**Lecture 6:**

**( how to be l33t )**

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

**Parallel Computer Architecture and Programming**
**CMU 15-418/15-618, Spring 2017**

# Tunes

# Kaleo
## Way Down We Go
## (A / B)

*"I thought the excitement of observing execution time dropping was a story that had surprisingly been untold by the great artists of our generation. The song is an homage to better work scheduling."*

*- Jökull Júlíusson*

# 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)**

**Time** **P1** **P2** **P3** **P4**
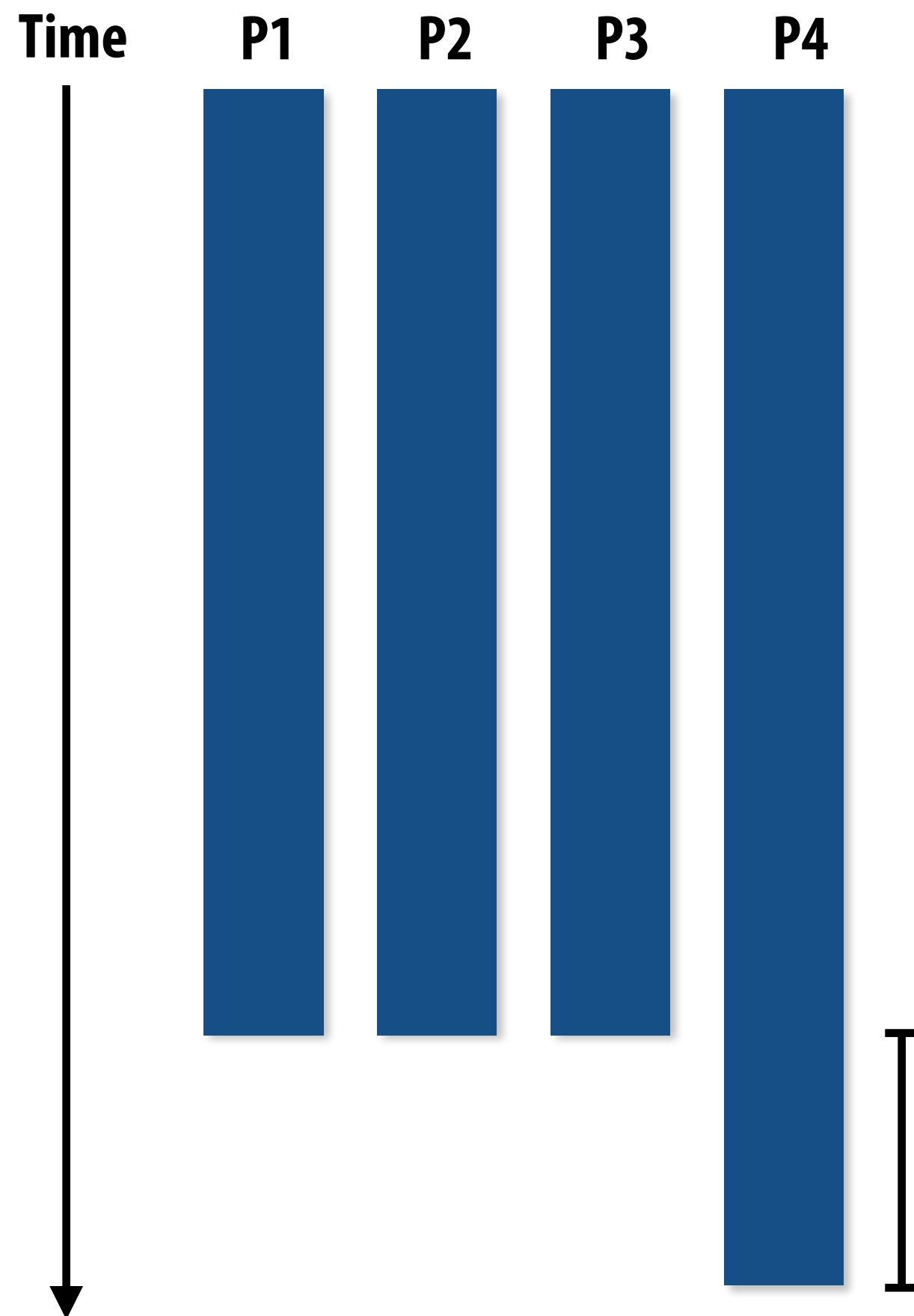
**Recall Amdahl's Law:**
**Only small amount of load imbalance can significantly bound maximum speedup**

**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)**
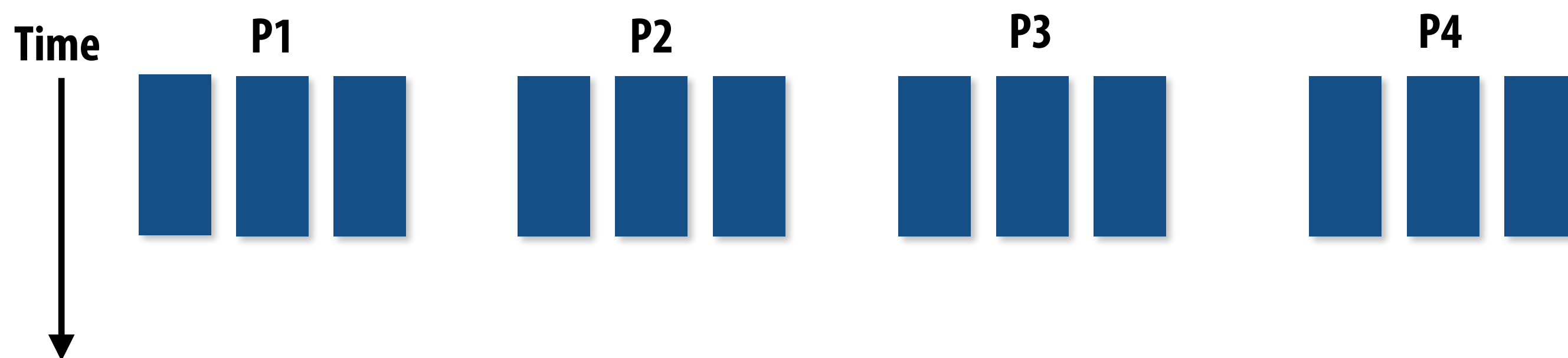
# Static assignment

- **Assignment of work to threads is pre-determined**
    - Not necessarily determined at compile-time (assignment algorithm may depend on runtime parameters such as input data size, number of threads, etc.)

- **Recall solver example: assign equal number of grid cells (work) to each thread (worker)**
    - We discussed two static assignments of work to workers (blocked and interleaved)
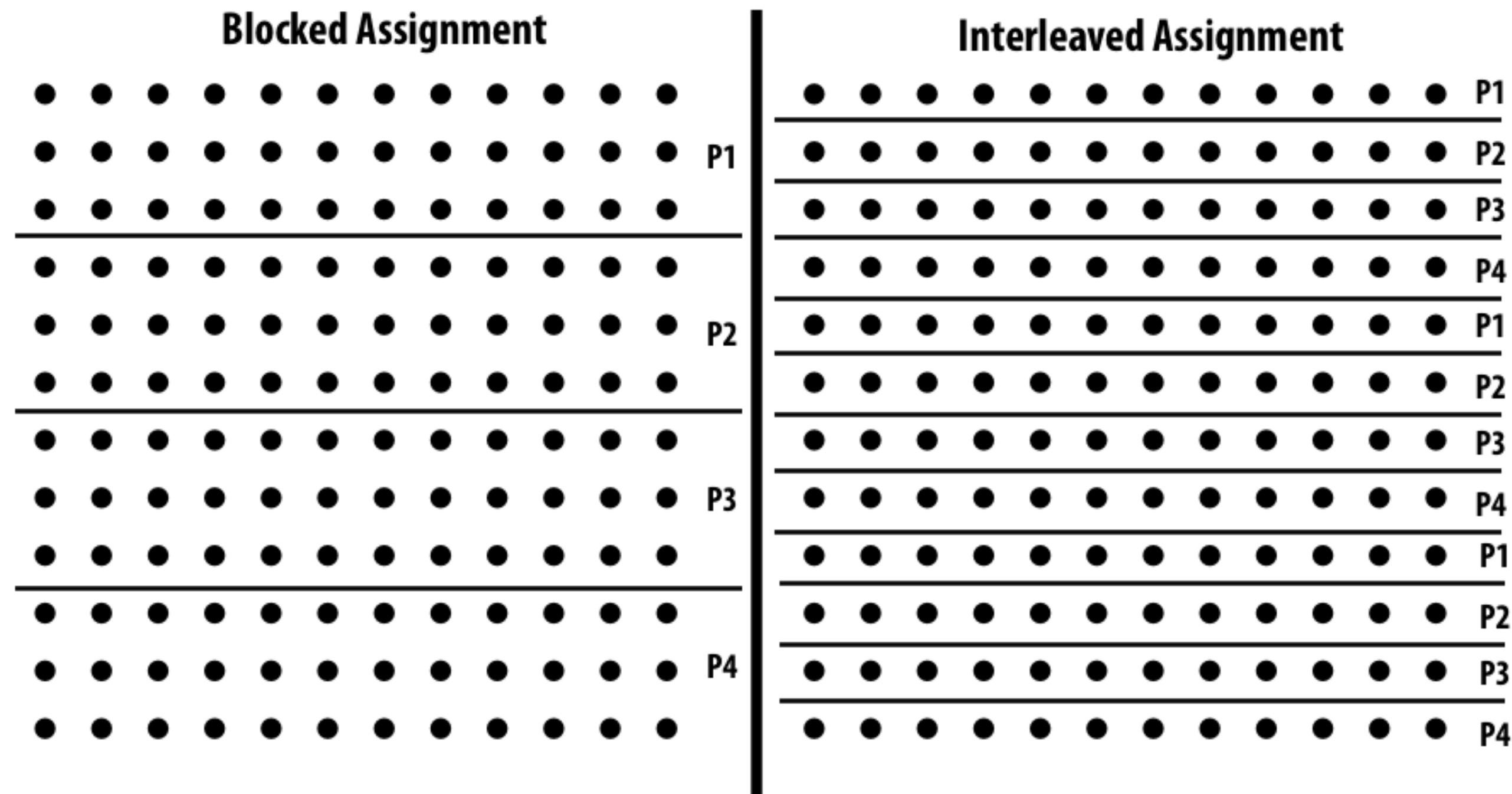


- **Good properties of static assignment: simple, essentially zero runtime overhead (in this example: extra work to implement assignment is a little bit of indexing math)**

# When is static assignment applicable?

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

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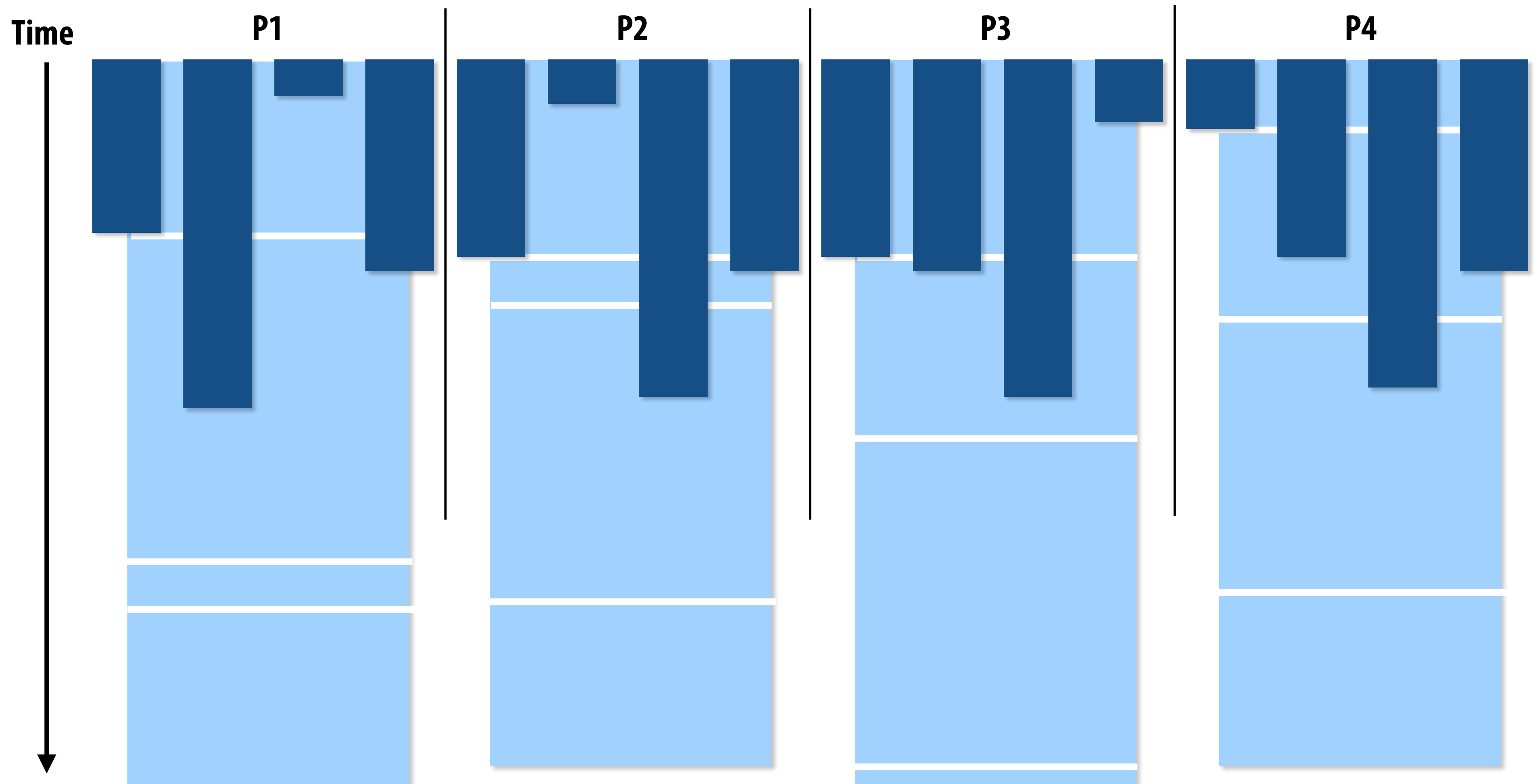


**In the example above:**

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

Assignment solution: statically assign three tasks to each of the four processors.

# When is static assignment applicable?

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

Time

| P1 | P2 | P3 | P4 |

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

# "Semi-static" assignment

- **Cost of work is predictable for near-term future**
  - Idea: recent past good predictor of near future

- **Application periodically profiles application and re-adjusts assignment**
  - Assignment is "static" for the interval between re-adjustments



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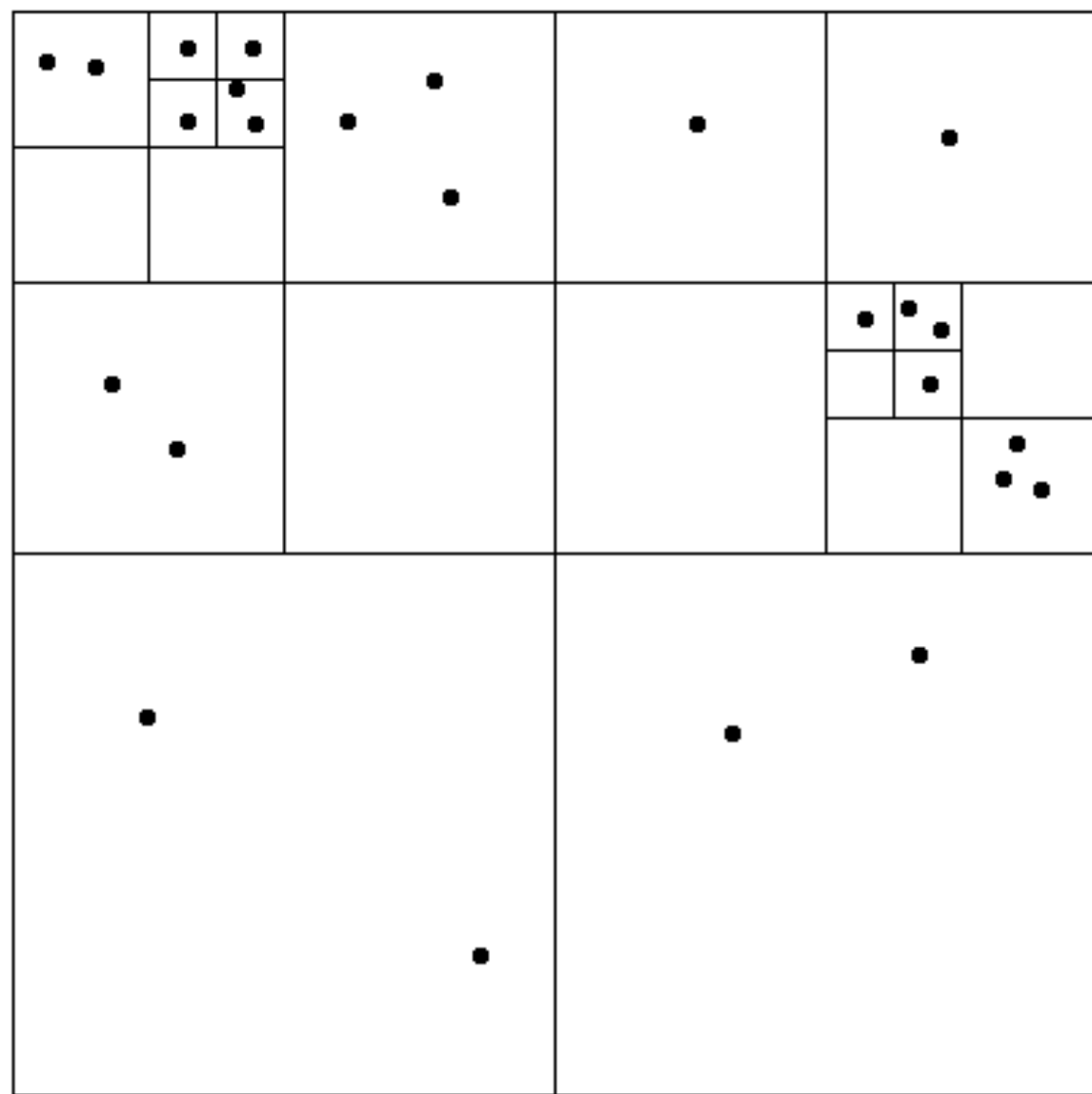
**Particle simulation:**

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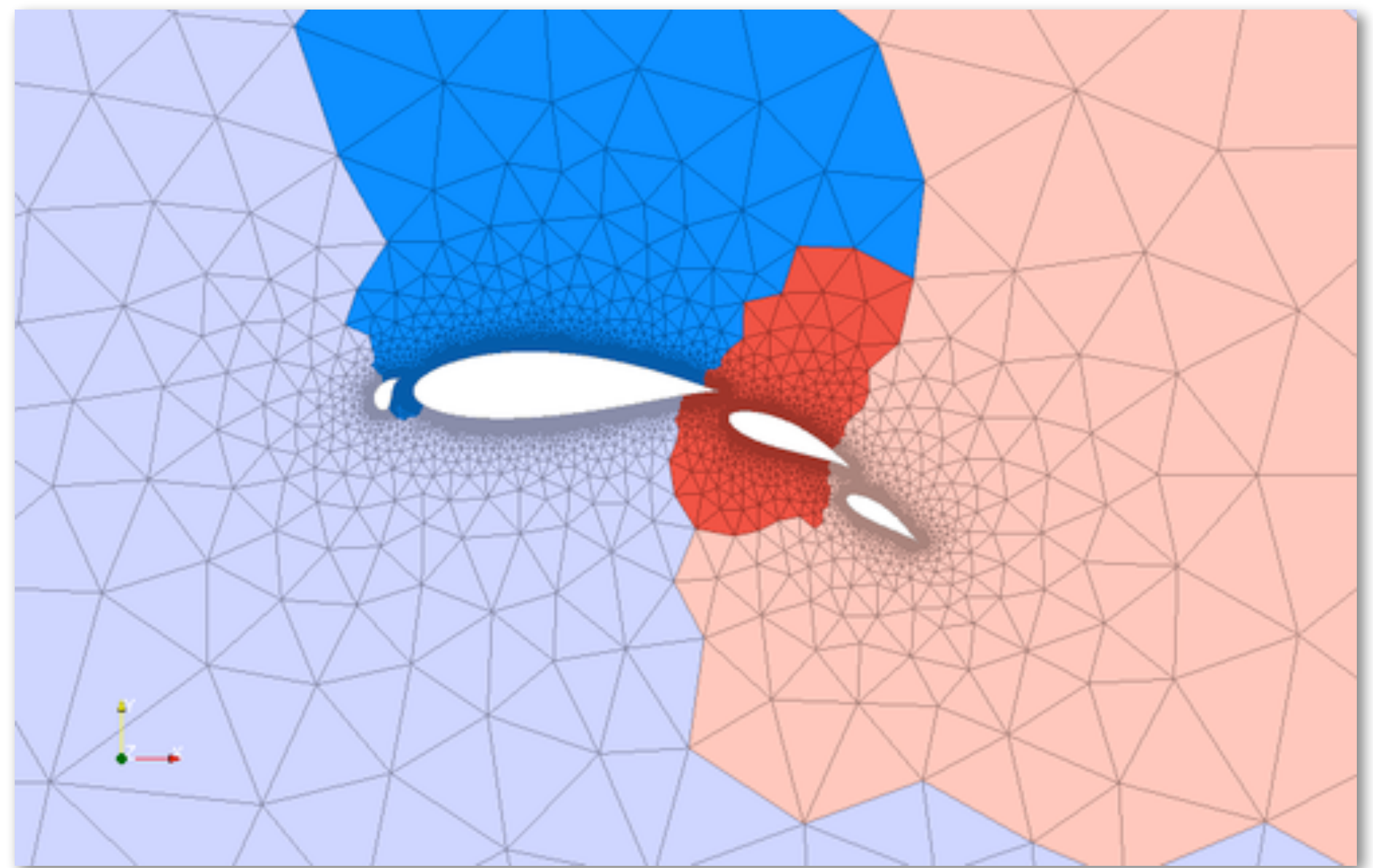
**Adaptive mesh:**

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

# Dynamic assignment

Program determines assignment dynamically at runtime to ensure a well distributed load. (The execution time of tasks, or the total number of tasks, is unpredictable.)

**Sequential program**
**(independent loop iterations)**

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

// initialize elements of x here

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

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

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

// initialize elements of x here

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

while (1) {
  int i;
  lock(counter_lock);
  i = counter++;              atomic_incr(counter);
  unlock(counter_lock);
  if (i >= N)
      break;
  is_prime[i] = test_primality(x[i]);
}
```
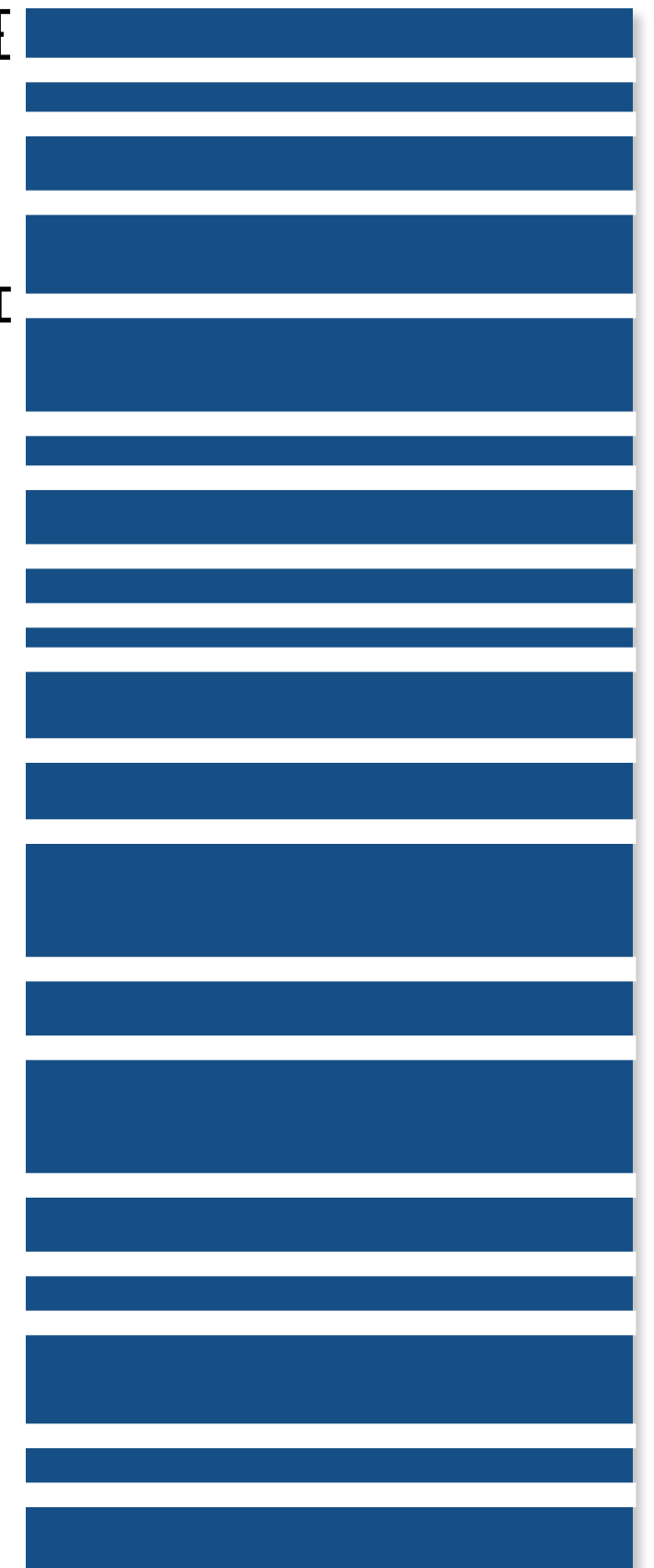
# Dynamic assignment using a work queue

**Sub-problems**
**(a.k.a. "tasks", "work")**
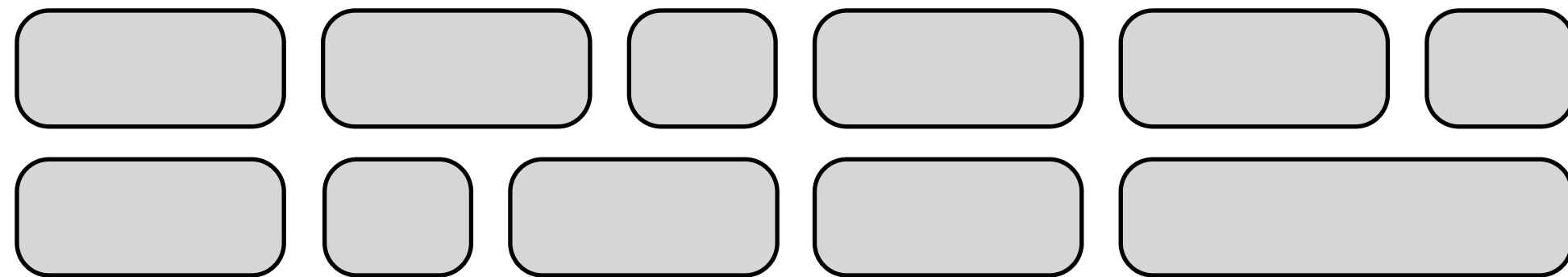
**Shared work queue: a list of work to do**
**(for now, let's assume each piece of work is independent)**

T1  T2  T3  T4
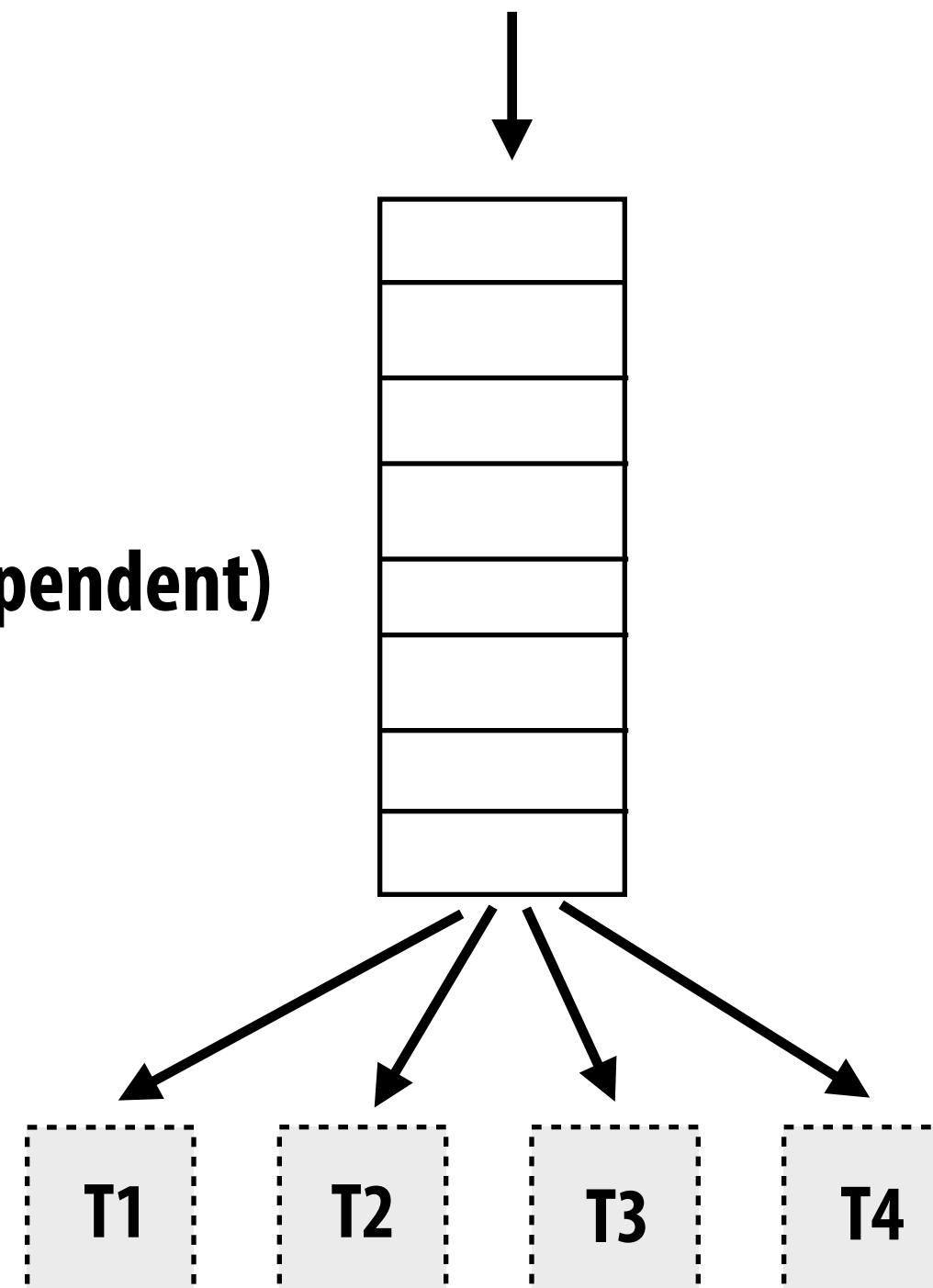
**Worker threads:**
**Pull data from shared work queue**
**Push new work to queue as it is created**

# What constitutes a piece of work?

## What is a potential problem with this implementation?

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

// initialize elements of x here

LOCK counter_lock;
int counter = 0;

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

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

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

**Time in task 0 ⎯⎯⎯⎯⎯⎯⎯⎯**

**Time in critical section ⎯⎯⎯⎯⎯**

**This is overhead that
does not exist in serial
program**

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

# So... IS IT a problem?

# Increasing task granularity

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

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

// initialize elements of x here

LOCK counter_lock;
int counter = 0;

while (1) {
  int i;
  lock(counter_lock);
  i = counter;
  counter += GRANULARITY;
  unlock(counter_lock);
  if (i >= N)
     break;
  int end = min(i + GRANULARITY, N);
  for (int j=i; j<end; j++)
     is_prime[i] = test_primality(x[i]);
}
```

