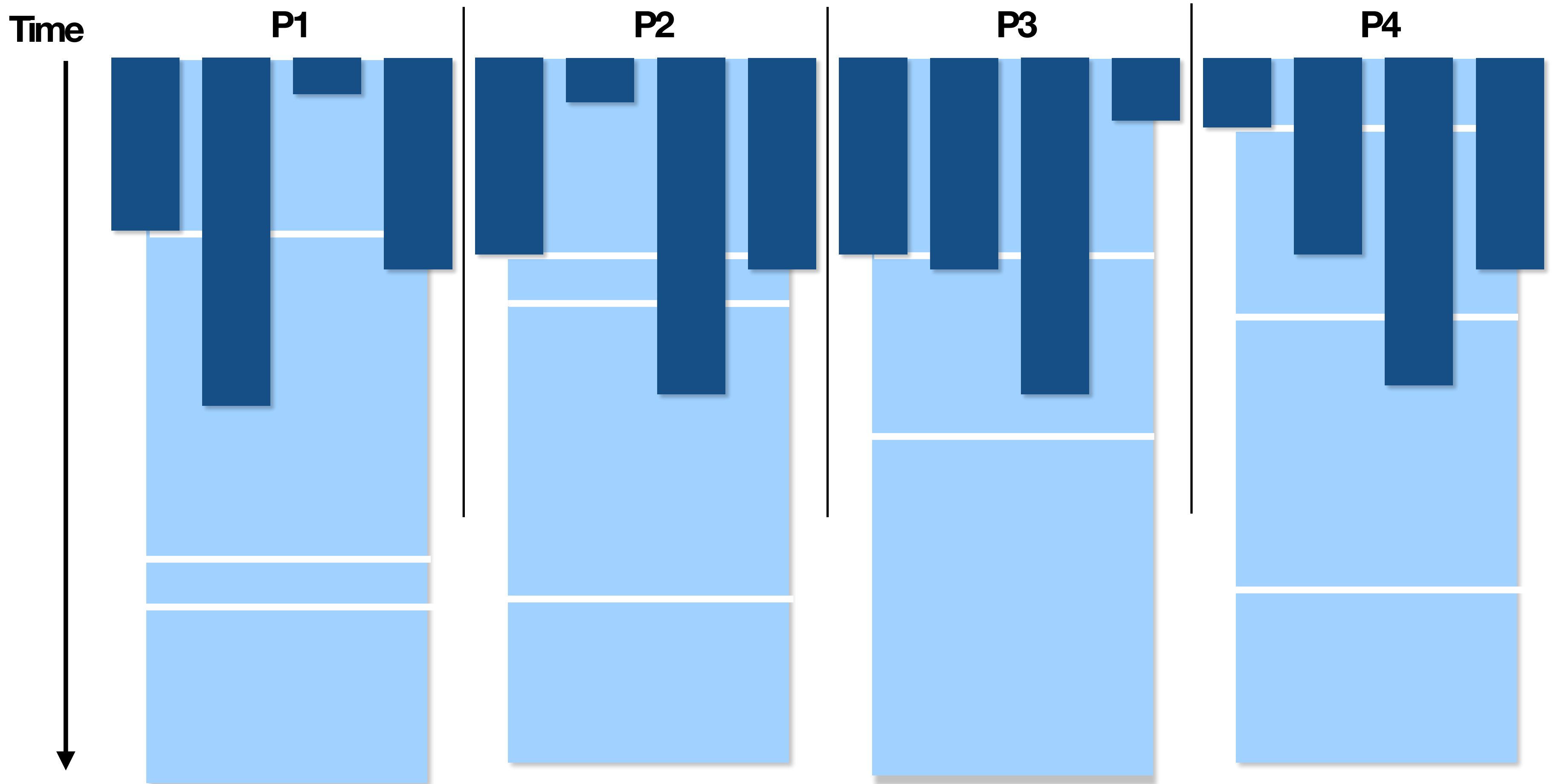
In the example above:

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
- When statistics about execution time are known (e.g., same cost)

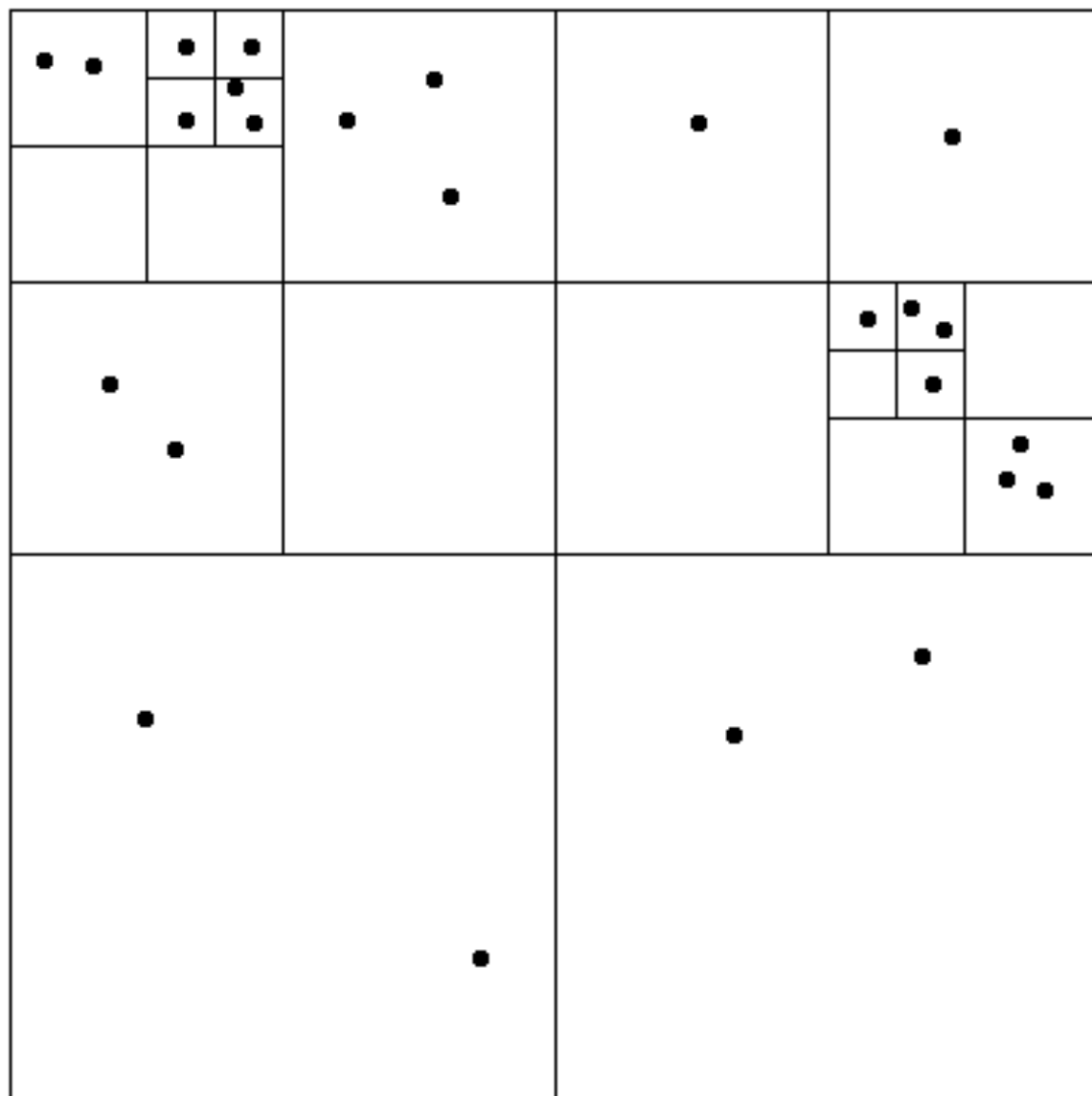


Jobs have unequal, but known cost: assign to processors to ensure overall good load balance



# “Semi-static” assignment

- Cost of work is predictable for near-term future
  - Idea: recent past is a 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)

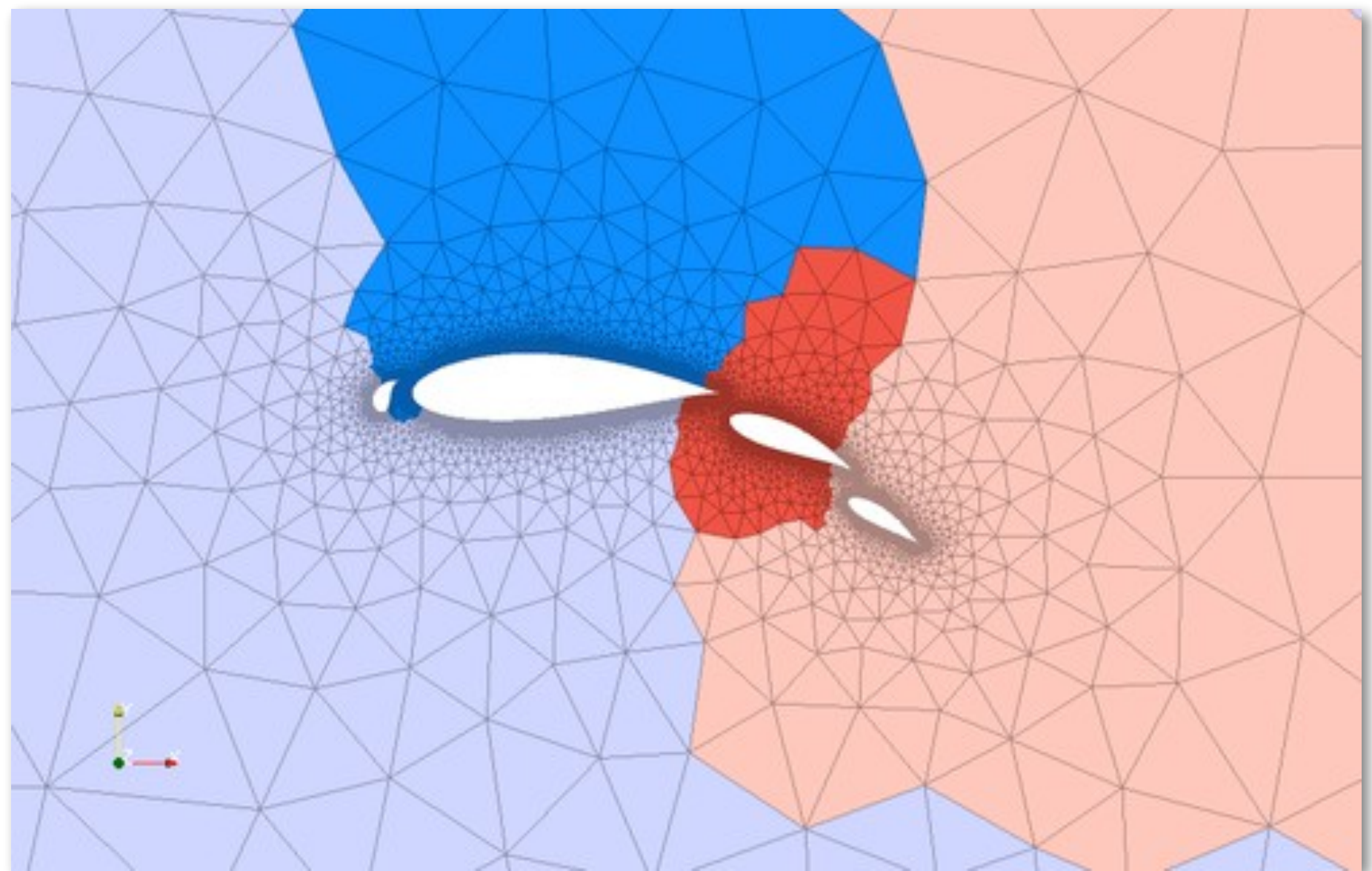


Image credit: <http://typhon.sourceforge.net/spip/spip.php?article22>

## 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.)

Parallel program

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

Sequential program  
(independent loop iterations)

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

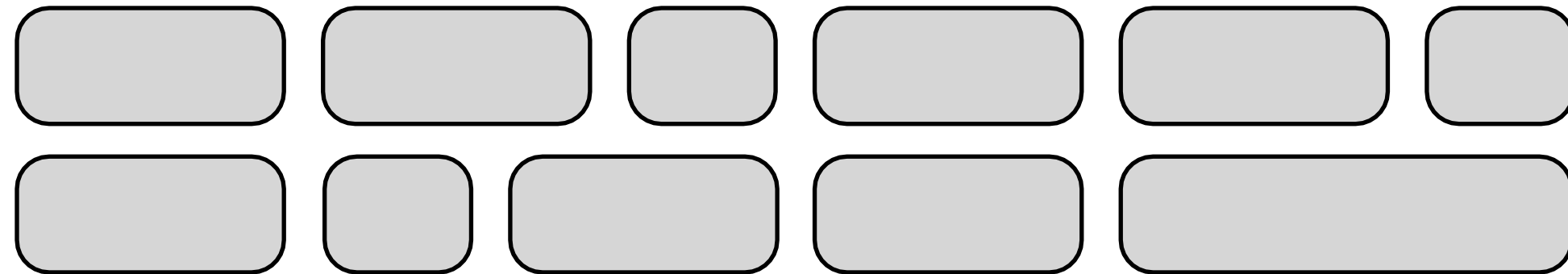
// assume elements of x initialized
for (int i=0; i<N; i++)
{
    // unknown execution time
    is_prime[i] = test_primalty(x[i]);
}
```

```
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

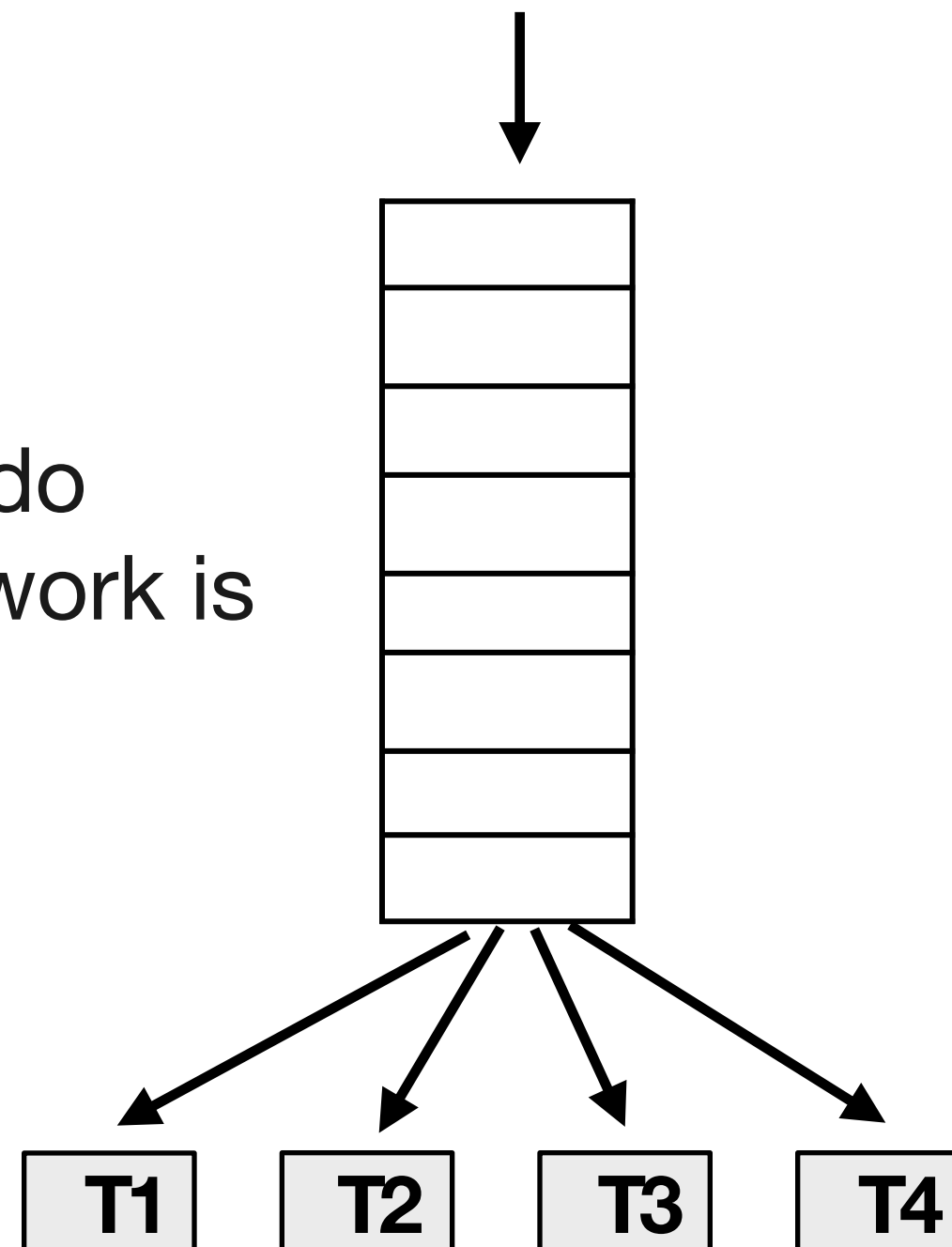
while (1) { int
    i;
    lock(counter_lock);
    i = counter++;
    atomic_incr(counter);
    unlock(counter_lock);
    if (i >= N)
        break;
    is_prime[i] = test_primalty(x[i]);
}
```

# 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

# What constitutes a piece of work?

## What is a potential problem with this design?

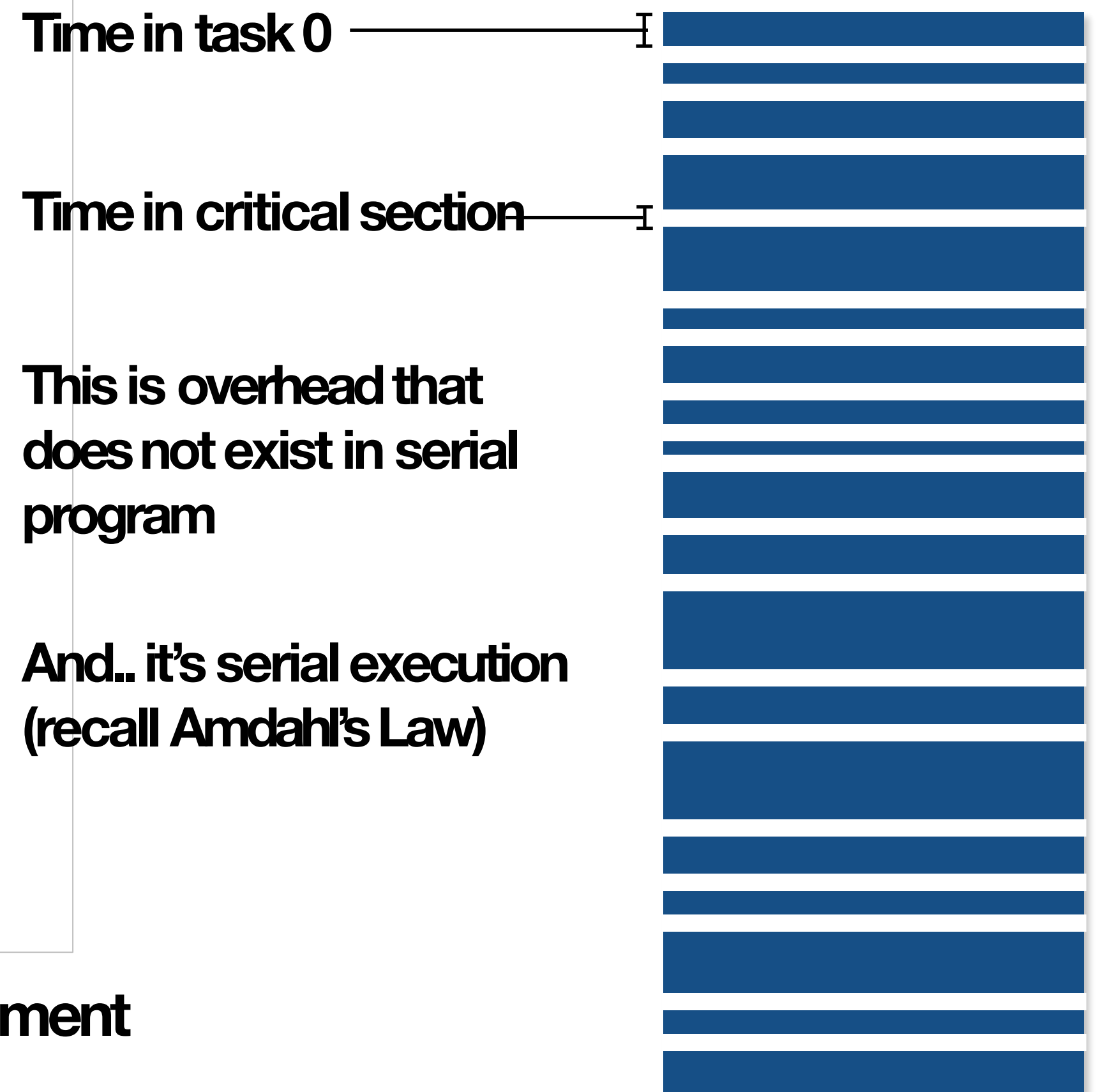
```
const int N = 1024;

// 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?

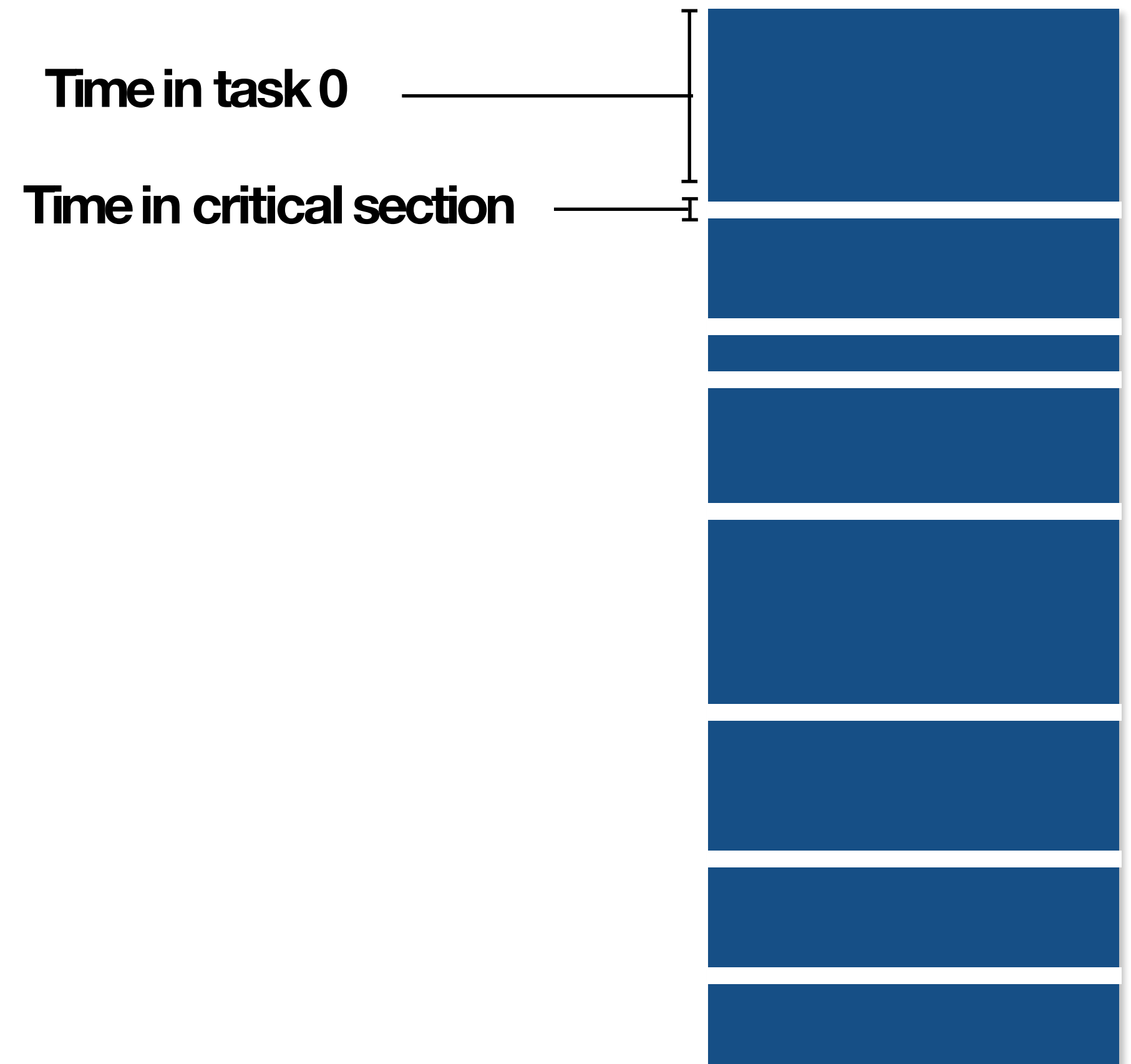
# Increasing task granularity

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



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

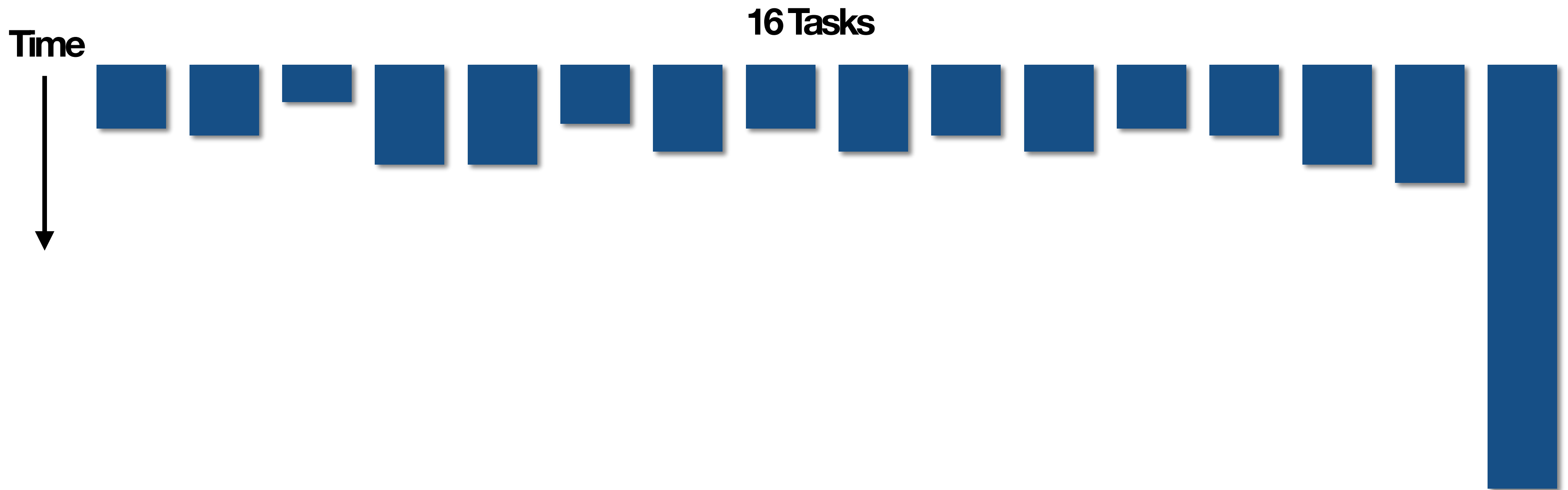
# Choosing task size

- 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)

# Smarter task scheduling

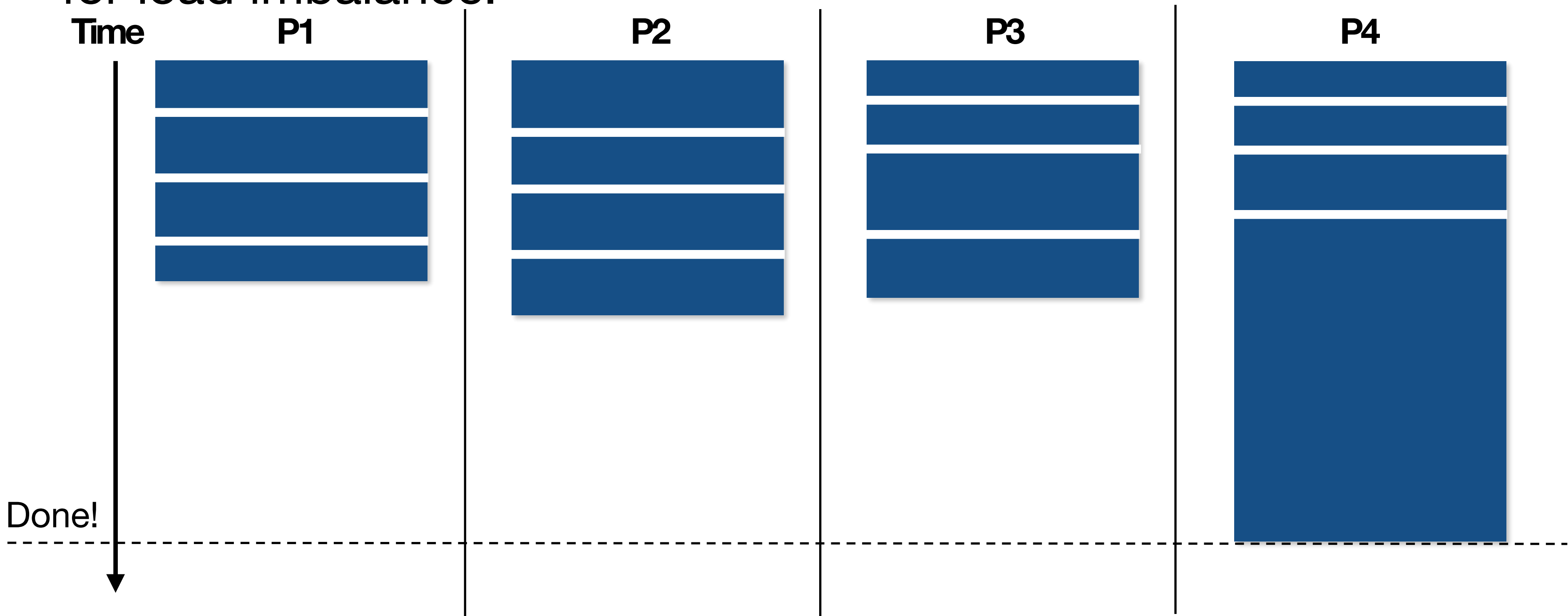
Consider dynamic scheduling via a shared work queue

