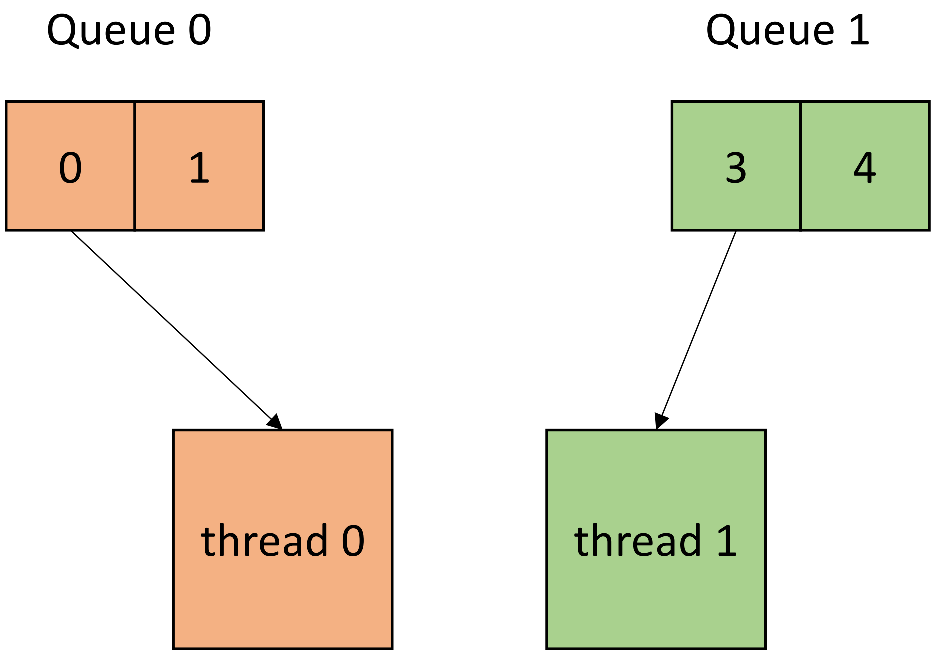
CSE113: Parallel Programming

May 11, 2021

- **Topic**: DOALL Loops and Workstealing
  - DOALL loops
  - Parallel Schedules:
    - Static
    - global worklist
    - local worklist



Announcements

- HW2 and midterm deadlines are over!
  - Aiming to have midterm grades by Thursday
  - Aiming to have HW2 grades next Thursday

- HW3 is out:
  - A day off schedule. Due on Friday the 21
  - Material will be be last weeks slides and this weeks slides

Announcements

- Last day of Module 3
  - Next week we move on to reasoning about parallel computations
  - Relaxed Memory models
  - Barriers
  - Forward Progress
- May 20: guest lecture
  - Message passing concurrency: Hugues Evrard
  - Testing GPU compilers: Alastair Donaldson

Announcements

- Office hours:
  - Mine will be canceled this week: sorry!
  - I will spend extra time on Piazza and mailing list

- HW 1:
  - Any questions about grades need to be asked by May 20
  - Any public git repos that you used to develop should be made private please!

Quiz

Quiz

- Go over answers

Schedule

- DOALL Loops

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

Schedule

- **DOALL Loops**

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

adds two arrays

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

adds elements with neighbors

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

are they the same if you traverse them backwards?

adds two arrays

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

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

adds elements with neighbors

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

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

are they the same if you traverse them backwards?

adds two arrays

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

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

adds elements with neighbors

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

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

No!

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]
}
```

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]
}
```

No!

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

These are **DOALL** loops:

- Loop iterations are independent
- You can do them in ANY order and get the same results

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

These are **DOALL** loops:

- Loop iterations are independent
- You can do them in ANY order and get the same results

- Most importantly: you can do the iterations in parallel!
- Assign each thread a set of indices to compute

DOALL Loops

- Given a nest of For loops, can we make the outer-most loop parallel?
  - Safely
  - Efficiently

DOALL Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays (only side-effects are array writes)
  - Array bases are disjoint and constant
  - Bounds, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1

```
for (int i = 0; i < dim1; i++) {
  for (int j = 0; j < dim3; j++) {
    for (int k = 0; k < dim2; k++) {
      a[i][j] += b[i][k] * c[k][j];
    }
  }
}
```

matrix multiplication example

DOALL Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays (only side-effects are array writes)
  - Array bases are disjoint and constant
  - Bounds, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1

```
for (int i = 2; i < 100; i+=3) {
  a[i] = c[i + 128];
}
```

DOALL Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays (only side-effects are array writes)
  - Array bases are disjoint and constant
  - Bounds, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1

```
for (int i = 2; i < 100; i+=3) {
  a[i] = c[i + 128];
}
```

substitute:

```
i = 3*j + 2
```

DOALL Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays (only side-effects are array writes)
  - Array bases are disjoint and constant
  - Bounds, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1

```
for (int j = 0; j < 33; j++) {
  a[3*j + 2] = c[3*j + 2 + 128];
}
```

substitute:

```
i = 3*j + 2
```

DOALL Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays (only side-effects are array writes)
  - Array bases are disjoint and constant
  - Bounds, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1

```
for (int i = 2; i < 100; i+=3) {
  a[i] = c[i + 128];
}
```

```
for (int j = 0; j < 33; j+=1) {
  a[3*j+2] = c[(3*j+2) + 128];
}
```

DOALL Loops

- Given a nest of ***candidate*** *For* loops, determine if we can we make the outer-most loop parallel?
  - Safely
  - efficiently

- Criteria: every iteration of the outer-most loop must be *independent*
  - The loop can execute in any order, and produce the same result

Safety Criteria

- How do we check this?
  - If the property doesn't hold then there exists 2 iterations, such that if they are re-ordered, it causes different outcomes for the loop.

  - **Write-Write conflicts**: two distinct iterations write different values to the same location

  - **Read-Write conflicts**: two distinct iterations where one iteration reads from the location written to by another iteration.

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

index calculation based on the loop variable

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

index calculation based on the loop variable
Computation to store in the memory location

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
    a[index(i)] = loop(i);
}
```

Write-write conflicts:

for two distinct iteration variables:

i_x != i_y

Check:

index(i_x) != index(i_y)

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
    a[index(i)] = loop(i);
}
```

Write-write conflicts:

for two distinct iteration variables:
i_x != i_y
Check:
index(i_x) != index(i_y)

Why?
Because if
index(i_x) == index(i_y)
then:
a[index(i_x)] will equal
either loop(i_x) or loop(i_y)
depending on the order

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*

```
for (i = 0; i < size; i++) {
   a[write_index(i)] = a[read_index(i)] + loop(i);
}
```

Read-write conflicts:

for two distinct iteration variables:

i_x != i_y

Check:

write_index(i_x) != read_index(i_y)

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*

```
for (i = 0; i < size; i++) {
    a[write_index(i)] = a[read_index(i)] + loop(i);
}
```

Read-write conflicts:

for two distinct iteration variables:
i_x != i_y
Check:
write_index(i_x) != read_index(i_y)

Why?

if i_x iteration happens first, then iteration i_y reads an updated value.

if i_y happens first, then it reads the original value

Examples:

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

Examples:

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

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

Examples:

```
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;
}
```

Examples:

```
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;
}
```

Examples:

```
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;
}
```

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

Schedule

- DOALL Loops

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

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];
}
```

Is this a DOALL loop?

Parallel Schedules

- Consider the following program:

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

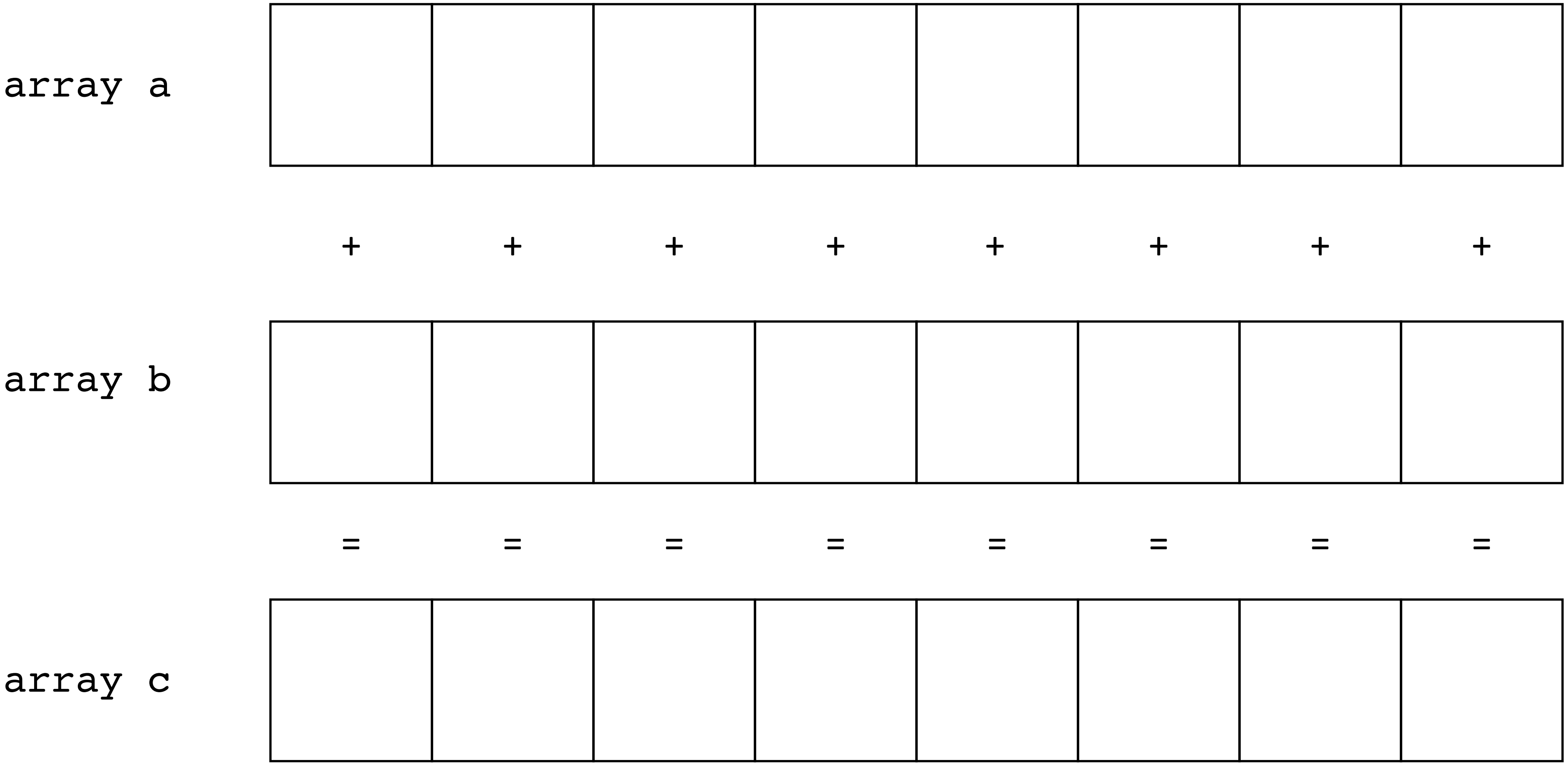
We want to compute

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

Is this a DOALL loop?

How should we parallelize it?

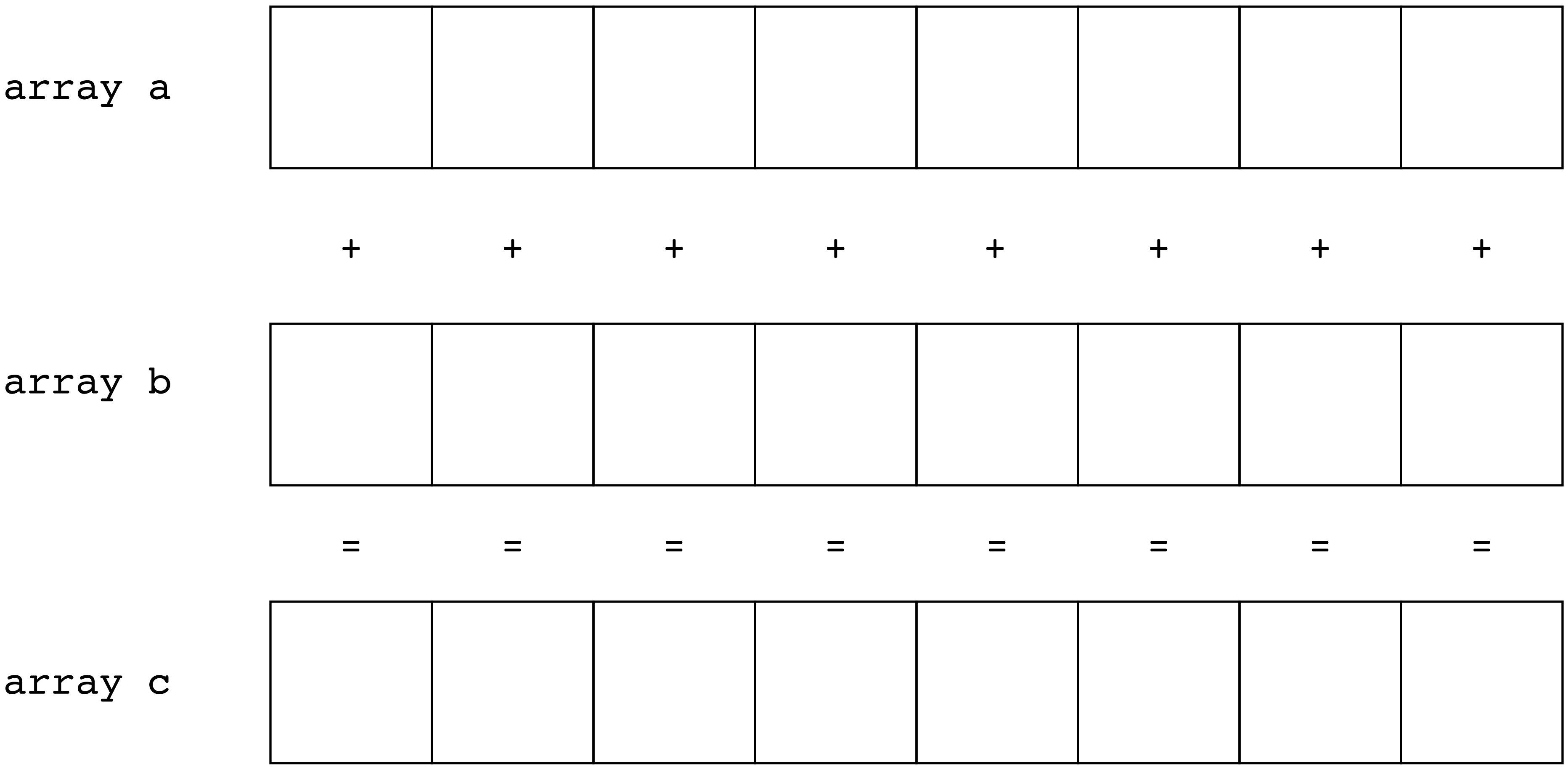
Parallel Schedules



Parallel Schedules

Computation can easily be divided into threads

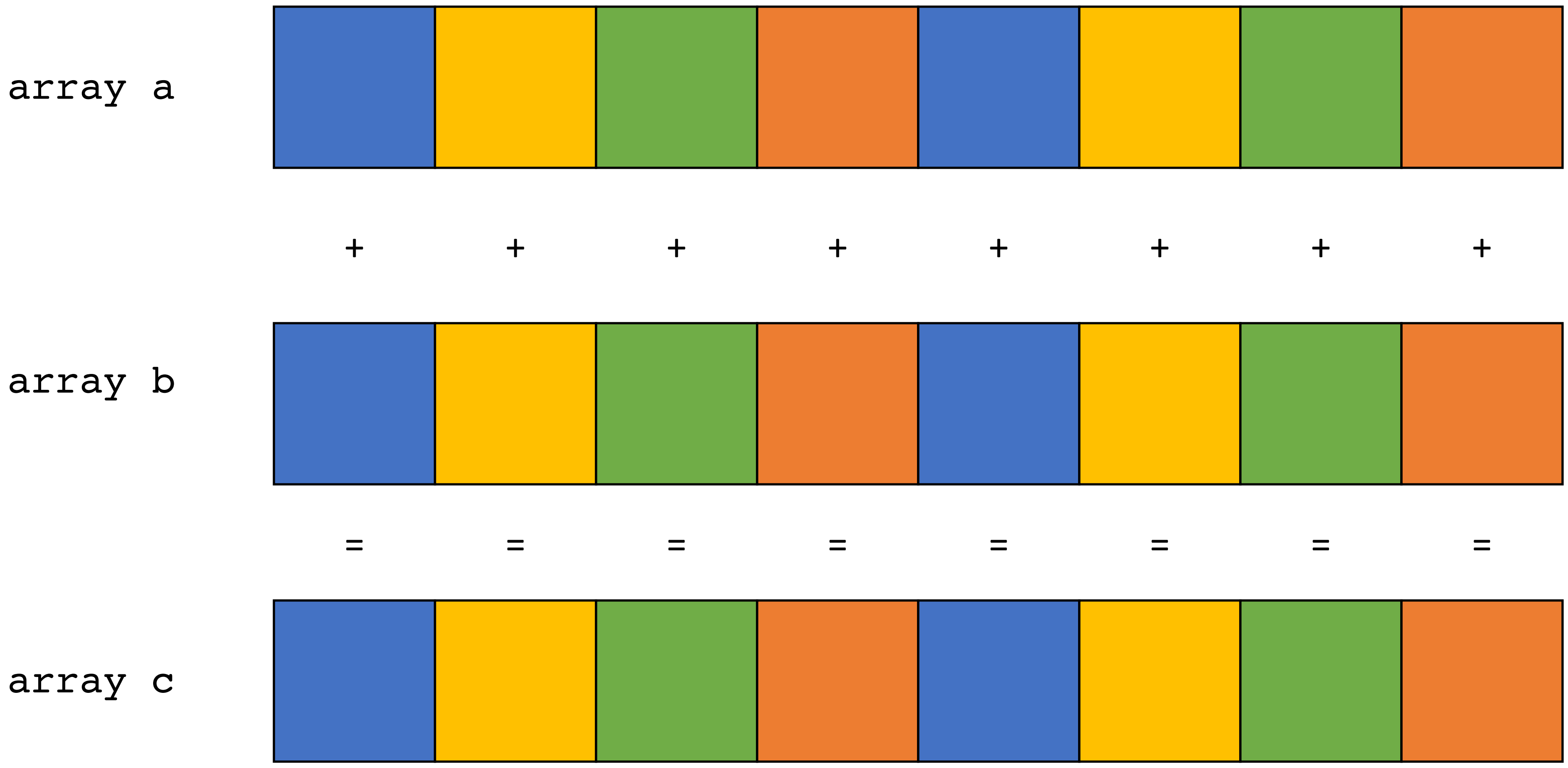
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

Computation can easily be divided into threads

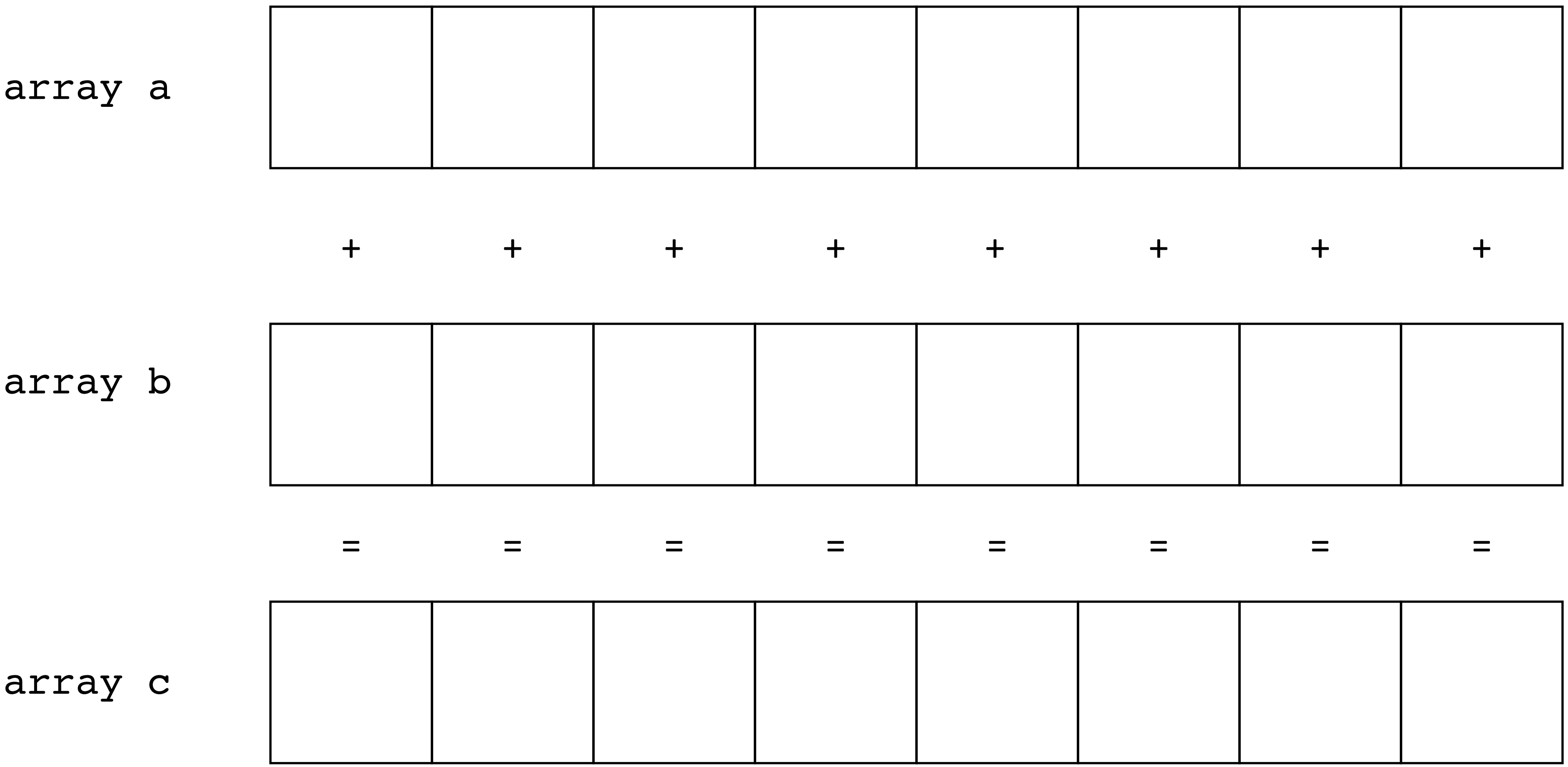
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

Computation can easily be divided into threads

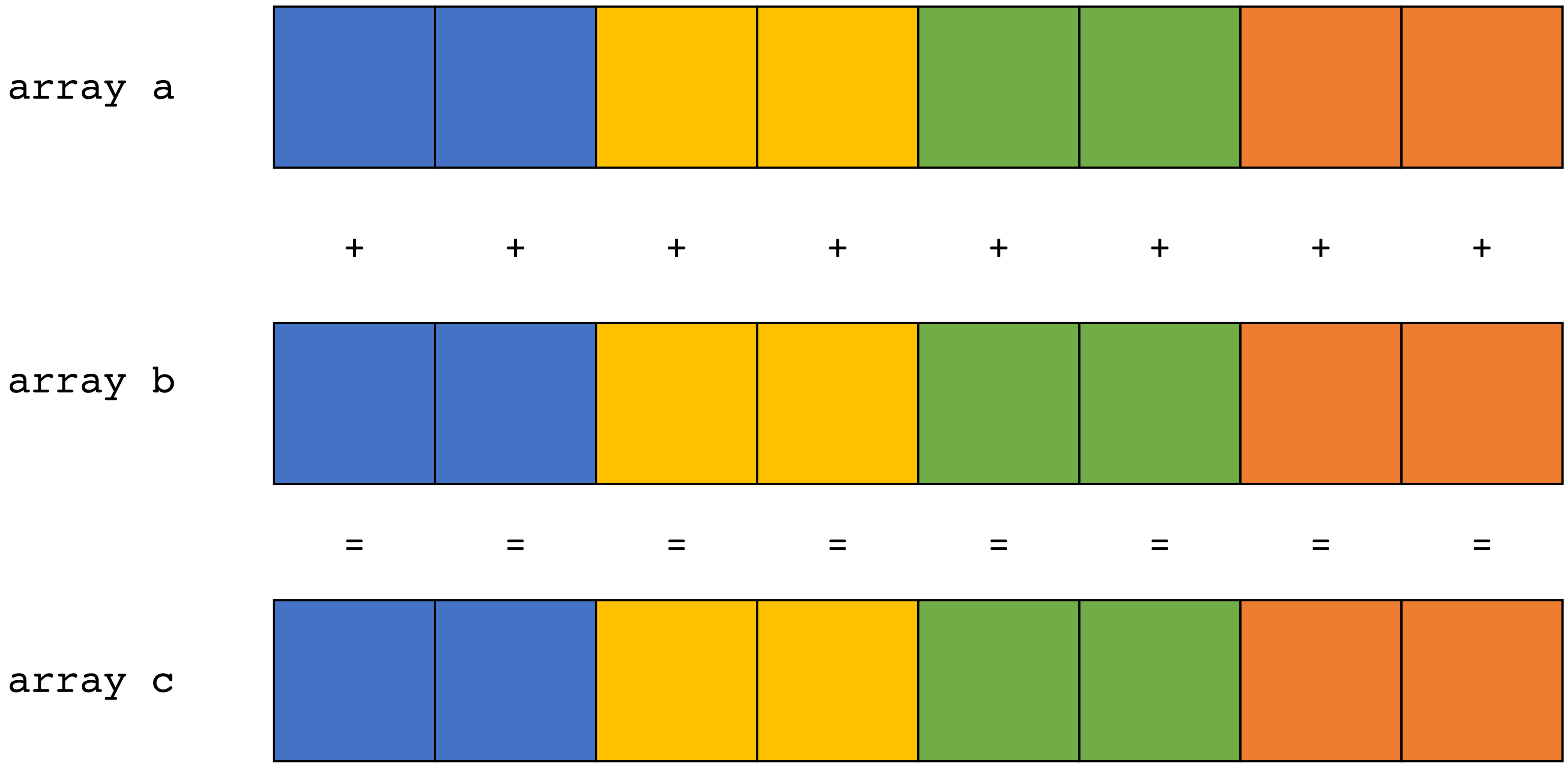
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

Computation can easily be divided into threads

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



Parallel Schedules

- Which one is more efficient?

Parallel Schedules

- Which one is more efficient?

- These are called Parallel Schedules for DOALL Loops
- We will discuss several of them today.

Schedule

- DOALL Loops

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

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

0	1	2	3	4	5	6	7		SIZE -1

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

say SIZE / NUM_THREADS = 4

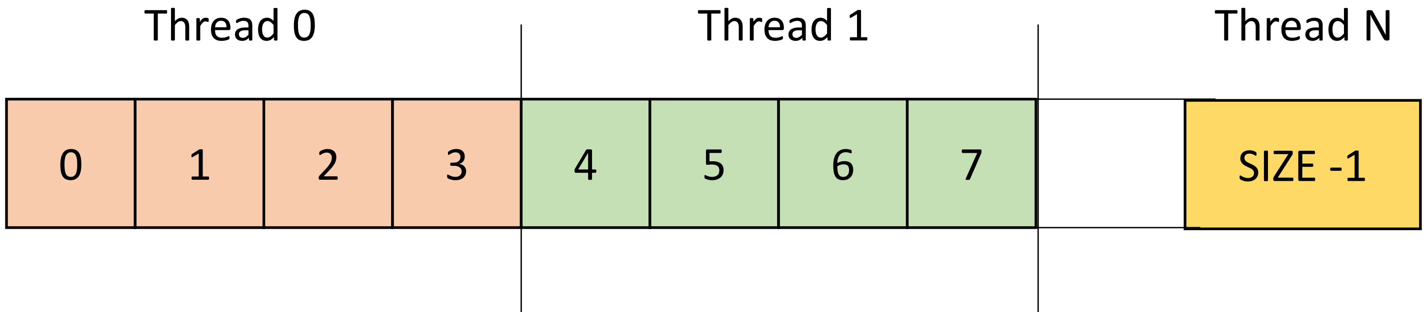
0	1	2	3	4	5	6	7		SIZE -1

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

say SIZE / NUM_THREADS = 4



Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

make a new function with the for loop inside. Pass all needed variables as arguments. Take an extra argument for a thread id

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
~~  for (int x = 0; x < SIZE; x++) {~~
~~  // Each iteration takes roughly~~
~~  // equal time~~
~~  }~~
...
}
```