Time in task 0

Time in critical section
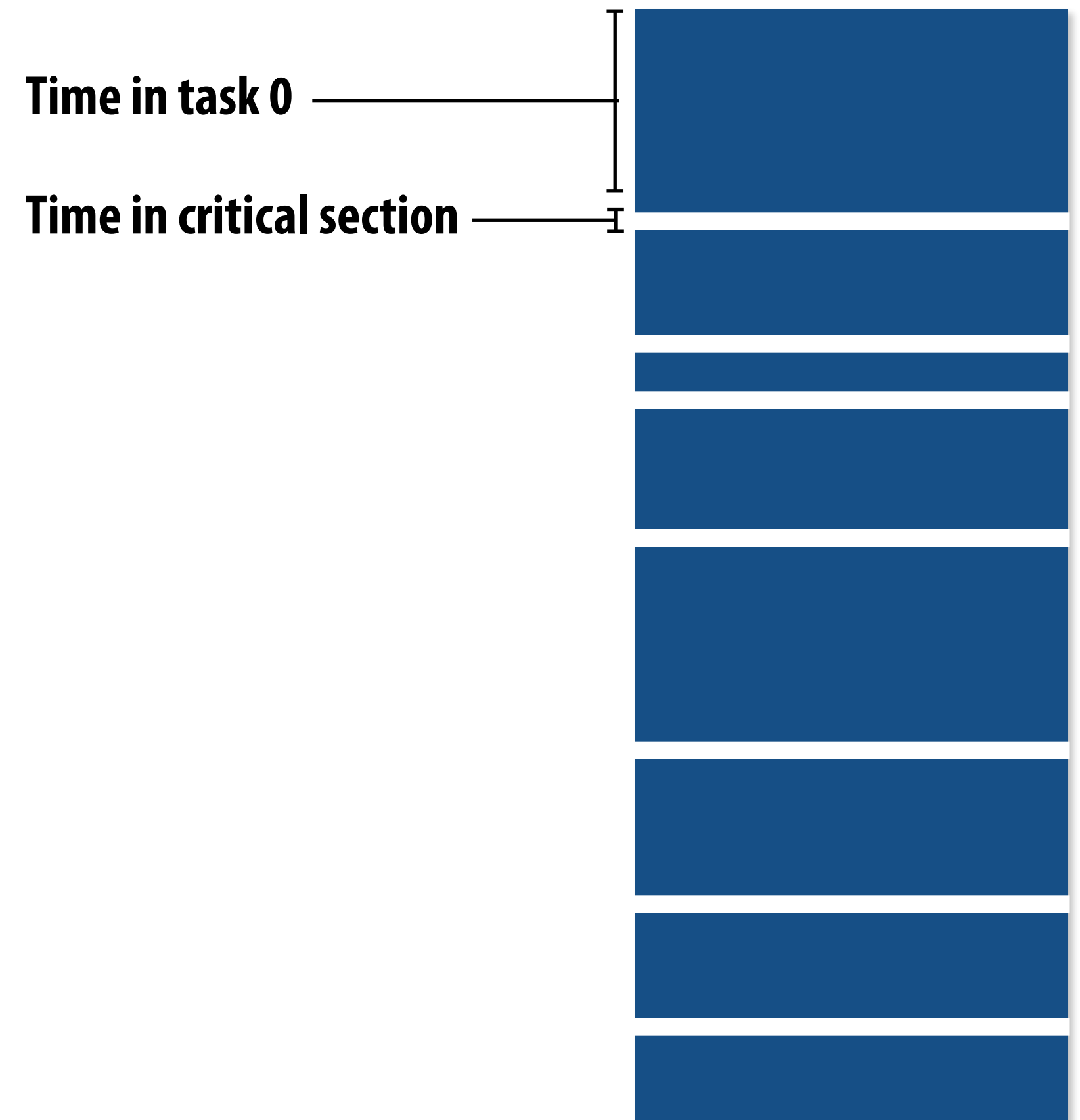
**Coarse granularity partitioning: 1 "task" = 10 elements**
**Decreased synchronization cost**
**(Critical section entered 10 times less)**

# Choosing task size

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

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

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

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

# Smarter task scheduling

**Consider dynamic scheduling via a shared work queue**

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



Time

16 Tasks
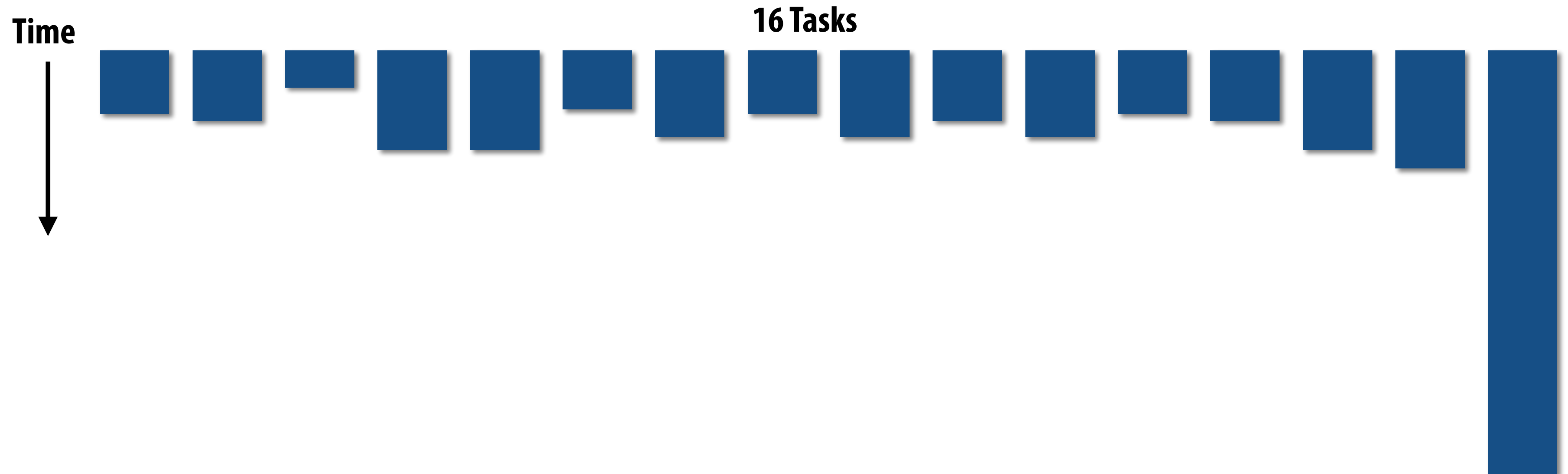
# Smarter task scheduling

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

Time

P1          P2          P3          P4

Done!
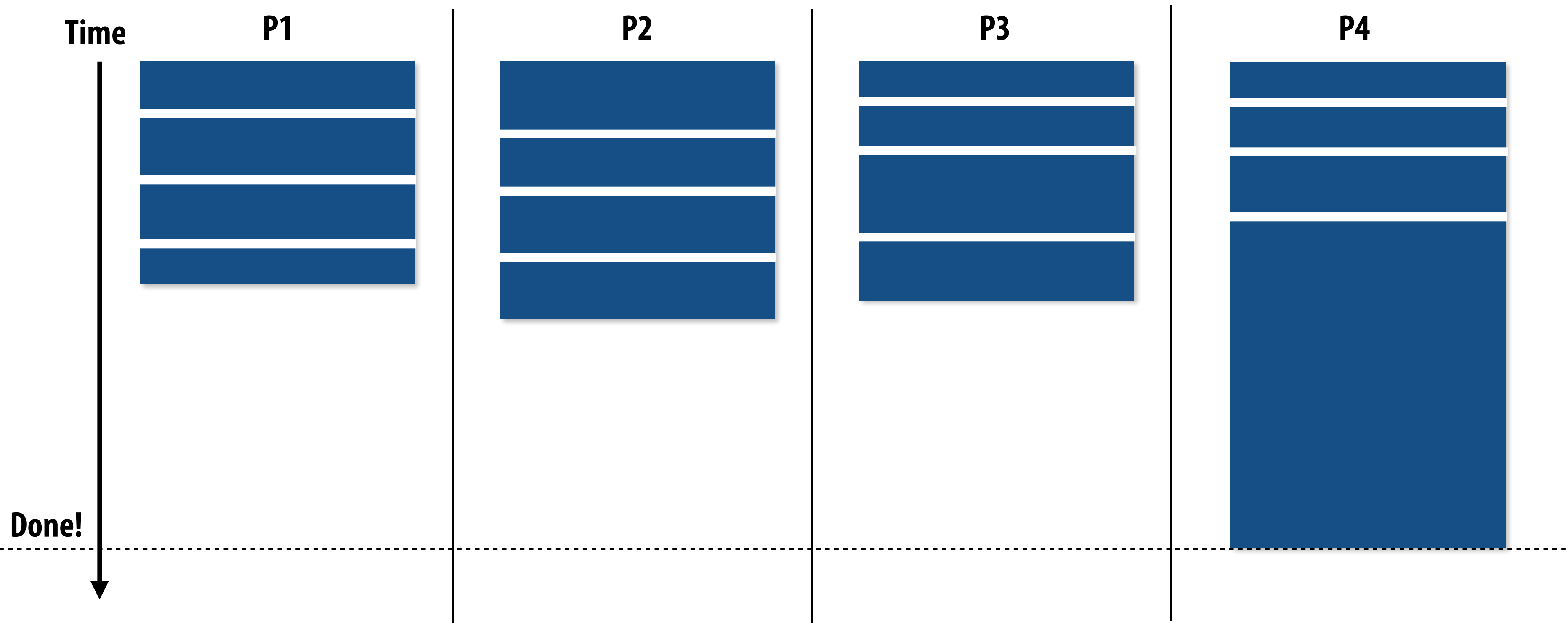
**One possible solution to imbalance problem:**

**Divide work into a larger number of smaller tasks**
- **Hopefully "long pole" gets shorter relative to overall execution time**
- **May increase synchronization overhead**
- **May not be possible (perhaps long task is fundamentally sequential)**

# Smarter task scheduling

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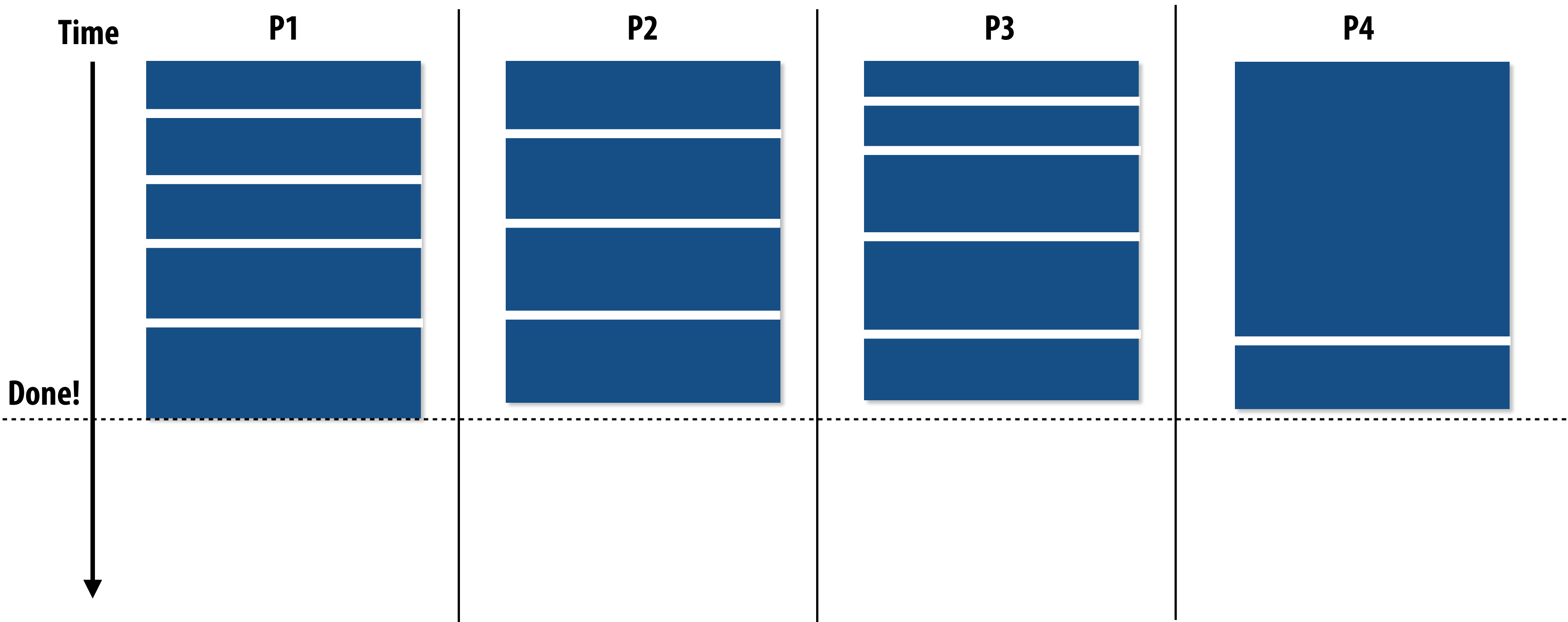


**Another solution: smarter scheduling**

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

# Decreasing synchronization overhead using a distributed set of queues

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

**Subproblems**
**(a.k.a. "tasks", "work to do")**
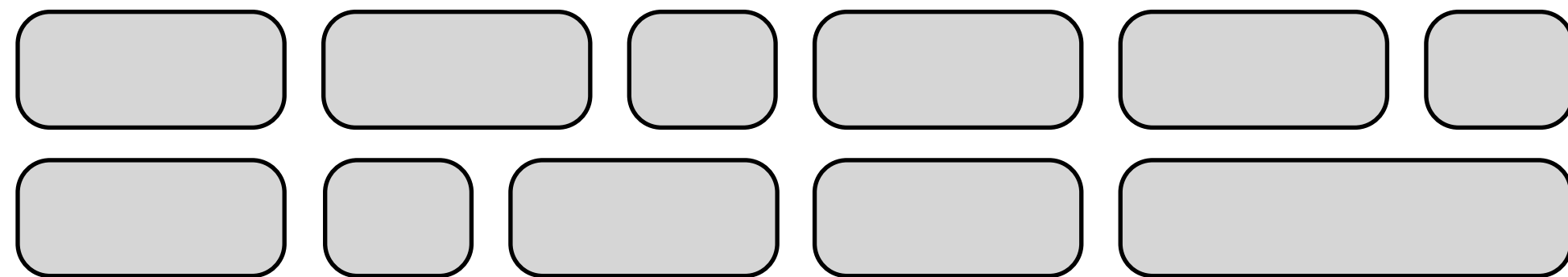
**Set of work queues**
**(In general, one per worker thread)**
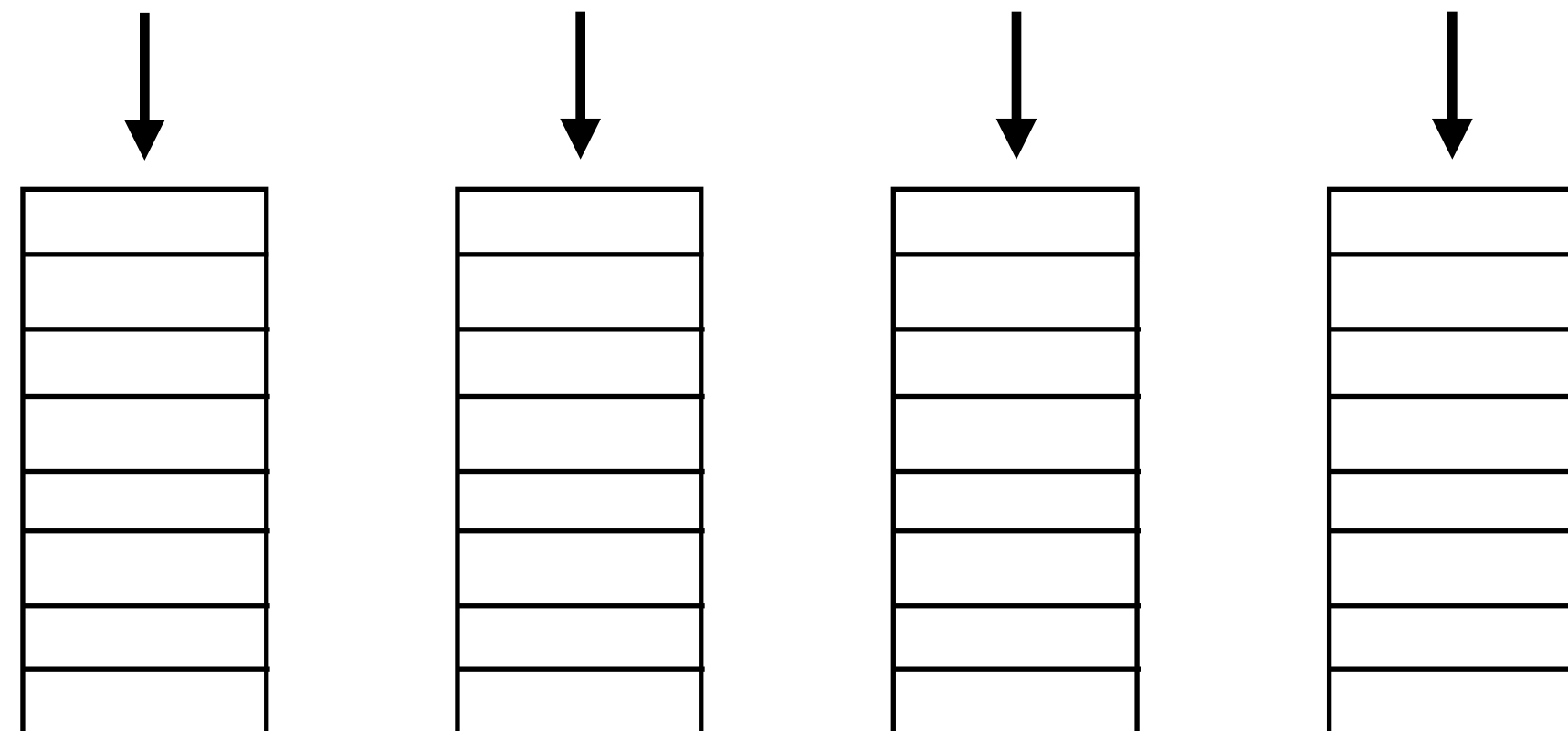
**Worker threads:**

Pull data from OWN work queue

Push new work to OWN work queue

**When local work queue is empty...**

**STEAL work from another work queue**

Steal!

T1    T2    T3    T4
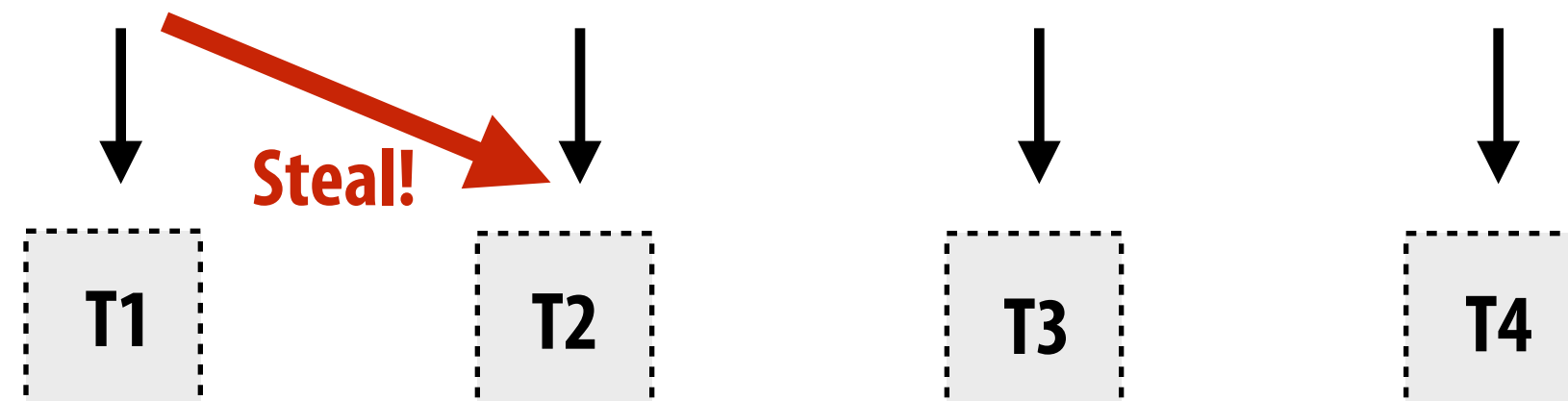
# Work in task queues need not be independent



= application-specified dependency

Task management system:
Scheduler manages dependencies
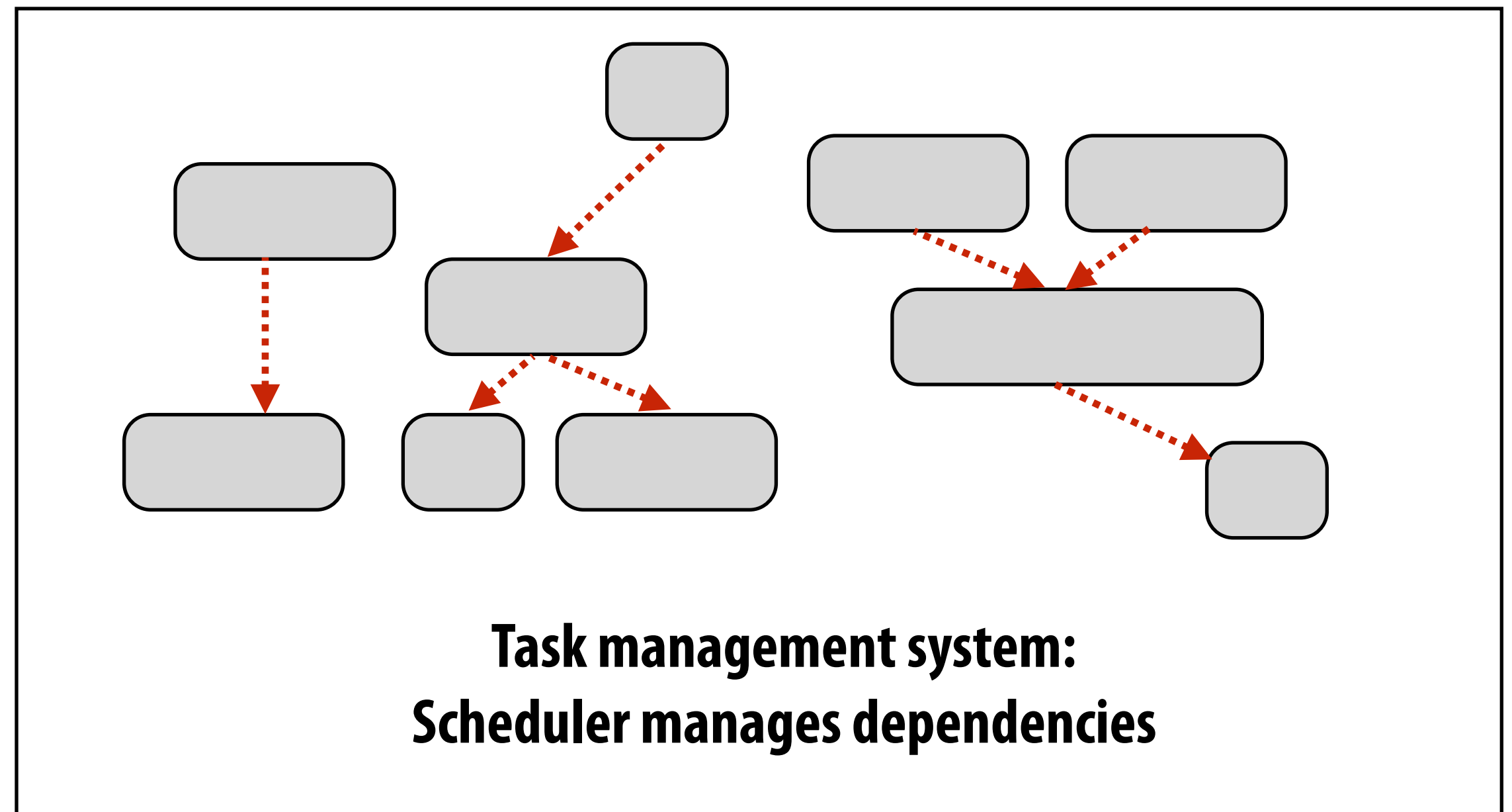
**A task is not removed from queue and assigned to worker thread until all task dependencies are satisfied**

T1  T2  T3  T4
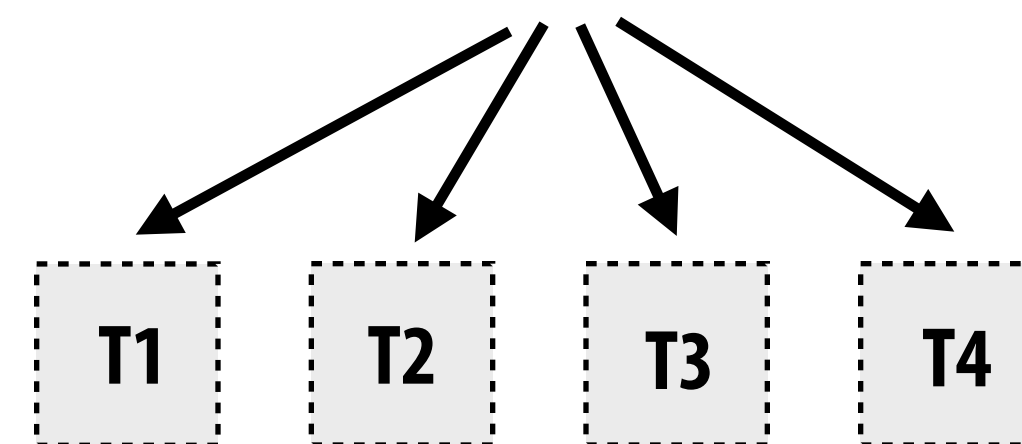
**Workers can submit new tasks (with optional explicit dependencies) to task system**

```
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)

- **Issues discussed today span 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]);
}
```

```
// ISPC bulk task launch
launch[numTasks] myFooTask(A, B);
```

```
// bulk CUDA thread launch
foo<<<numBlocks, threadsPerBlock>>>(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]);
}
```

foo()
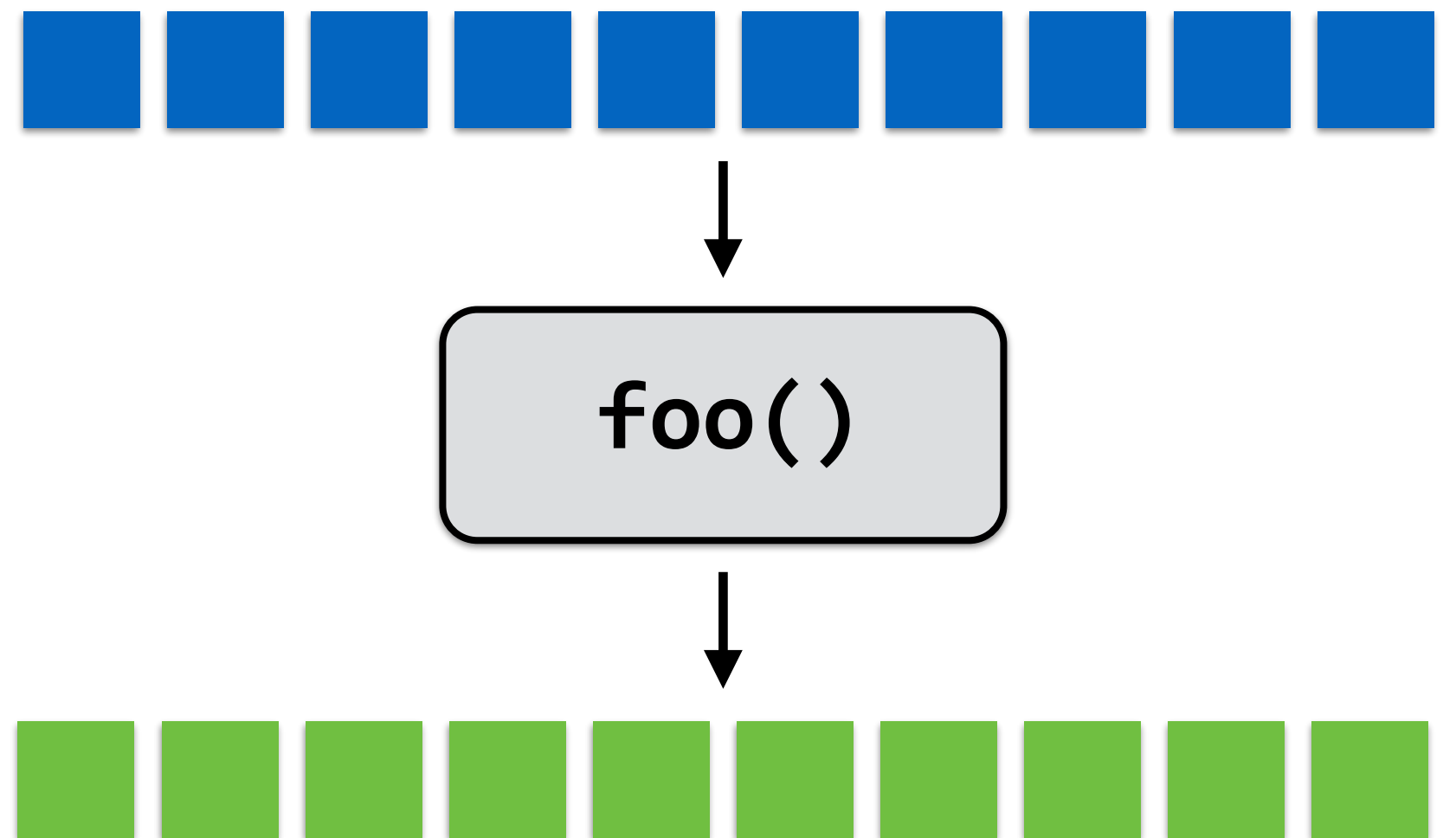
# Common parallel programming patterns