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



# Smarter task scheduling

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



One possible solution to imbalance problem:

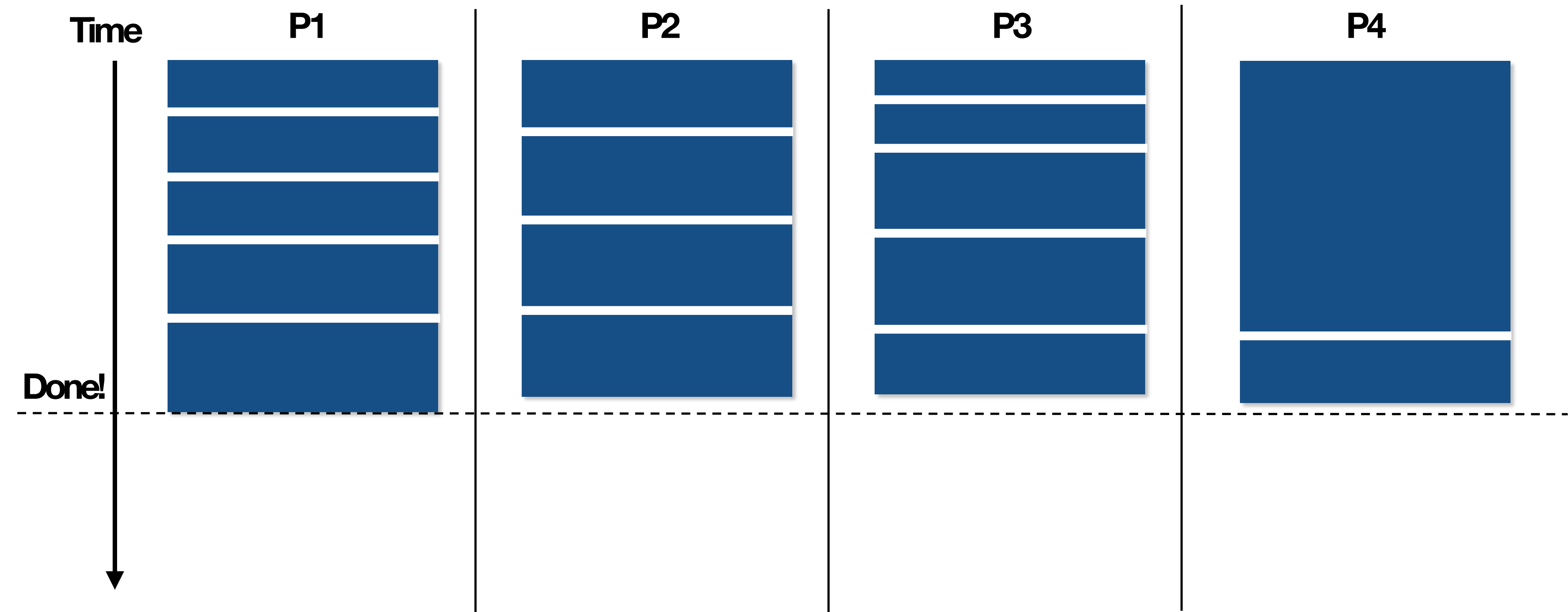
Divide work into a larger number of smaller tasks

- 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 “slack” at end of computation



Another solution: smarter

Schedule long tasks first

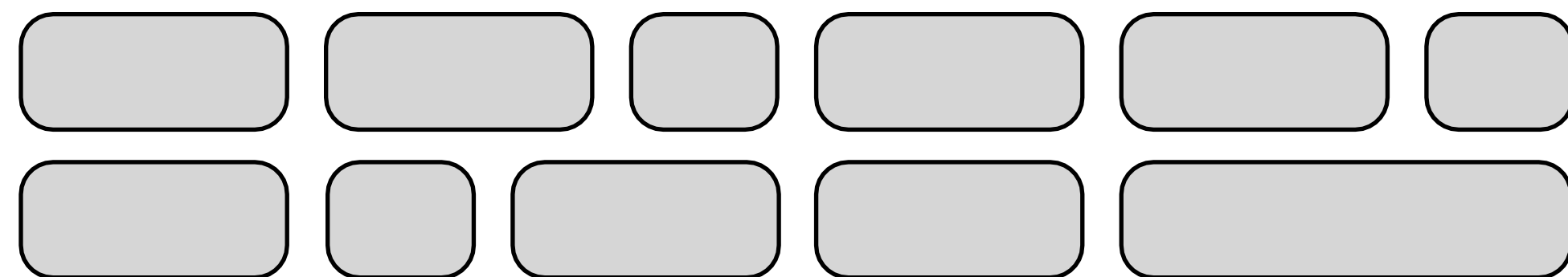
- Thread performing long task performs fewer overall tasks, but approximately the same amount of work as the other threads.
- Requires some knowledge of workload (some predictability of cost)

# Decreasing synchronization overhead using a distributed set of queues

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

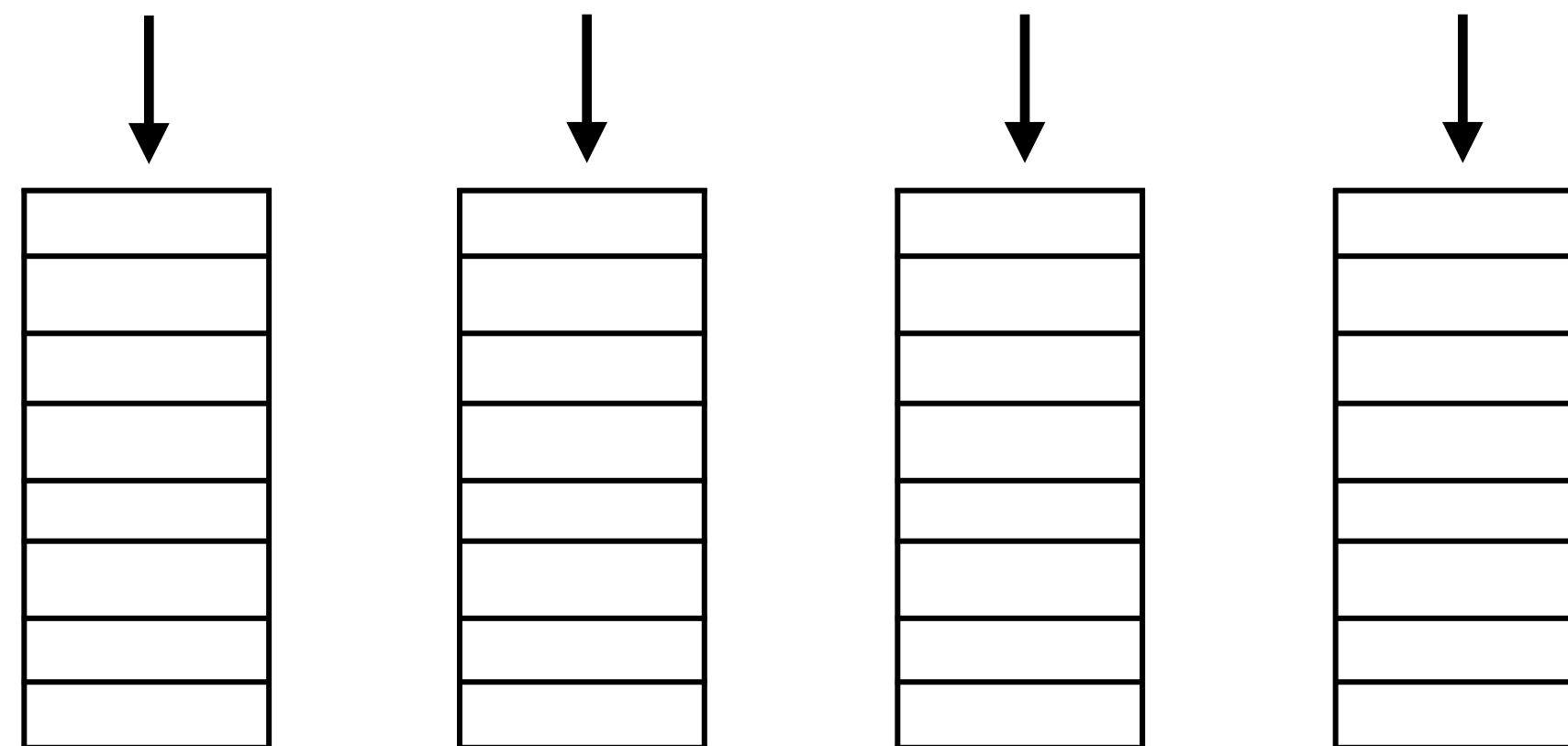
Subproblems

(a.k.a. “tasks”, “work to do”)



Set of work queues

(In general, one per worker thread)



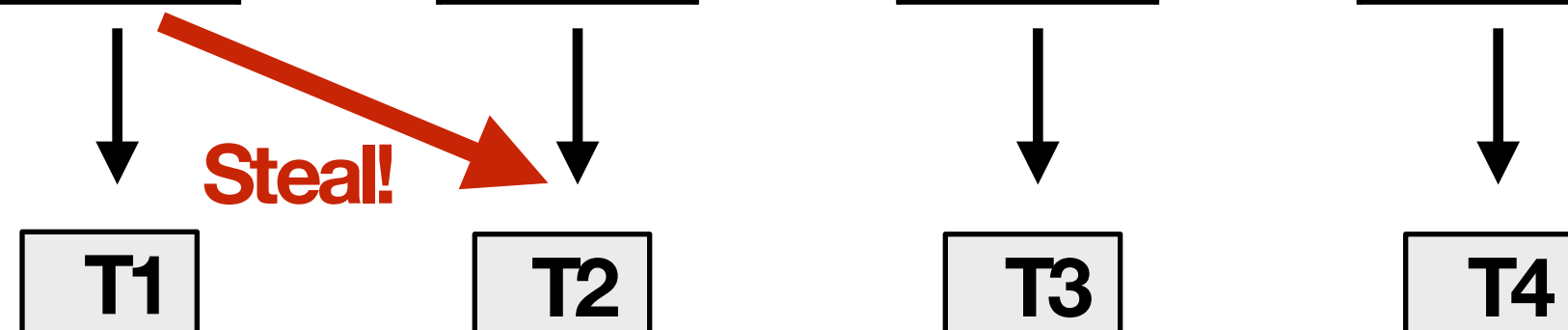
**Worker threads:**

Pull data from OWN work queue

Push new work to OWN work queue

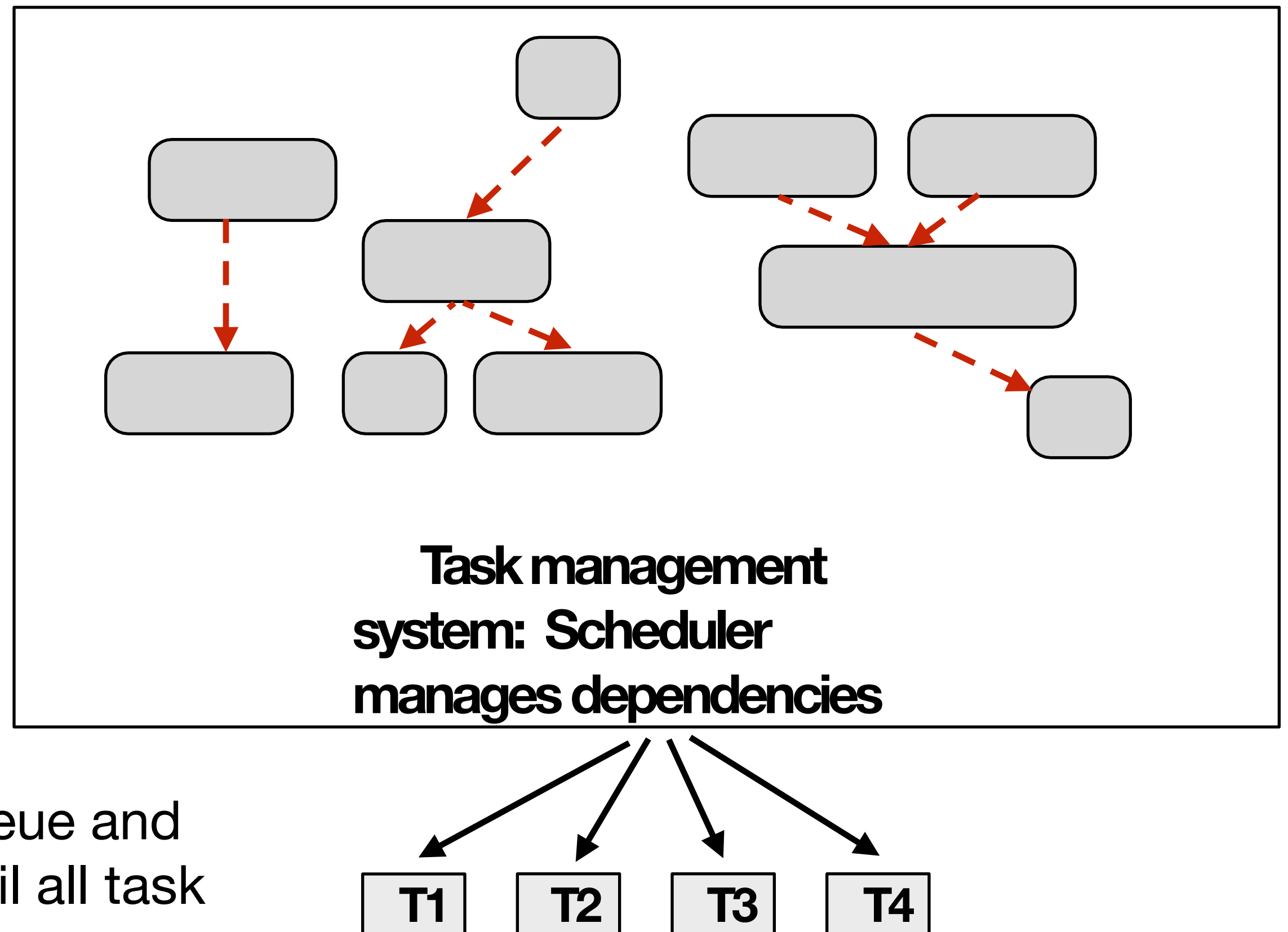
When local work queue is empty...

STEAL work from another work queue



# Work in task queues need not be independent

- -> = application-specified dependency



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

```
foo_handle = enqueue_task(foo);  
bar_handle = enqueue_task(bar, foo_handle);
```

```
// enqueue task foo (independent of all prior tasks)  
// enqueue task bar, cannot run until foo is complete
```

# Summary

- Challenge: achieving good workload balance
  - Want all processors working all the time
  - But want low-cost solution for achieving this balance
    - Minimize computational overhead
    - Minimize synchronization costs
- Static assignment vs. dynamic assignment
  - Really, it is not an either/or decision, 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 aspects of task decomposition, assignment, and orchestration

# **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]);  
}
```

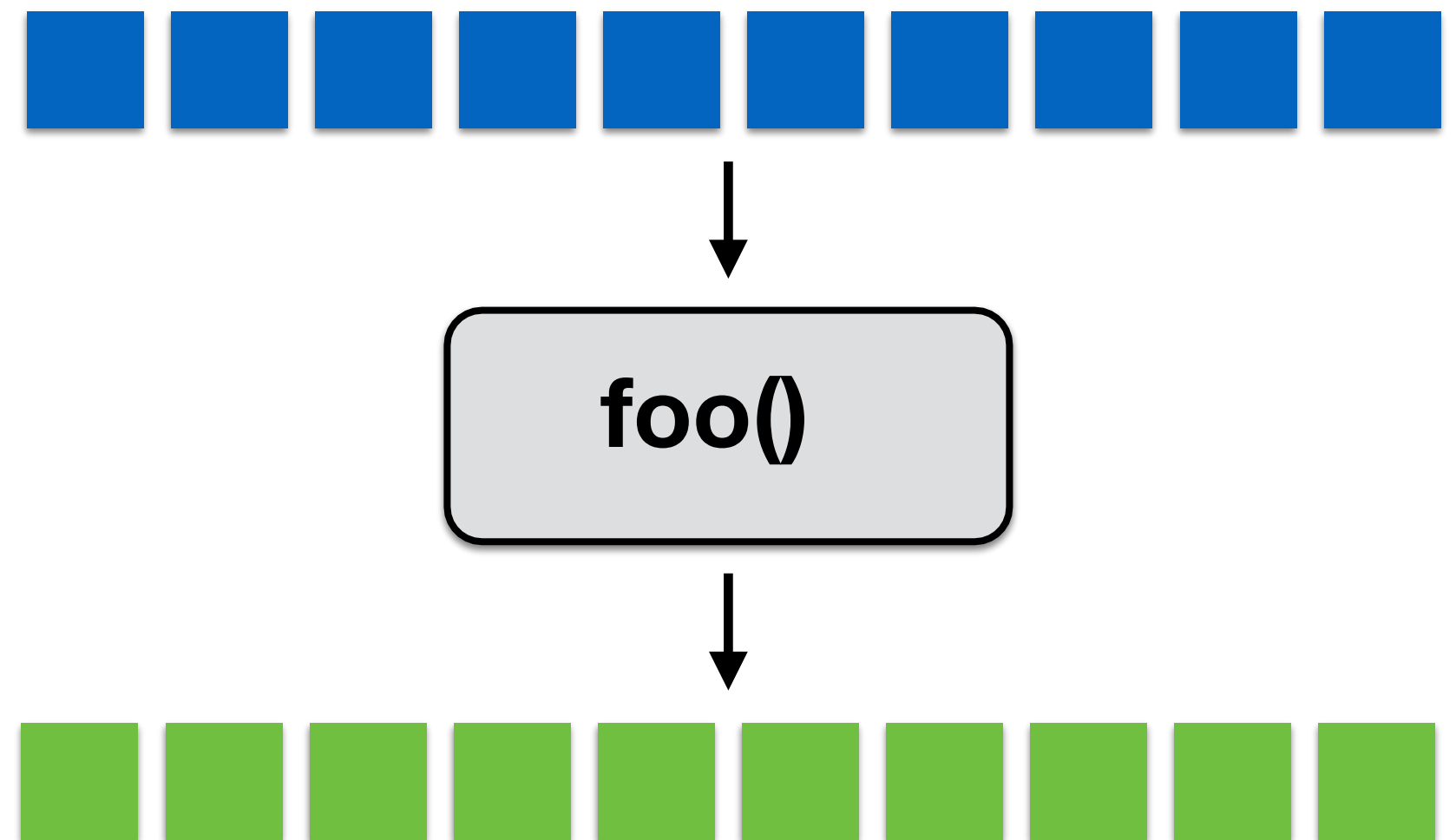
```
// cilk_for(i=0 ... N) {  
    B[i] = foo(A[i]);  
}
```

```
// 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)  
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 pthreads

```
struct thread_args {  
    float* A;  
    float* B;  
};  
int thread_id[MAX_THREADS];  
  
thread_args args;  
  
args.A = A;  
args.B = B;  
  
for (int i=0; i<num_cores; i++)  
    { pthread_create(&thread_id[i], NULL, myFunctionFoo, &args);  
    }  
  
for (int i=0; i<num_cores; i++) {  
    pthread_join(&thread_id[i]);  
}
```



# 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);
```

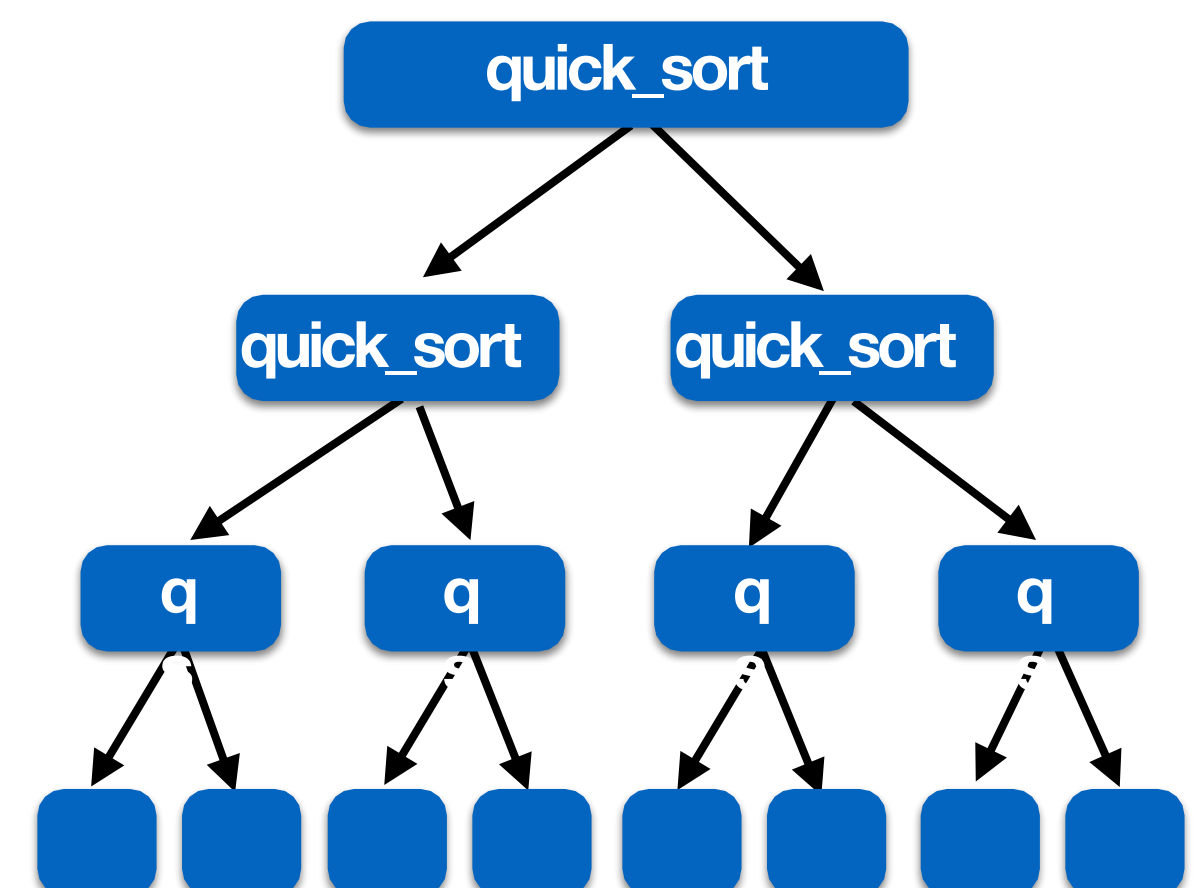
```
        quick_sort(middle+1, end);
```

```
    }
```

```
}
```

**independent  
work!**

## Dependencies



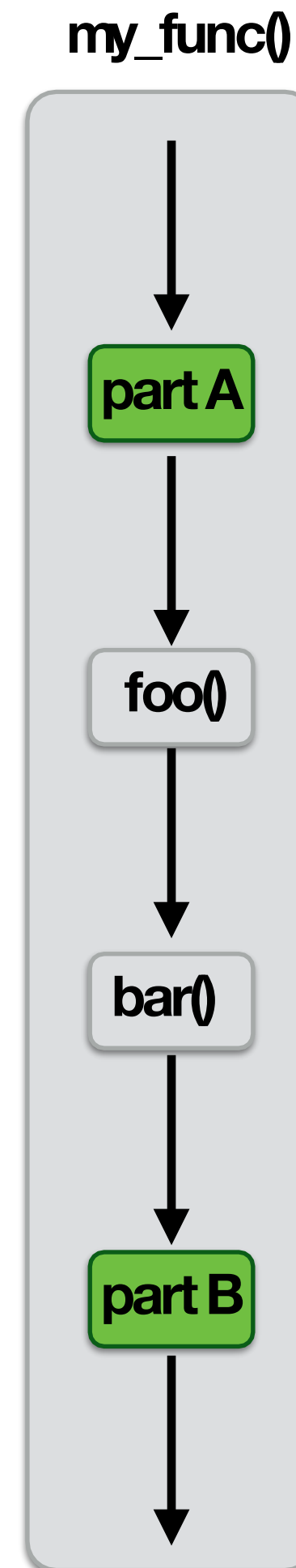
# Call-return of a C function

```
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)



# Fork-join pattern

- Natural way to express independent work in divide-and-conquer algorithms
- This lecture's code examples will be in Cilk Plus
  - C++ language extension
  - Originally developed at MIT, acquired by Intel
  - But Intel is deprecating it. Best to stick with MIT version

`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)

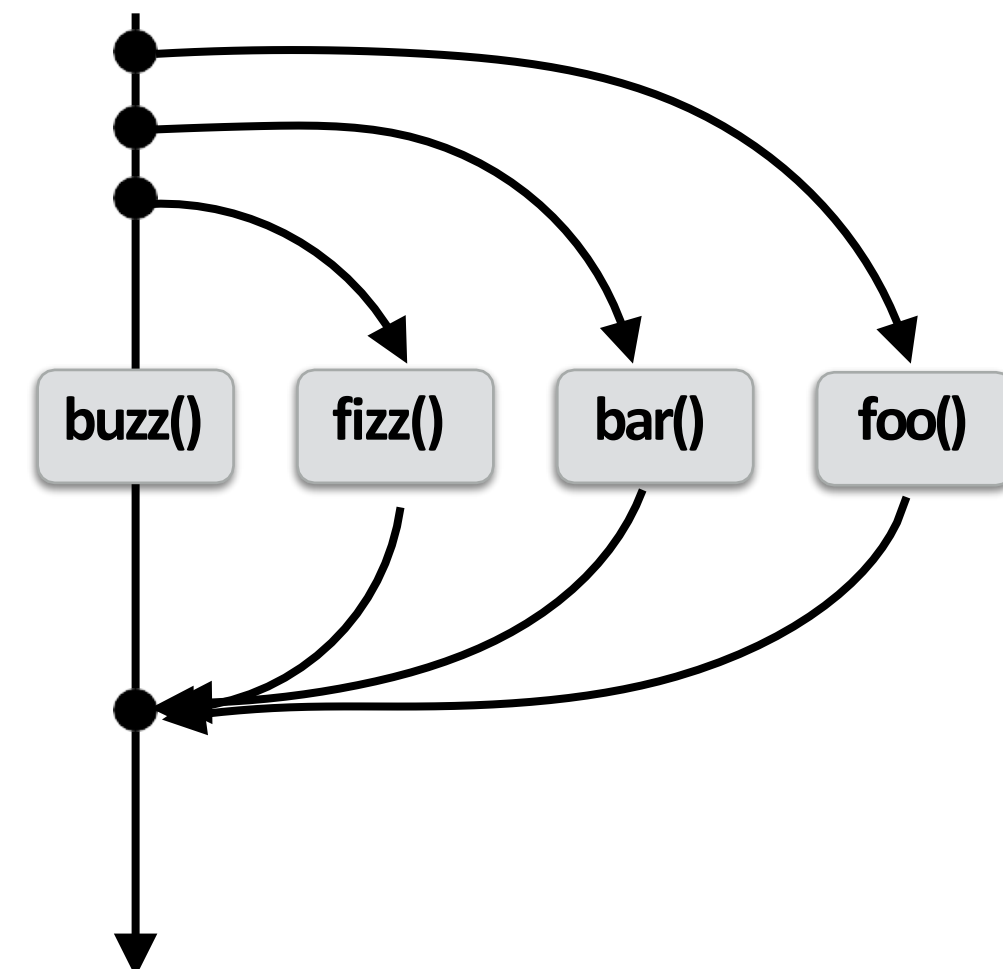
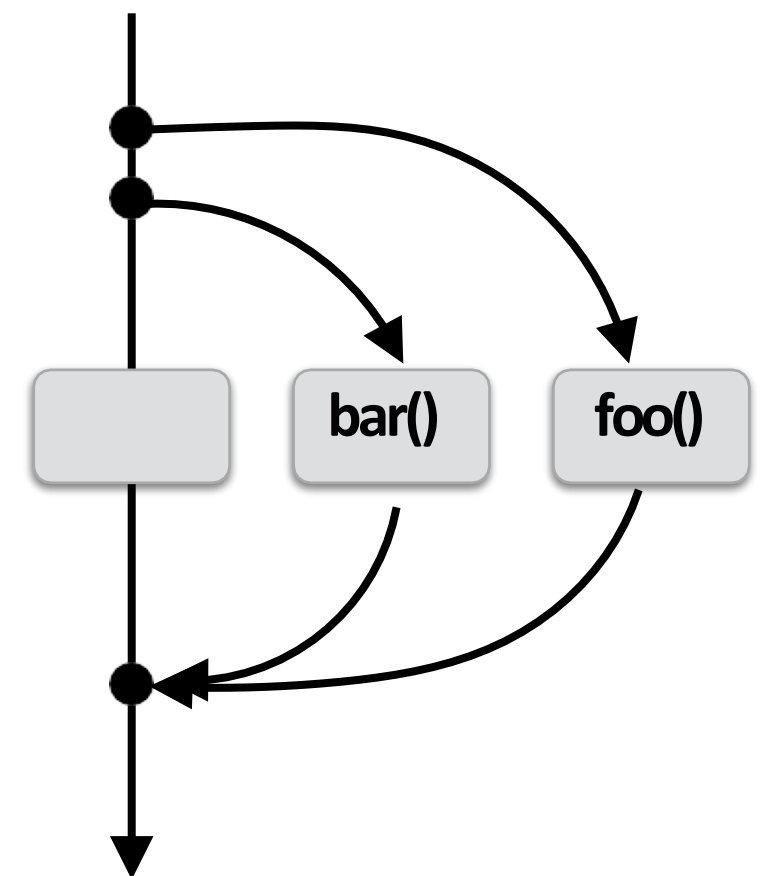
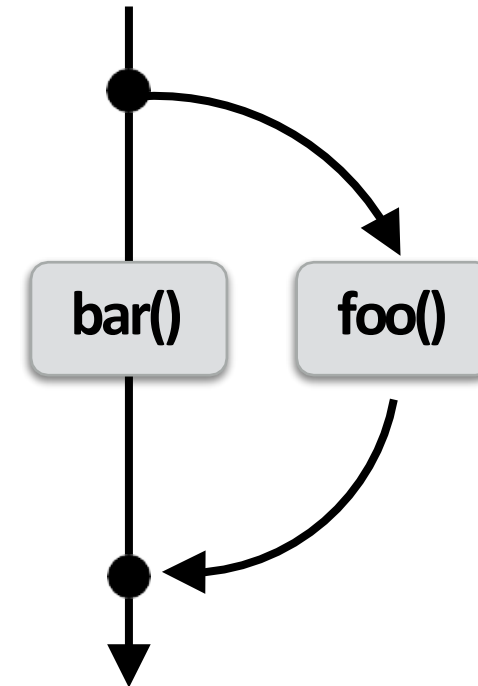
# Basic Cilk Plus examples

```
// 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;
```

