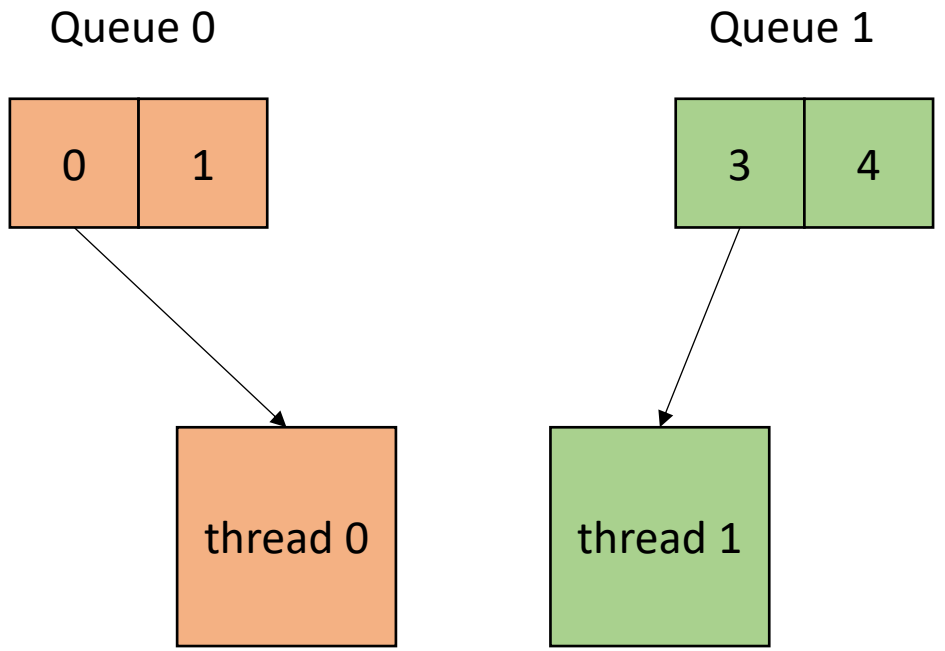
CSE113: Parallel Programming

Feb. 9, 2022

- **Topics**:
  - Finish workstealing
    - Shared worklists



Announcements

- Midterm is out!
  - You have until next Monday at midnight to do it.
  - Do not discuss with your classmates
  - Do not google specific questions or ask on online forums
  - Ask any clarifying questions as a private post on piazza
  - Late tests will not be accepted
  - You can ask me or Reese about the midterm, not Tim or Sanya
- Homework 3 is out
  - You should have everything you need by end of today
  - Due next Friday by midnight
- Grades for HW 1 are released
  - You have until next Tuesday to discuss any issues

Today’s Quiz

- Due tomorrow by midnight. Please do it!

Previous quiz

A DOALL Loop must have:

- ◯ A loop variable that starts at 0 and is incremented by 1
- ◯ loop iterations that are independent
- ◯ be unrolled and interleaved
- ◯ not access any memory locations

Previous quiz

A parallel schedule for a DOALL loop is:

- ◯ a hint to the OS to schedule the thread executing the loop
- ◯ a time sharing scheme for any shared memory across loop iterations
- ◯ a method to distribute loop iterations to different threads

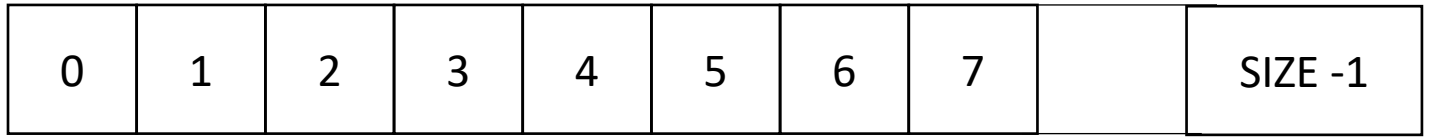
Previous quiz

Which one of the following is NOT a drawback of a global workstealing parallel schedule

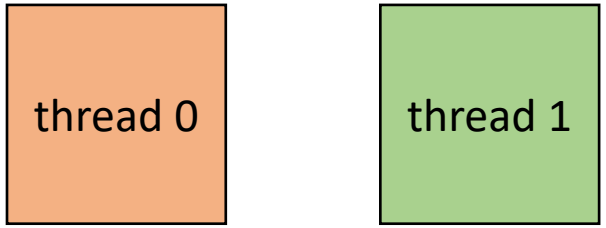
- ◯ requires a concurrent data structure
- ◯ contention on shared cache lines
- ◯ contention on a single location with RMWs

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

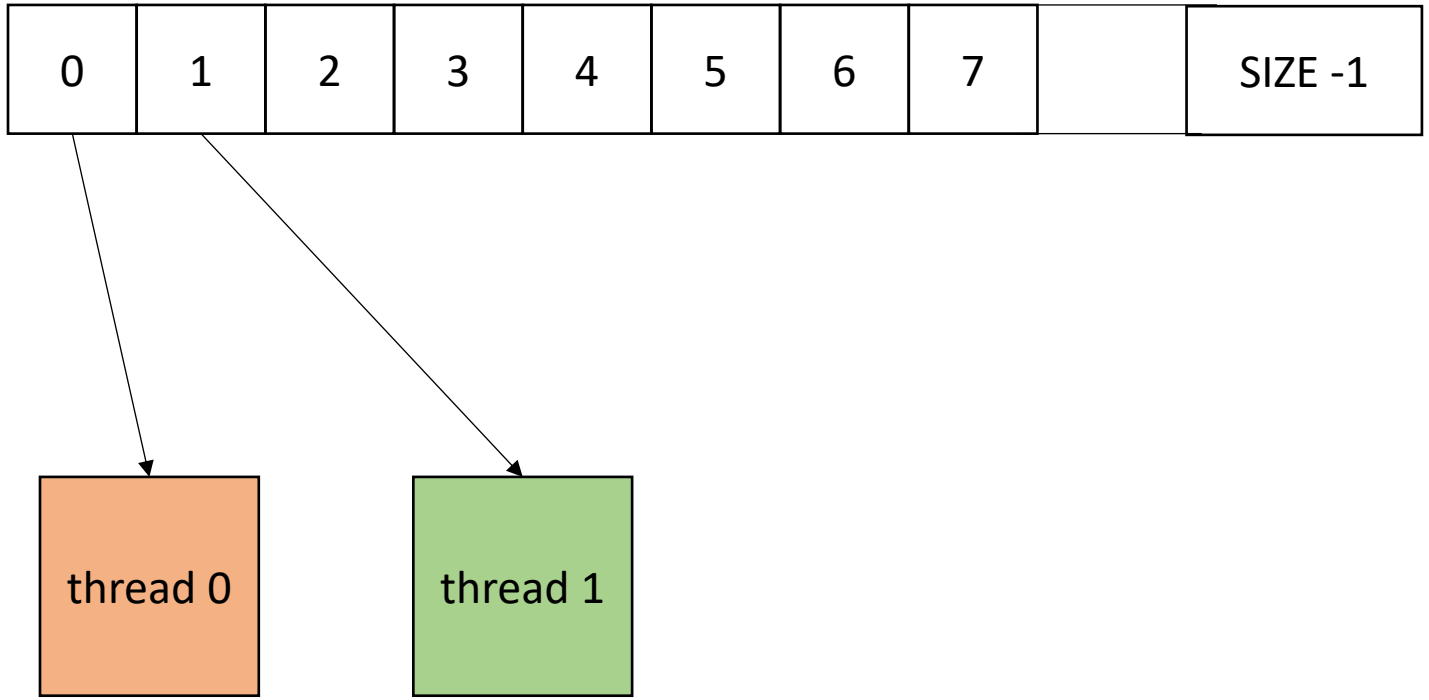


cannot color initially!



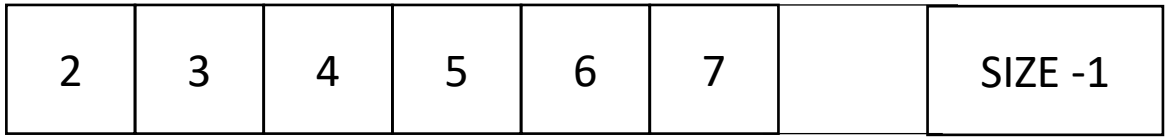
Work stealing - global implicit worklist

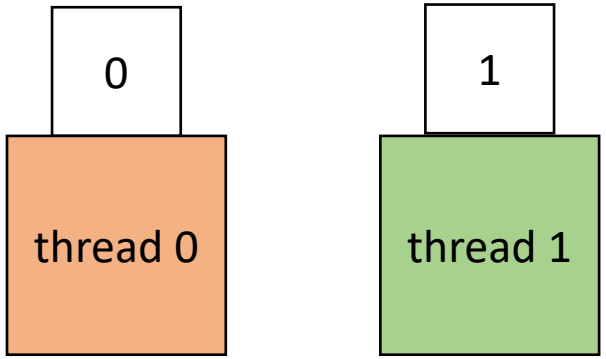
- Global worklist: threads take tasks (iterations) dynamically



Work stealing - global implicit worklist

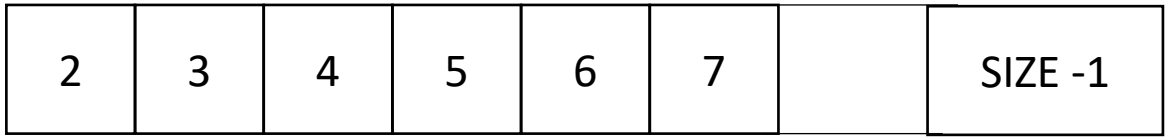
- Global worklist: threads take tasks (iterations) dynamically

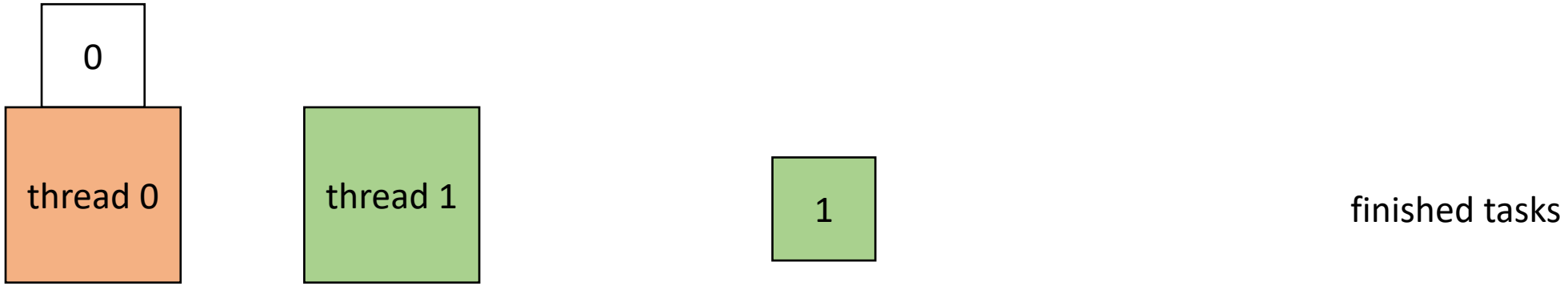




Work stealing - global implicit worklist

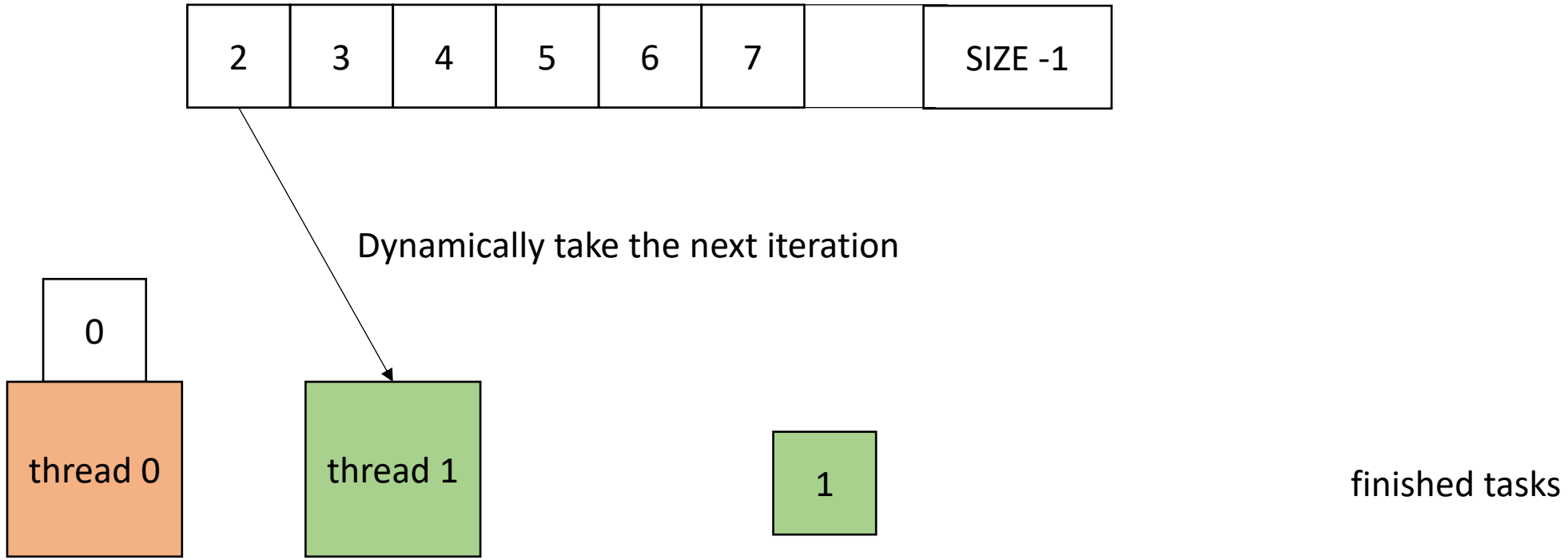
- Global worklist: threads take tasks (iterations) dynamically





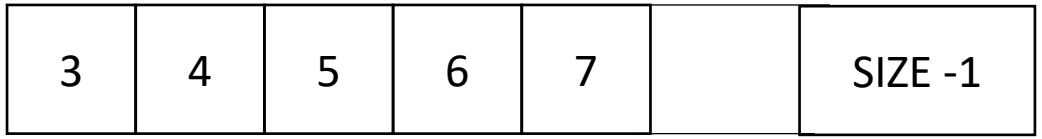
Work stealing - global implicit worklist

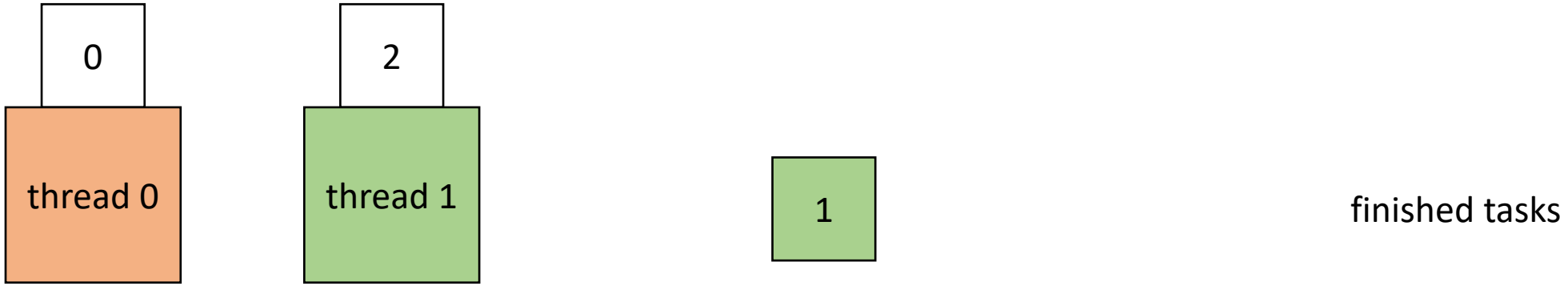
- Global worklist: threads take tasks (iterations) dynamically



Work stealing - global implicit worklist

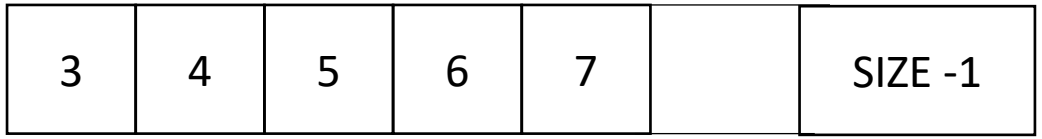
- Global worklist: threads take tasks (iterations) dynamically