**Explicit management of parallelism with threads:**

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

    - **Example below: C code with pthreads**

```c
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, last);
   }
}
```

**independent work!**

**Dependencies**
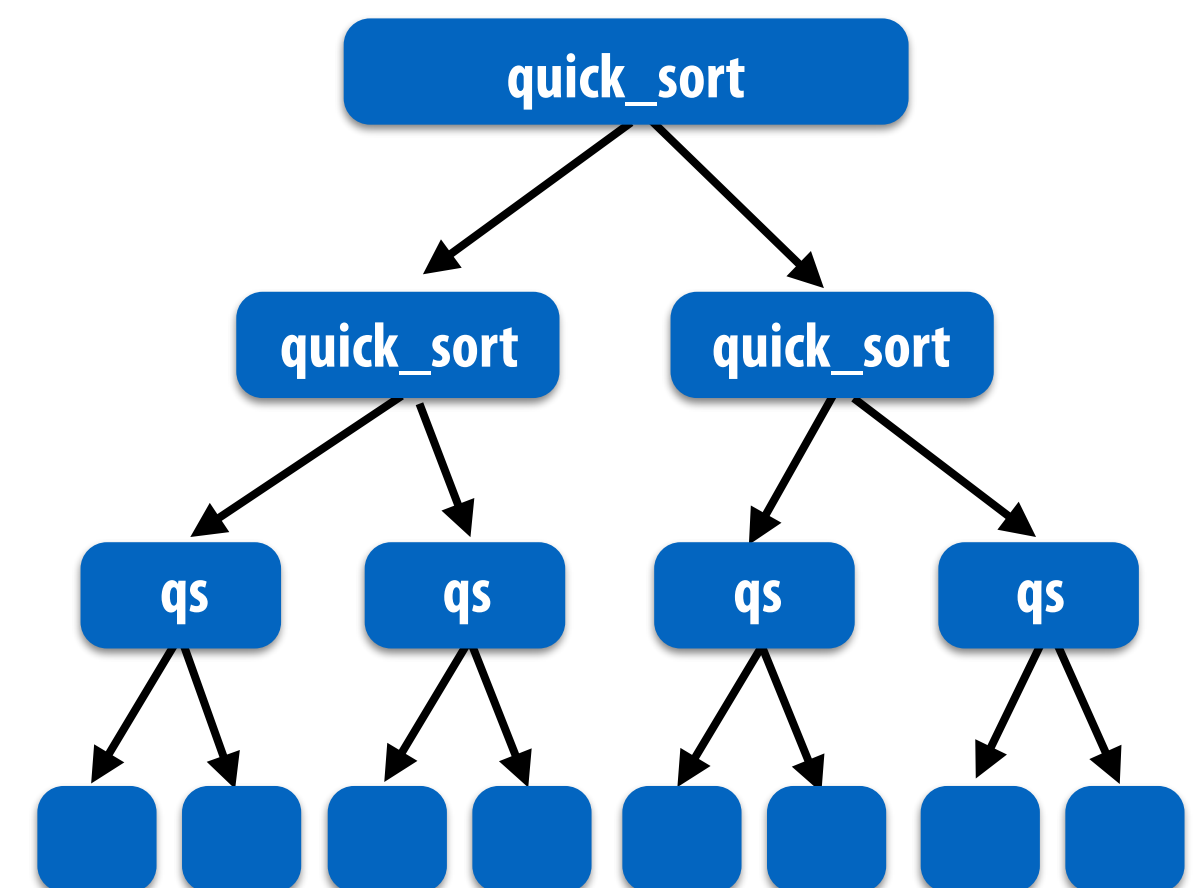
# Fork-join pattern

- **Natural way to express independent work 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 <u>may continue executing asynchronously</u> 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;
```
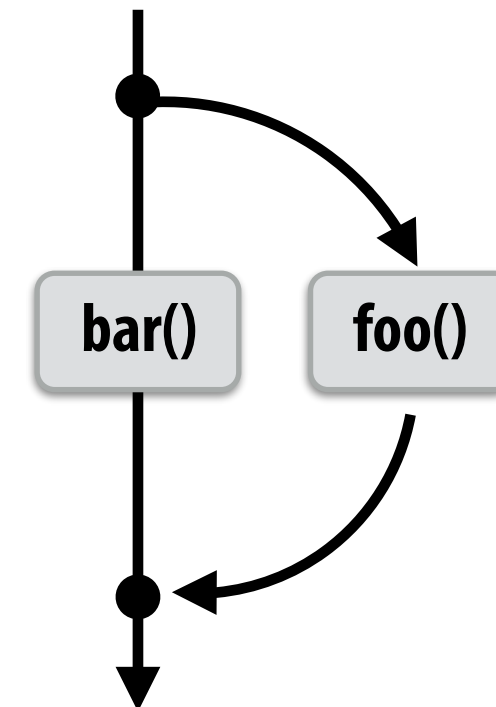
```
// foo() and bar() may run in parallel
cilk_spawn foo();
cilk_spawn bar();
cilk_sync;
```
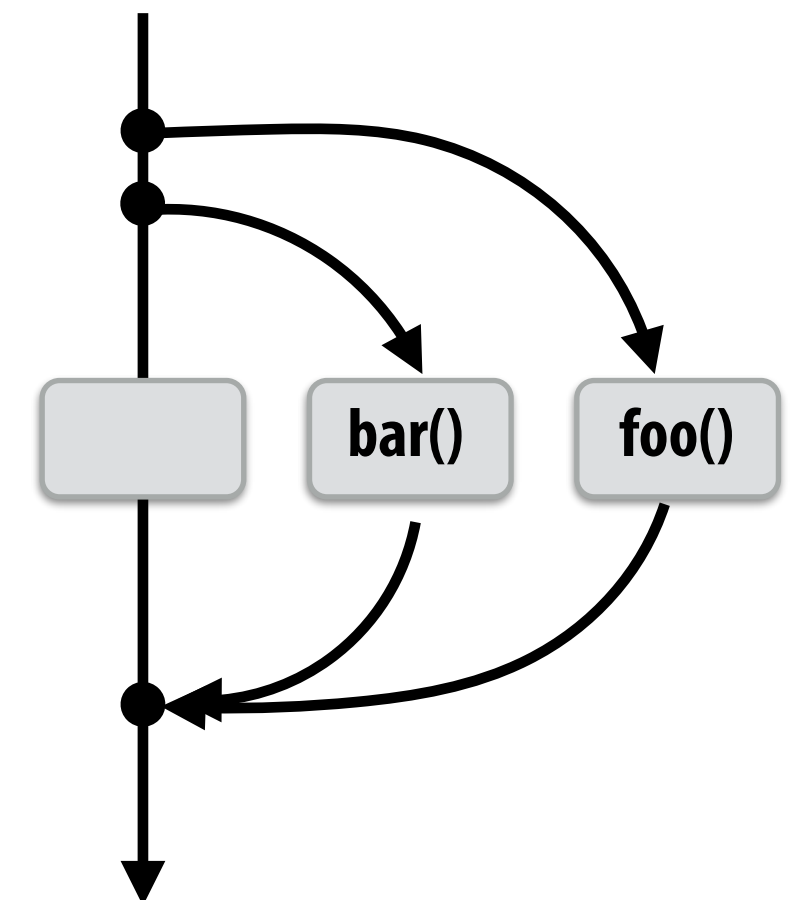
**Same amount of independent work first example, but potentially higher runtime overhead (due to two spawns vs. one)**

```
// foo, bar, fizz, buzz, may run in parallel
cilk_spawn foo();
cilk_spawn bar();
cilk_spawn fizz();
buzz();
cilk_sync;
```
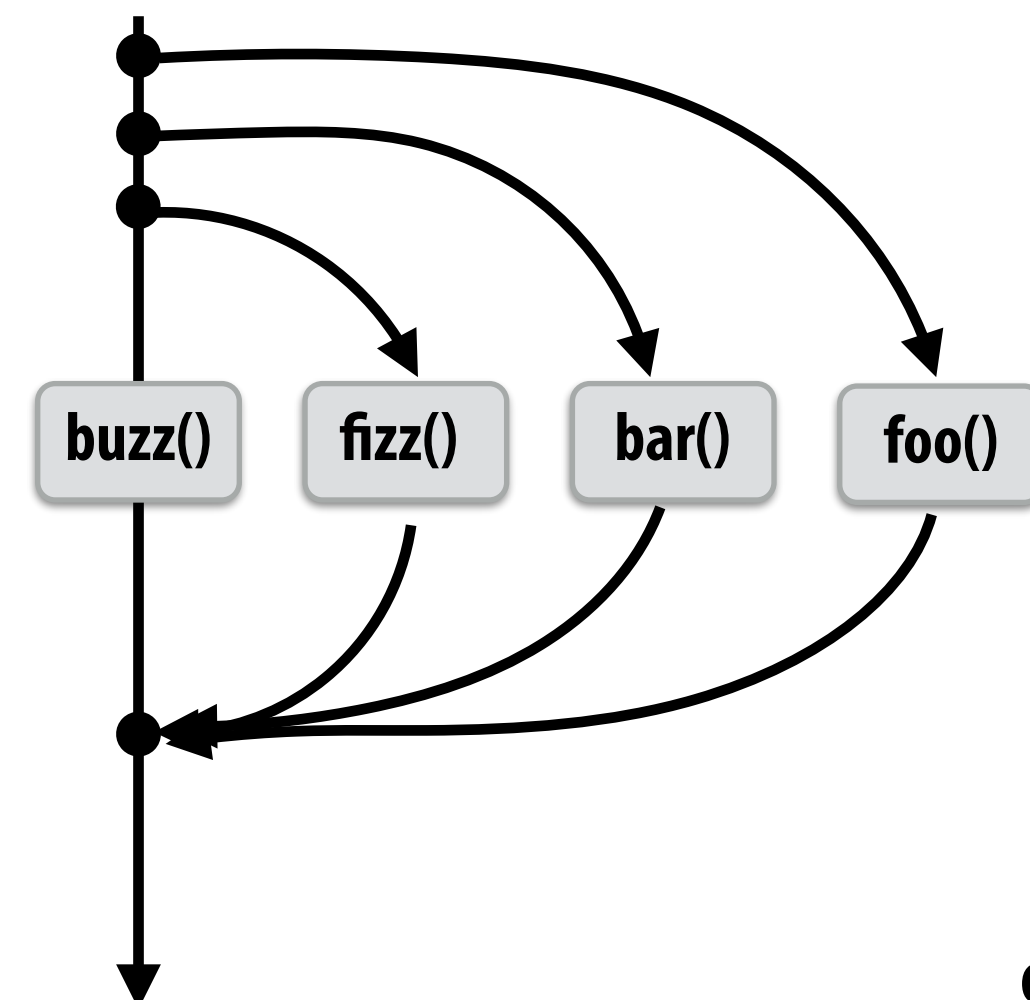
# Abstraction vs. implementation

- **Notice that the `cilk_spawn` abstraction does not specify how or when spawned calls are scheduled to execute**

  - Only that they <u>may</u> 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

# 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)**
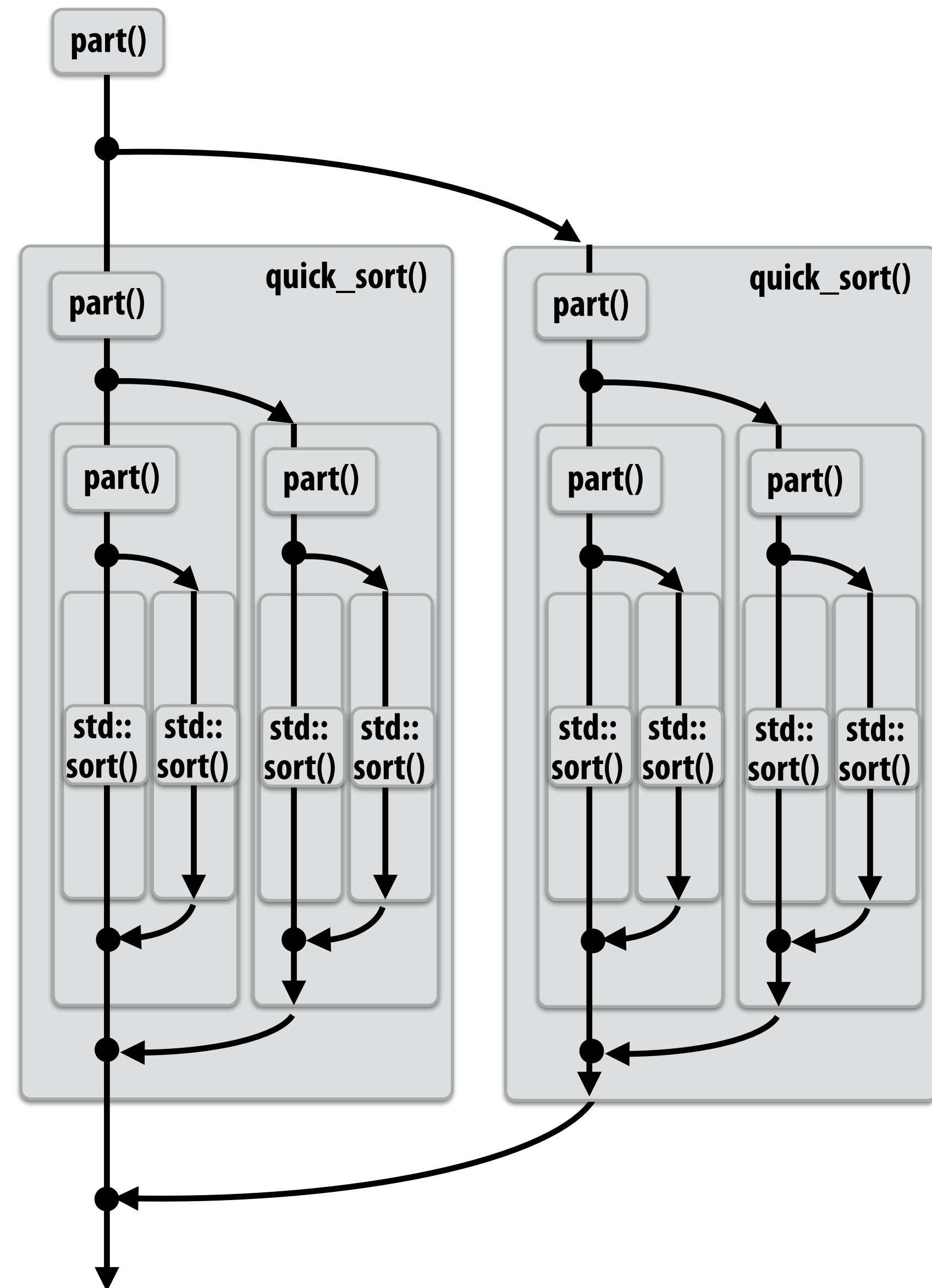
# Writing fork-join programs

- **Main idea: expose independent work (potential parallelism) to the system using `cilk_spawn`**

- **Recall parallel programming rules of thumb**
  - Want <u>at least as much work</u> as parallel execution capability (e.g., program should probably spawn at least as much work as there are cores)

  - Want <u>more independent work</u> 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 <u>not too much independent work</u> 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 worker threads**
  - Think: all threads created at application launch *
  - Exactly as many worker threads as execution contexts in the machine

| | | | |
|---|---|---|---|
| Thread 0 | Thread 1 | Thread 2 | Thread 3 |
| Thread 4 | Thread 5 | Thread 6 | Thread 7 |
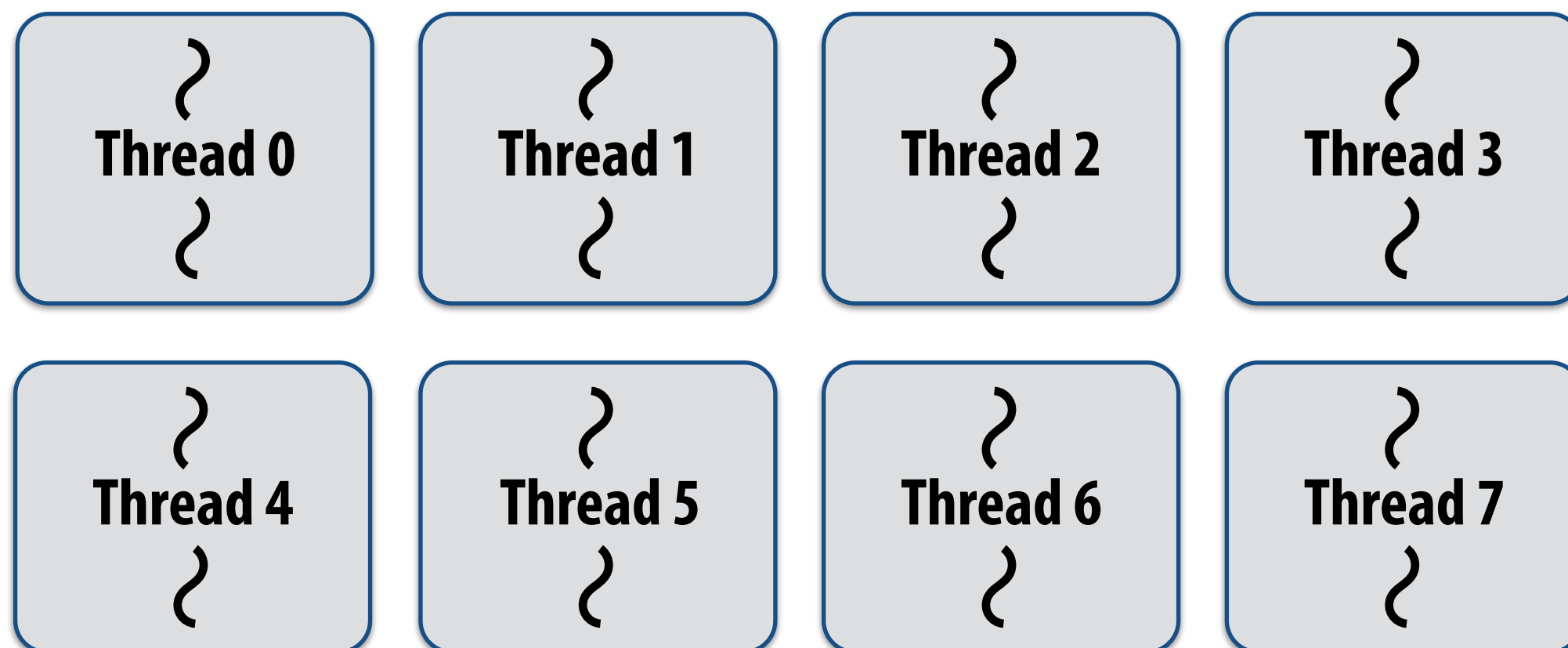
Example: Eight thread worker pool for my quad-core laptop with Hyper-Threading

```
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**

```
cilk_spawn foo();

bar();

cilk_sync;
```

**spawned child**

**continuation (rest of calling function)**

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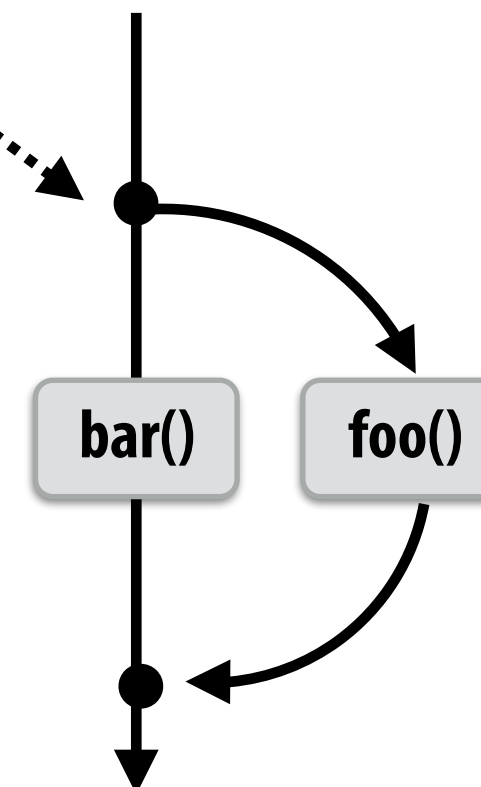
## What threads should foo() and bar() be executed by?

Thread 0    Thread 1
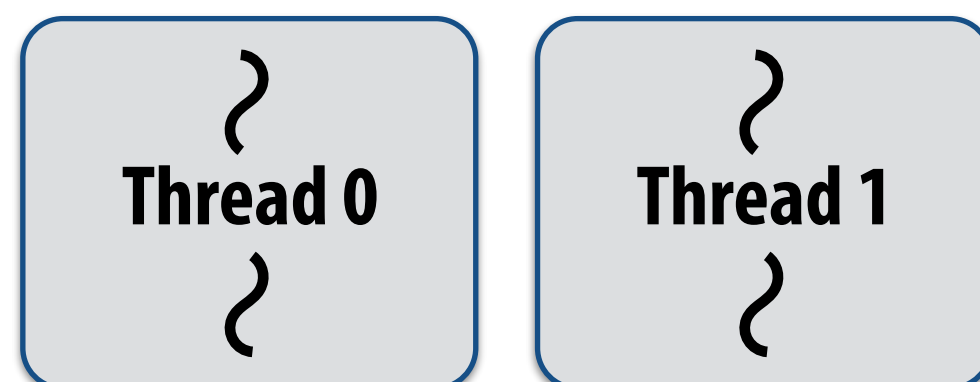
# First, consider a serial implementation

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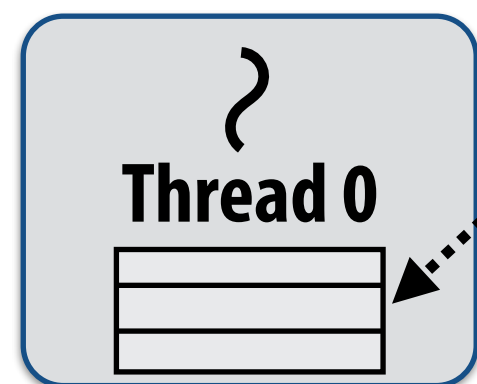
Traditional thread call stack
(indicates bar will be performed
next after return)

**Thread 0**

**Thread 1**
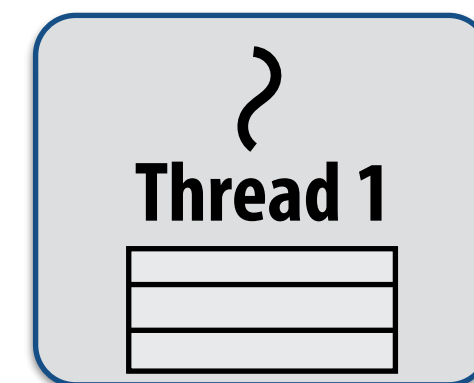
**Executing foo()…**

**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().**
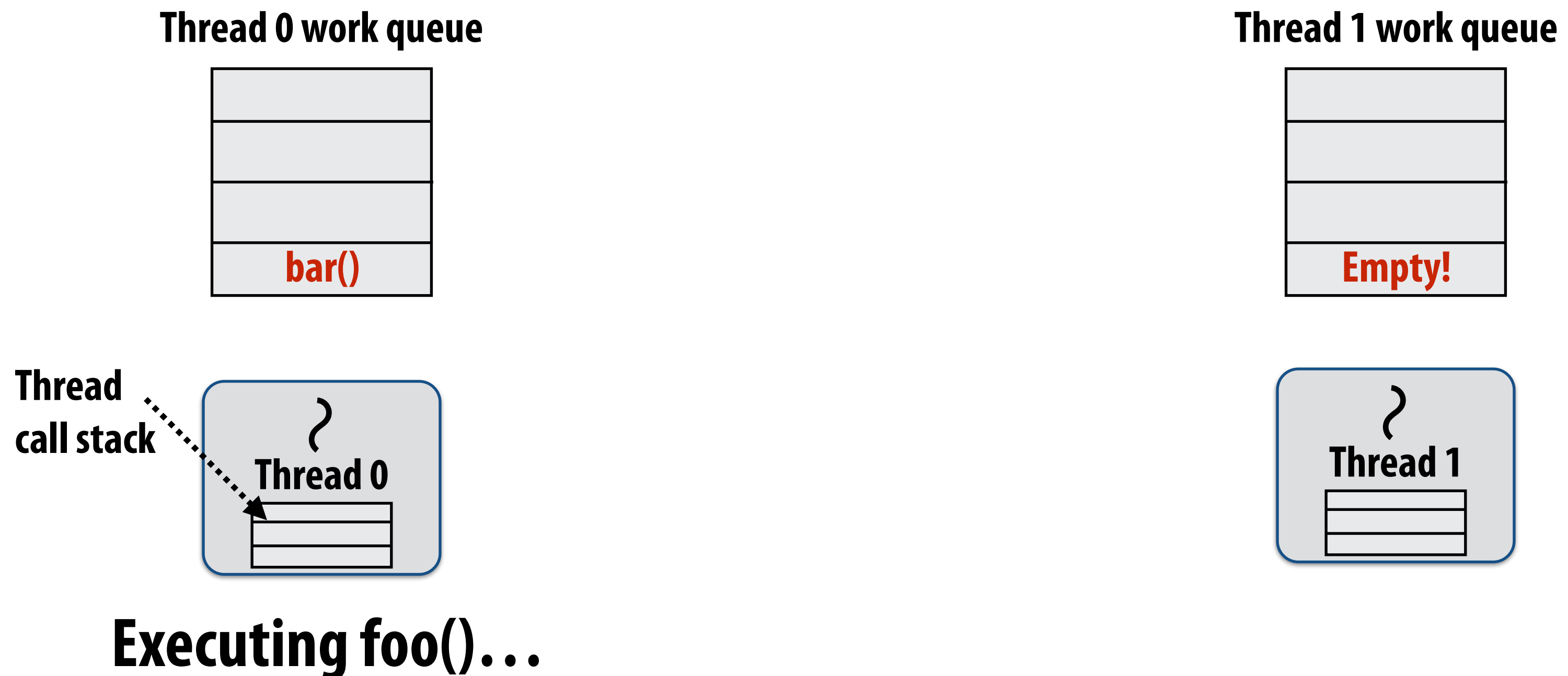
**Thread 0 work queue**

| |
|---|
| |
| |
| |
| **bar()** |

**Thread 1 work queue**

| |
|---|
| |
| |
| |
| **Empty!** |

Thread
call stack

```
～
Thread 0
```

**Executing foo()…**

```
～
Thread 1
```

# 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.**

Thread 0 work queue

Thread 1 work queue

bar()

Thread call stack

Thread 0

**1. Idle thread looks in busy thread's queue for work**

Thread 1

**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.**

Thread 0 work queue

Thread 1 work queue

**2. Idle thread moves work from busy thread's queue to its own queue**

bar()