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

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

make a new function with the for loop inside. Pass all needed variables as arguments. Take an extra argument for a thread id

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

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

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

determine chunk size in new function

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

```
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
  }
}
```

Set new loop bounds

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int t = 0; t < NUM_THREADS; t++) {
    spawn(parallel_loop(..., t, NUM_THREADS))
  }
  join();
...
}
```

```
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
  }
}
```

You will need to adapt the thread spawn, join to C++

Spawn threads

Static schedule

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

0	1	2	3	4	5	6	7

```
chunk_size = 4

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

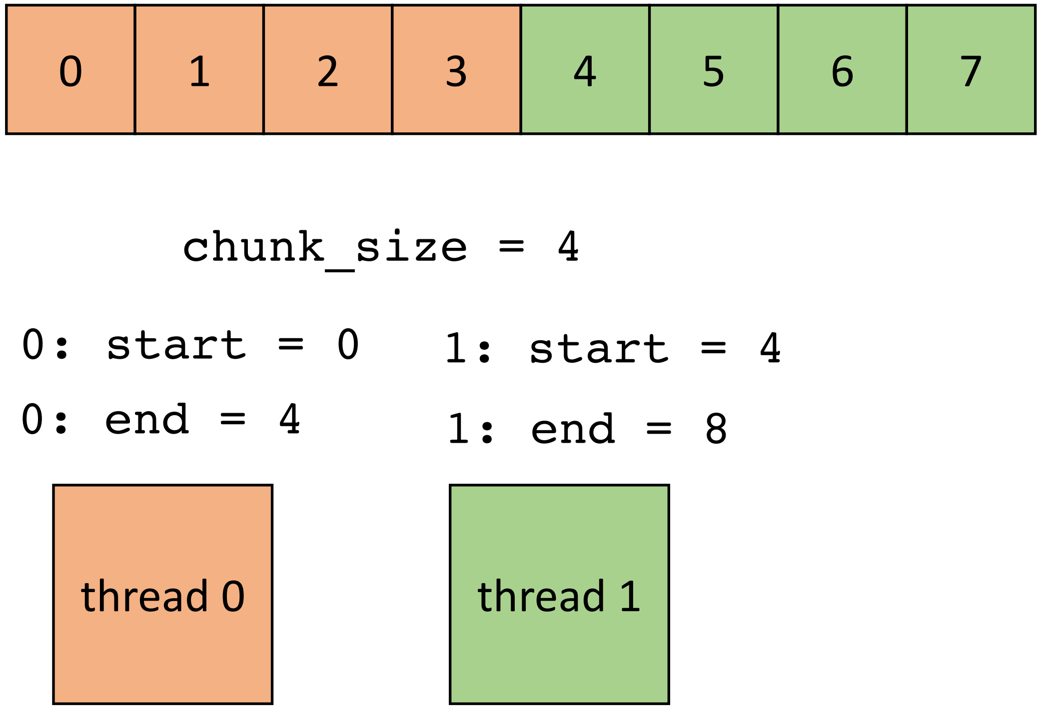
thread 0

thread 1

```
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 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
  }
}
```

End example

Static schedule

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

0	1	2	3	4	5	6	7	8

```
chunk_size = ?

0: start = ?     1: start = ?
0: end = ?       1: end = ?
```

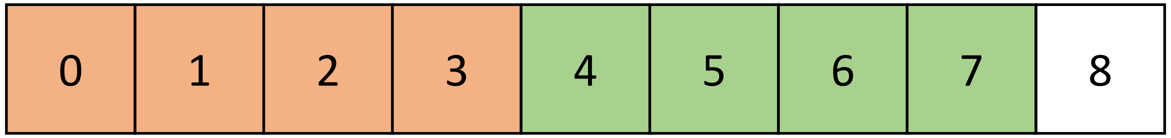
thread 0

thread 1

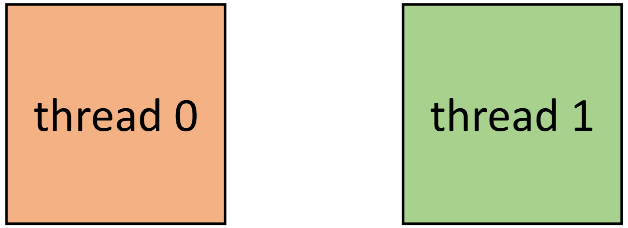
```
void parallel_loop(..., int tid, int num_threads)
{

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



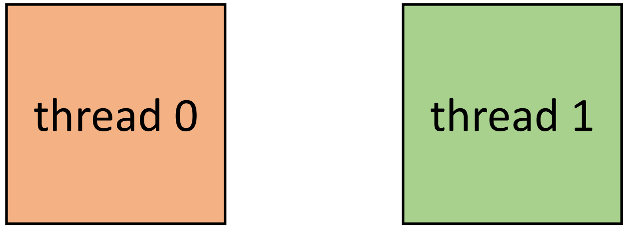

```
void parallel_loop(..., int tid, int num_threads)
{

  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

0	1	2	3	4	5	6	7	8

```
chunk_size = 4

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

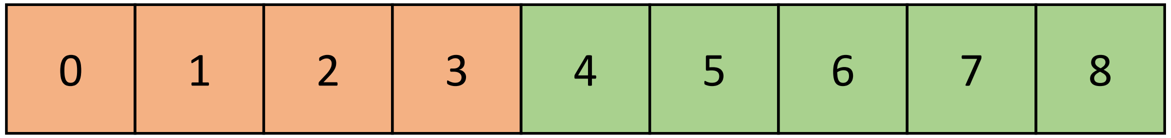



```
void parallel_loop(..., int tid, int num_threads)
{

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

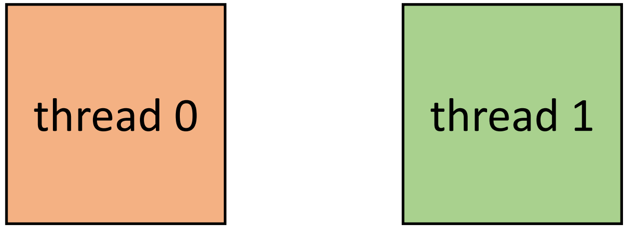
Static schedule

last thread gets more work

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




```
chunk_size = 4

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



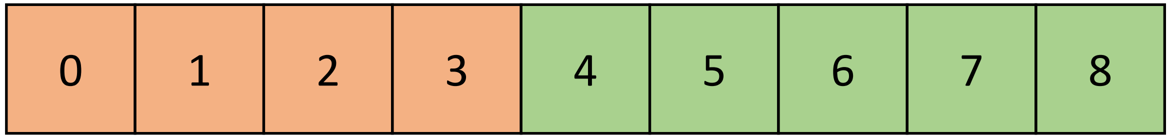

```
void parallel_loop(..., int tid, int num_threads)
{

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

Static schedule

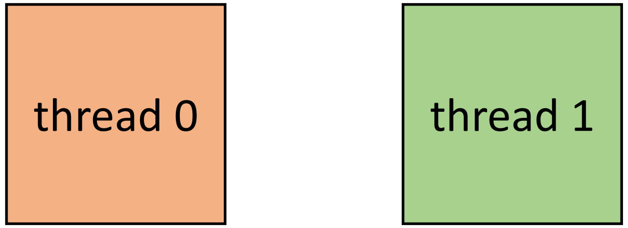
- Example, 2 threads/cores, array of size 9

last thread gets more work

What is the worst case?




```
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;
  if (tid == num_threads - 1) {
    end = SIZE;
  }
  for (int x = start; x < end; x++) {
    // work based on x
  }
}
```

End example

Static schedule

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

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

0	1	2	3	4	5	6	7	8

```
chunk_size = 4

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

thread 0

thread 1

```
void parallel_loop(..., int tid, int num_threads)
{
  int chunk_size =
  (SIZE+(NUM_THREADS-1))/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

0	1	2	3	4	5	6	7	8	9

out of bounds

```
chunk_size = 5

0: start = 0      1: start = 5
0: end = 5        1: end = 10
```

thread 0

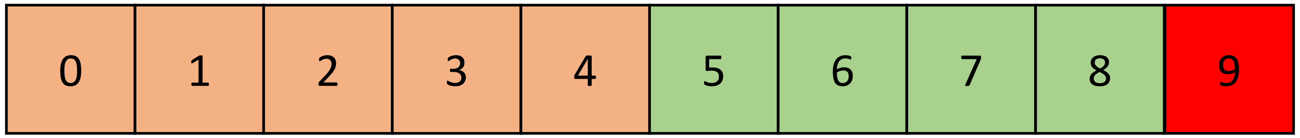
thread 1

```
void parallel_loop(..., int tid, int num_threads)
{
  int chunk_size =
  (SIZE+(NUM_THREADS-1))/NUM_THREADS;
  int 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
  }
}
```

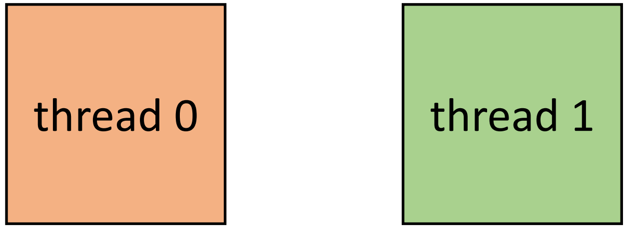
Static schedule

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

out of bounds




```
chunk_size = 5

0: start = 0    1: start = 5
0: end = 5      1: end = 10
```




```
void parallel_loop(..., int tid, int num_threads)
{
  int chunk_size =
  (SIZE+(NUM_THREADS-1))/NUM_THREADS;
  int chunk_size = SIZE / 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

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?

0	1	2	3	4	5	6	7	8

```
chunk_size = 5

0: start = 0     1: start = 5
0: end = 5       1: end = 9
```

thread 0

thread 1