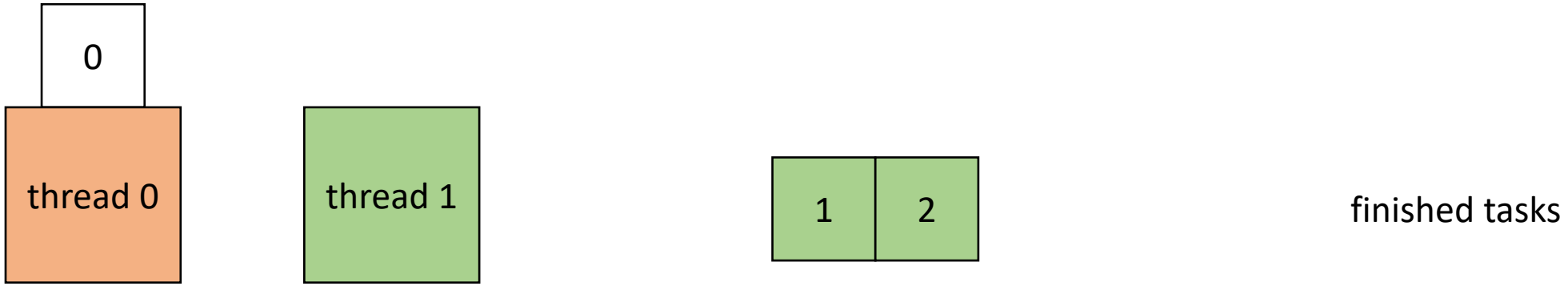




Work stealing - global implicit worklist

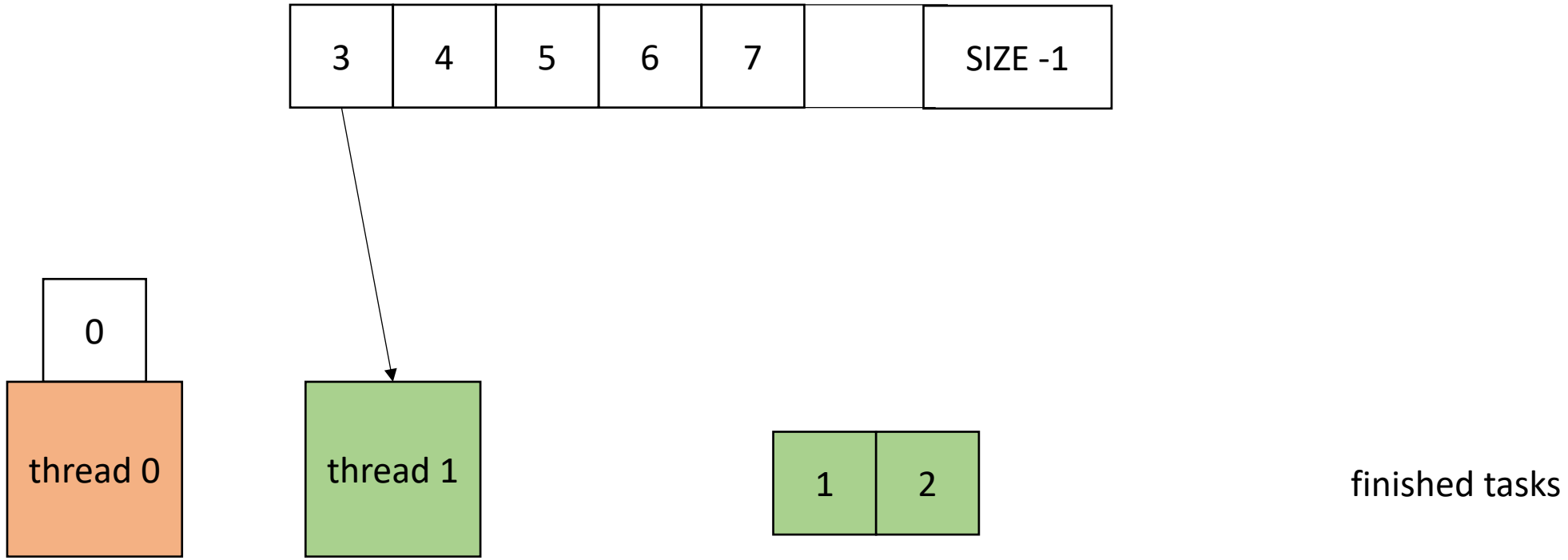
- Global worklist: threads take tasks (iterations) dynamically





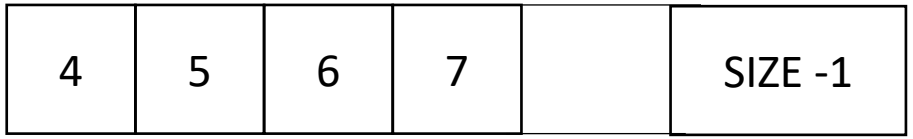
Work stealing - global implicit worklist

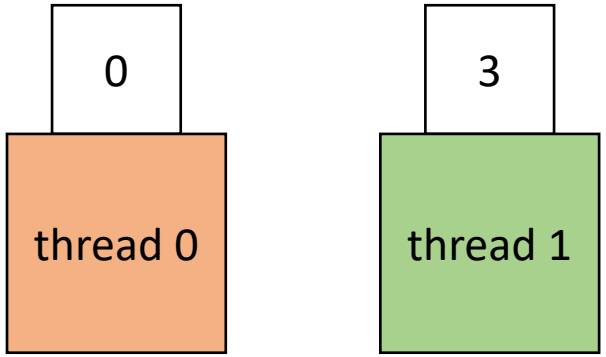
- Global worklist: threads take tasks (iterations) dynamically

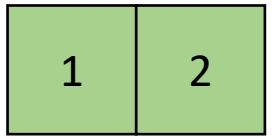


Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



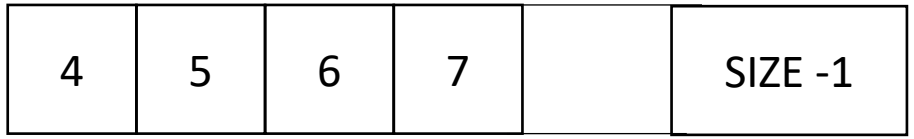


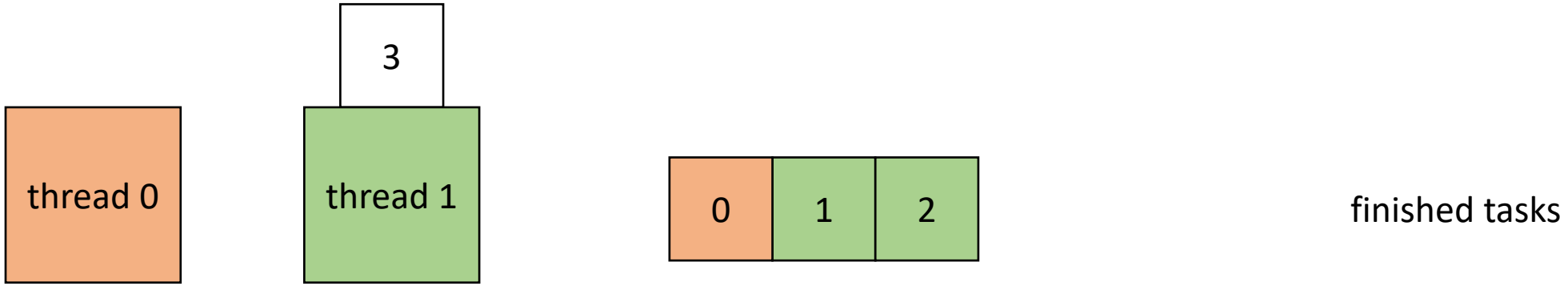


finished tasks

Work stealing - global implicit worklist

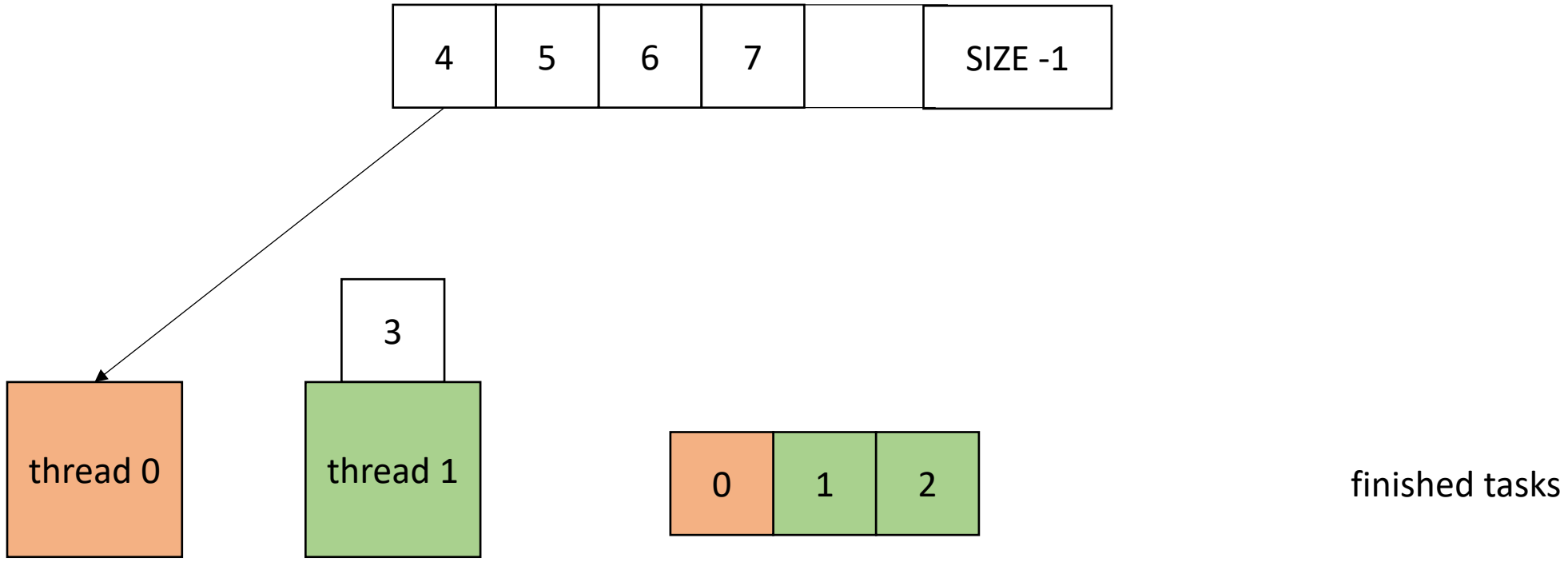
- Global worklist: threads take tasks (iterations) dynamically





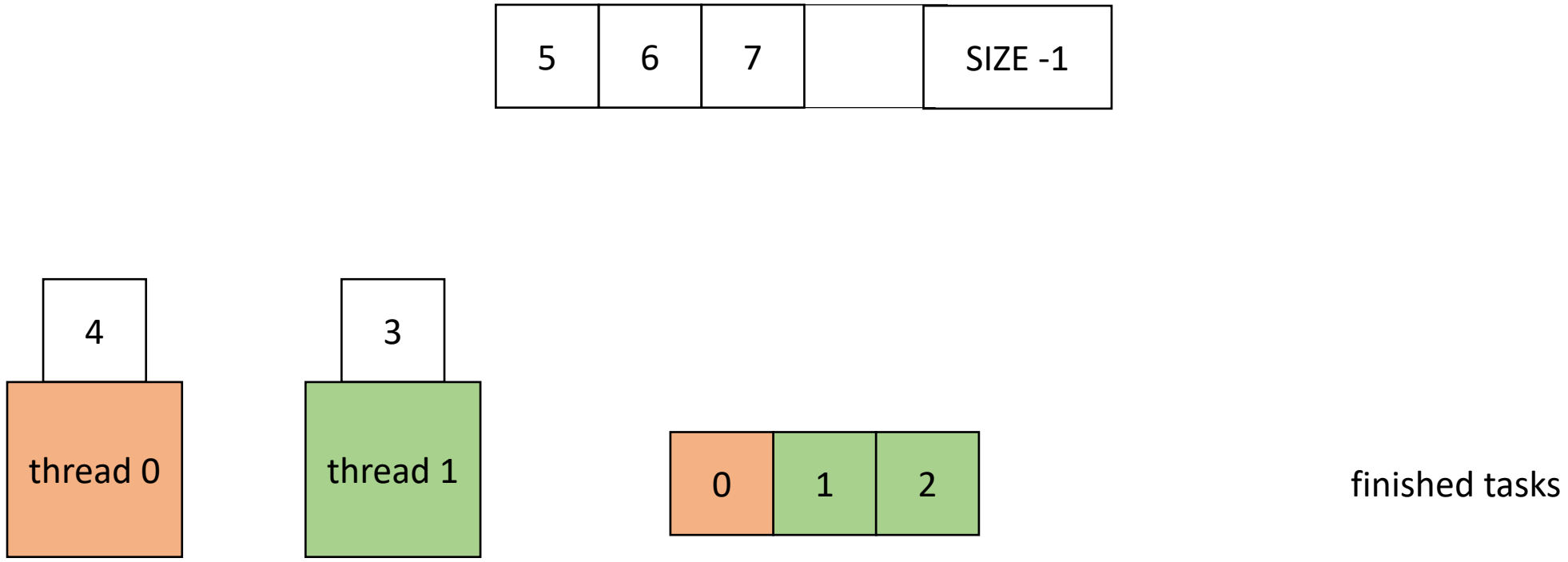
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

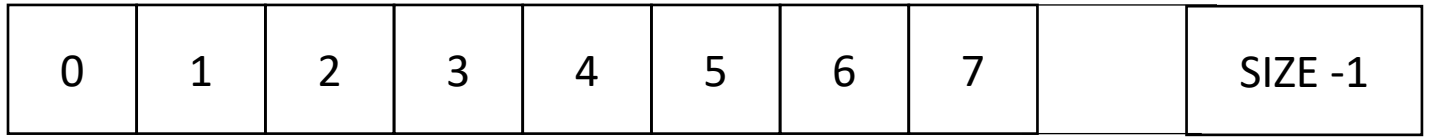


Work stealing - global implicit worklist

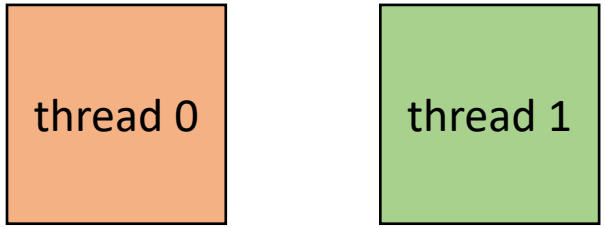
- But what if each task took roughly the same amount of time?

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

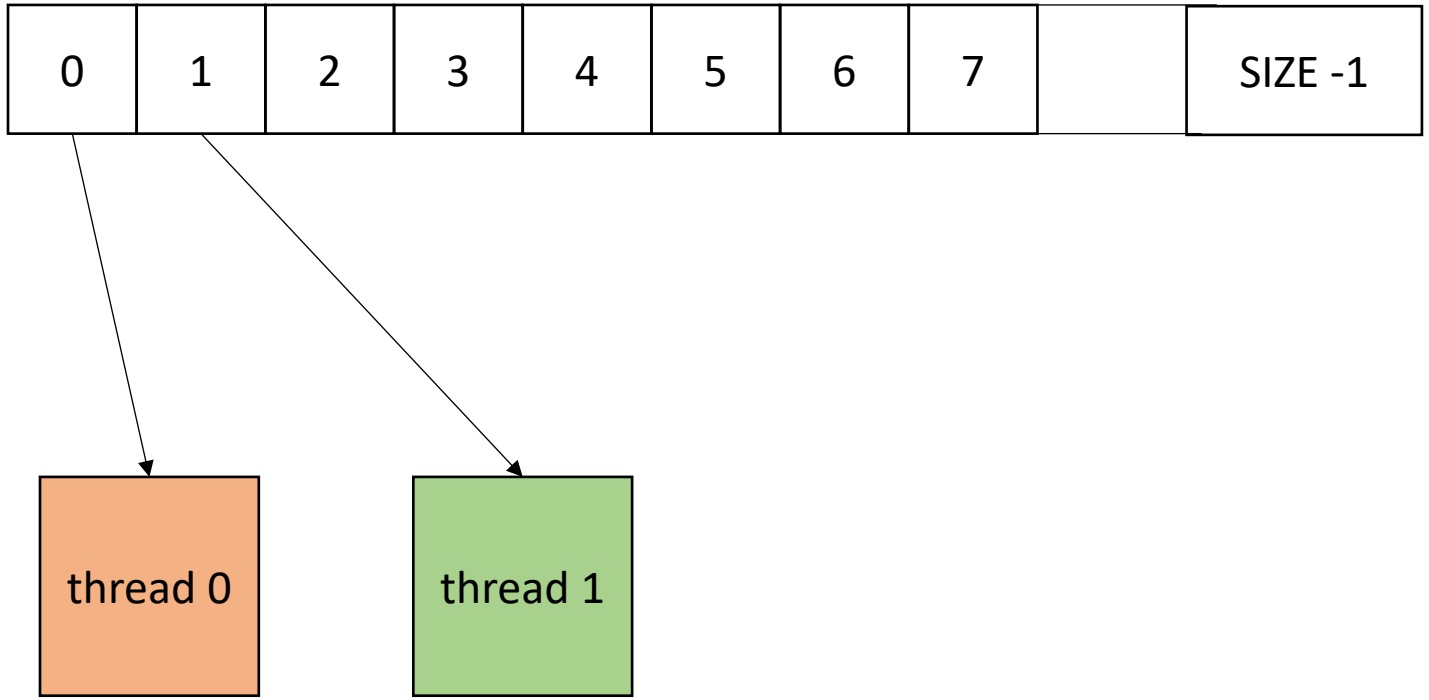


cannot color initially!