Thread call stack

Thread 0

**1. Idle thread looks in busy threads queue for work**

Thread 1

**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.**

Thread 0 work queue

Thread 1 work queue

**2. Idle thread moves work from busy thread's queue to its own queue**

Thread call stack

Thread 0

**1. Idle thread looks in busy threads queue for work**

Thread 1

## Executing foo()...

## Executing bar()...

**3. Thread resumes execution**
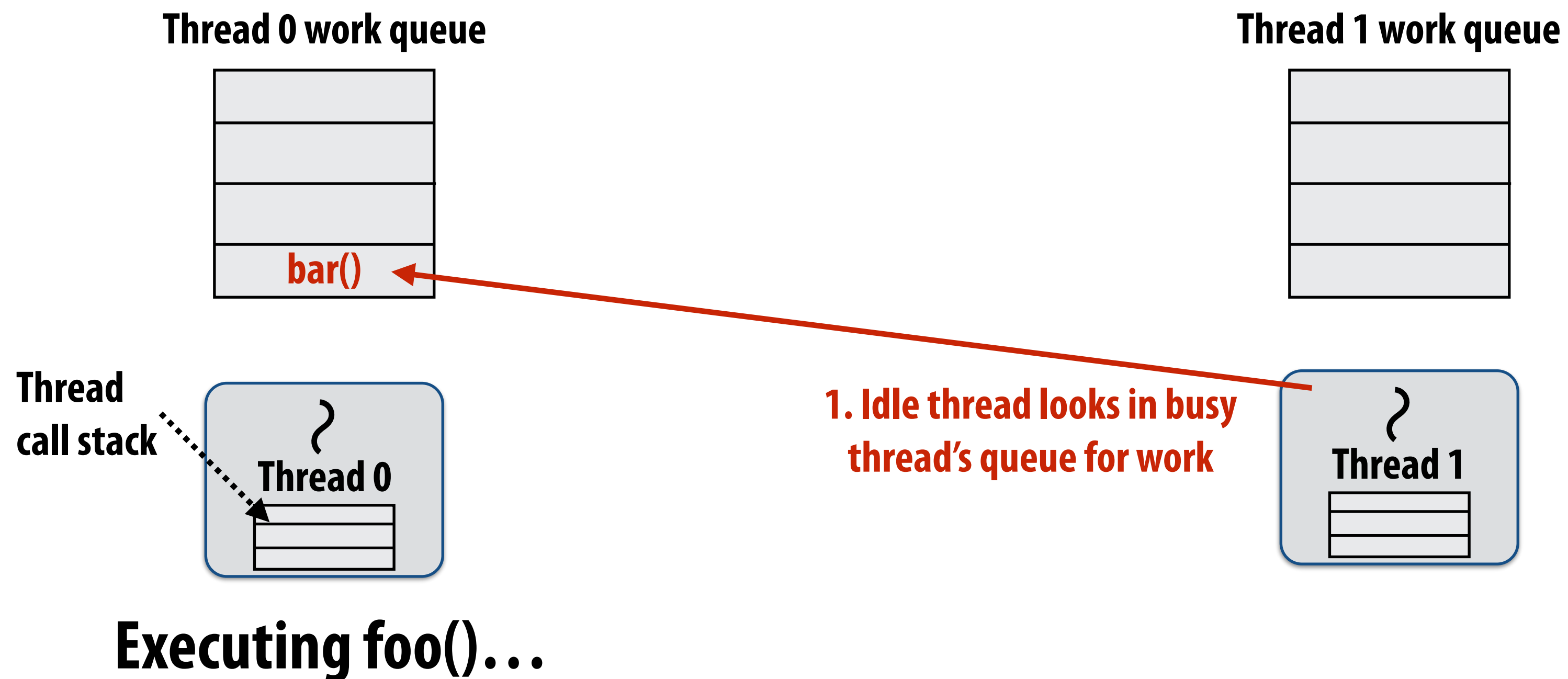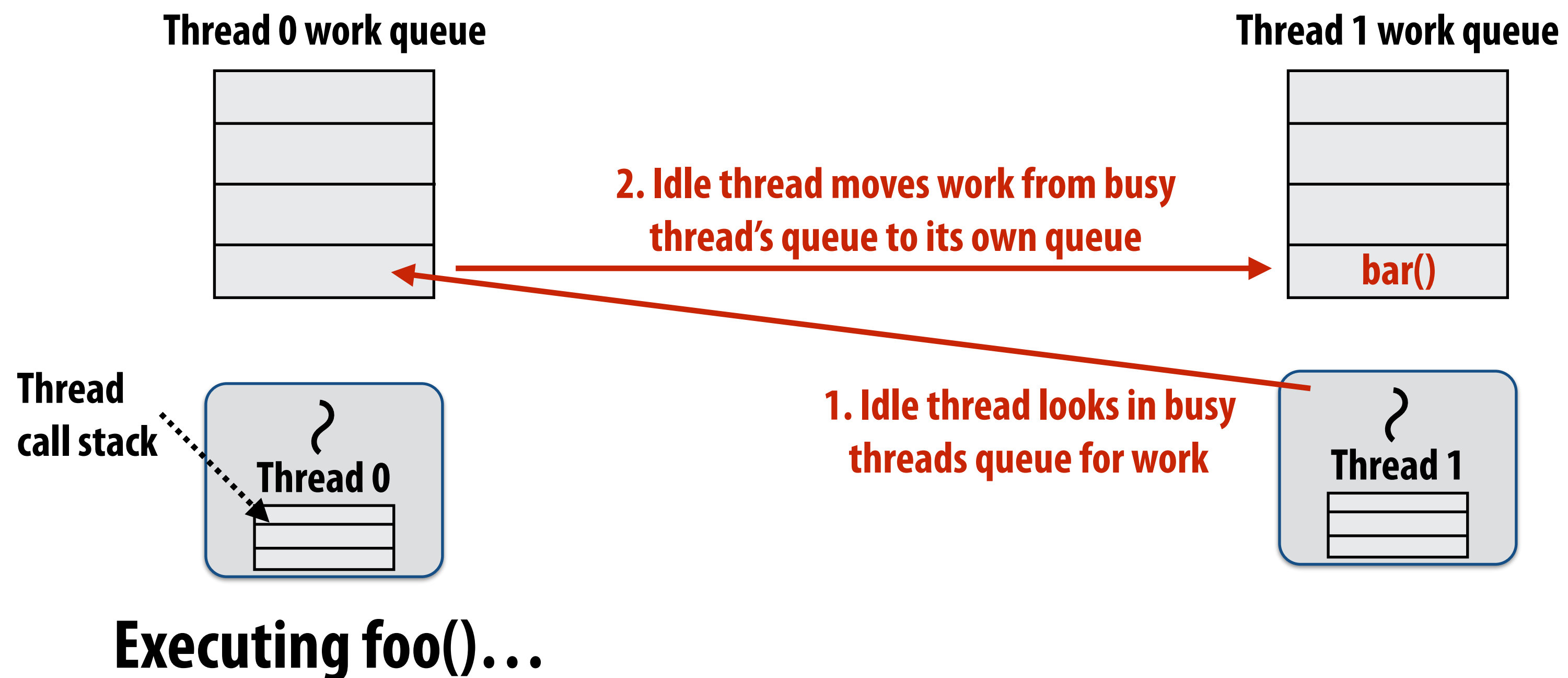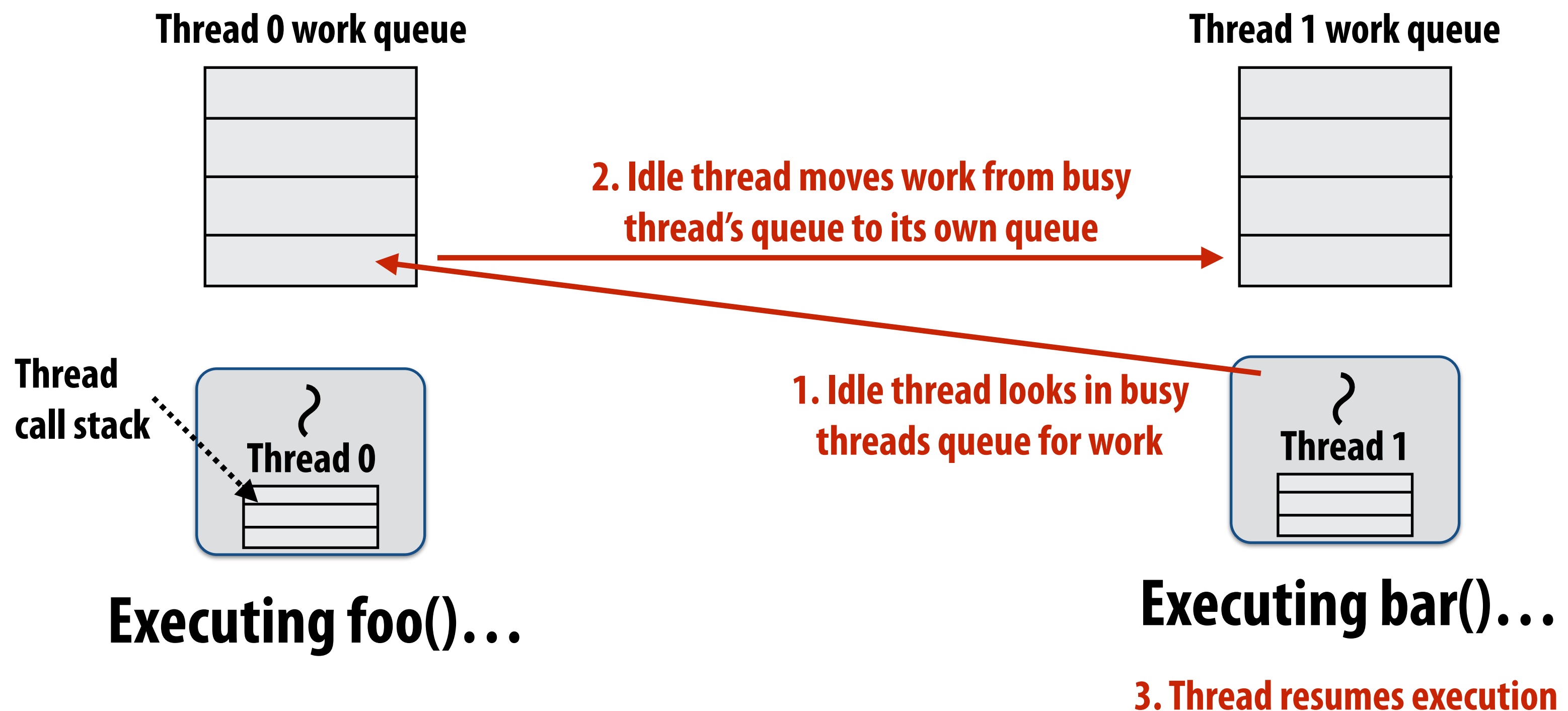
# At spawn, should thread run child or continuation?

```
cilk_spawn foo();
bar();
cilk_sync;
```

spawned child

continuation (rest of calling function)
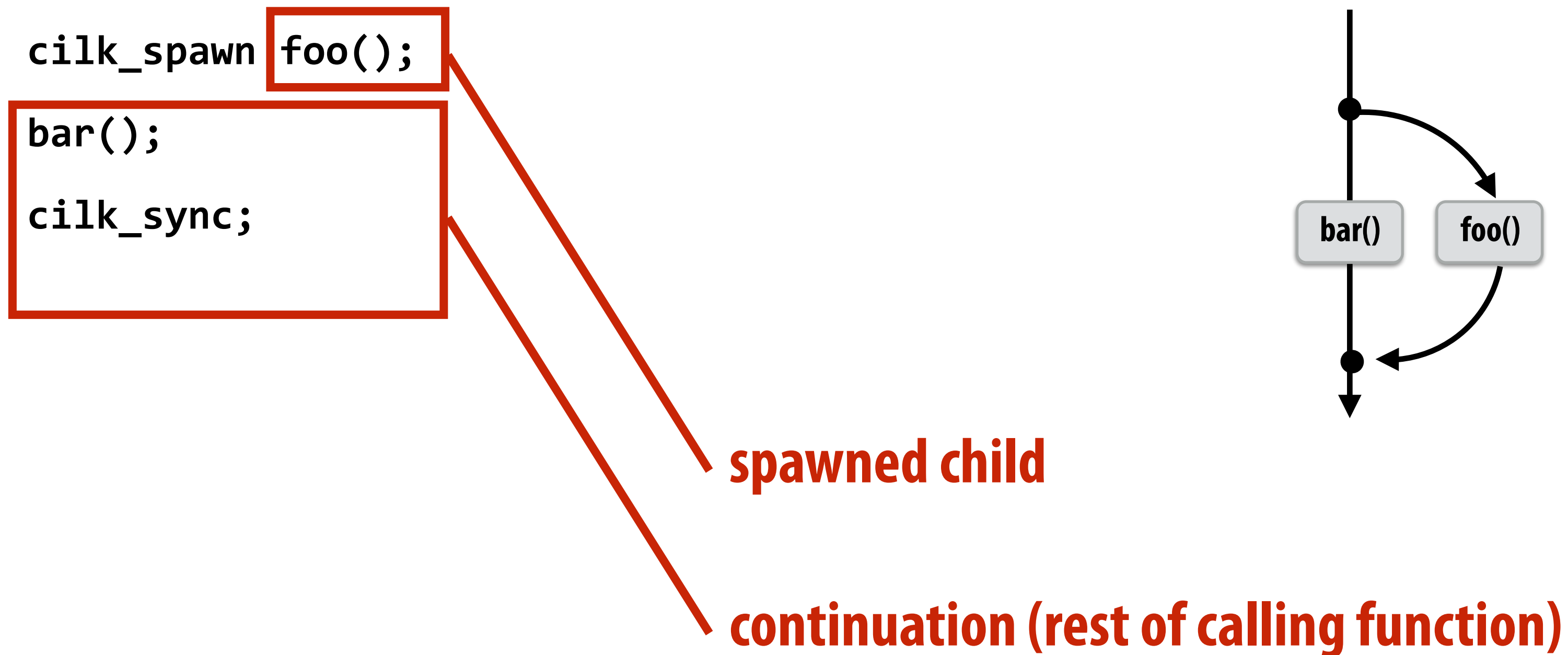
**Run continuation first: record child for later execution**

- **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;
```



foo(N-1)  …  foo(3)  foo(2)  foo(1)  foo(0)
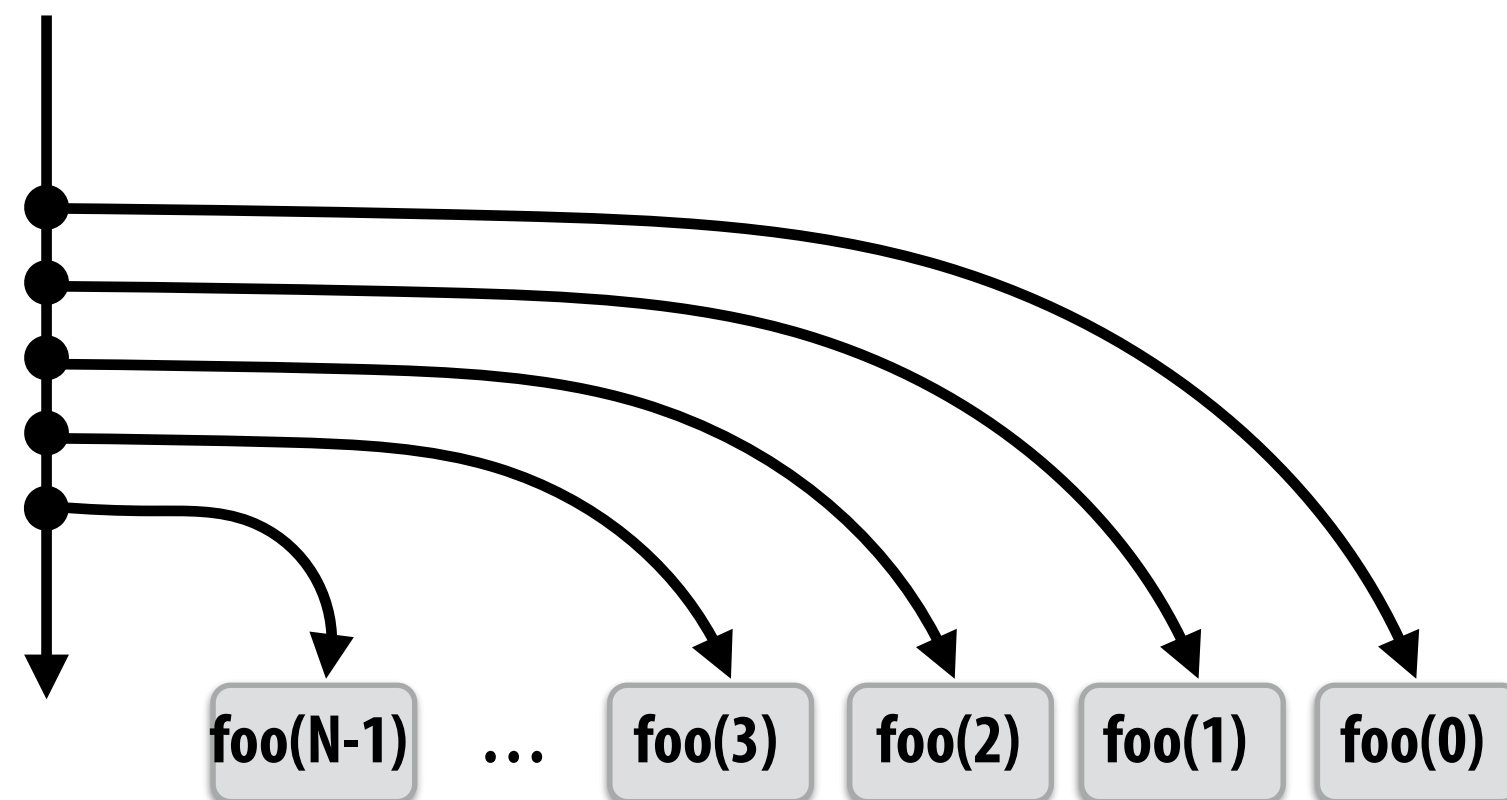
- ## Run continuation first ("child stealing")

  - 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**
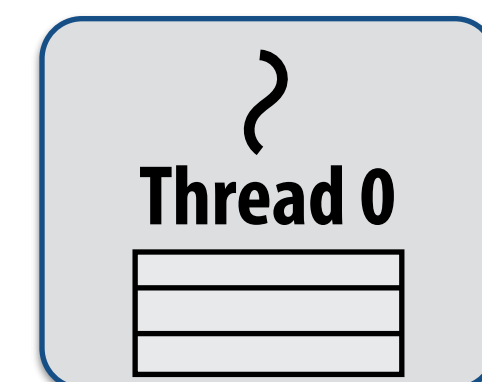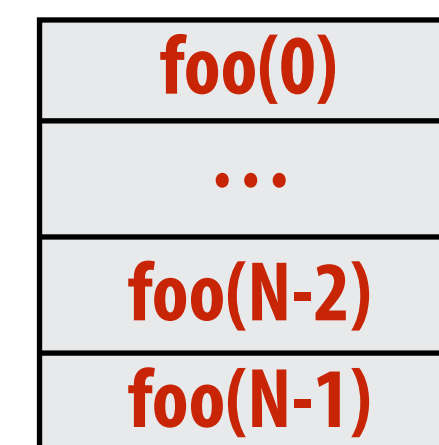
| foo(0) |
| --- |
| … |
| foo(N-2) |
| foo(N-1) |

**Thread 0**

# Consider thread executing the following code

```
for (int i=0; i<N; i++) {

  cilk_spawn foo(i);

}

cilk_sync;
```

| foo(N-1) | ... | foo(3) | foo(2) | foo(1) | foo(0) |
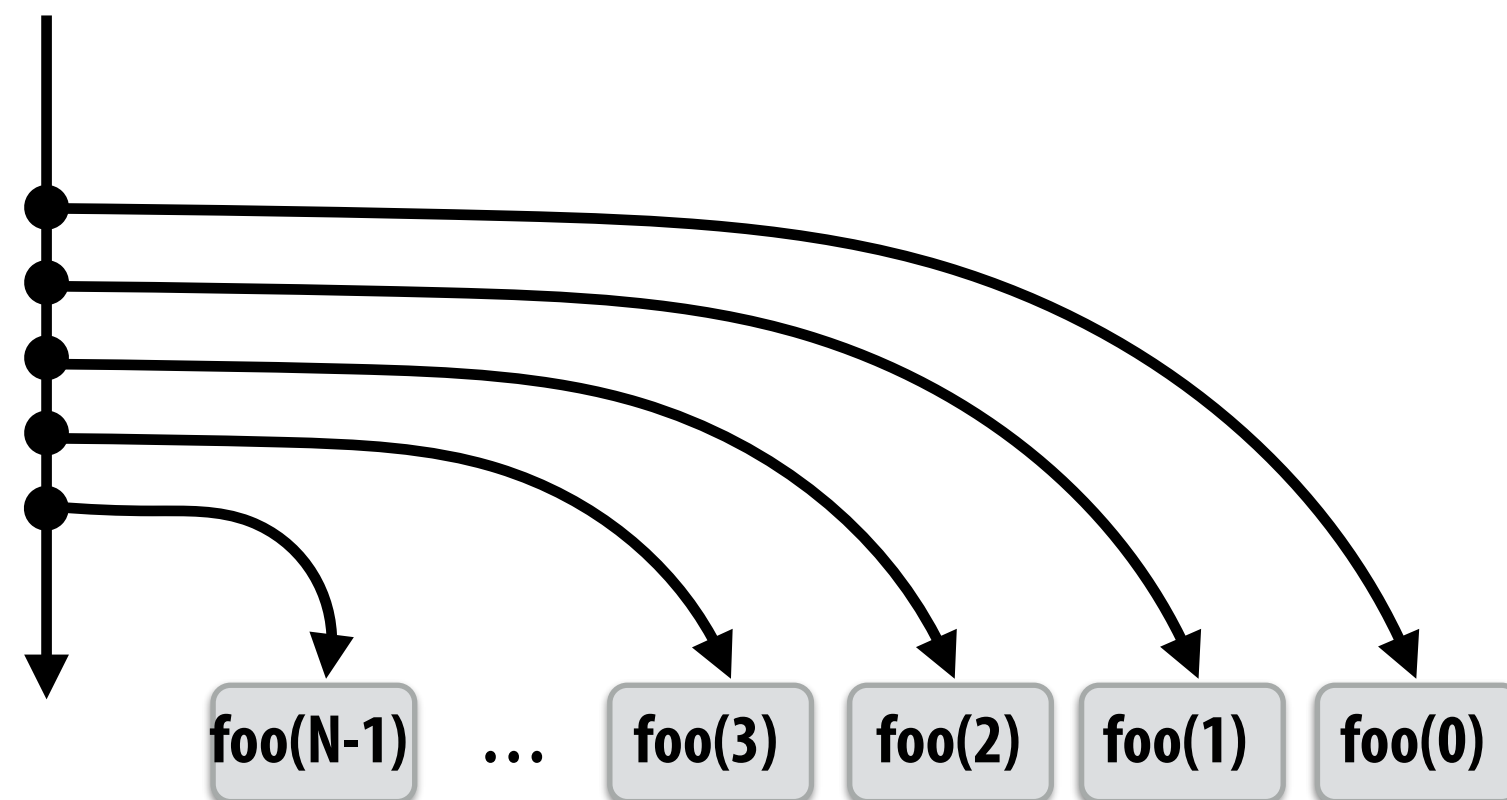
- ## Run child first ("continuation stealing")

  - 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**

| |
| --- |
| |
| |
| **cont: i=1** |
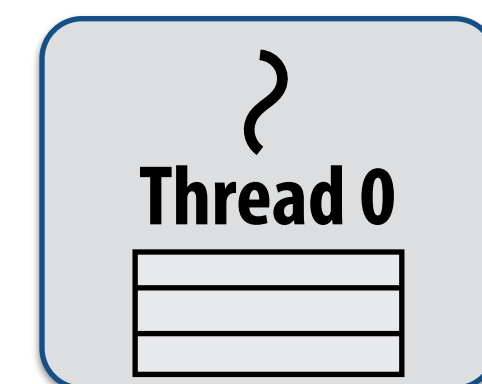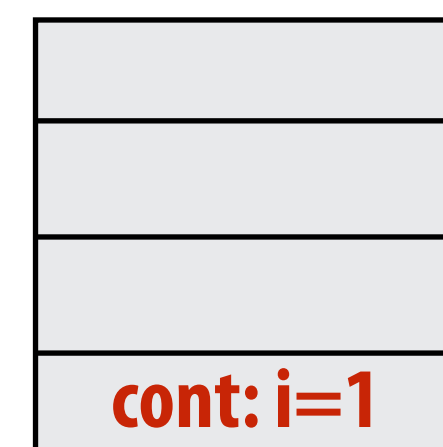
**Thread 0**

**Executing foo(0)...**

# Consider thread executing the following code

```
for (int i=0; i<N; i++) {

  cilk_spawn foo(i);