```
void parallel_loop(..., int tid, int num_threads)
{
  int chunk_size =
  (SIZE+(NUM_THREADS-1))/NUM_THREADS;
  int chunk_size = SIZE / 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
  }
}
```

End example

Schedule

- DOALL Loops

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

5 minute break here

Irregular parallelism in loops

- Tasks are not balanced
- Appears in lots of emerging workloads

Irregular parallelism in loops

- Tasks are not balanced

- Appears in lots of emerging workloads

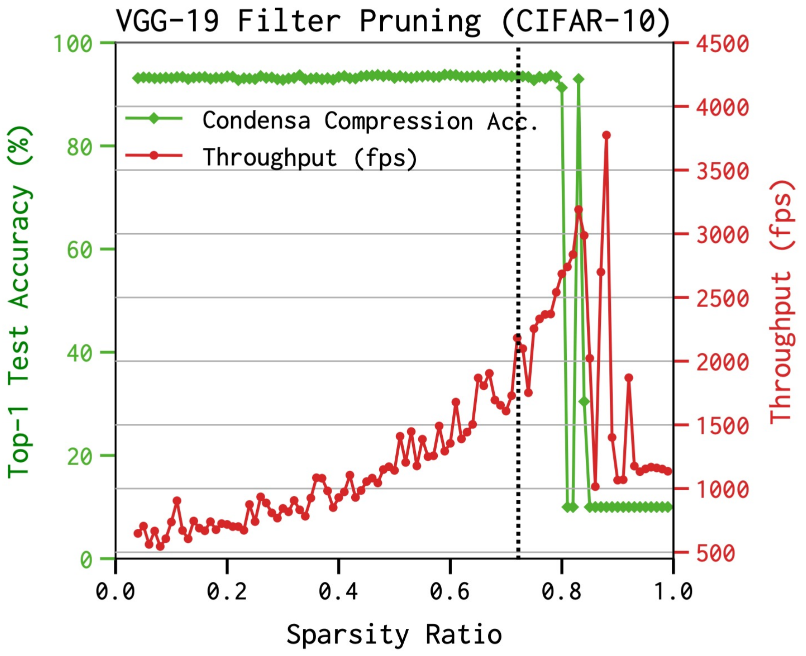
social network analytics where threads are parallel across users

Irregular parallelism in loops

- Tasks are not balanced

- Appears in lots of emerging workloads

sparse DNNs where a large percentage of weights are dropped



From: https://arxiv.org/pdf/1911.02497.pdf

Irregular parallelism in loops

- Independent iterations have different amount of work to compute
- Threads with longer tasks take longer to compute.
- Threads with shorter tasks are under utilized.

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

irregular (or unbalanced) parallelism: each x iteration performs different amount of work.

Irregular parallelism in loops

- Calculate imbalance cost if x is chunked:
  - Thread 1 takes iterations 0 - SIZE/2
  - Thread 2 takes iterations SIZE/2 - SIZE

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

Irregular parallelism in loops

- Calculate imbalance cost if x is chunked:
  - Thread 1 takes iterations 0 - SIZE/2
  - Thread 2 takes iterations SIZE/2 - SIZE

Calculate how much total work:

$$\text{total\_work} = \sum_{n=0}^{SIZE} n$$

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

Irregular parallelism in loops

- Calculate imbalance cost if x is chunked:
  - Thread 1 takes iterations 0 - SIZE/2
  - Thread 2 takes iterations SIZE/2 - SIZE

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

Calculate how much total work:

$$\text{total\_work} = \sum_{n=0}^{SIZE} n$$

Calculate work done by first thread:

$$\text{t1\_work} = \sum_{n=0}^{SIZE/2} n$$

Irregular parallelism in loops

- Calculate imbalance cost if x is chunked:
  - Thread 1 takes iterations 0 - SIZE/2
  - Thread 2 takes iterations SIZE/2 - SIZE

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

Calculate how much total work:

$$\text{total\_work} = \sum_{n=0}^{SIZE} n$$

Calculate work done by first thread:

$$\text{t1\_work} = \sum_{n=0}^{SIZE/2} n$$

Calculate work work done by second thread:

t2_work = total_work − t1_work

Irregular parallelism in loops

Example: SIZE = 64

total_work = 2016
t2_work = 496
t1_work = 1520

t1 does ~3x more work than t2

Only provides ~1.3x speedup

Potential solution:
Have T1 do only ¼ of the iterations
Gives a better speedup of 1.77x

Not a feasible solution because often times load imbalance is not given by a static equation on loop bounds!

Calculate how much total work:

$$\text{total\_work} = \sum_{n=0}^{SIZE} n$$

Calculate work done by first thread:

$$\text{t1\_work} = \sum_{n=0}^{SIZE/2} n$$

Calculate work work done by second thread:

t2_work = total_work − t1_work

Work stealing

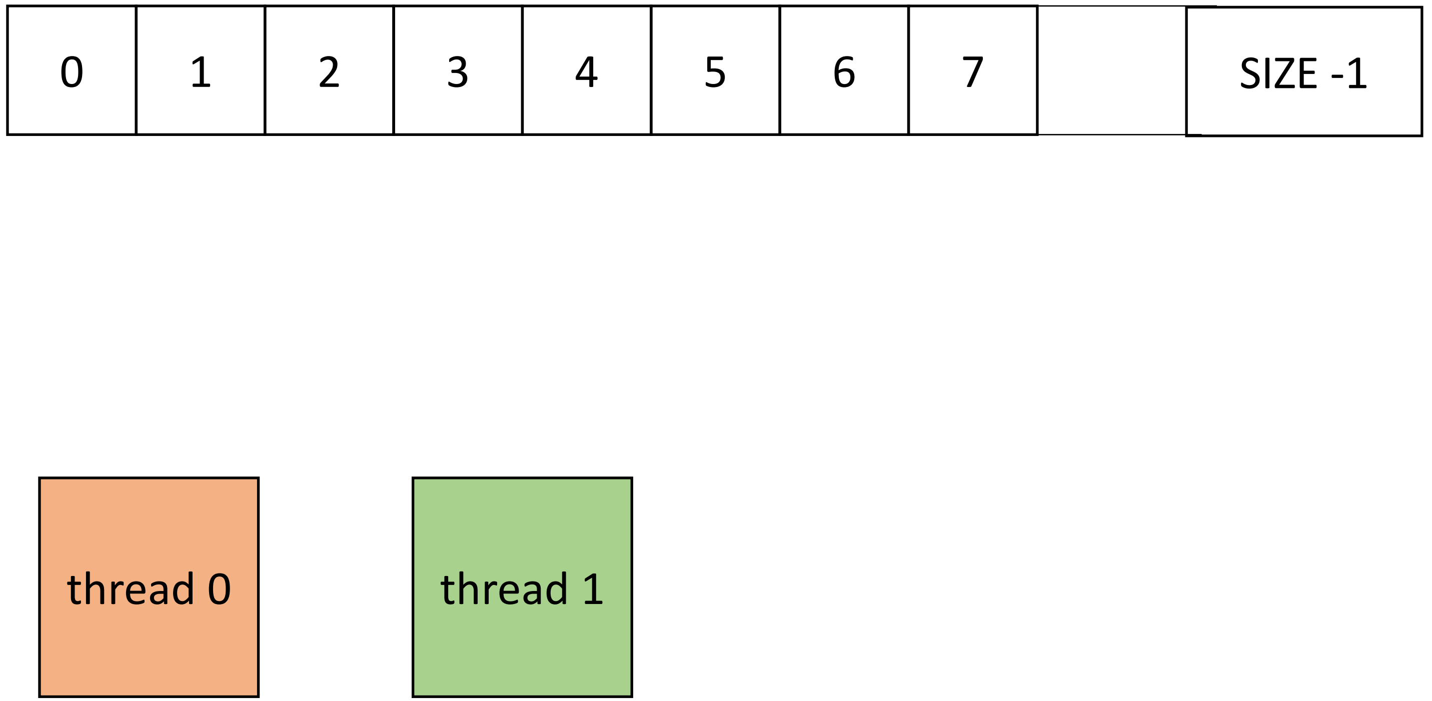
- Tasks are dynamically assigned to threads.

Work stealing - global implicit worklist

- Pros
  - Simple to implement
- Cons:
  - High contention on global counter
  - Potentially bad memory locality.

Work stealing - global implicit worklist

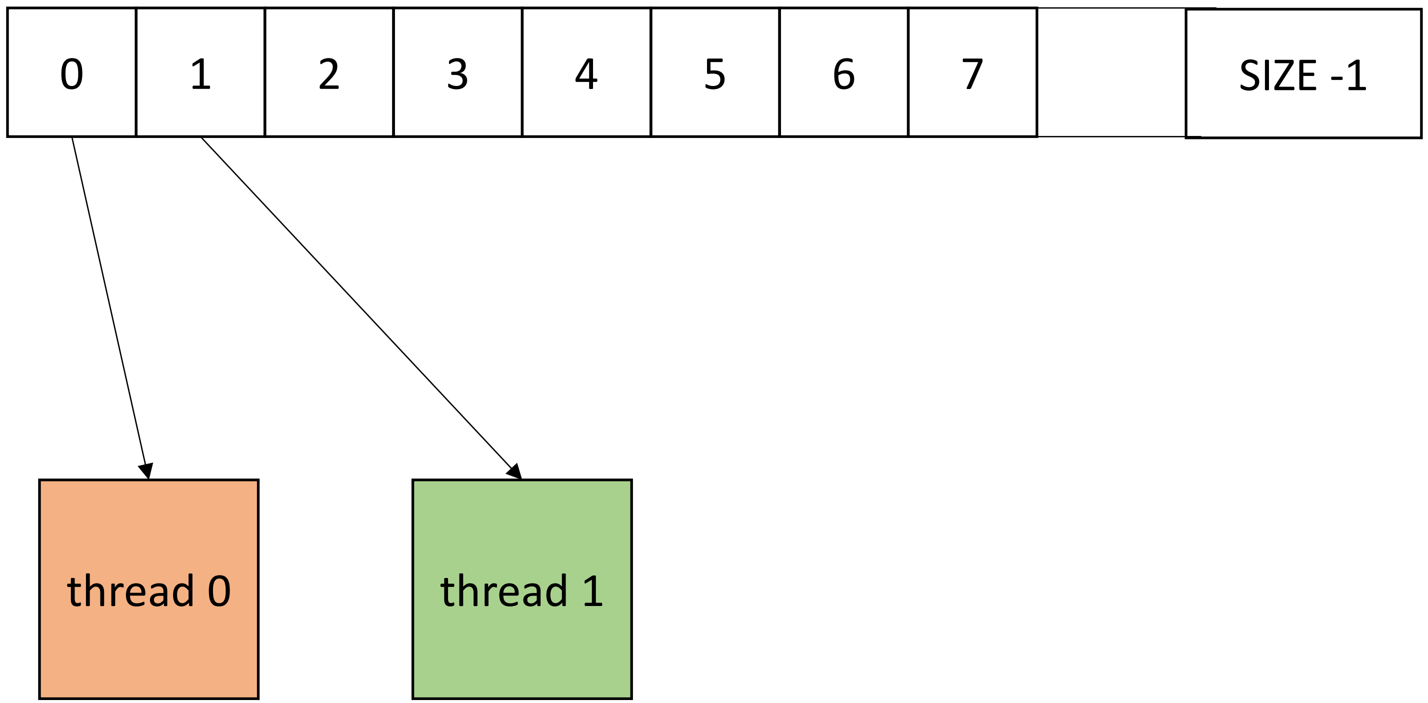
- Global worklist: threads take tasks (iterations) dynamically



cannot color initially!

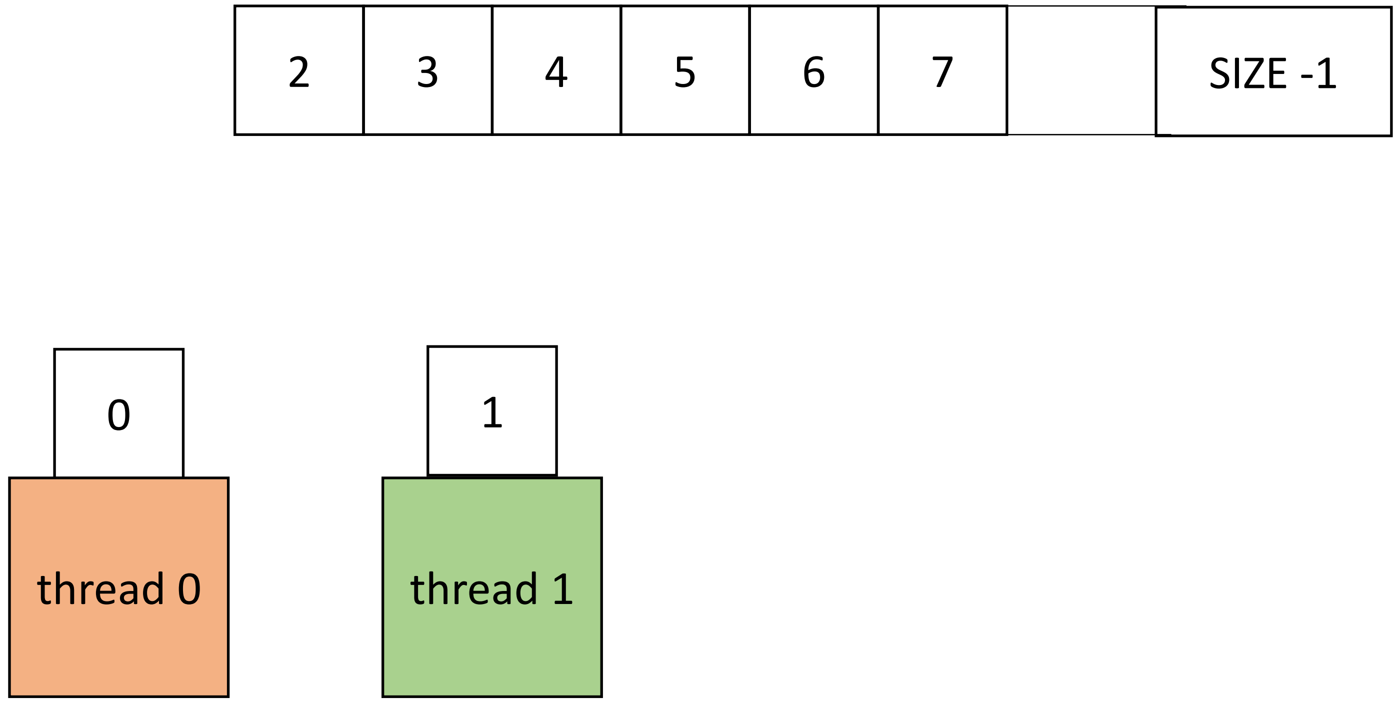
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



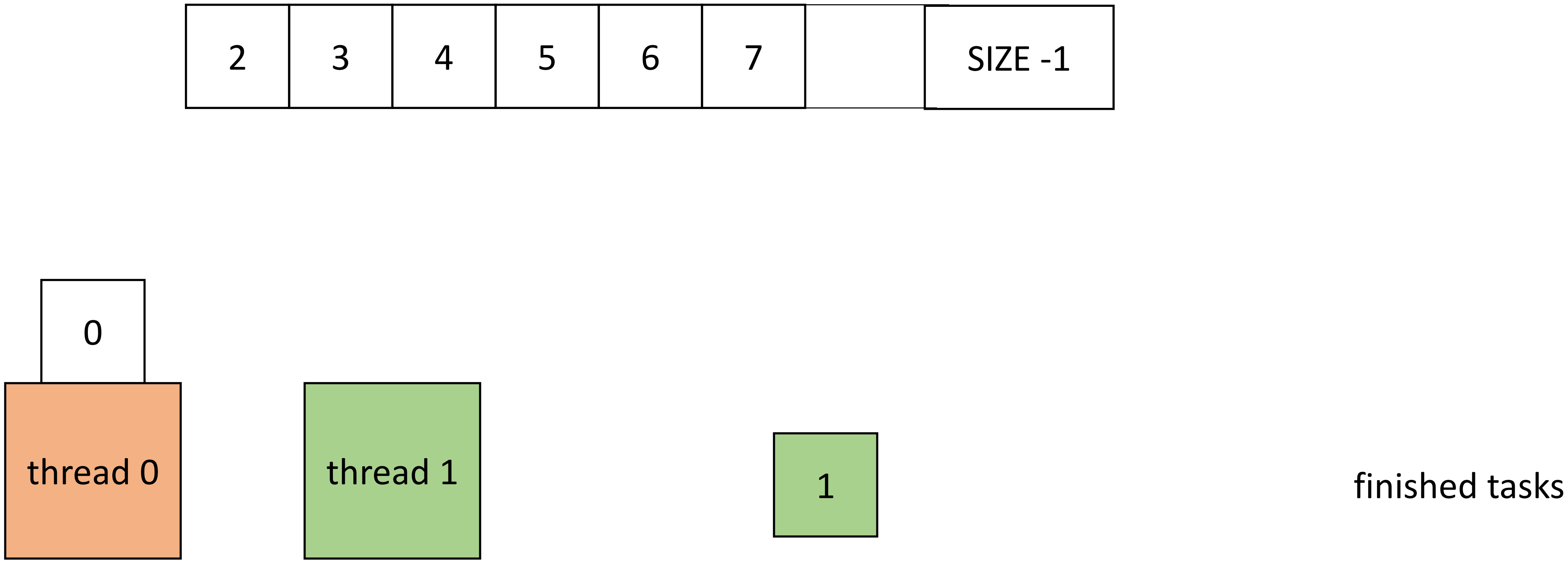
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



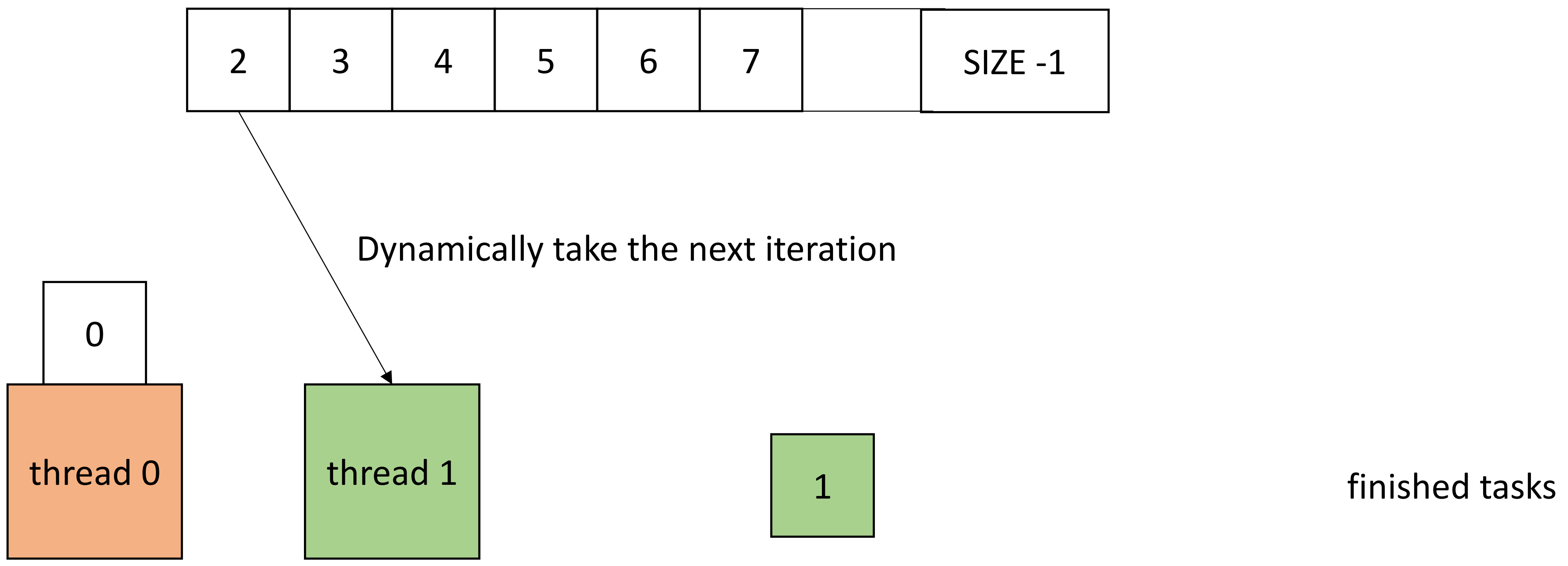
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



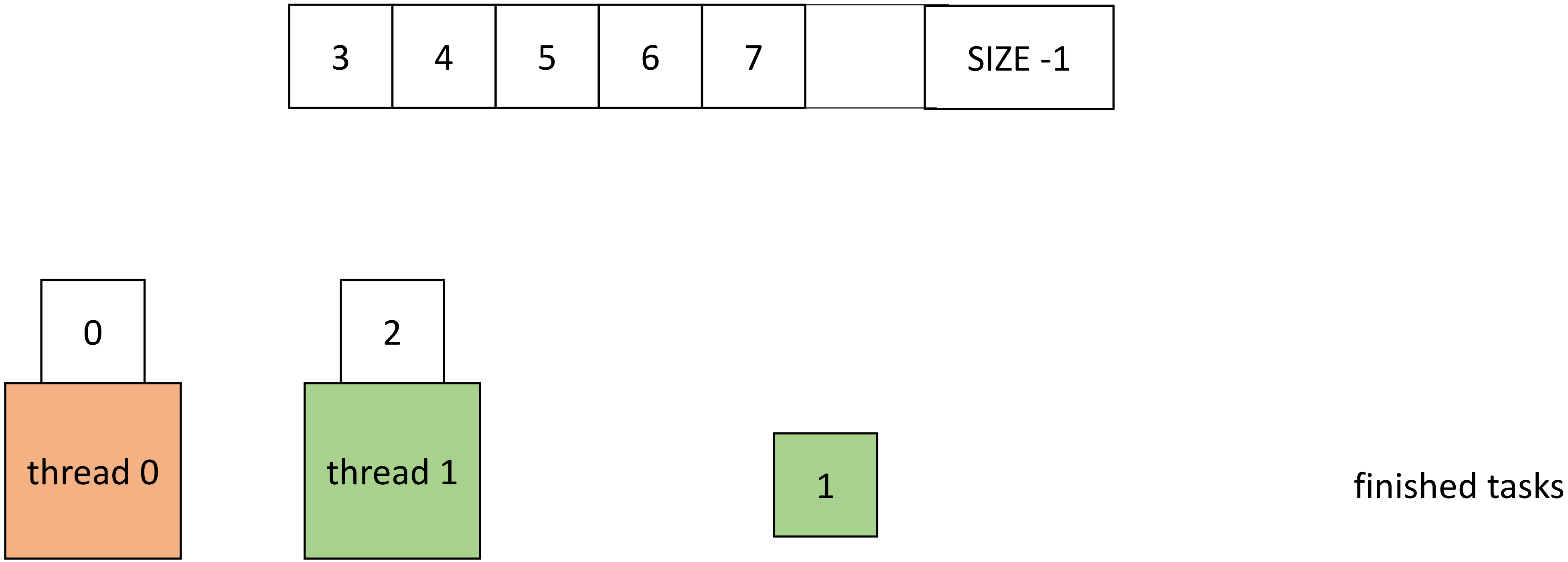
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



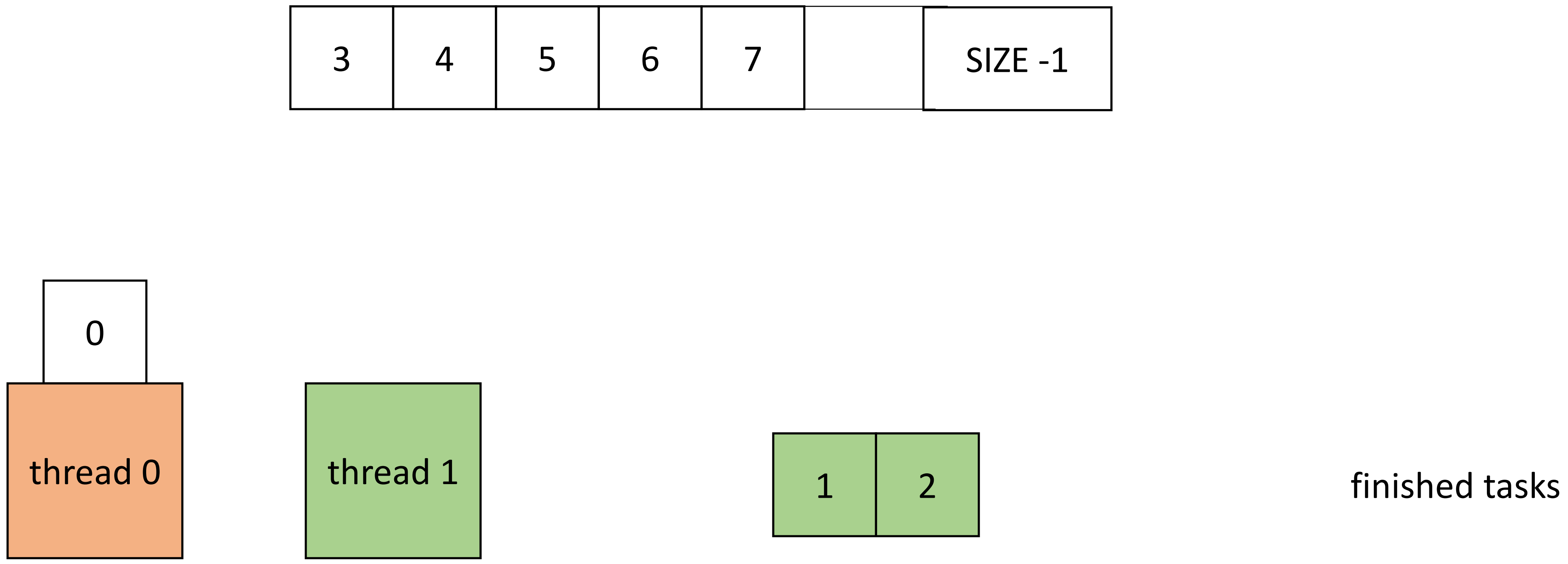
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



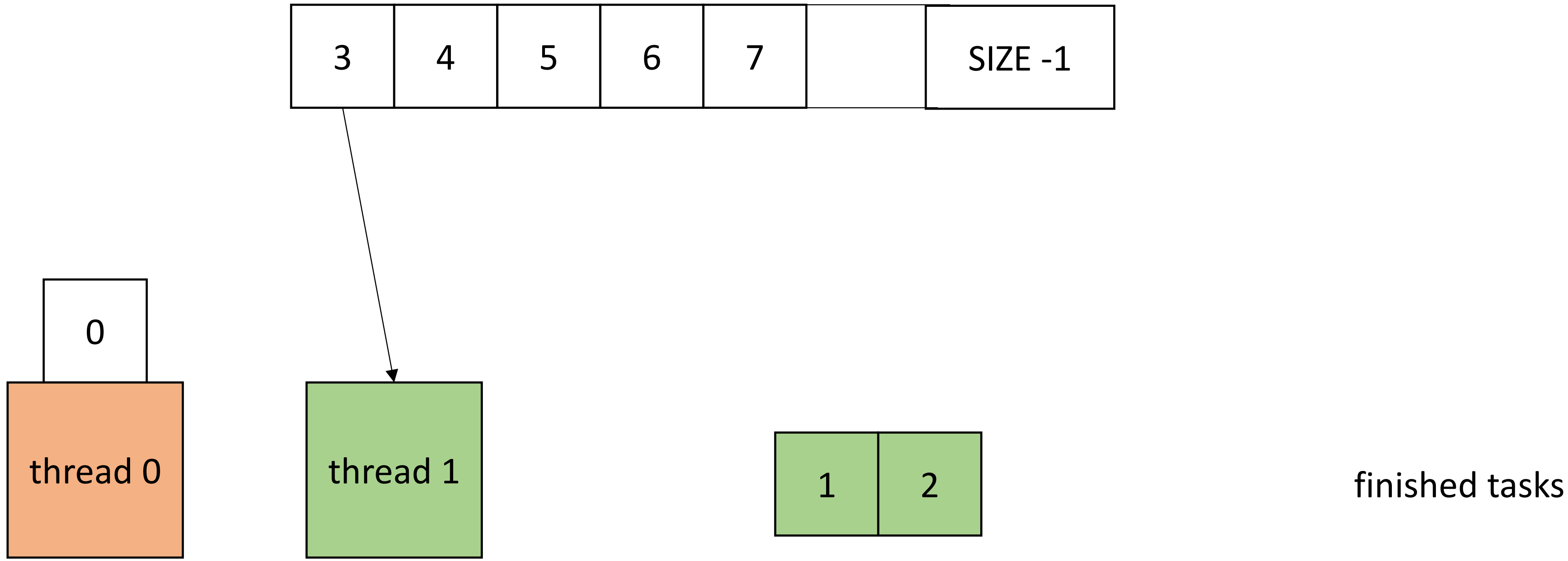
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



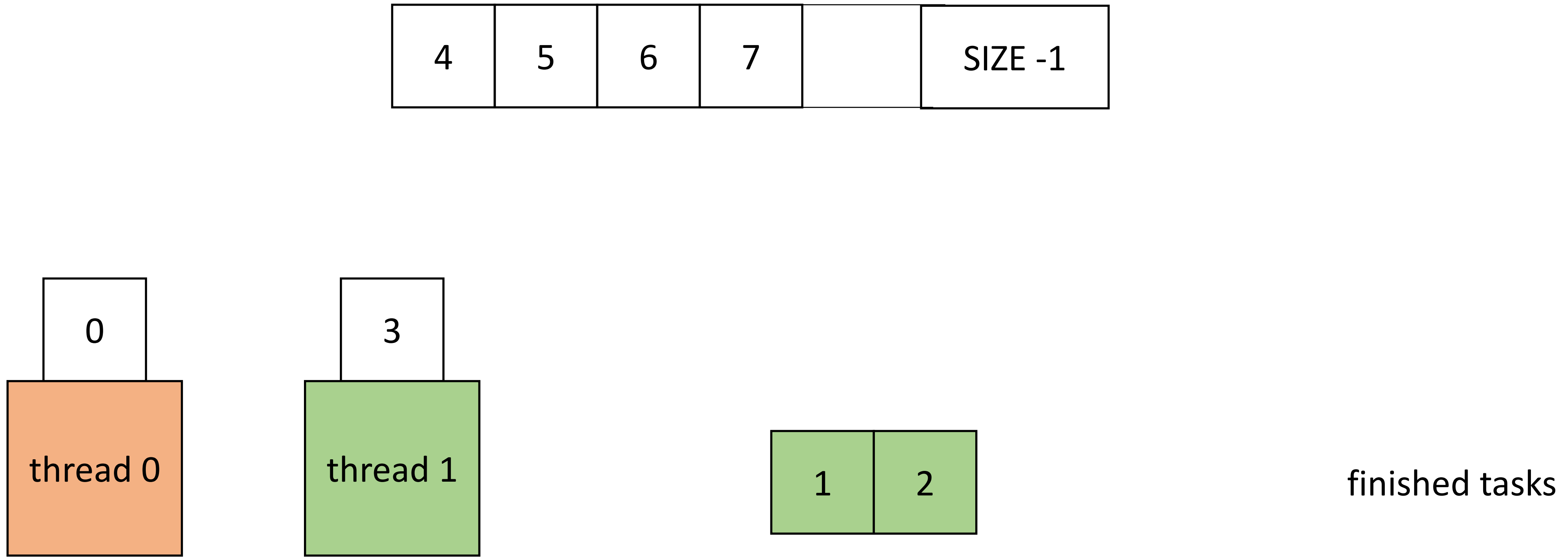
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



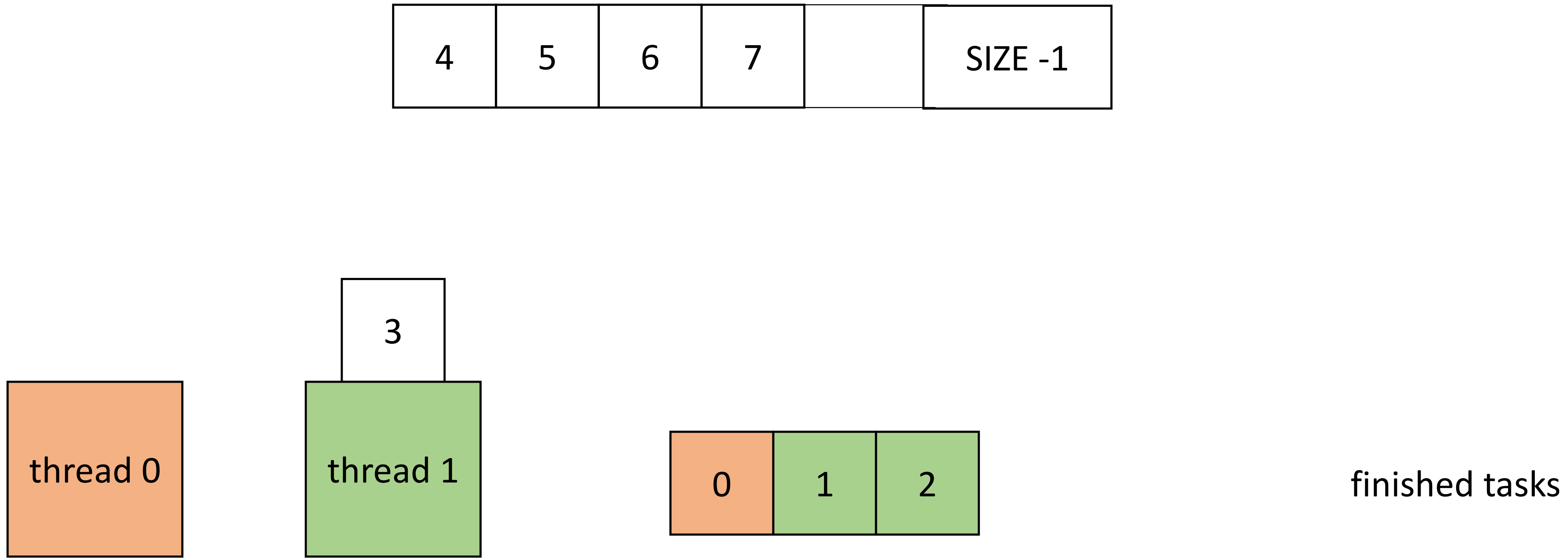
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



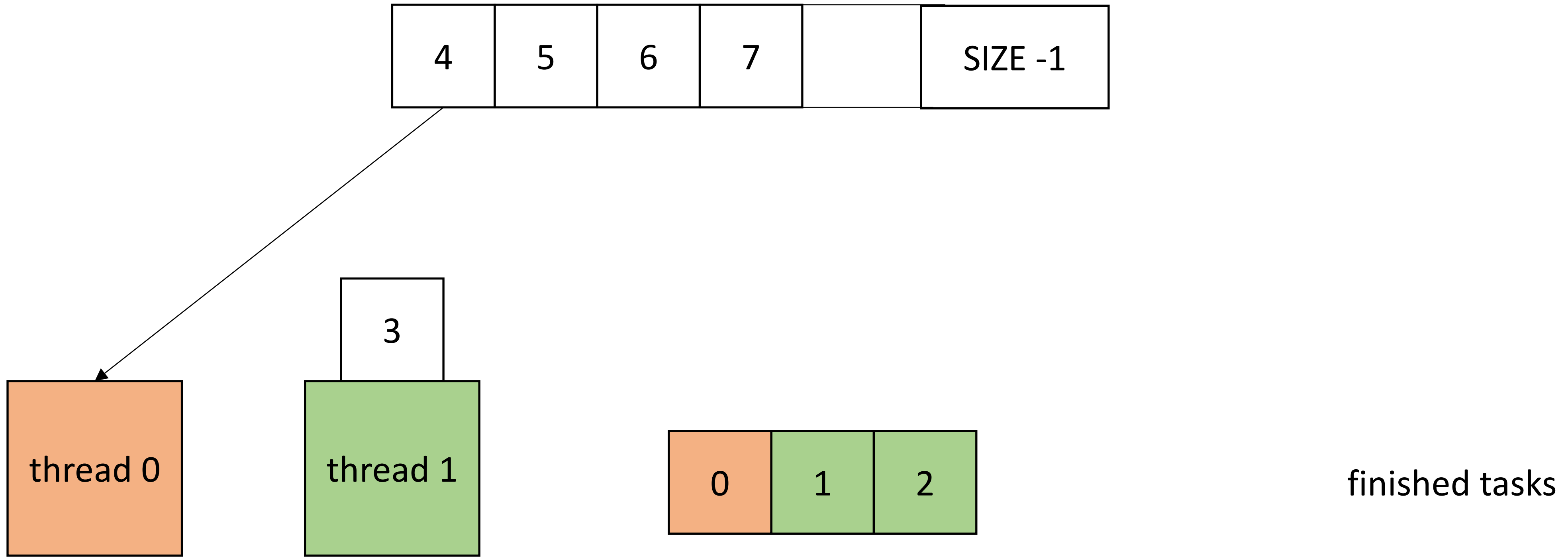
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



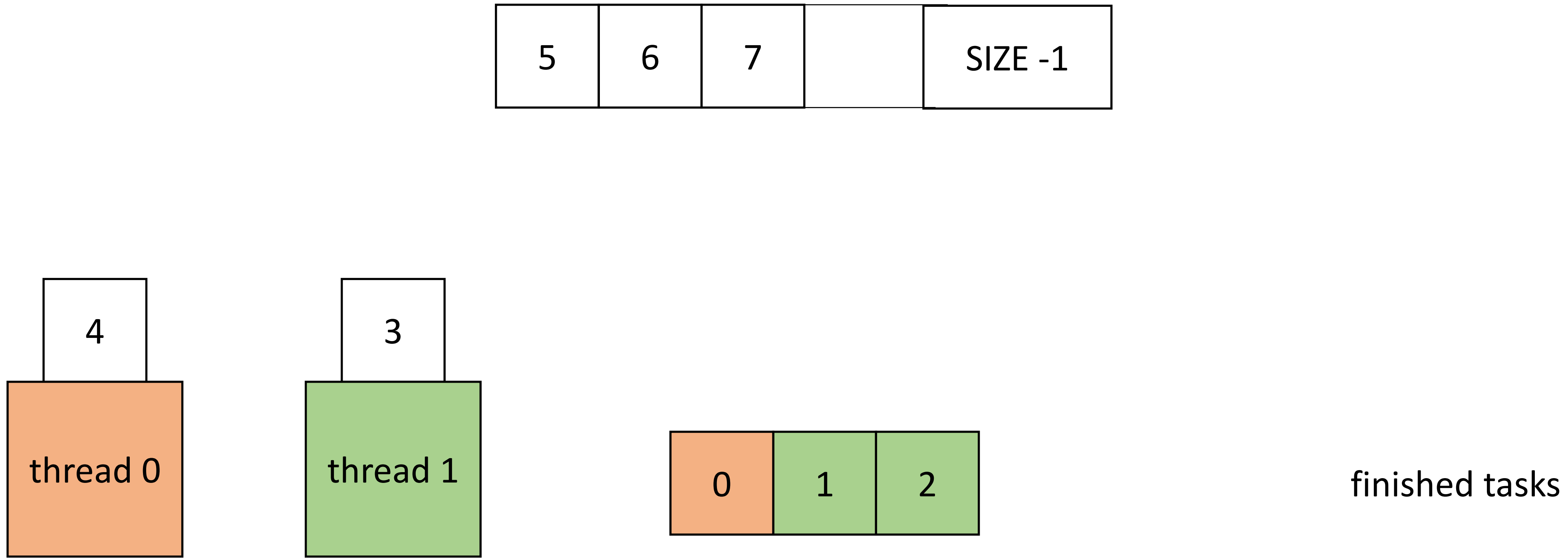
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



End example

Work stealing - global implicit worklist

- How to implement

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

Work stealing - global implicit worklist

- How to implement

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