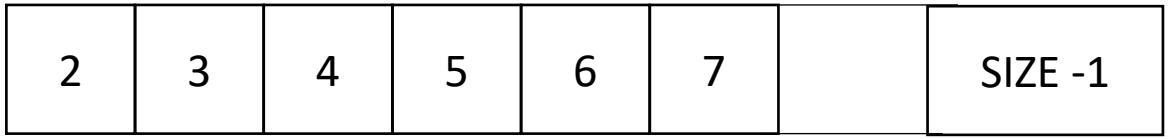
Work stealing - global implicit worklist

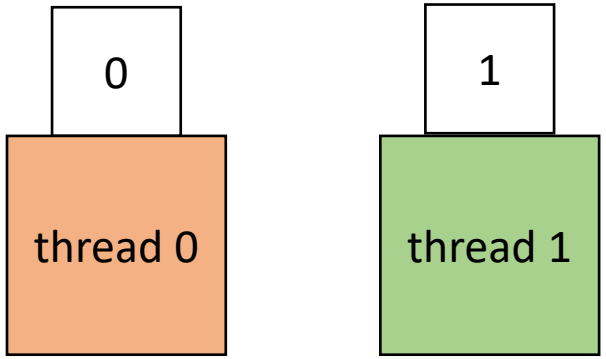
- Global worklist: threads take tasks (iterations) dynamically



Work stealing - global implicit worklist

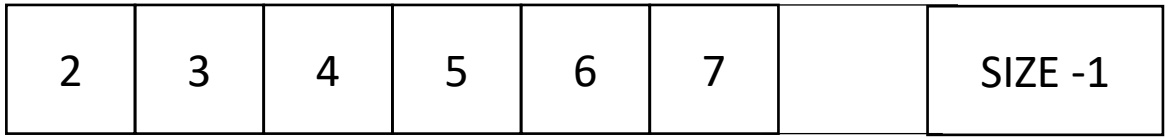
- Global worklist: threads take tasks (iterations) dynamically

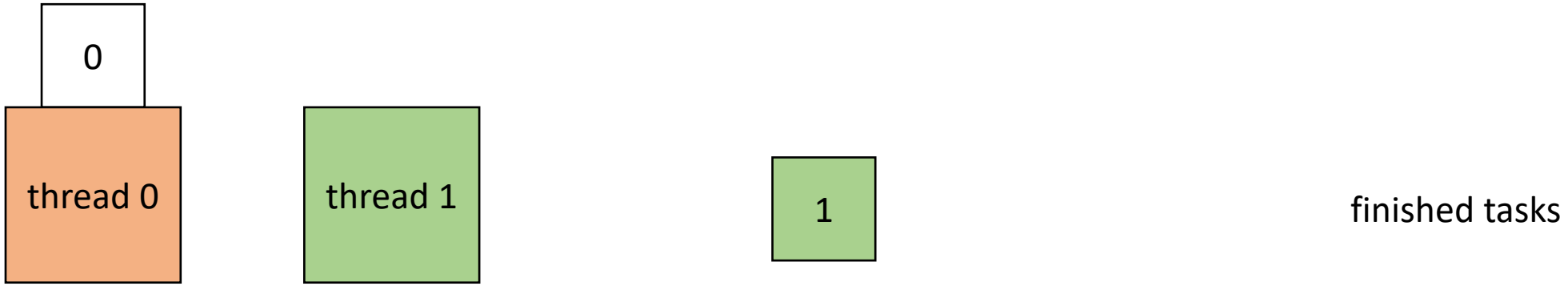




Work stealing - global implicit worklist

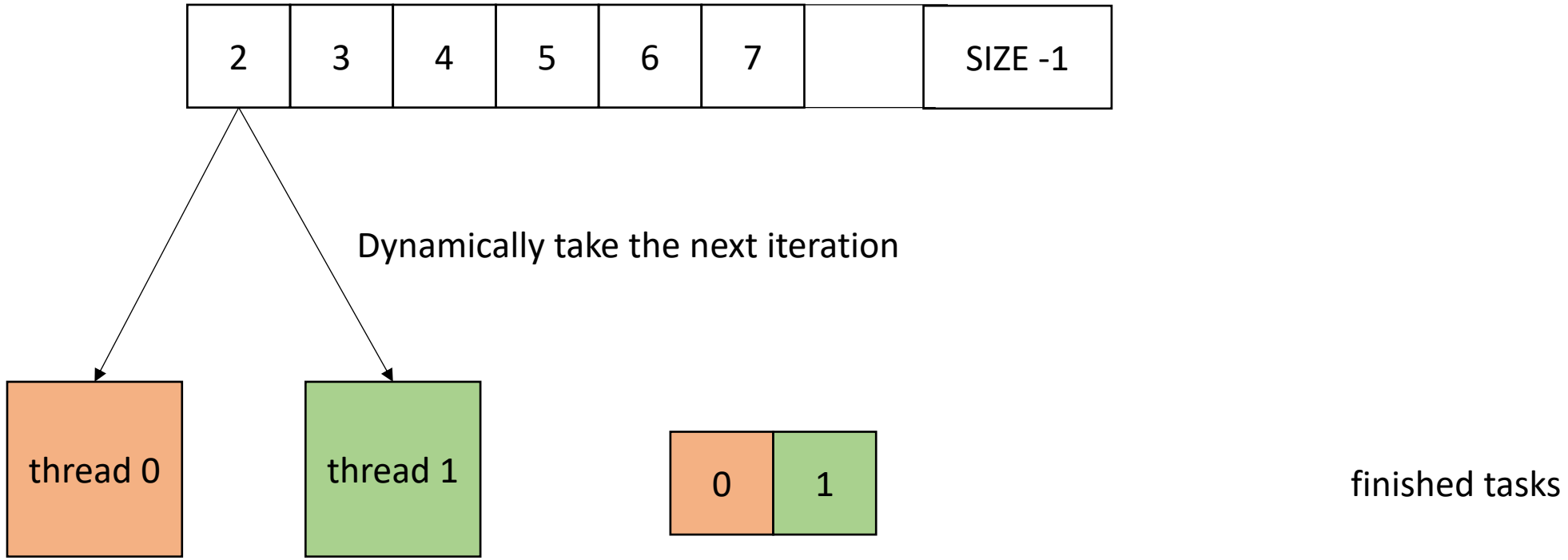
- Global worklist: threads take tasks (iterations) dynamically





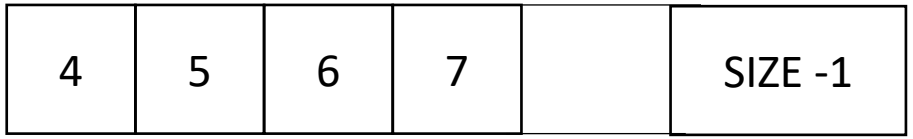
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

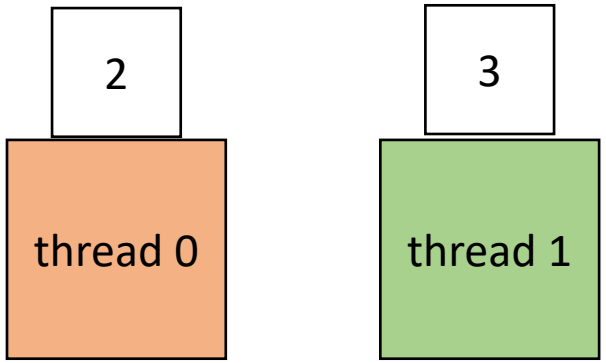


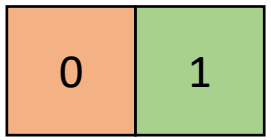
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



Dynamically take the next iteration

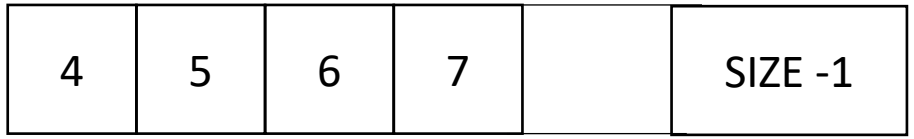




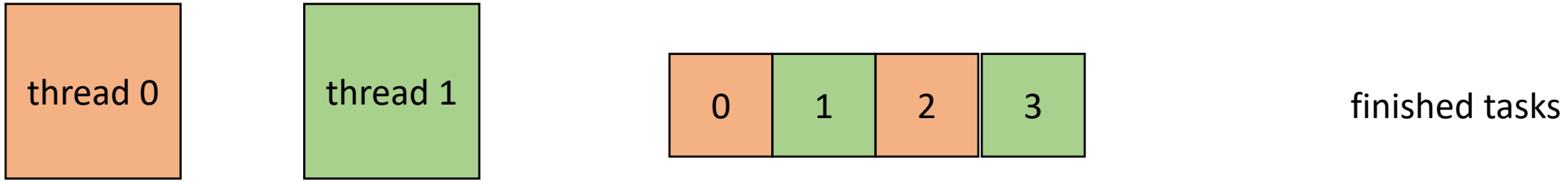
finished tasks

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



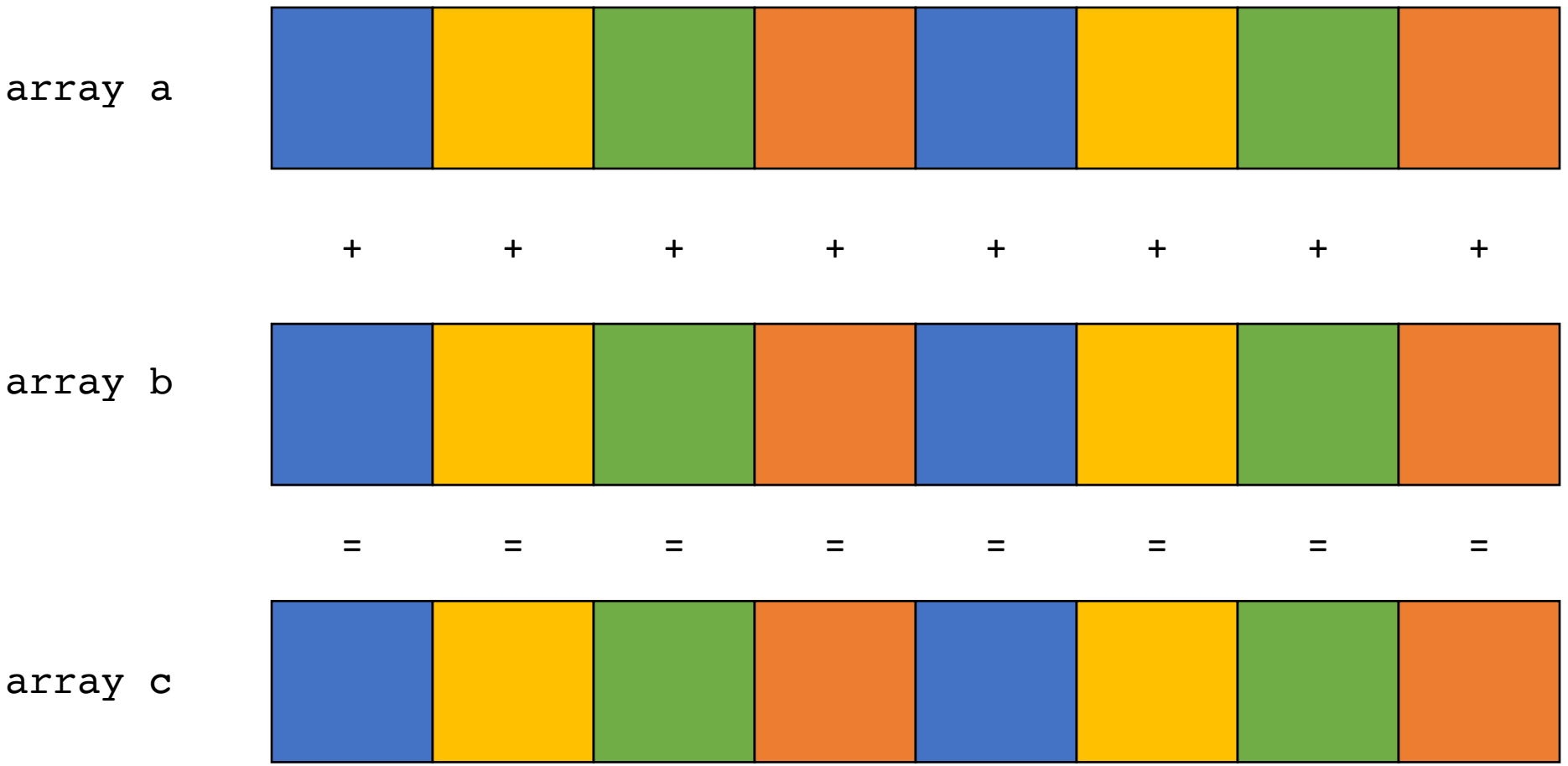
Dynamically take the next iteration



It can end up looking a lot like this:

Computation can easily be divided into threads

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Previous quiz

Which of the following is NOT an overhead of the local worklist workstealing parallel schedule (that we studied in class)

- ◯ initialization of the queues
- ◯ checking a global variable to ensure all work is completed
- ◯ managing concurrent enqueues to the worklists

Previous quiz

Given what we've learned: what role do you believe the compiler should play in parallelizing DOALL loops?

For example, should it: (1) identify them? (2) parallelize them? (3) pick a parallel schedule?

There is no right or wrong answer here, but it is interesting to think about!

We will revisit this later on in lecture!

