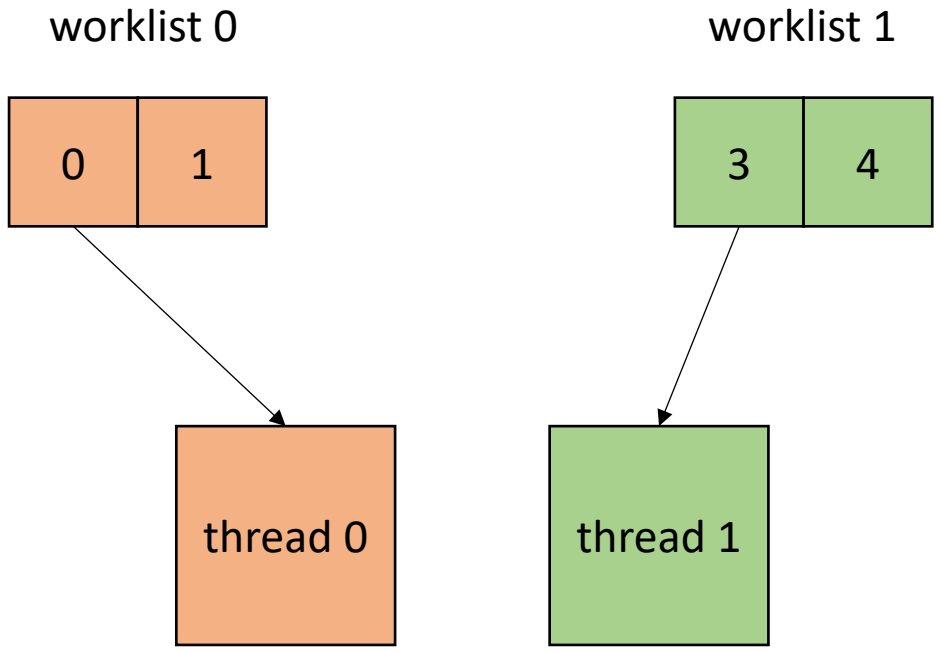
CSE211: Compiler Design

Nov. 19, 2020

- **Topic**: SMP parallelism
  - Compiler implementations!

- **Discussion questions**:
  - Do modern compilers automatically parallelize your code?
  - Have you ever used a auto-parallelizing compiler?



Announcements

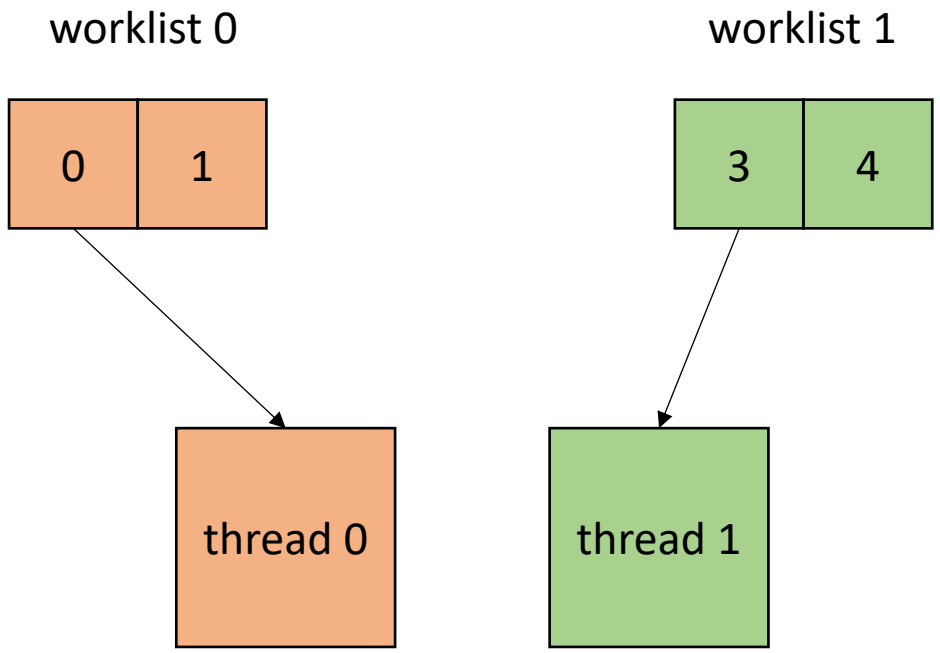
- Midterm is due today. Clarification questions are posted as discussions on Canvas. Resubmit by emailing me if you'd like
- HW3 is released. Due Dec. 4
- Paper/projects proposals due Nov. 24
- Guest speaker next lecture

CSE211: Compiler Design

Nov. 19, 2020

- **Topic**: SMP parallelism
  - Compiler implementations!

- **Discussion questions**:
  - Do modern compilers automatically parallelize your code?
  - Have you ever used a auto-parallelizing compiler?



Implementing SMP parallelism in a compiler

- Where to do it?
  - **High-level**: DSL, Python, etc.
  - **Mid-level**: C/C++
  - **Low-level**: LLVM-IR
  - **ISA**: e.g. x86

Implementing SMP parallelism in a compiler

- Where to do it?
  - **High-level**: DSL, Python, etc.
  - **Mid-level**: C/C++
  - **Low-level**: LLVM-IR
  - **ISA**: e.g. x86

Tradeoffs at all levels

Implementing SMP parallelism in a compiler

- Where to do it?
  - **High-level**: DSL, Python, etc.
  - **Mid-level**: C/C++
  - **Low-level**: LLVM-IR
  - **ISA**: e.g. x86

Here you've lost information about for loops, but SSA provides a nice foundation for analysis

Implementing SMP parallelism in a compiler

- Where to do it?
  - **High-level**: DSL, Python, etc.
  - **Mid-level**: C/C++
  - **Low-level**: LLVM-IR
  - **ISA**: e.g. x86

Good frameworks available for managing threads (C++, OpenMP).
Good tooling for analysis and codegen clang visitors, pycparser, etc.

Implementing SMP parallelism in a compiler

- Where to do it?
  - **High-level**: DSL, Python, etc.
  - **Mid-level**: C/C++
  - **Low-level**: LLVM-IR
  - **ISA**: e.g. x86

In many cases, DSLs compiler down to, or link to C/C++: DNN libraries, Graph analytic DSLs, Numpy.

Some DSLs compile to LLVM: Numba

Implementing SMP parallelism in a compiler

- Where to do it?
  - **High-level**: DSL, Python, etc.
  - **Mid-level**: C/C++
  - **Low-level**: LLVM-IR
  - **ISA**: e.g. x86

We will assume this level for the lecture

Regular Parallel Loops

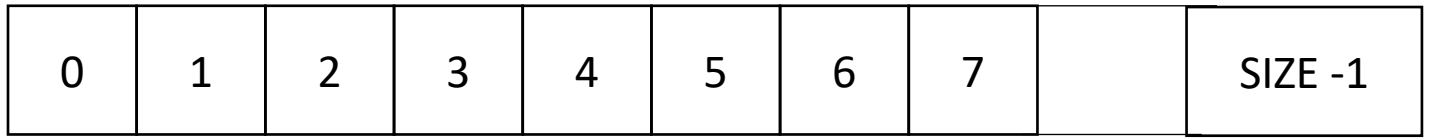
- How to implement in a compiler:

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

Regular Parallel Loops

- How to implement in a compiler:

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

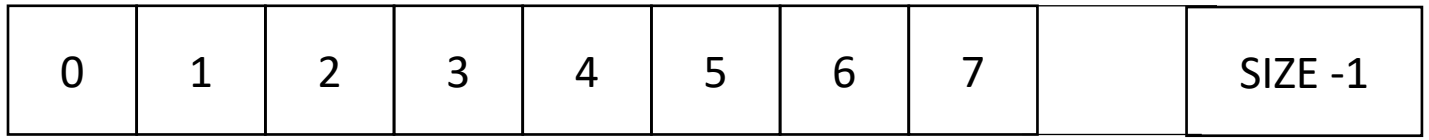


Regular Parallel Loops

- How to implement in a compiler:

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

say SIZE / NUM_THREADS = 4

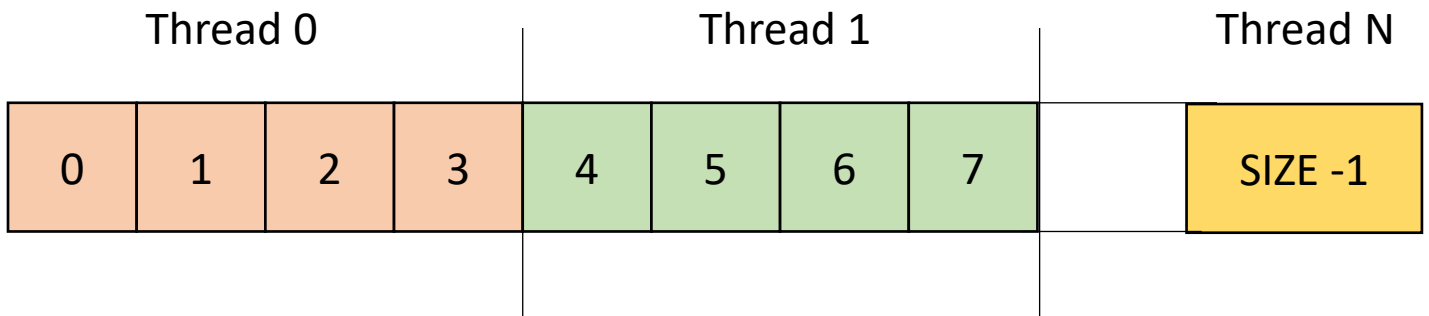


Regular Parallel Loops

- How to implement in a compiler:

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

say SIZE / NUM_THREADS = 4



Regular Parallel Loops

- How to implement in a compiler:

```
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

Regular Parallel Loops

- How to implement in a compiler:

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

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

  for (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

Regular Parallel Loops

- How to implement in a compiler:

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

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

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

determine chunk size in new function

Regular Parallel Loops

- How to implement in a compiler:

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

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

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

Set new loop bounds

Regular Parallel Loops

- How to implement in a compiler:

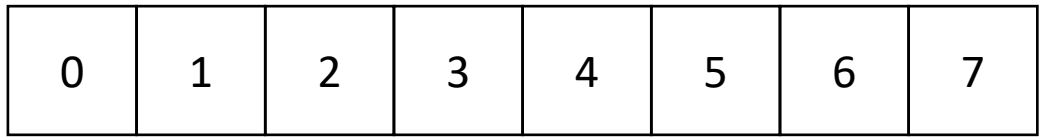
```
void foo() {
...
  for (int t = 0; t < NUM_THREADS; t++) {
    spawn(parallel_loop(..., t))
  }
  join();
...
}
```

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

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

Spawn threads

Regular Parallel Loops

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




```
chunk_size = 4
```

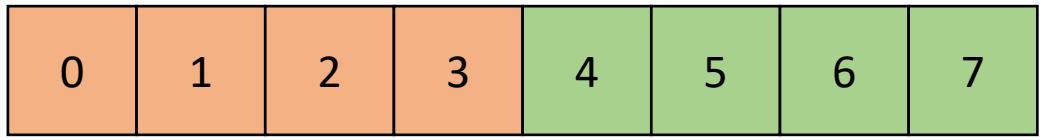
```
0: start= 0
0: end= 4
```

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


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

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

Regular Parallel Loops

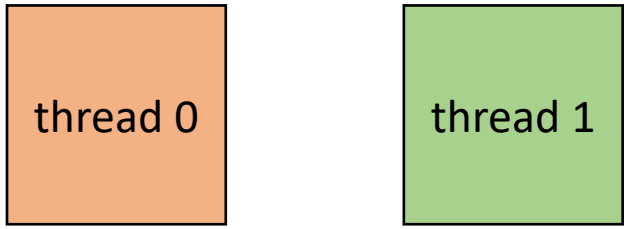
- Example, 2 threads/cores, array of size 8




```
chunk_size = 4
```

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

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

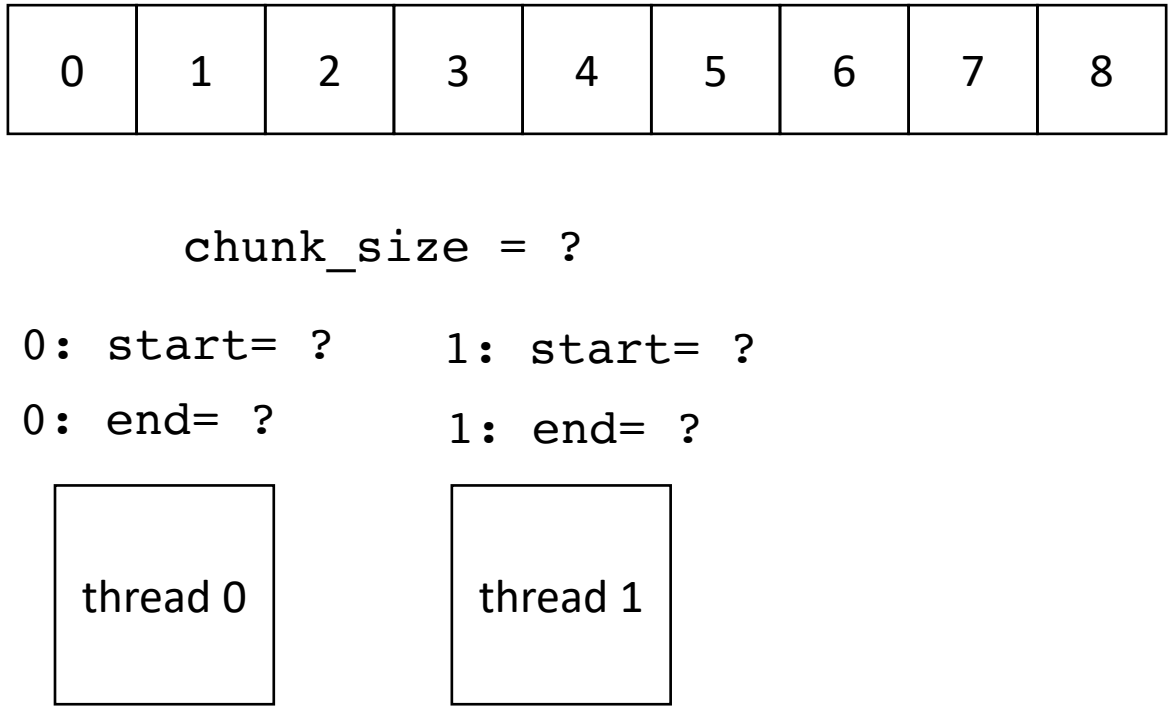



```
void parallel_loop(..., int tid) {

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

End example

Regular Parallel Loops

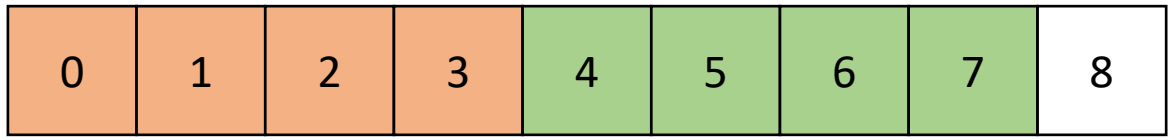
- Example, 2 threads/cores, array of size 9




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

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

Regular Parallel Loops

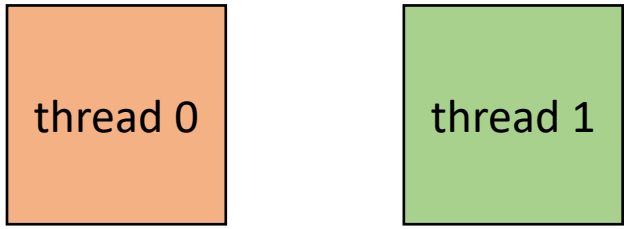
- Example, 2 threads/cores, array of size 9




```
chunk_size = 4
```

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

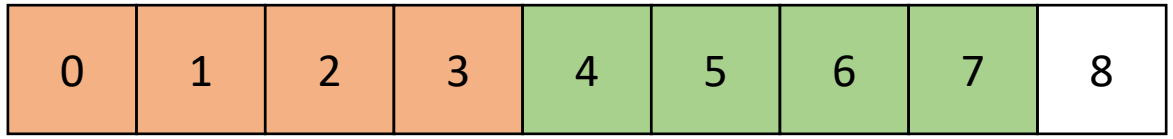
```
1: start = 4
1: end = 8
```



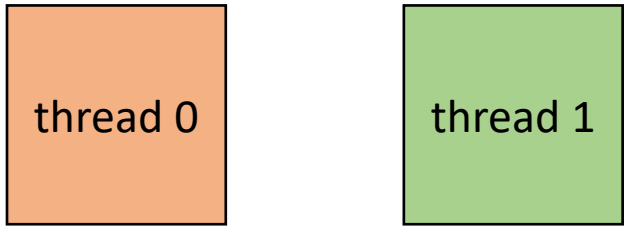

```
void parallel_loop(..., int tid) {

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

Regular Parallel Loops

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




```
chunk_size = 4

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

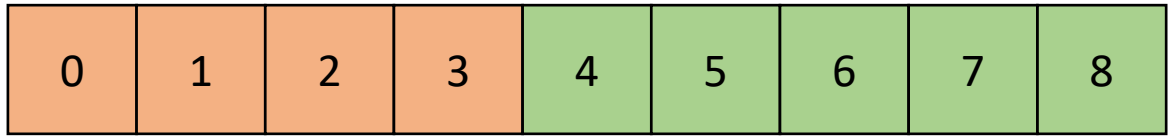



```
void parallel_loop(..., int tid) {

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

Regular Parallel Loops

last thread gets more work

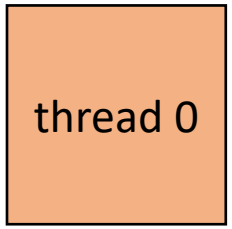
- Example, 2 threads/cores, array of size 9