```
void parallel_loop(...) {

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

Replicate code in a new function. Pass all needed variables as arguments.

Work stealing - global implicit worklist

- How to implement

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

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

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

move loop variable to be a global atomic variable

Work stealing - global implicit worklist

- How to implement

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

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

  for (int local_x = ??
           local_x < SIZE;
           local_x = ??) {

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

change loop bounds in new function to use a local variable using global variable.

Work stealing - global implicit worklist

- How to implement

These must be atomic updates!

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

```
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
  }
}
```

change loop bounds in new function to use a local variable using global variable.

Work stealing - global implicit worklist

- How to implement

```
void foo() {
  ...
  for (t = 0; x < THREADS; t++) {
    spawn(parallel_loop);
  }
  join();
  ...
}
```

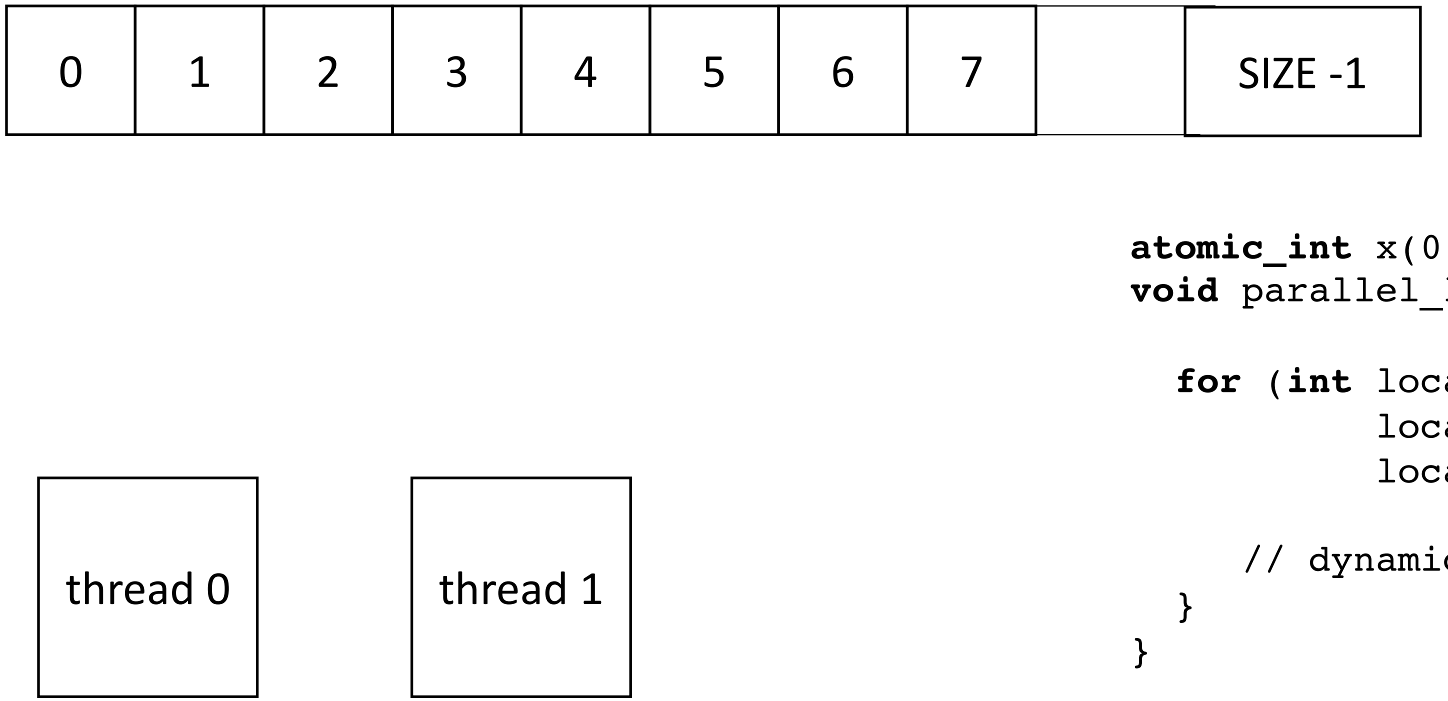
```
atomic_int x(0);
void parallel_loop(...) {

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

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

Spawn threads in original function and join them afterwards

You will have to change to C++ syntax for the homework!

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

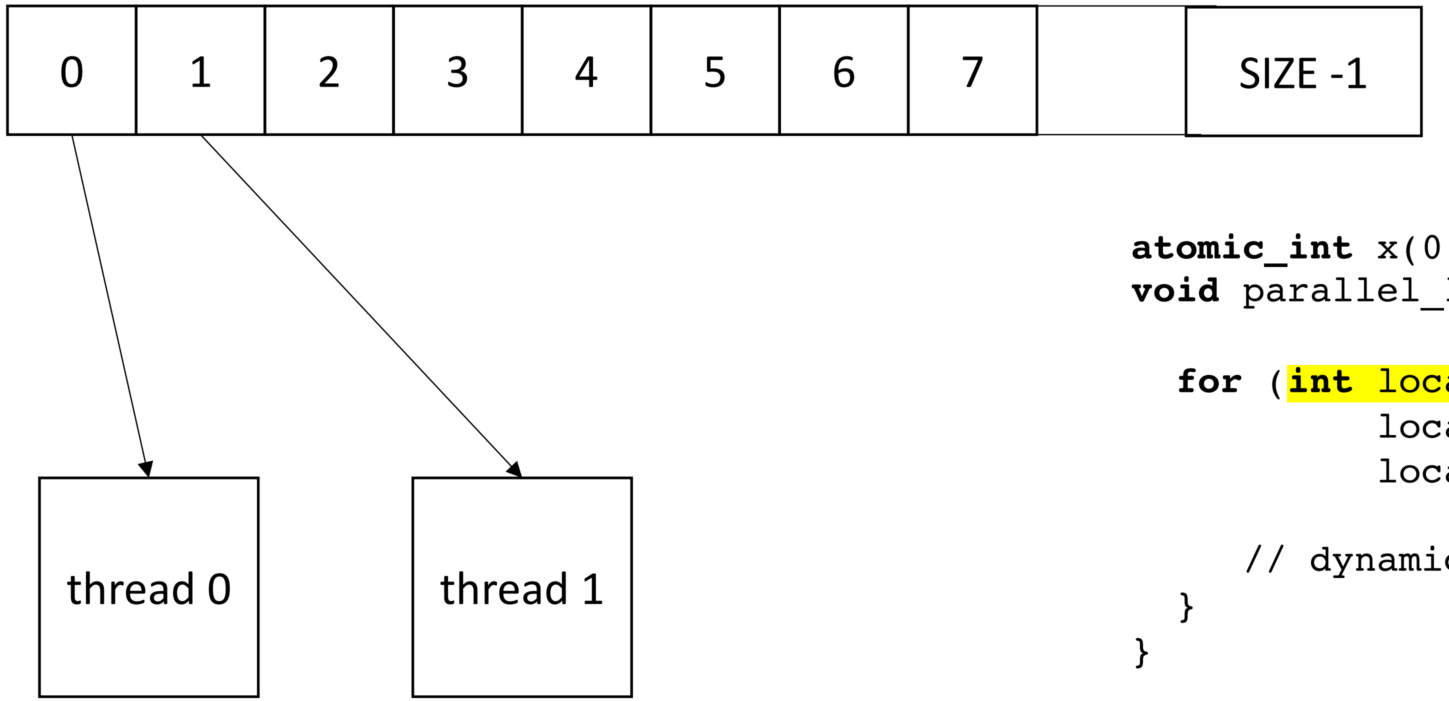