Review

DOALL Loops

adds two arrays

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

what about a random order?

```
for (pick i randomly) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

```
for (pick i randomly) {
  a[i] += a[i+1]
}
```

DOALL Loops

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i]*2;
}
```

DOALL Loops

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 1; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i]*2;
}
```

Parallel Schedules

- Consider the following program:

There are 3 arrays: `a, b, c.`

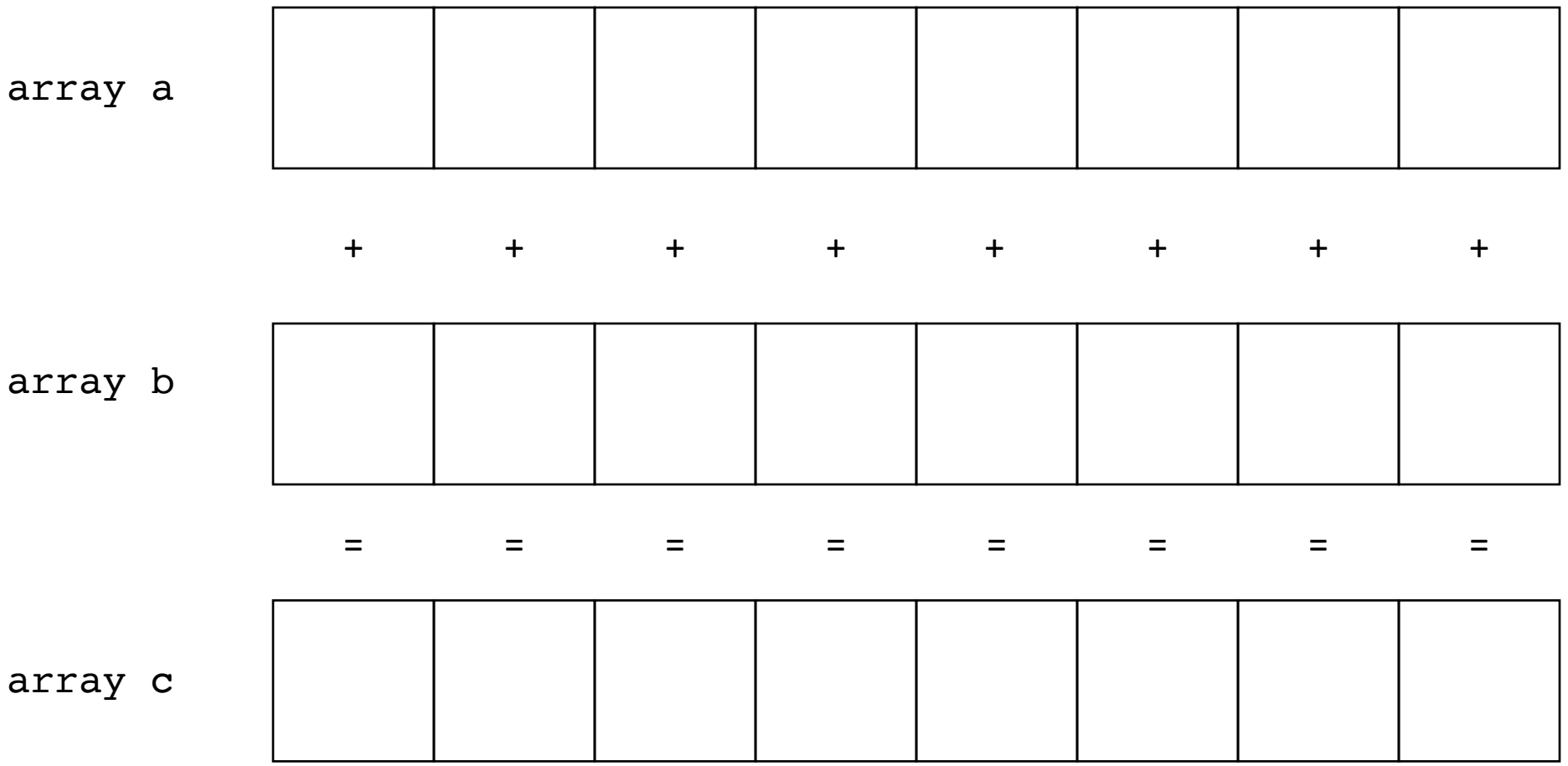
We want to compute

```
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
```

Parallel Schedules

Computation can easily be divided into threads

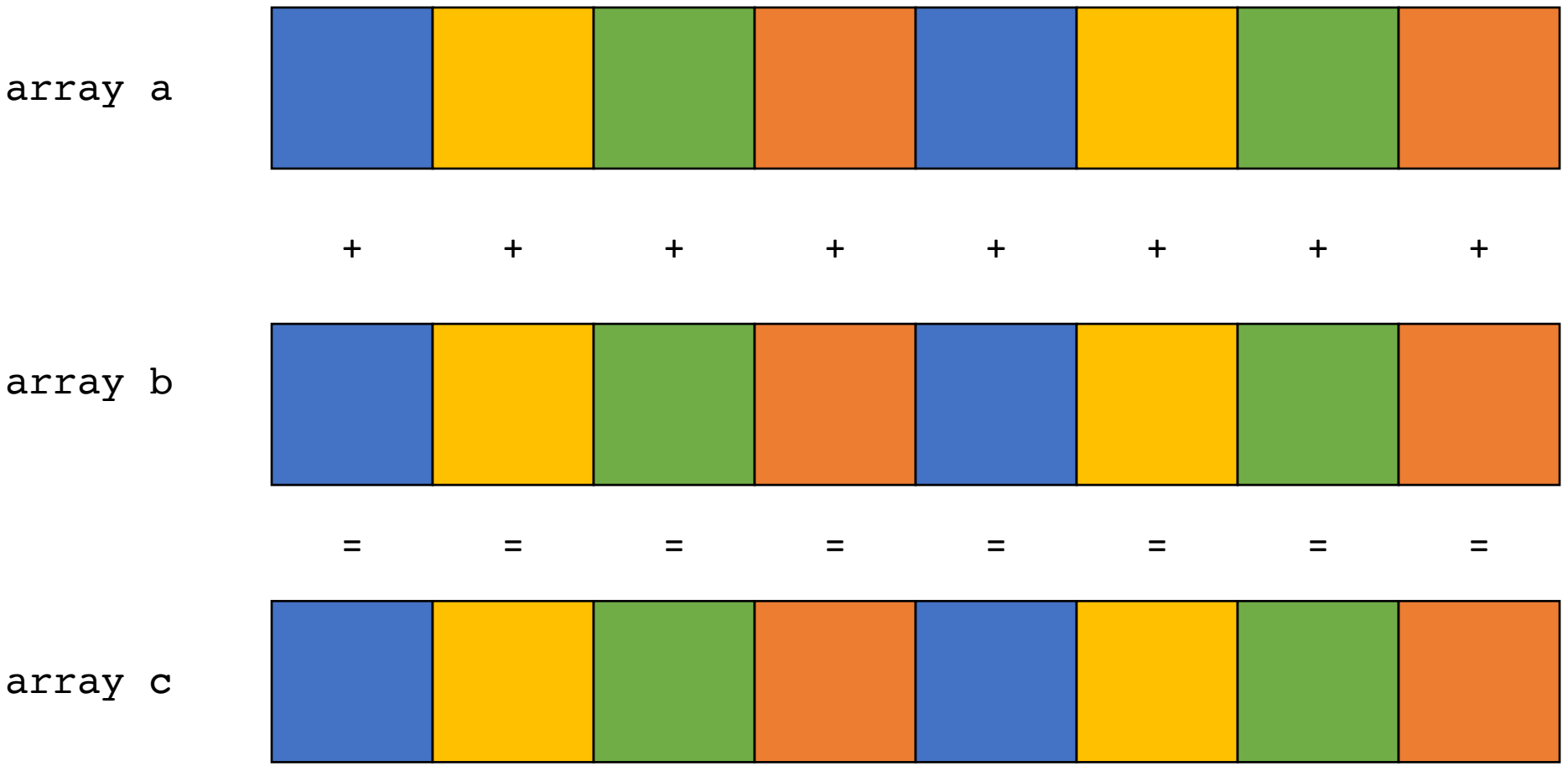
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

Computation can easily be divided into threads

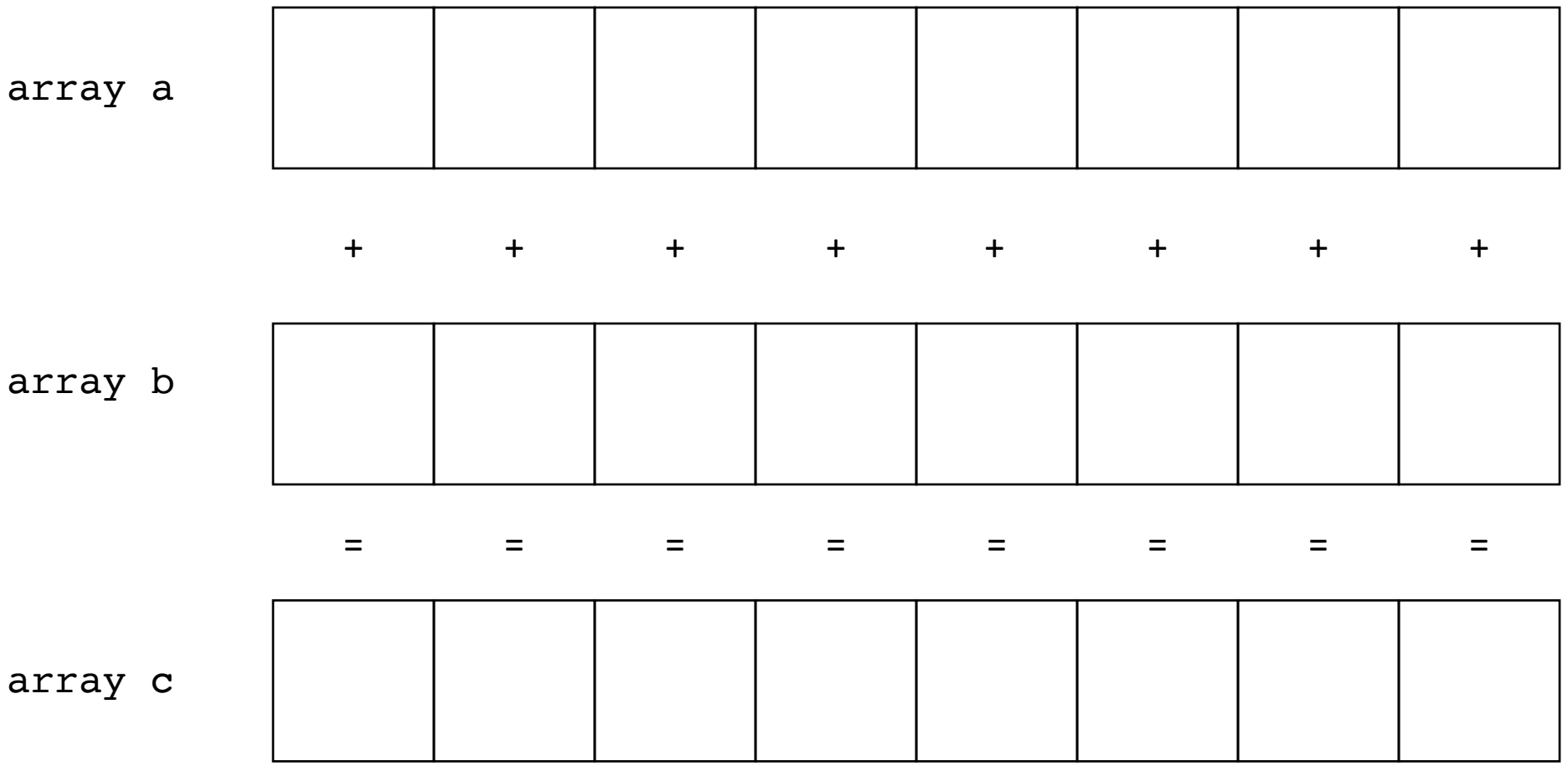
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

Computation can easily be divided into threads

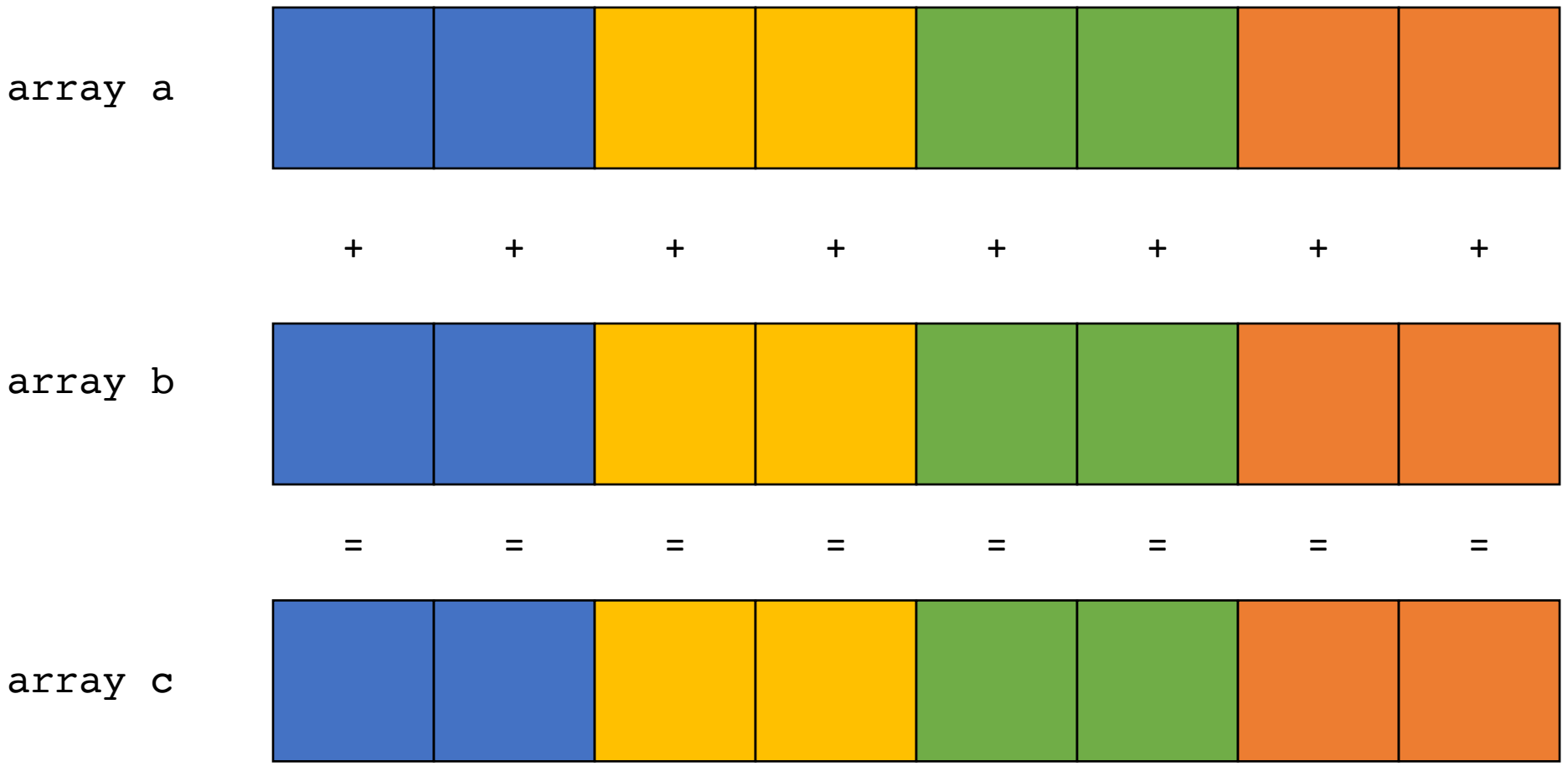
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

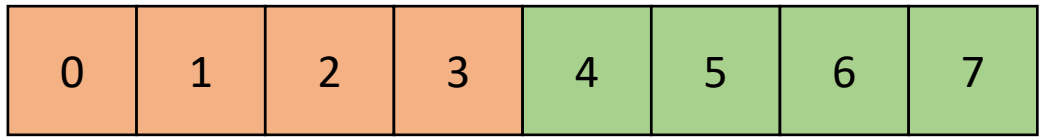
Computation can easily be divided into threads

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange

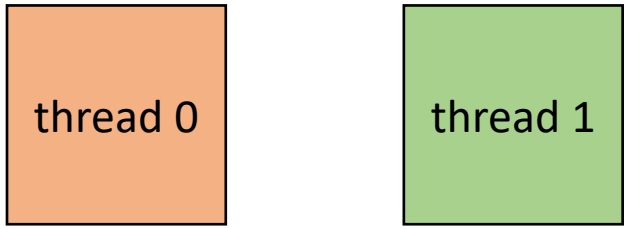


Static schedule

- Example, 2 threads/cores, array of size 8



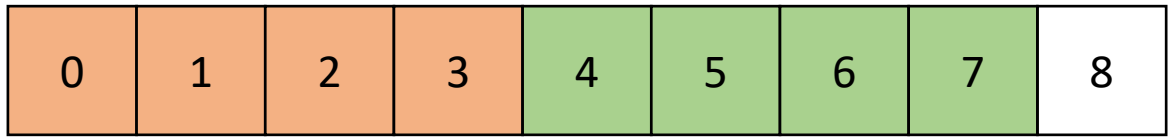

```
chunk_size = 4

0: start = 0    1: start = 4
0: end = 4      1: end = 8
```




```
void parallel_loop(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    // work based on x
  }
}
```

Static schedule

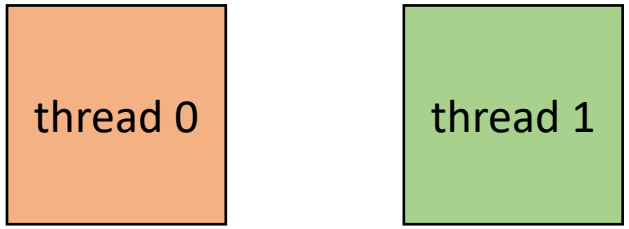
- Example, 2 threads/cores, array of size 9




```
chunk_size = 4
```

```
0: start = 0
0: end = 4
```

```
1: start = 4
1: end = 8
```



ceiling division, this will distribute uneven work in the last thread to all other threads