```
chunk_size = 4
```

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

```
1: start = 4
1: end = 9
```



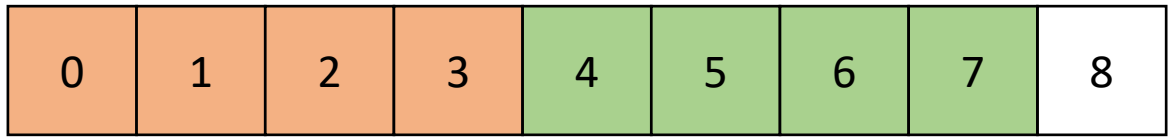

```
void parallel_loop(..., int tid) {

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

End example

Regular Parallel Loops

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

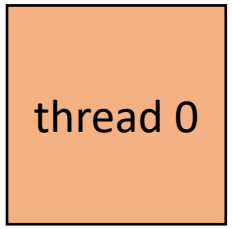
ceiling division




```
chunk_size = 4
```

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

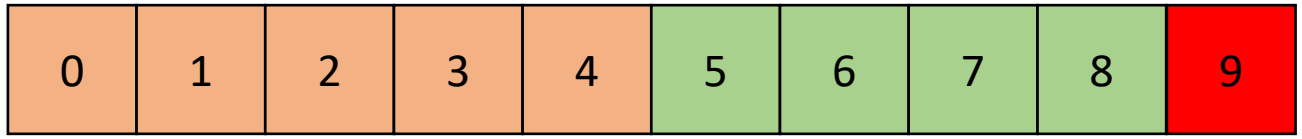
```
1: start = 4
1: end = 8
```




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

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

Regular Parallel Loops

- 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
```

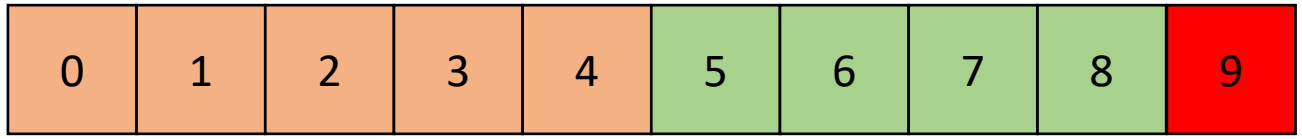
thread 0

thread 1

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

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

Regular Parallel Loops

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

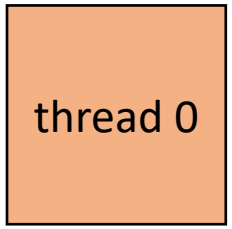


out of bounds

```
chunk_size = 5
```

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

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

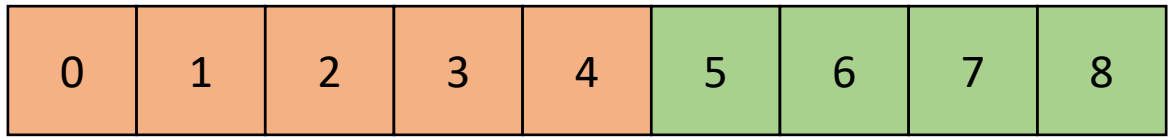



```
void parallel_loop(..., int tid) {

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

Regular Parallel Loops

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

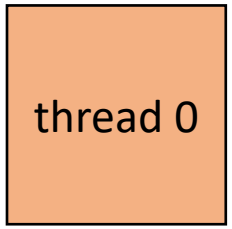
- Example, 2 threads/cores, array of size 9




```
chunk_size = 5
```

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

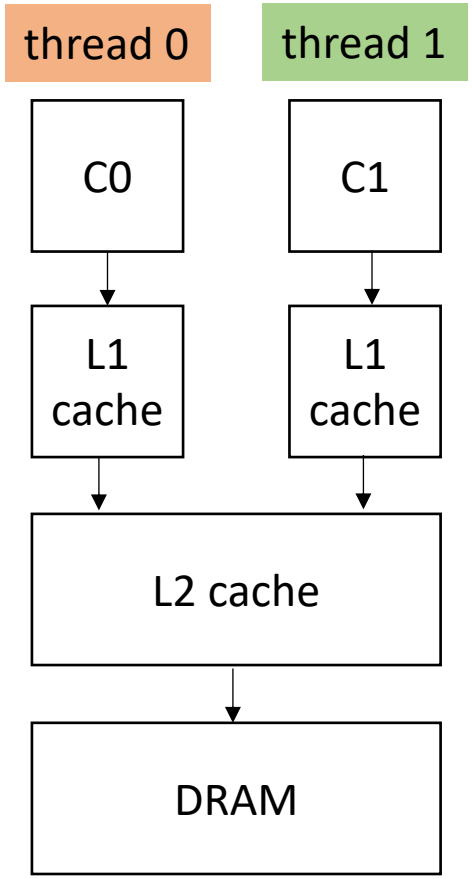
```
1: start = 5
1: end = 9
```



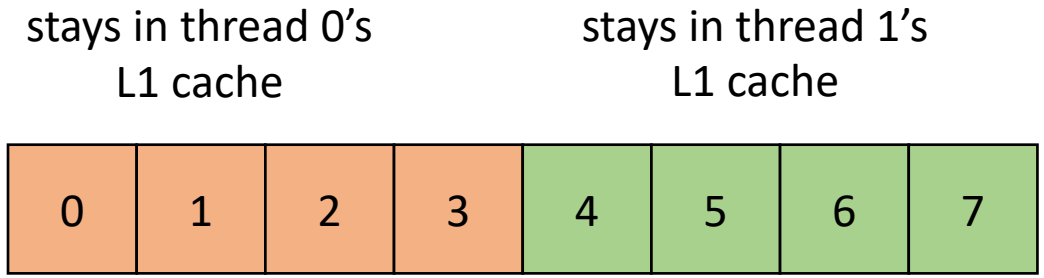

```
void parallel_loop(..., int tid) {

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

End example

Good for SMP parallelism



SMP parallelism

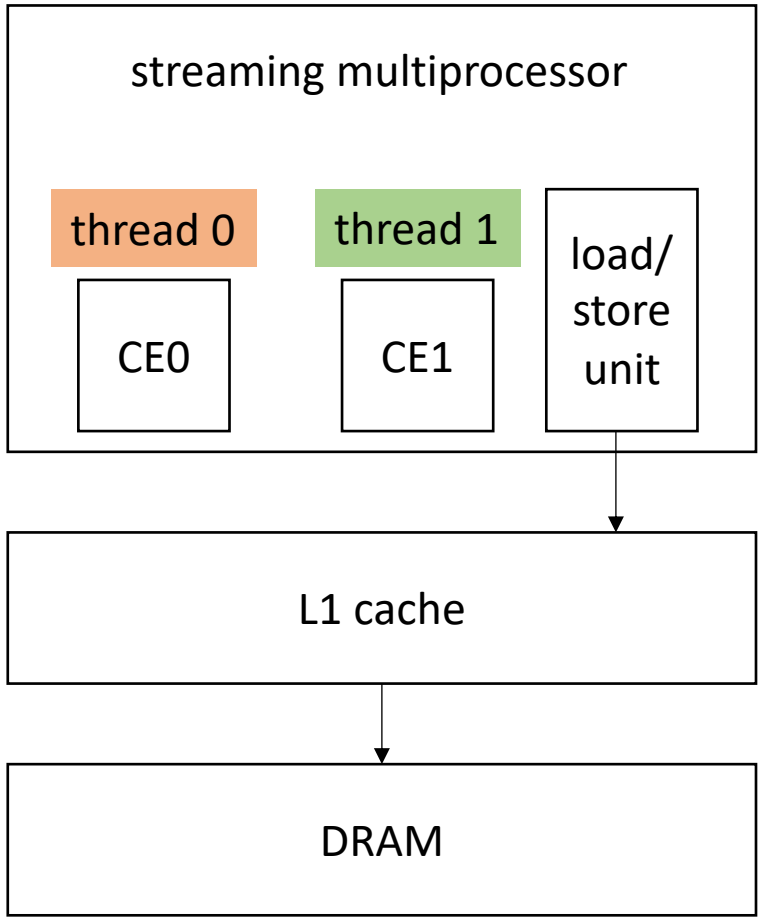


What about streaming multiprocessors (GPUs)?

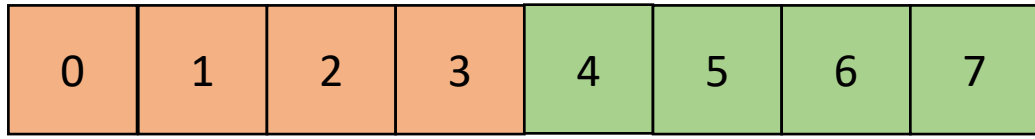
one streaming multiprocessor contains many small Compute Elements (CE)

CEs Can load adjacent memory locations simultaneously.

CEs execute iterations synchronously



is this partition good for GPUs??

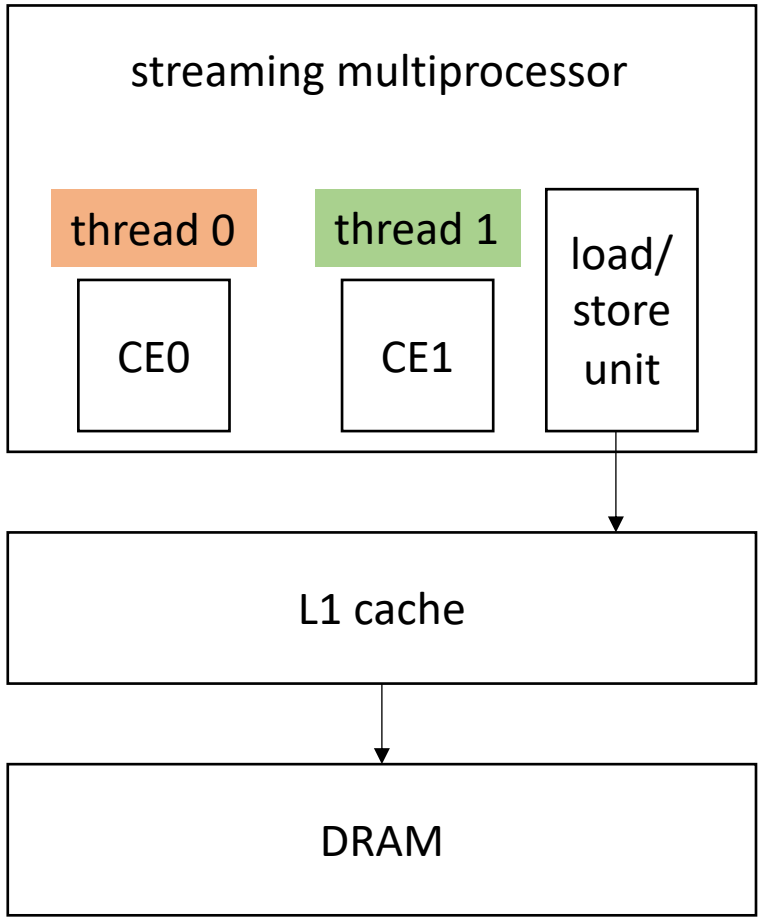


What about streaming multiprocessors (GPUs)?

one streaming multiprocessor contains many small Compute Elements (CE)

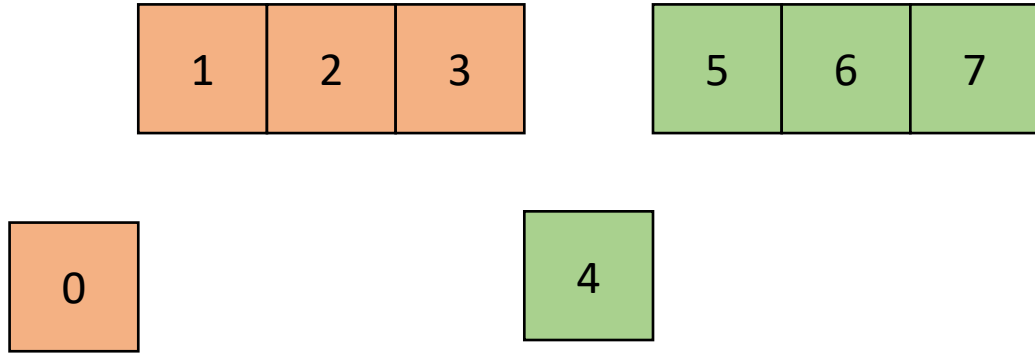
CEs Can load adjacent memory locations simultaneously.

CEs execute iterations synchronously



is this partition good for GPUs??

ITER 0:

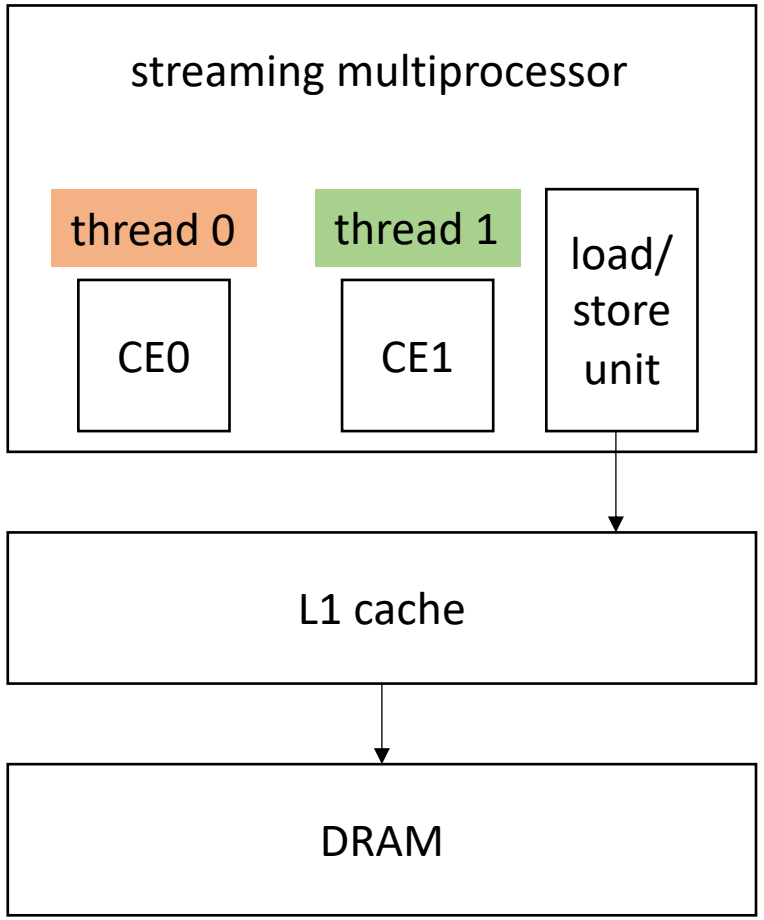


What about streaming multiprocessors (GPUs)?

one streaming multiprocessor contains many small Compute Elements (CE)

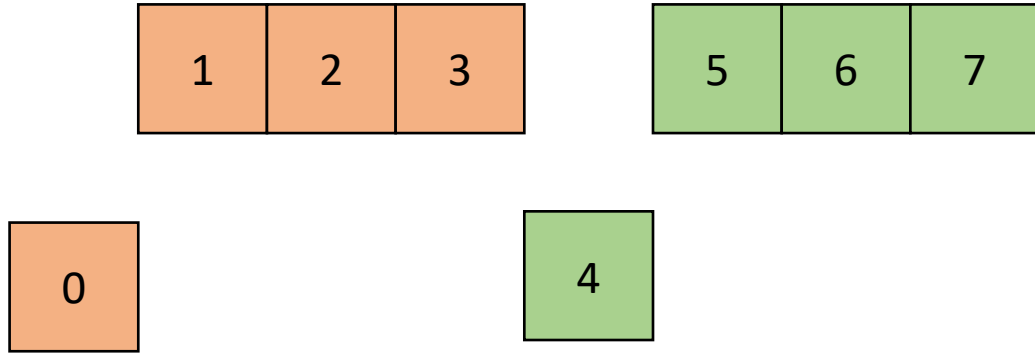
CEs Can load adjacent memory locations simultaneously.

CEs execute iterations synchronously



is this partition good for GPUs??

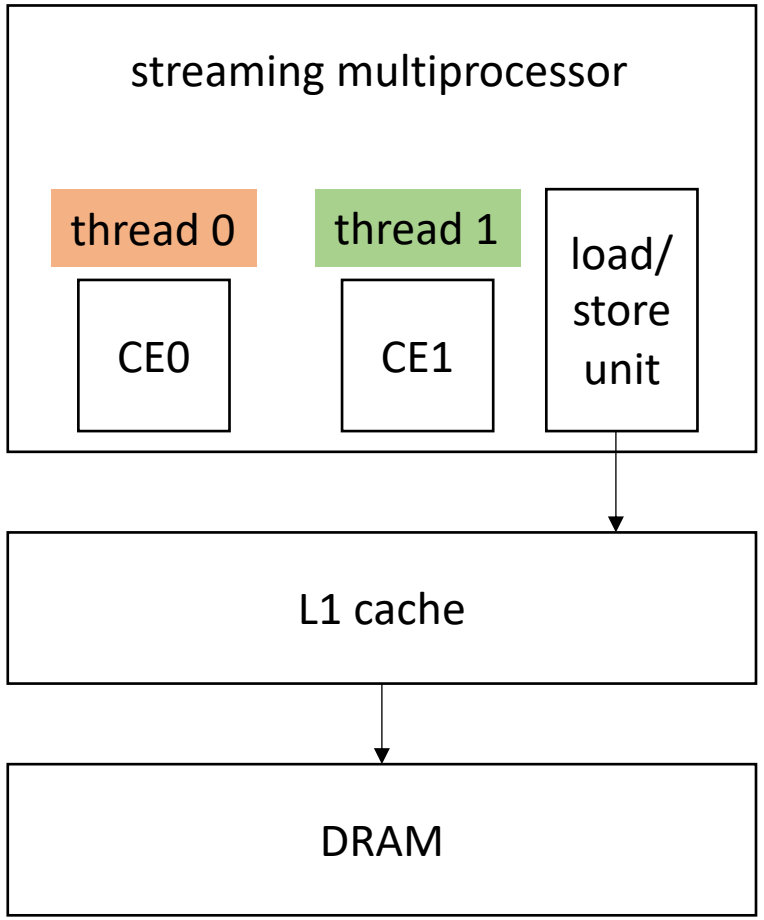
ITER 0:



not adjacent, so the loads have to be serialized

What about streaming multiprocessors (GPUs)?

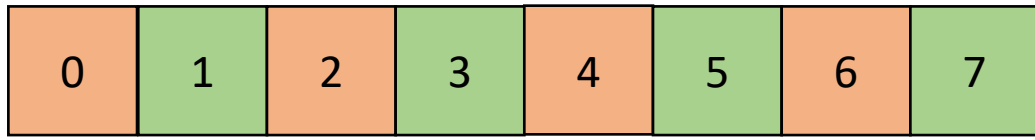
one streaming multiprocessor contains many small Compute Elements (CE)



CEs Can load adjacent memory locations simultaneously

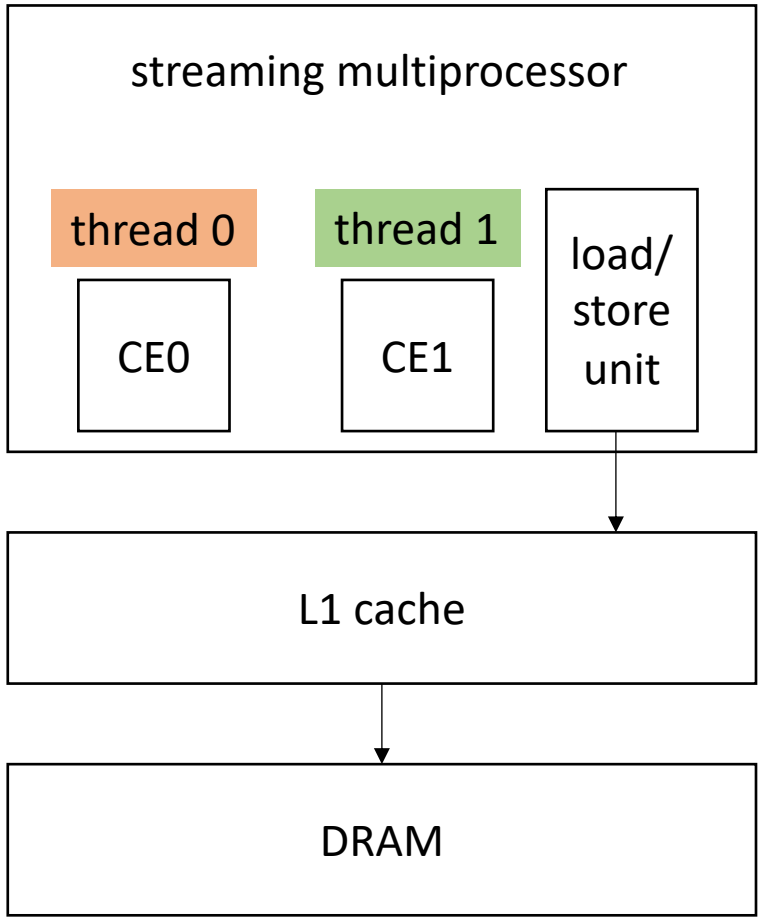
ITER 0:

What about a striped pattern?



What about streaming multiprocessors (GPUs)?

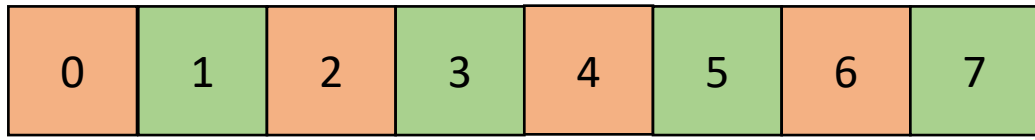
one streaming multiprocessor contains many small Compute Elements (CE)



CEs Can load adjacent memory locations simultaneously

ITER 0:

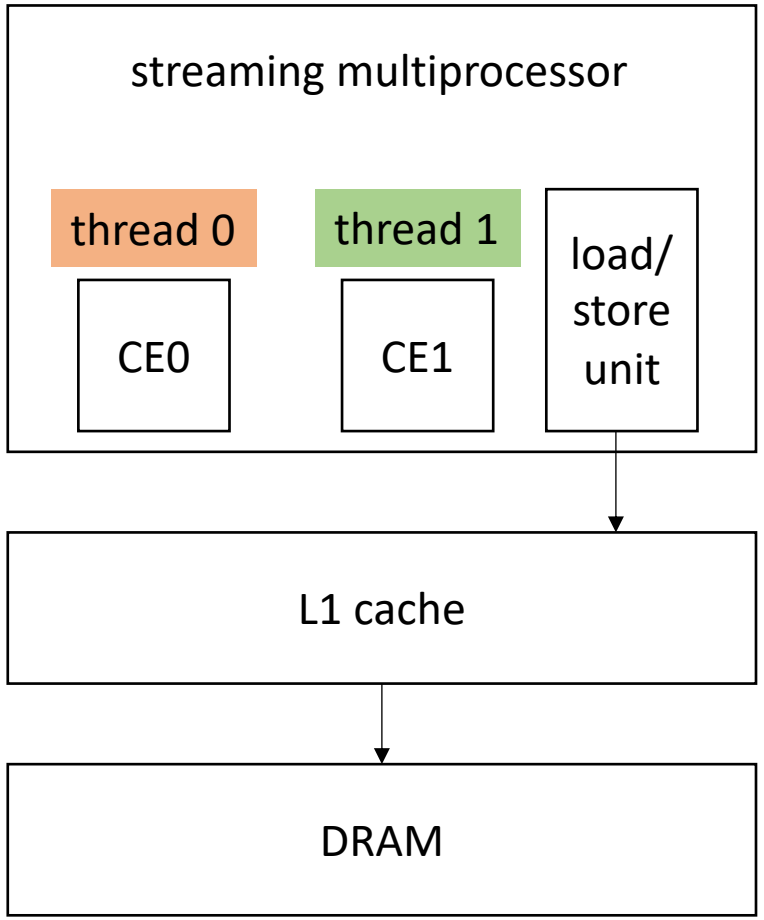
What about a striped pattern?



What about streaming multiprocessors (GPUs)?

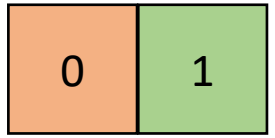
one streaming multiprocessor contains many small Compute Elements (CE)

CEs Can load adjacent memory locations simultaneously



What about a striped pattern?

ITER 0:



adjacent memory locations can be loaded at the same time!

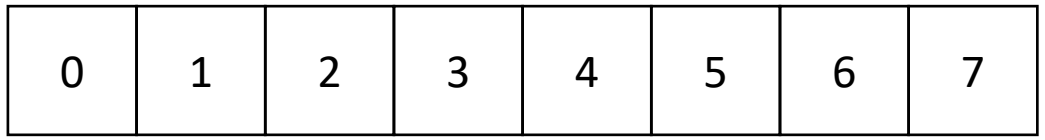
End example

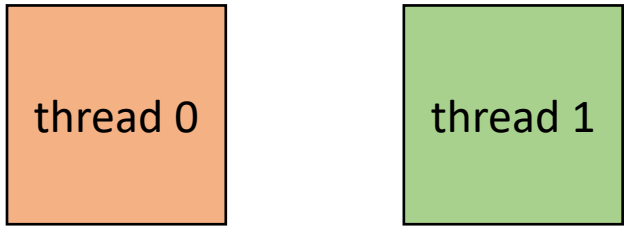
Kepler architecture

From:
https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

How to compiler for GPUs?

- Example, 2 threads/cores, array of size 8.
  Change code for a GPU

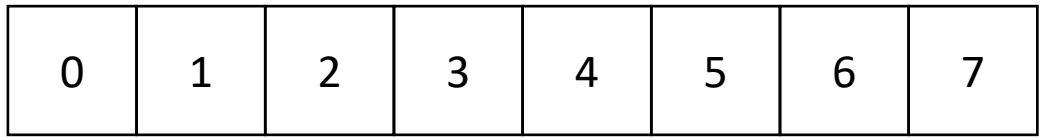