```
x: 0
0 - local_x - UNDEF
1 - local_x - UNDEF
```

```
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
  }
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

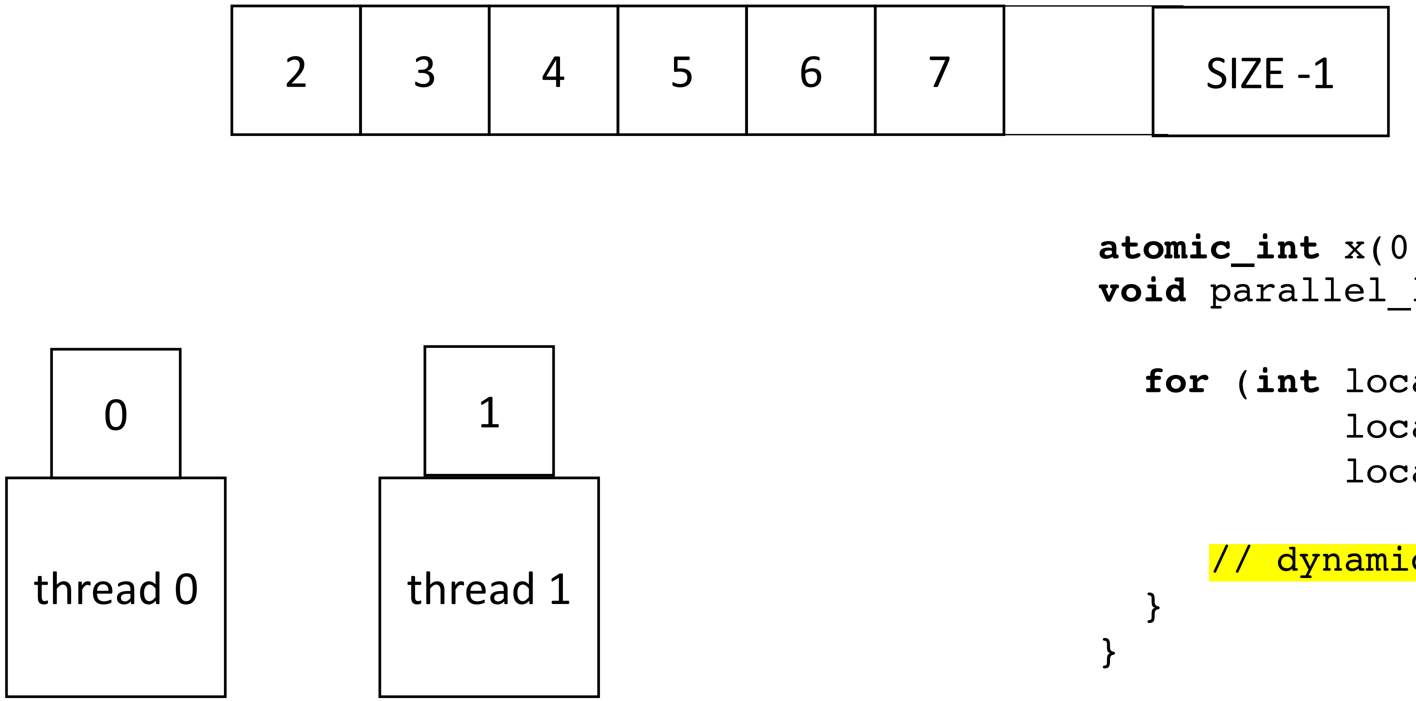



```
x: 2
0 – local_x – 0
1 – local_x – 1
```

```
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
  }
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

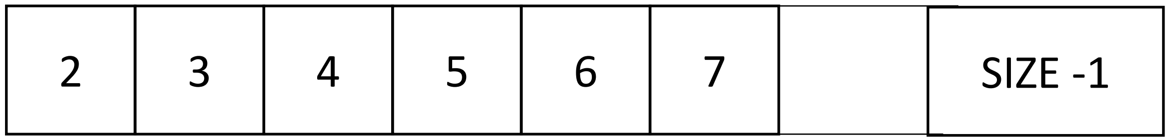



```
x: 2
0 – local_x – 0
1 – local_x – 1
```

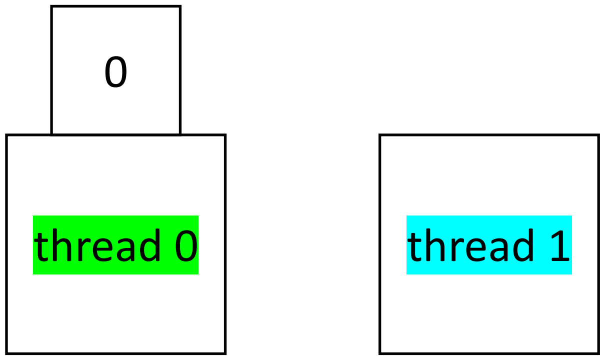
```
atomic_int x(0);
void parallel_loop(...) {

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

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



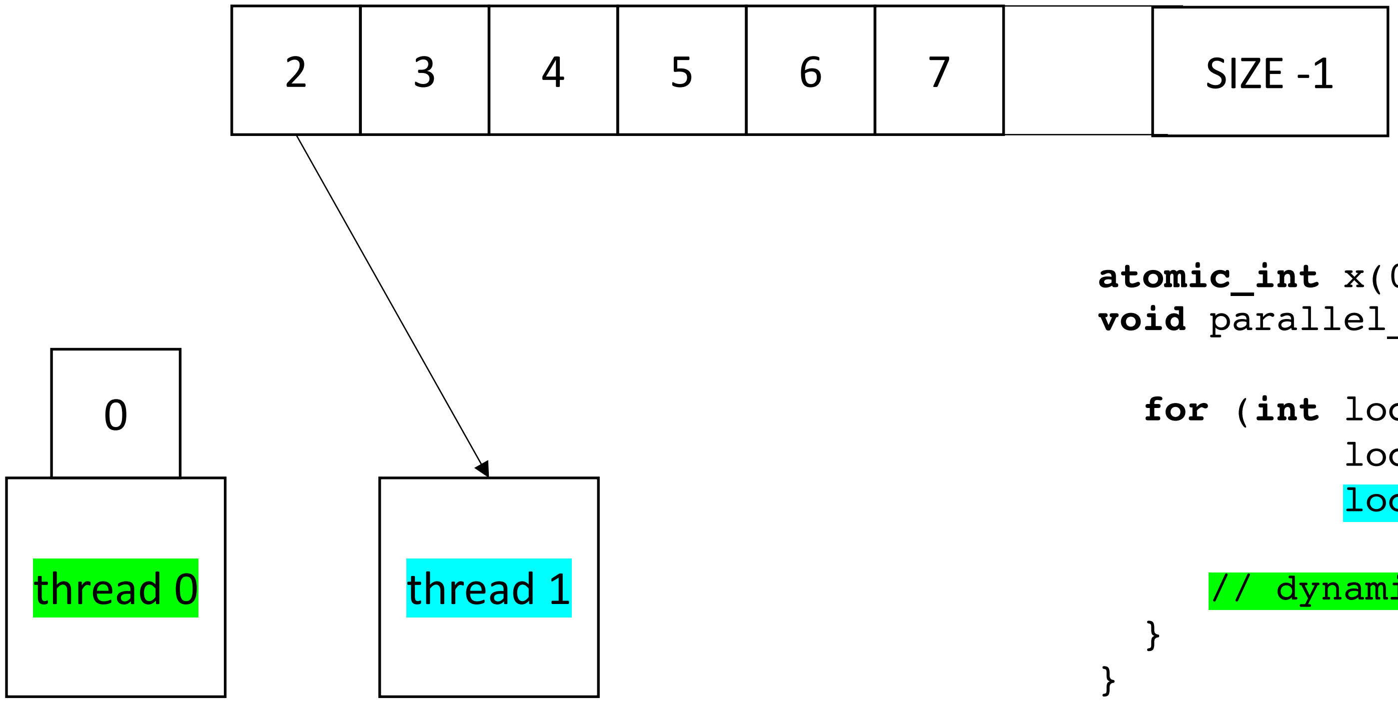

```
x: 2
0 – local_x – 0
1 – local_x – 1
```




```
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
  }
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

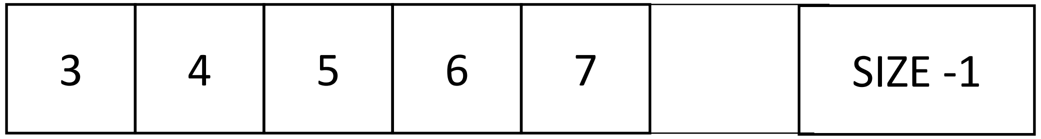



```
x: 3
0 – local_x – 0
1 – local_x – 2
```

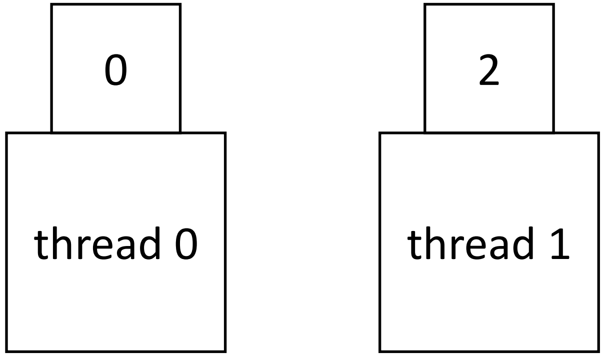
```
atomic_int x(0);
void parallel_loop(...) {

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

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



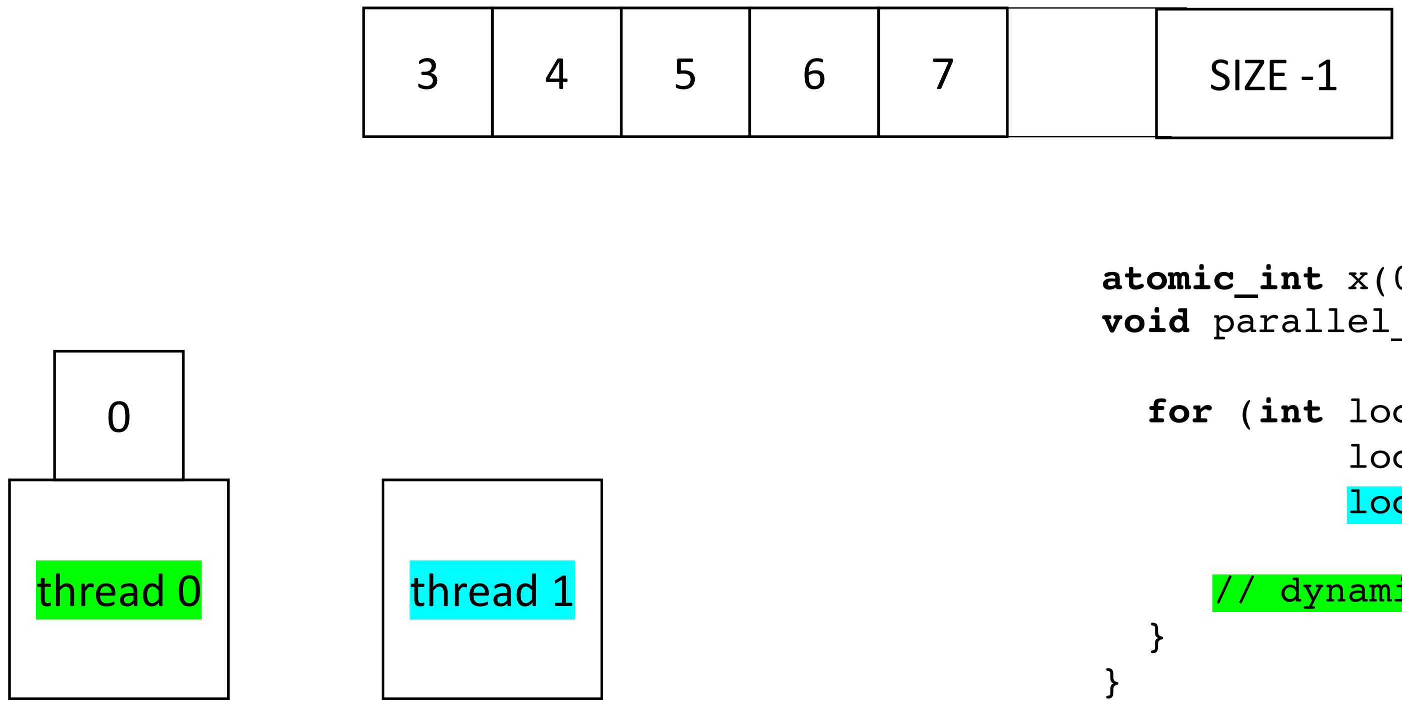

```
x: 3
0 – local_x – 0
1 – local_x – 2
```




```
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
  }
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

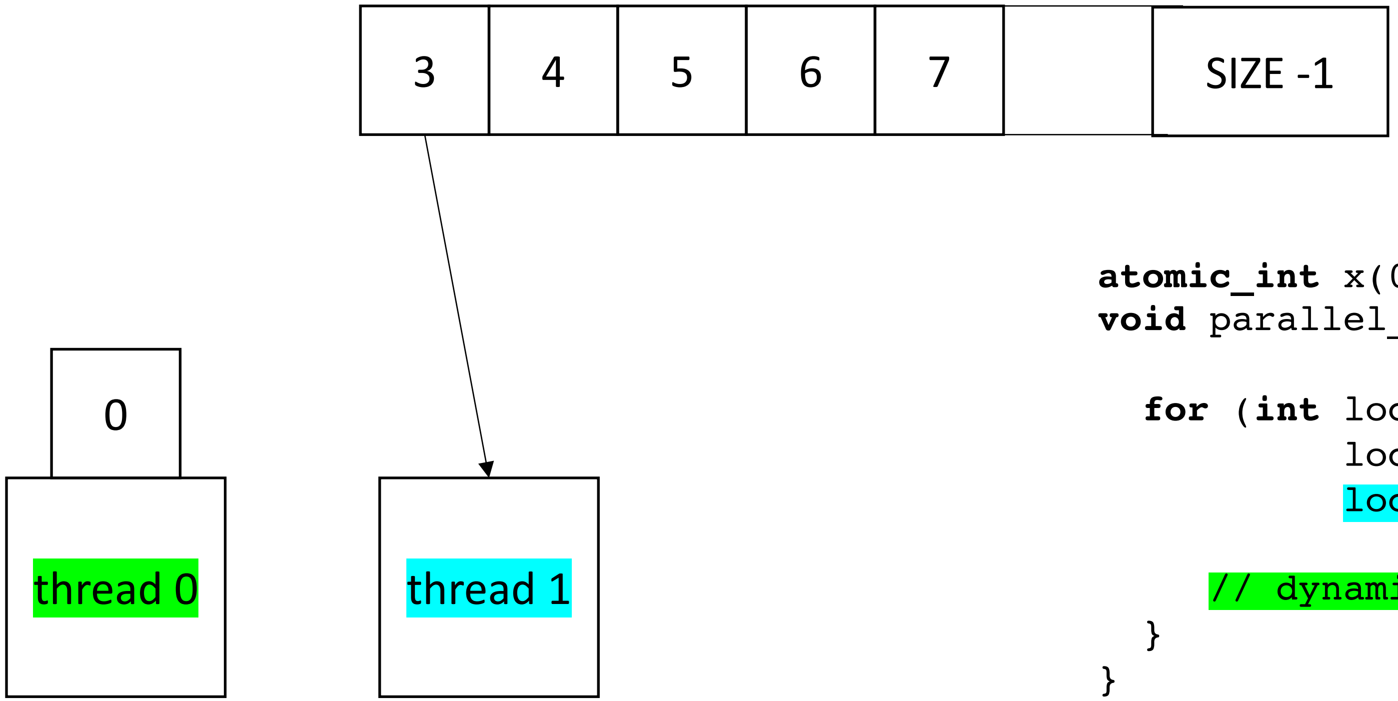



```
x: 3
0 – local_x – 0
1 – local_x – 2
```

```
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
  }
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

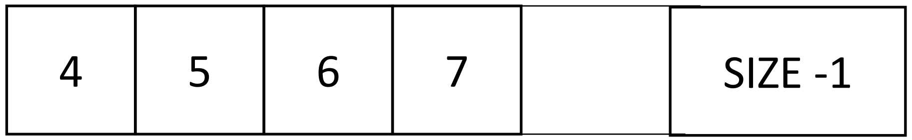



```
x: 4
0 – local_x – 0
1 – local_x – 3
```

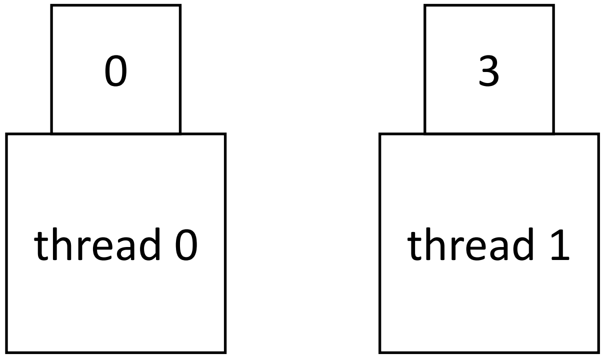
```
atomic_int x(0);
void parallel_loop(...) {

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

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



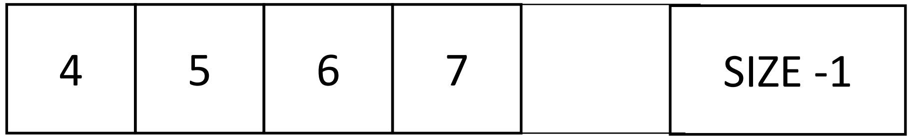

```
x: 4
0 – local_x – 0
1 – local_x – 3
```



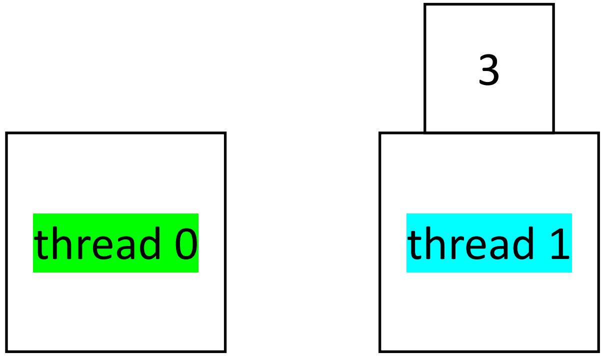