Same amount of independent work first example, but potentially higher runtime overhead (due to two spawns vs. one)

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



# Abstraction vs. implementation

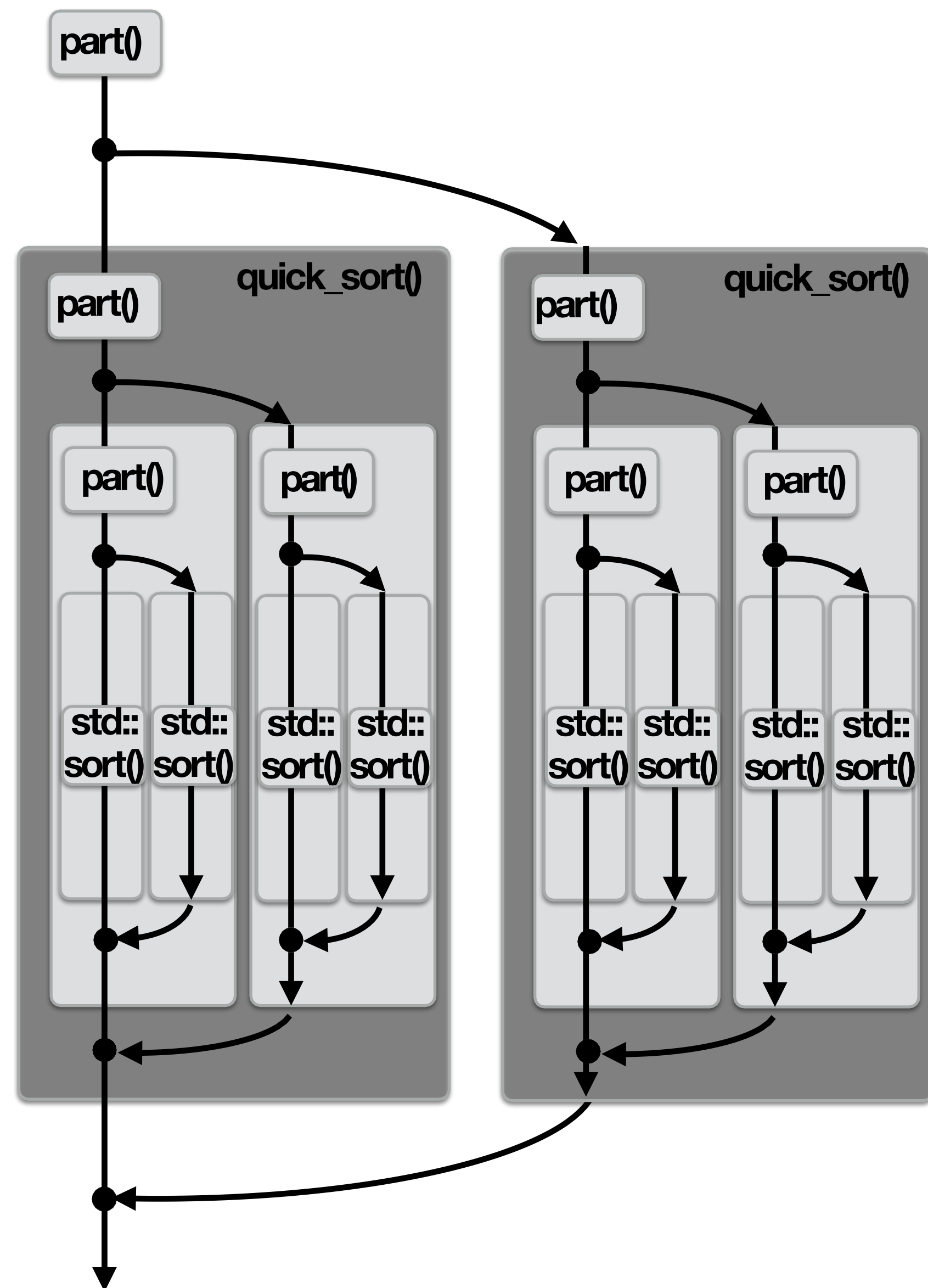
- Notice that the `cilk_spawn` abstraction does not specify how or when spawned calls are scheduled to execute
  - Only that they `may be run concurrently with caller` (and with all other calls spawned by the caller)
- But `cilk_sync` does serve as a `constraint on scheduling`
  - All spawned calls must complete before `cilk_sync` returns

```

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 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
  - Want more independent work than execution capability to allow for good workload balance
  - “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)

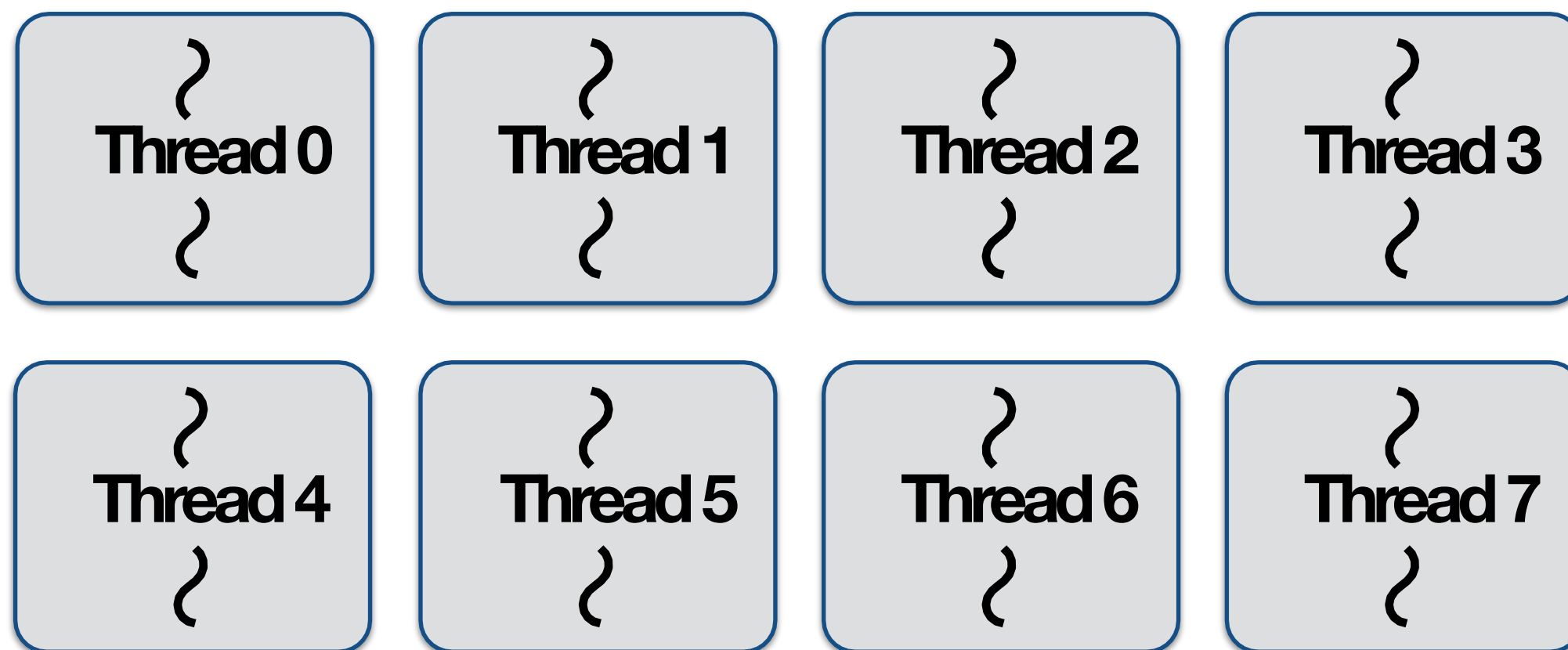


# Scheduling fork-join programs

- **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

# Pool of worker threads

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



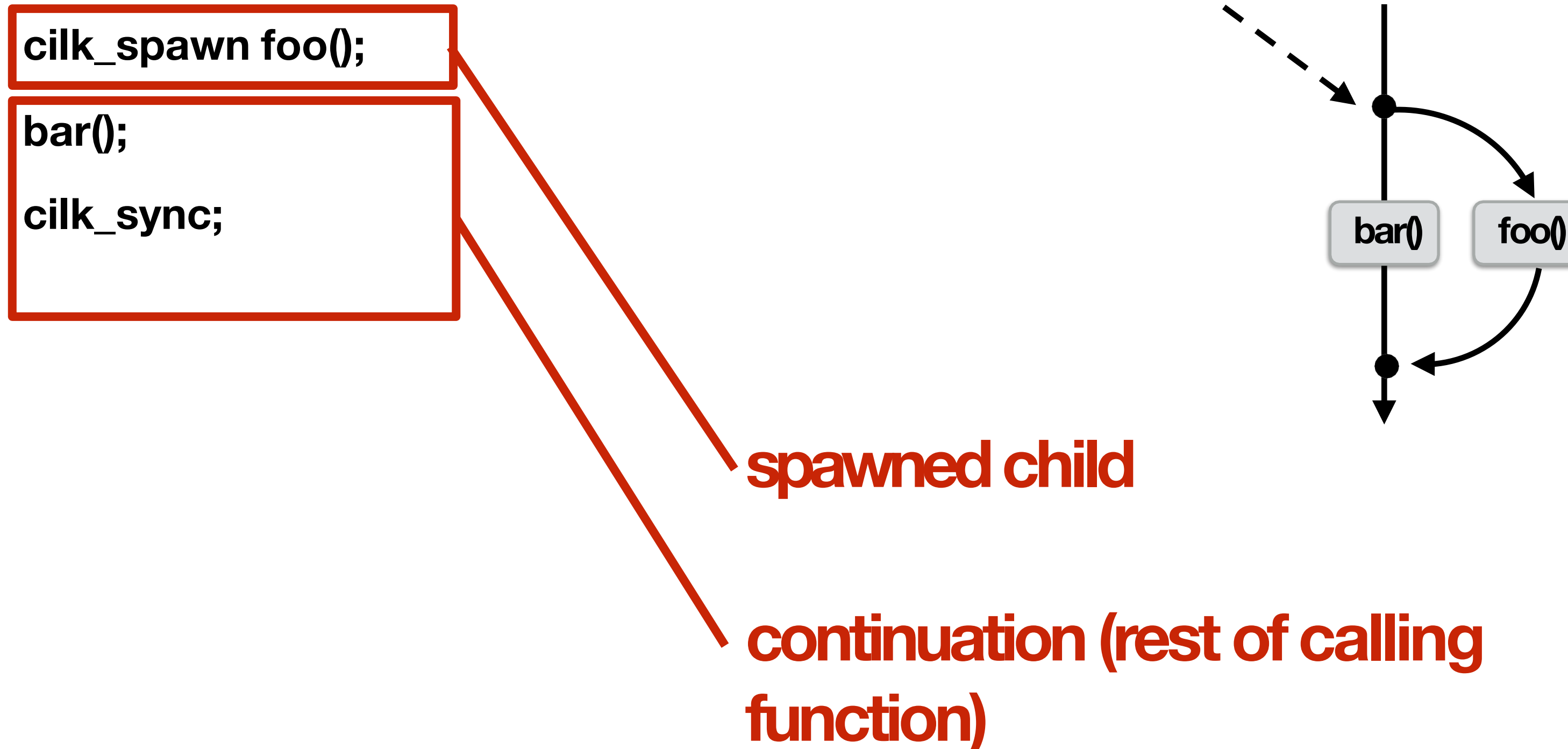
Example: Eight thread worker pool for my quad-core laptop with Hyper-Threading

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

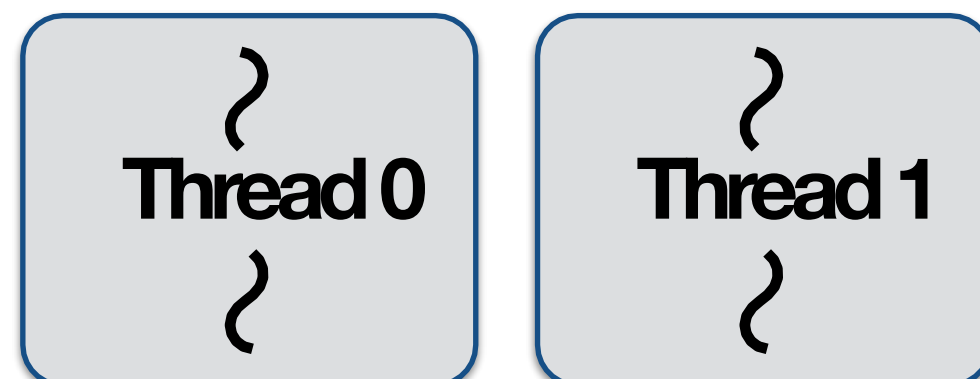
\* It's perfectly fine to think about it this way, but in reality, runtimes tend to be lazy and initialize worker threads on the first Cilk spawn. (This is a common implementation strategy, ISPC does the same with worker threads that run ISPC tasks.)

# Consider execution of the following code

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



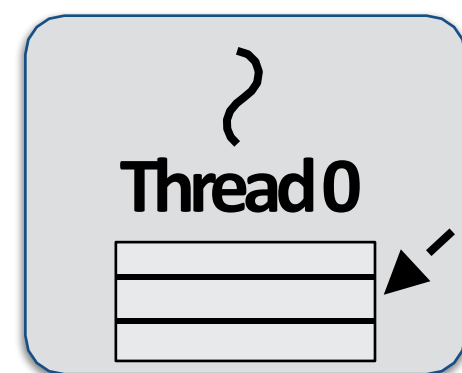
What threads should `foo()` and `bar()` be executed by?



# First, consider a **serial** implementation

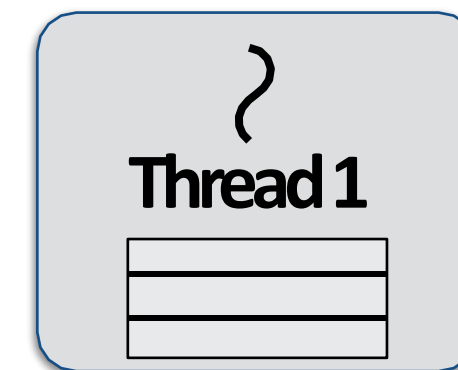
Run child first... via a regular function call

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



Executing `foo()`...

Traditional thread call stack  
(indicates bar will be performed  
next after return)



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

**Inefficient: thread 1 could be  
performing `bar()` at this time!**

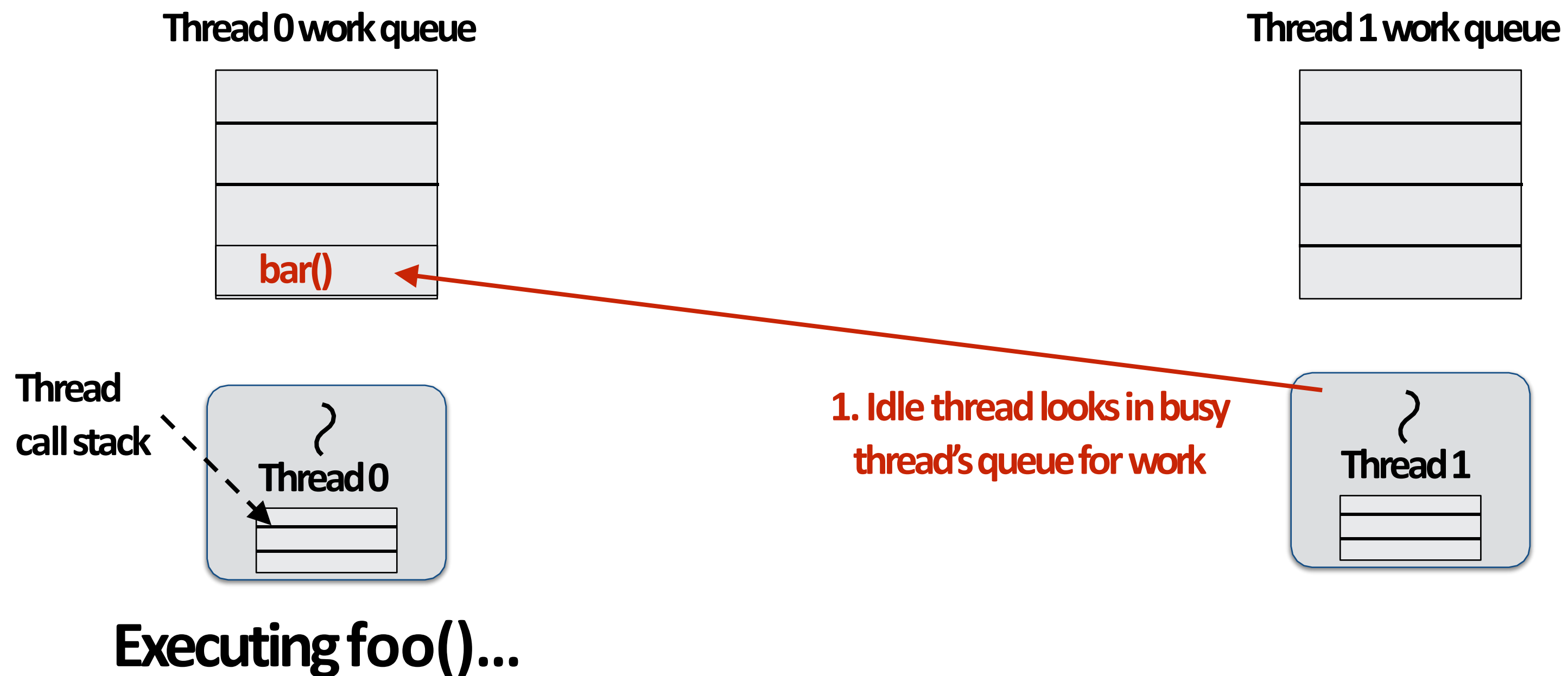
# Per-thread work queues store “work to do”

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



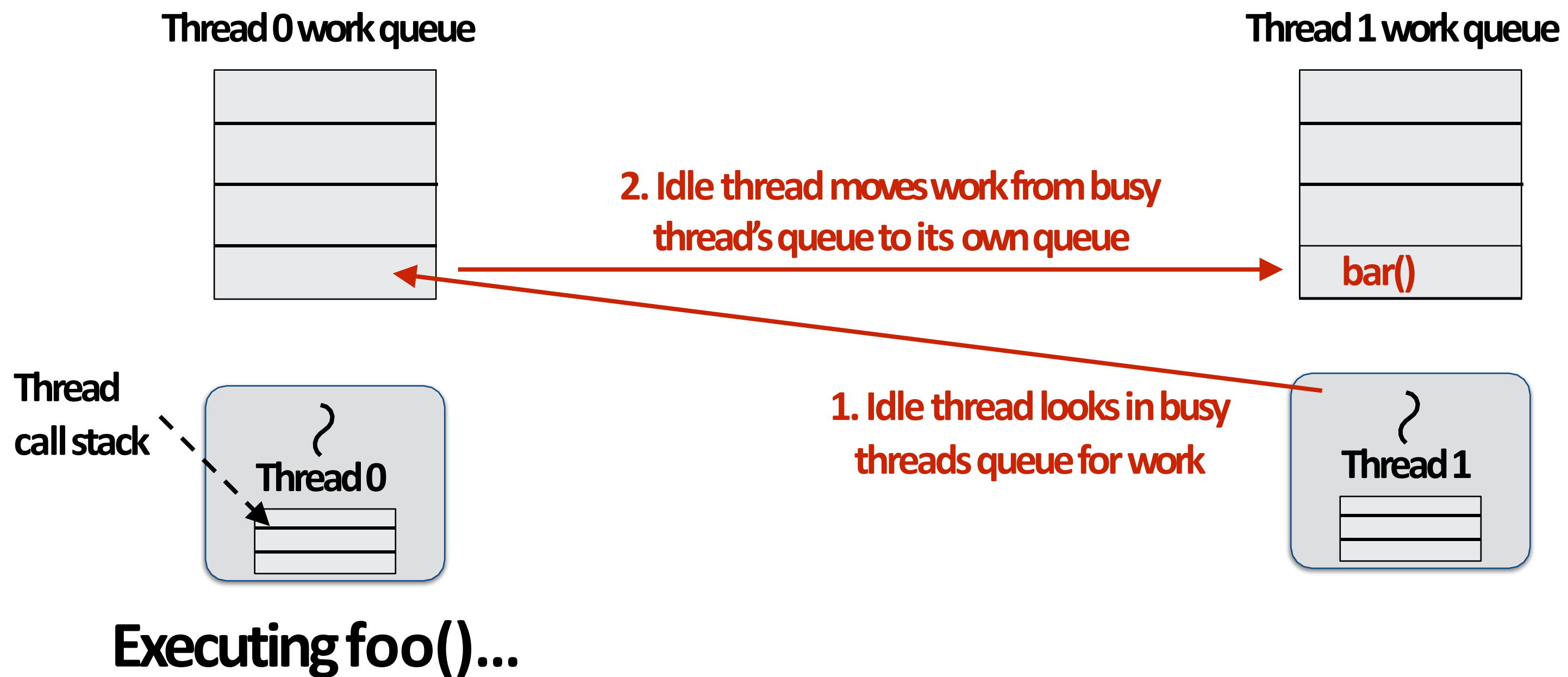
# Idle threads “steal” work from busy threads

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

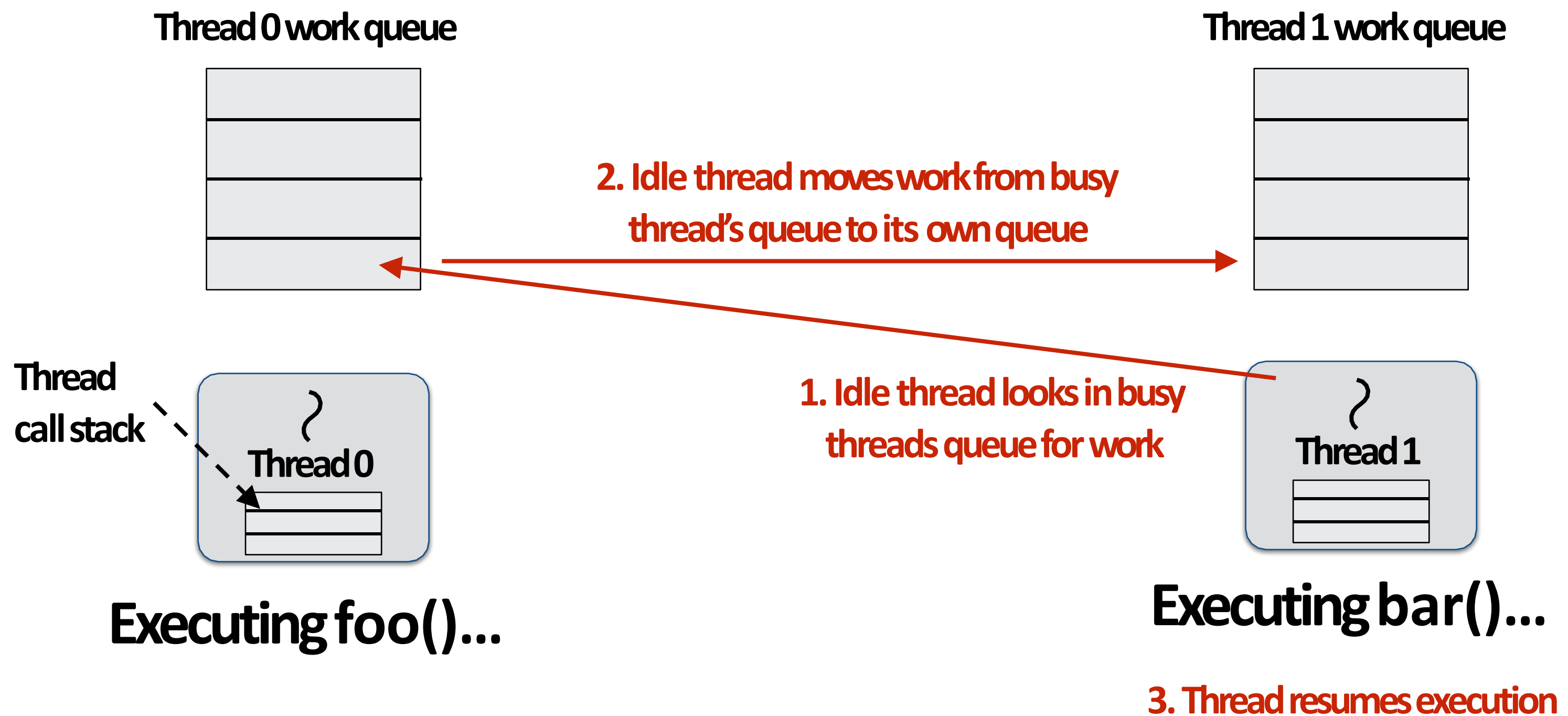
If **thread 1 goes idle** (a.k.a. there is no work in its own queue), then it looks in thread 0's queue for work to do.





# 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 thread run child or

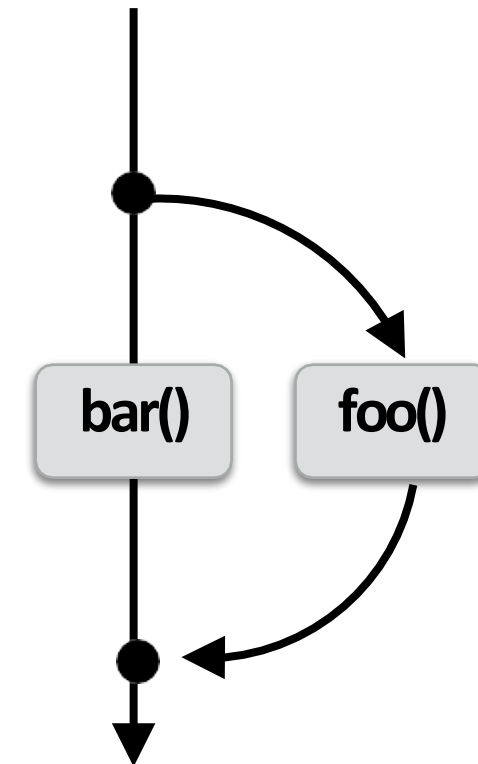
```
cilk_spawn foo();
```