```
void parallel_loop(..., int tid) {

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

How to compiler for GPUs?

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

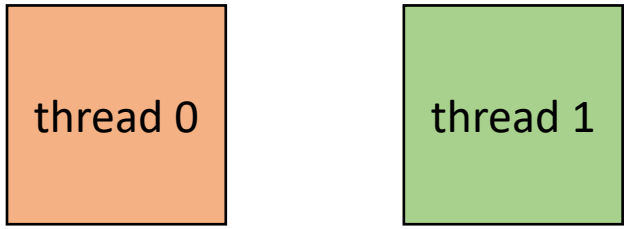



```
void parallel_loop(..., int tid) {

  for (x = tid; x < end; x+=NUM_THREADS) {
    // work based on x
  }
}
```

ITER 0

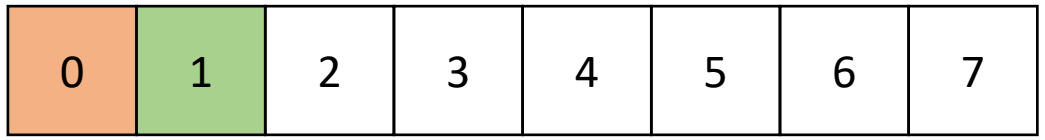
x: ?

x: ?



How to compiler for GPUs?

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

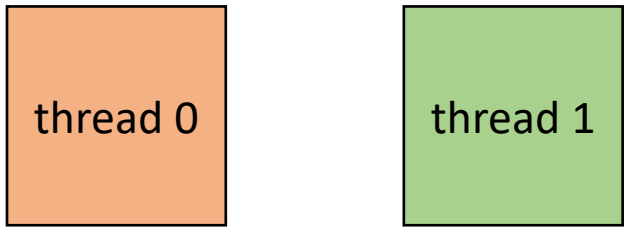



```
void parallel_loop(..., int tid) {

  for (x = tid; x < end; x+=NUM_THREADS) {
    // work based on x
  }
}
```

ITER 0

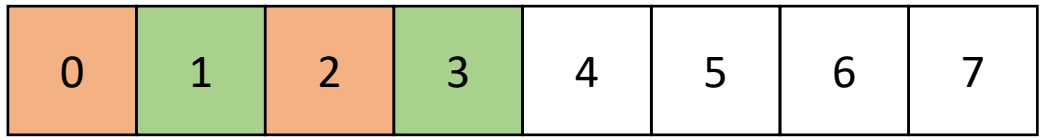
x: 0

x: 1



How to compiler for GPUs?

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

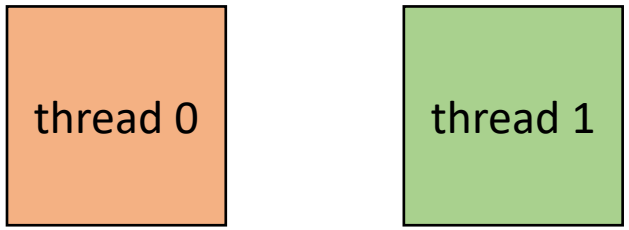



```
void parallel_loop(..., int tid) {

  for (x = tid; x < end; x+=NUM_THREADS) {
    // work based on x
  }
}
```

ITER 1

x: 2

x: 3



End example

Takeaways:

- Chunk data for SMP parallelism. Cores have disjoint L1 caches.
- Stride data for SM (GPU) parallelism, adjacent threads can more efficiently access adjacent memory.
- Easily compute bounds using runtime variables
  - `SIZE, NUM_THREADS, THREAD_ID`
- Create one function parameterized by thread id (SPMD parallelism)

Irregular parallelism in loops

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

Irregular parallelism in loops

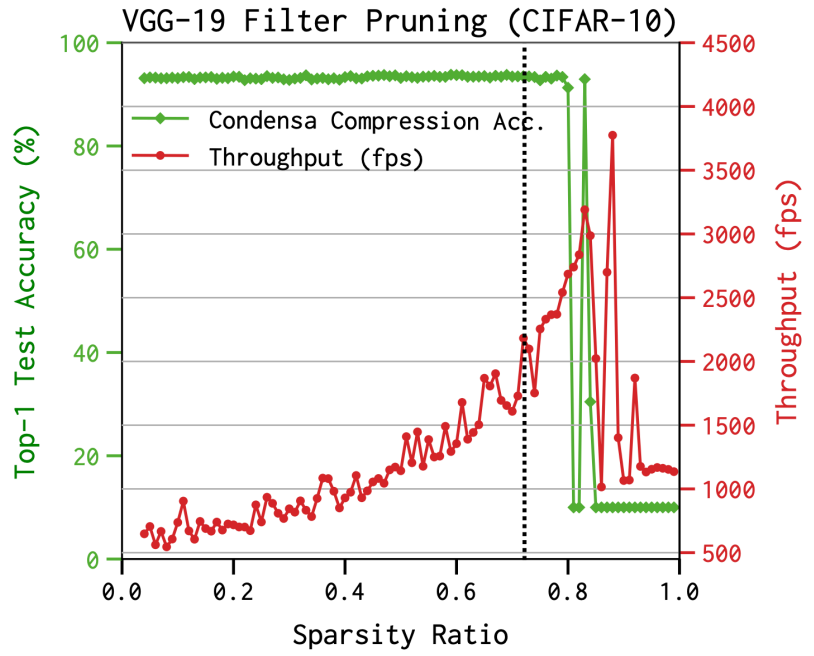
- Tasks are not balanced
- Appears in lots of emerging workloads

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



Irregular parallelism in loops

- Tasks are not balanced

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

Given the end of Dennard's scaling, we should aim to get these applications to scale better!

Simple parallelism across the x loop only gives 1.3x speedup with 2 threads

This can be improved using load-balancing strategies, like workstealing

Work stealing

- Tasks are dynamically assigned to threads.

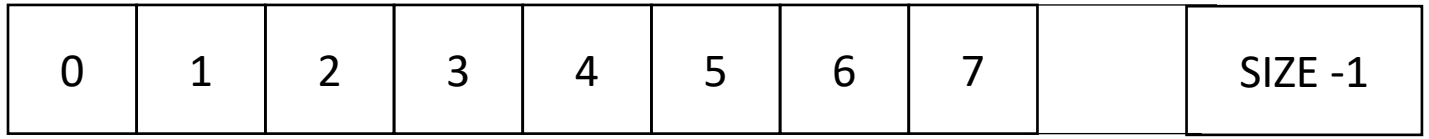
Work stealing - global implicit worklist

- Pros
  - Simple to implement

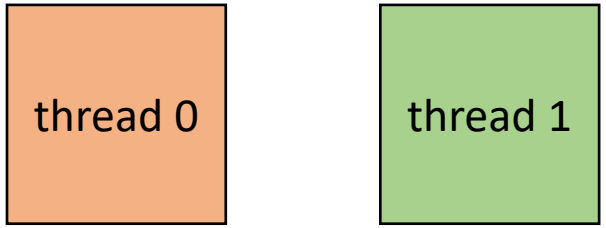
- Cons:
  - High contention on global counter
  - Potentially bad memory locality.