```
atomic_int x(0);
void parallel_loop(...) {

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

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



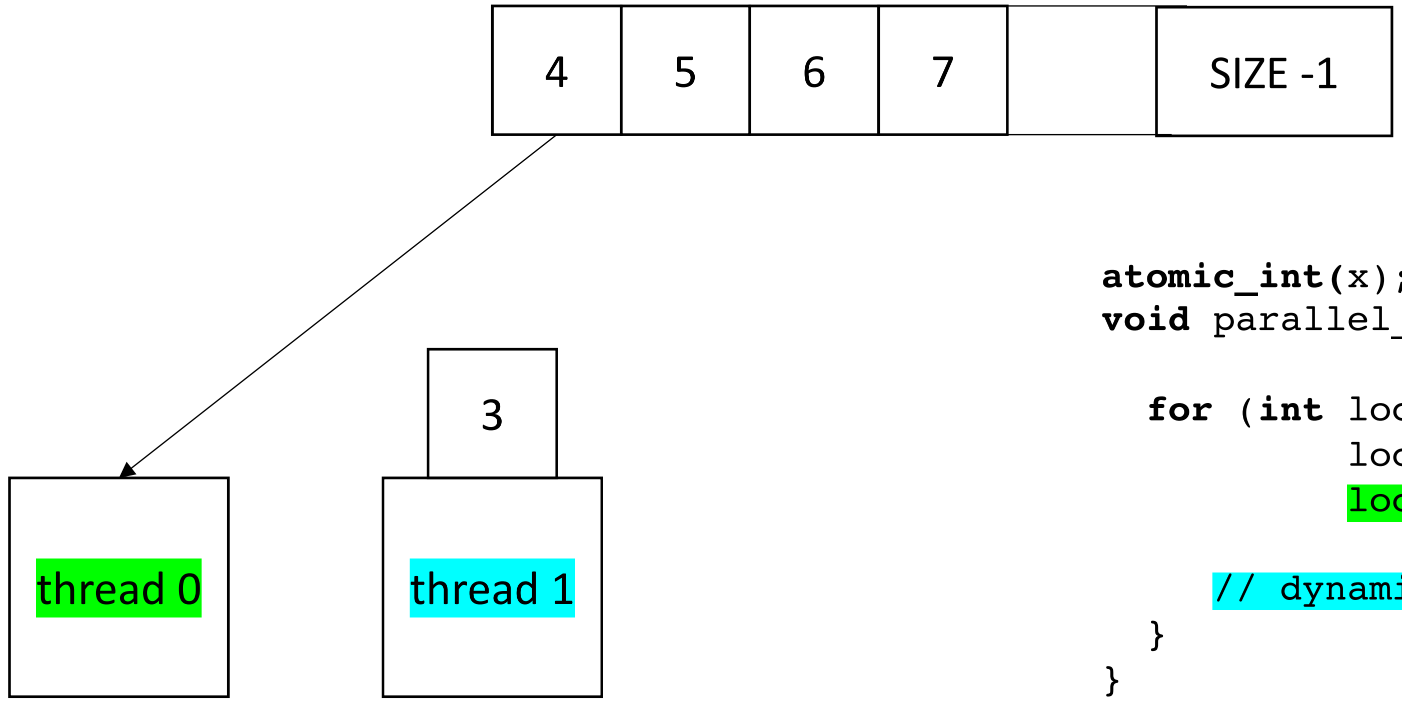

```
x: 4
0 – local_x – 0
1 – local_x – 3
```




```
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
  }
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

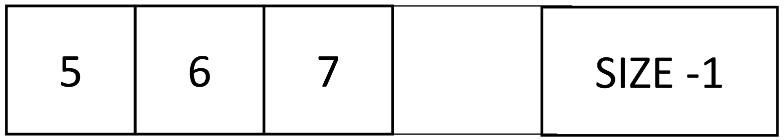



```
x: 5
0 – local_x – 4
1 – local_x – 3
```

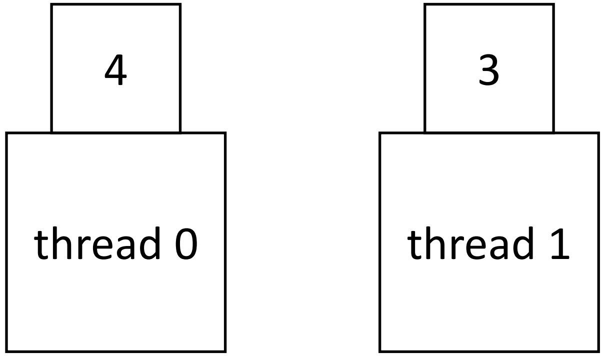
```
atomic_int(x);
void parallel_loop(...) {

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

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

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

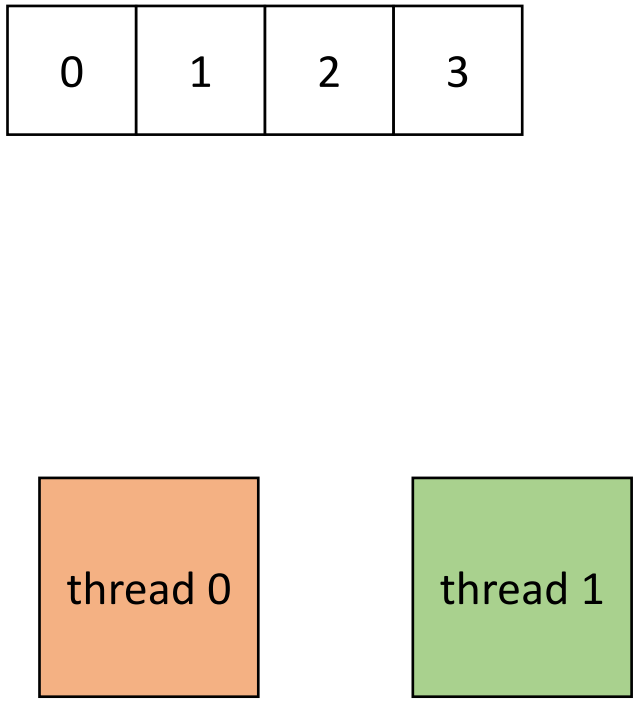
- **Parallel Schedules:**
  - Static
  - Global Worklists
  - **Local Worklists**