```
bar();
```

```
cilk_sync;
```

spawned child

continuation (rest of calling function)



Run **continuation first**: record child for later execution

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

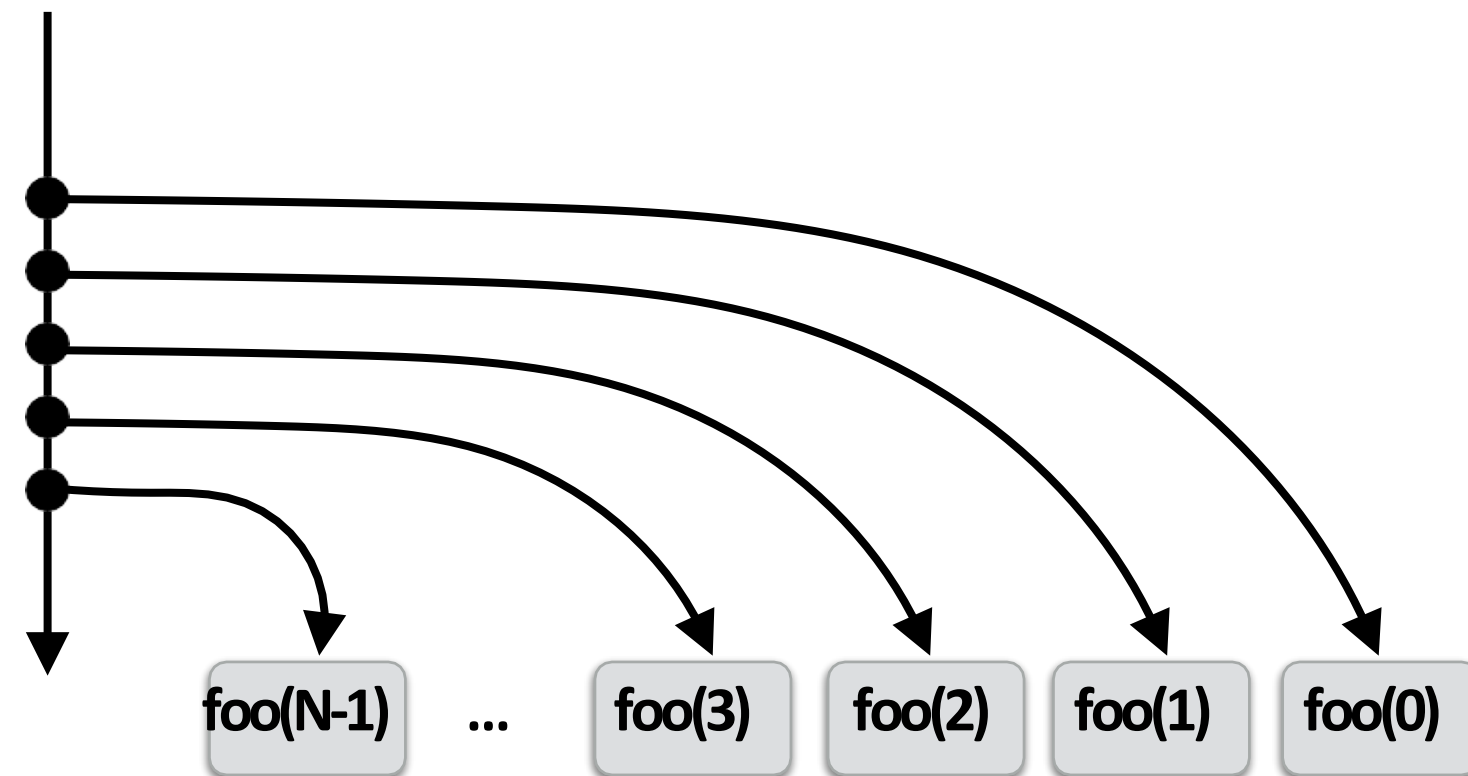
Run **child first**: record continuation for later execution

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

Which implementation do we choose?

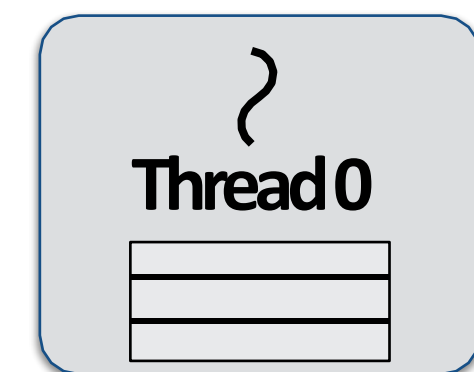
# Consider thread executing the following code

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



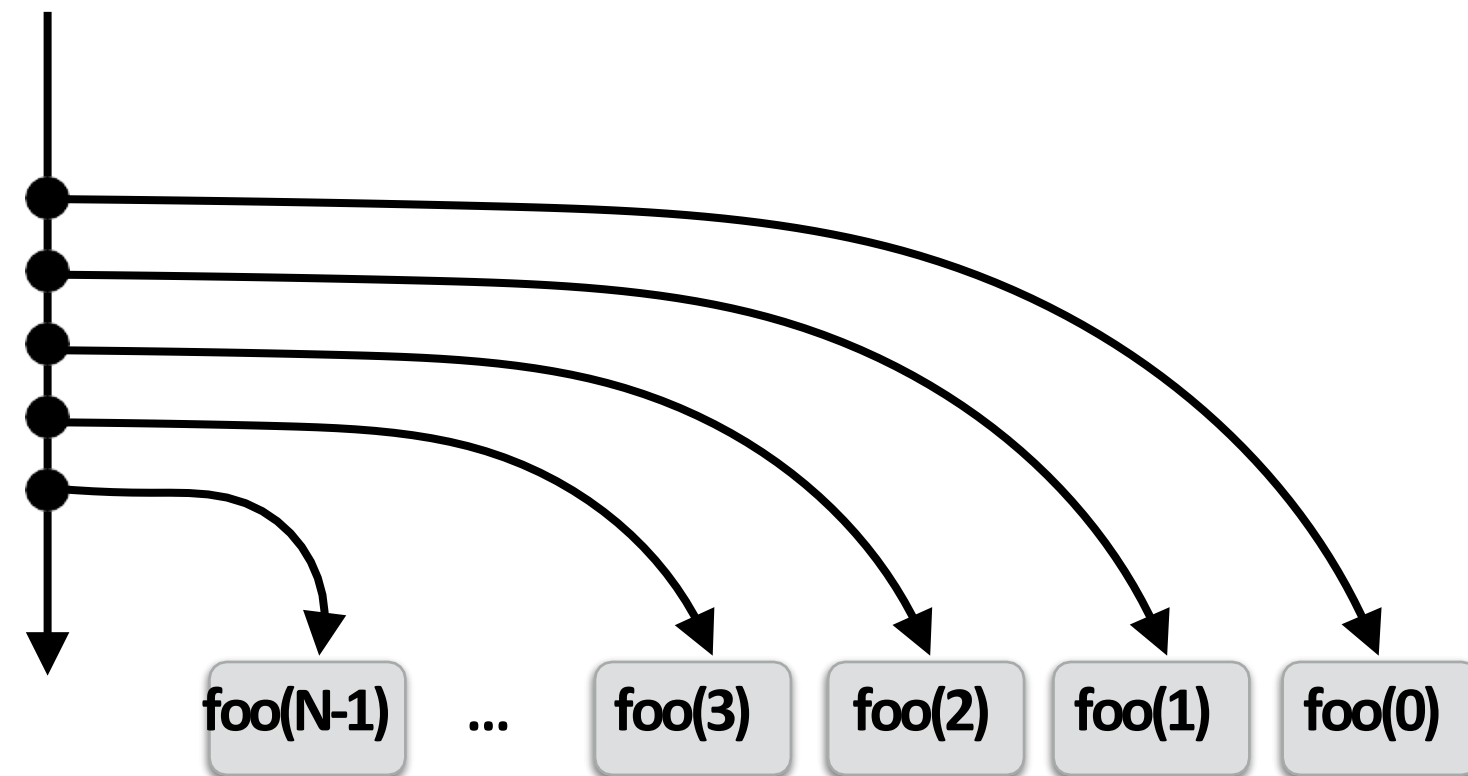
- Run **continuation first** (“**child stealing**”)
  - Caller thread spawns work for all iterations before executing any of it
    - Think: **breadth-first traversal of call graph**.  $O(N)$  space for spawned work (maximum space)
    - If no stealing, **execution order is very different than that of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

Thread 0 work queue



# Consider thread executing the following code

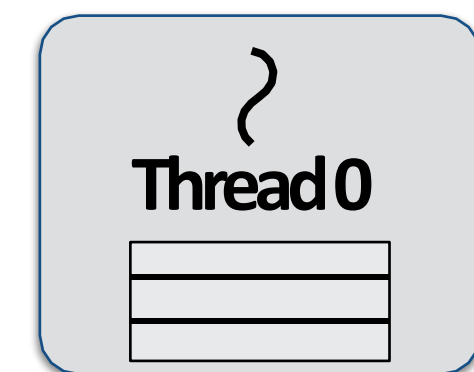
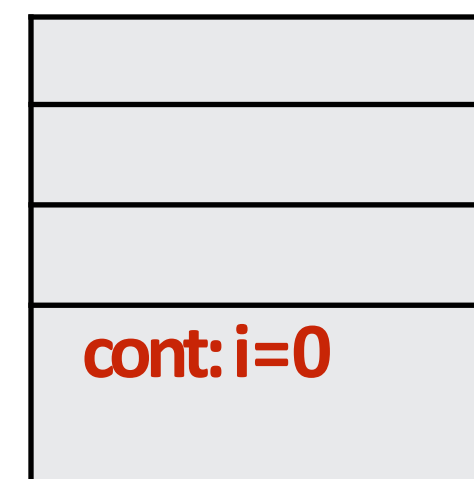
```
for (int i=0; i<N; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync;
```



## ■ 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**

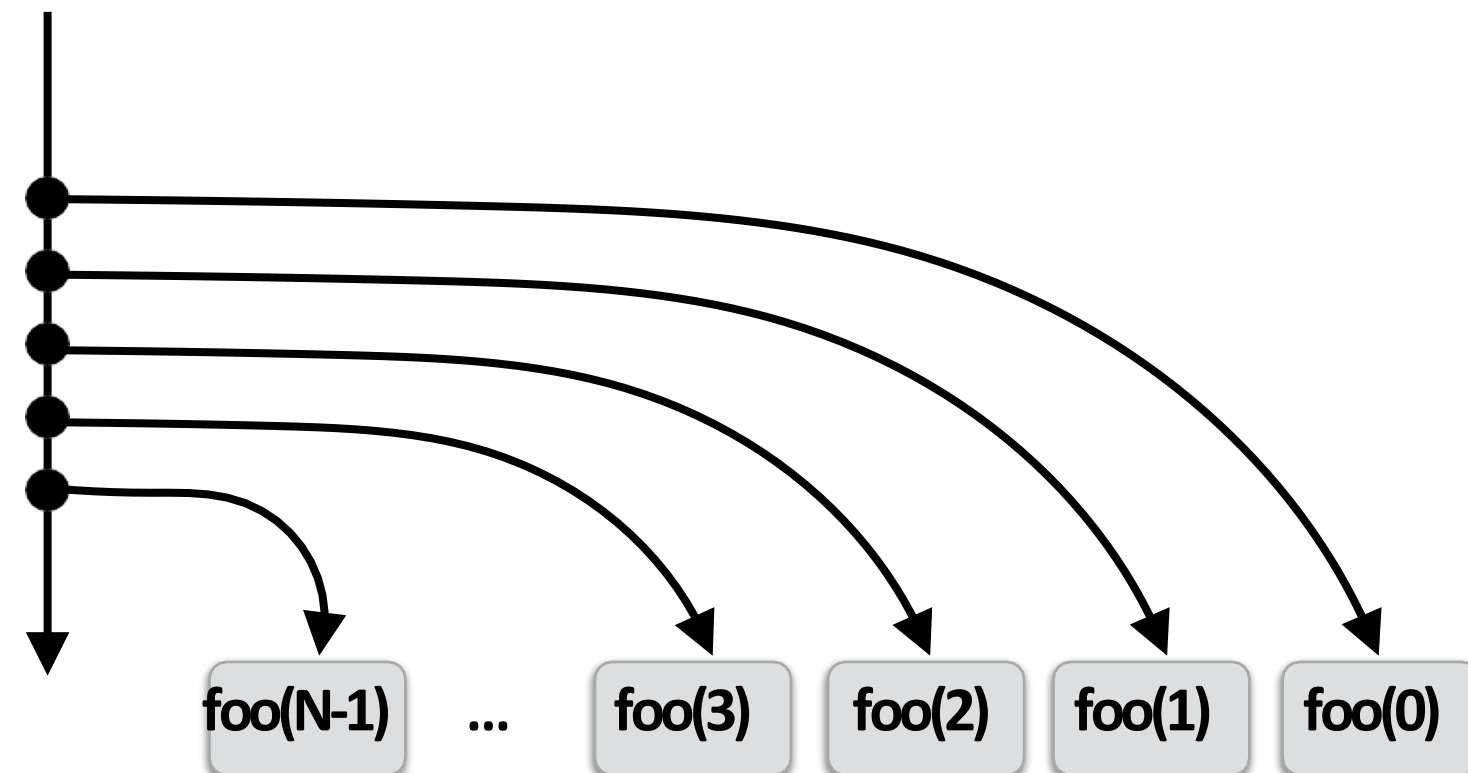
Thread 0 work queue



Executing `foo(0)`...

# Consider thread executing the following code

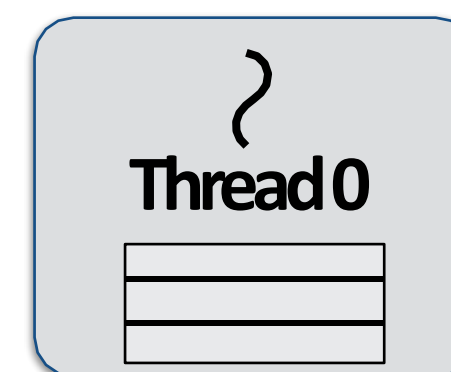
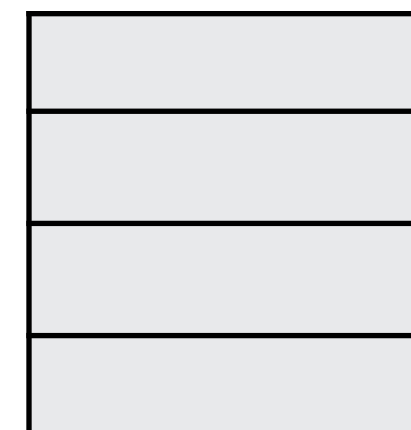
```
for (int i=0; i<N; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync;
```



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

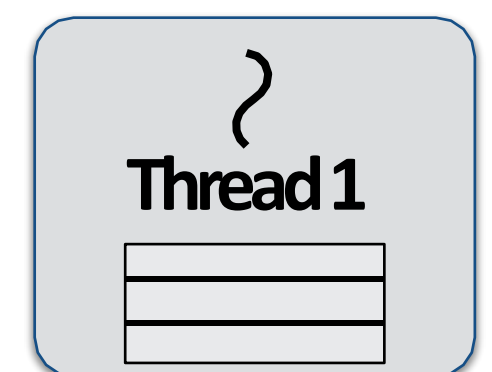
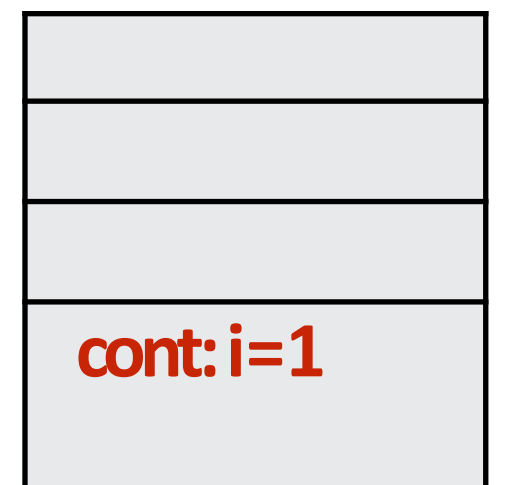
- If continuation is stolen, stealing thread spawns and executes next iteration
- Enqueues continuation with `i` advanced by 1
- Can prove that **work queue storage** for system with `T` threads is **no more than `T` times that of stack storage for single threaded execution**

Thread 0 work queue



Executing `foo(0)`...

Thread 1 work queue



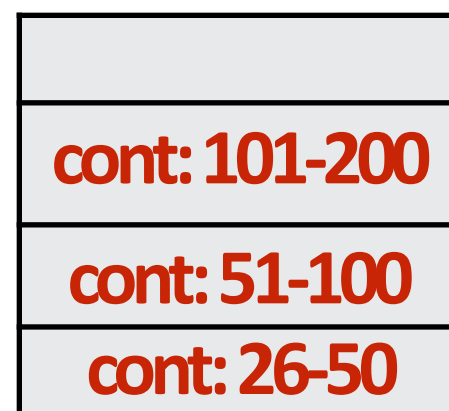
Executing `foo(1)`...

# Scheduling quicksort: assume 200 elements

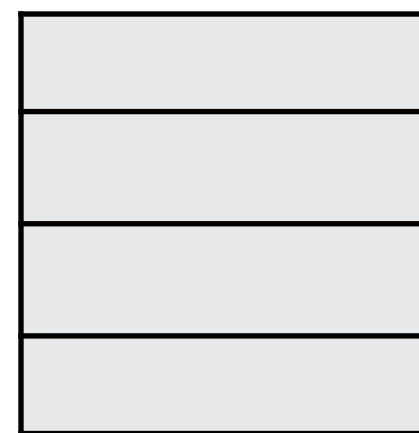
```
void quick_sort(int* begin, int* end)
{  if (begin >= end - PARALLEL_CUTOFF)
    std::sort(begin, end);
  else {
    int* middle = partition(begin, end);
    cilk_spawn quick_sort(begin, middle);
    quick_sort(middle+1, last);
  }
}
```

**What work in the queue  
should other threads steal?**

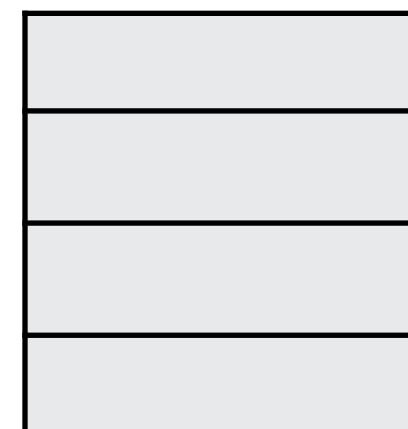
Thread 0 work queue



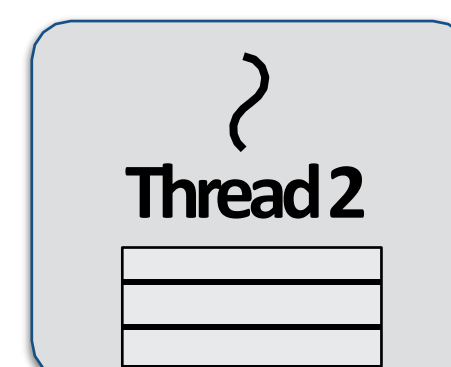
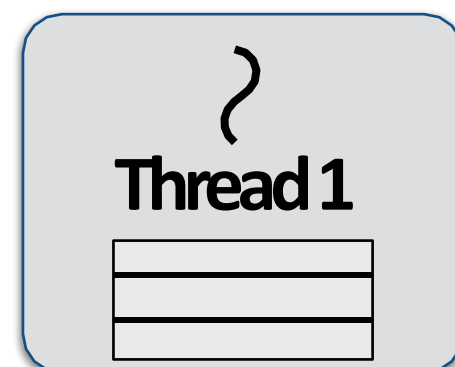
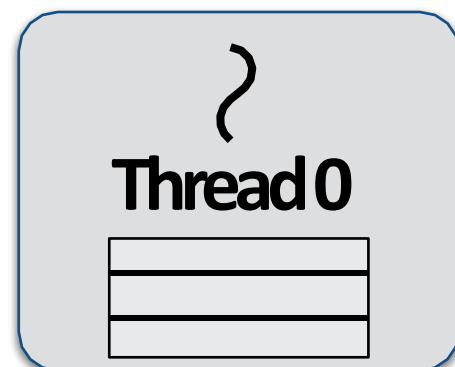
Thread 1 work queue



Thread 2 work queue



...

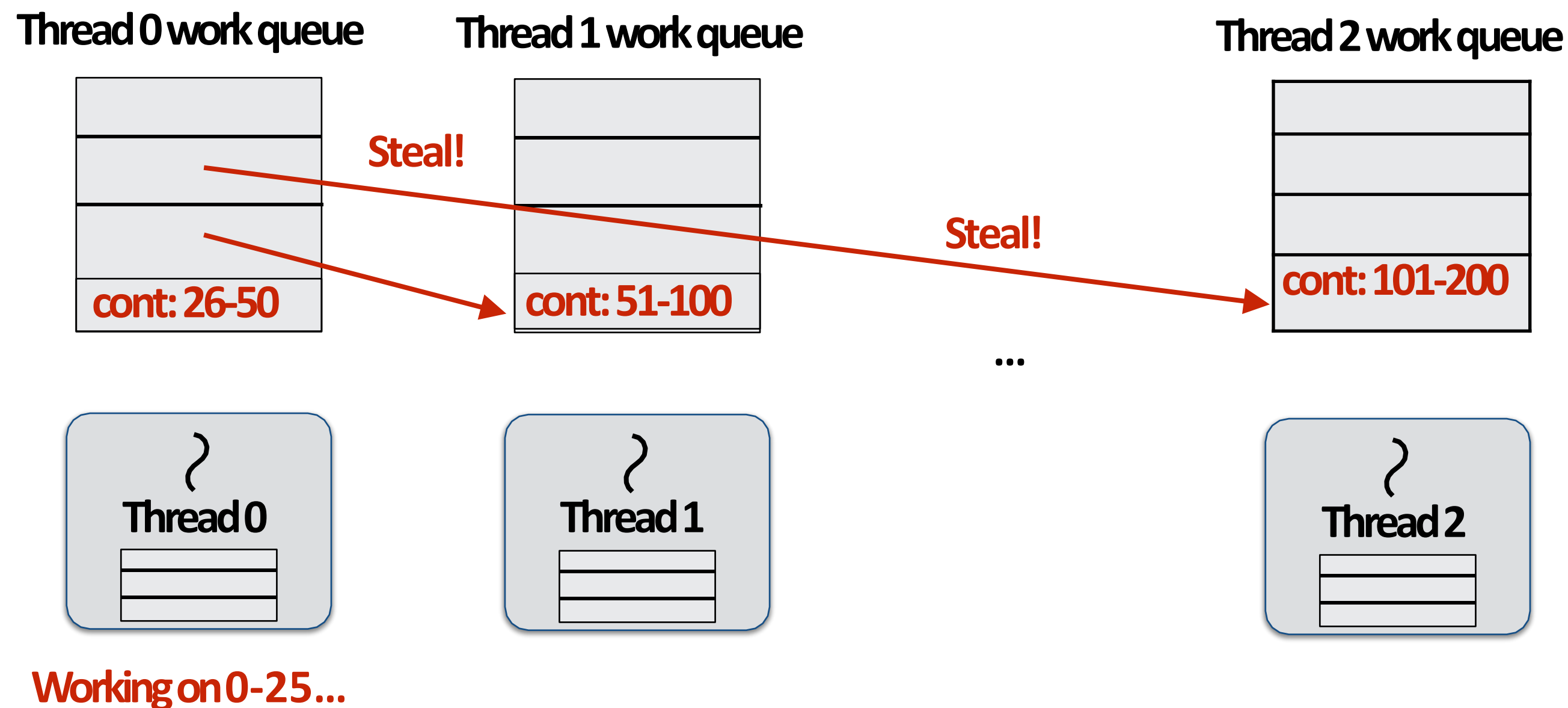


**Working on 0-25...**

# Implementing workstealing: dequeue per worker

Workqueue implemented as a **dequeue** (double ended queue)

- Local thread pushes/pops from the “tail” (bottom)
- Remote threads **steal from “head”** (top)
- Efficient lock-free dequeue implementations exist

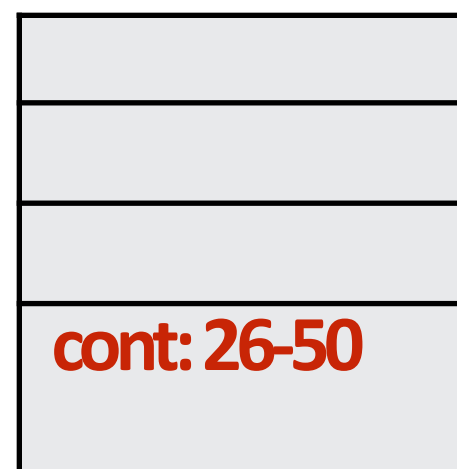


# Implementing workstealing: dequeue per worker

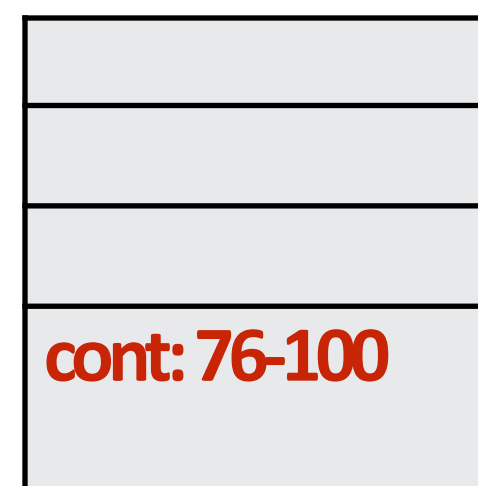
Workqueue implemented as a **dequeue** (double ended queue)

- Local thread pushes/pops from the “tail” (bottom)
- Remote threads **steal from “head”** (top)
- Efficient lock-free dequeue implementations exist

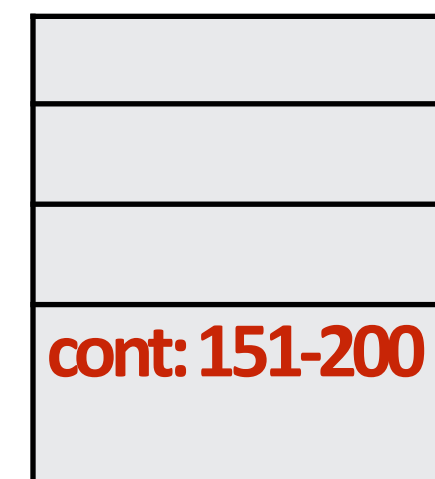
Thread 0 work queue



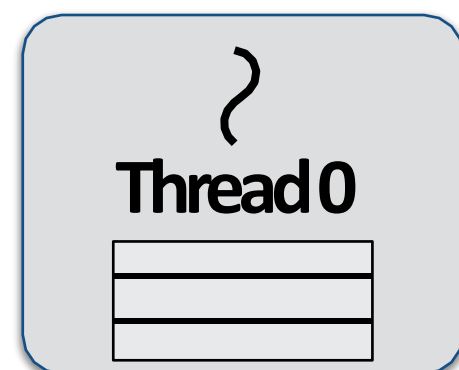
Thread 1 work queue



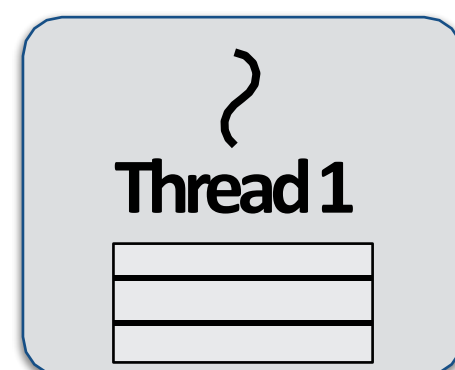
Thread 2 work queue



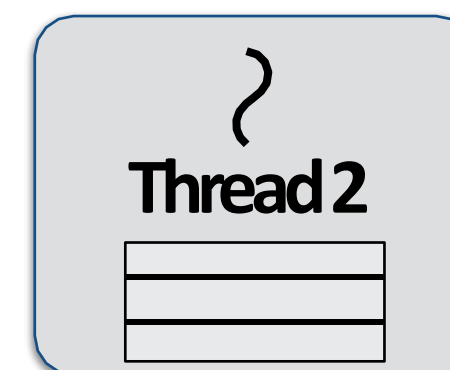
...



Working on 0-25...



Working on 51-75...

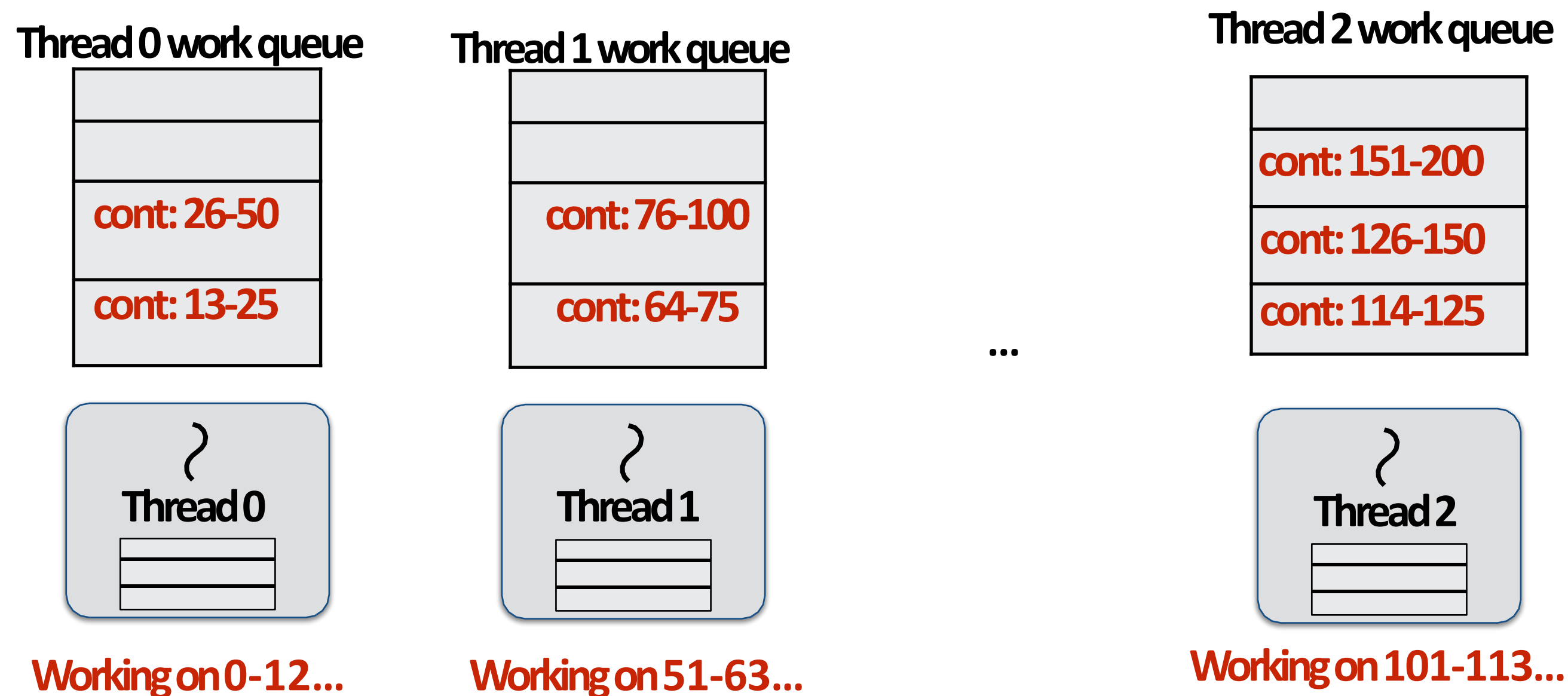


Working on 101-150...



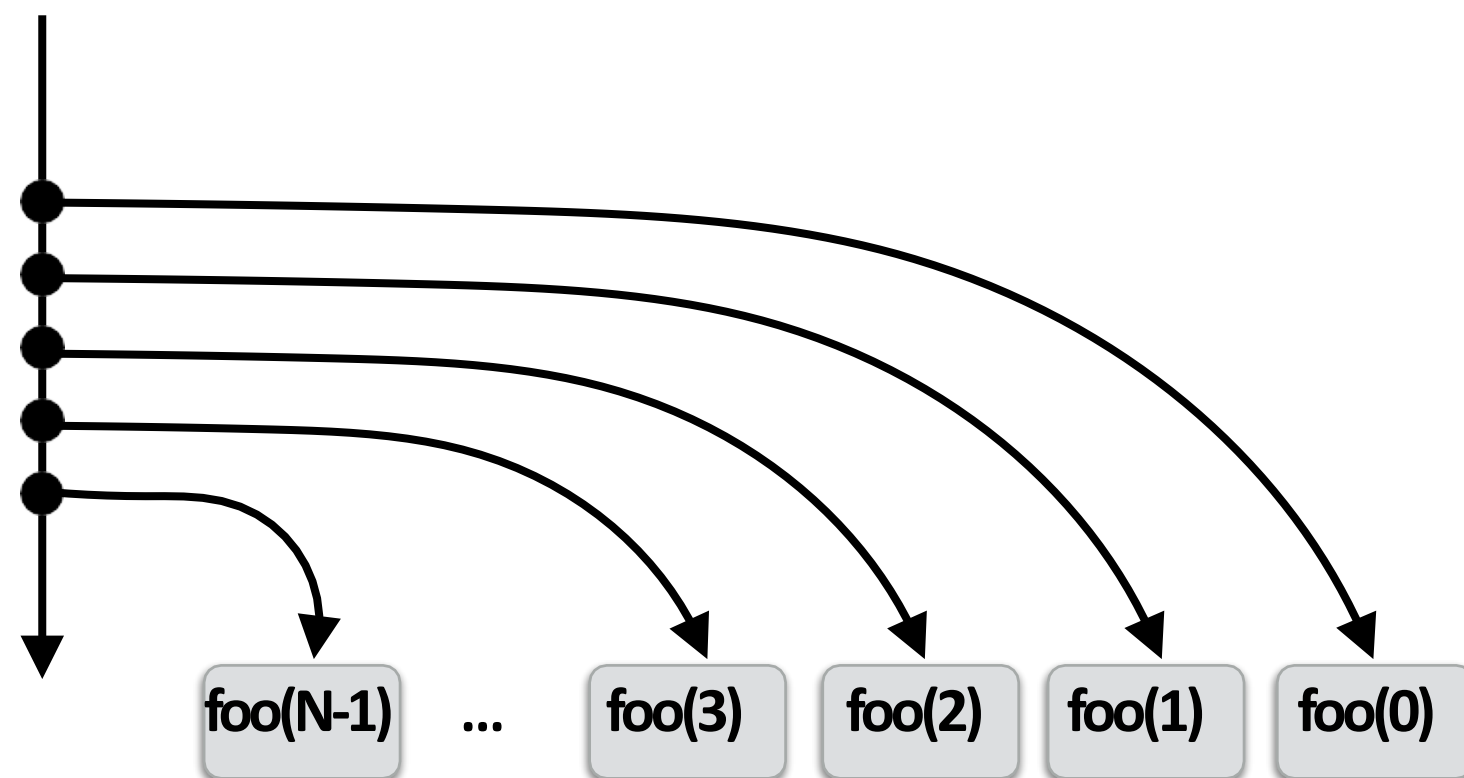
# Implementing workstealing: random choice of victim

- Idle threads randomly choose a thread to attempt to steal from
- Stealing from top of dequeue...
  - Reduces contention with local thread: local thread is not accessing same part of dequeue that stealing threads do!
  - Steals work at beginning of call tree: this is a “larger” piece of work, so the cost of performing a steal is amortized over longer future computation
  - Maximizes locality: (in conjunction with run-child-first policy) local thread works on local part of call tree



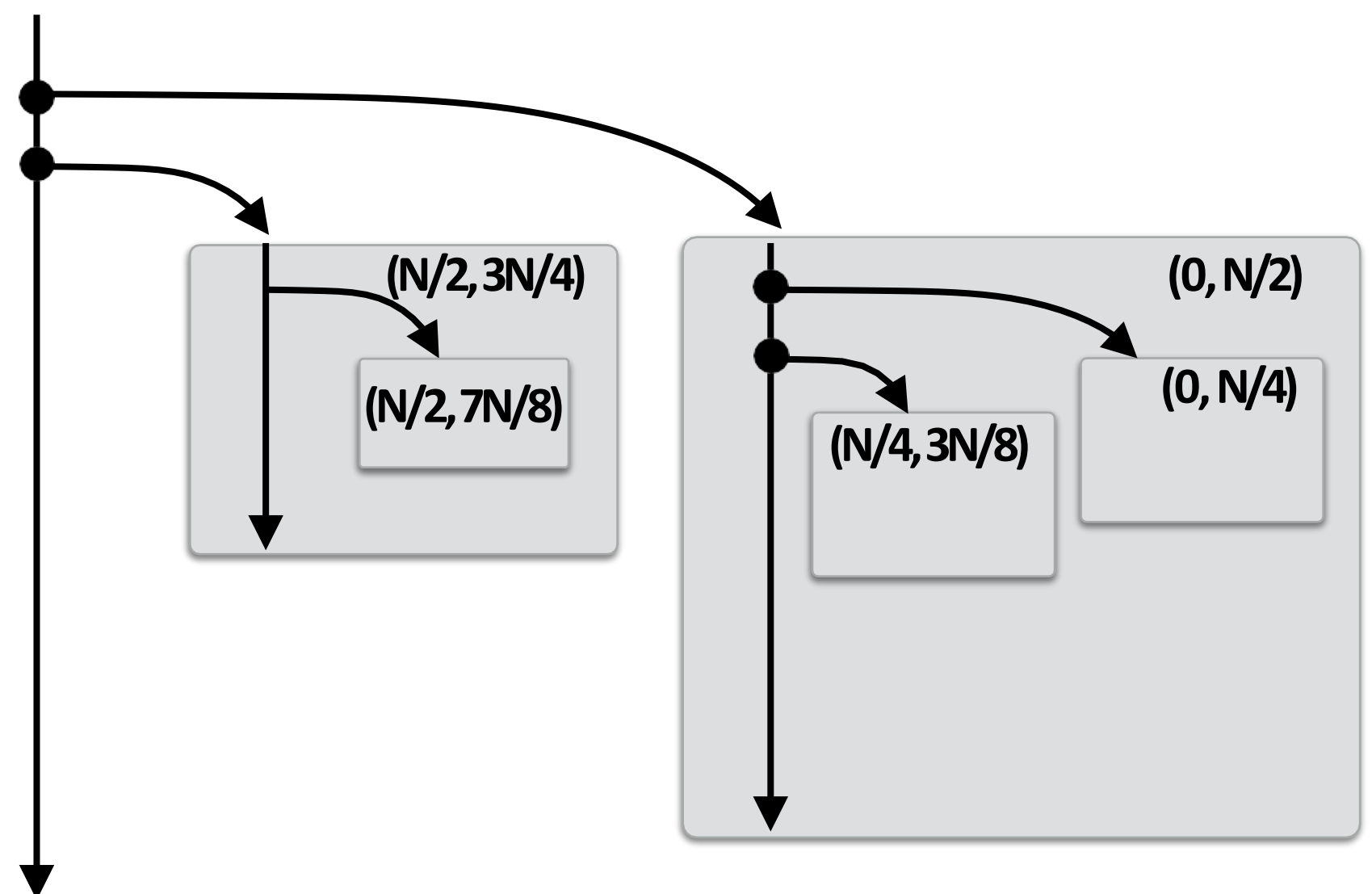
# Child-first work stealing scheduler anticipates divide-and-conquer parallelism

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



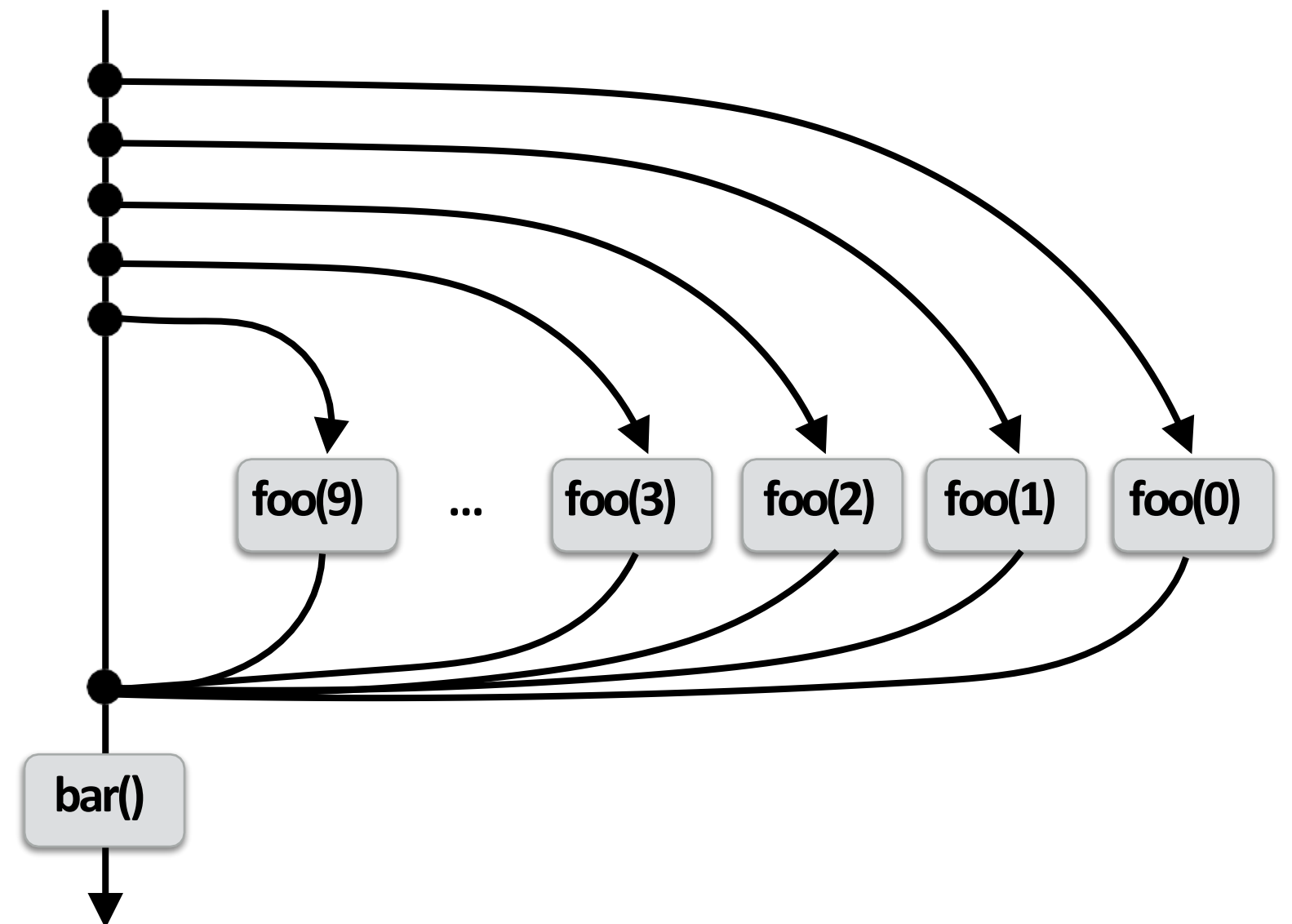
Code at right generates work in parallel, unlike the code at left (so it more quickly fills machine)