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

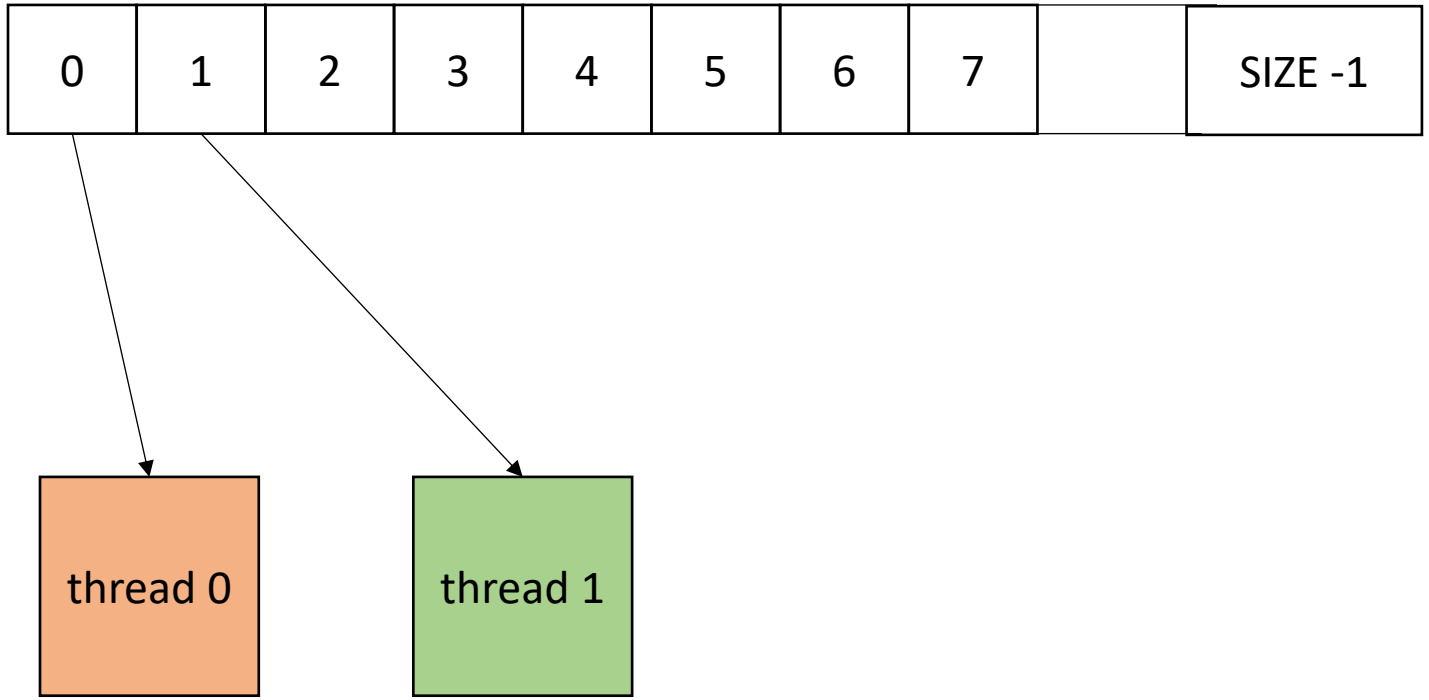


cannot color initially!



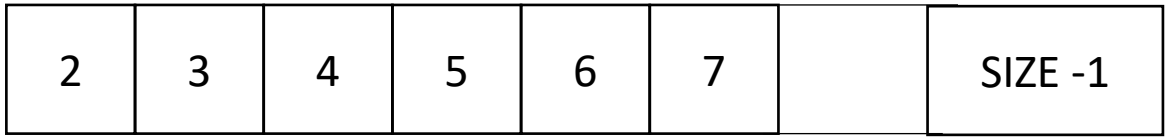
Work stealing - global implicit worklist

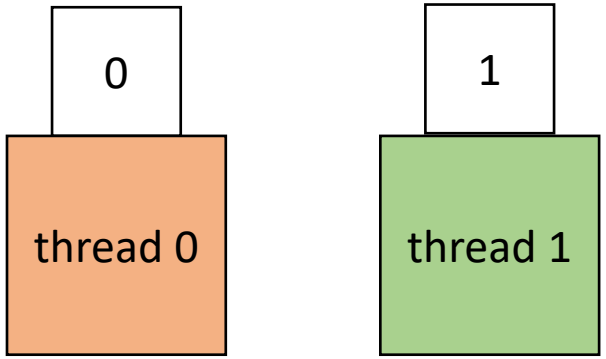
- Global worklist: threads take tasks (iterations) dynamically



Work stealing - global implicit worklist

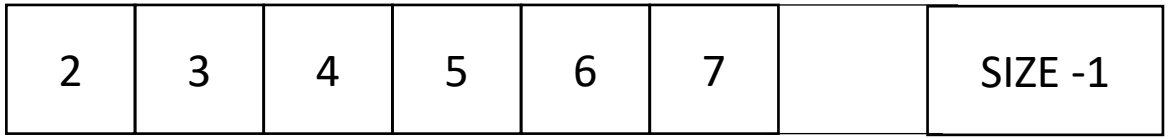
- Global worklist: threads take tasks (iterations) dynamically

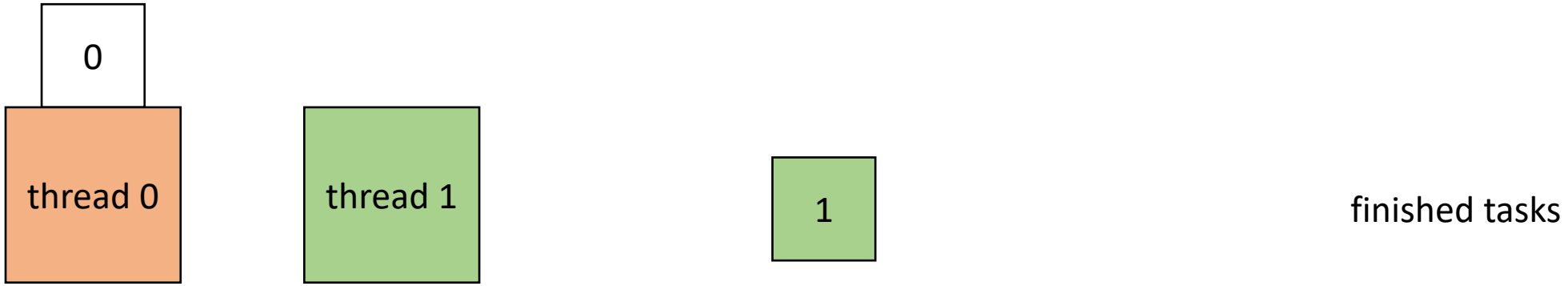




Work stealing - global implicit worklist

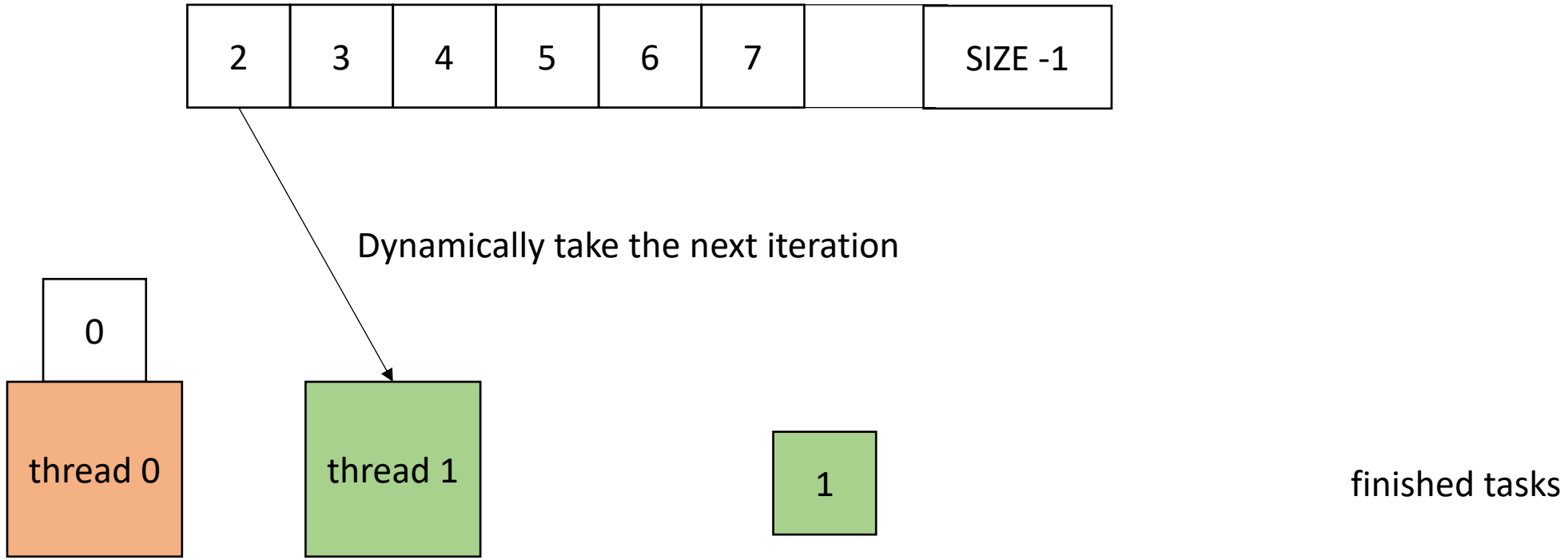
- Global worklist: threads take tasks (iterations) dynamically





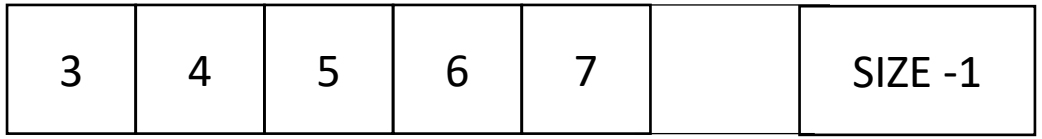
Work stealing - global implicit worklist

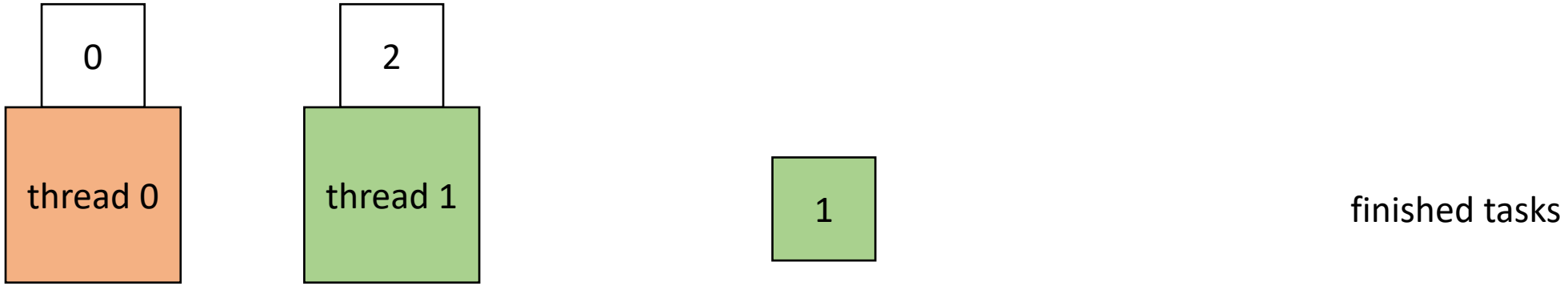
- Global worklist: threads take tasks (iterations) dynamically



Work stealing - global implicit worklist

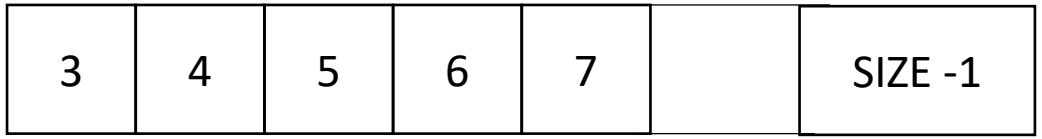
- Global worklist: threads take tasks (iterations) dynamically

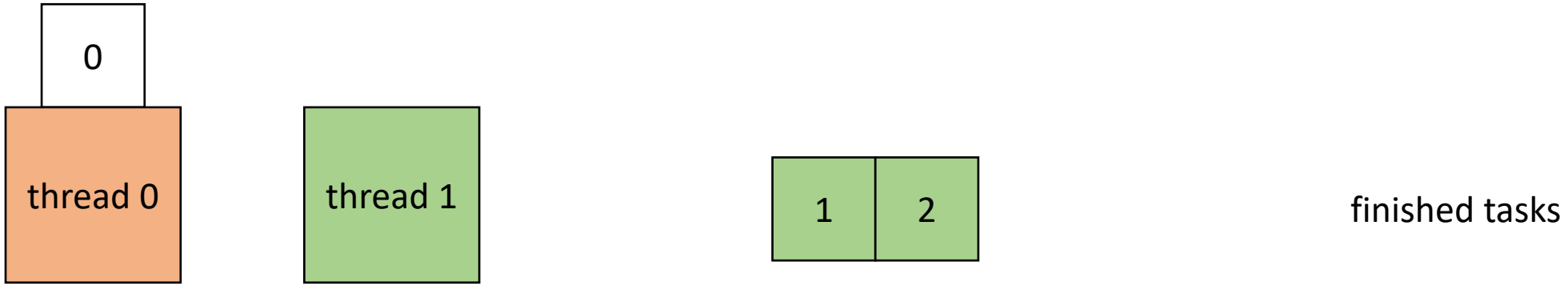




Work stealing - global implicit worklist

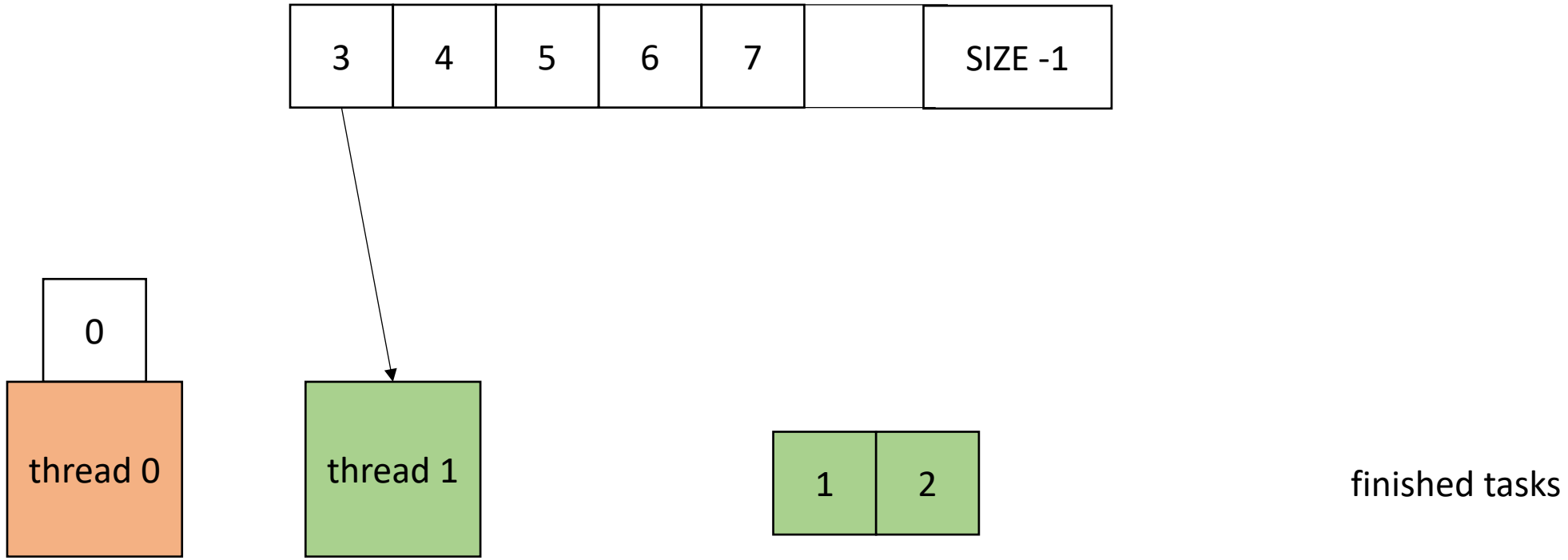
- Global worklist: threads take tasks (iterations) dynamically





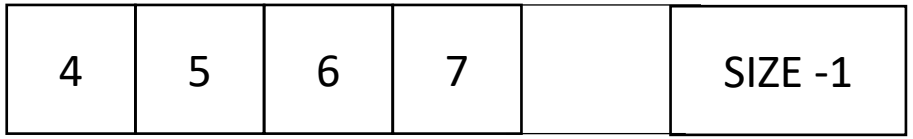
Work stealing - global implicit worklist

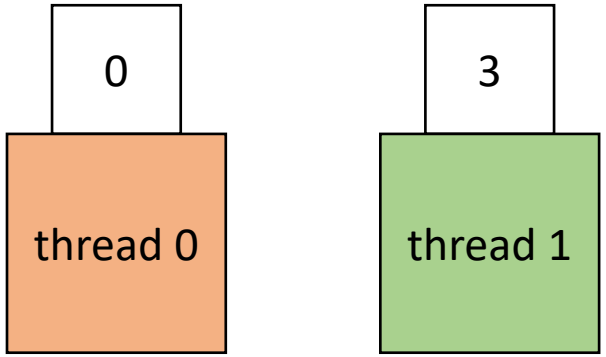
- Global worklist: threads take tasks (iterations) dynamically