Work stealing - local worklists

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

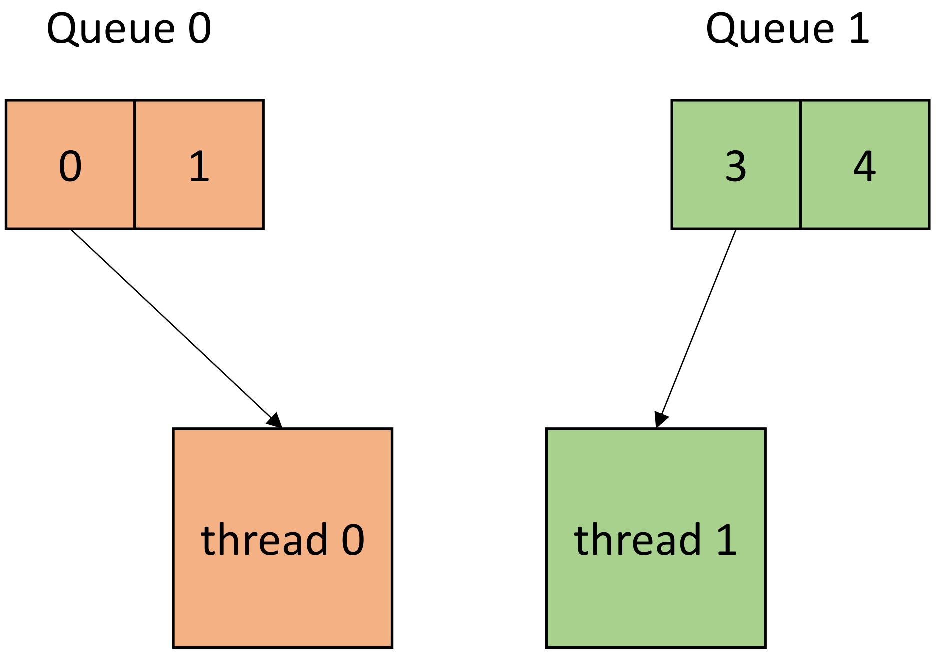
Work stealing - local worklists

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



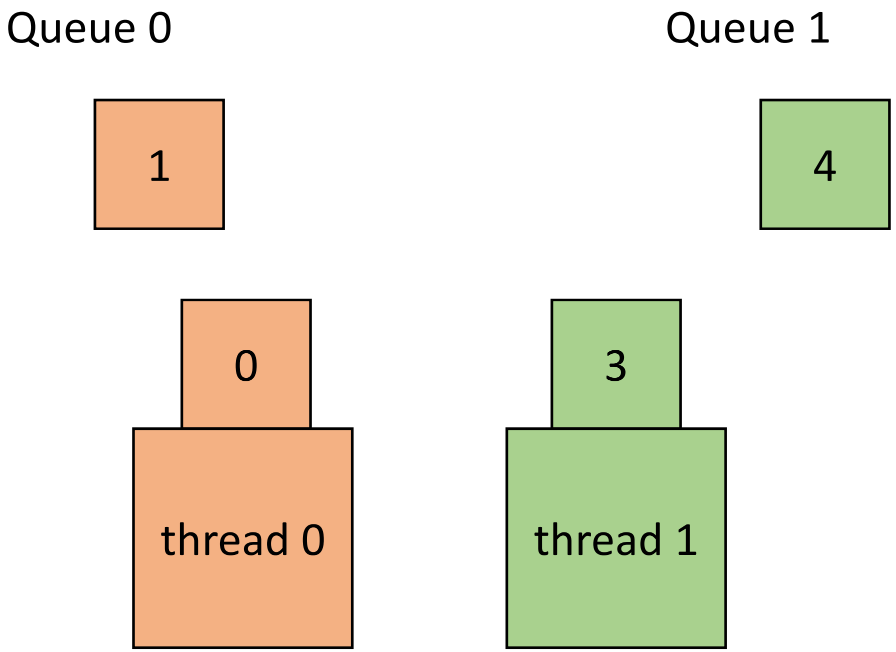
Work stealing - local worklists

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



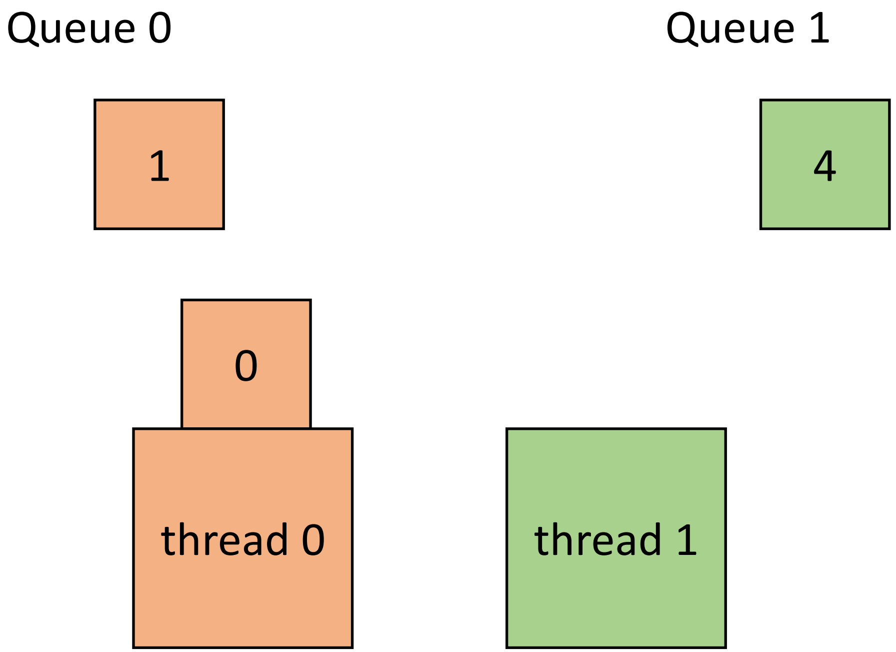
Work stealing - local worklists

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



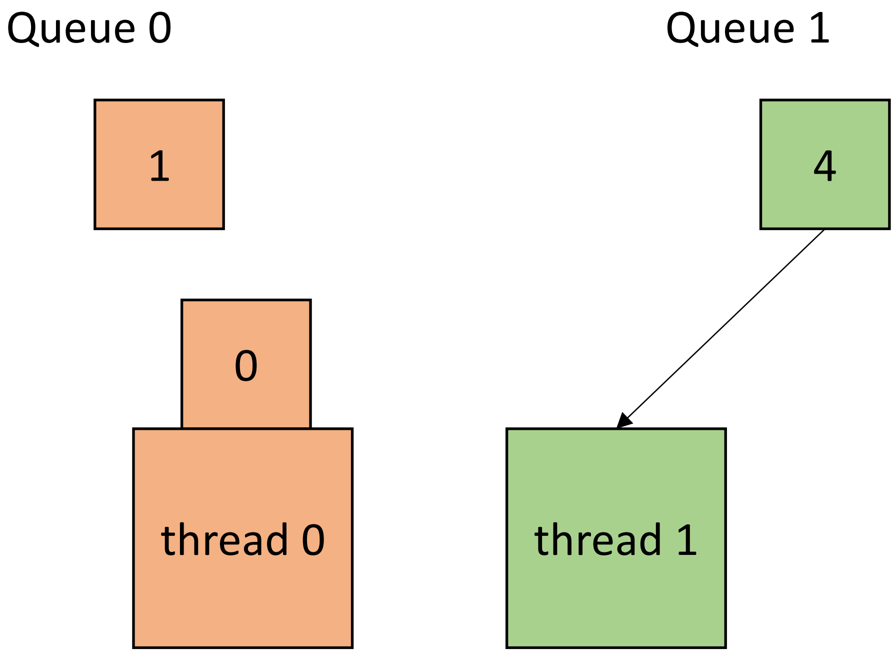
Work stealing - local worklists

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



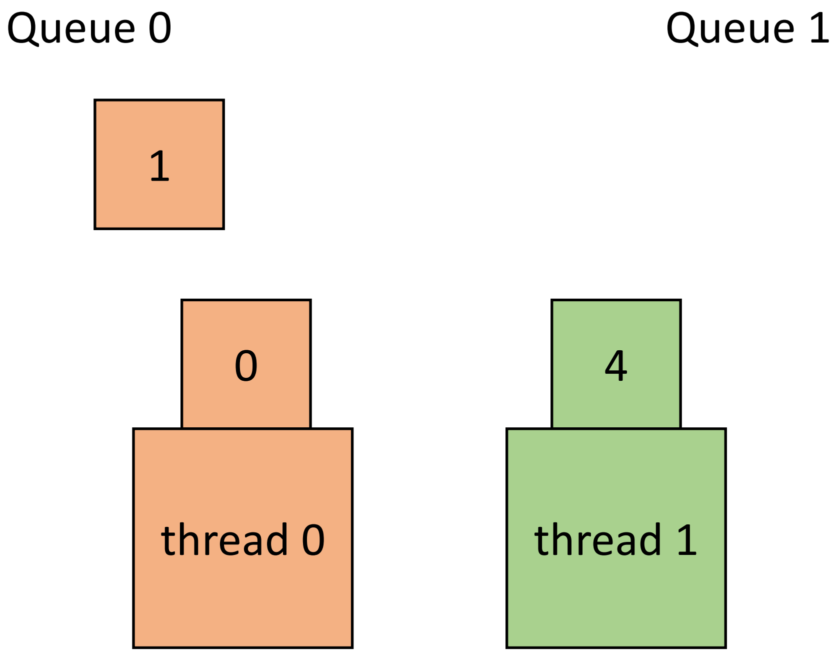
Work stealing - local worklists

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



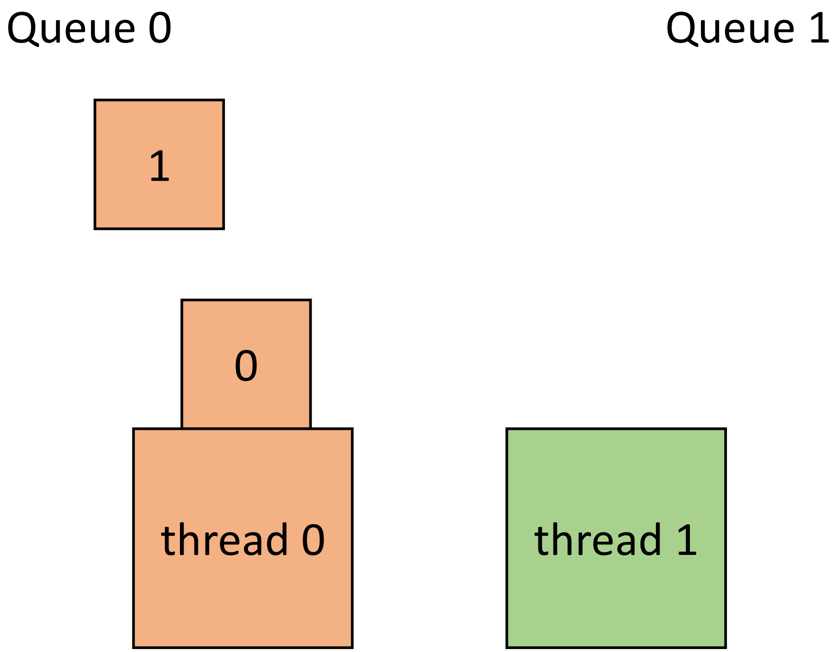
Work stealing - local worklists

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



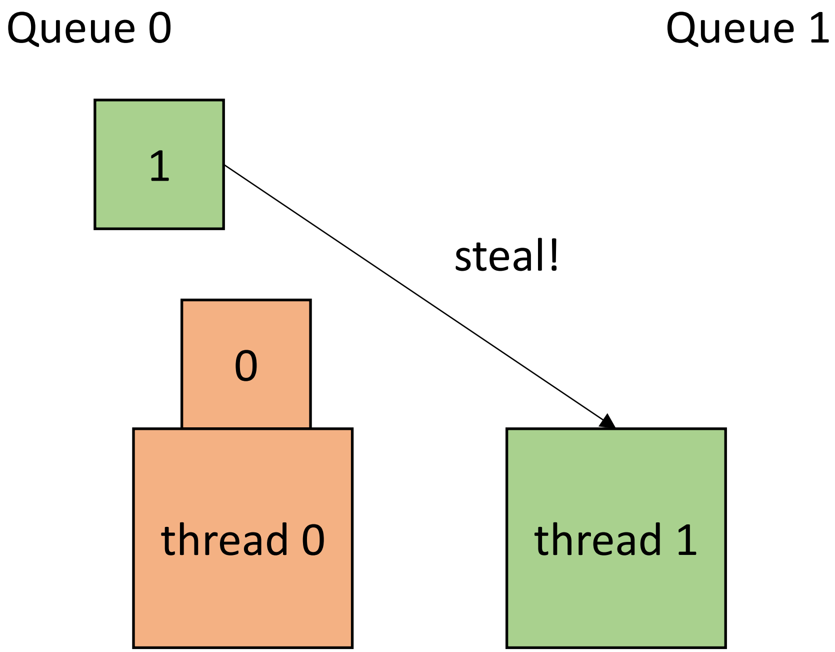
Work stealing - local worklists

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



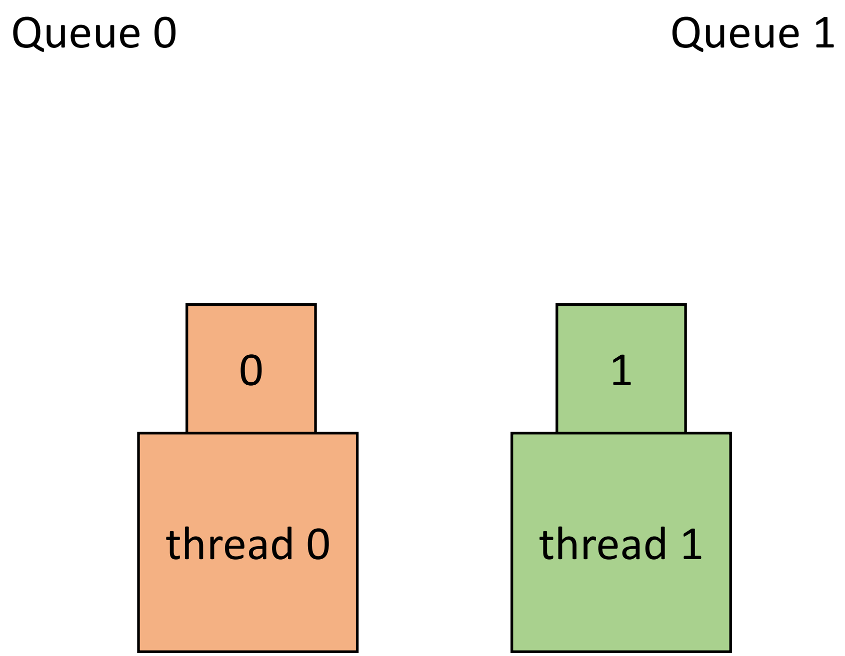
Work stealing - local worklists

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



Work stealing - local worklists

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



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
  }
  ...
}
```

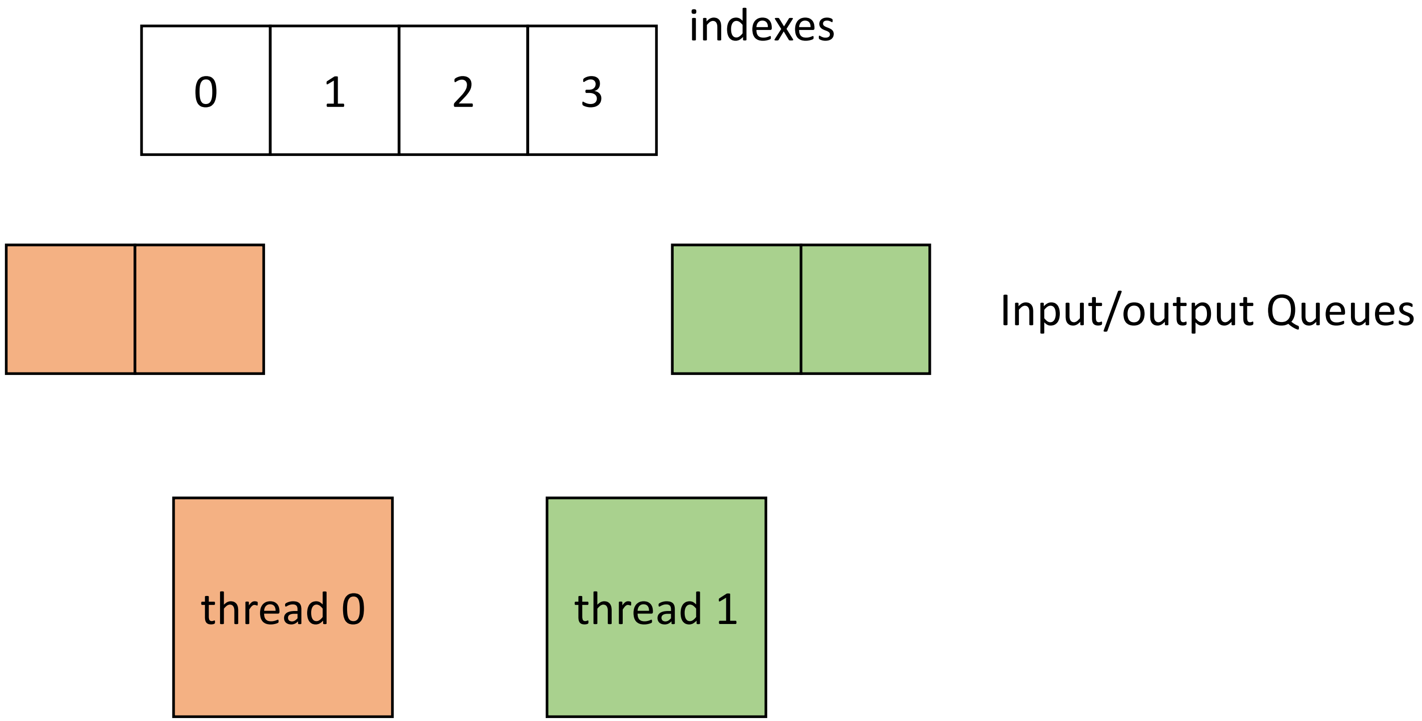
```
void parallel_loop(..., int tid) {

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

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

Make a global array of concurrent queues

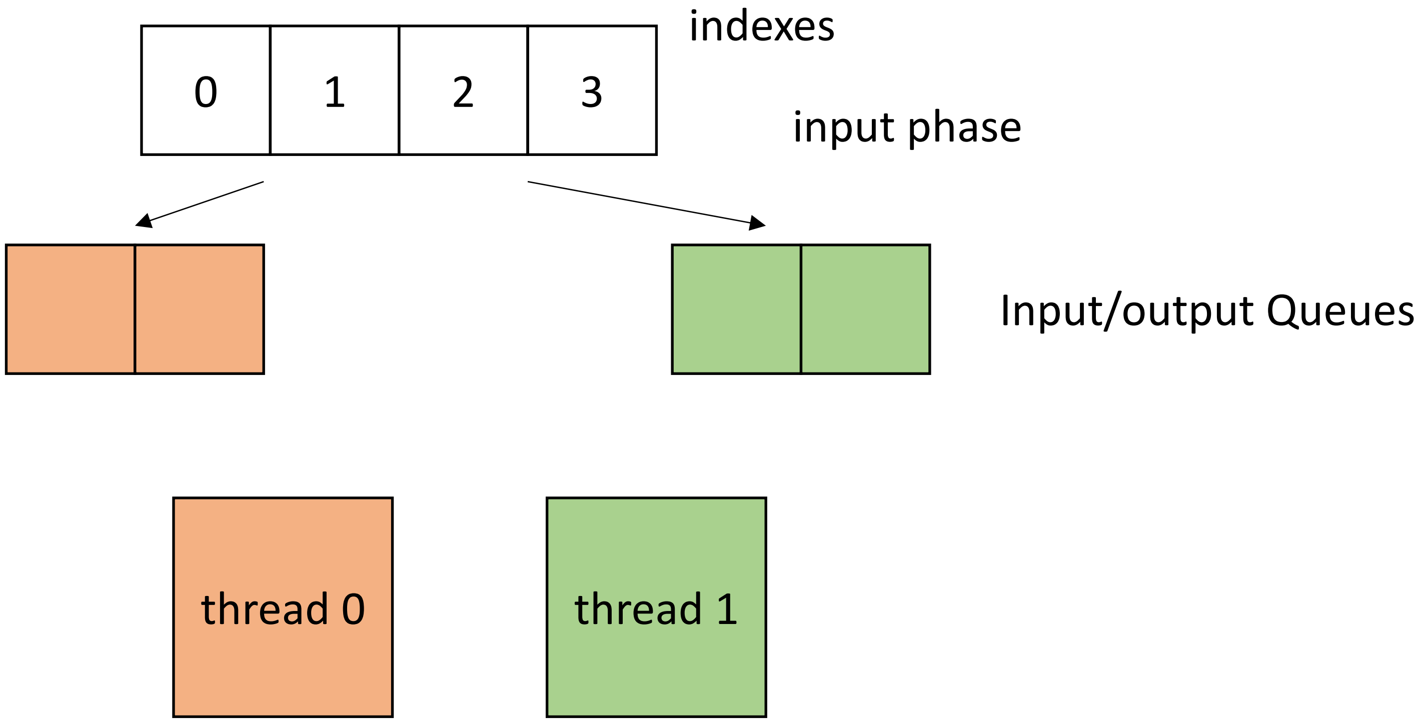
Work stealing - local worklists

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



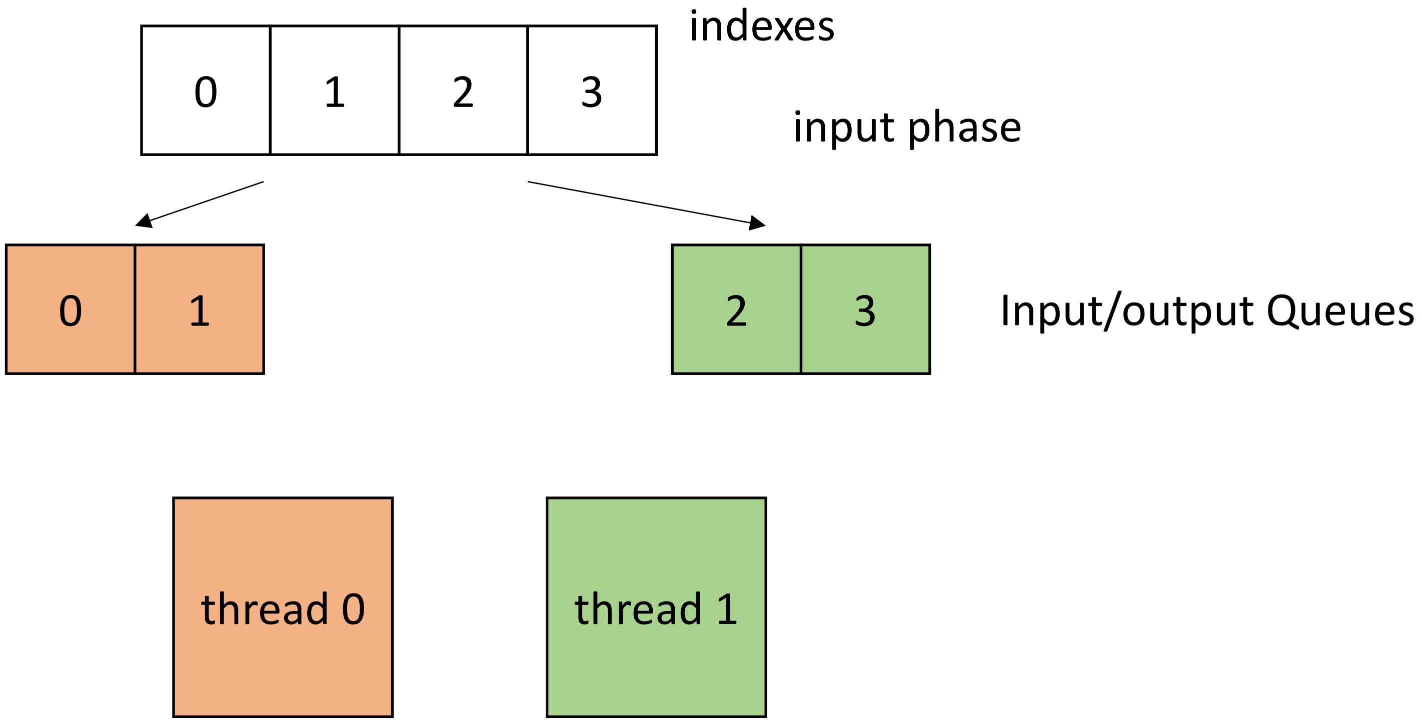
Work stealing - local worklists

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



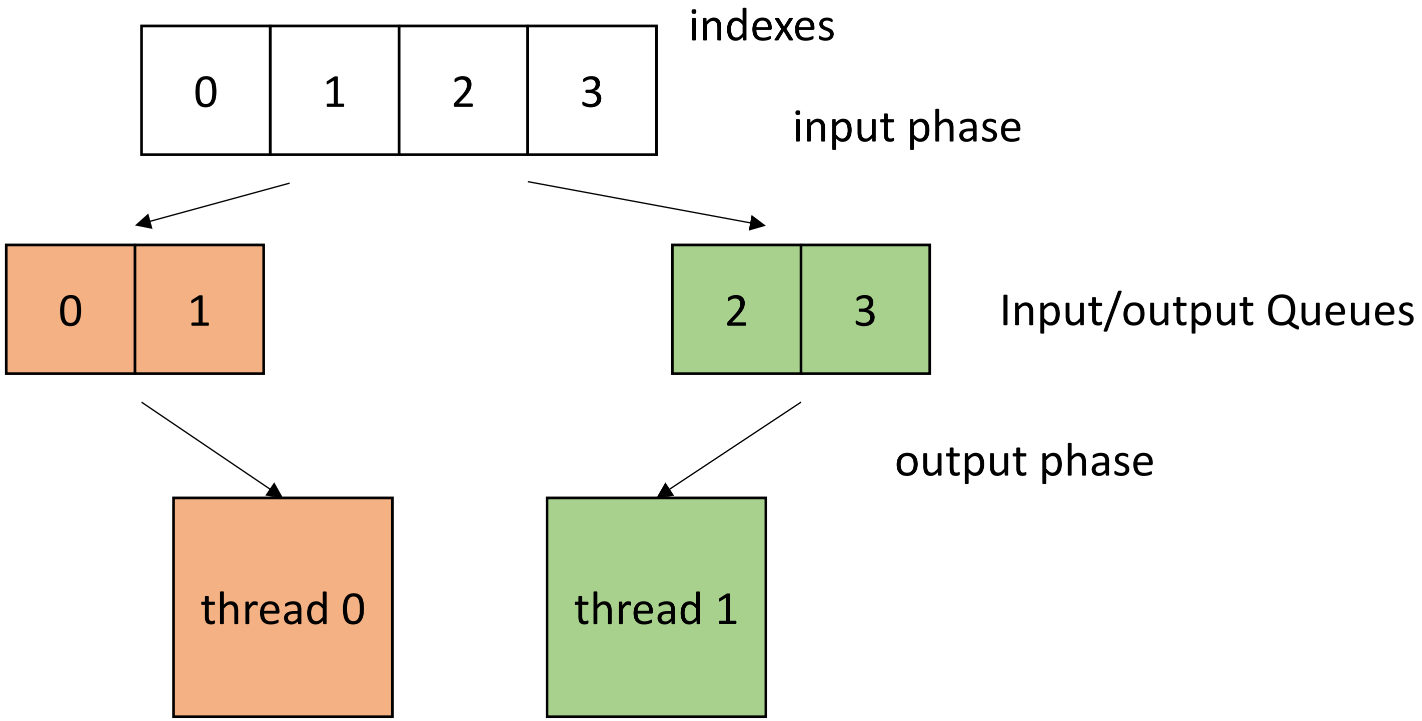
Work stealing - local worklists

- 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

- How to implement in a compiler:

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


  ...
}
```

First we need to initialize the queues

Work stealing - local worklists

- How to implement in a compiler:

```
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

- How to implement in a compiler:

```
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

- How to implement in a compiler:

```
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

- How to implement in a compiler:

```
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

- How to implement in a compiler:

```
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

- How to implement in a compiler:

```
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

- How to implement in a compiler:

```
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

- How to implement in a compiler:

```
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

- How to implement in a compiler:

```
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

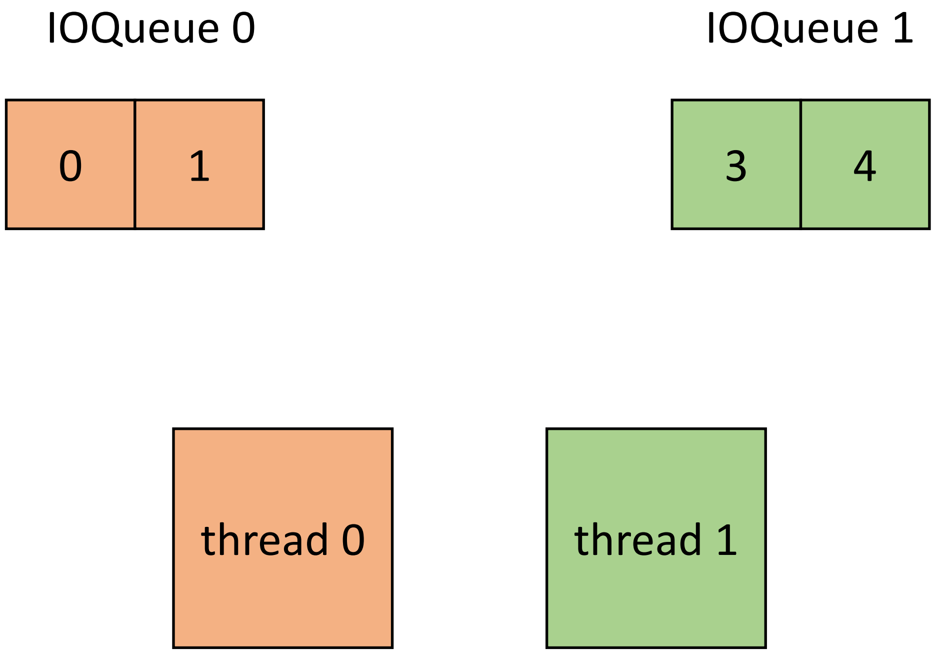
- How to implement in a compiler:

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

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

join the threads

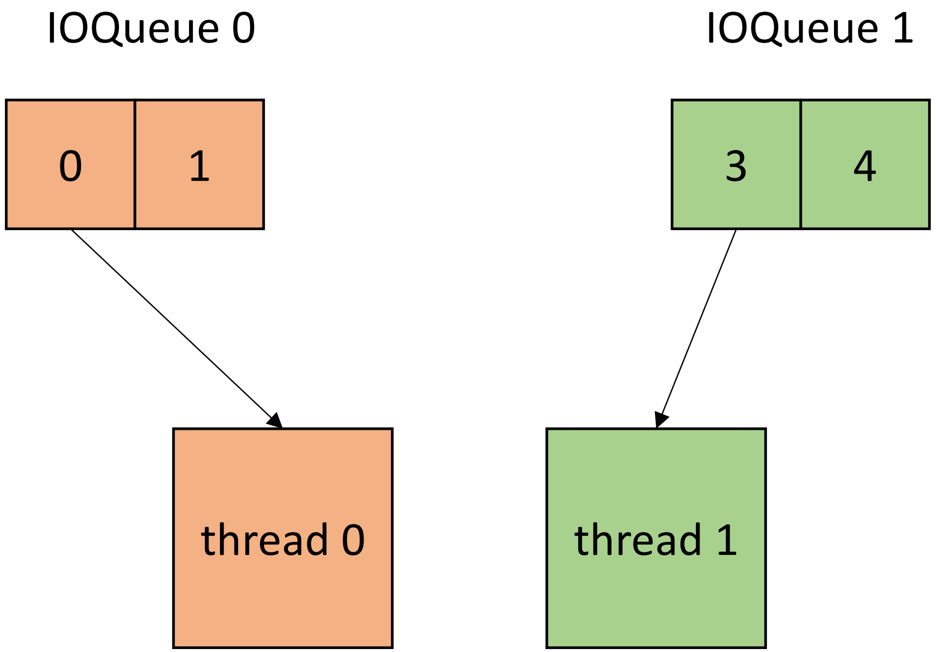
```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  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



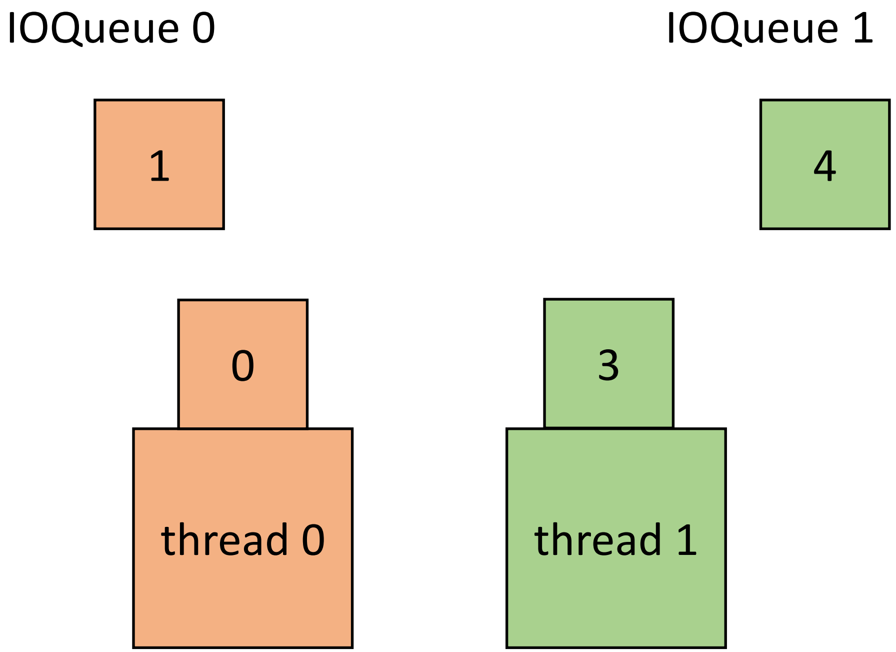

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  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



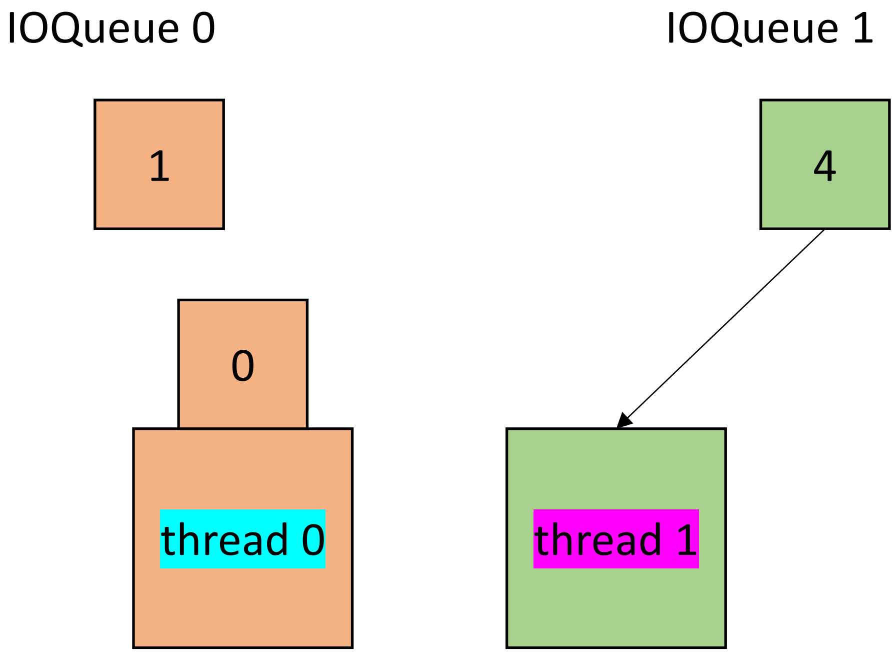

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  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



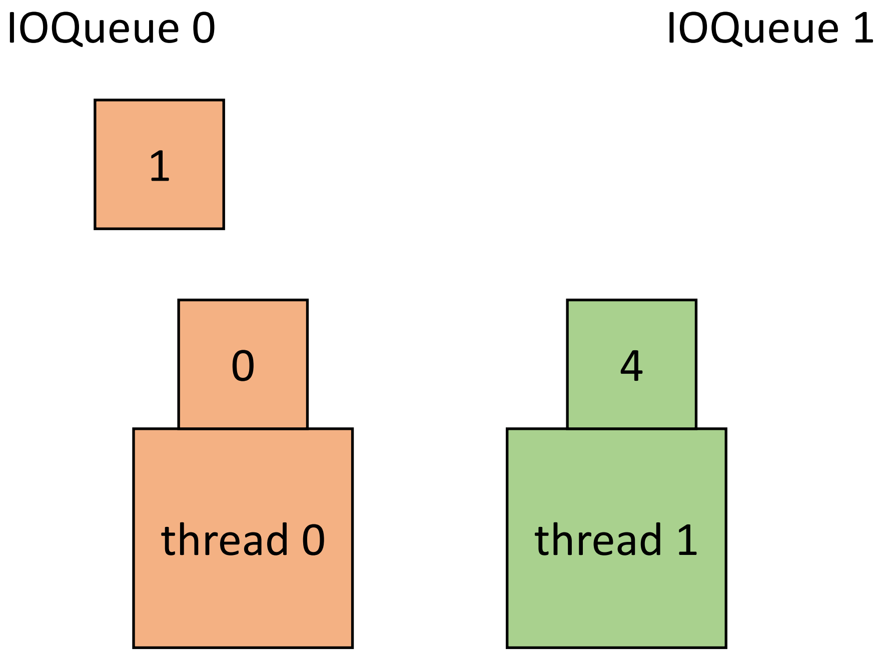

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  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



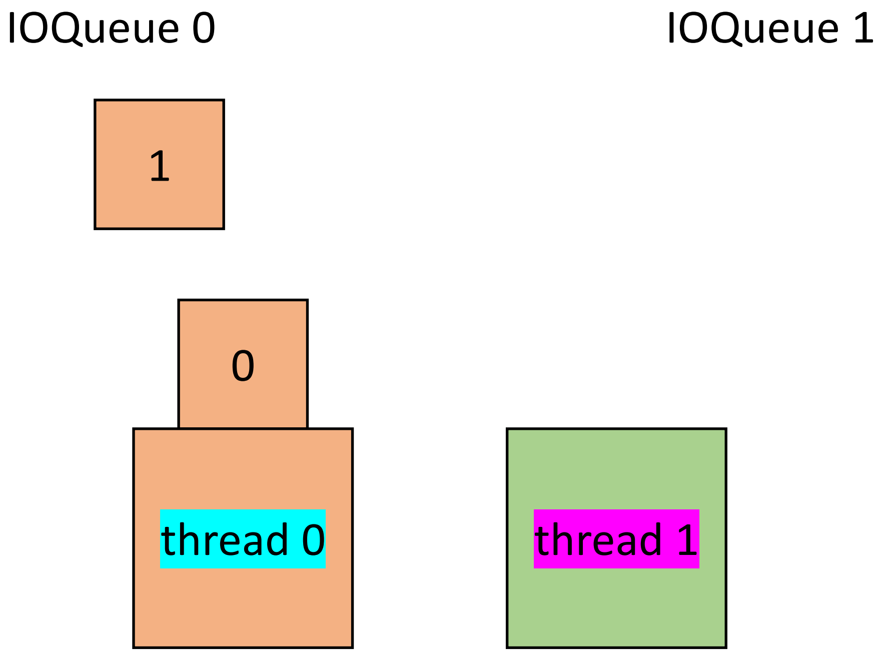

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  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
    }
  }
}
```

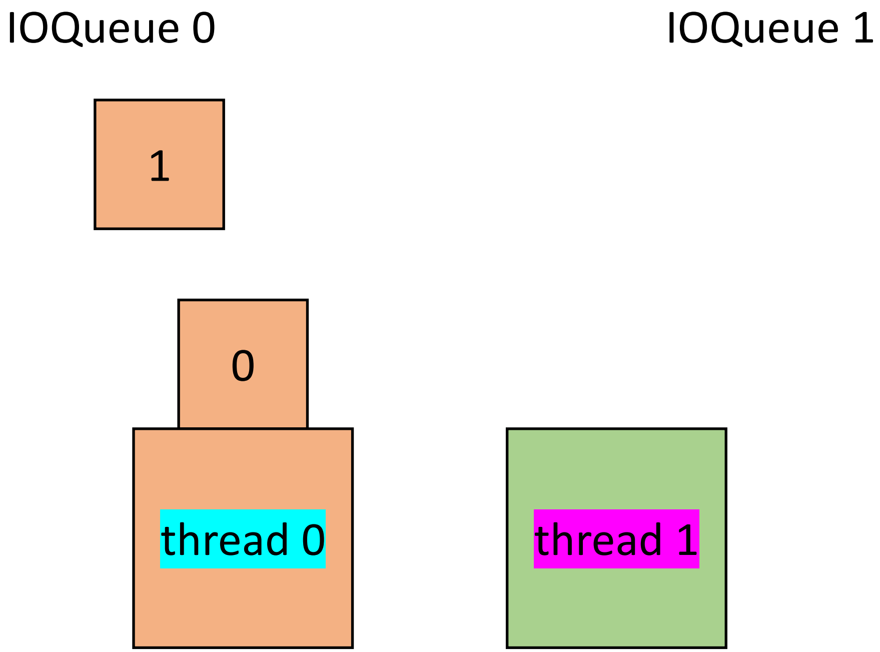
Work stealing - local worklists




```
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
    }
  }
}
```

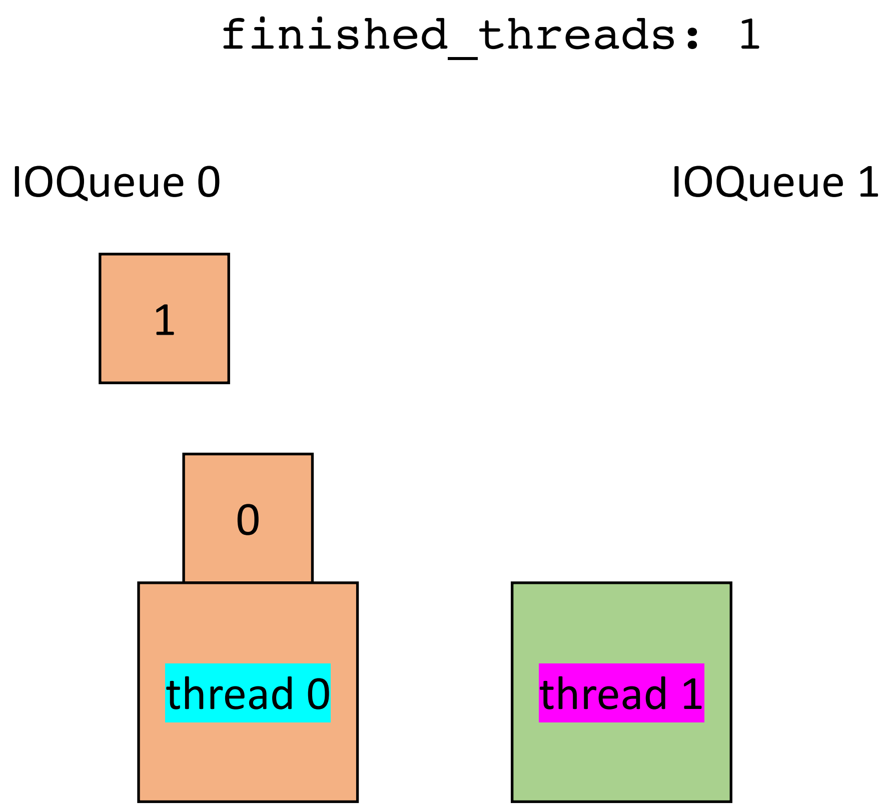
Work stealing - local worklists

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
    }
  }
}
```

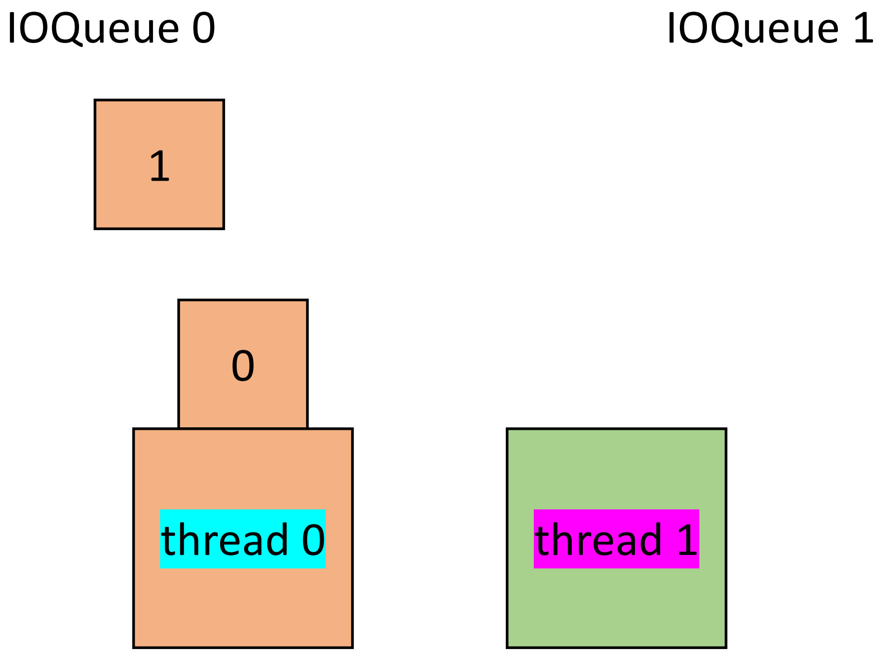
Work stealing - local worklists




```
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

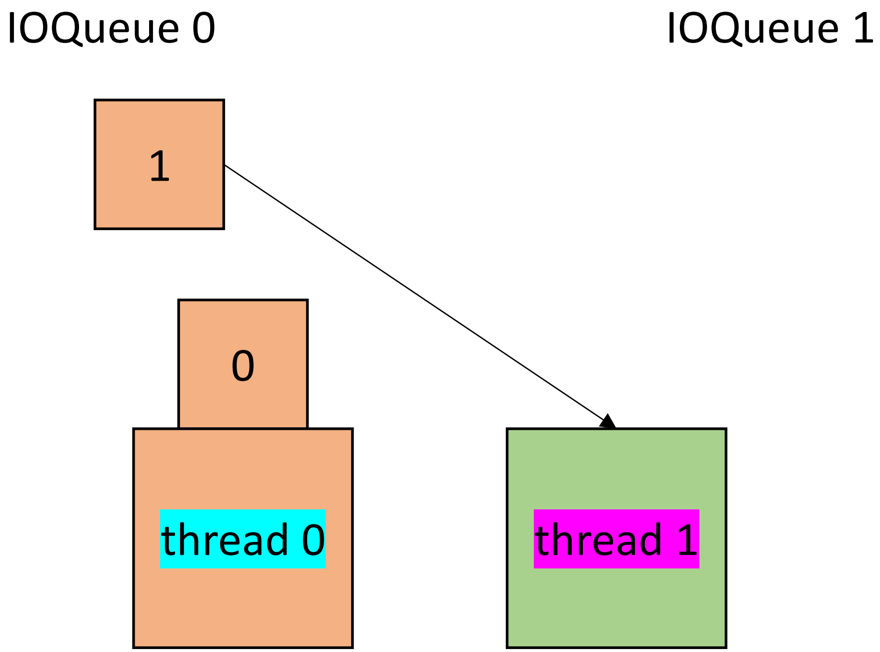
```
finished_threads: 1
```




```
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

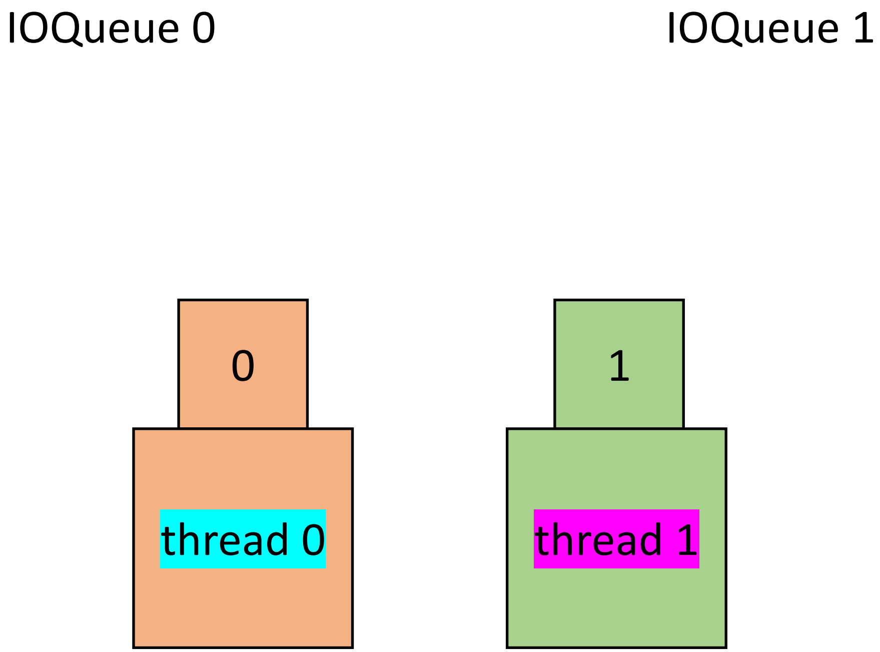
`finished_threads: 1`




```
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

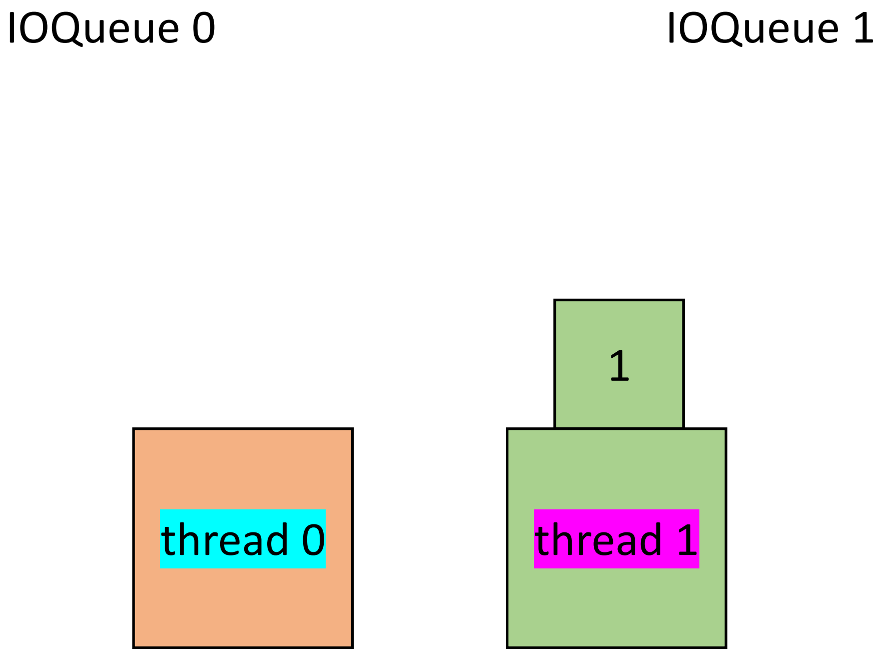
```
finished_threads: 1
```




```
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
```

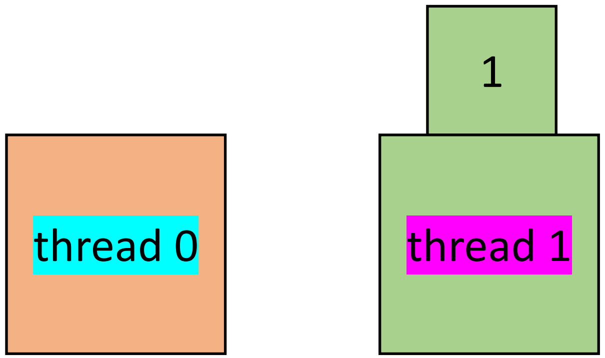



```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  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: 2
```

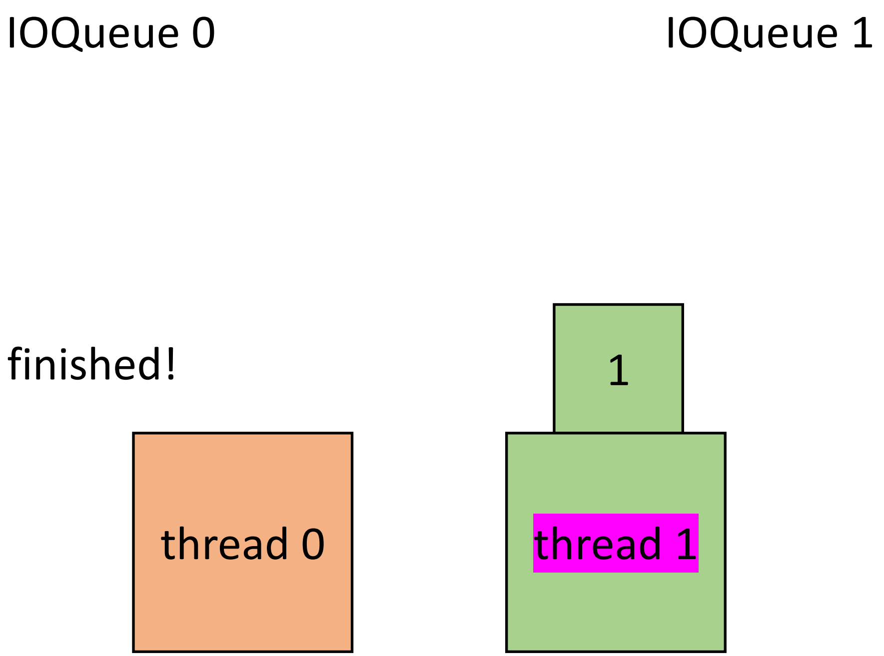
IOQueue 0 IOQueue 1




```
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`



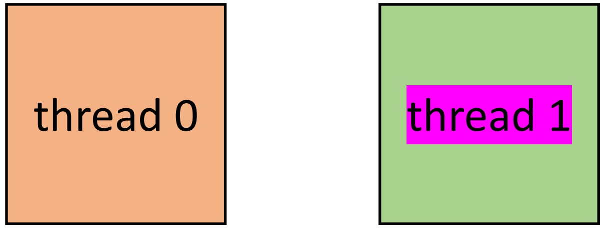

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  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



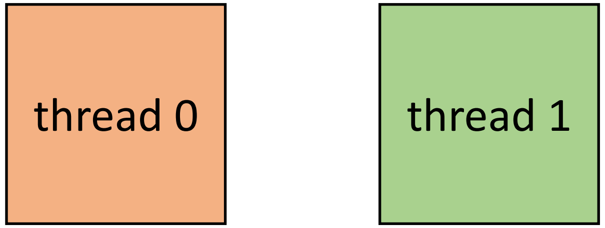

```
atomic_int finished_threads(0);
void parallel_loop(..., int tid, int num_threads) {

  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
    }
  }
}
```

Next week

- Further Reading:
  - Chapter 16 in the book discusses different types of workstealing
- We’ll see some of this implemented in OpenMP!
- Memory consistency models!