```
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);
```

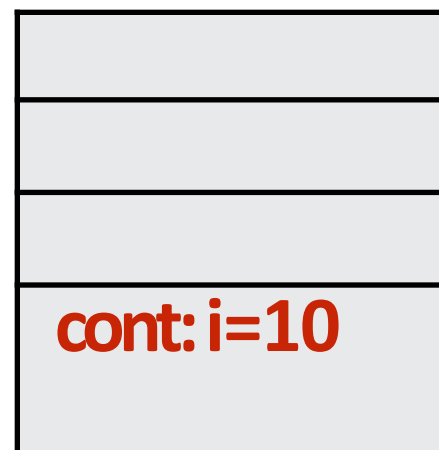


# Implementing sync

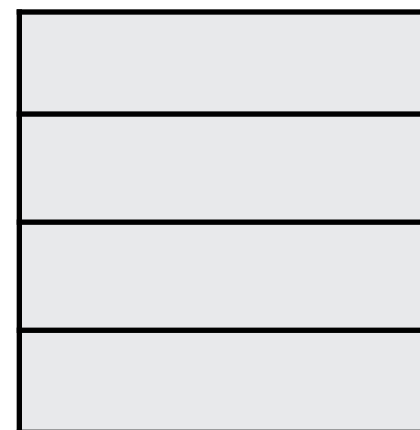
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
  
cilk_sync;  
bar();
```



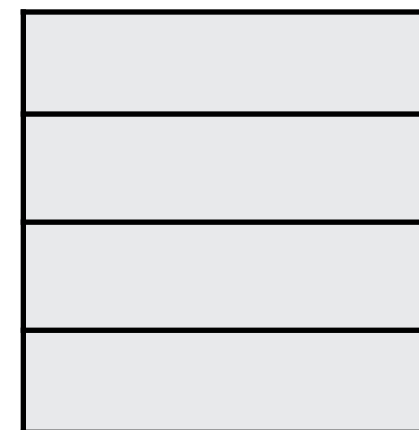
Thread 0 work queue



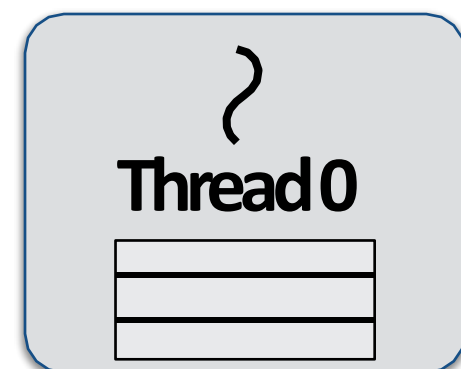
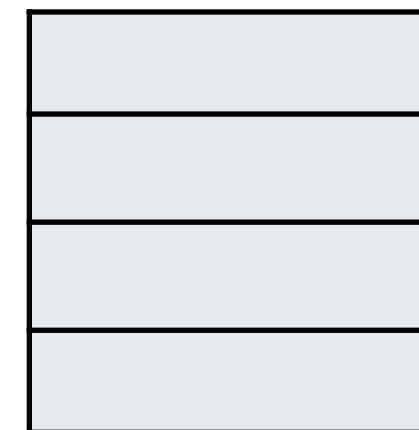
Thread 1 work queue



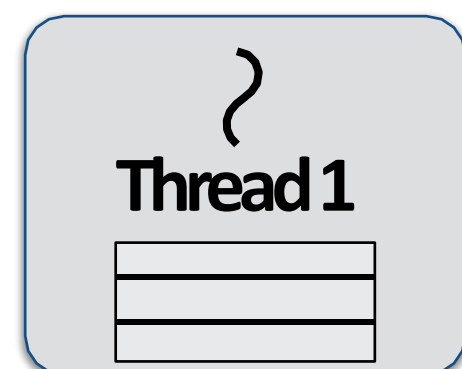
Thread 2 work queue



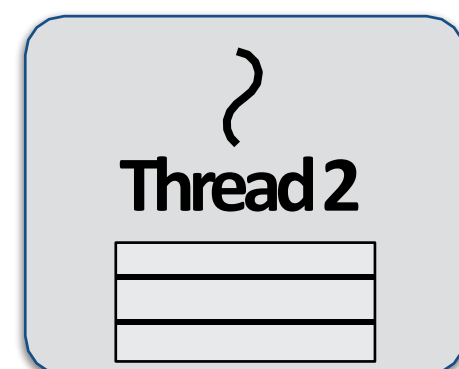
Thread 3 work queue



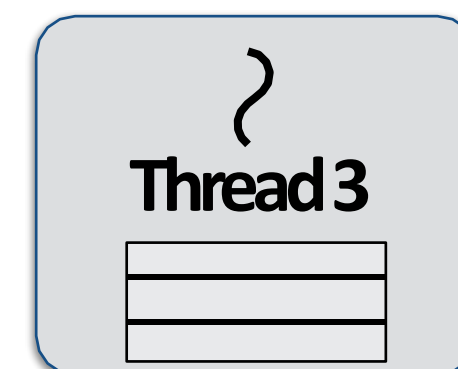
Working on foo(9)...



Working on foo(7)...



Working on foo(8)...



Working on foo(6)...

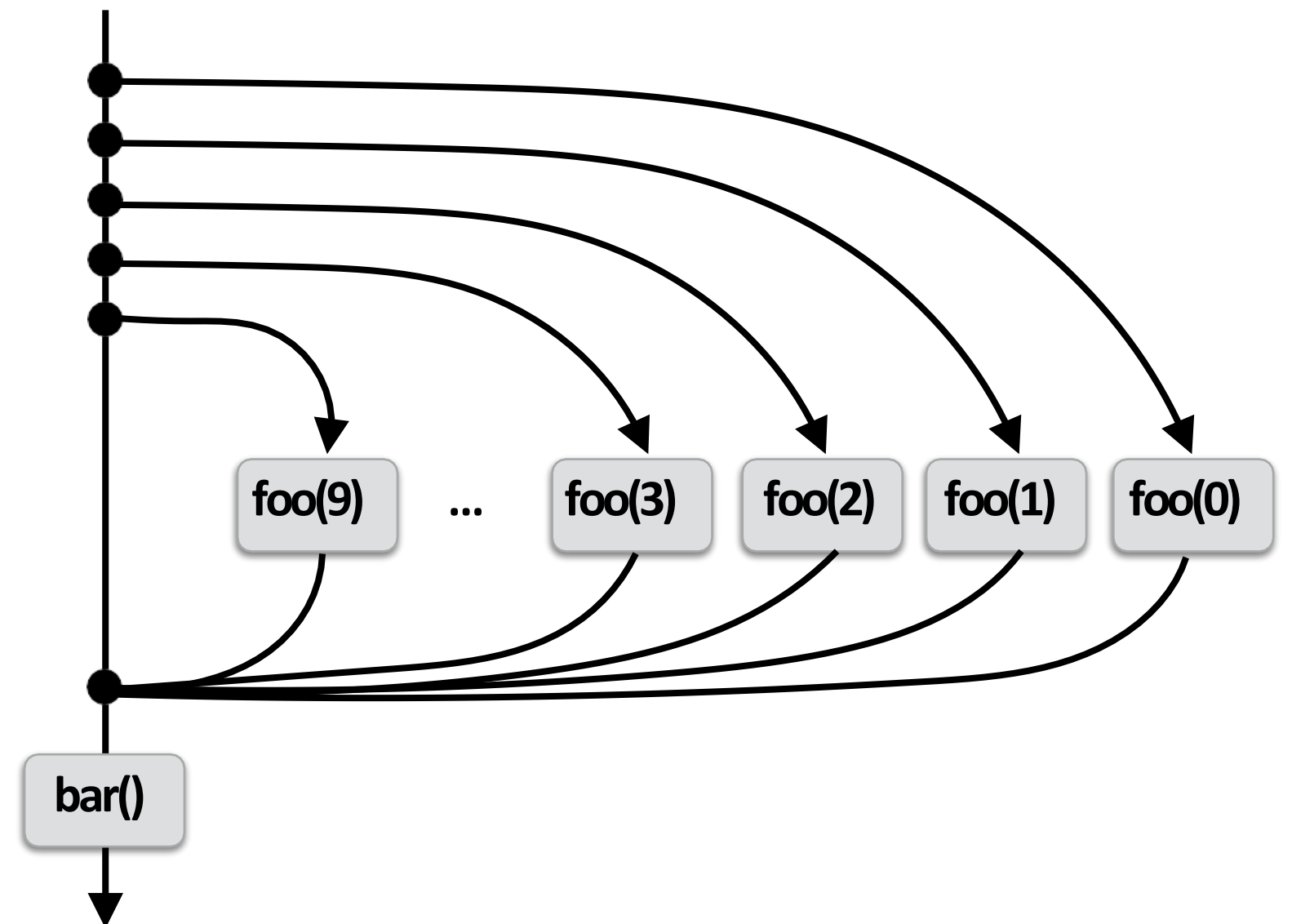
State of worker threads  
when all work from loop  
is nearly complete



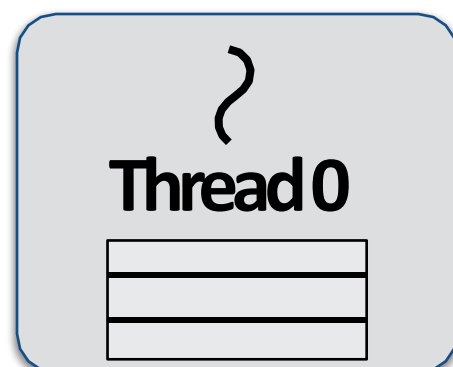
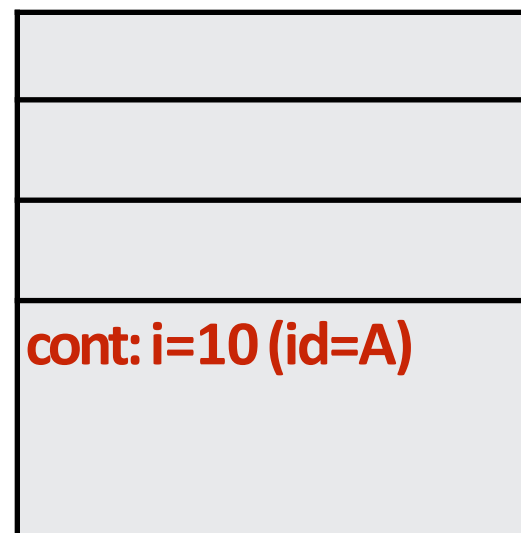
# Implementing sync: **no stealing**

**block (id: A)**

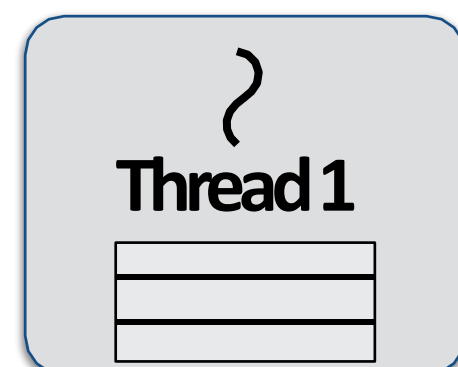
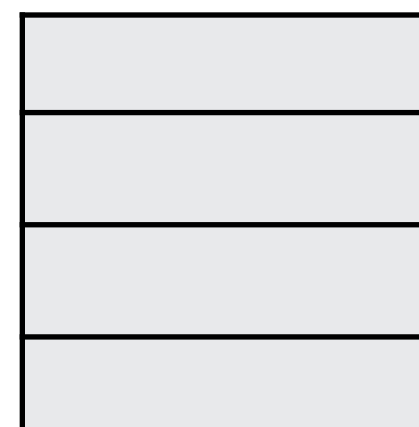
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



Thread 0 work queue



Thread 1 work queue



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.