```
void parallel_loop(..., int tid, int num_threads)
{
  int chunk_size =
  (SIZE+(NUM_THREADS-1))/NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    // work based on x
  }
}
```

Static schedule

- Example, 2 threads/cores, array of size 9

out of bounds

```
chunk_size = 5
```

```
0: start = 0
0: end = 5
```

```
1: start = 5
1: end = 10
```

thread 0

thread 1

```
void parallel_loop(..., int tid, int num_threads)
{
  int chunk_size =
  (SIZE+(NUM_THREADS-1))/NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    // work based on x
  }
}
```

Static schedule

- Example, 2 threads/cores, array of size 9

out of bounds

```
chunk_size = 5
```

```
0: start = 0
0: end = 5
```

```
1: start = 5
1: end = 10
```

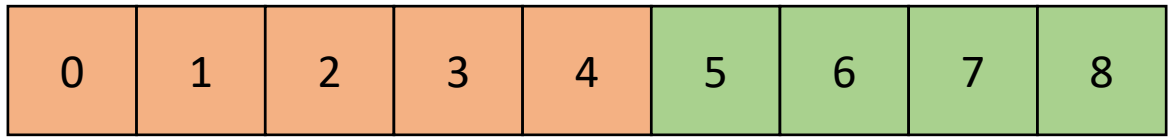
thread 0

thread 1

```
void parallel_loop(..., int tid, int num_threads)
{
  int chunk_size =
  (SIZE+(NUM_THREADS-1))/NUM_THREADS;
  int start = chunk_size * tid;
  int end =
   min(start+chunk_size, SIZE)
for (int x = start; x < end; x++) {
    // work based on x
  }
}
```

Static schedule

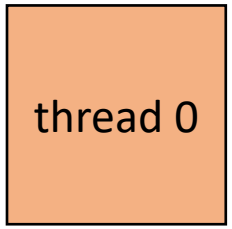
- Example, 2 threads/cores, array of size 9




```
chunk_size = 5
```

```
0: start = 0
0: end = 5
```

```
1: start = 5
1: end = 9
```



most threads do equal amounts of work, last thread may do less.

Which one is better/worse?
Max slowdown for last thread does all the extra work?

Max slowdown for ceiling?

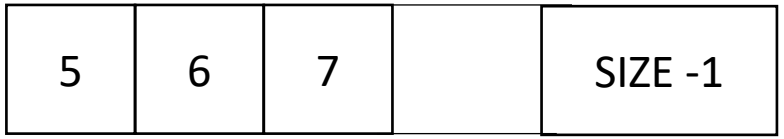
```
void parallel_loop(..., int tid, int num_threads)
{
  int chunk_size =
  (SIZE+(NUM_THREADS-1))/NUM_THREADS;
  int start = chunk_size * tid;
  int end =
   min(start+chunk_size, SIZE)
for (int x = start; x < end; x++) {
    // work based on x
  }
}
```

Global worklist schedule

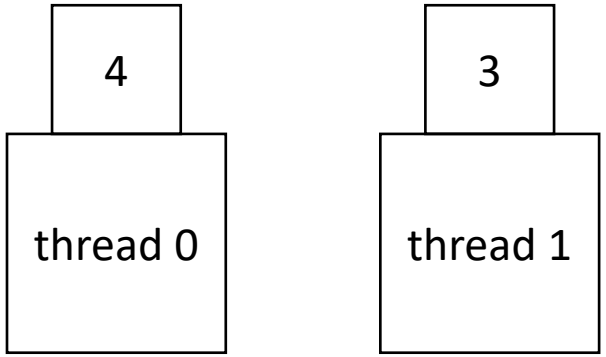
- We discussed in quiz review

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically




```
x: 5
0 - local_x - 4
1 - local_x - 3
```




```
atomic_int x(0);
void parallel_loop(...) {

  for (int local_x = atomic_fetch_add(&x,1);
            local_x < SIZE;
            local_x = atomic_fetch_add(&x,1)) {

     // dynamic work based on x
  }
}
```

Schedule

- DOALL Loops

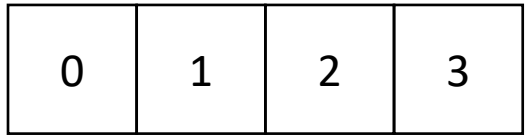
- **Parallel Schedules:**
  - Static
  - Global Worklists
  - **Local Worklists**

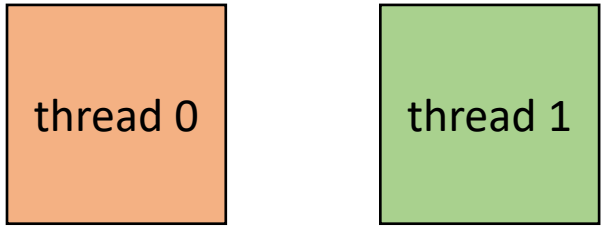
Work stealing - local worklists

- More difficult to implement: typically requires concurrent data-structures
- low contention on local data-structures
- potentially better cache locality

Work stealing - local worklists

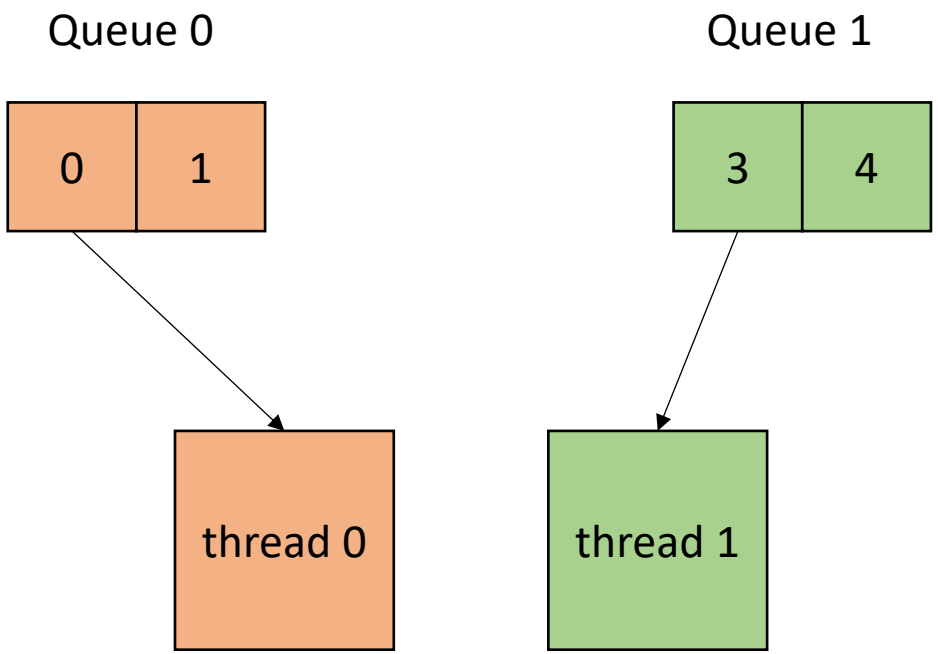
- local worklists: divide tasks into different worklists for each thread





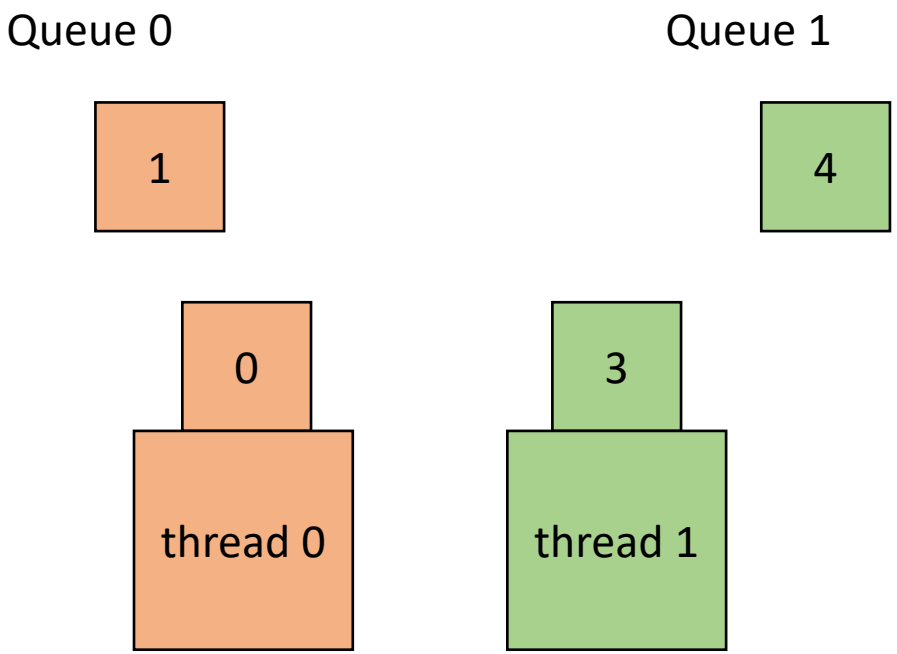
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



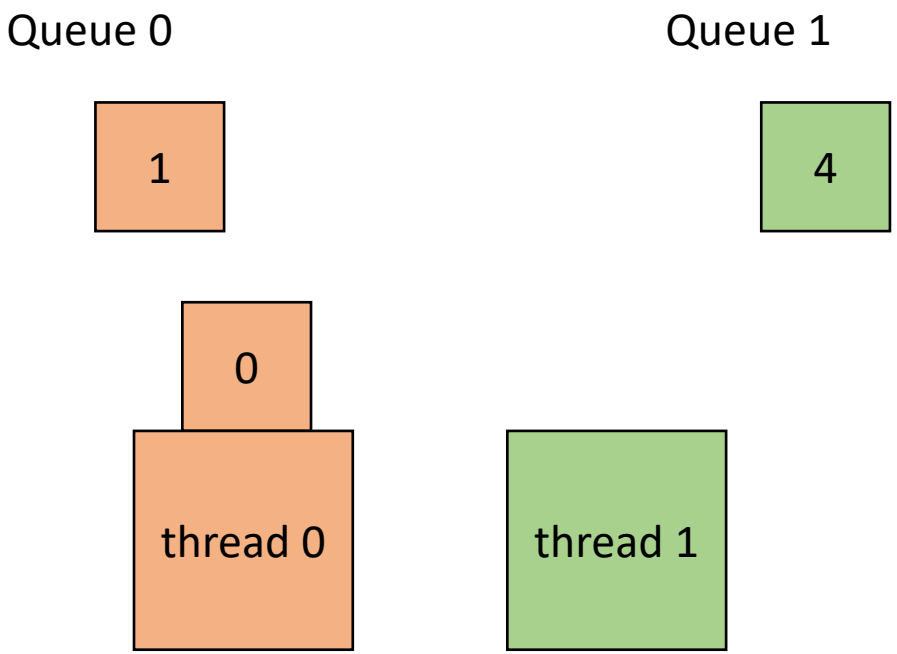
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



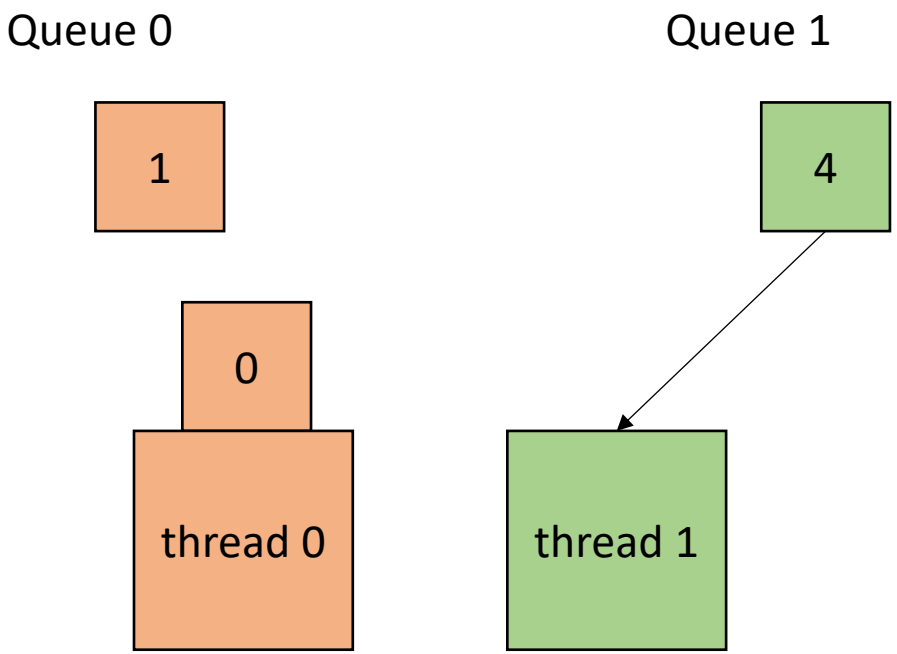
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



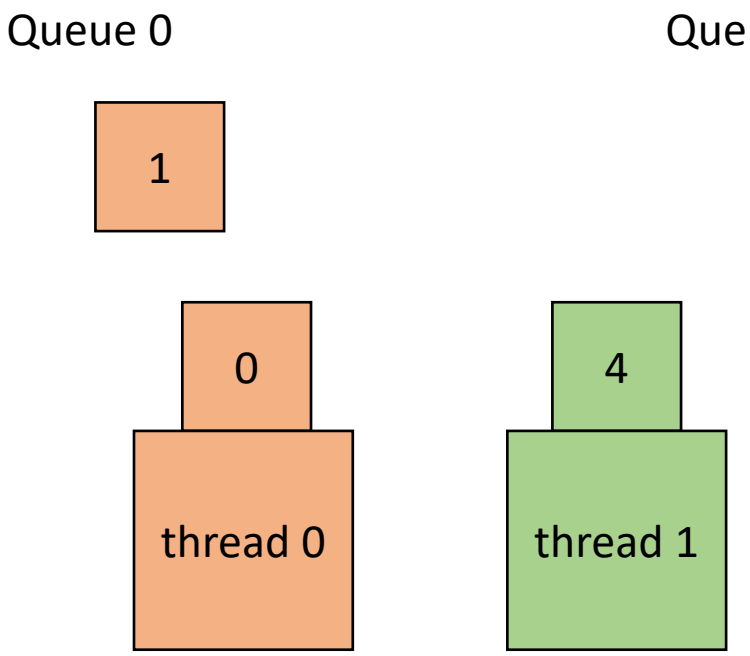
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



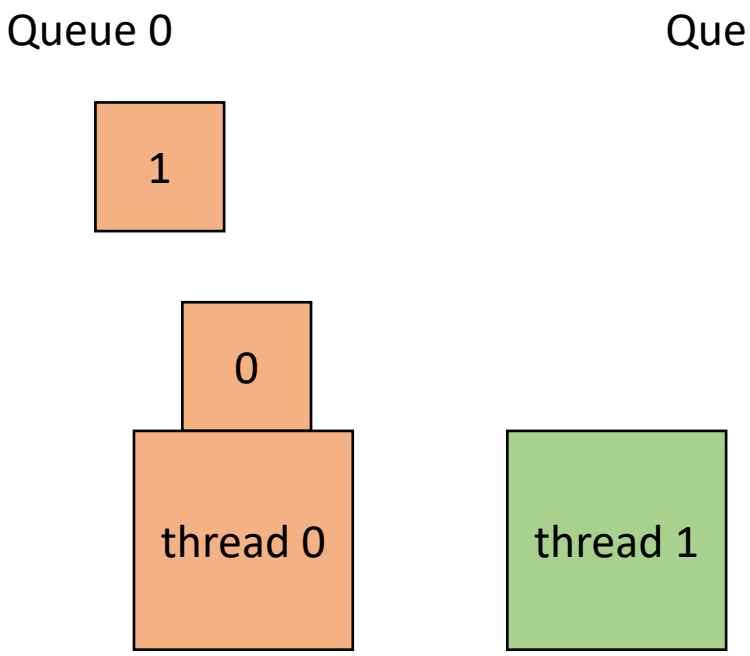
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



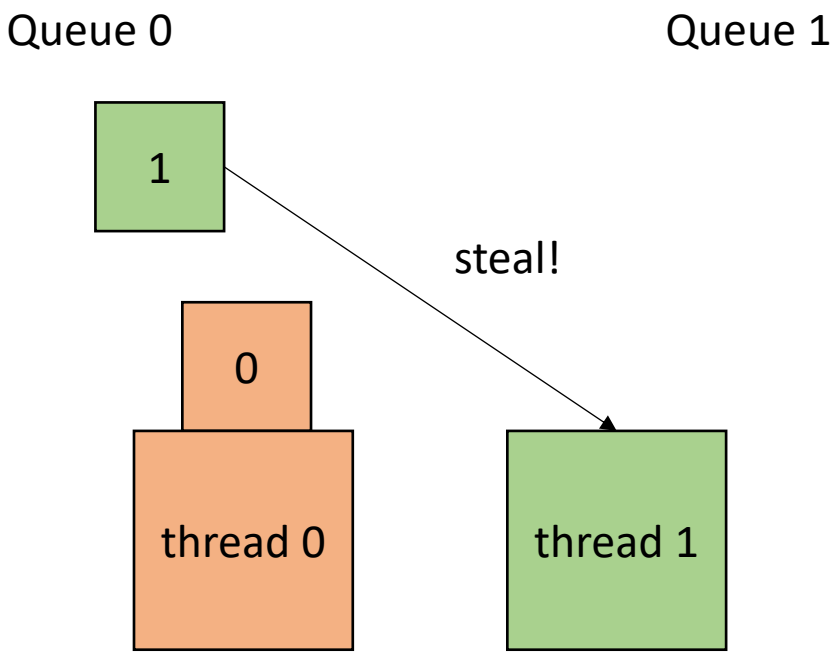
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread

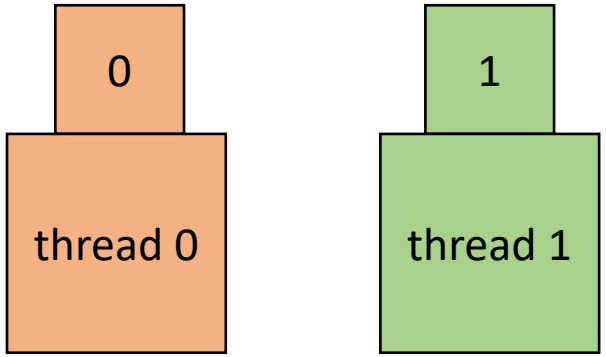


Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread

Queue 0

Queue 1



Work stealing - local worklists

- How to implement:

```
void foo() {
  ...
  for (x = 0; x < SIZE; x++) {
    // dynamic work based on x
  }
  ...
}
```

Work stealing - local worklists

- How to implement:

```
void foo() {
  ...
  for (x = 0; x < SIZE; x++) {
    // dynamic work based on x
  }
  ...
}
```

```
void parallel_loop(..., int tid) {

  for (x = 0; x < SIZE; x++) {
    // dynamic work based on x
  }
}
```

Make a new function, taking any variables used in loop body as args. Additionally take in a thread id

Work stealing - local worklists

- How to implement:

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  for (x = 0; x < SIZE; x++) {
    // dynamic work based on x
  }
  ...
}
```

```
void parallel_loop(..., int tid) {

  for (x = 0; x < SIZE; x++) {
    // dynamic work based on x
  }
}
```

Make a global array of concurrent queues

Work stealing - local worklists

- How to implement:

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  for (x = 0; x < SIZE; x++) {
    // dynamic work based on x
  }
  ...
}
```

```
void parallel_loop(..., int tid) {

  for (x = 0; x < SIZE; x++) {
    // dynamic work based on x
  }
}
```

What type of queues?

Make a global array of concurrent queues

Work stealing - local worklists

- How to implement:

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  for (x = 0; x < SIZE; x++) {
    // dynamic work based on x
  }
  ...
}
```

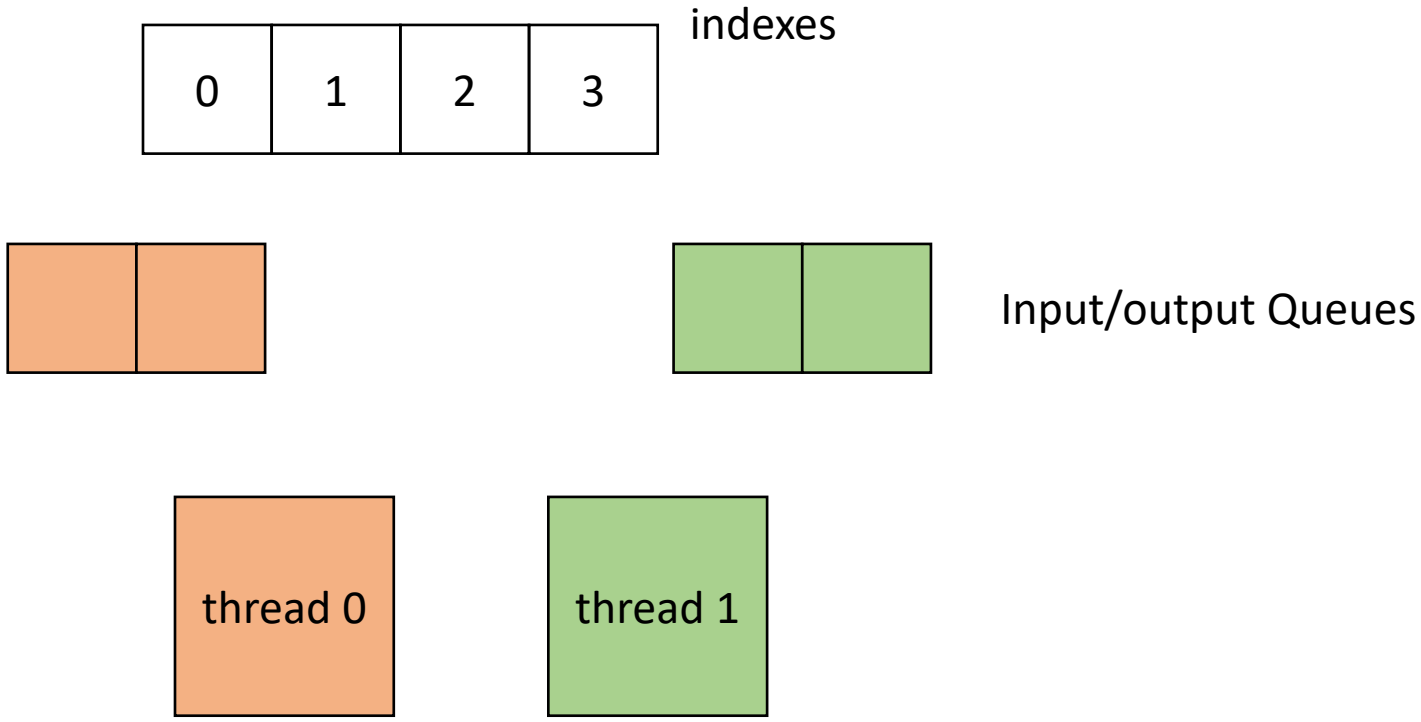
```
void parallel_loop(..., int tid) {

  for (x = 0; x < SIZE; x++) {
    // dynamic work based on x
  }
}
```

Make a global array of concurrent queues

What type of queues?
We're going to use InputOutput Queues!

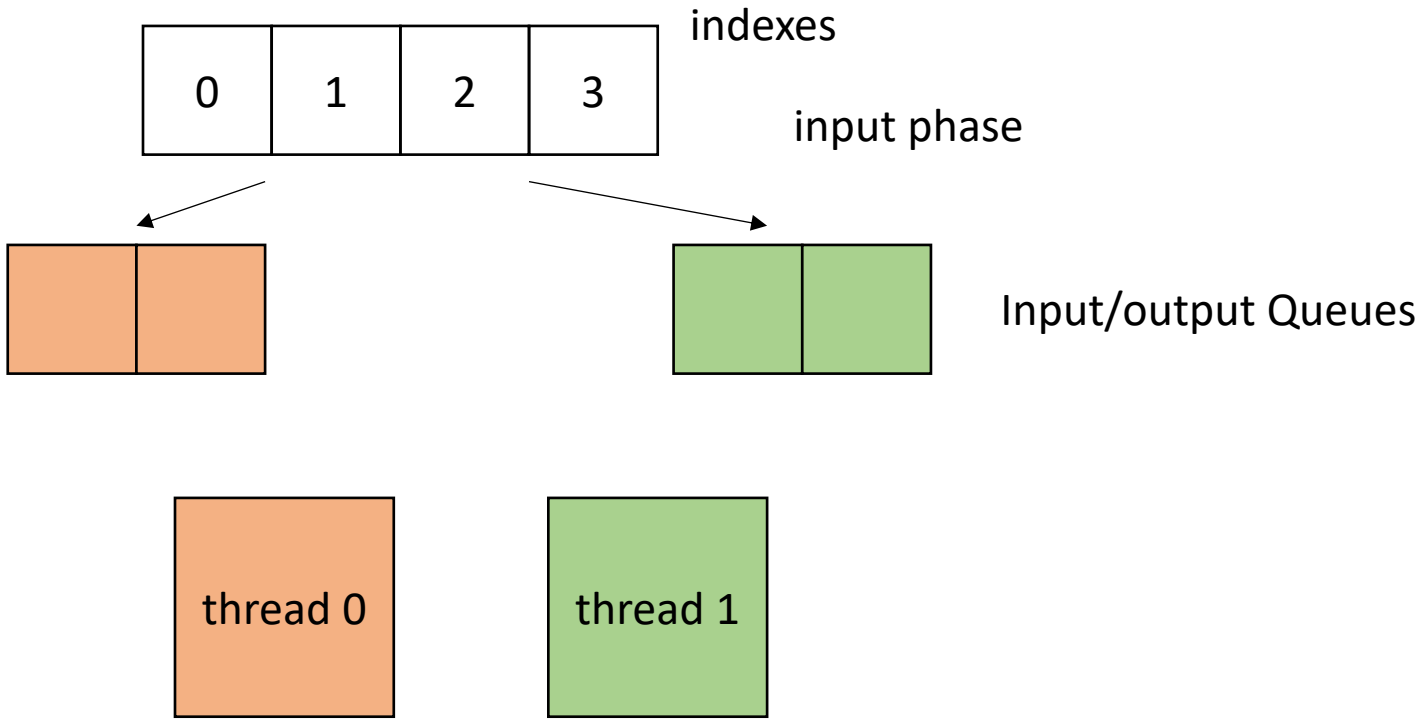
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



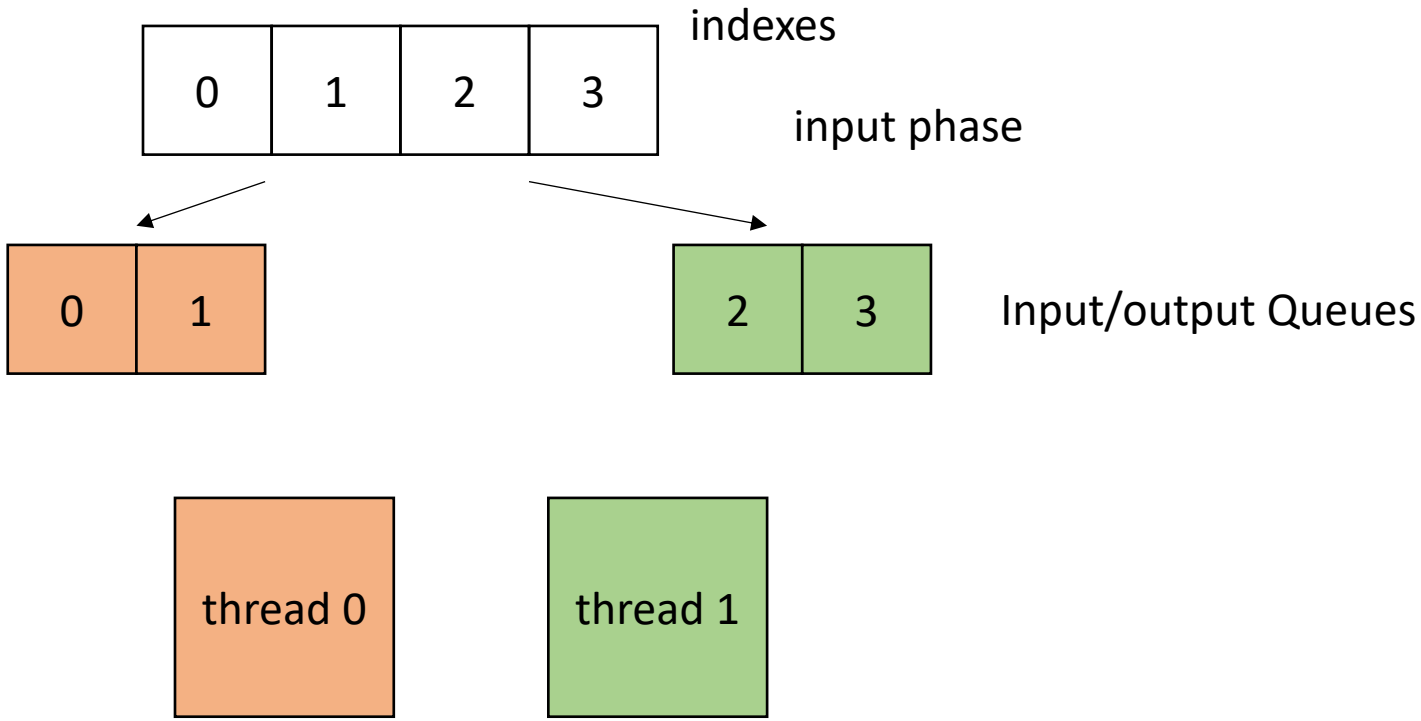
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



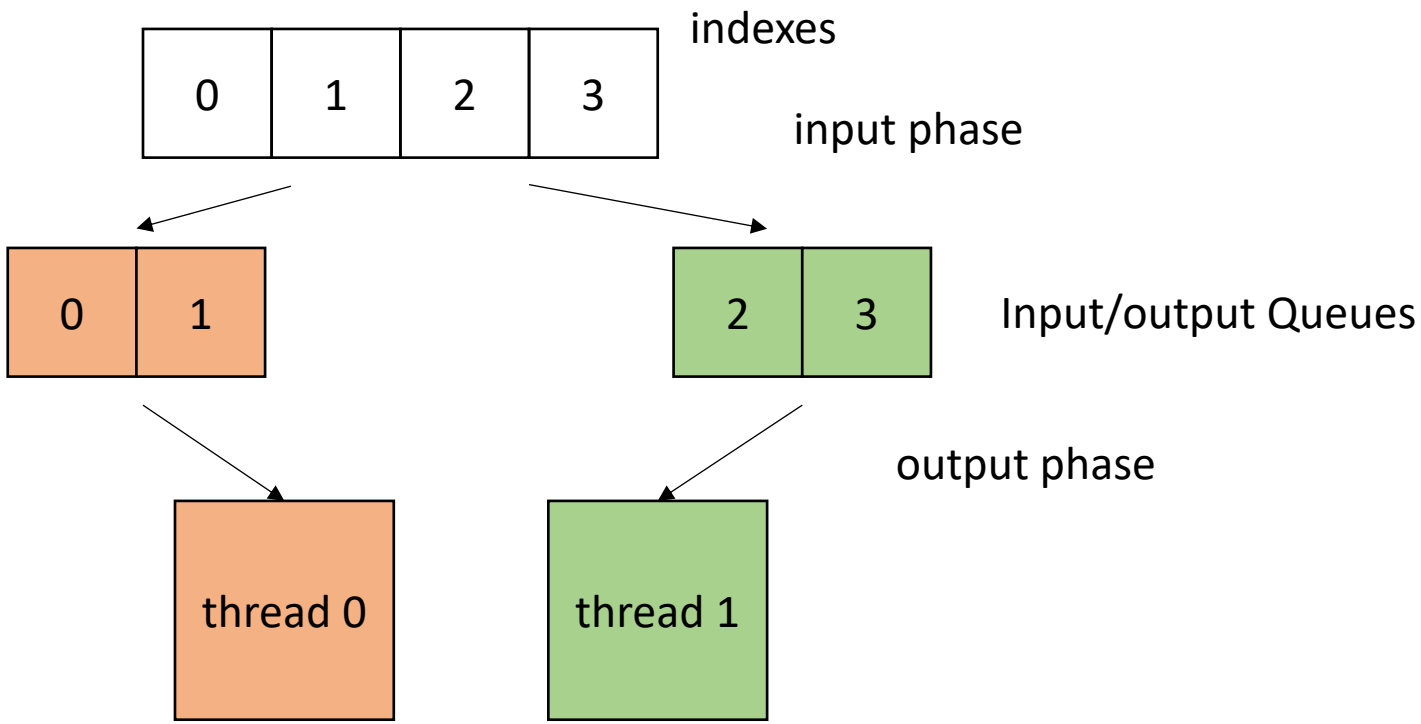
Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