}

cilk_sync;
```

foo(N-1) ... foo(3) foo(2) foo(1) foo(0)
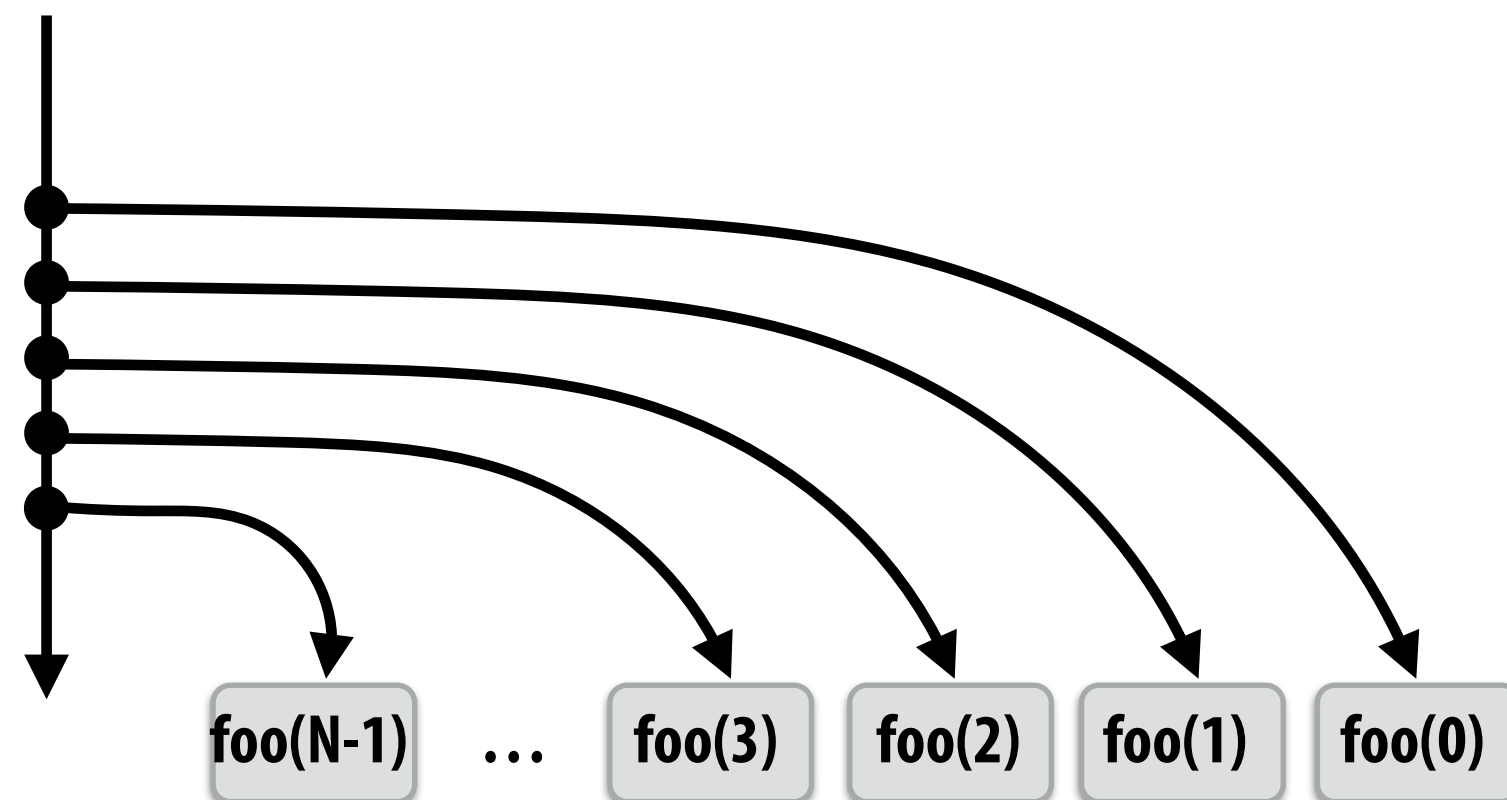
- **Run child first ("continuation stealing")**

  - **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**

cont: i=1
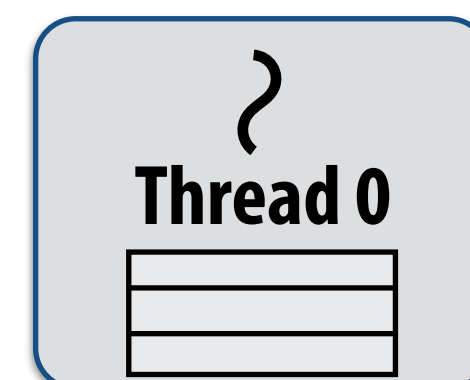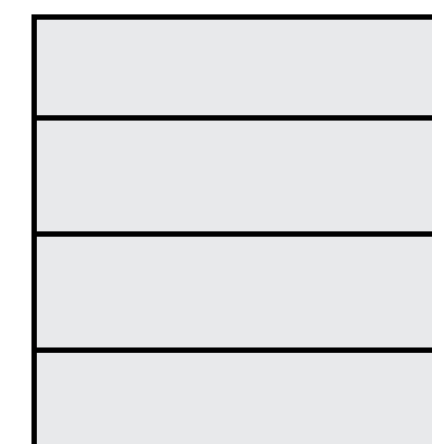
Thread 0

Thread 0
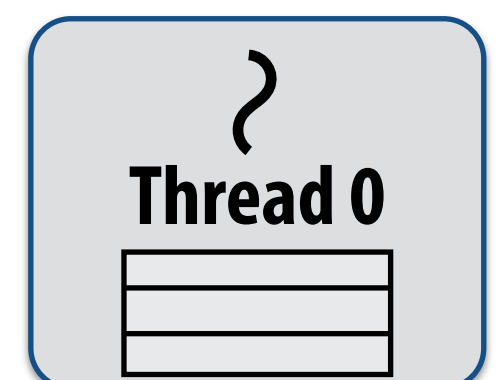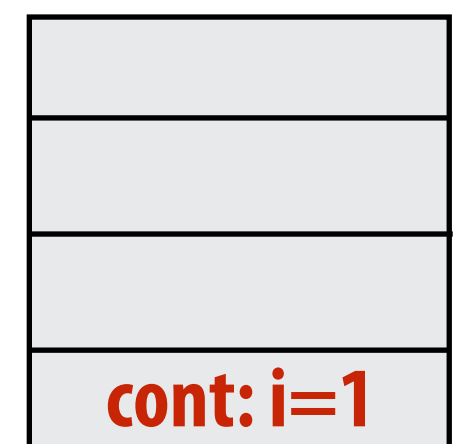
**Executing foo(0)…**

**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?**

**Thread 0 work queue**
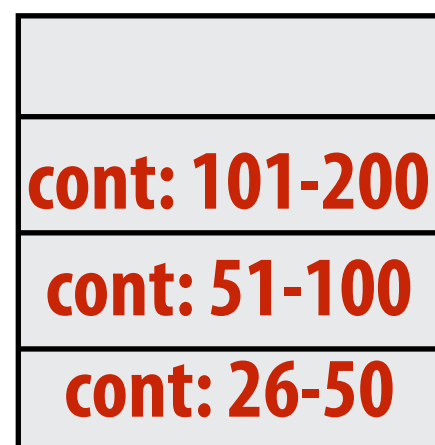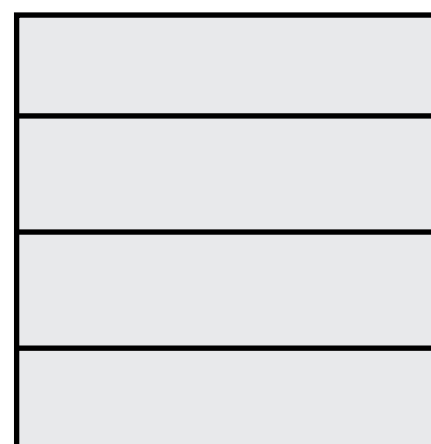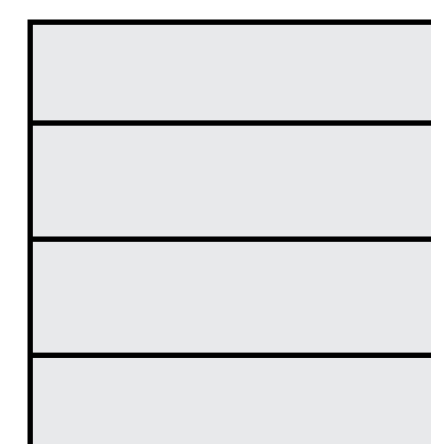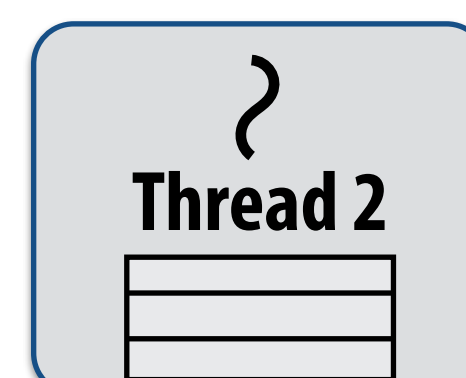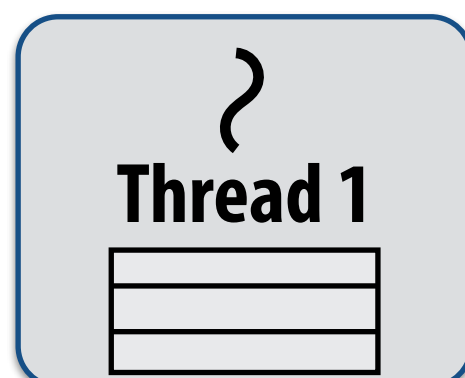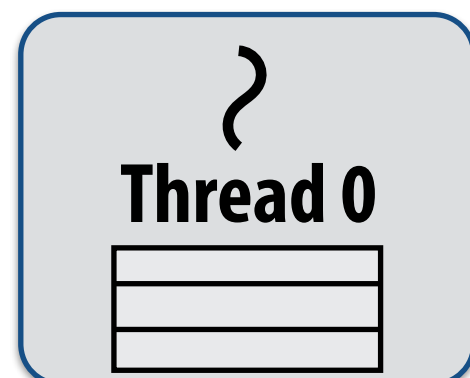
|  |
|---|
| cont: 101-200 |
| cont: 51-100 |
| cont: 26-50 |

**Thread 1 work queue**

|  |
|---|
|  |
|  |
|  |

...

**Thread 2 work queue**

|  |
|---|
|  |
|  |
|  |



**Thread 0**



**Thread 1**



**Thread 2**

**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)



**Thread 0 work queue**

**Thread 1 work queue**

**Thread 2 work queue**

cont: 26-50

cont: 101-200

cont: 51-100

**Steal!**

**Steal!**

...

Thread 0

Thread 1

Thread 2

**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)**

**Thread 0 work queue**

| |
|---|
| |
| |
| |
| **cont: 26-50** |

**Thread 1 work queue**

| |
|---|
| |
| |
| |
| **cont: 151-200** |

**Thread 2 work queue**

| |
|---|
| |
| |
| |
| **cont: 76-100** |

...
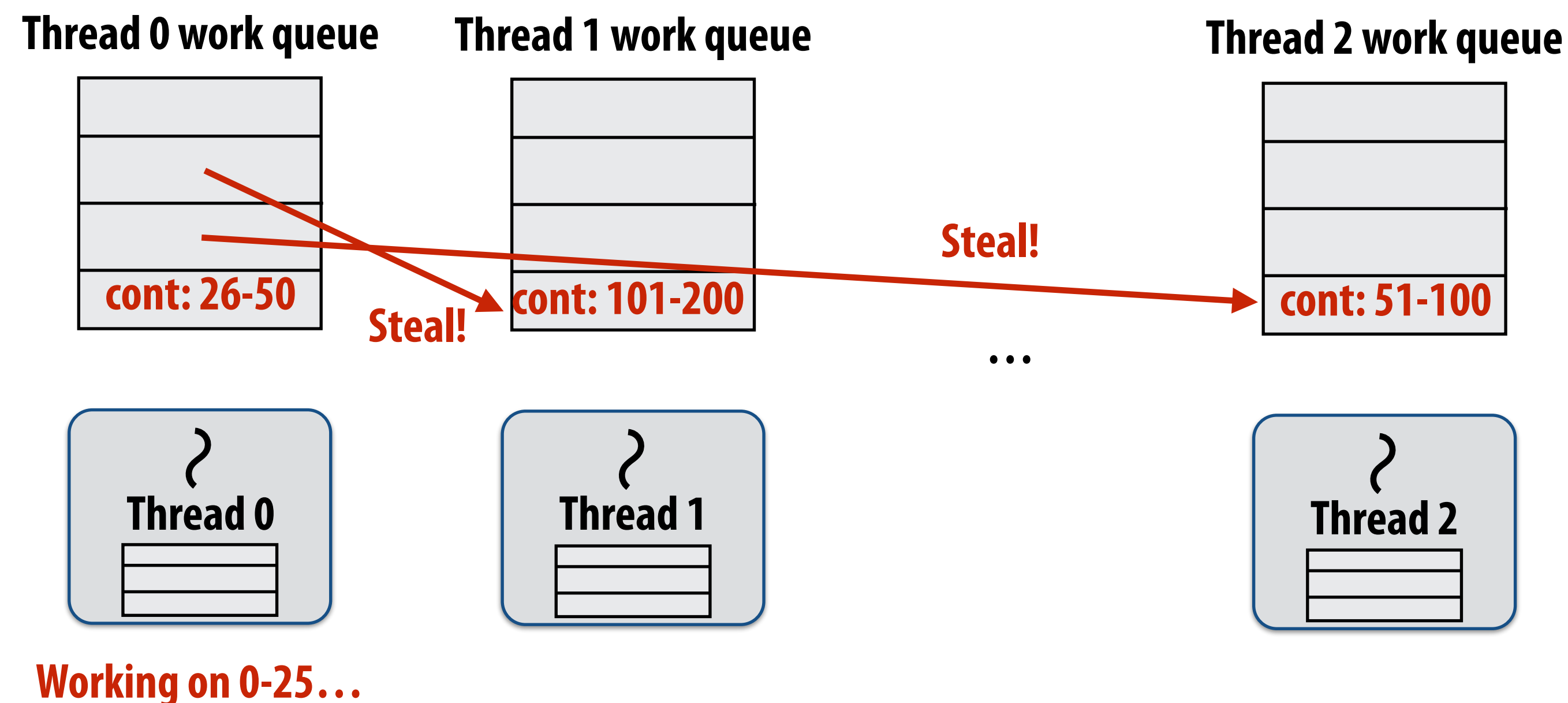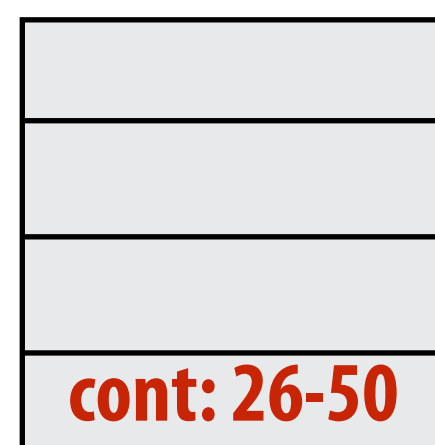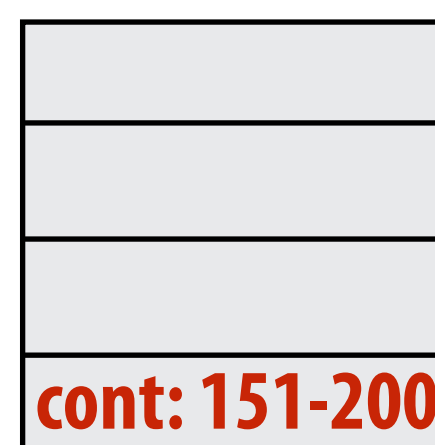
**Thread 0**
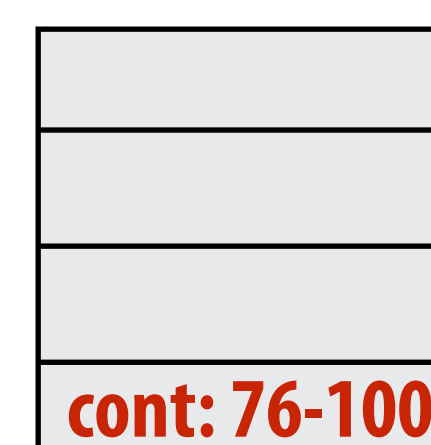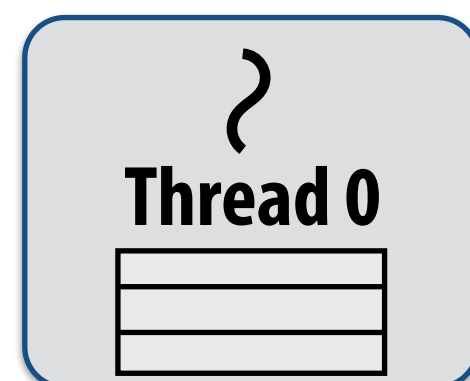
**Thread 1**

**Thread 2**
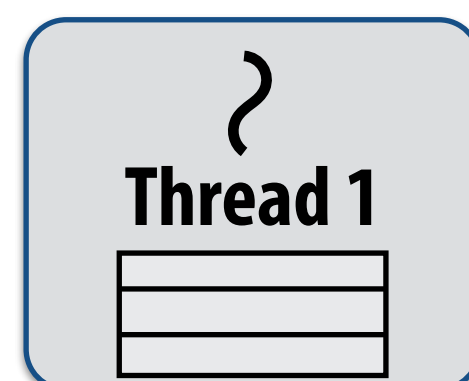
**Working on 0-25...**   **Working on 101-150...**   **Working on 51-75...**
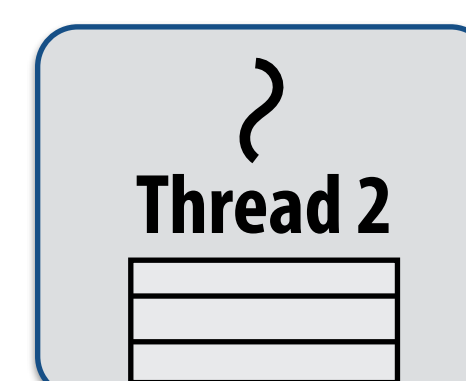
# Implementing work stealing: dequeue per worker

## 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**

|  |
| --- |
|  |
| **cont: 26-50** |
| **cont: 13-25** |

**Thread 1 work queue**

|  |
| --- |
| **cont: 151-200** |
| **cont: 126-150** |
| **cont: 114-125** |

**Thread 2 work queue**

|  |
| --- |
|  |
| **cont: 76-100** |
| **cont: 64-75** |

...
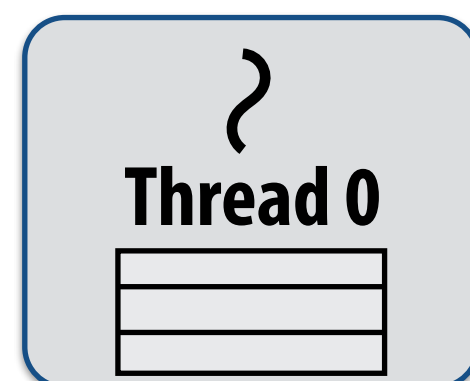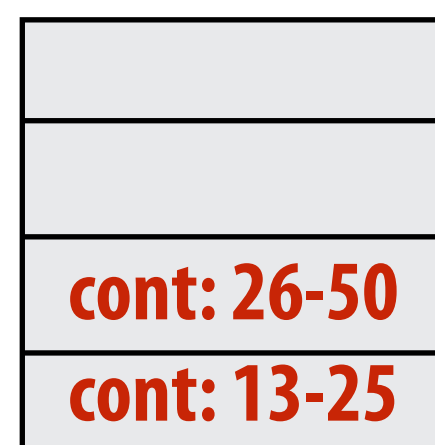
**Thread 0**
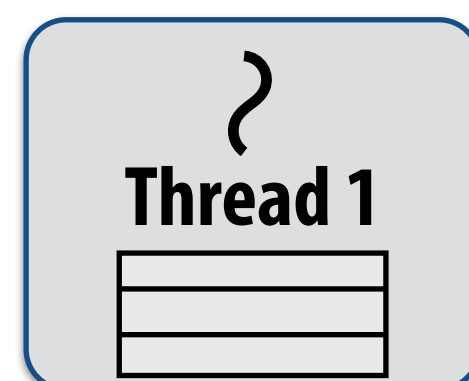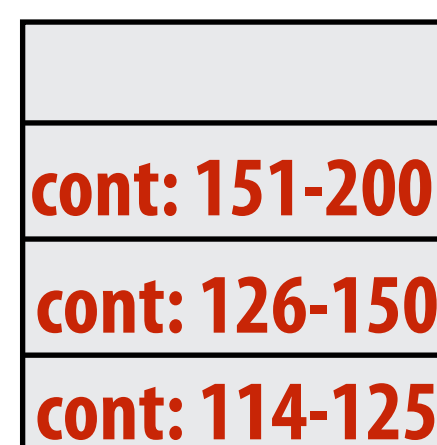
**Thread 1**

**Thread 2**

**Working on 0-12...**

**Working on 101-113...**
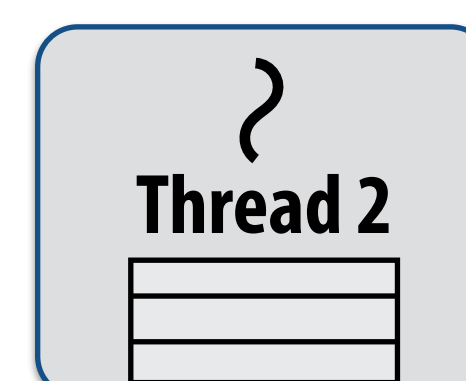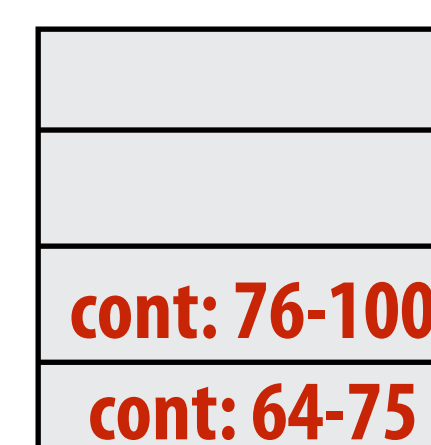
**Working on 51-63...**

# 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)

- Stealing thread and local thread don't contend for same elements of dequeue (efficient lock-free implementations of dequeue exist)

**Thread 0 work queue**   **Thread 1 work queue**                    **Thread 2 work queue**

|  |
|---|
| |
| |
| **cont: 26-50** |
| **cont: 13-25** |

| |
|---|
| |
| **cont: 151-200** |
| **cont: 126-150** |
| **cont: 114-125** |

| |
|---|
| |
| |
| **cont: 76-100** |
| **cont: 64-75** |

. . .
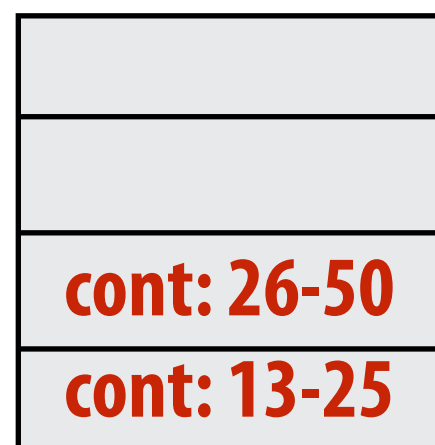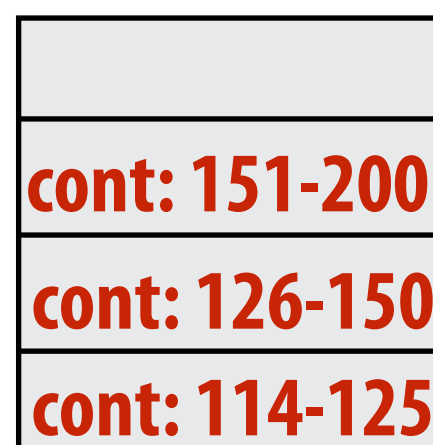
**Thread 0**

**Thread 1**

**Thread 2**
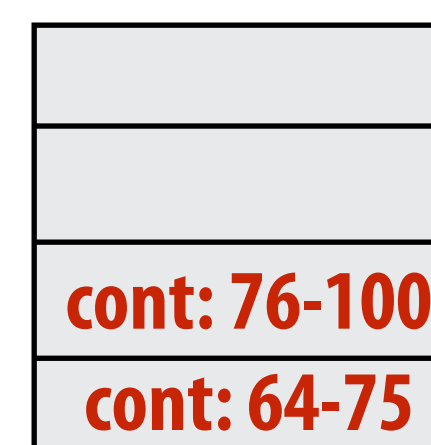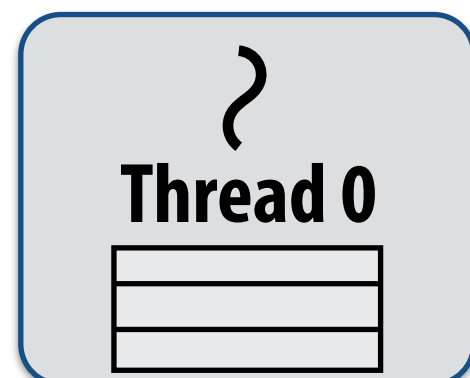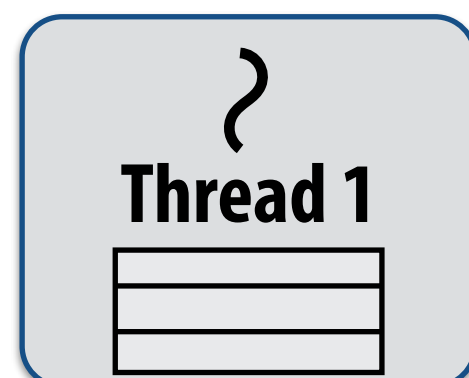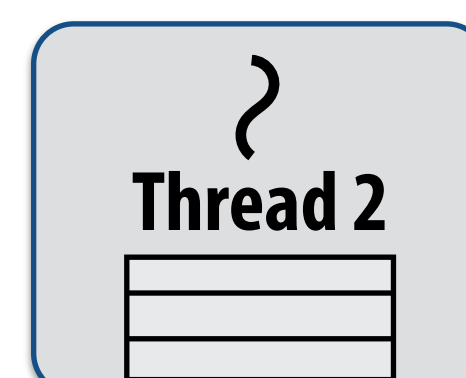
**Working on 0-12…**        **Working on 101-113…**            **Working on 51-63…**

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

```
for (int i=0; i<N; i++) {

  cilk_spawn foo(i);

}

cilk_sync;
```

```
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);
```



foo(N-1) … foo(3) foo(2) foo(1) foo(0)
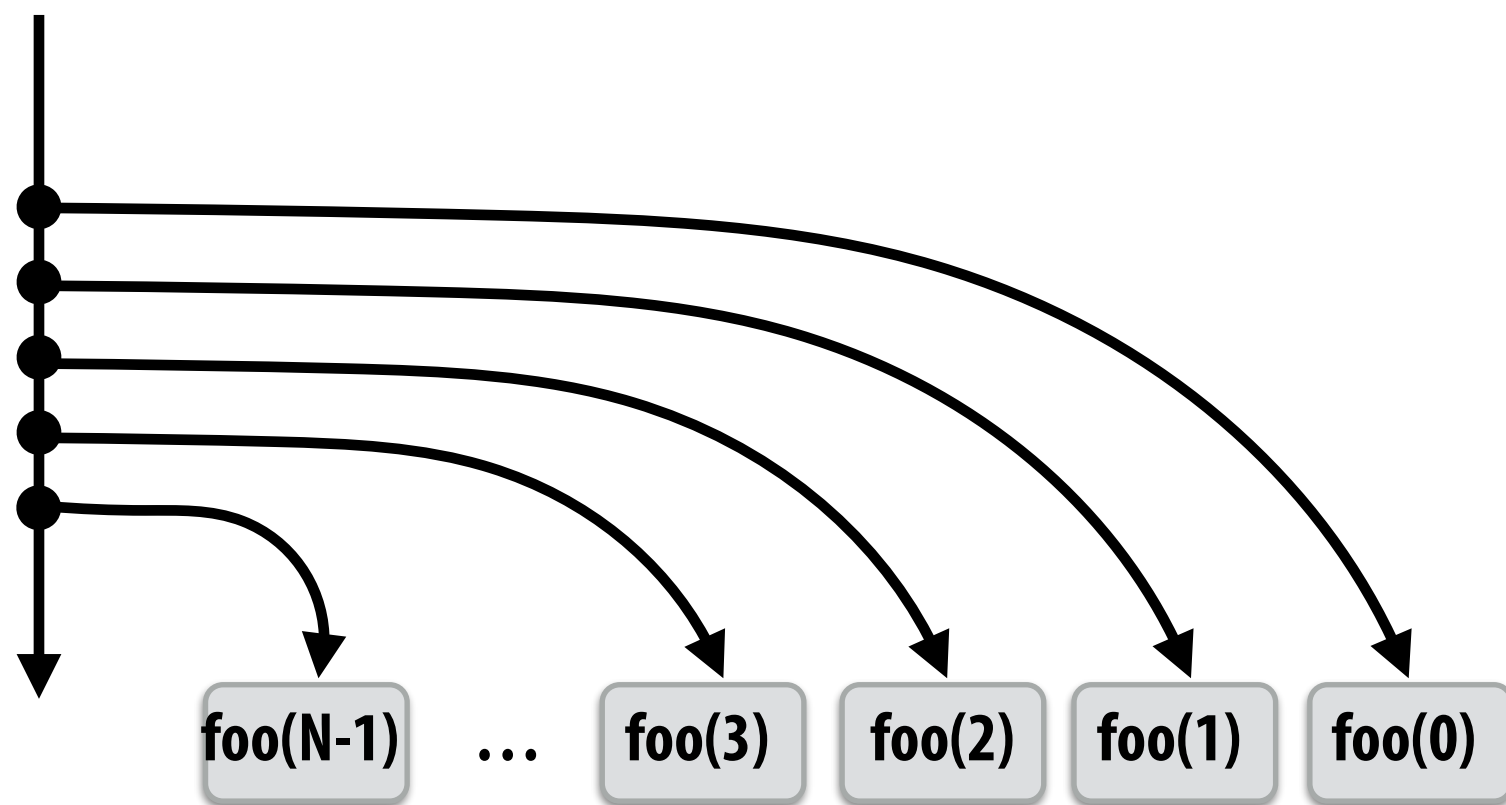
**Code at right generates work in parallel, (code at left does not), so it more quickly fills machine**

(N/2, 3N/4)
(N/2, 7N/8)
(0, N/2)
(N/4, 3N/8)
(0, N/4)
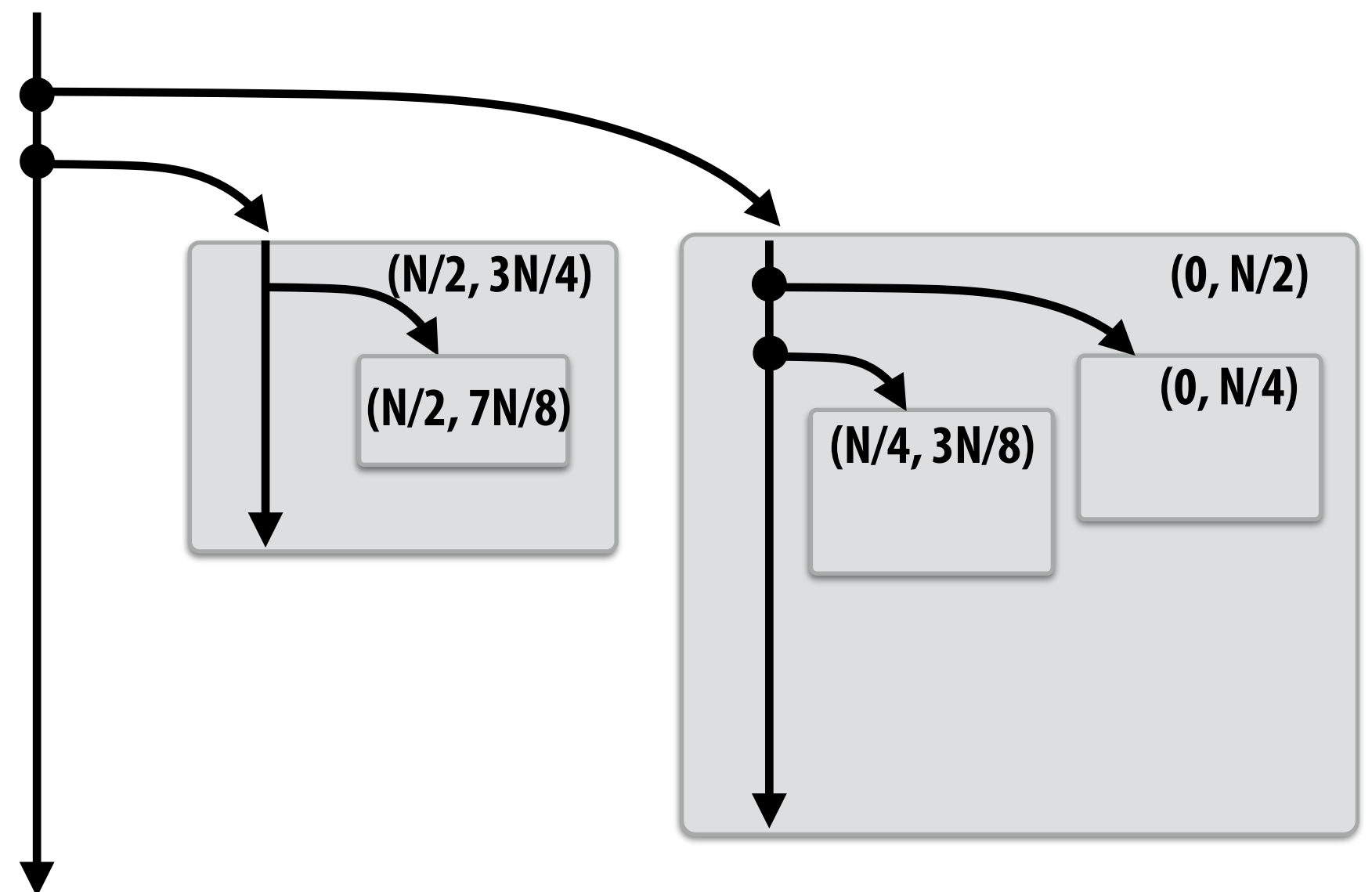
# Implementing sync

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

    cilk_spawn foo(i);

}

cilk_sync;

bar();
```



foo(9)  ...  foo(3)  foo(2)  foo(1)  foo(0)

bar()
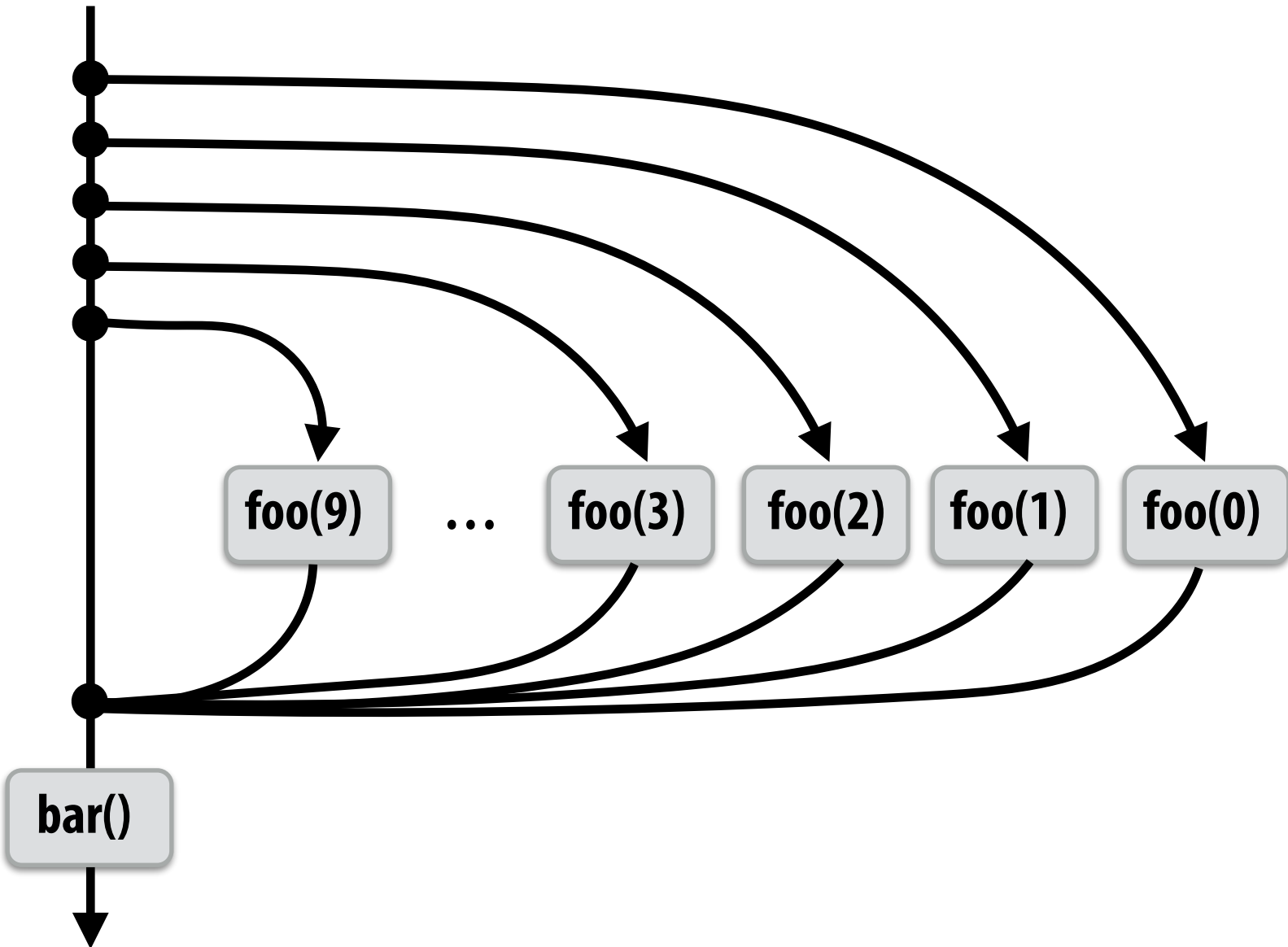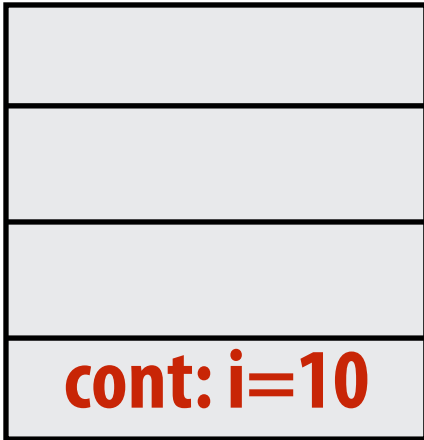
| Thread 0 work queue | Thread 1 work queue | Thread 2 work queue | Thread 3 work queue |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| **cont: i=10** | | | |