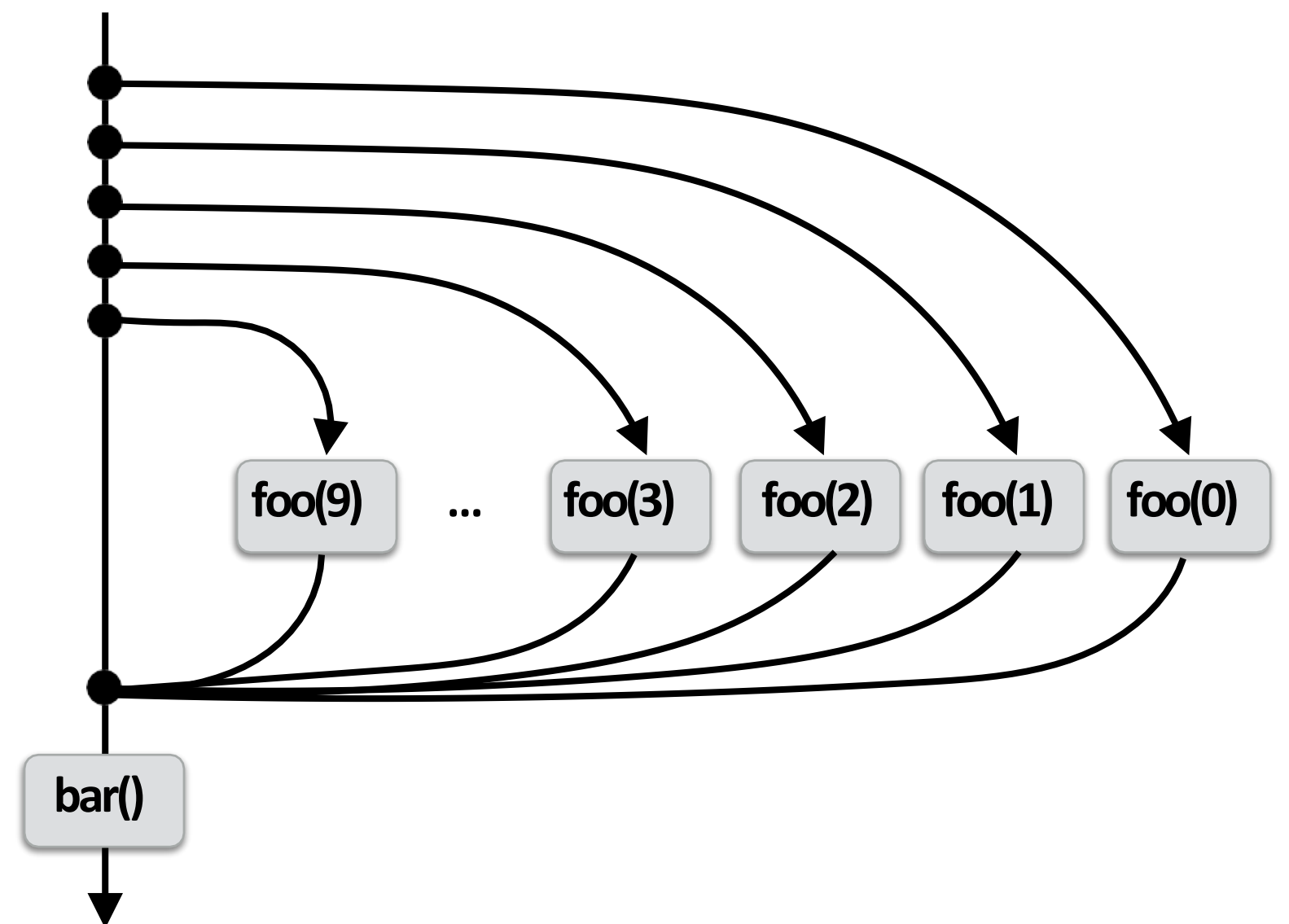
**Same control flow as serial execution**

**Working on foo(9), i d=A...**

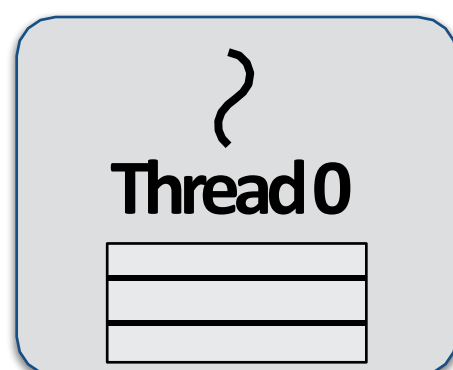
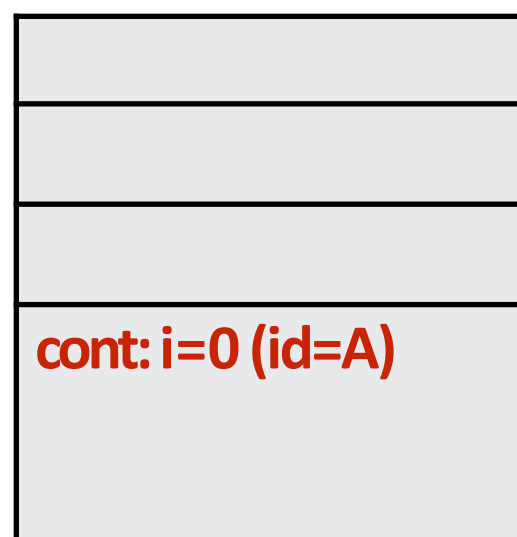
# Implementing sync: stalling join

**block (id: A)**

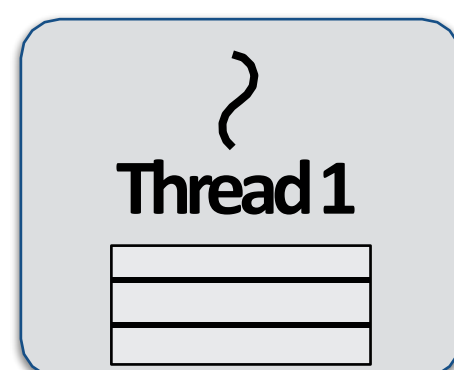
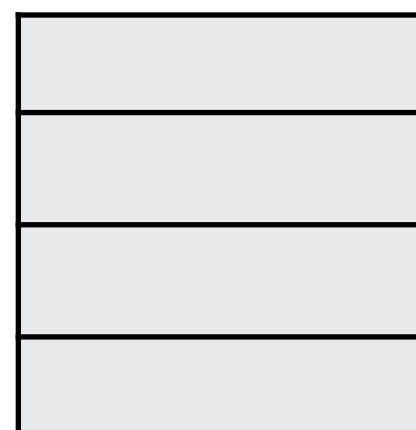
```
[ for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();  
]
```



Thread 0 work queue



Thread 1 work queue



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

## Example 1: “stalling” join policy

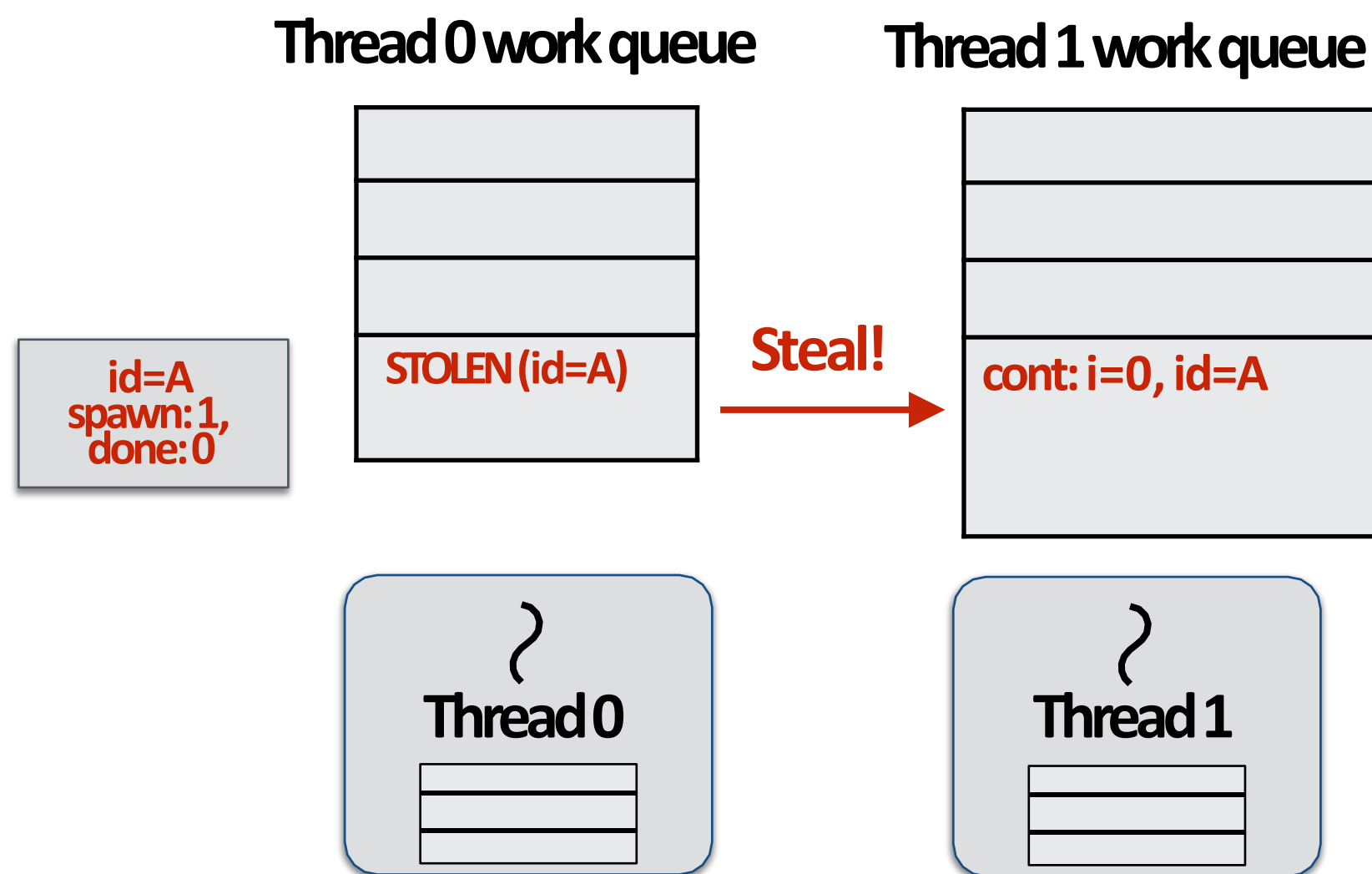
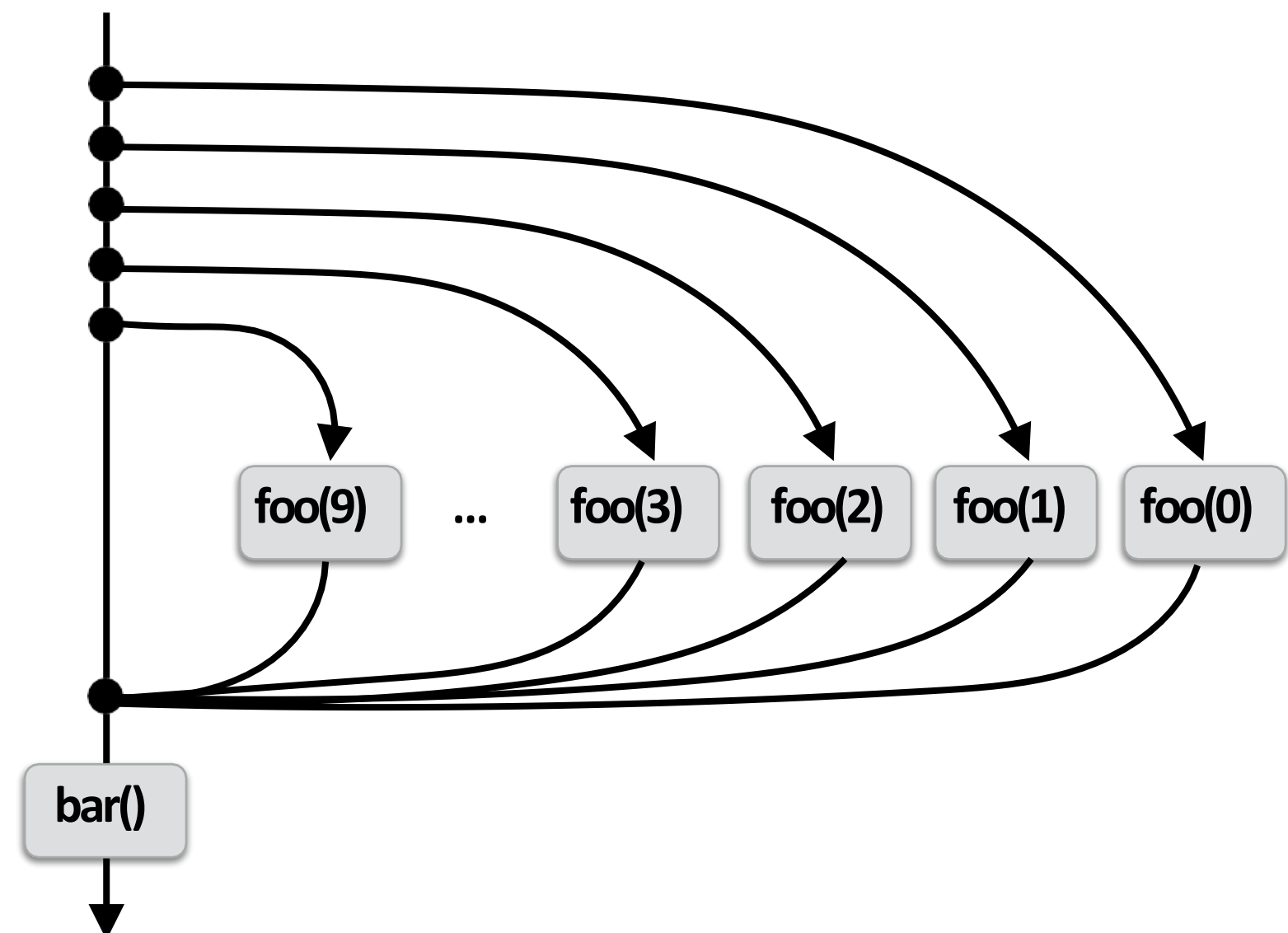
**Thread that initiates the fork must perform the sync.**

Therefore it waits for all spawned work to be complete. In this case, thread 0 is the thread initiating the fork