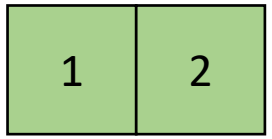


Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



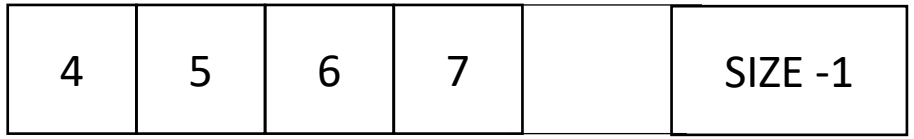


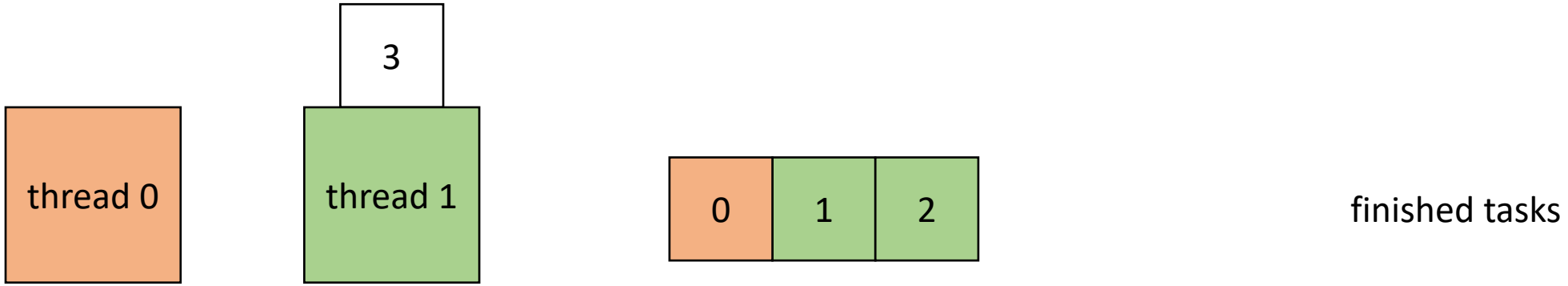


finished tasks

Work stealing - global implicit worklist

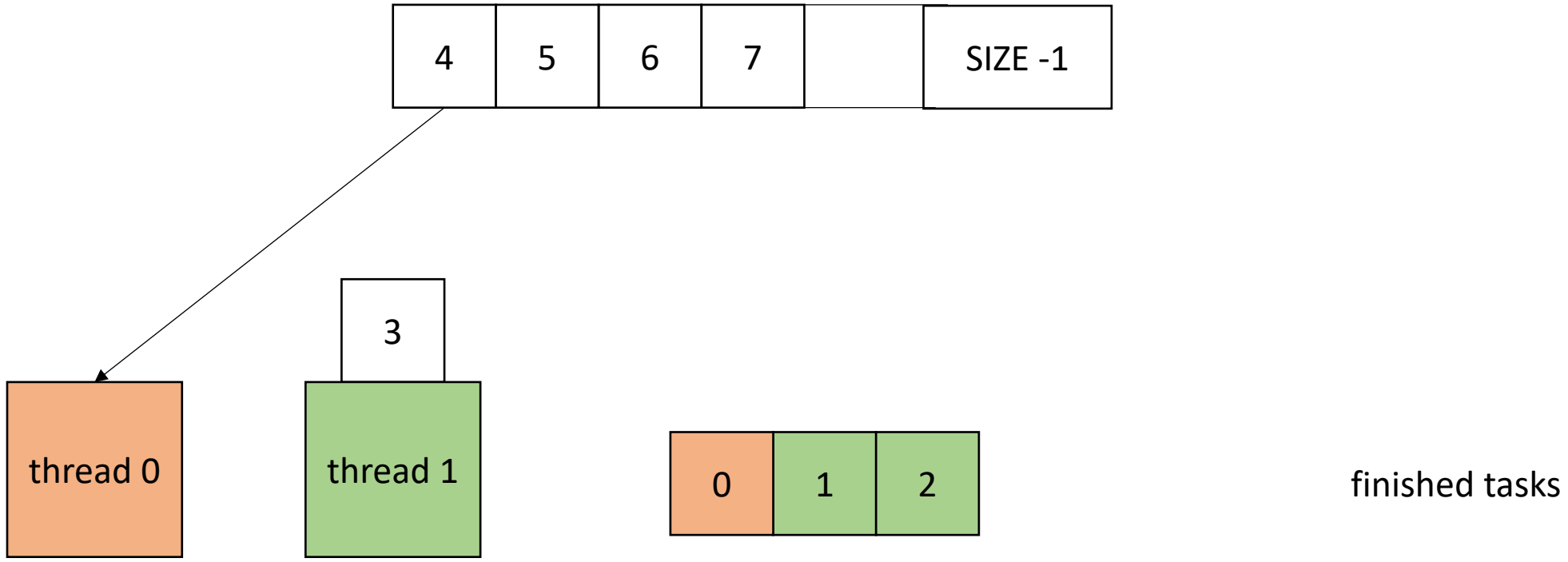
- Global worklist: threads take tasks (iterations) dynamically





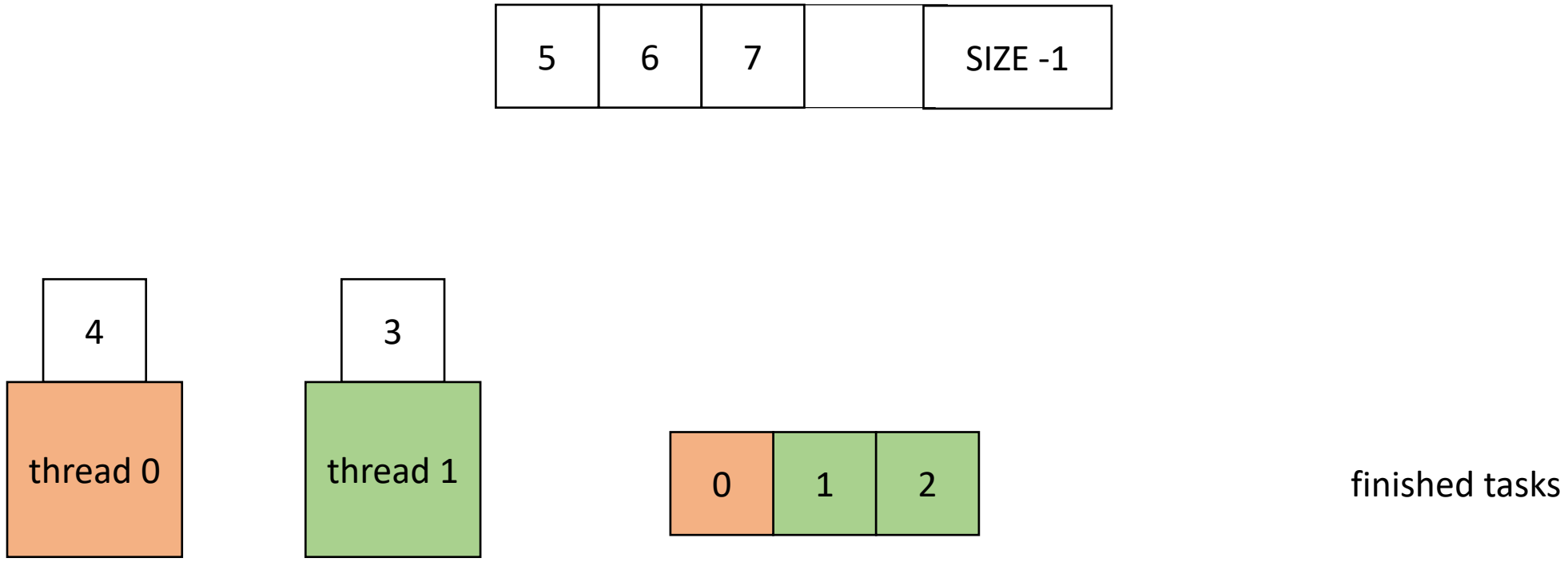
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



End example

Work stealing - global implicit worklist

- How to implement in a compiler:

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

Work stealing - global implicit worklist

- How to implement in a compiler:

```
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.
This creates SPMD parallelism.

Work stealing - global implicit worklist

- How to implement in a compiler:

```
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 in a compiler:

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

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

   for (int local_x = x++;
            local_x < SIZE;
            local_x = 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 in a compiler:

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

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

   for (int local_x = x++;
            local_x < SIZE;
            local_x = x++) {

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

These must be atomic updates!

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

Work stealing - global implicit worklist

- How to implement in a compiler:

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

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

  for (int local_x = x++;
           local_x < SIZE;
           local_x = x++) {

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

Spawn threads in original function and join them afterwards

Work stealing - global implicit worklist

- How to implement in a compiler:

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

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

  for (int local_x = x++;
           local_x < SIZE;
           local_x = x++) {

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

Are we finished?

Work stealing - global implicit worklist

- How to implement in a compiler:

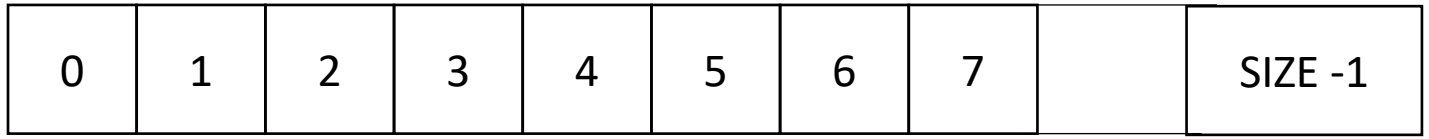
```
void foo() {
  ...
  for (t = 0; x < THREADS; t++) {
    spawn(parallel_loop);
  }
  join();
  x = 0;
  ...
}
```

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

  for (int local_x = x++;
           local_x < SIZE;
           local_x = x++) {

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

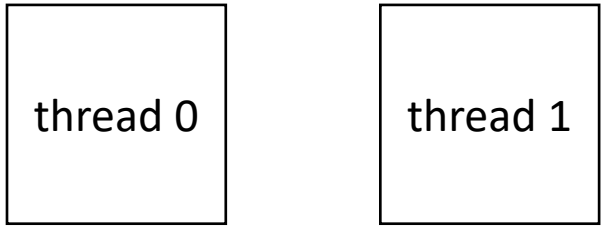
Are we finished?

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



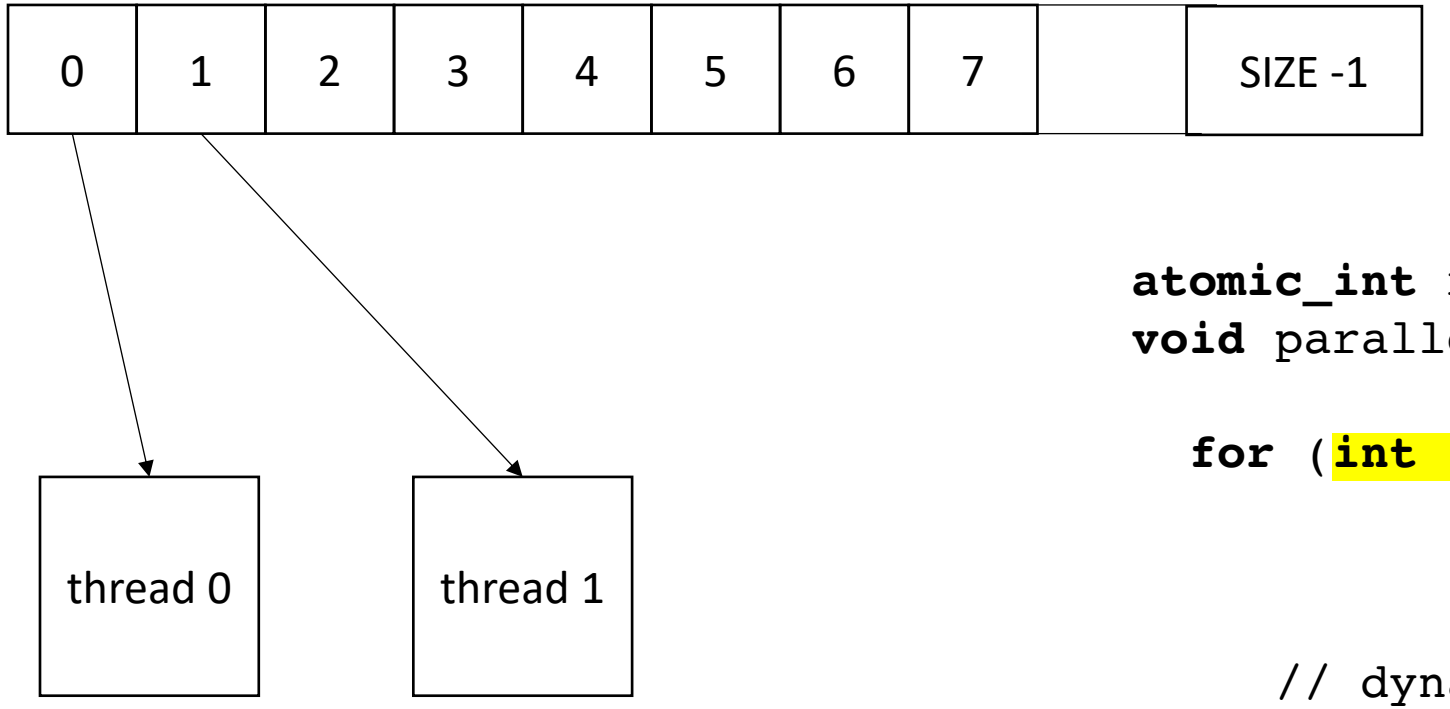

```
x: 0
0 - local_x - UDEF
1 - local_x - UNDEF
```




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

  for (int local_x = x++;
           local_x < SIZE;
           local_x = x++) {

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

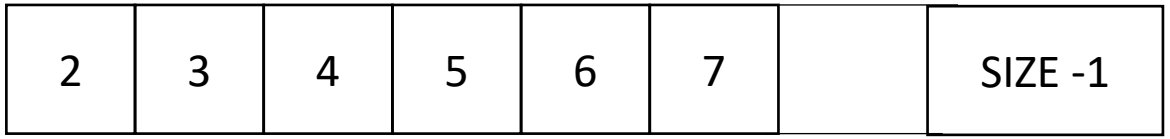



```
x: 2
0 - local_x - 0
1 - local_x - 1
```

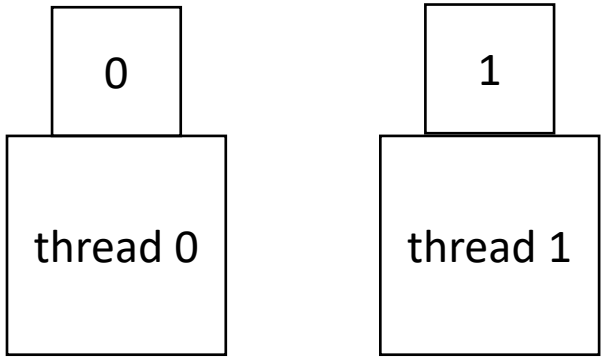
```
atomic_int x = 0;
void parallel_loop(...) {

  for (int local_x = x++;
           local_x < SIZE;
           local_x = x++) {

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



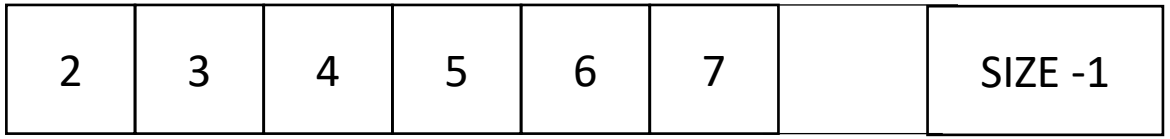

```
x: 2
0 - local_x - 0
1 - local_x - 1
```



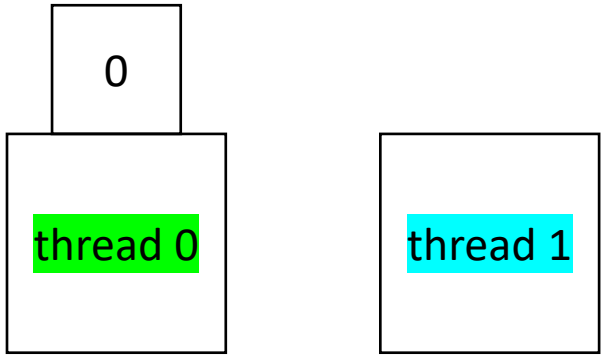

```
atomic_int x = 0;
void parallel_loop(...) {

  for (int local_x = x++;
           local_x < SIZE;
           local_x = x++) {

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



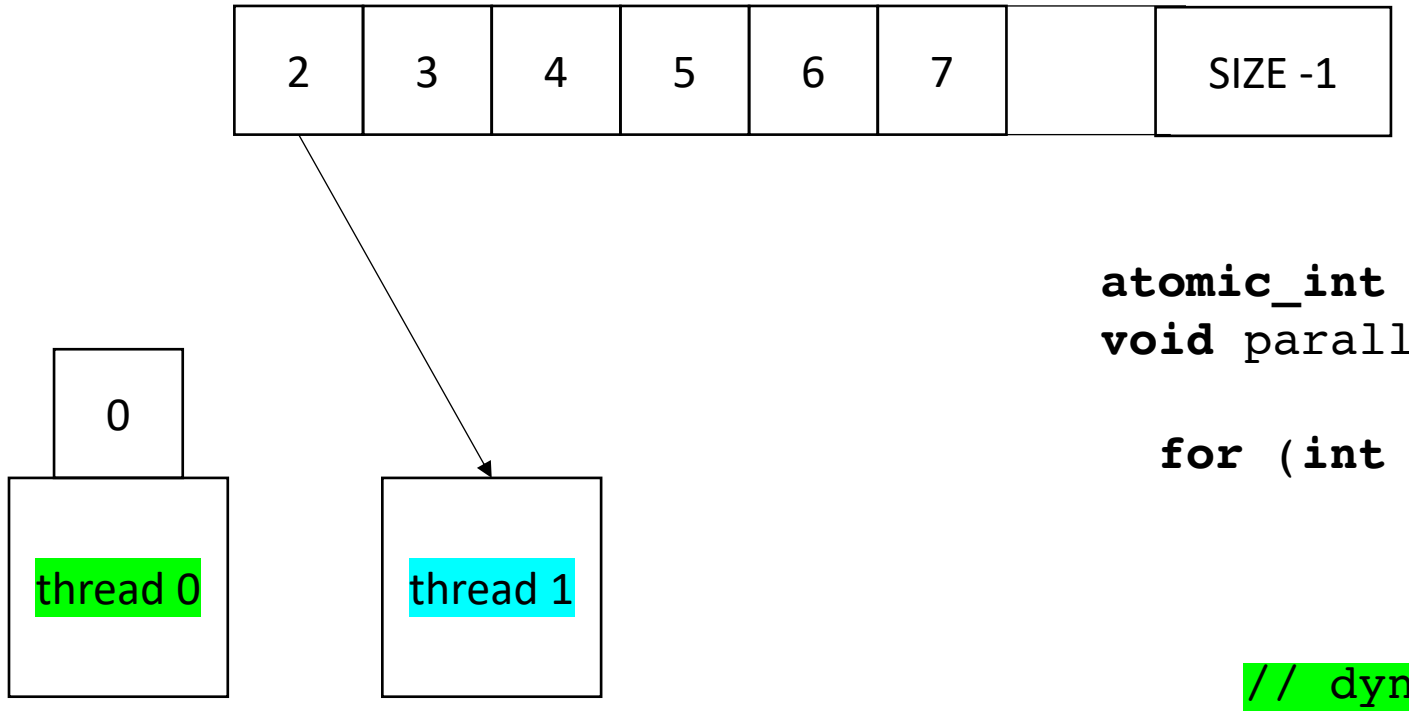

```
x: 2
0 - local_x - 0
1 - local_x - 1
```