Thread 0

Thread 1

Thread 2

Thread 3

**State of worker threads when all work from loop is nearly complete**
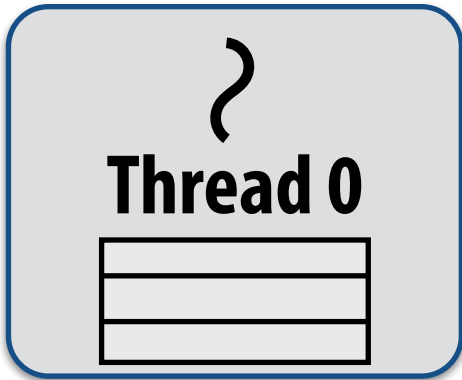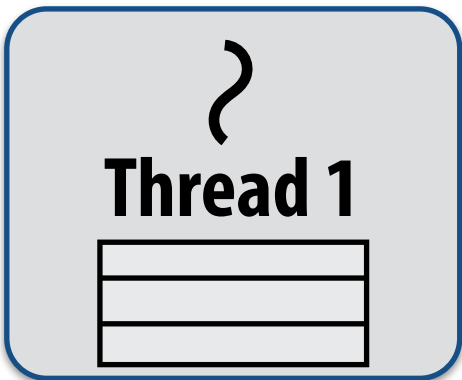
**Working on foo(9)…**      **Working on foo(7)…**      **Working on foo(8)…**      **Working on foo(6)…**
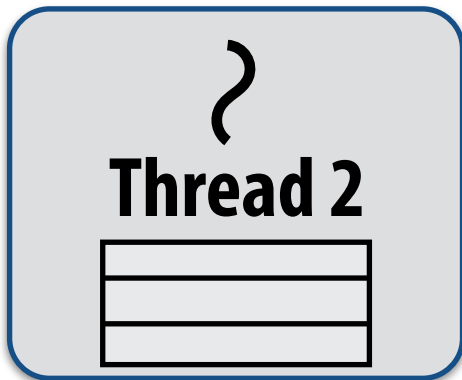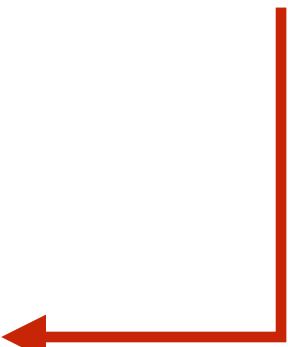
# Implementing sync: no stealing case

block (id: A)
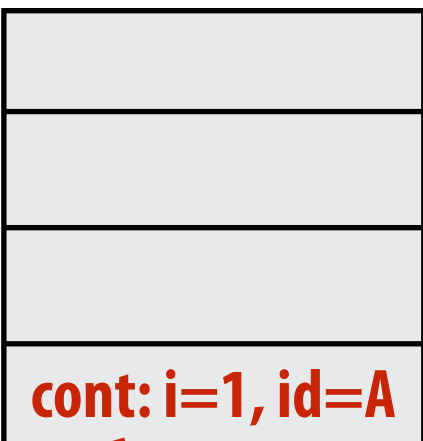
```
for (int i=0; i<10; i++) {

    cilk_spawn foo(i);

}

cilk_sync;   Sync for all calls spawned within block A

bar();
```
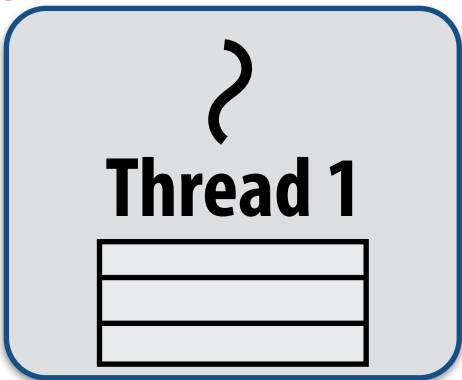


**Thread 0 work queue**

| |
|---|
| |
| |
| |
| cont: i=10 (id=A) |
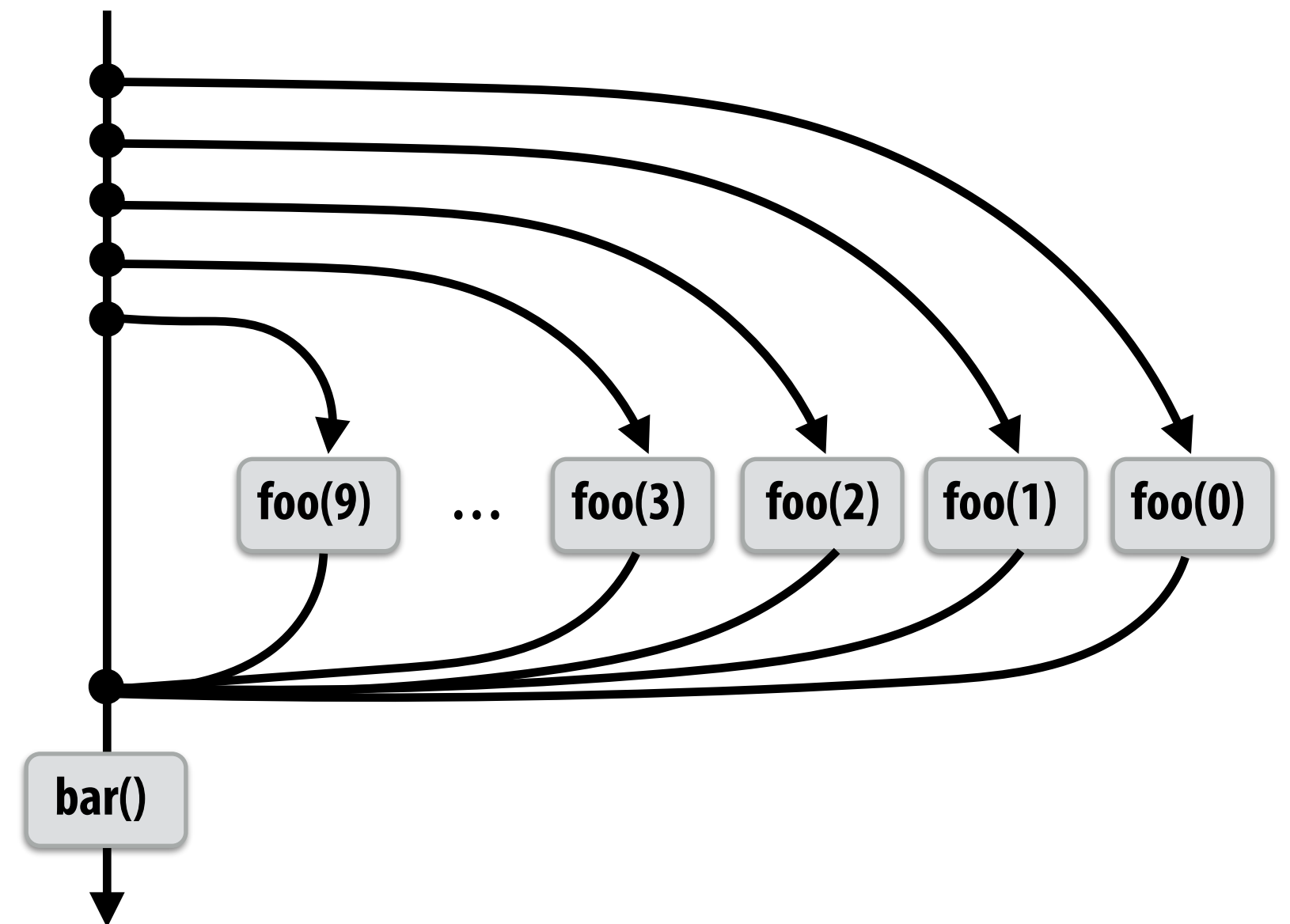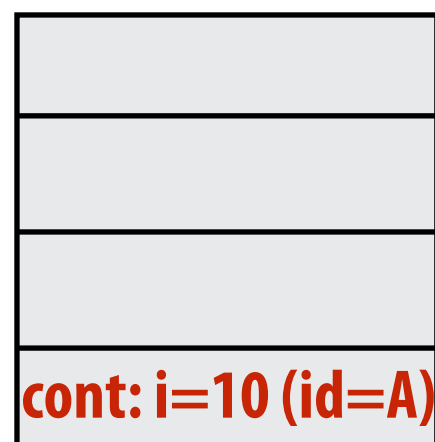
**Thread 1 work queue**

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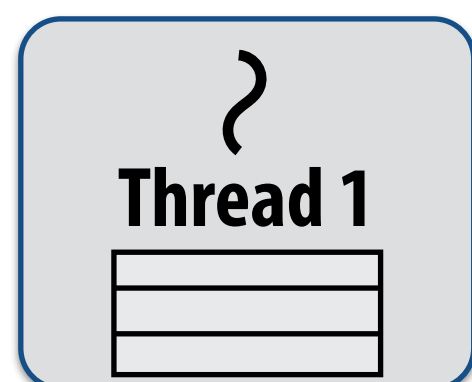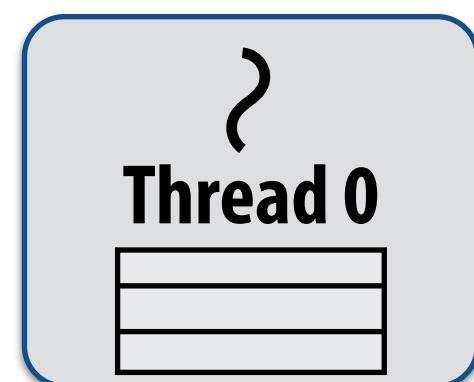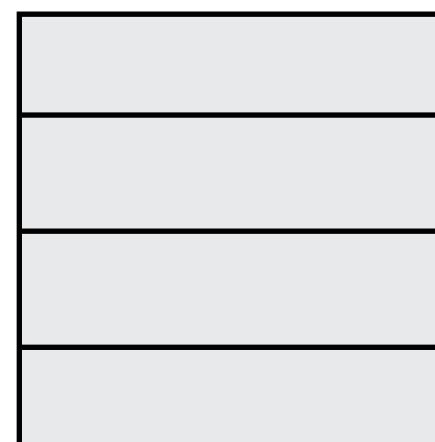
If no work has been stolen by other threads, then there's nothing to do at the sync point.

`cilk_sync` is a no-op.

Thread 0

Thread 1

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();
```
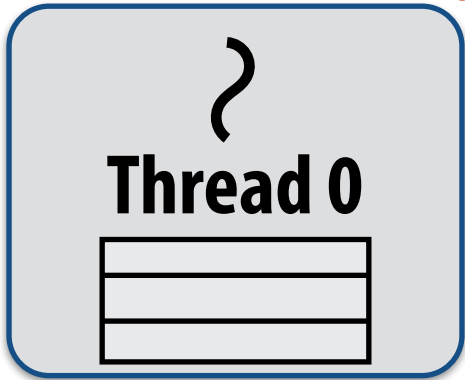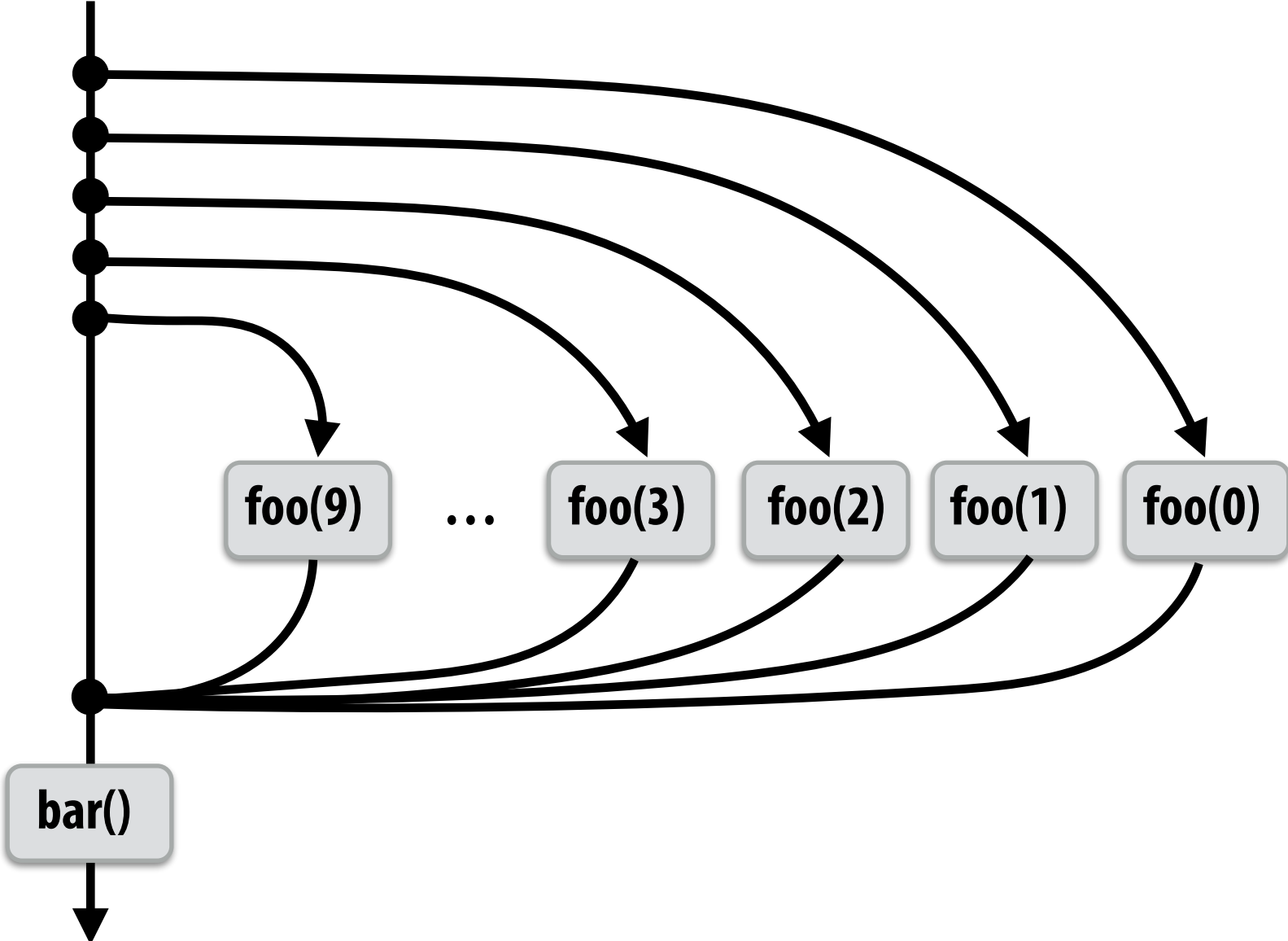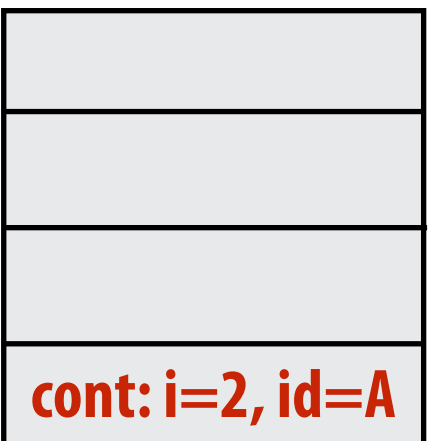


**Thread 0 work queue**

| |
|---|
| |
| |
| |
| cont: i=0 (id=A) |

**Thread 1 work queue**

| |
|---|
| |
| |
| |
| |
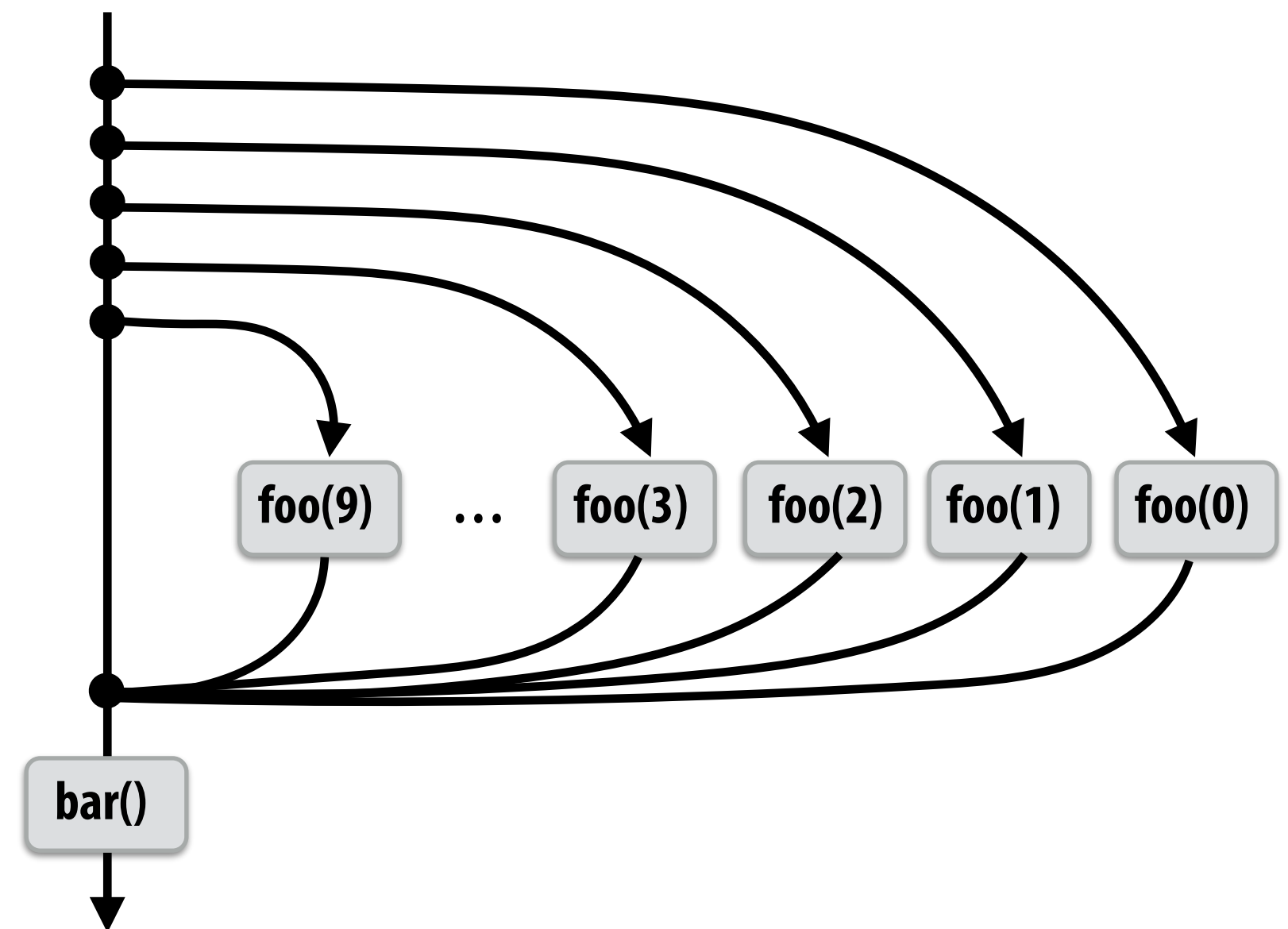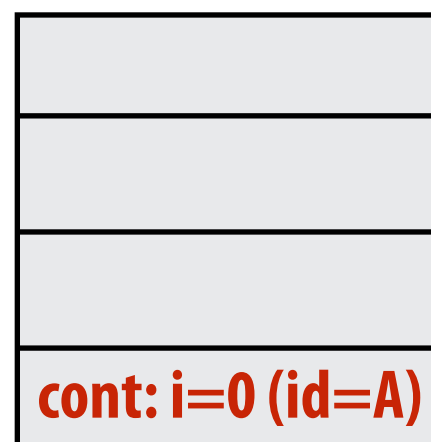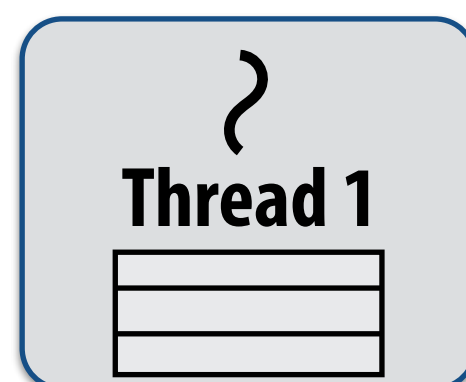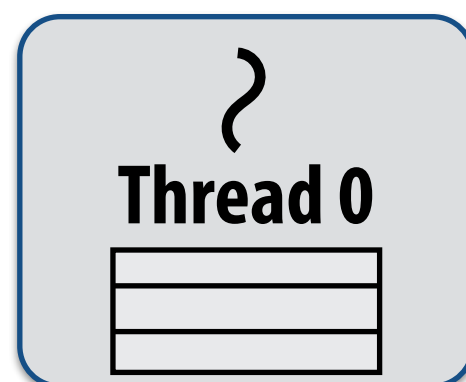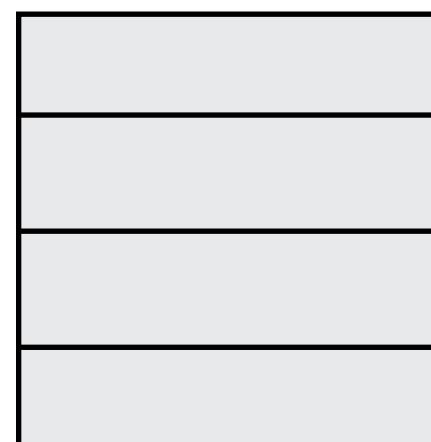
**Thread 0**

**Thread 1**

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();
```
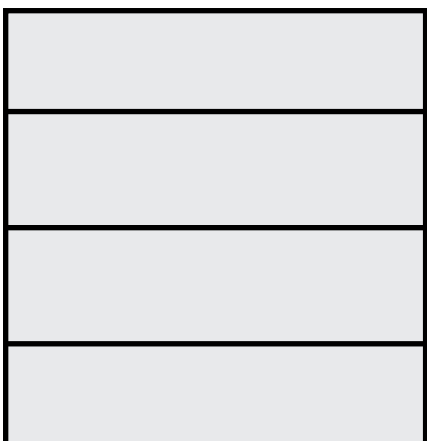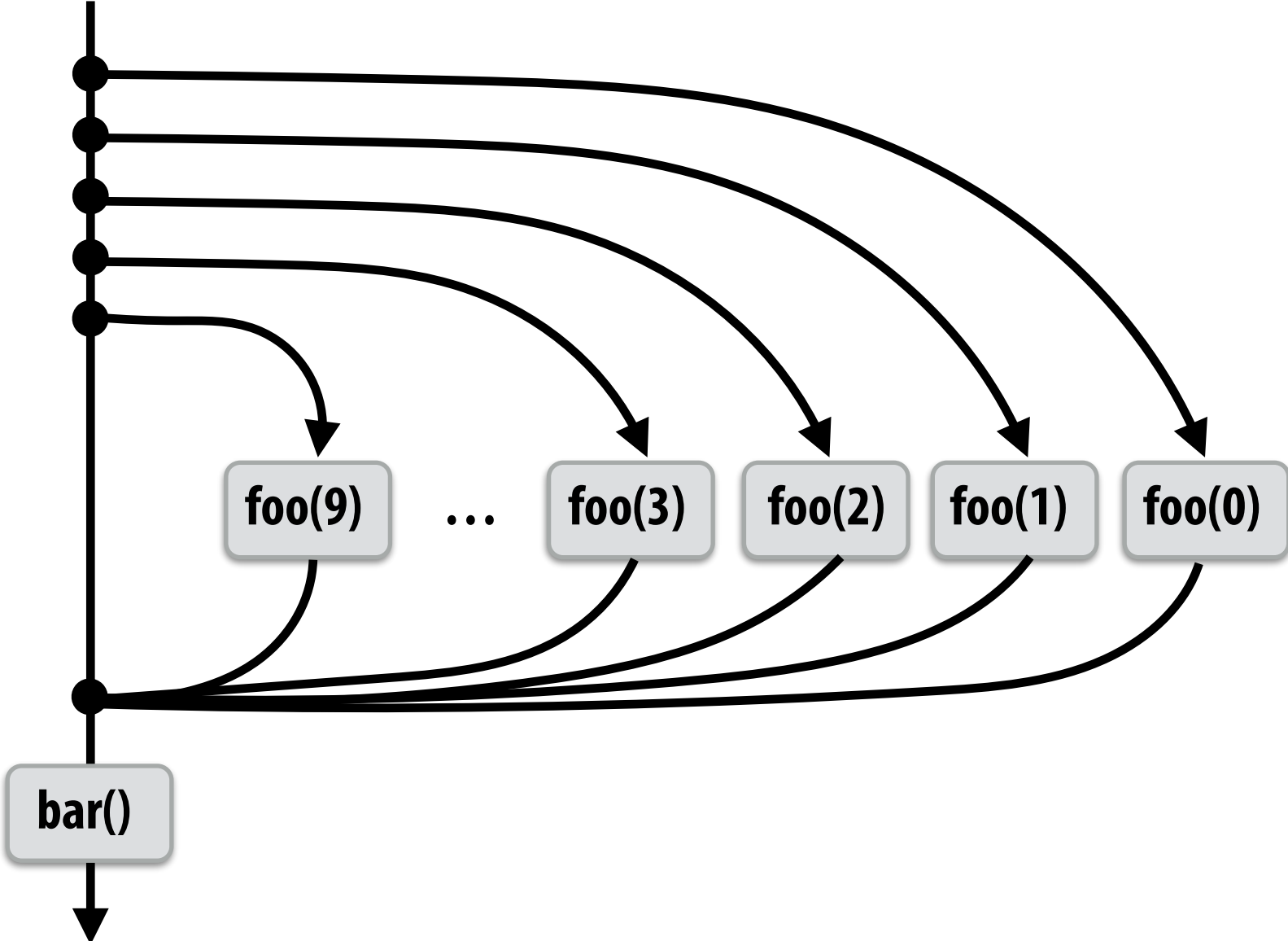
foo(9) … foo(3) foo(2) foo(1) foo(0)

bar()
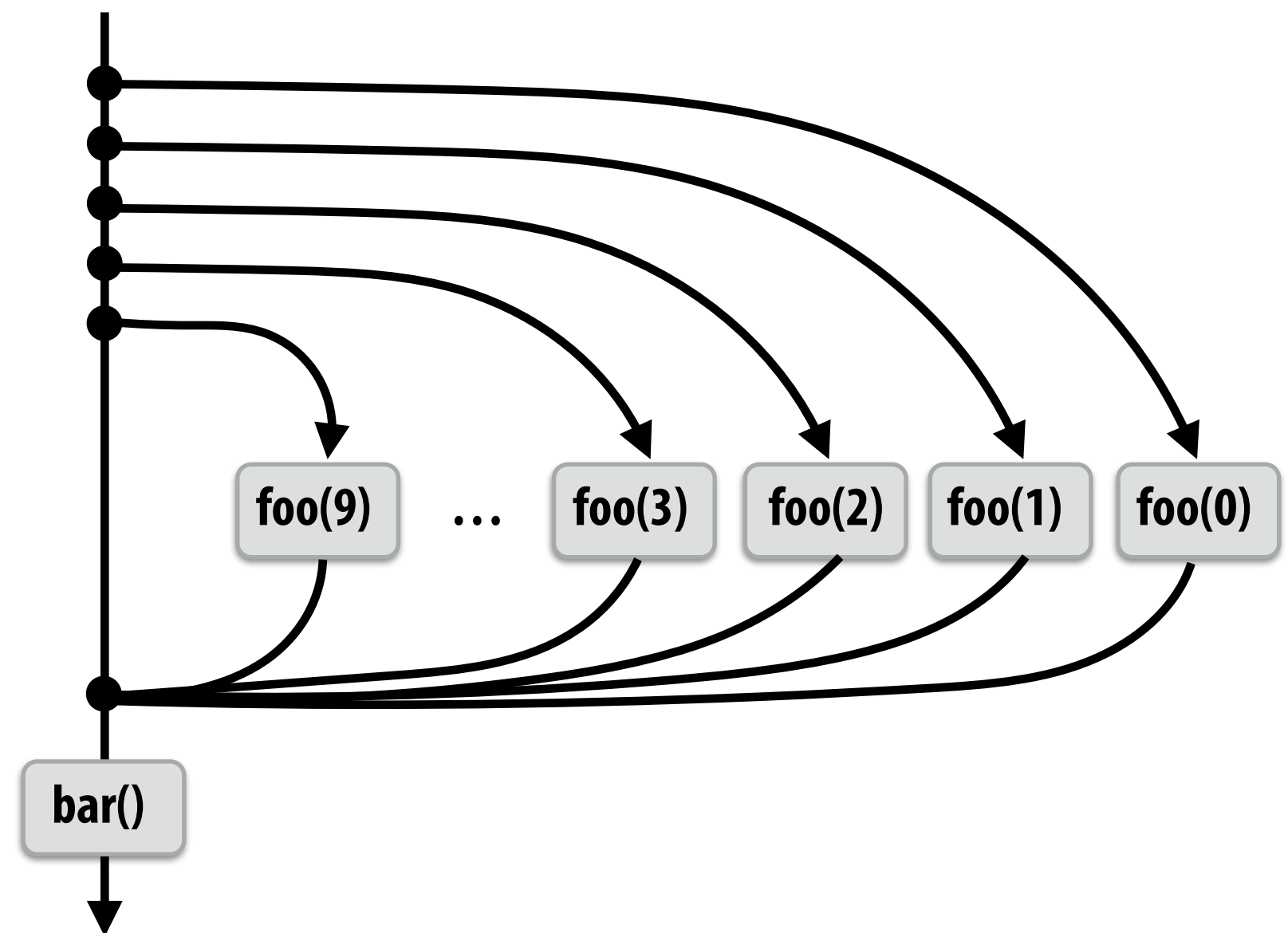
**Thread 0 work queue**     **Thread 1 work queue**

id=A
spawn: 1, done: 0

STOLEN (id=A)     **Steal!**     cont: i=0, id=A
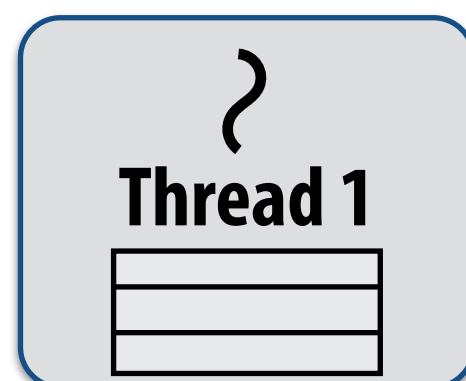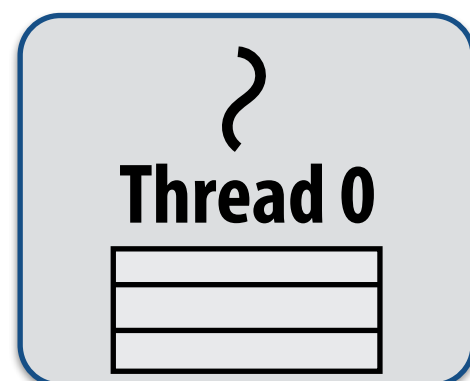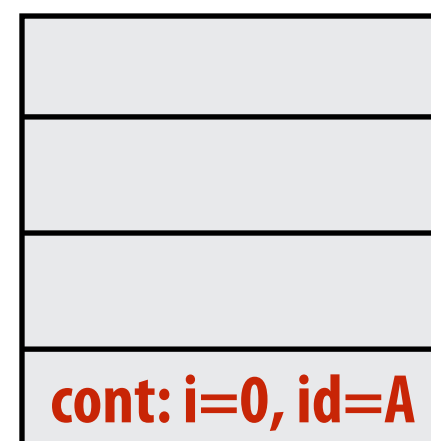
Thread 0     Thread 1

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.