# Implementing sync: stalling join

**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();
```



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

**Idle thread 1 steals from busy thread 0**  
**Note: descriptor for block A created**

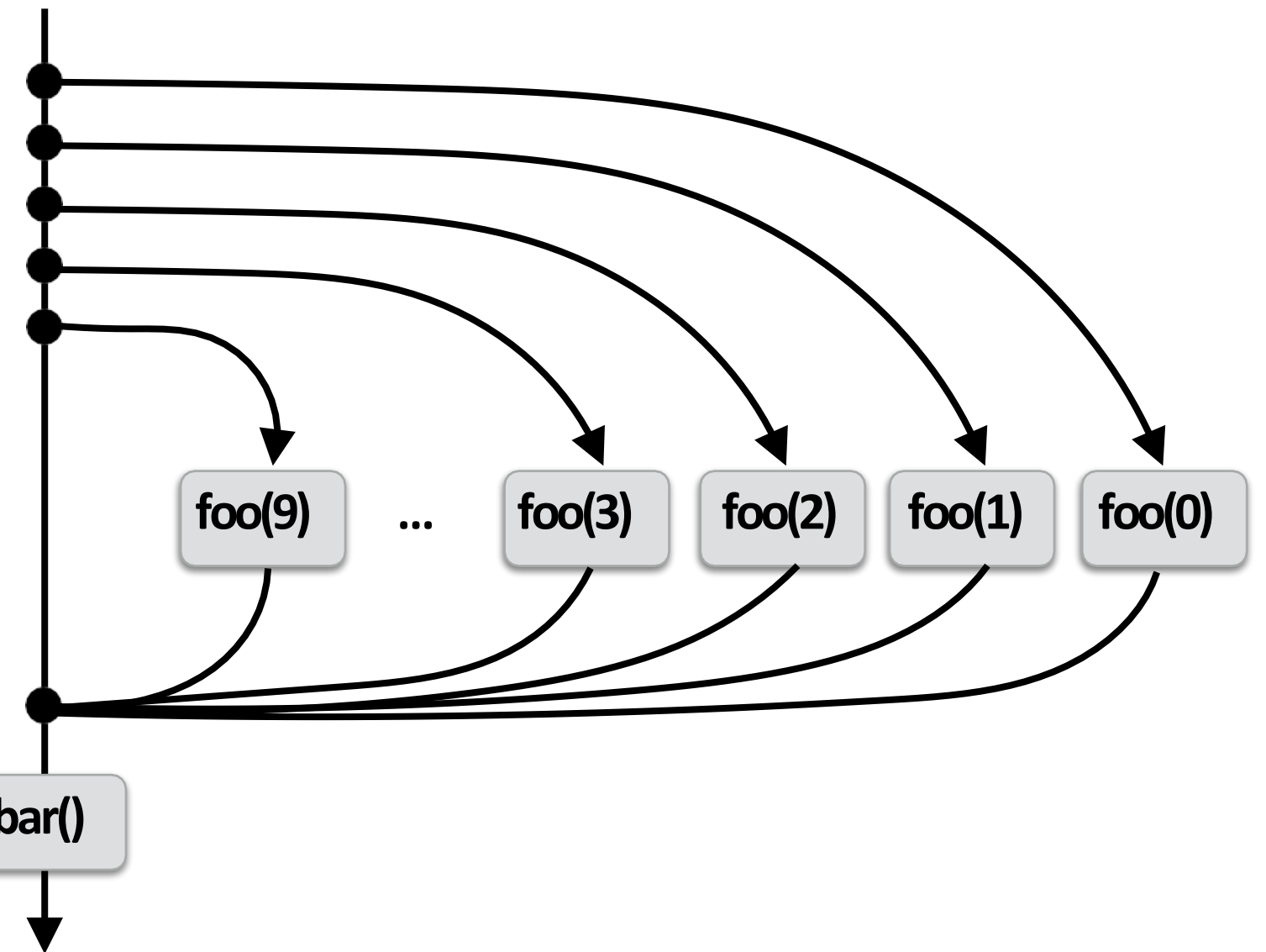
The descriptor tracks the number of outstanding spawns for the block, and the number of those spawns that have completed.

Here, the 1 spawn corresponds to foo(0) being run by thread 0.

# Implementing sync: stalling join

**block (id: A)**

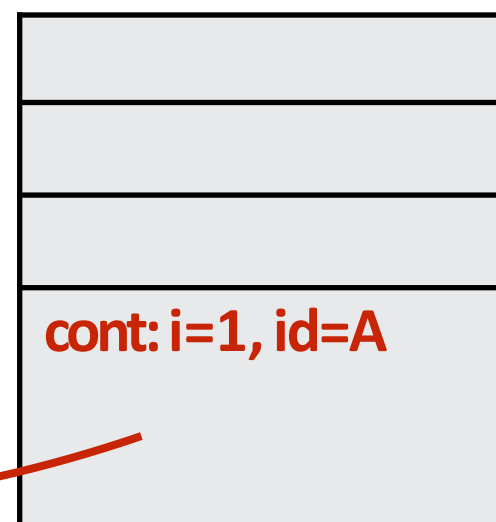
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



Thread 0 work queue



Thread 1 work queue

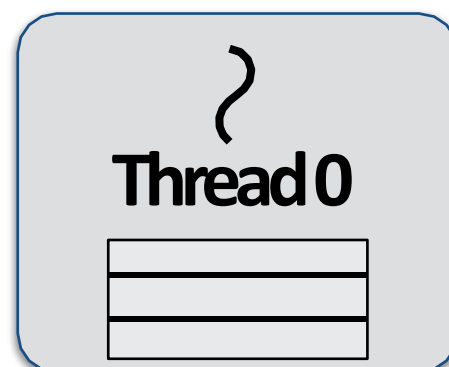


**Thread 1 is now running `foo(1)`**

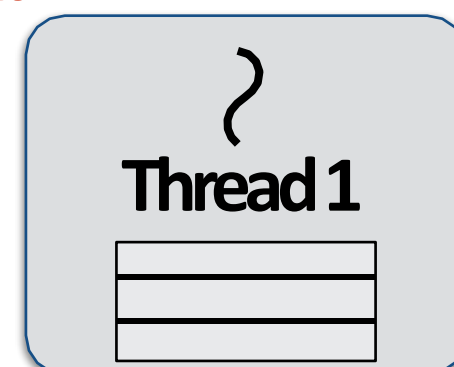
**Note: spawn count is now 2**

**id=A  
spawn: 2,  
done: 0**

**Update count**



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

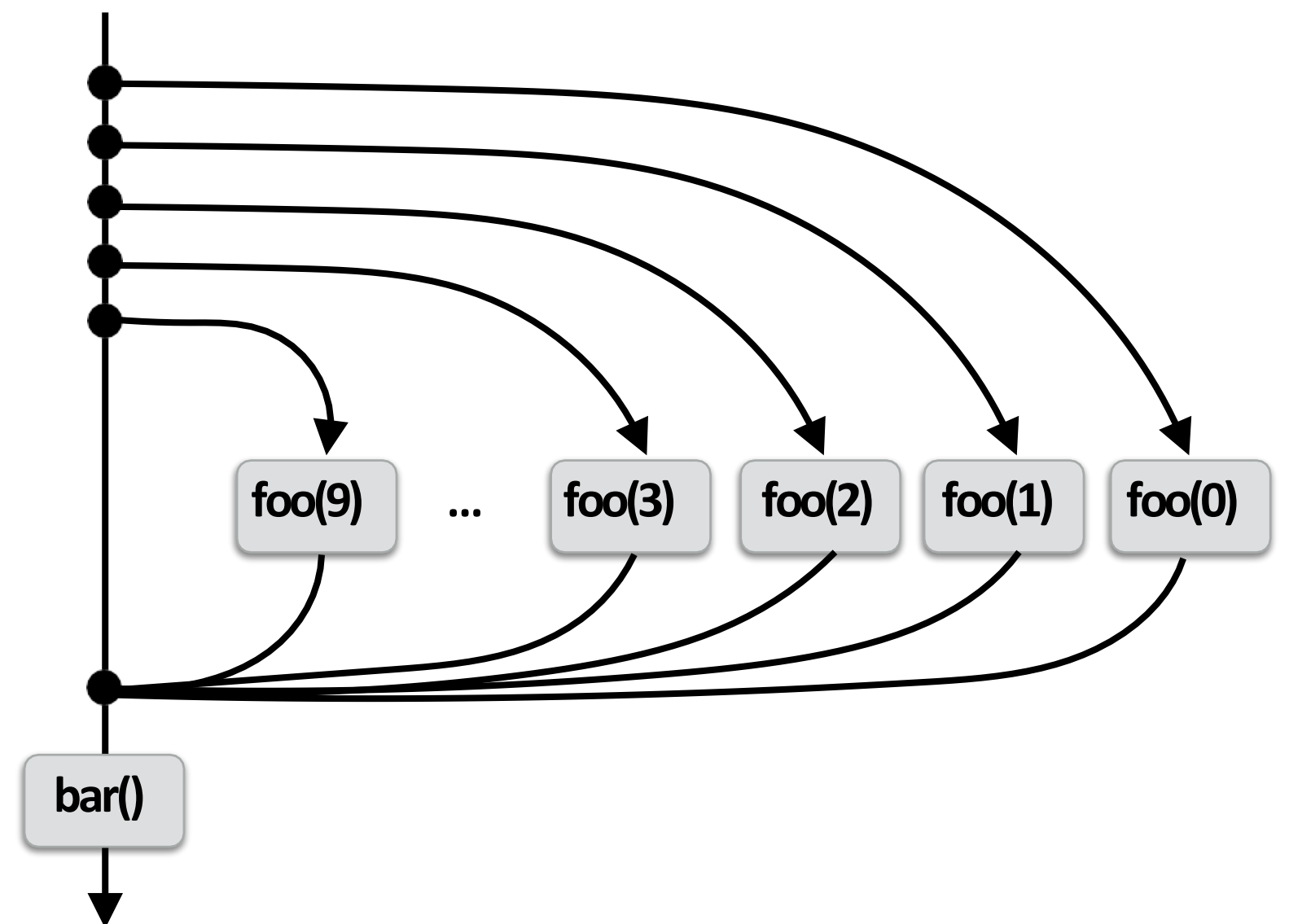


**Working on `foo(1)`, id=A...**

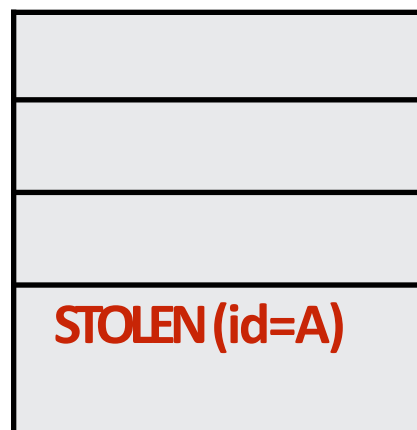
# Implementing sync: stalling join

**block (id: A)**

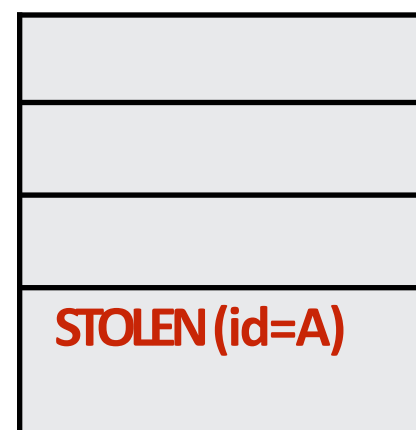
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



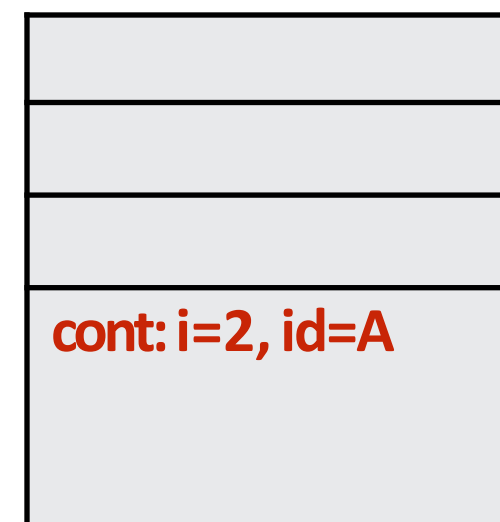
Thread 0 work queue



Thread 1 work queue



Thread 2 work queue

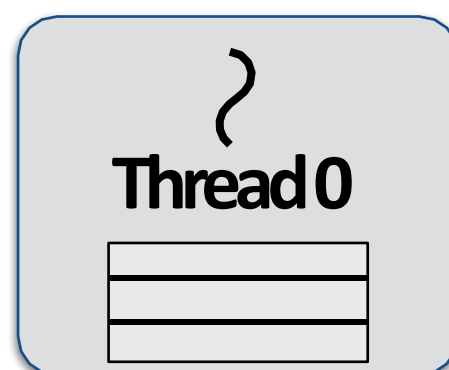


**Steal!**

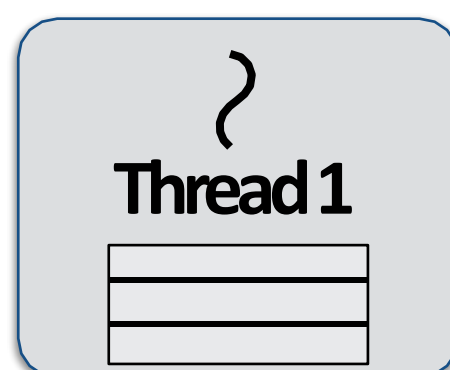
Thread 0 completes foo(0)  
(updates spawn descriptor)

Thread 2 now running foo(2)

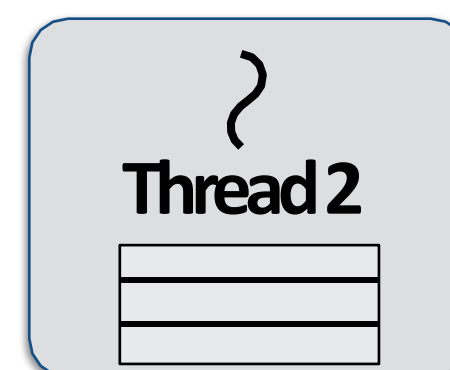
id=A  
spawn:3,  
done:0



Idle!



Working on foo(1), id=A...



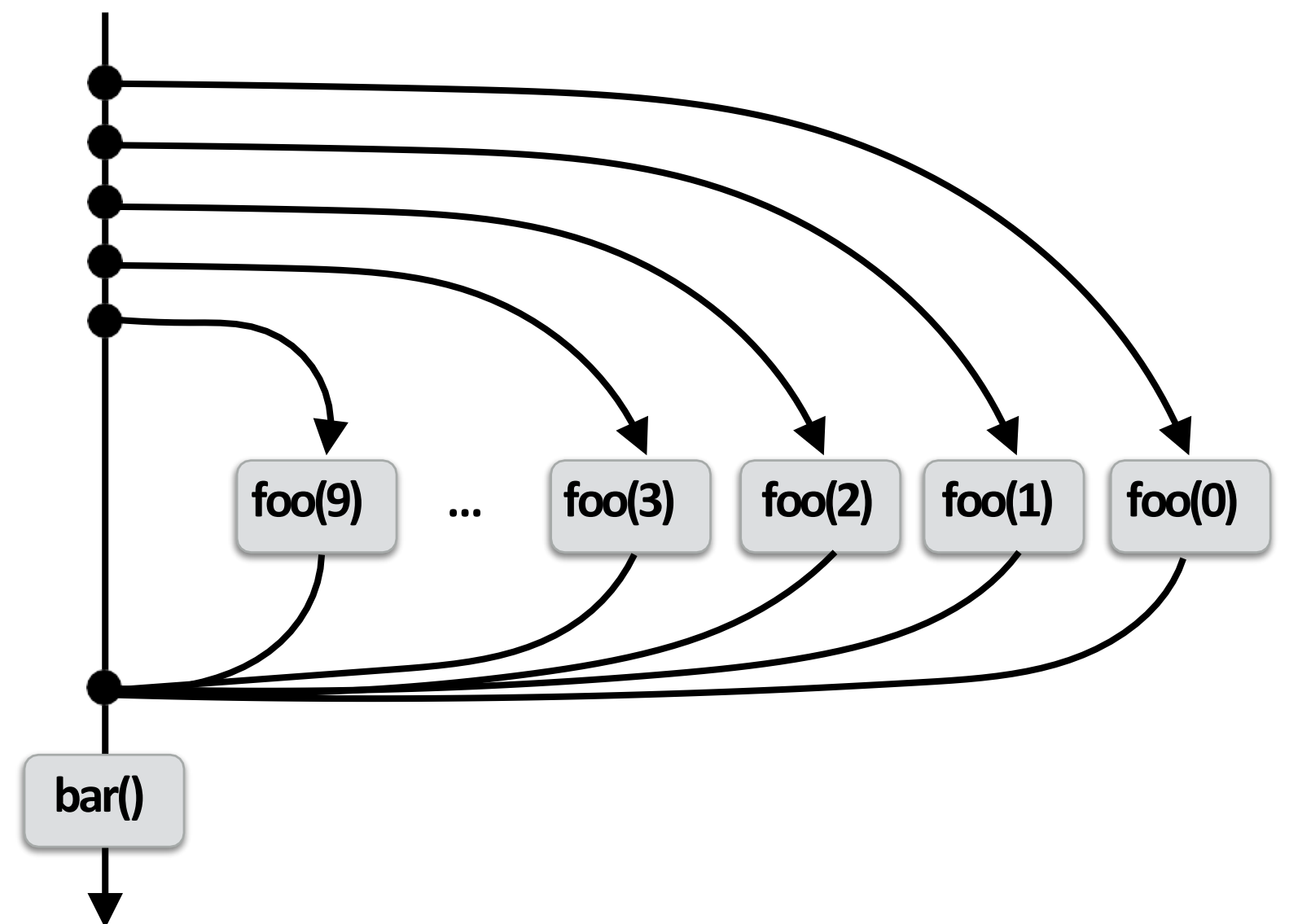
Working on foo(2), id=A...



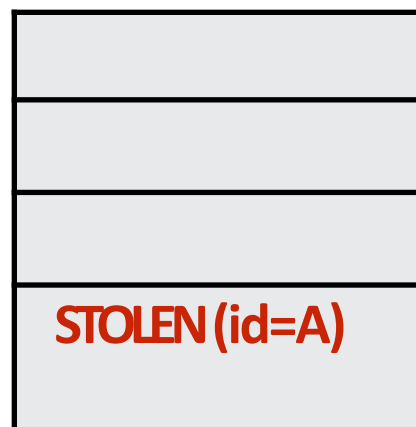
# Implementing sync: stalling join

**block (id: A)**

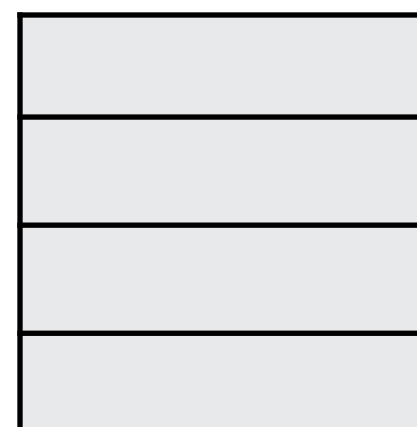
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



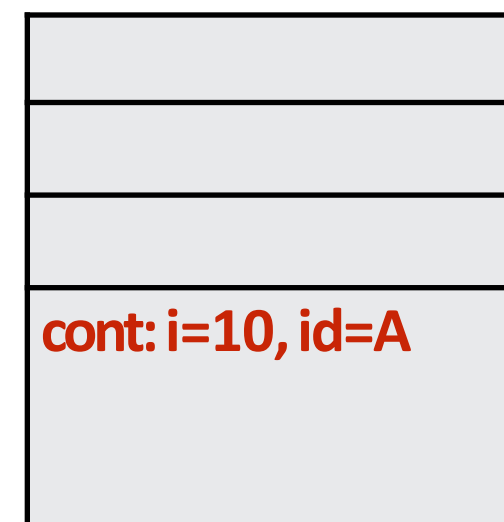
Thread 0 work queue



Thread 1 work queue



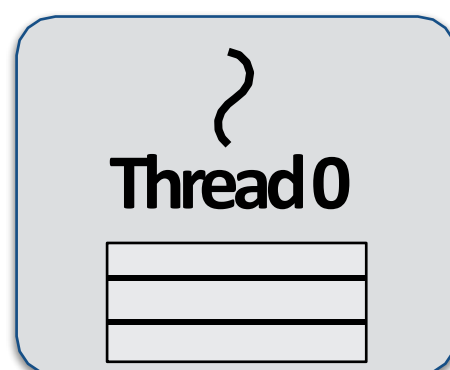
Thread 2 work queue



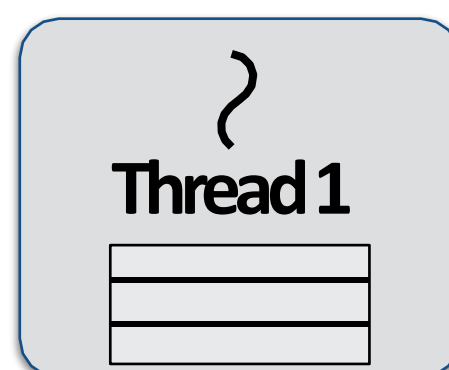
Computation nearing end...

Only foo(9) remains to be completed.

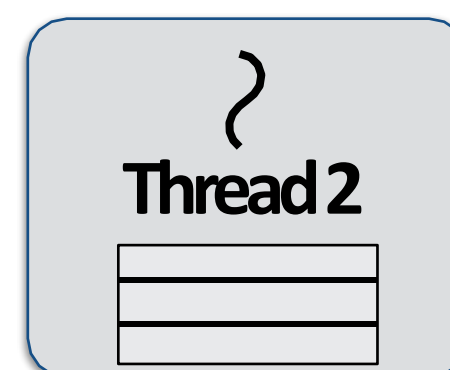
id=A  
spawn:10,  
done:9



Idle!



Idle!

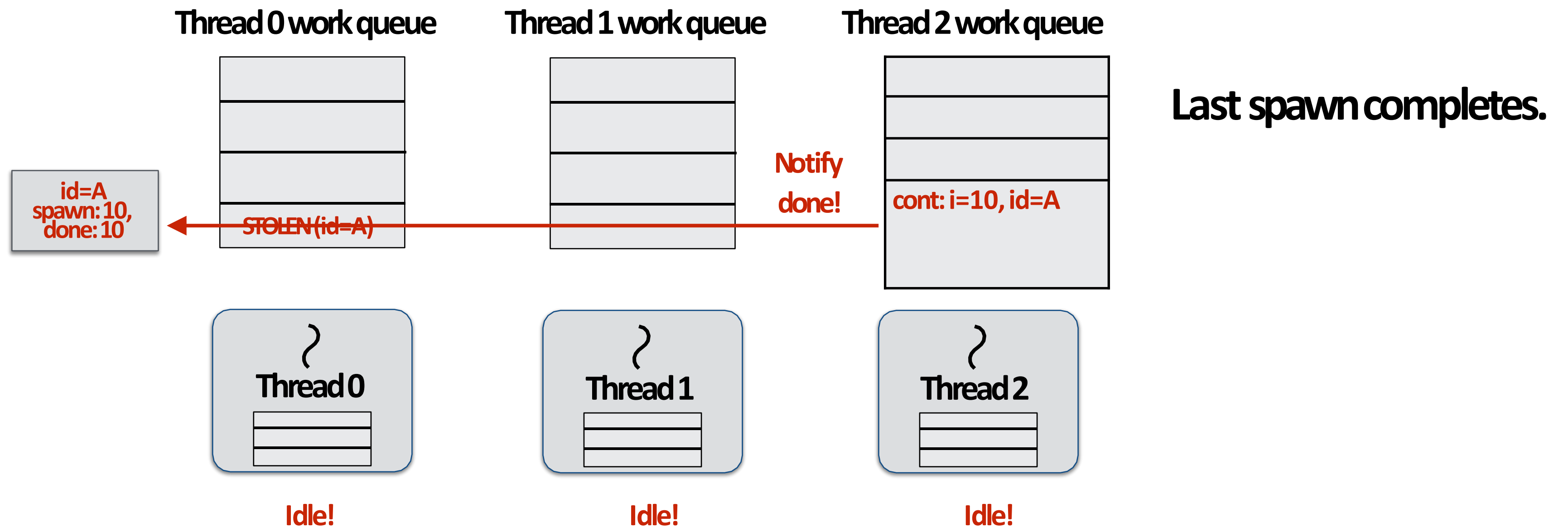
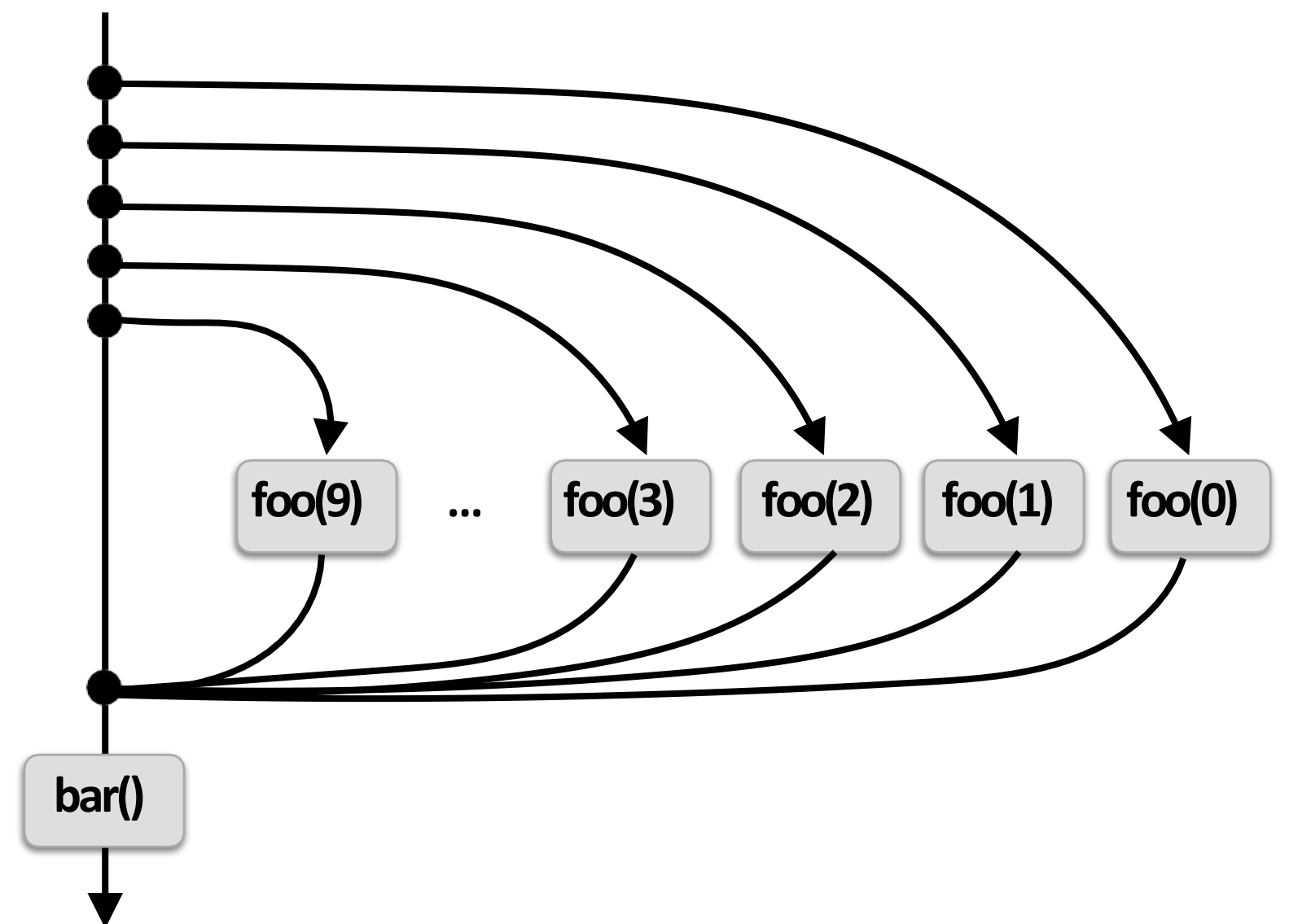


Working on foo(9), id=A...

# Implementing sync: stalling join

**block (id: A)**

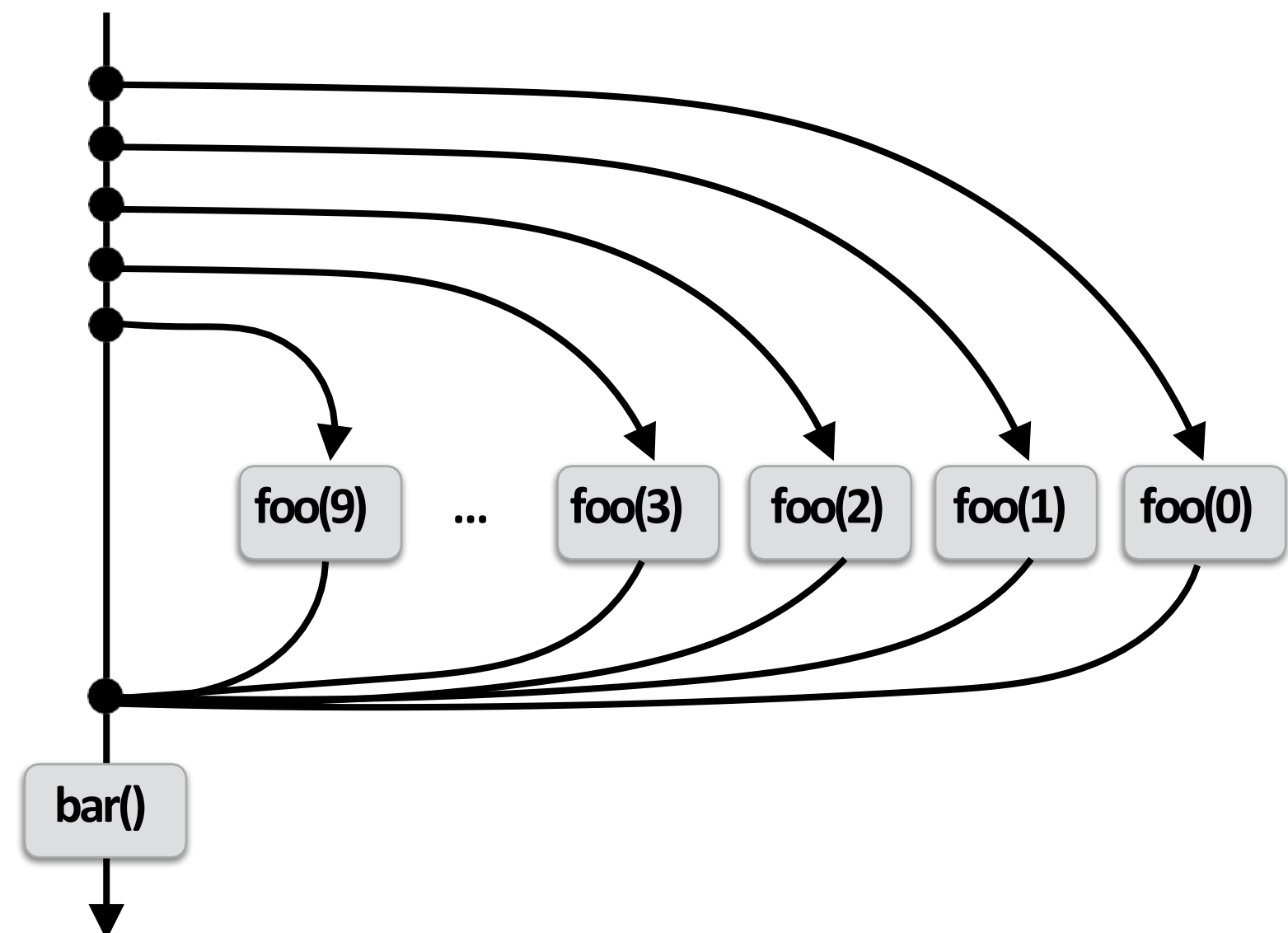
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



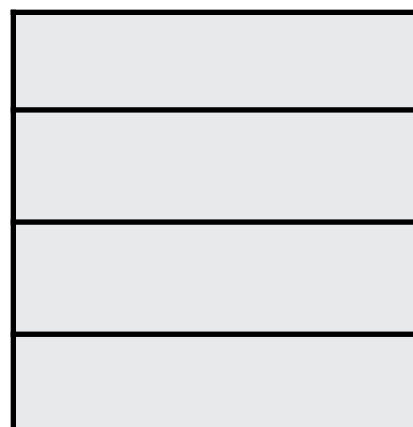
# Implementing sync: stalling join

block (id: A)

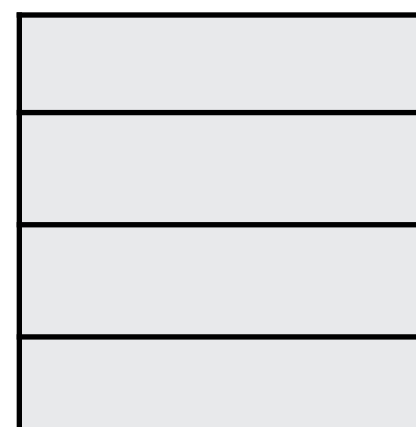
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



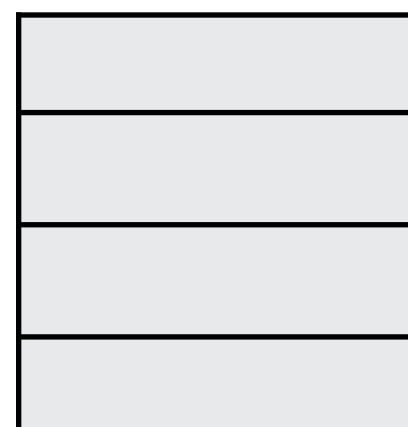
Thread 0 work queue



Thread 1 work queue

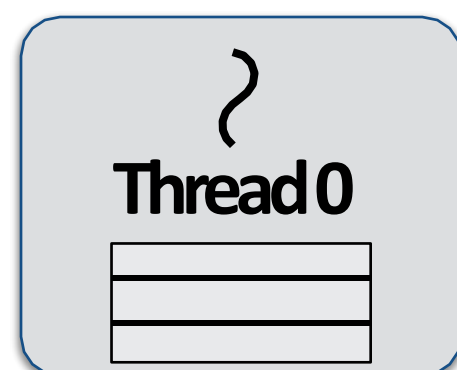


Thread 2 work queue

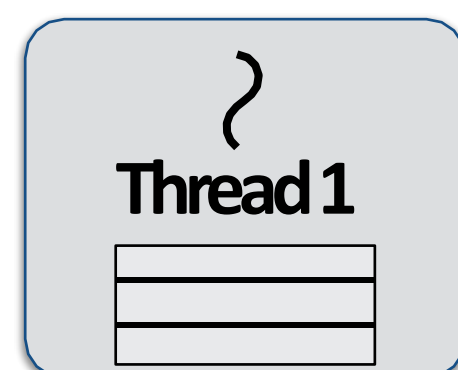


Thread 0 now resumes continuation  
and executes `bar()`

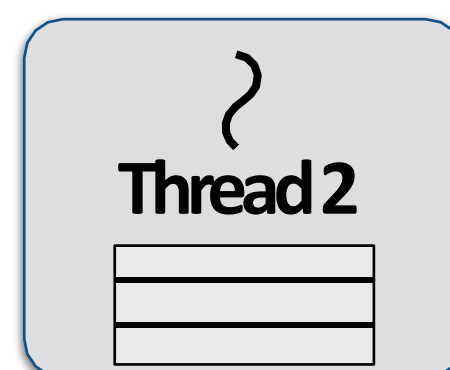
Note block A descriptor is now free.



Working on `bar()`...



Idle!



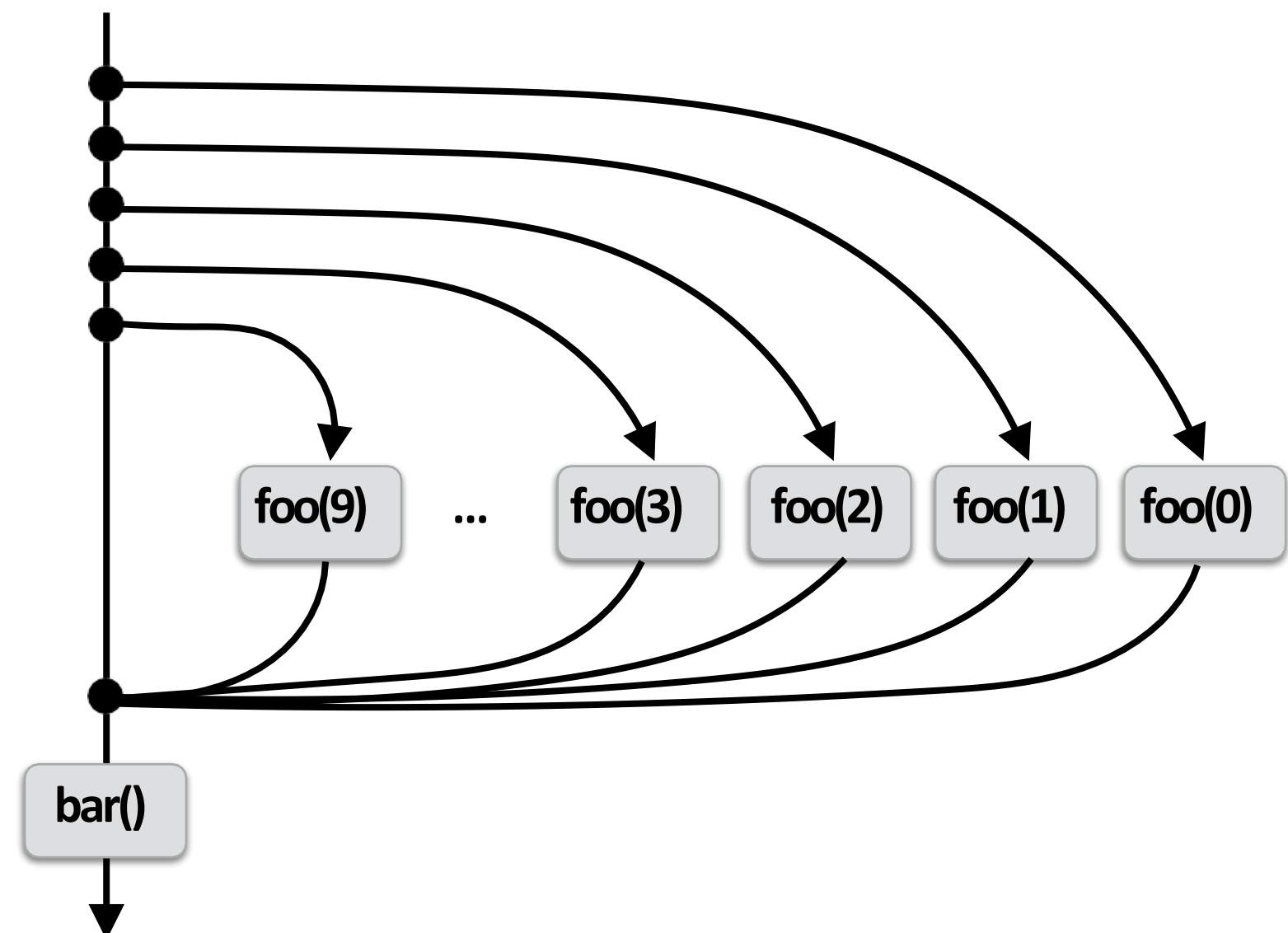
Idle!

*With this approach,  
thread that starts block  
will also complete it*

# Implementing sync: greedy policy

**block (id: A)**

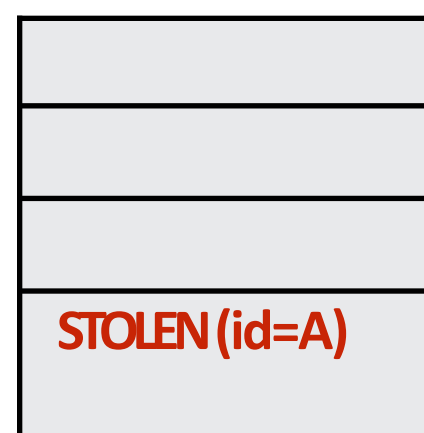
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



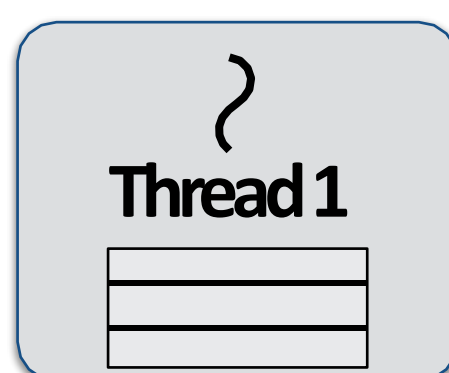
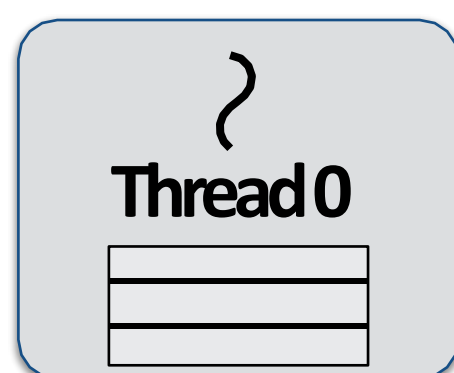
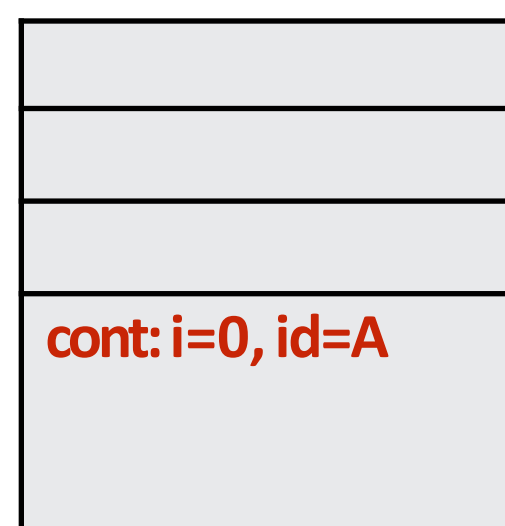
Thread 0 work queue

Thread 1 work queue

id=A  
spawn:0,  
done:0



**Steal!**



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

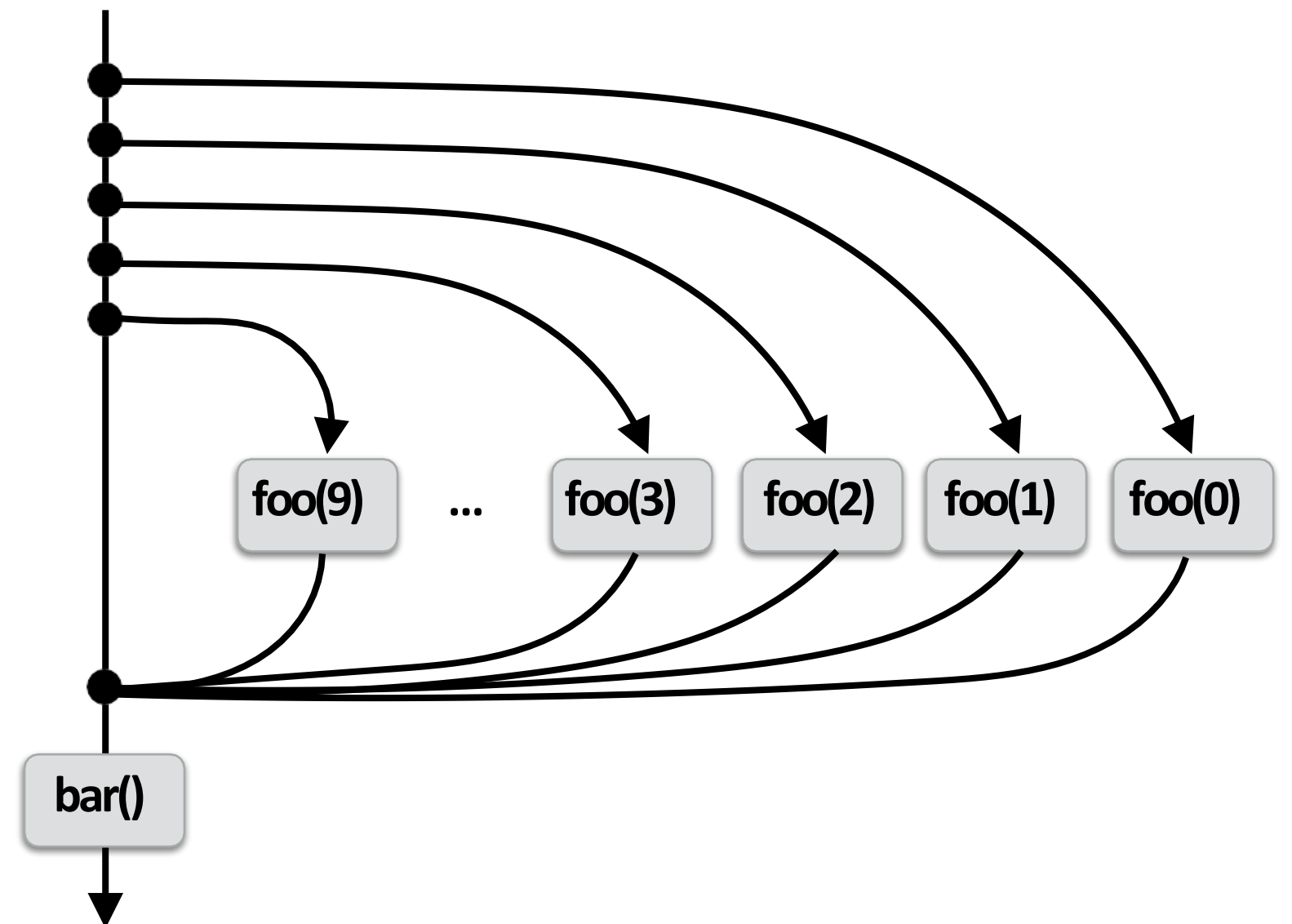
## Example 2: “greedy” policy

- When thread that initiates the fork goes idle, it looks to steal new work
- Last thread to reach the join point continues execution after sync

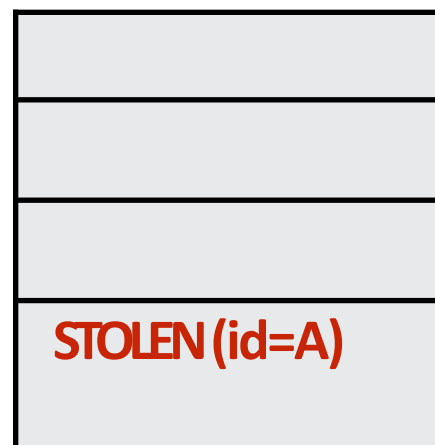
# Implementing sync: greedy policy

**block (id: A)**

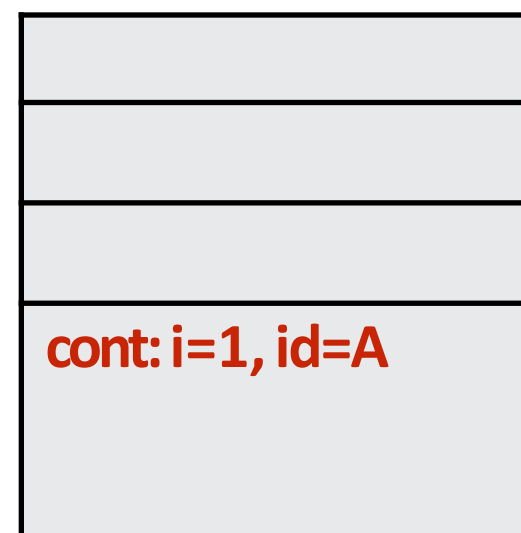
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



Thread 0 work queue

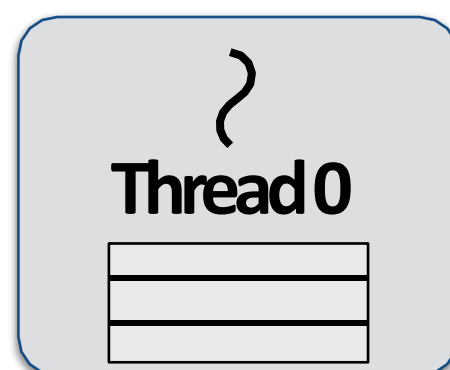


Thread 1 work queue

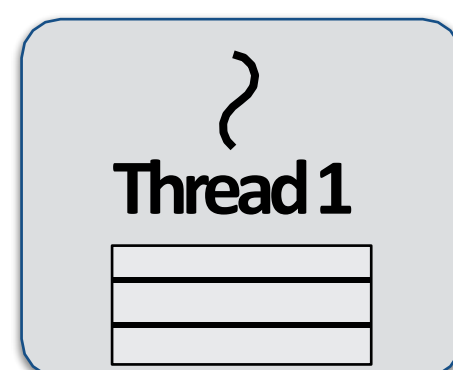


**Idle thread 1 steals from busy thread 0  
(as in the previous case)**

**id=A  
spawn: 2,  
done: 0**



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

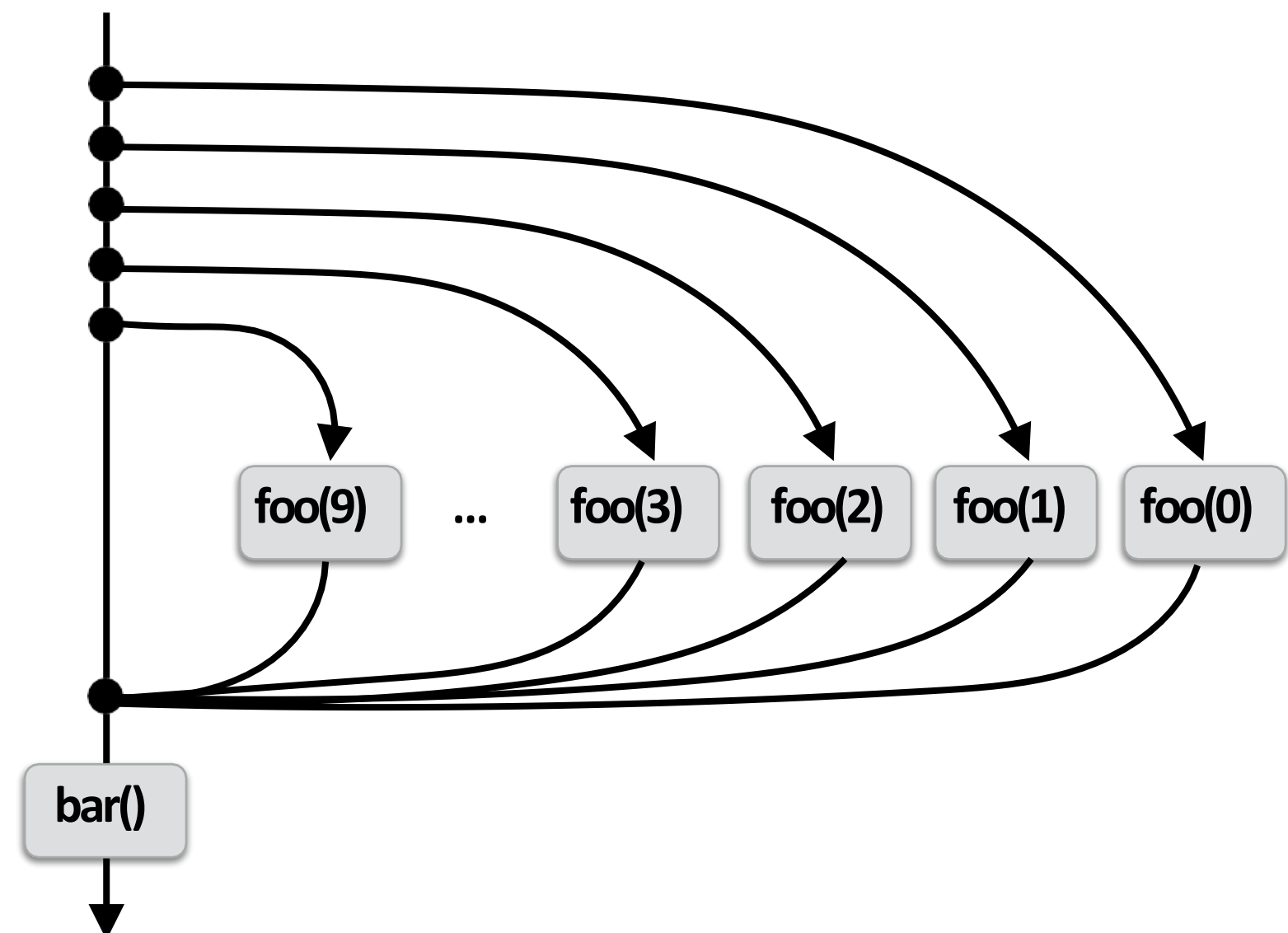


**Working on foo(1), id=A...**

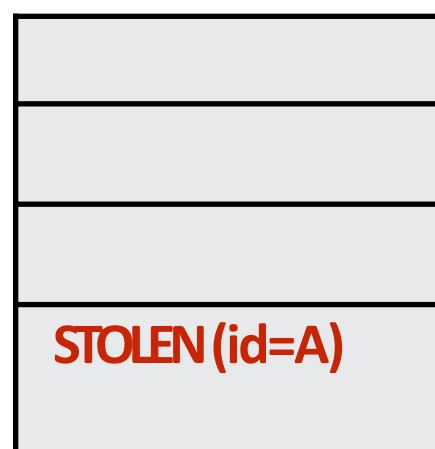
# Implementing sync: greedy policy

**block (id: A)**

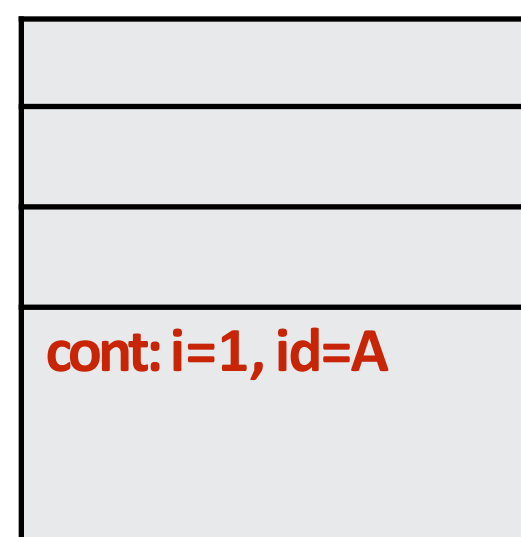
```
[ for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();  
]
```



Thread 0 work queue

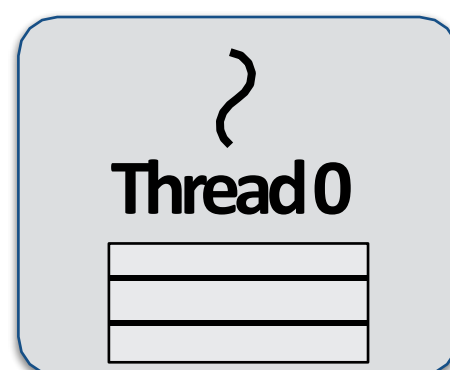


Thread 1 work queue

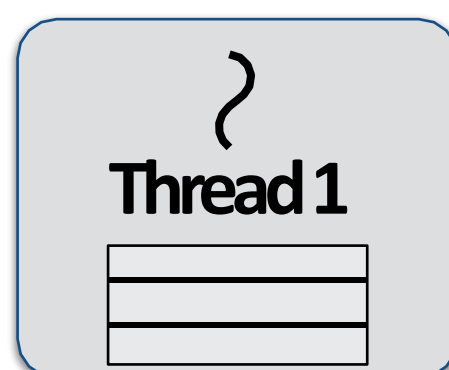


**Thread 0 completes foo(0)**  
**No work to do in local dequeue, so thread 0**  
**looks to steal!**

**id=A**  
**spawn: 2,**  
**done: 0**



**Done with foo(0)!**

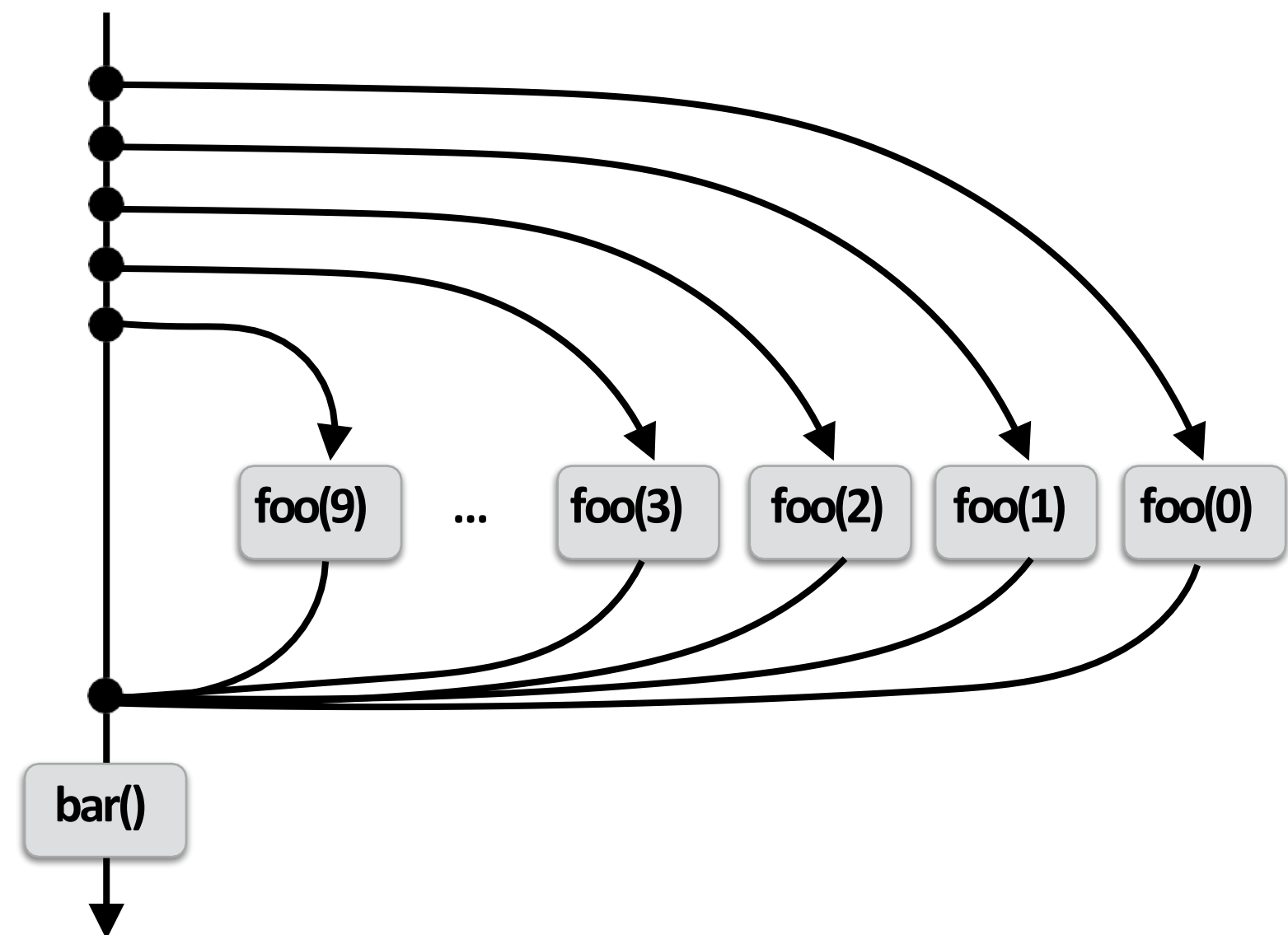


**Working on foo(1), id=A...**

# Implementing sync: greedy policy

**block (id: A)**

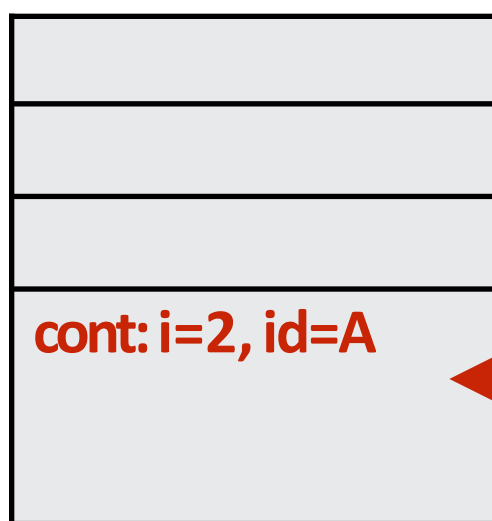
```
for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();
```



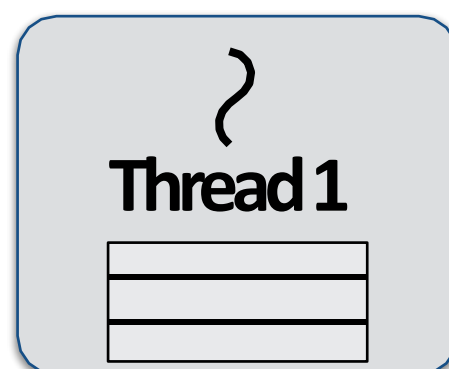
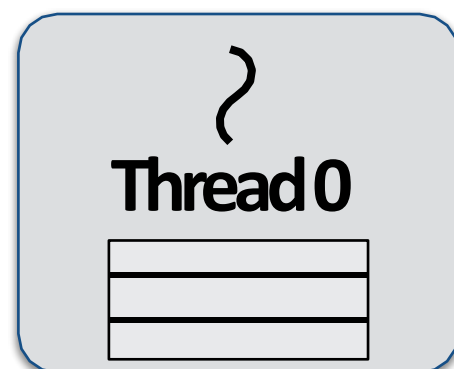
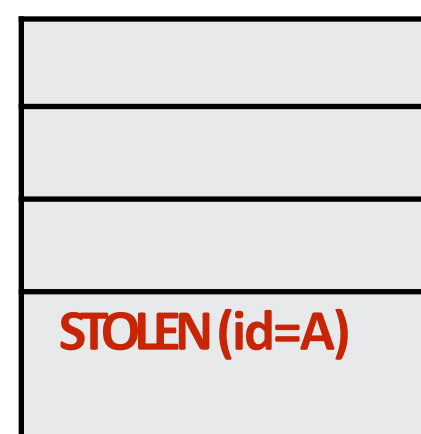
Thread 0 work queue

Thread 1 work queue

id=A  
spawn:3,  
done:1



**Steal!**



Working on `foo(2)`, `id=A`...

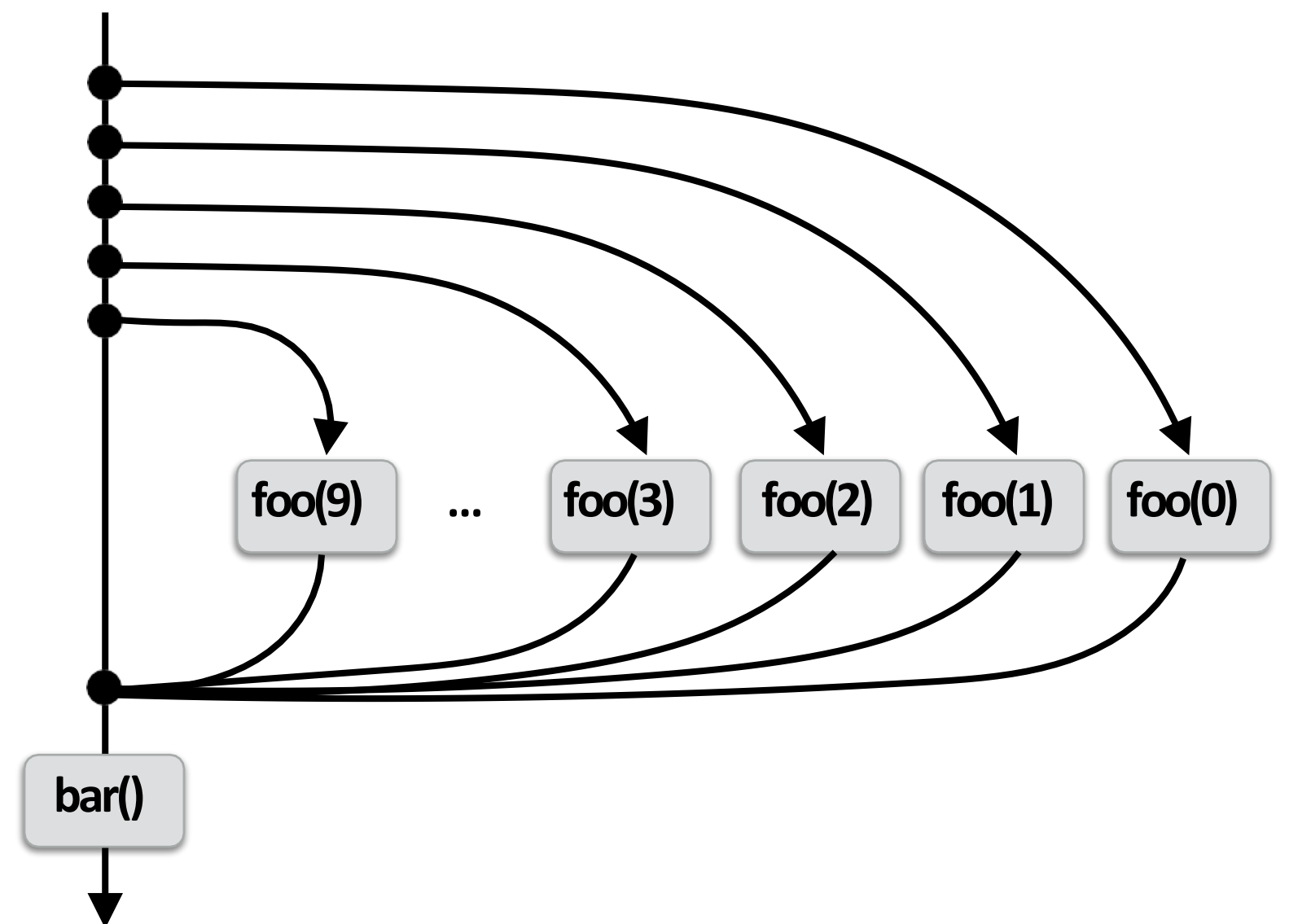
Working on `foo(1)`, `id=A`...

Thread 0 now working on `foo(2)`

# Implementing sync: greedy policy

**block (id: A)**

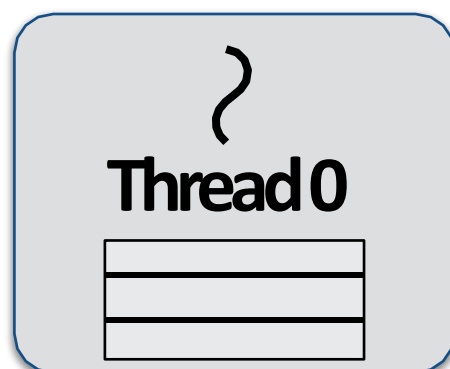
```
[ for (int i=0; i<10; i++) {  
    cilk_spawn foo(i);  
}  
cilk_sync; Sync for all calls spawned within block A  
bar();  
]
```



Thread 0 work queue

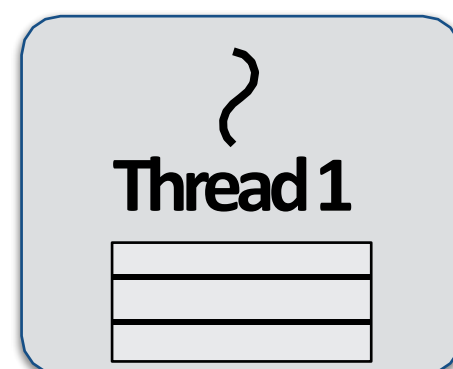
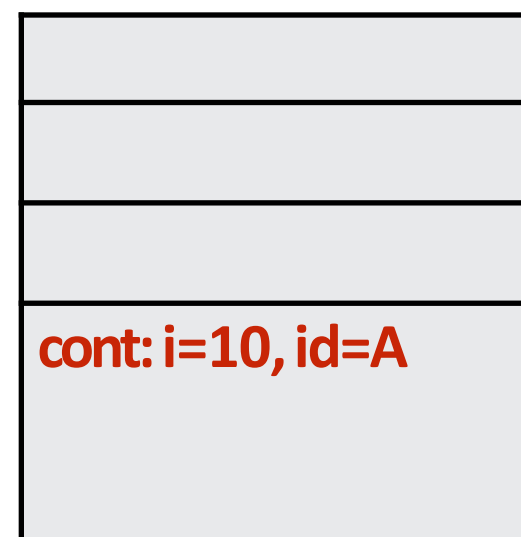


id=A  
spawn:10,  
done:9



Idle

Thread 1 work queue



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

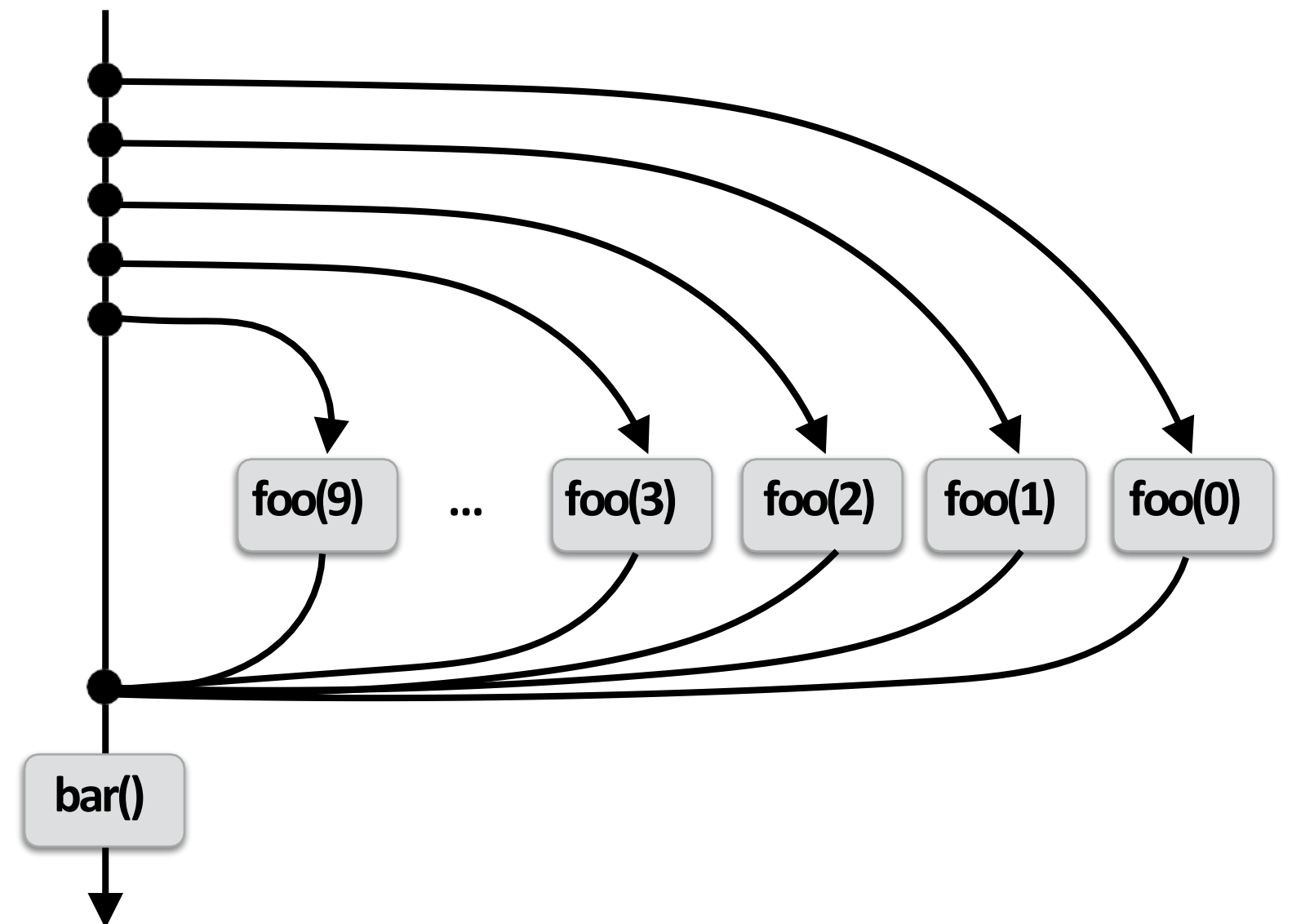
Assume thread 1 is the last to finish spawned calls for block A.



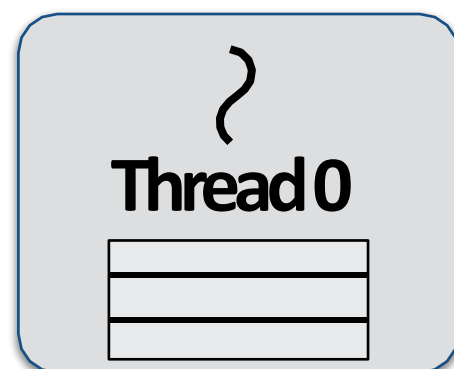
# Implementing sync: greedy policy

**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();  
]
```