Work stealing - local worklists

- local worklists: divide tasks into different worklists for each thread



Work stealing - local worklists

```
concurrent_queues cq[NUM_THREADS];
void foo() {
   ...


   ...
}
```

First we need to initialize the queues

Work stealing - local worklists

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  // Spawn threads to initialize
  // join initializing threads

  ...
}
```

```
void parallel_enq(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    cq[tid].enq(x);
  }
}
```

Just like the static schedule, except we are enqueuing

Work stealing - local worklists

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  // Spawn threads to initialize
  // join initializing threads

  ...
}
```

Make sure to account for boundary conditions!

```
void parallel_enq(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    cq[tid].enq(x);
  }
}
```

Just like the static schedule, except we are enqueuing

Work stealing - local worklists

- How to implement in a compiler:

```
NUM_THREADS = 2;
SIZE = 4;
CHUNK = 2;
```

x	0	1	2	3

tid	0	0	1	1

Make sure to account for boundary conditions!

```
void parallel_enq(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    cq[tid].enq(x);
  }
}
```

Just like the static schedule, except we are enqueuing

Work stealing - local worklists

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  // initialize queues
  // join threads

  // launch loop function
  ...
}
```

```
void parallel_loop(..., int tid, int num_threads) {

  for (x = 0; x < SIZE; x++) {
    // dynamic work based on x
  }
}
```

How do we modify the parallel loop?

Work stealing - local worklists

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  // initialize queues
  // join threads

  // launch loop function
  ...
}
```

```
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
}
```

loop until the queue is empty

Work stealing - local worklists

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  // initialize queues
  // join threads

  // launch loop function
  ...
}
```

```
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
}
```

loop until the queue is empty
Are we finished?

Work stealing - local worklists

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  // initialize queues
  // join threads

  // launch loop function
  ...
}
```

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
}
```

Track how many threads are finished

Work stealing - local worklists

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  // initialize queues
  // join threads

  // launch loop function
  ...
}
```

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {

  }
}
```

While there are threads that are still working

Work stealing - local worklists

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  // initialize queues
  // join threads

  // launch loop function
  ...
}
```

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
  }
}
```

pick a random target and steal a task

Work stealing - local worklists

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  // initialize queues
  // join threads

  // launch loop function
  // join loop threads
  ...
}
```

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
      // perform task
    }
  }
}
```

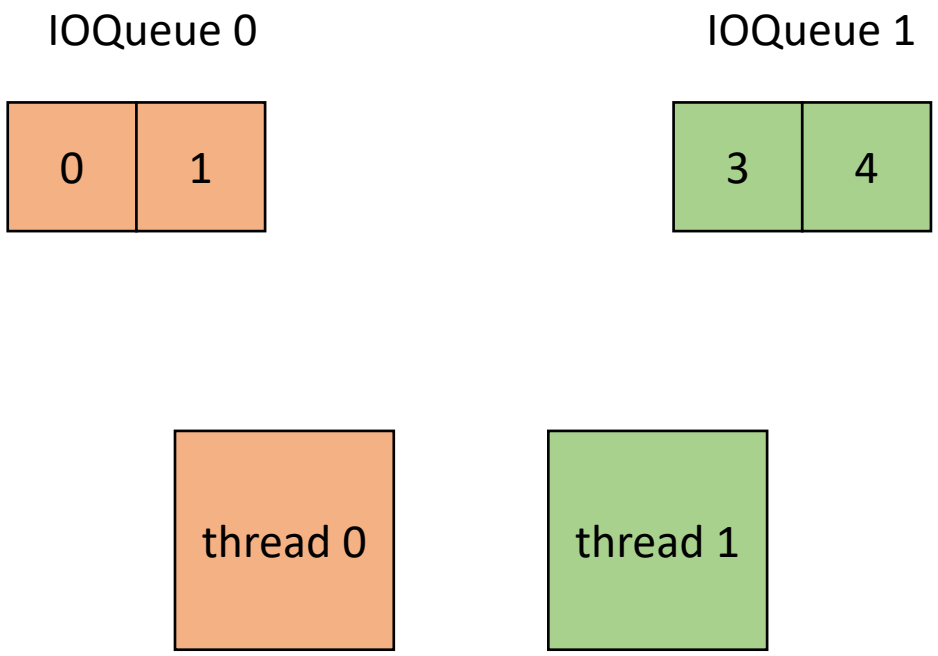
Work stealing - local worklists

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  // initialize queues
  // join threads

  // launch loop function
  // join loop threads
  ...
}
```

join the threads

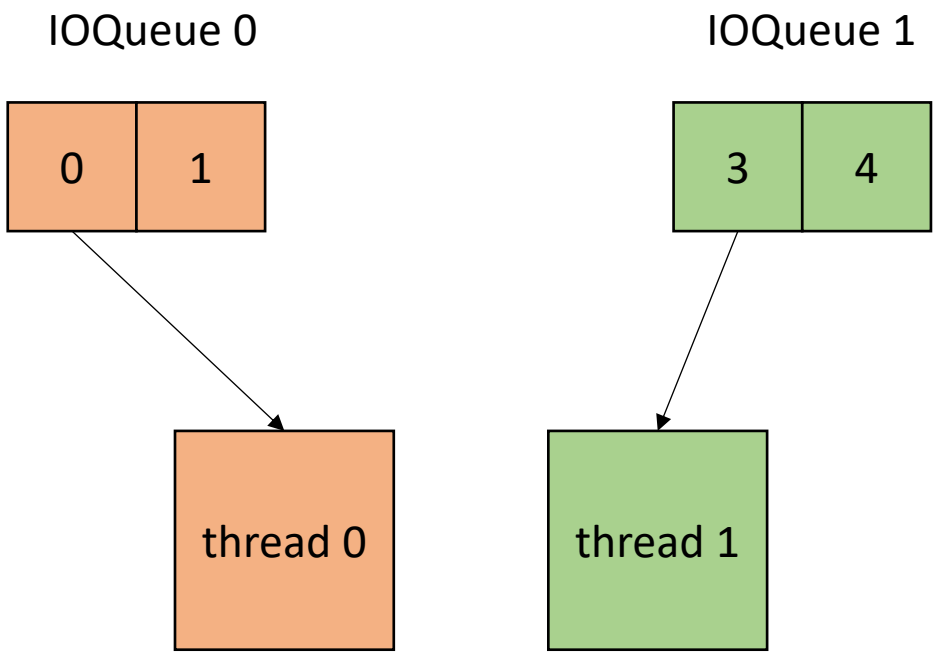
```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

Work stealing - local worklists



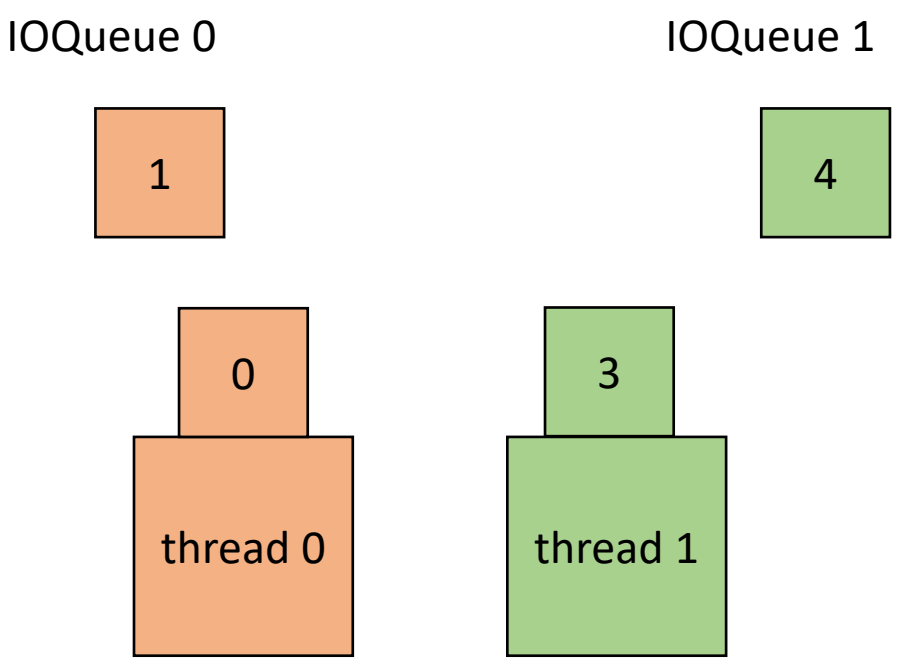

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

Work stealing - local worklists



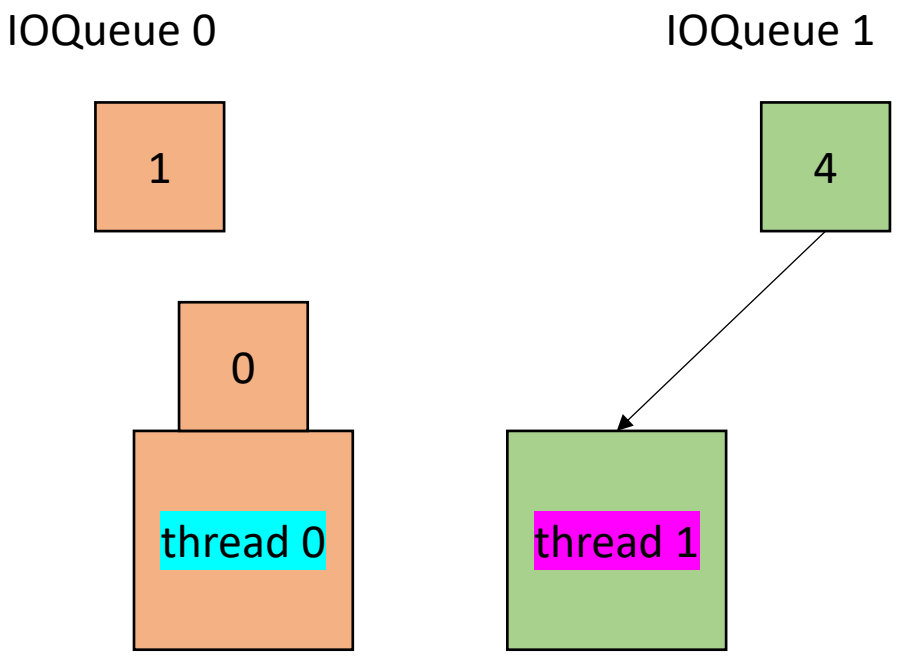

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

Work stealing - local worklists



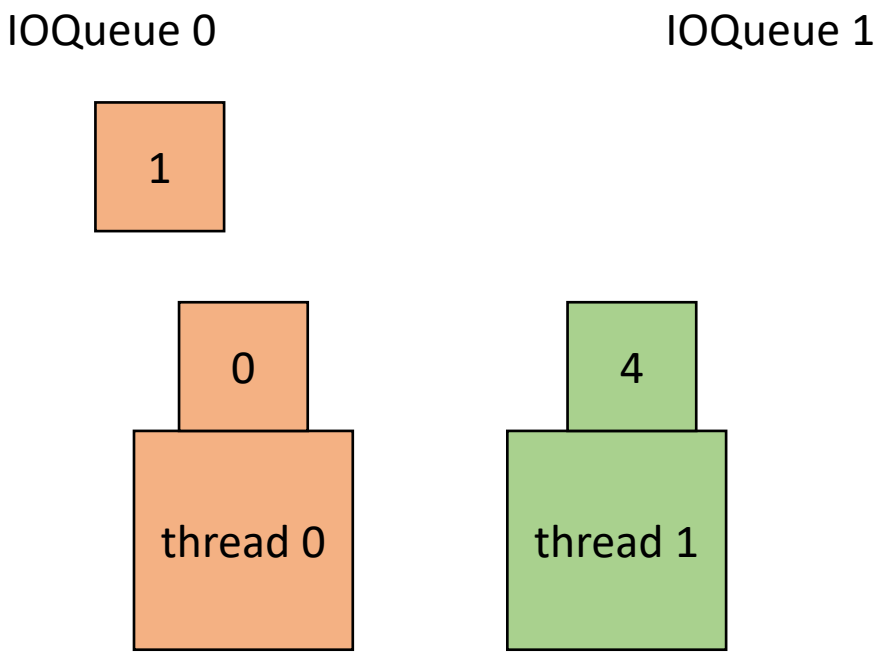

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

Work stealing - local worklists



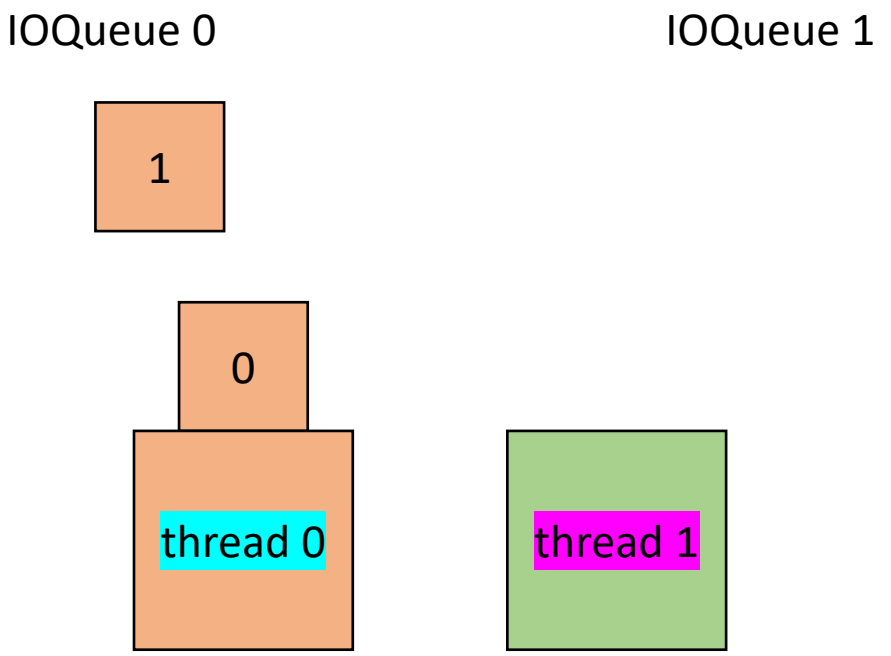

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

Work stealing - local worklists




```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

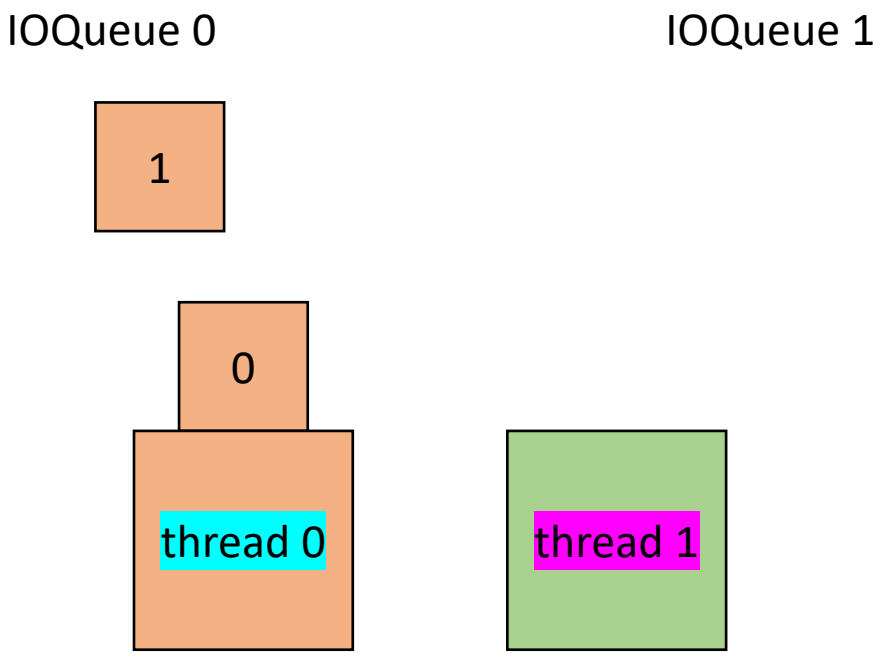
Work stealing - local worklists




```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

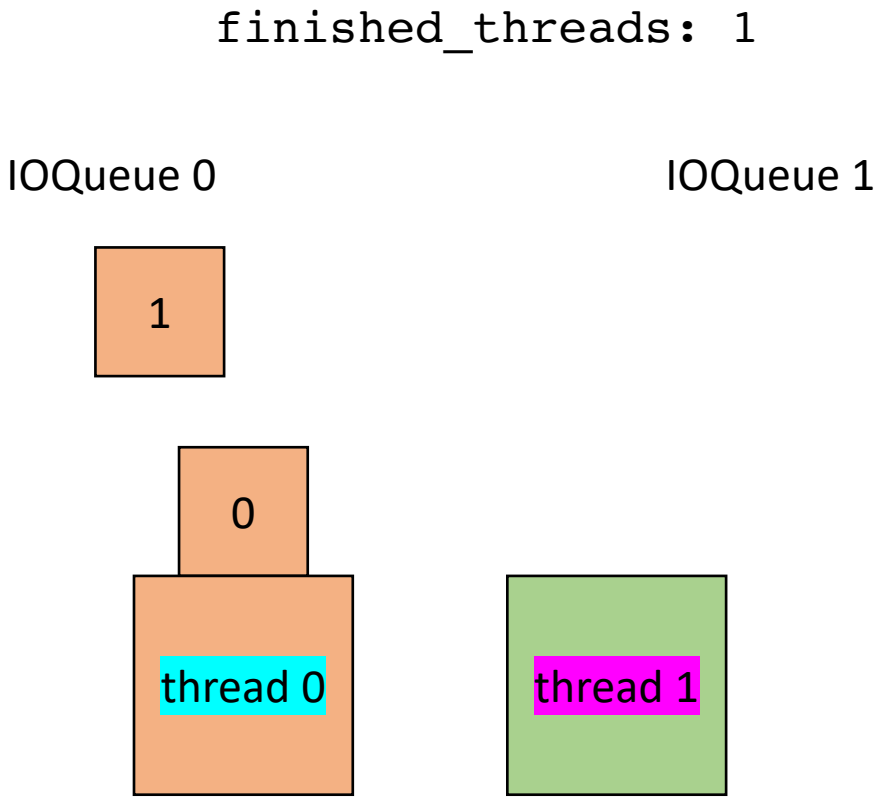
Work stealing - local worklists

finished_threads: 1