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: 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();
```
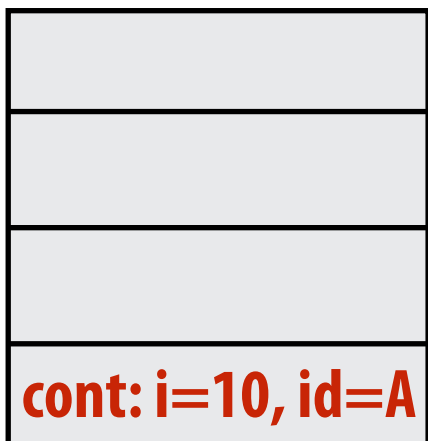
foo(9)   …   foo(3)   foo(2)   foo(1)   foo(0)

bar()

**Thread 0 work queue**

STOLEN (id=A)

**Thread 1 work queue**

cont: i=1, id=A
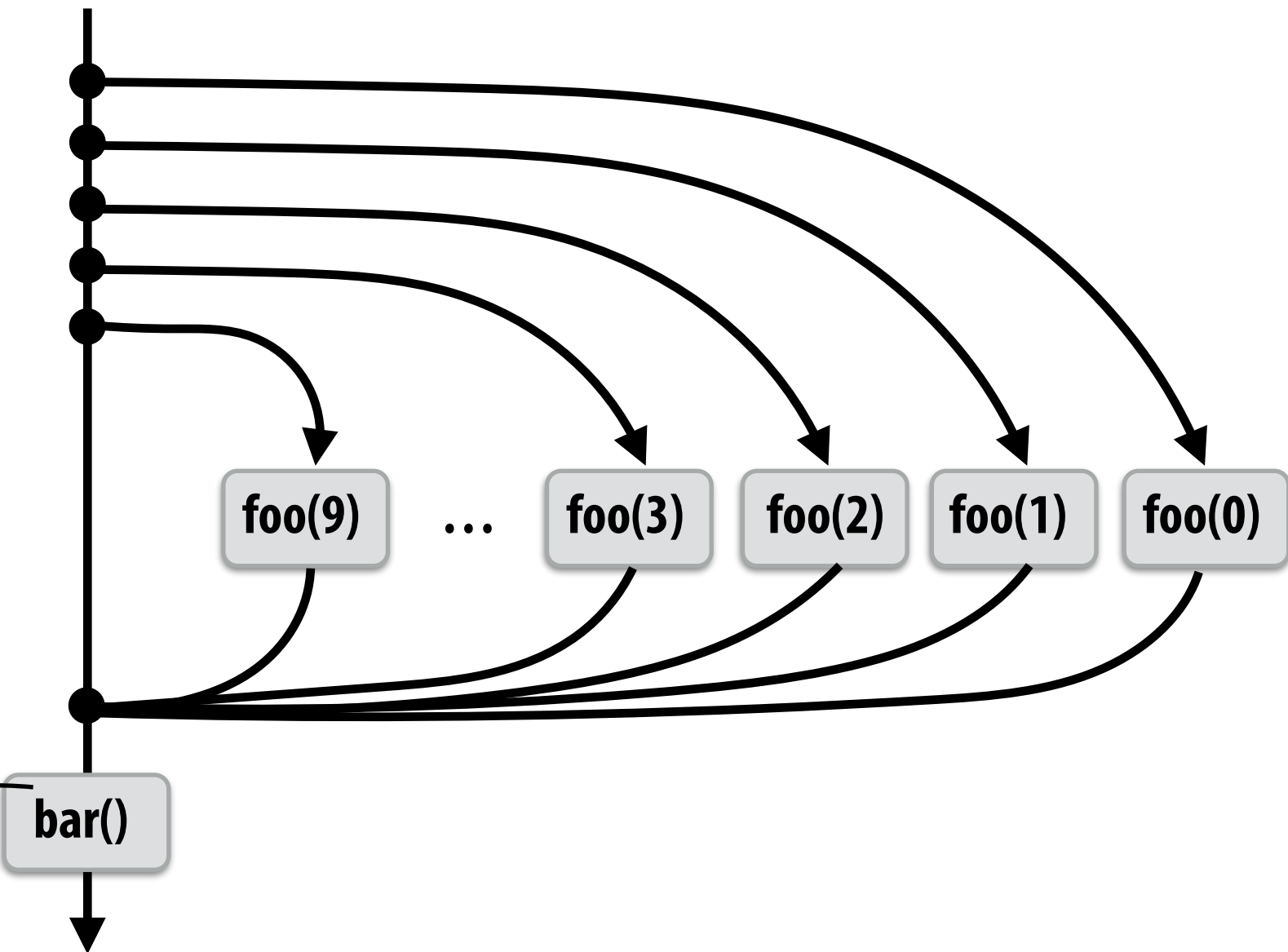
**Thread 1 is now running foo(1)**

**Note: spawn count is now 2**

id=A
spawn: 2, done: 0

**Update count**

**Thread 0**

**Thread 1**

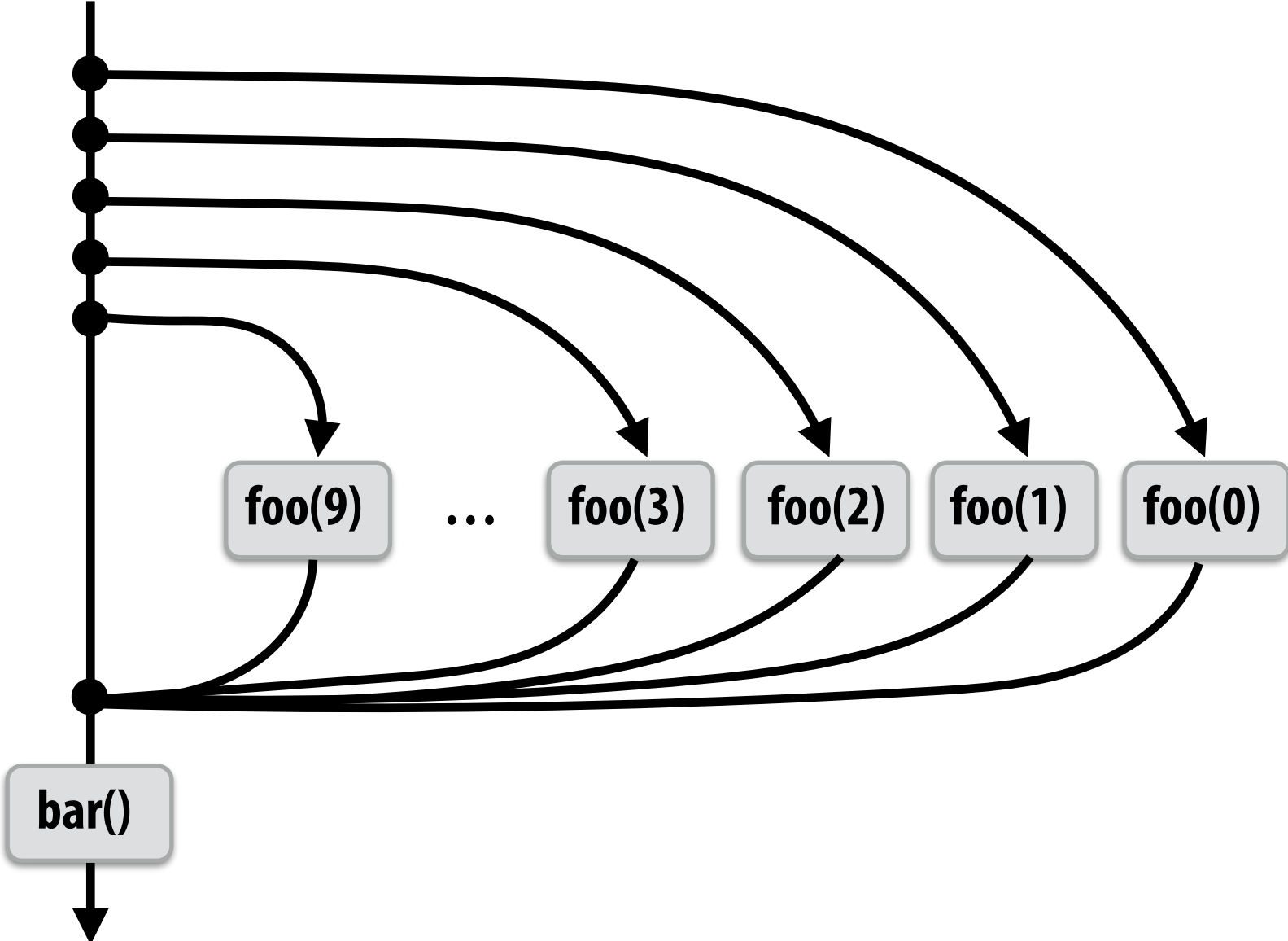**Working on foo(0), id=A...**   **Working on foo(1), id=A...**
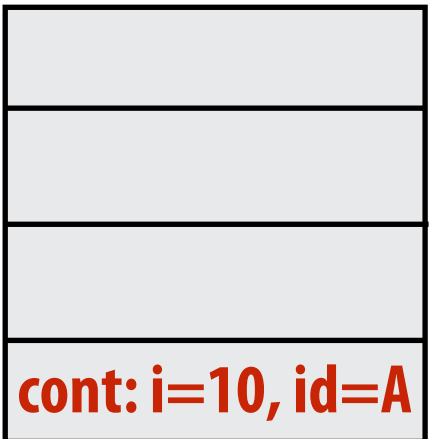
# Implementing sync: stalling join

block (id: A)

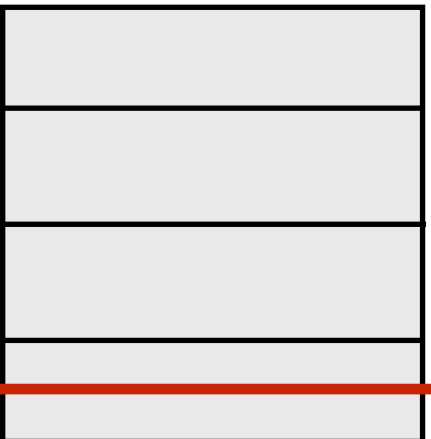```
for (int i=0; i<10; i++) {

    cilk_spawn foo(i);

}

cilk_sync;   Sync for all calls spawned within block A

bar();
```



foo(9)  …  foo(3)  foo(2)  foo(1)  foo(0)

bar()

**Thread 0 work queue**

| |
|---|
| |
| |
| |
| STOLEN (id=A) |

id=A
spawn: 3, done: 1

**Thread 1 work queue**

| |
|---|
| |
| |
| |
| STOLEN (id=A) |

**Steal!**

**Thread 2 work queue**

| |
|---|
| |
| |
| |
| cont: i=2, id=A |

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

**Thread 2 now running foo(2)**

~
**Thread 0**

~
**Thread 1**

~
**Thread 2**
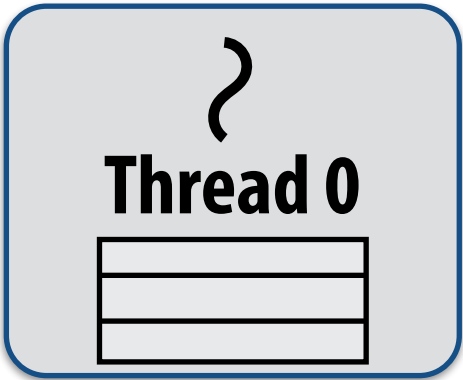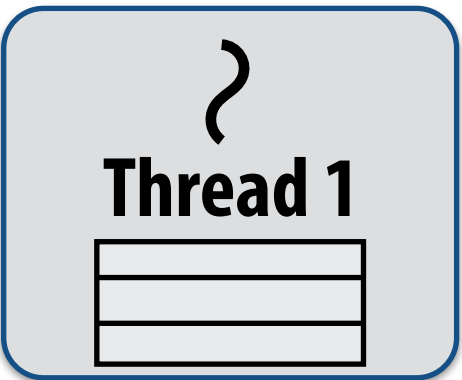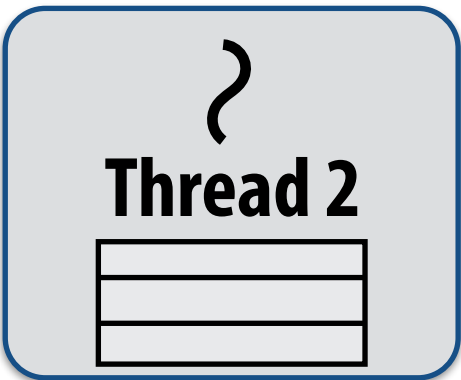
Idle!          Working on foo(1), id=A…    Working on foo(2), id=A…

# 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();
```

foo(9) … foo(3) foo(2) foo(1) foo(0)

bar()

**Thread 0 work queue**

**Thread 1 work queue**

**Thread 2 work queue**

id=A
spawn: 10, done: 9

STOLEN (id=A)

cont: i=10, id=A

**Computation nearing end...**

**Only foo(9) remains to be completed.**
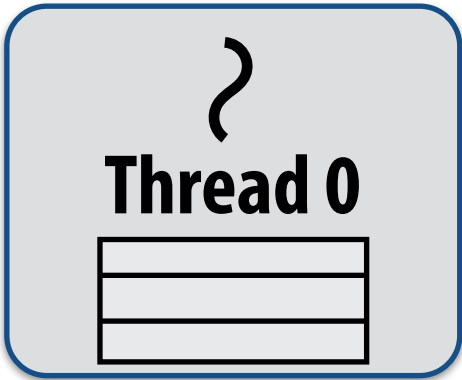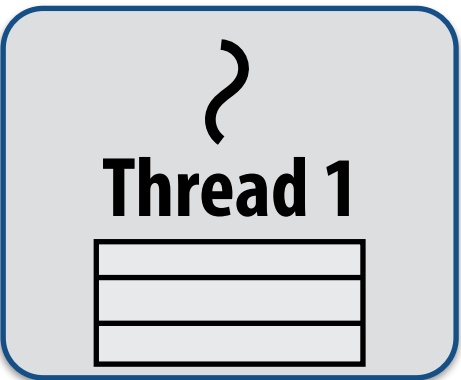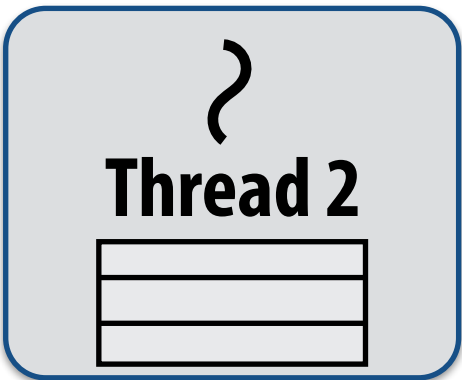
Thread 0

Thread 1

Thread 2

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();
```
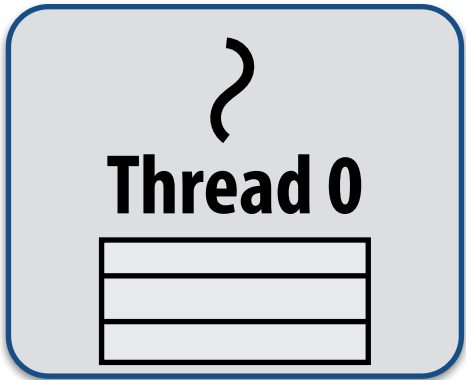
foo(9) ... foo(3) foo(2) foo(1) foo(0)

bar()

**Thread 0 work queue**

**Thread 1 work queue**

**Thread 2 work queue**

**Last spawn completes.**

id=A
spawn: 10, done: 10

~~STOLEN (id=A)~~

**Notify done!**

cont: i=10, id=A
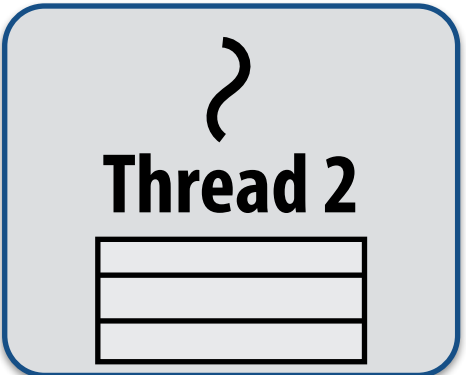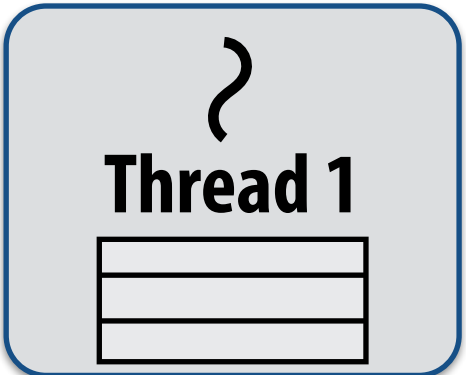
Thread 0

Thread 1

Thread 2

**Idle!**

**Idle!**

**Idle!**

# 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();
```

foo(9)  …  foo(3)  foo(2)  foo(1)  foo(0)
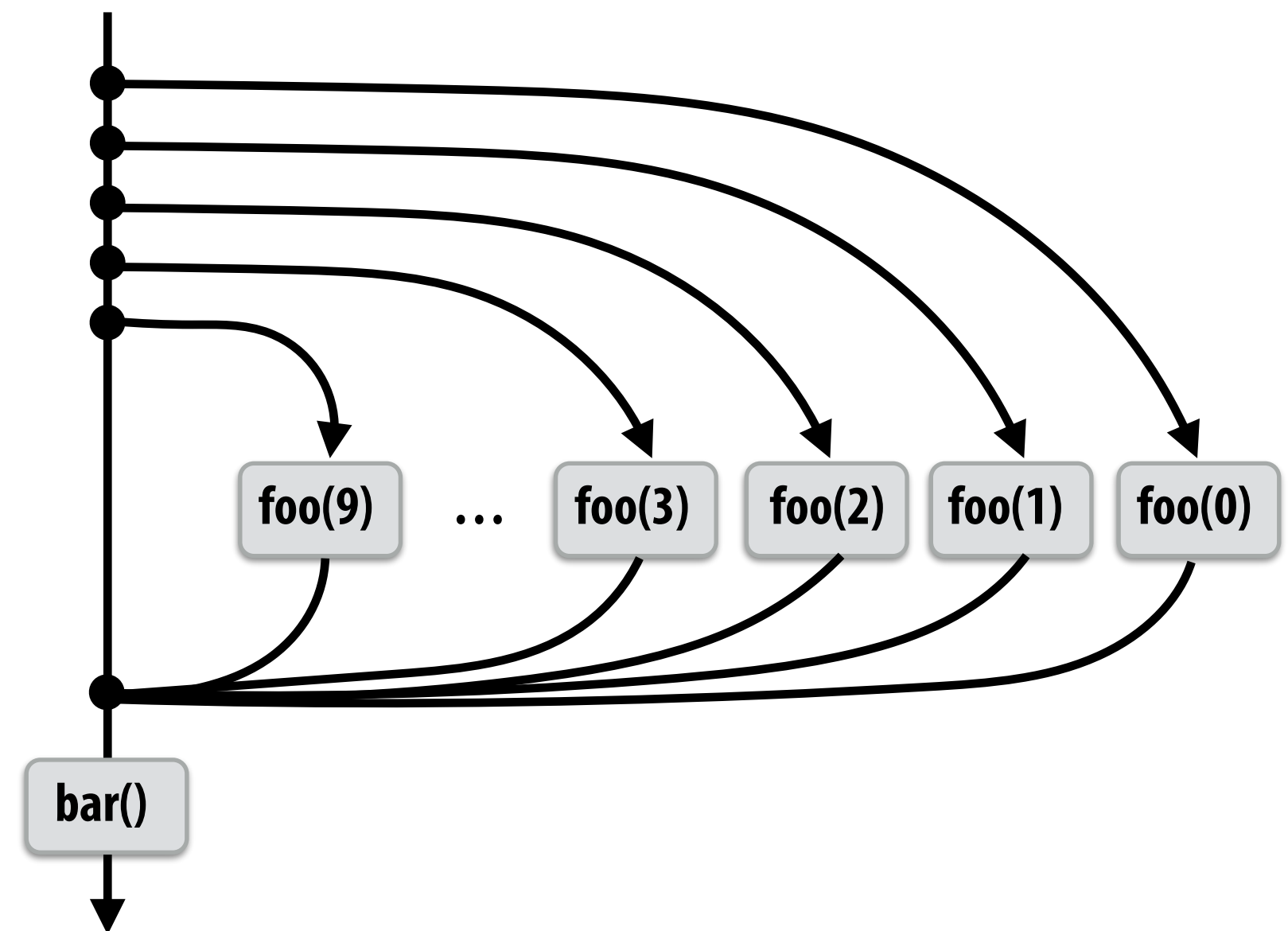
bar()
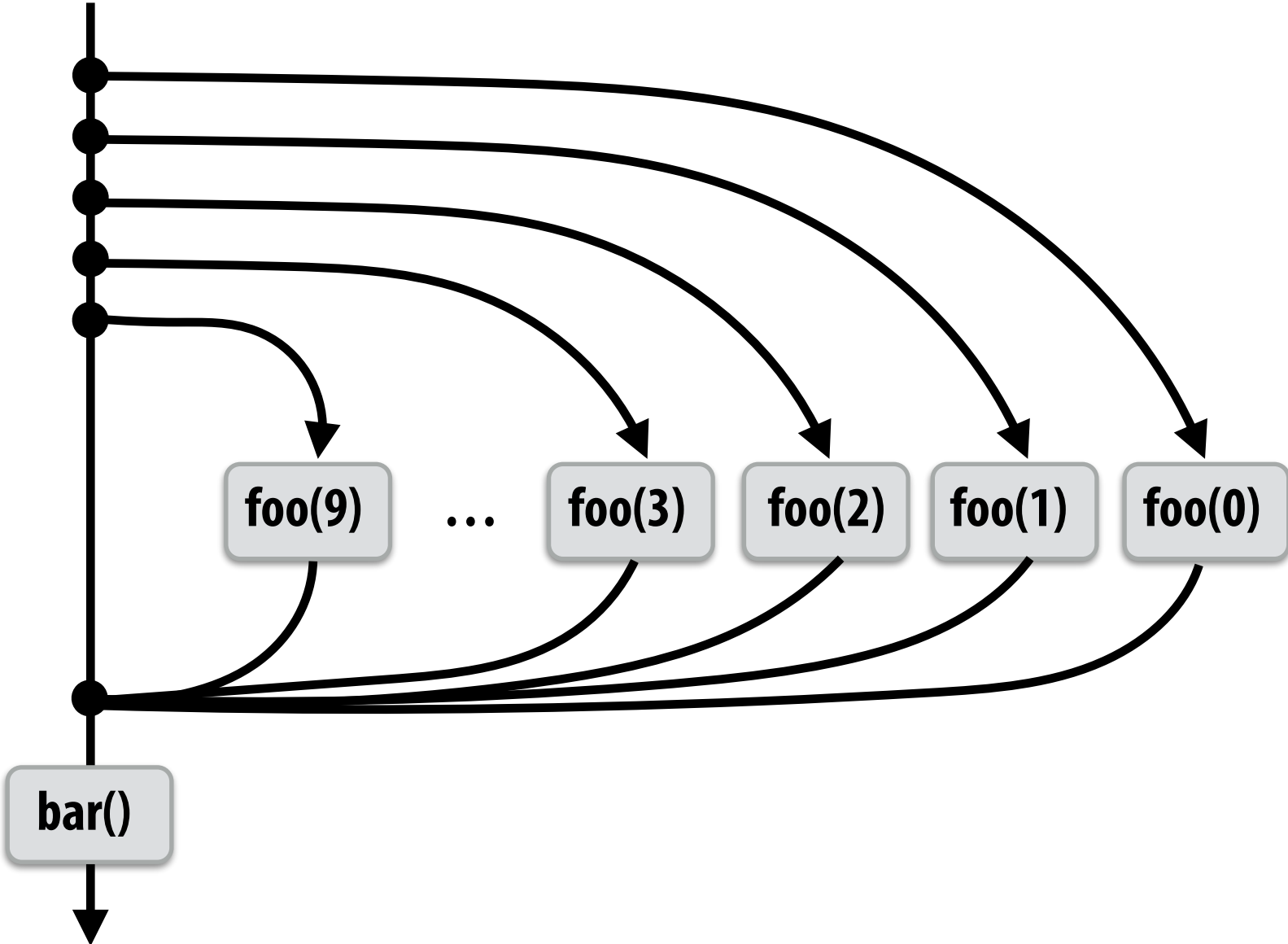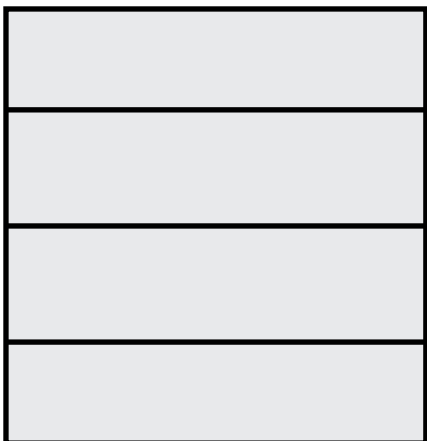
**Thread 0 work queue**

**Thread 1 work queue**

**Thread 2 work queue**

**Thread 0 now resumes continuation and executes bar()**
**Note block A descriptor is now free.**
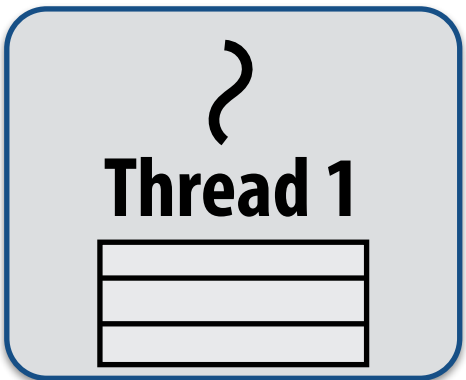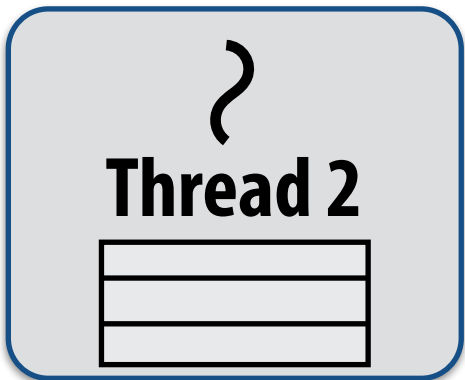
Thread 0

Thread 1

Thread 2

**Working on bar()...**          **Idle!**          **Idle!**

# 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();
```
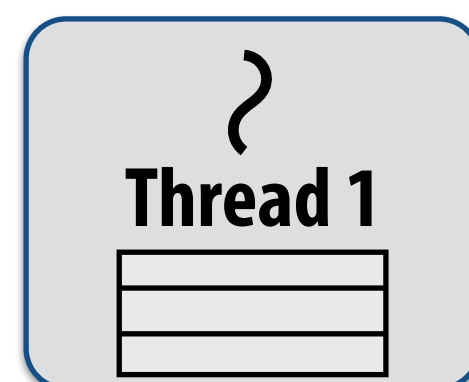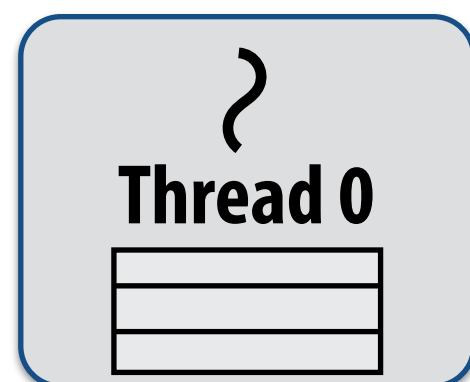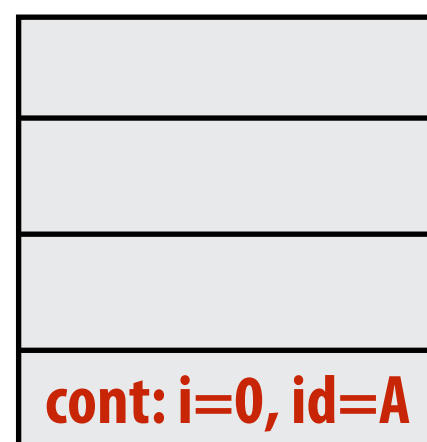