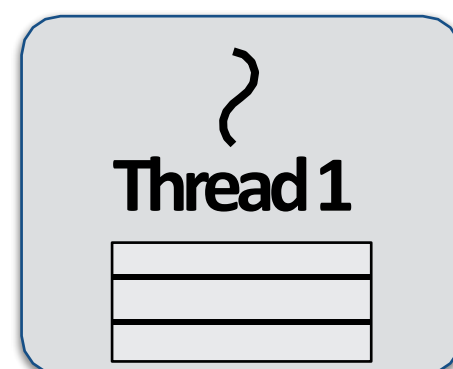
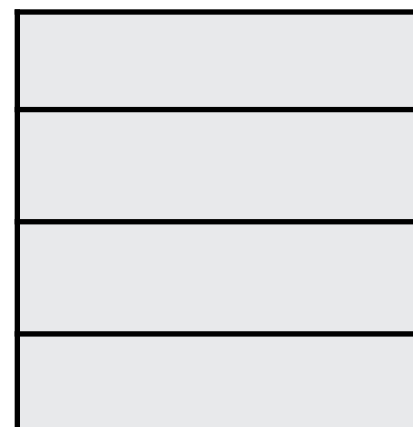


Thread 0 work queue



Idle

Thread 1 work queue



Working on bar()

Thread 1 continues on to run `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** (thread only goes idle if no work to steal is present in 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 dequeue**

# Adjusting Granularity

## ■ Challenge

- If partition computation into too many tasks (fine grained), will be swamped by scheduling overhead
- If too few tasks (coarse grained), won't have enough parallelism and more prone to workload imbalance
- Requiring programmer to determine granularity in advance is tedious and non-portable

## ■ Cilk

- Generates two versions of every program
  - “Fast clone” optimized for sequential execution.
    - Spawning  $\approx$  procedure call
  - “Slow clone” does full fork/join parallelism

# Cilk Granularity

## ■ Action

- When procedure spawned, runs fast clone
- When thief steals procedure, convert to slow clone

## ■ Effect

- If no stealing tasks place, runs fast clone sequentially
- Stealing initiates parallel execution
- But once workers have enough to do, they shift to fast clones as they spawn more procedures
- Since stealing is done on big chunks of work, tend to partition big chunks of worker into smaller ones, but shift to sequential execution once there are enough tasks to keep workers busy

# 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)