```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

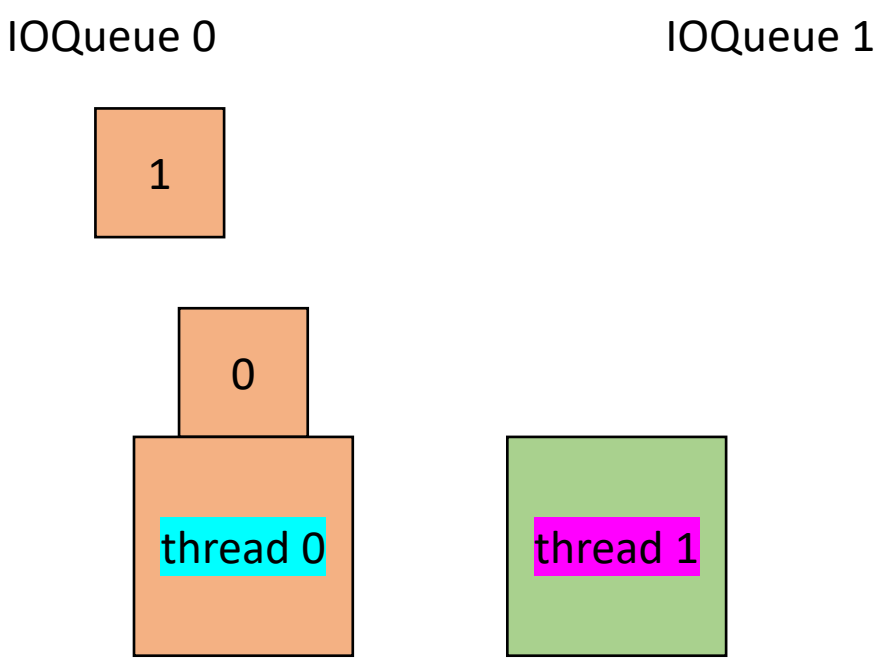
Work stealing - local worklists




```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

Work stealing - local worklists

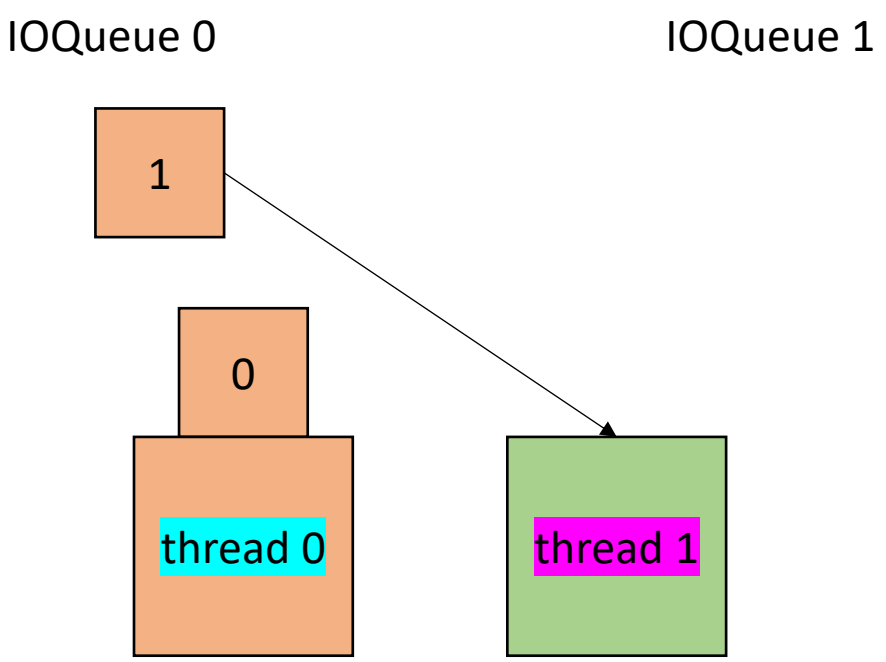
finished_threads: 1




```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

Work stealing - local worklists

finished_threads: 1



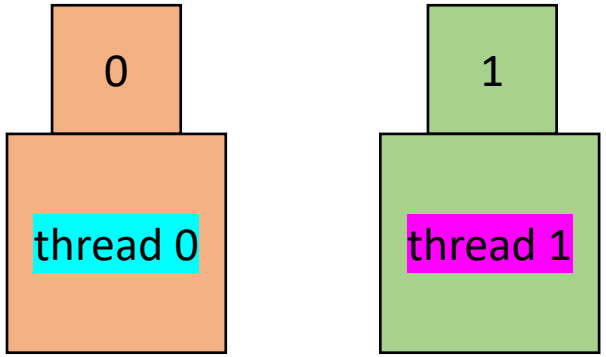

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
     // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
     int target = // pick a thread to steal from
     int task = cq[target].deq();
     if (task != -1) {
        // perform task
     }
  }
}
```

Work stealing - local worklists

`finished_threads: 1`

IOQueue 0

IOQueue 1



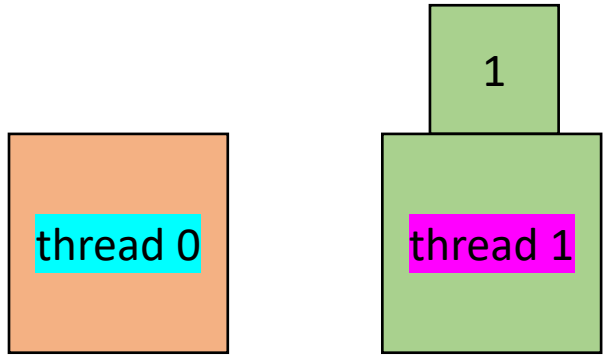

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

Work stealing - local worklists

`finished_threads: 1`

IOQueue 0

IOQueue 1



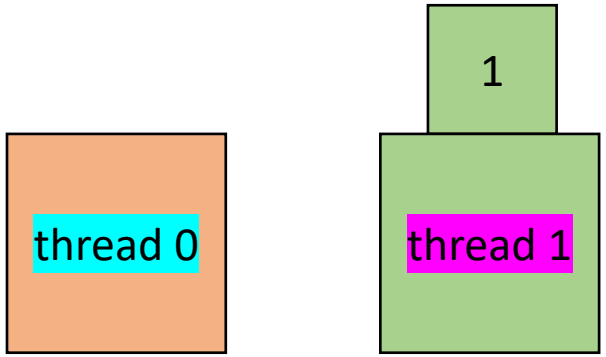

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

Work stealing - local worklists

finished_threads: 2

IOQueue 0

IOQueue 1




```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

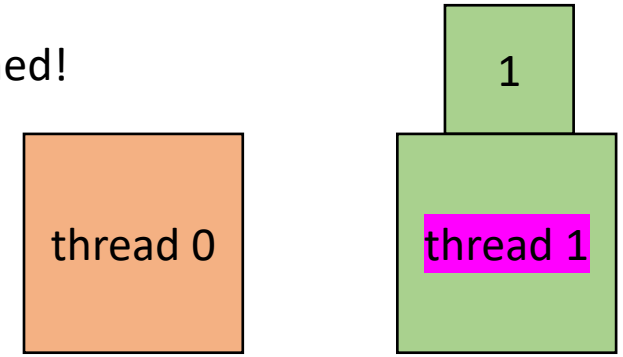
Work stealing - local worklists

finished_threads: 2

IOQueue 0

IOQueue 1

finished!



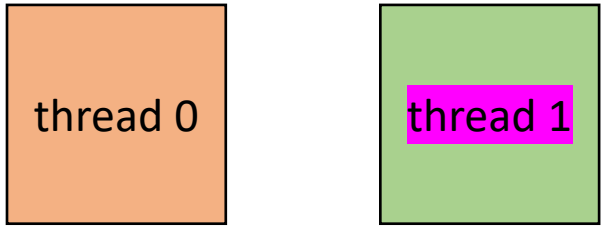

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

Work stealing - local worklists

finished_threads: 2

IOQueue 0

IOQueue 1



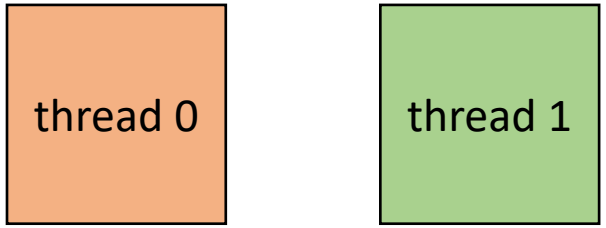

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

Work stealing - local worklists

finished_threads: 2

IOQueue 0

IOQueue 1




```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  int task = 0;
  for (x = cq[tid].deq(); x != -1; x = cq[tid].deq())
  {
    // dynamic work based on task
  }
  atomic_fetch_add(&finished_threads,1);
  while (finished_threads.load() != num_threads) {
    int target = // pick a thread to steal from
    int task = cq[target].deq();
    if (task != -1) {
       // perform task
    }
  }
}
```

Practical Parallel DOALL Loops

- Languages have various features to enable easy and flexible parallel DOALL Loops

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                   //do stuff with item
              });
```

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                   //do stuff with item
              });
```

Iterateble-object

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                   //do stuff with item
              });
```

Higher order function for iterating over object

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                   //do stuff with item
              });
```

Execution policy types

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

more in a few slides!

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                   //do stuff with item
              });
```

Iterator range

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                  //do stuff with item
              });
```

Functor or Lambda:
Execute the function
with each item in the iterated range

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                   //do stuff with item
              });
```

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

Difference between these two?

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::string> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                   //do stuff with item
              });
```

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

par_unseq requires independent loop iterations, but also allows the ability to interleave.

From: https://stackoverflow.com/questions/36246300/parallel-loops-in-c

C++

```
std::vector<std::int> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                    tmp += 1.0;
                    tmp += 2.0;
                    tmp += 3.0;
                    ...
              });
```

what would we like to do here?

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

par_unseq requires independent loop iterations, but also allows the ability to interleave.

C++

```
std::vector<std::int> foo;
std::for_each(std::execution::par_unseq,
              foo.begin(), foo.end(),
              [](auto& item) {
                    tmp += 1.0;
                    tmp += 2.0;
                    tmp += 3.0;
                    ...
              });
```

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

par_unseq requires independent loop iterations, but also allows the ability to interleave.

what would we like to do here?

```
tmp0 += 1.0;  // for item0
tmp1 += 1.0;  // for item1
tmp2 += 1.0;  // for item2
....
```

Just like in HW 1!

par_unseq requires that instructions in loops can interleaved!

C++

```
std::vector<std::int> foo;
std::for_each(std::execution::par,
              foo.begin(), foo.end(),
              [](auto& item) {
                 tyler_account += item
              });
```

global variable account, now we'd have a data race!

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

par_unseq requires independent loop iterations, but also allows the ability to interleave.

C++

```
std::vector<std::int> foo;
std::mutex m;
std::for_each(std::execution::par,
              foo.begin(), foo.end(),
              [](auto& item) {
                 m.lock();
                 tyler_account += item
                 m.unlock();
              });
```

We can fix it with mutexes

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

par_unseq requires independent loop iterations, but also allows the ability to interleave.

C++

```
std::vector<std::int> foo;
std::mutex m;
std::for_each(std::execution::par,
              foo.begin(), foo.end(),
              [](auto& item) {
                 m.lock();
                 tyler_account += item
                 m.unlock();
              });
```

Back to execution policies

options:
`seq` - sequential
`par` - parallel
`par_unseq` - also parallel

par_unseq requires independent loop iterations, but also allows the ability to interleave.

But now we can't interleave

deadlock!

```
m.lock(); // for item 0
m.lock(); // for item 1
tyler_account += item0;
tyler_account += item1;
```

We need to use `std::execution::par` if iterations cannot be interleaved (e.g. if they use mutexes)

C++ shortcomings

- Have to modify code
- No control over the parallel schedule

OpenMP

- Pragma based extension to C/C++/Fortran

```
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
```

OpenMP

- Pragma based extension to C/C++/Fortran

```
#pragma omp parallel for
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
// add -fopenmp to compile line
```

OpenMP

- Pragma based extension to C/C++/Fortran

```
#pragma omp parallel for
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
// add -fopenmp to compile line
```

launches threads to perform loop in parallel. Joins threads afterward

OpenMP

- Pragma based extension to C/C++/Fortran

```
#pragma omp parallel for
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
// add -fopenmp to compile line
```

if its so easy, why don't compilers just do this for us automatically?

OpenMP

- Pragma based extension to C/C++/Fortran

```
#pragma omp parallel for
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
// add -fopenmp to compile line
```

Performance considerations:
when is parallelism going to provide a speedup vs. slowdown?

Correctness considerations:
very difficult to determine if loop is safe to do in parallel

if its so easy, why don't compilers just do this for us automatically?

OpenMP

- Pragma based extension to C/C++/Fortran

```
for (x = 0; x < SIZE; x++) {
  for (y = x; y < SIZE; y++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

What about irregular loops?

OpenMP

- Pragma based extension to C/C++/Fortran

```
#pragma omp parallel for schedule(dynamic)
for (x = 0; x < SIZE; x++) {
  for (y = x; y < SIZE; y++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

What about irregular loops?

Schedule keyword

OpenMP

- Pragma based extension to C/C++/Fortran

```
#pragma omp parallel for schedule(dynamic)
for (x = 0; x < SIZE; x++) {
  for (y = x; y < SIZE; y++) {
    a[x,y] = b[x,y] + c[x,y];
  }
}
```

What about irregular loops?

Schedule keyword

different types of schedules

OpenMP

- Schedules:
  - From http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

schedule(static, chunk-size)

```
schedule(static):
****************
                ****************
                                ****************
                                                ****************
```

```
schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****
```

```
schedule(static, 8):
********                        ********
        ********                        ********
                ********                        ********
                        ********                        ********
```

from: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

schedule(dynamic, chunk-size)

schedule(dynamic, 1):

schedule(dynamic, 4):

schedule(dynamic, 8):

from: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

schedule(guided)

from: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

`schedule(runtime)`

You set the schedule in your code!

```
void omp_set_schedule(omp_sched_t kind, int chunk_size);
```

`schedule(auto)`

You let the system/compiler decide

from: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

Rest of module

- We will look at general concurrent sets:
  - concurrent add and remove methods
  - start off with locks
  - move to coarse-grained locks
  - end with lock-free

- May take 2 lectures, that’s okay. We have a spare slot in the schedule

See you on Friday

- I hope to be feeling better by then. I'll take another test Friday morning and let you know.

- Work on midterm!
  - Ask on piazza if you have questions or comments

- Homework 3 is out and you have everything you need to do it!

- Do the quiz!