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

  for (int local_x = x++;
            local_x < SIZE;
            local_x = x++) {

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

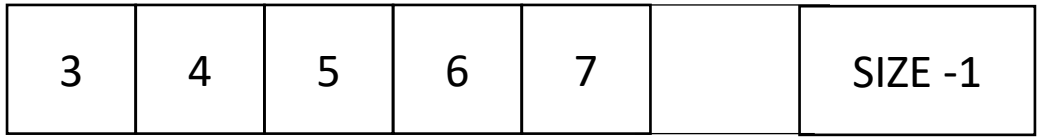



```
x: 3
0 - local_x - 0
1 - local_x - 2
```

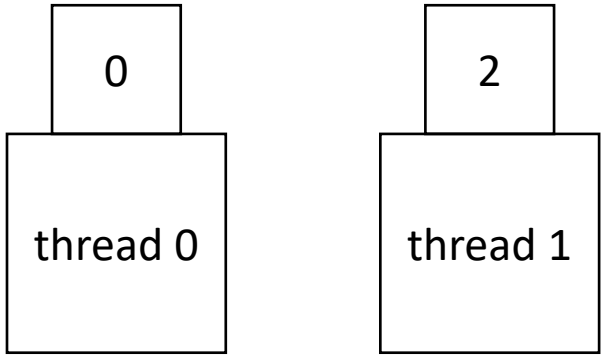
```
atomic_int x = 0;
void parallel_loop(...) {

  for (int local_x = x++;
            local_x < SIZE;
            local_x = x++) {

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



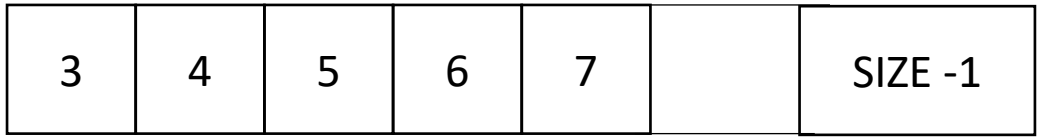

```
x: 3
0 - local_x - 0
1 - local_x - 2
```



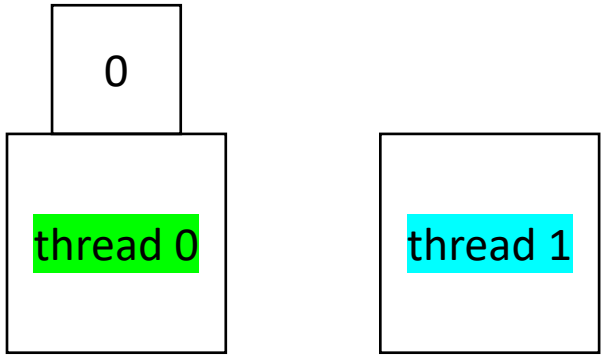

```
atomic_int x = 0;
void parallel_loop(...) {

  for (int local_x = x++;
           local_x < SIZE;
           local_x = x++) {

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



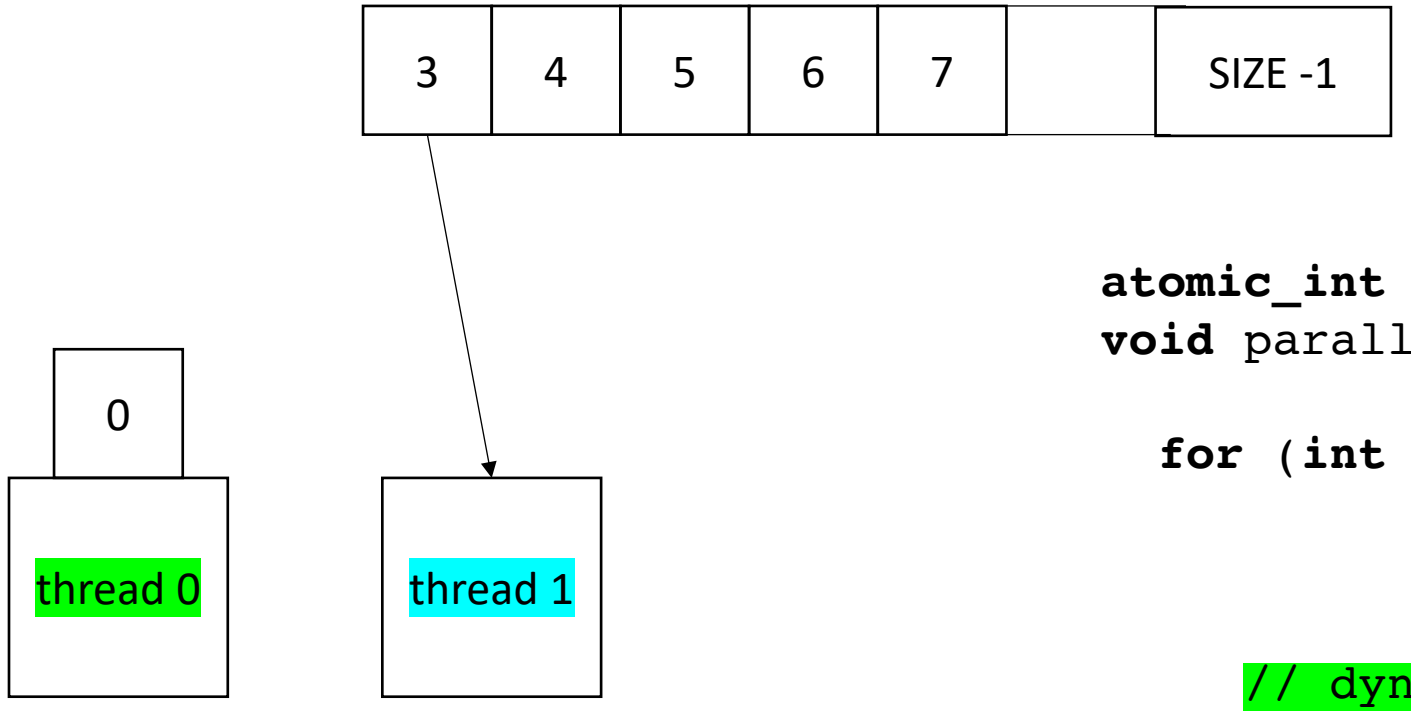

```
x: 3
0 - local_x - 0
1 - local_x - 2
```




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

  for (int local_x = x++;
            local_x < SIZE;
            local_x = x++) {

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

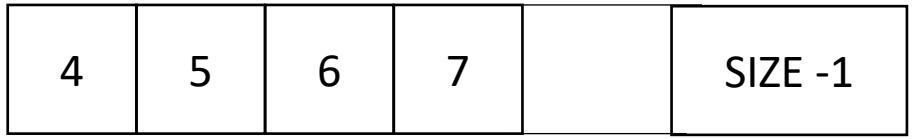



```
x: 4
0 - local_x - 0
1 - local_x - 3
```

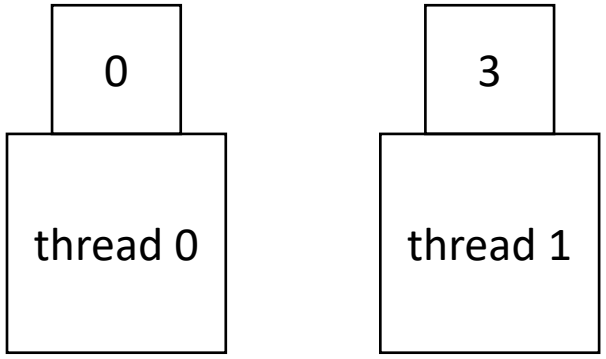
```
atomic_int x = 0;
void parallel_loop(...) {

  for (int local_x = x++;
            local_x < SIZE;
            local_x = x++) {

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



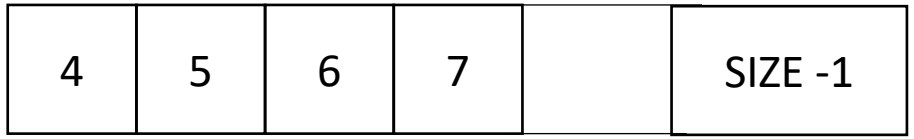

```
x: 4
0 - local_x - 0
1 - local_x - 3
```



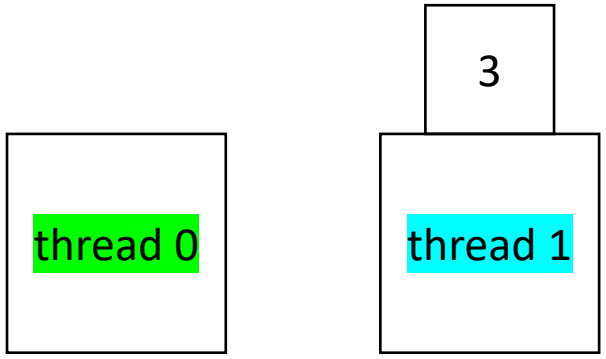

```
atomic_int x = 0;
void parallel_loop(...) {

  for (int local_x = x++;
           local_x < SIZE;
           local_x = x++) {

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



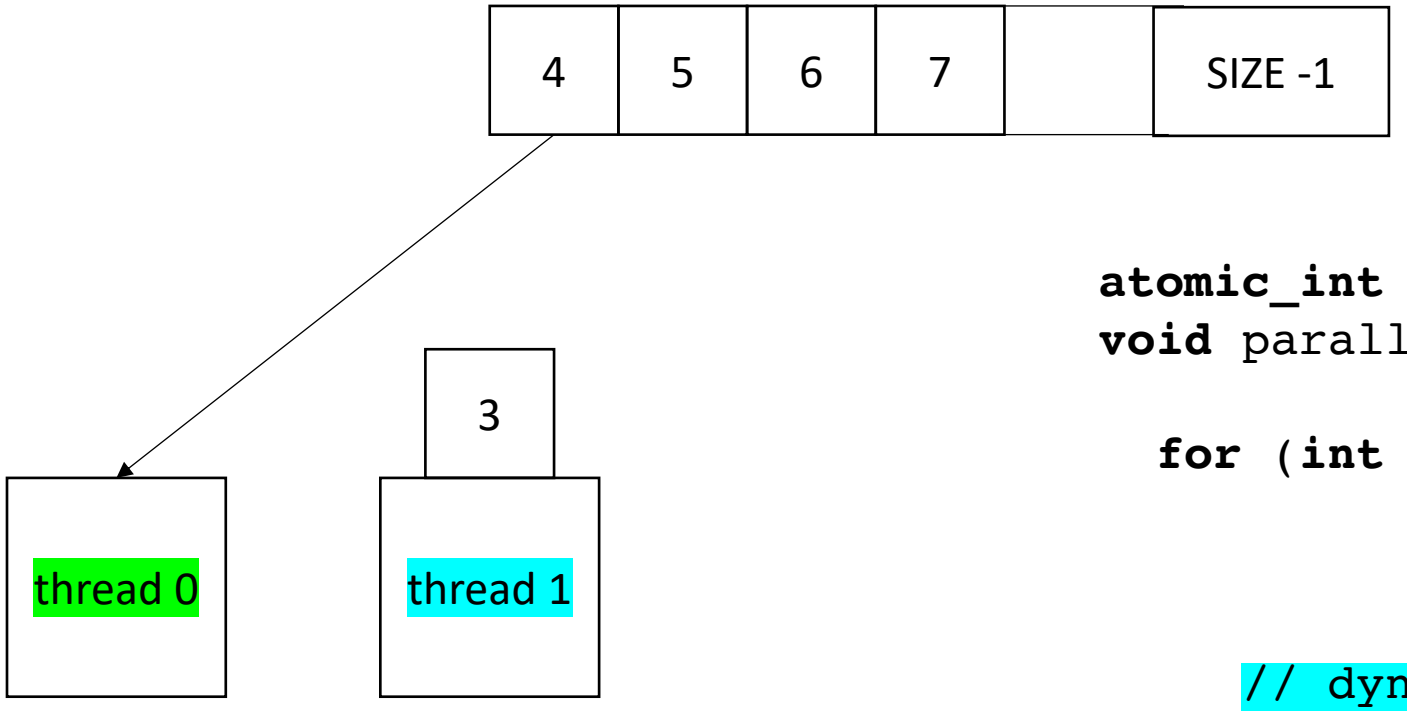

```
x: 4
0 - local_x - 0
1 - local_x - 3
```




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

  for (int local_x = x++;
            local_x < SIZE;
            local_x = x++) {

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

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

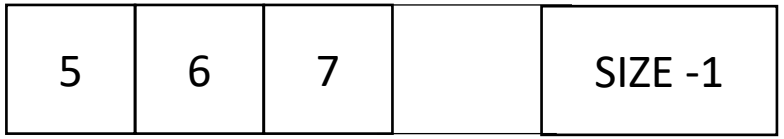



```
x: 5
0 - local_x - 4
1 - local_x - 3
```