**Thread 0 work queue**    **Thread 1 work queue**

**Steal!**

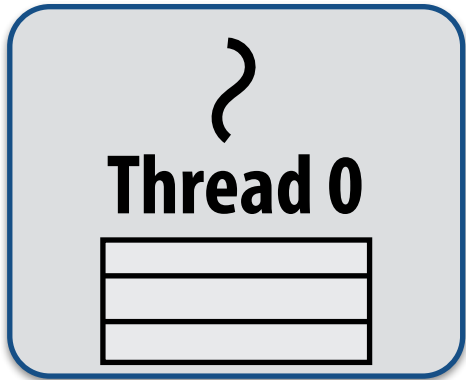| |
|---|
| |
| |
| |
| STOLEN (id=A) |

| |
|---|
| |
| |
| |
| cont: i=0, id=A |

**id=A**
**spawn: 0, done: 0**
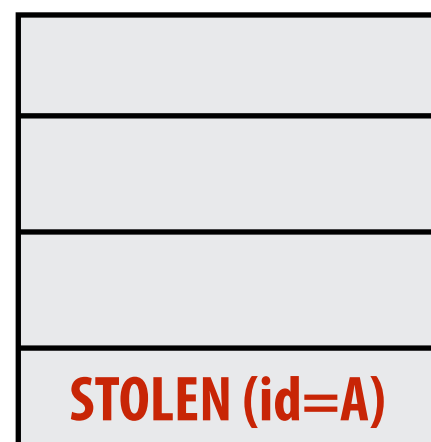
## Example 2: "greedy" policy
- **When thread that initiates the fork goes idle, it looks to steal new work**
- **Last thread to reach the join point continues execution after sync**

**Thread 0**

**Thread 1**

**Working on foo(0), 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();
```



foo(9)  …  foo(3)  foo(2)  foo(1)  foo(0)

bar()

**Thread 0 work queue**

**Thread 1 work queue**
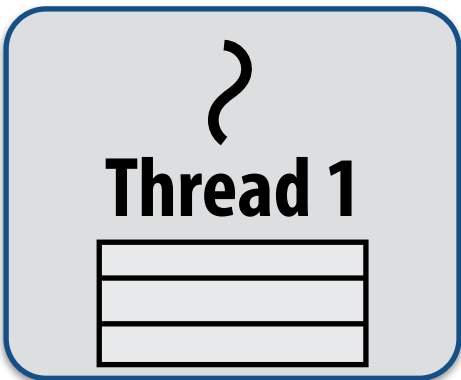
id=A
spawn: 2, done: 0

STOLEN (id=A)

cont: i=1, id=A
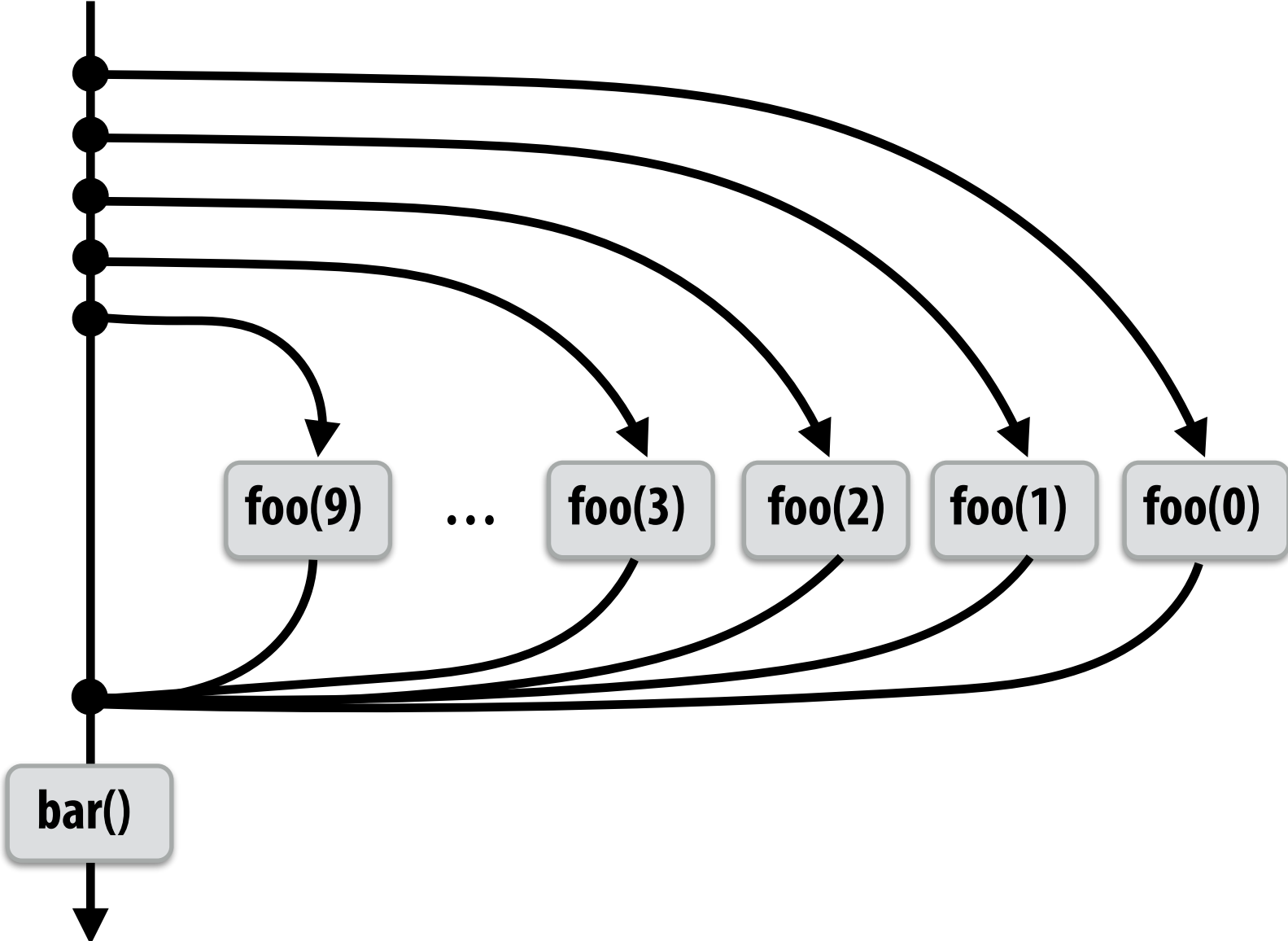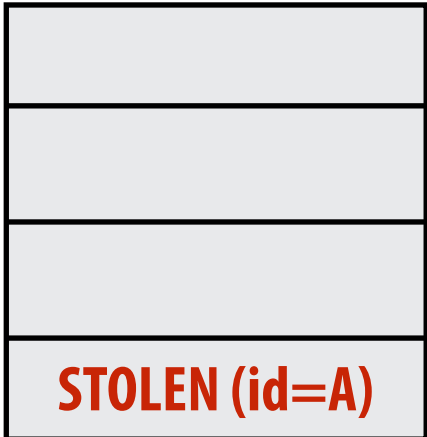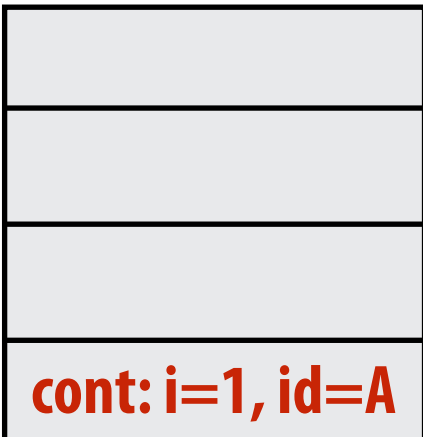
**Idle thread 1 steals from busy thread 0
(as in the previous case)**

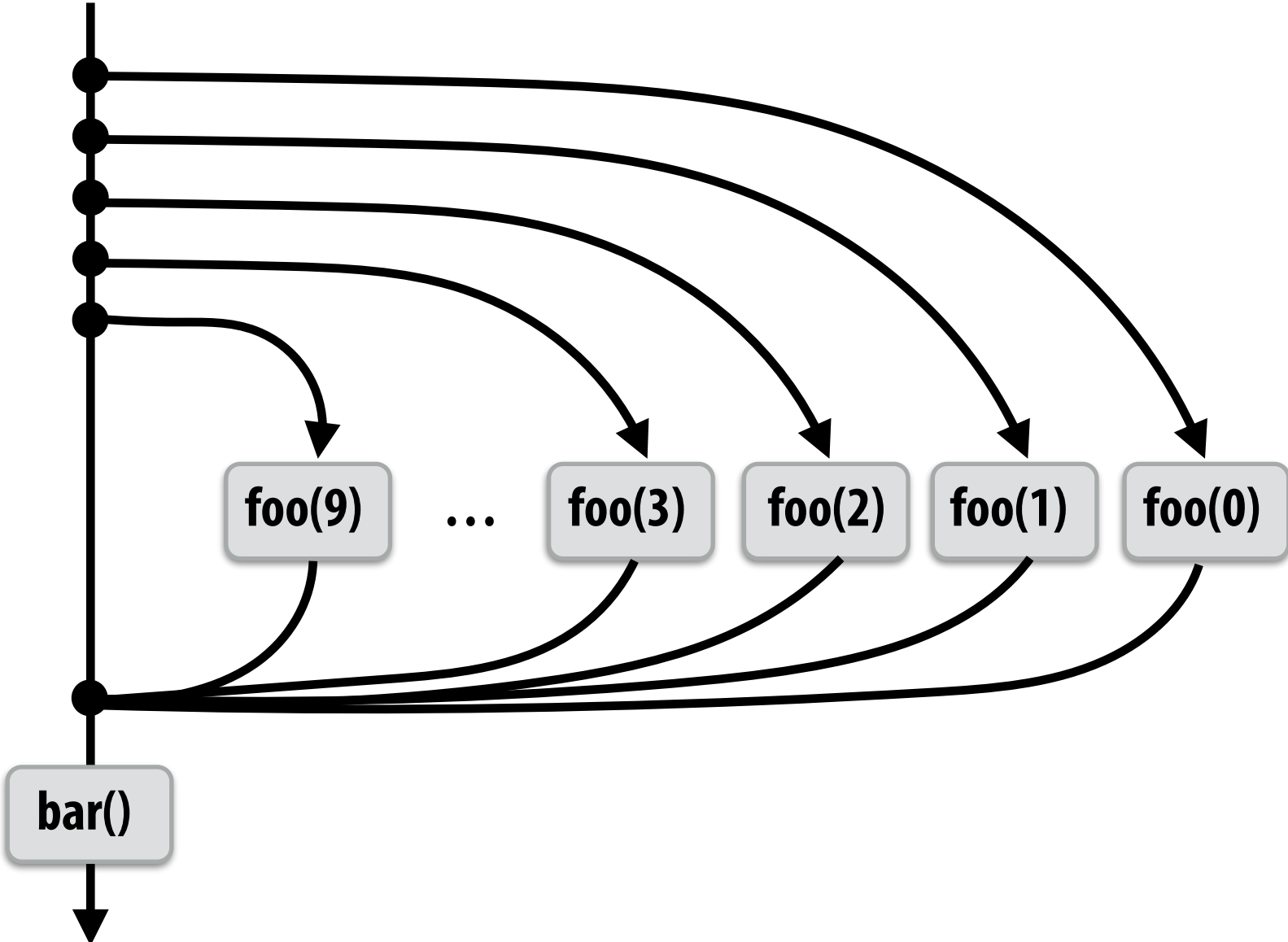Thread 0

Thread 1

Working on foo(0), id=A…

Working on foo(1), id=A…

# Implementing sync: greedy policy

**block (id: A)**
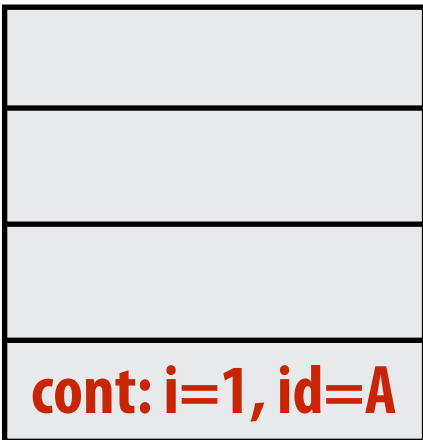
```
for (int i=0; i<10; i++) {

    cilk_spawn foo(i);

}

cilk_sync;  Sync for all calls spawned within block A

bar();
```
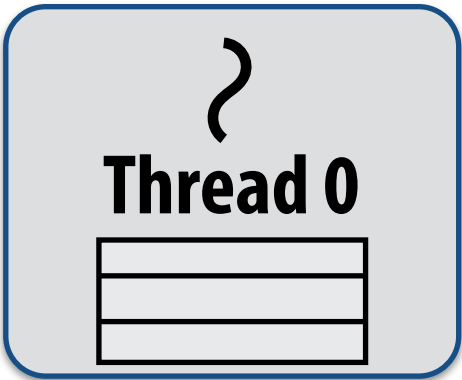


foo(9) … foo(3) foo(2) foo(1) foo(0)

bar()

**Thread 0 work queue**

| |
|---|
| |
| |
| |
| **STOLEN (id=A)** |

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

**Thread 1 work queue**

| |
|---|
| |
| |
| |
| **cont: i=1, id=A** |

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

**Thread 0**

**Thread 1**

**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();
```
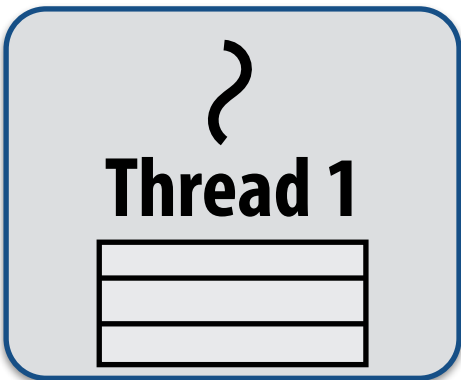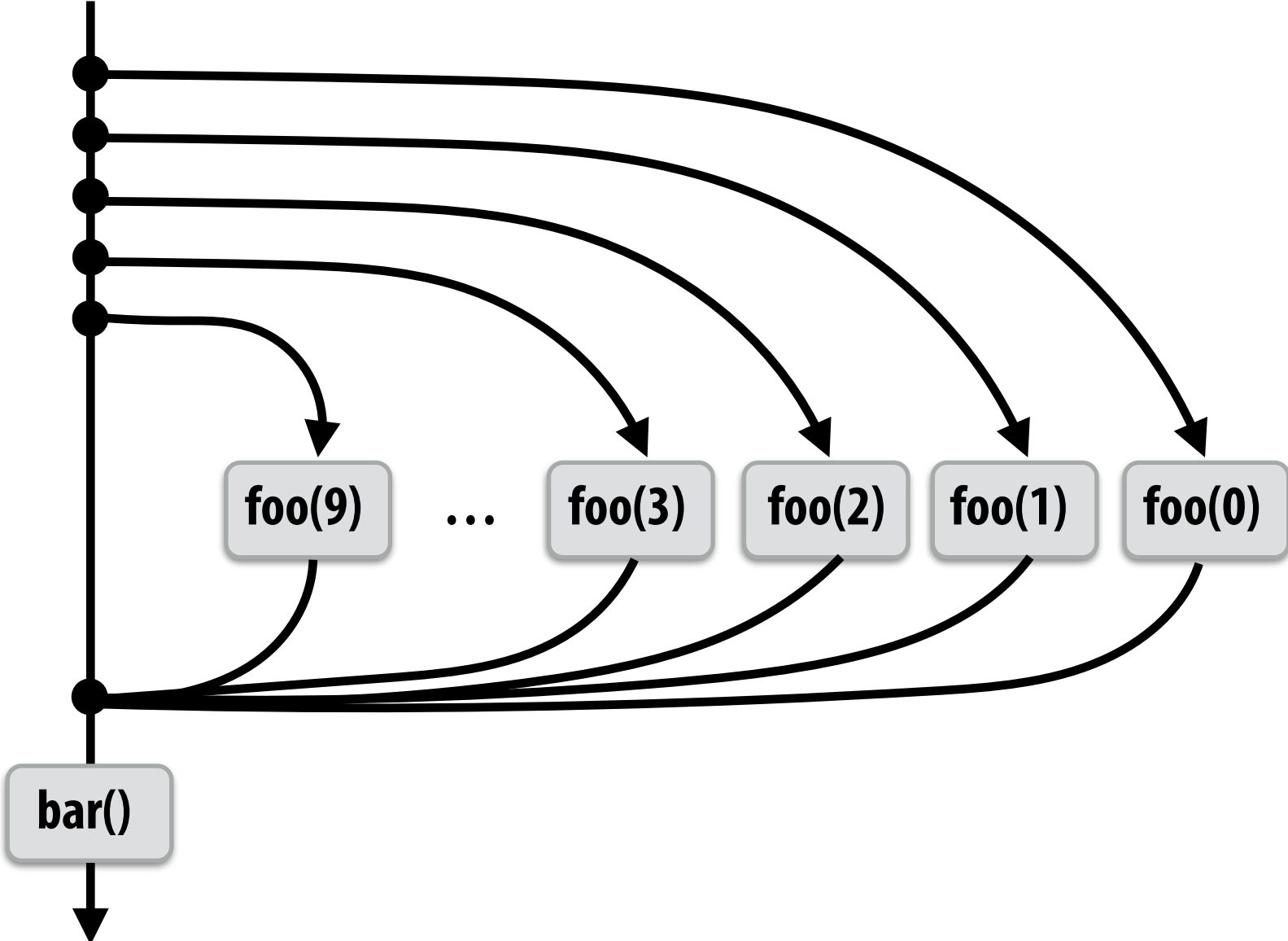


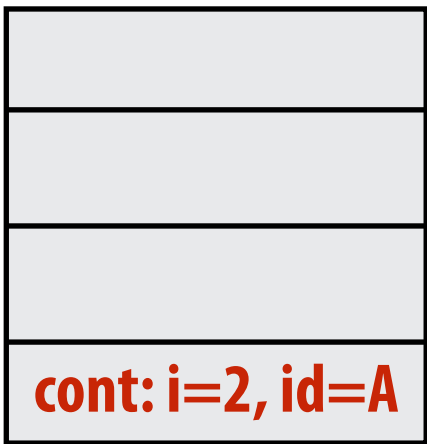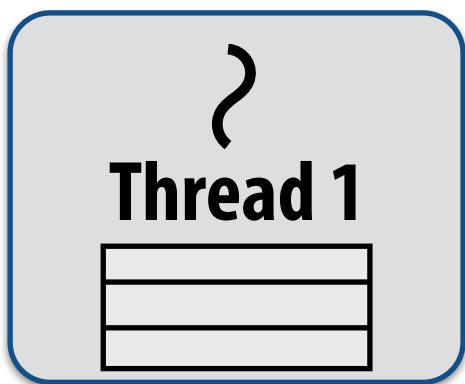**Thread 0 work queue**      **Thread 1 work queue**

**Thread 0 now working on foo(2)**

**Steal!**

id=A
spawn: 3, done: 1

cont: i=2, id=A ⟵ STOLEN (id=A)

Thread 0

Thread 1

Working on foo(2), 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();
```
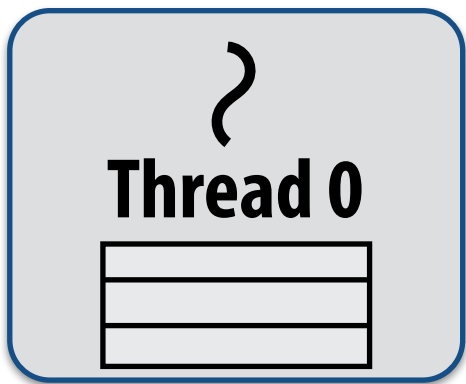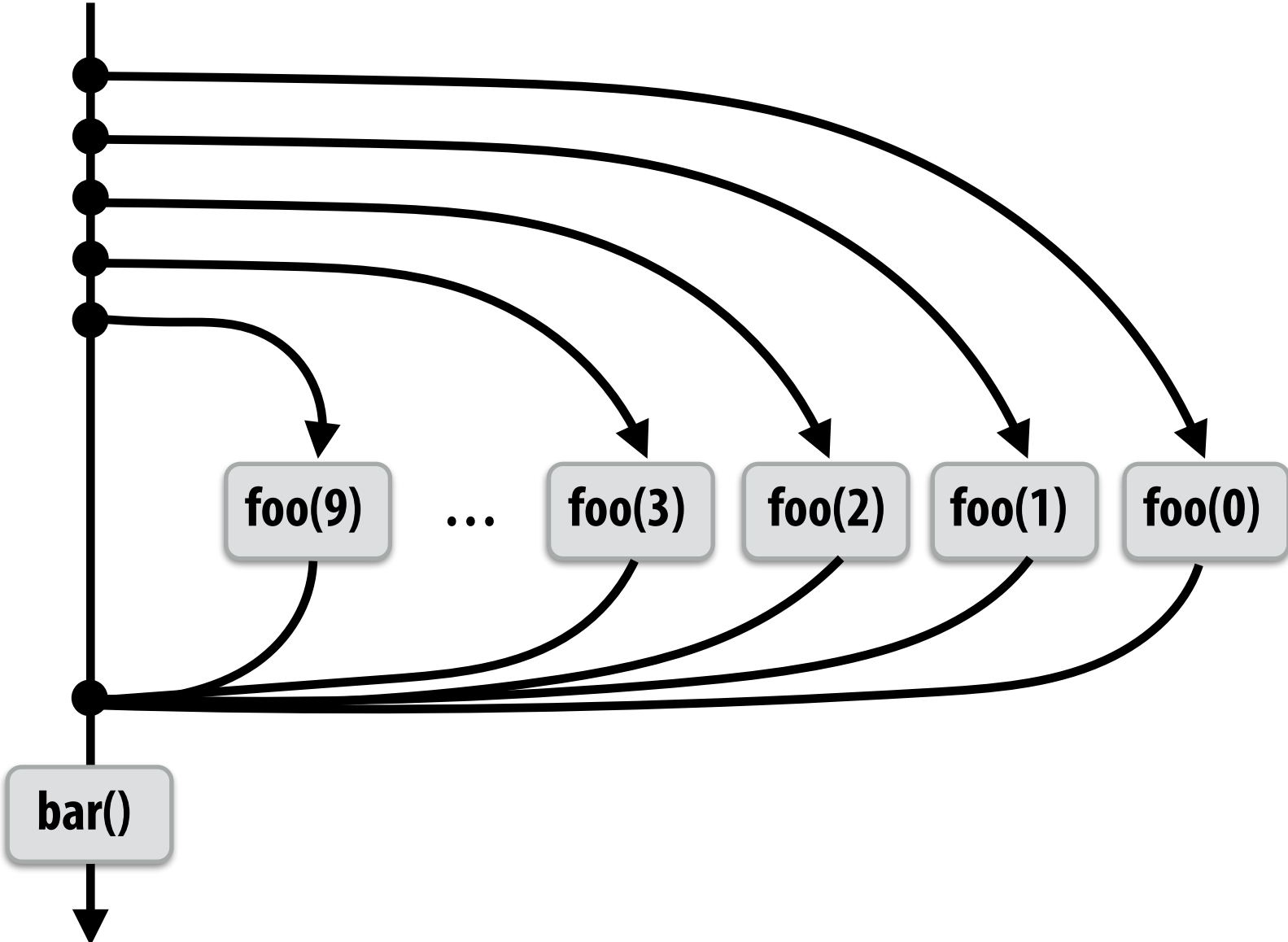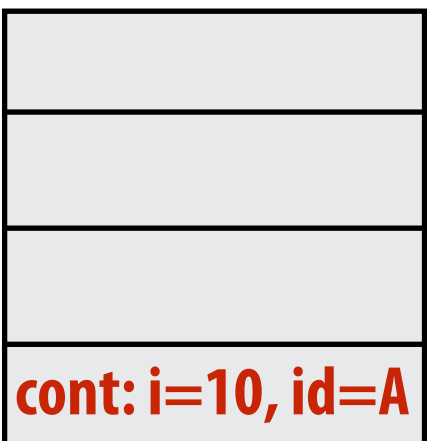


**Thread 0 work queue**          **Thread 1 work queue**

| | | |

**id=A**
**spawn: 10, done: 9**

cont: i=10, id=A

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

**Thread 0**

**Thread 1**

**Idle**          **Working on foo(9), 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();
```
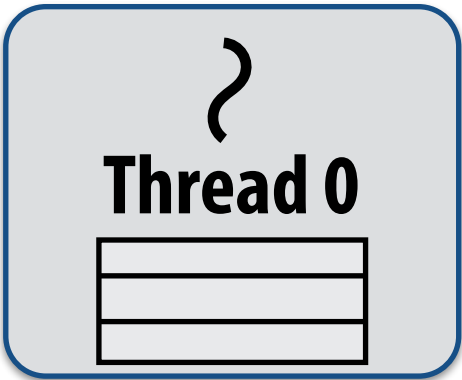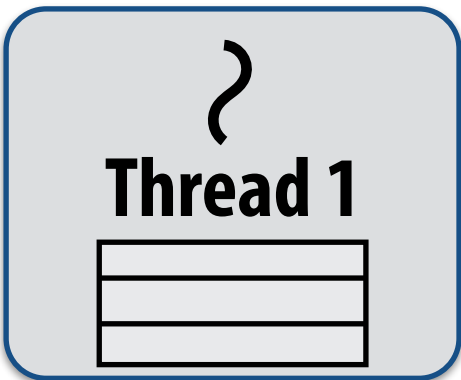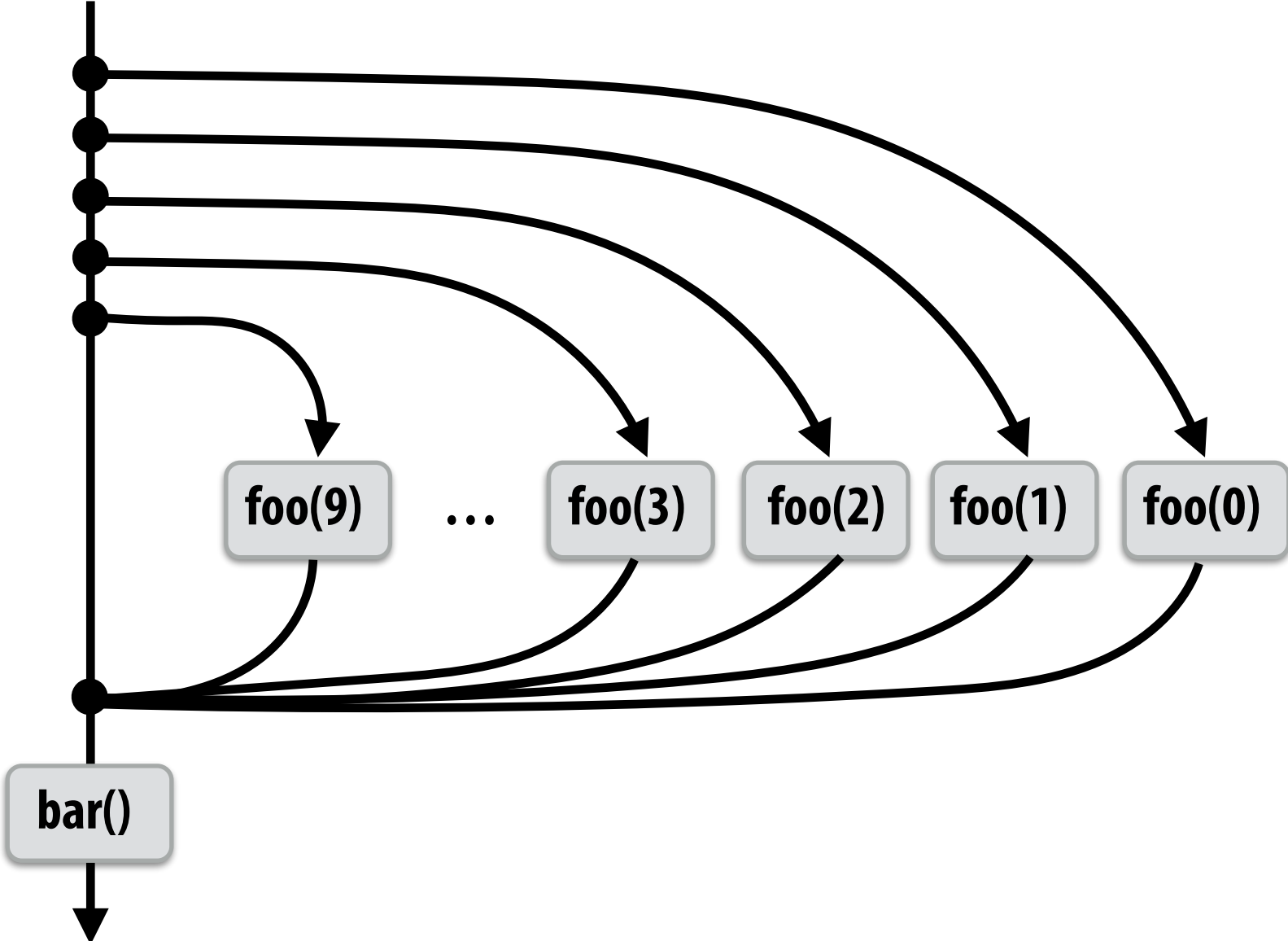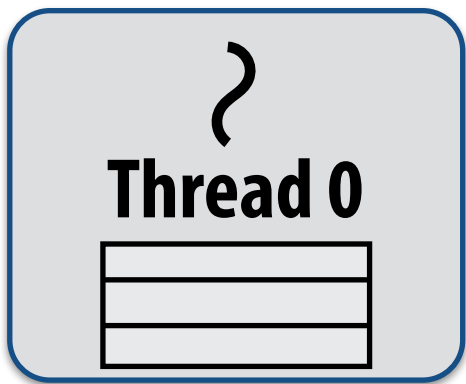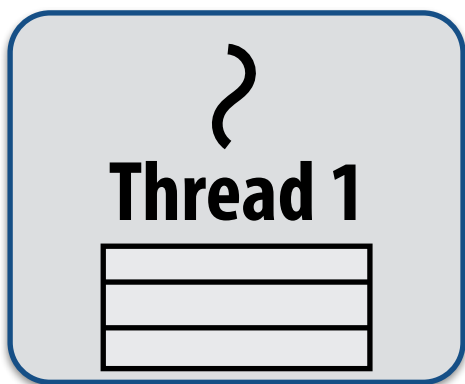
foo(9)  …  foo(3)  foo(2)  foo(1)  foo(0)

bar()

**Thread 0 work queue**

**Thread 1 work queue**

**Thread 1 continues on to run bar()**
**Note block A descriptor is now free.**

**Thread 0**

**Thread 1**

**Idle**

**Working on bar()**

# 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

# Summary

- **Fork-join parallelism: a natural way to express divide-and-conquer algorithms**

  - Discussed Cilk Plus, but OpenMP also has fork/join primitives

- **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)