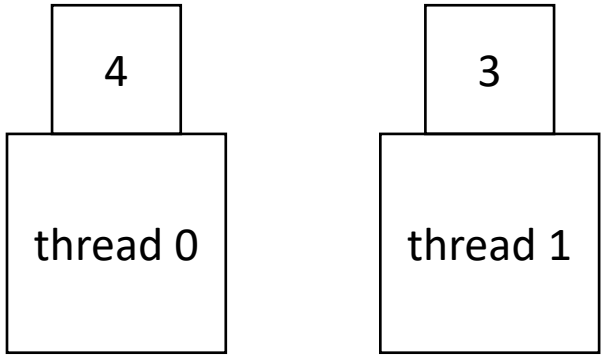
```
atomic_int x = 0;
void parallel_loop(...) {

  for (int local_x = x++;
            local_x < SIZE;
            local_x = x++) {

     // 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 = x++;
           local_x < SIZE;
           local_x = x++) {

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

End example

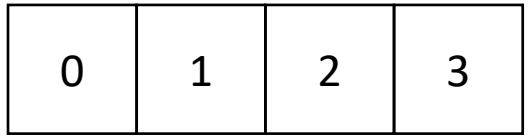
Next implementation

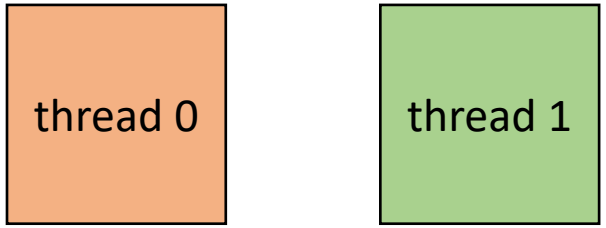
Work stealing - local worklists

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

Work stealing - local worklists

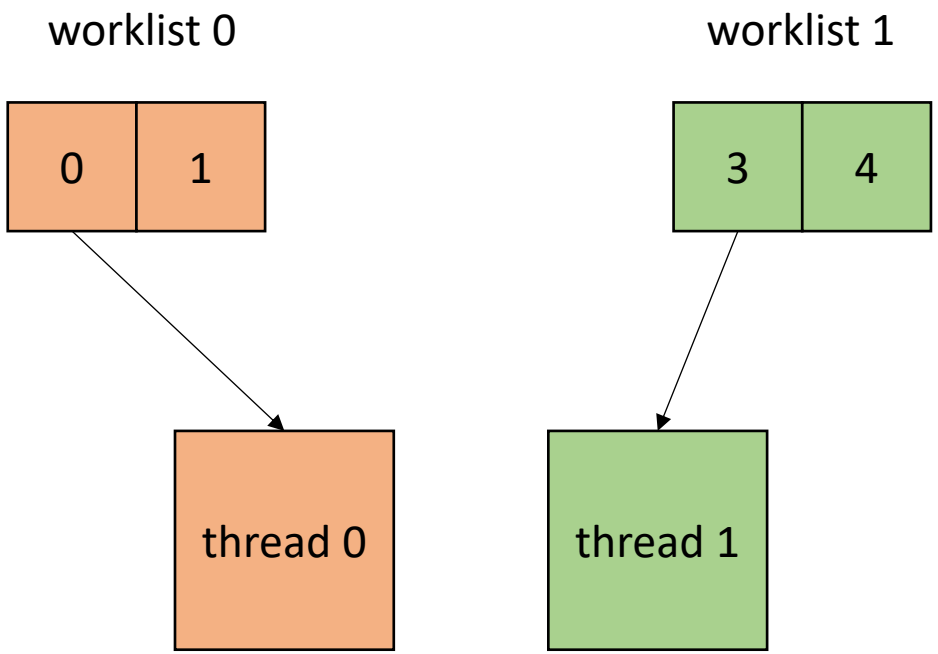
- local worklists: divide tasks into different worklists for each thread





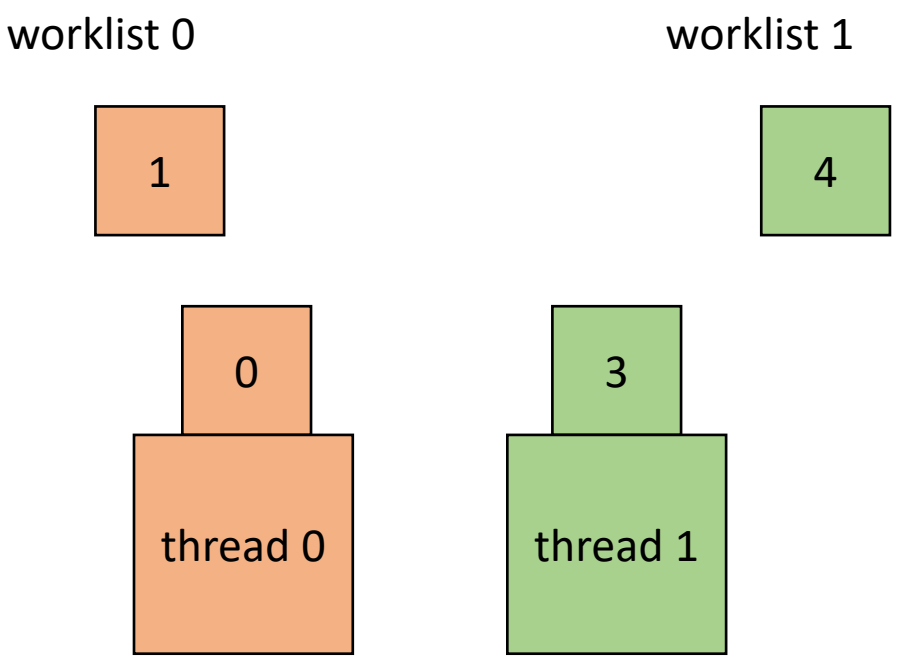
Work stealing - local worklists

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



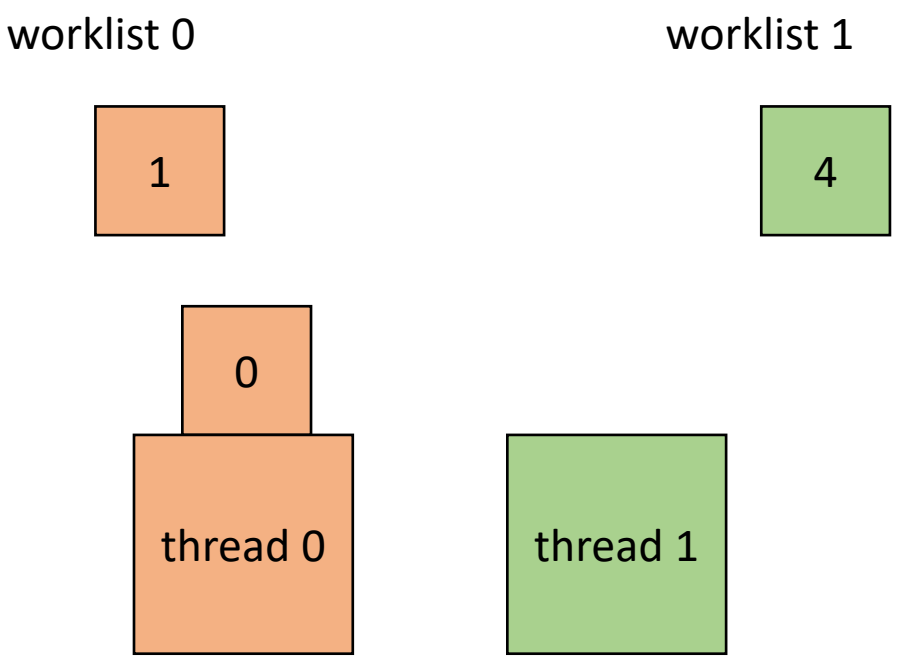
Work stealing - local worklists

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



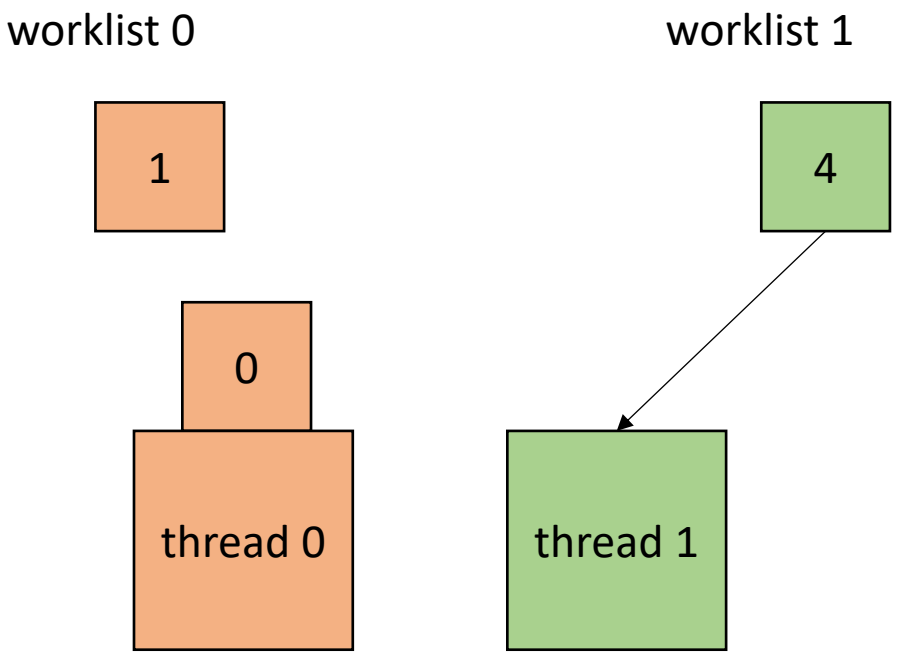
Work stealing - local worklists

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



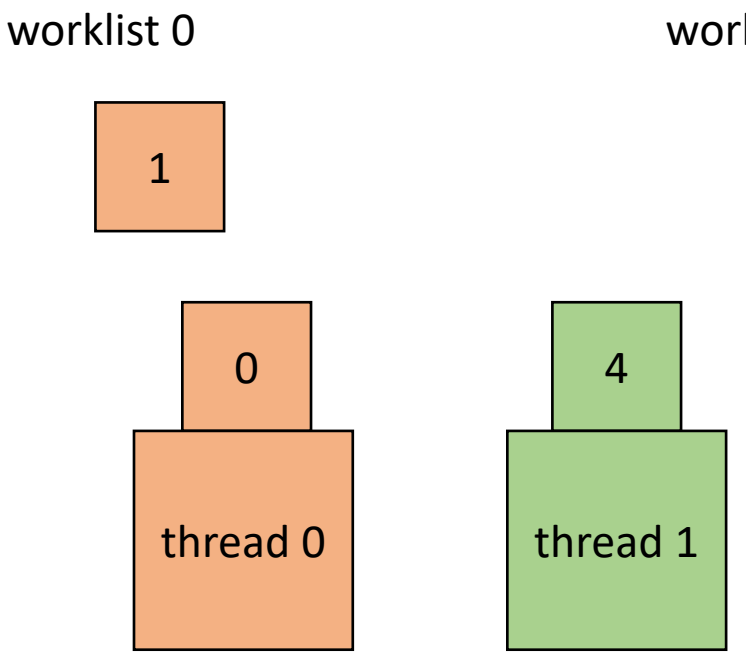
Work stealing - local worklists

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



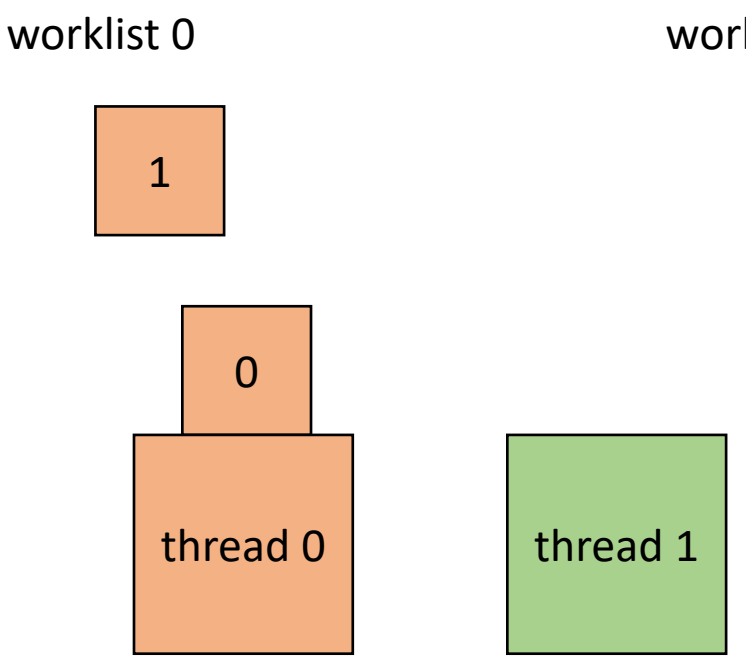
Work stealing - local worklists

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



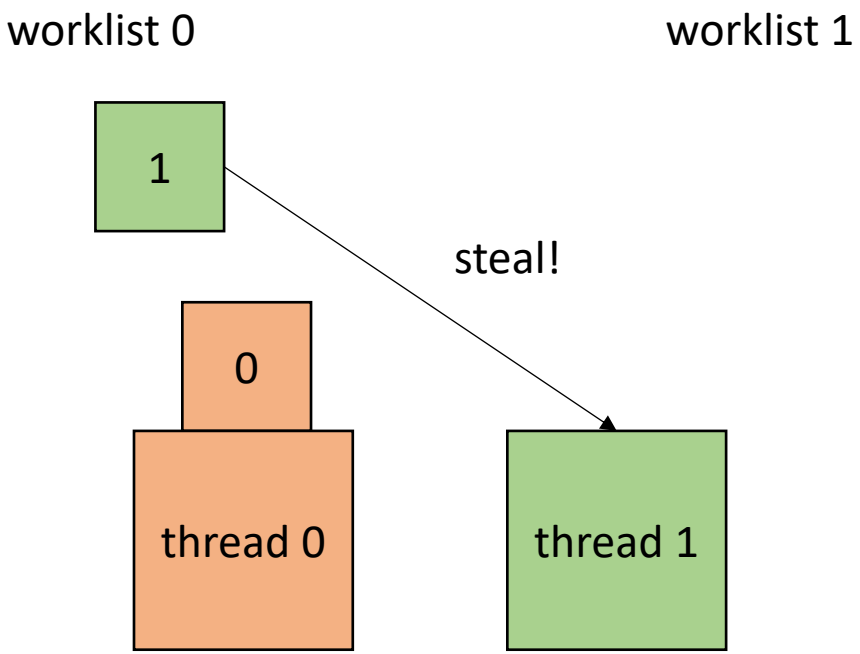
Work stealing - local worklists

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



Work stealing - local worklists

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

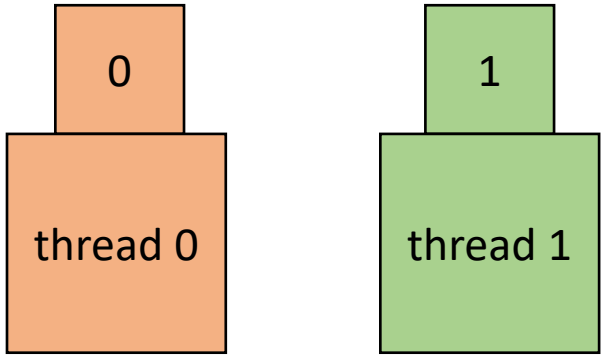


Work stealing - local worklists

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

worklist 0

worklist 1



Work stealing - local worklists

- How to implement in a compiler:

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

Work stealing - local worklists

- How to implement in a compiler:

```
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 in a compiler:

```
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 in a compiler:

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  int chunk = SIZE/NUM_THREADS;
  for (x = 0; x < SIZE; x++) {
    int tid = x / chunk;
    cq[tid].enqueue(x);
  }
  ...
}
```

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

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

initialize queues in main thread

Work stealing - local worklists

- How to implement in a compiler:

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  int chunk = SIZE/NUM_THREADS;
  for (x = 0; x < SIZE; x++) {
    int tid = x / chunk;
    cq[tid].enqueue(x);
  }
  ...
}
```

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

x	0	1	2	3
tid	0	0	1	1

initialize queues in main thread

Work stealing - local worklists

- How to implement in a compiler:

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  int chunk = ceil(SIZE/NUM_THREADS);
  for (x = 0; x < SIZE; x++) {
    int tid = x / chunk;
    cq[tid].enqueue(x);
  }
  ...
}
```

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

x	0	1	2	3

tid	0	0	1	1

initialize queues in main thread

Work stealing - local worklists

- How to implement in a compiler:

use ceiling division to make sure all work gets assigned to a valid thread

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  int chunk = ceil(SIZE/NUM_THREADS);
  for (x = 0; x < SIZE; x++) {
    int tid = x / chunk;
    cq[tid].enqueue(x);
  }
  ...
}
```

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

x	0	1	2	3
tid	0	0	1	1

initialize queues in main thread

Work stealing - local worklists

- How to implement in a compiler:

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  int chunk = ceil(SIZE/NUM_THREADS);
  for (x = 0; x < SIZE; x++) {
    int tid = x / chunk;
    cq[tid].enqueue(x);
  }
  ...
}
```

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

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

loop bounds in parallel function

Work stealing - local worklists

- How to implement in a compiler:

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  int chunk = ceil(SIZE/NUM_THREADS);
  for (x = 0; x < SIZE; x++) {
    int tid = x / chunk;
    cq[tid].enqueue(x);
  }
  ...
}
```

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

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
}
```

loop bounds in parallel function, enqueue stores result in argument, returns false if queue is empty.

Work stealing - local worklists

- How to implement in a compiler:

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  int chunk = ceil(SIZE/NUM_THREADS);
  for (x = 0; x < SIZE; x++) {
    int tid = x / chunk;
    cq[tid].enqueue(x);
  }
  ...
}
```

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

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
}
```

new global variable to track the number of threads that are finished

Work stealing - local worklists

- How to implement in a compiler:

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  int chunk = ceil(SIZE/NUM_THREADS);
  for (x = 0; x < SIZE; x++) {
    int tid = x / chunk;
    cq[tid].enqueue(x);
  }
  ...
}
```

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

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != num_threads) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Steal values from threads that are not finished

Work stealing - local worklists

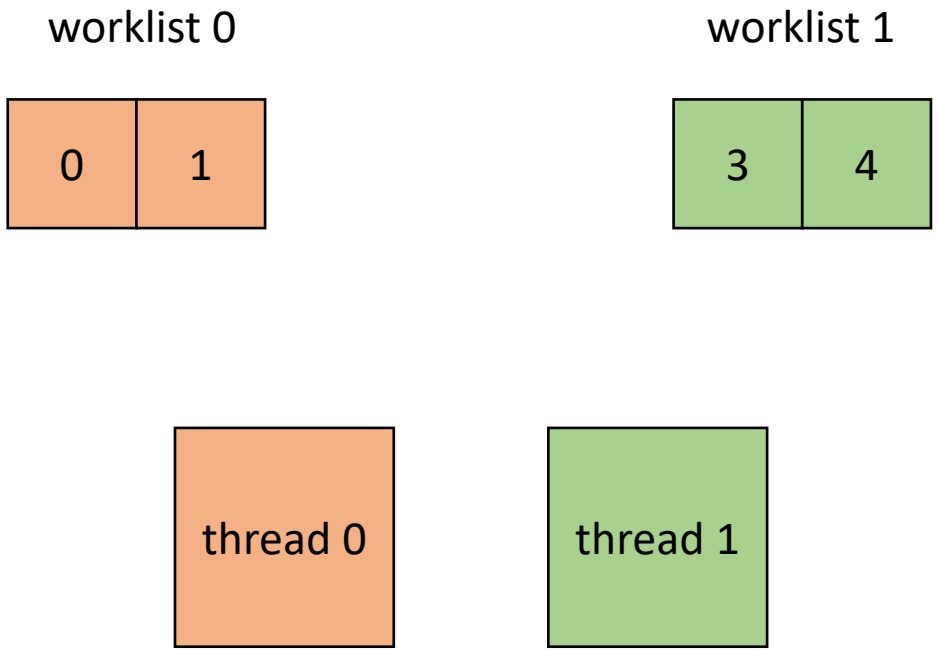
- How to implement in a compiler:

```
concurrent_queues cq[NUM_THREADS];
void foo() {
  ...
  int chunk = ceil(SIZE/NUM_THREADS);
  for (x = 0; x < SIZE; x++) {
    int tid = x / chunk;
    cq[tid].enqueue(x);
  }
  for (t = 0; t < NUM_THREADS; t++) {
     spawn(parallel_loop(..., t)
  }
  join();
  finished_threads = 0;
  ...
}
```

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

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

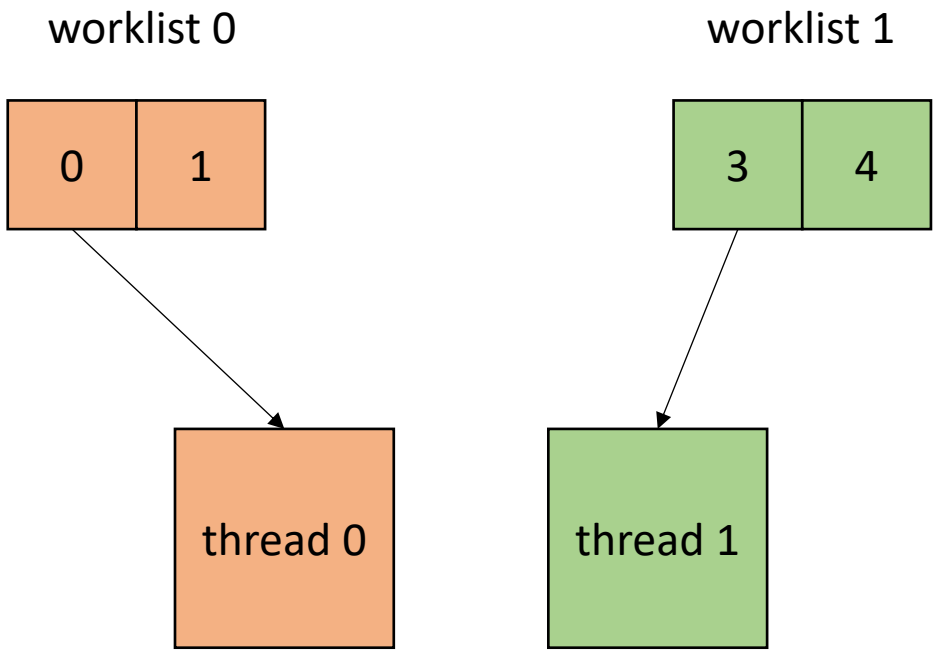
launch threads, join, reinitialize

Work stealing - local worklists



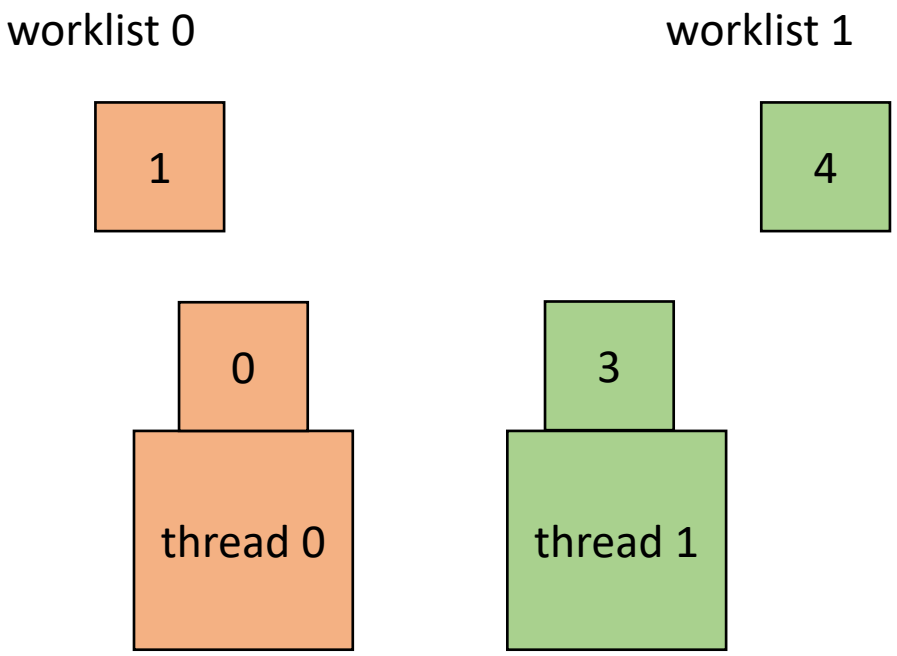

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists



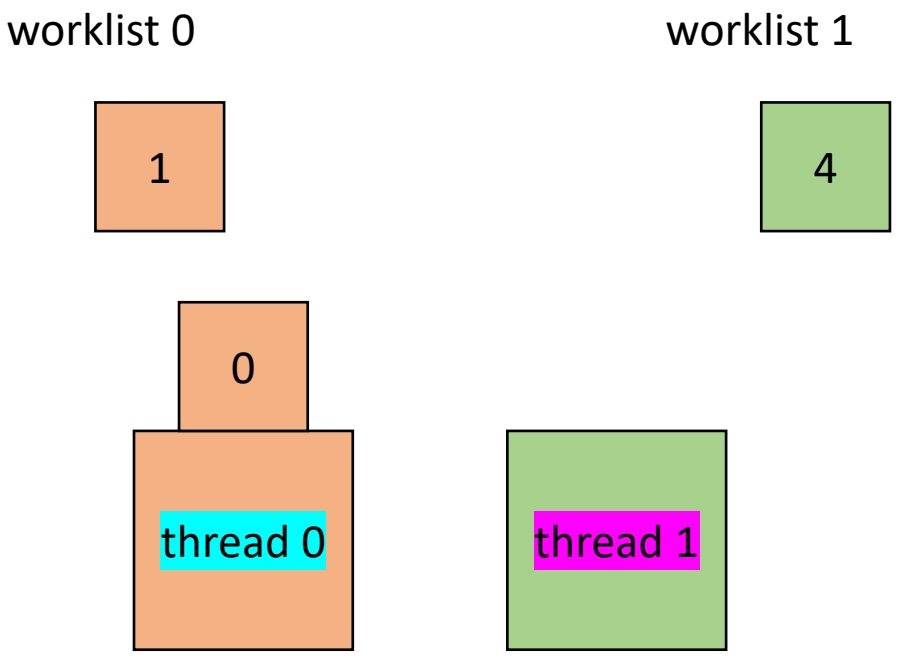

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists



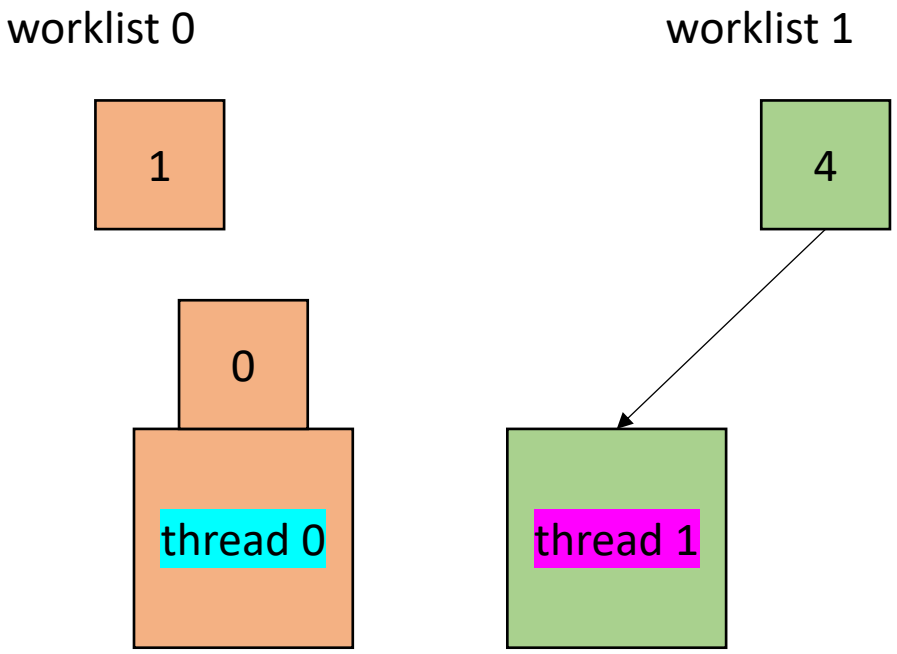

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists



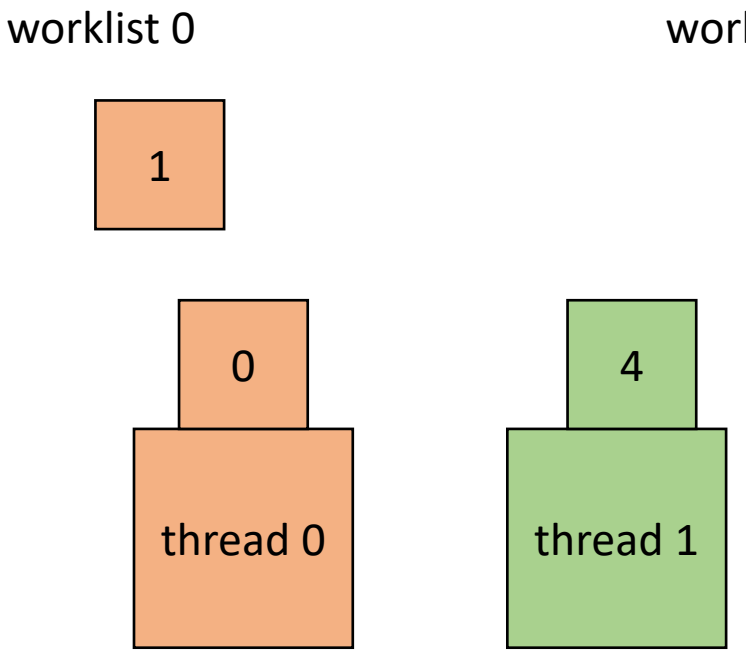

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists



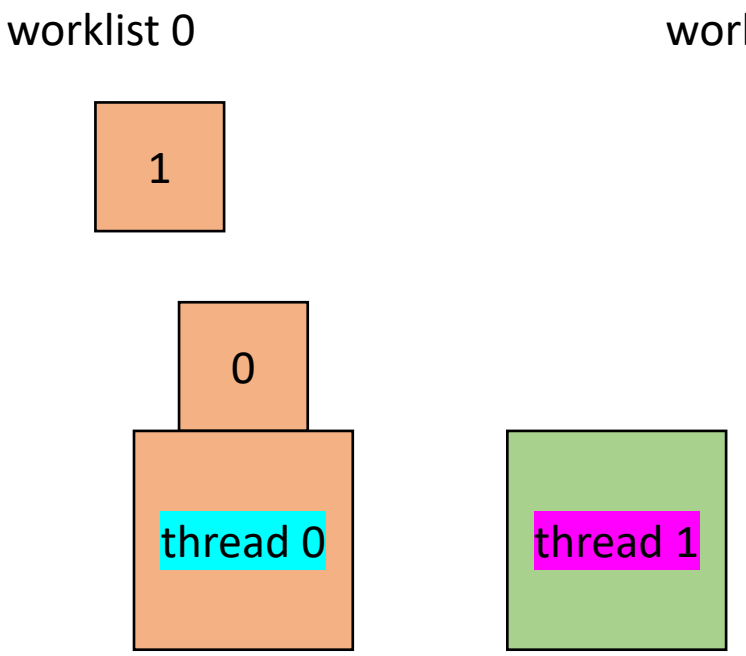

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists



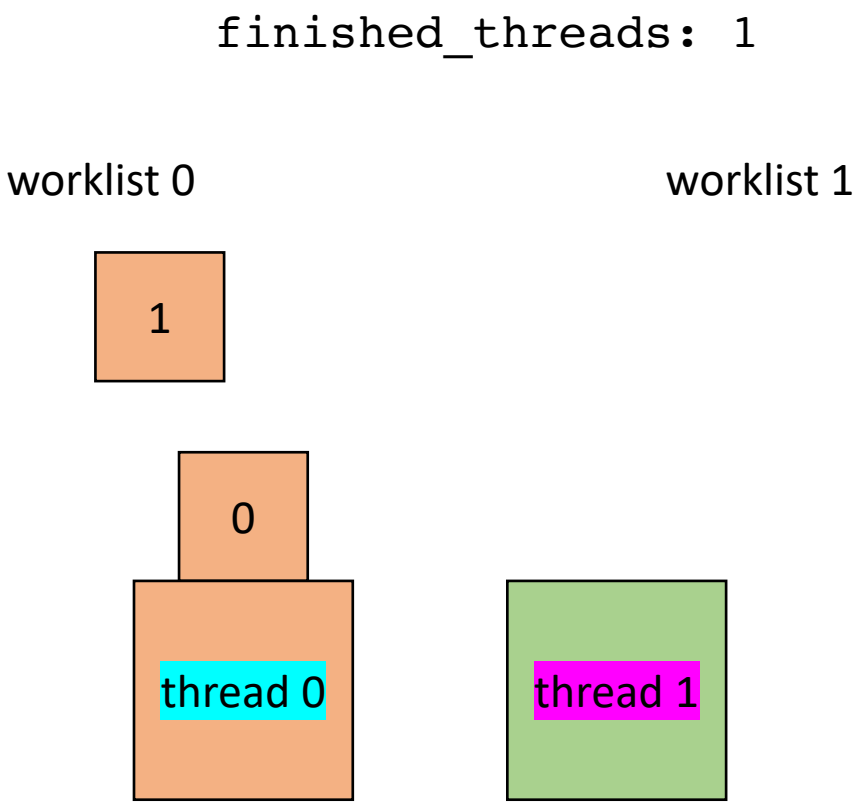

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists



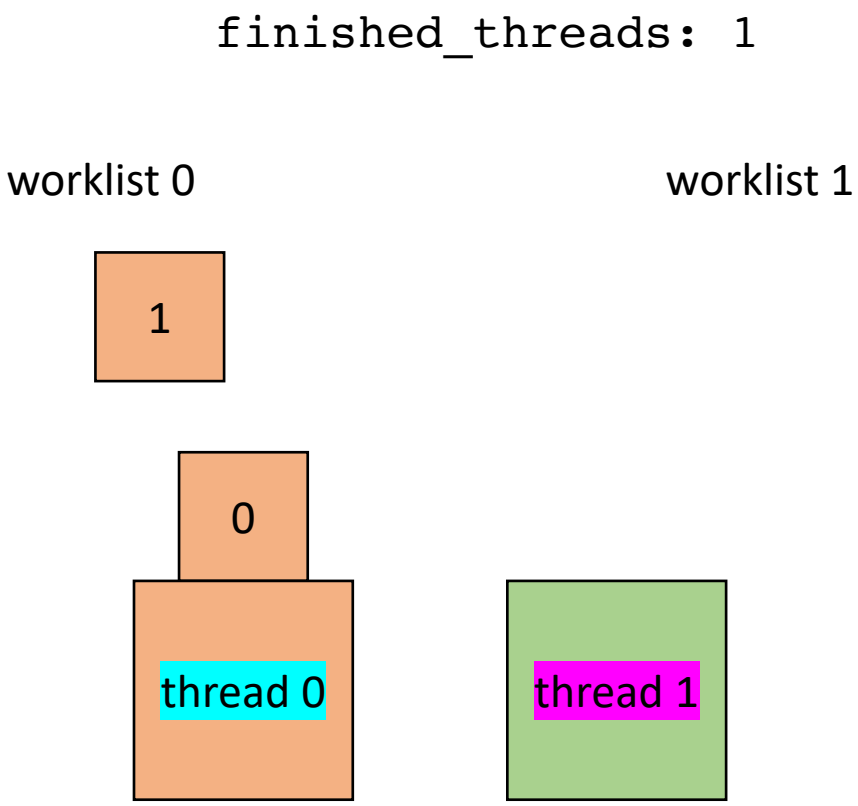

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists



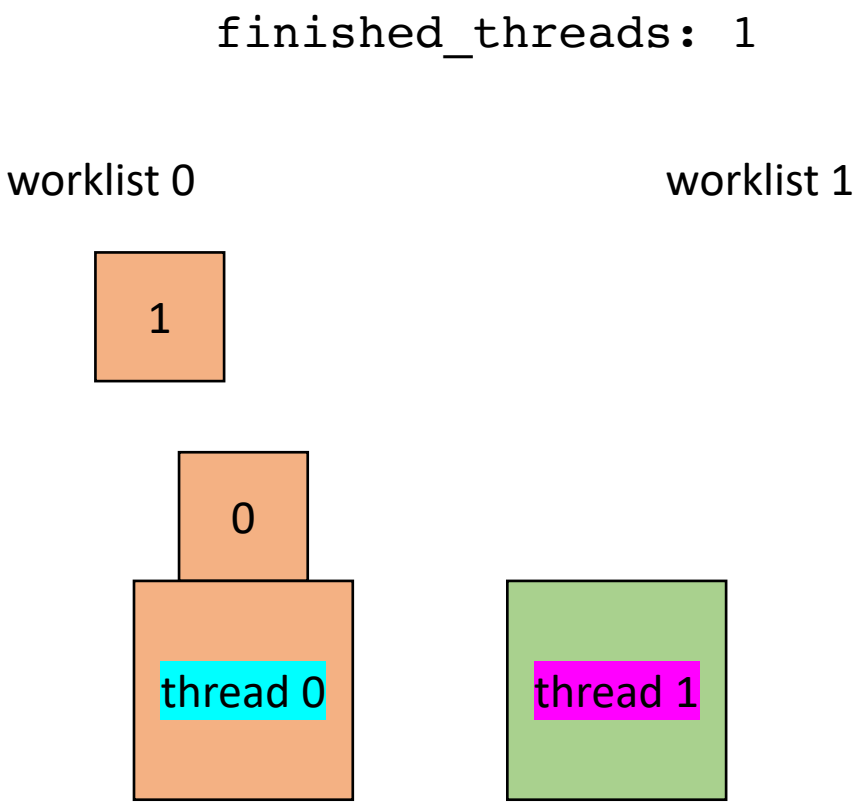

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists



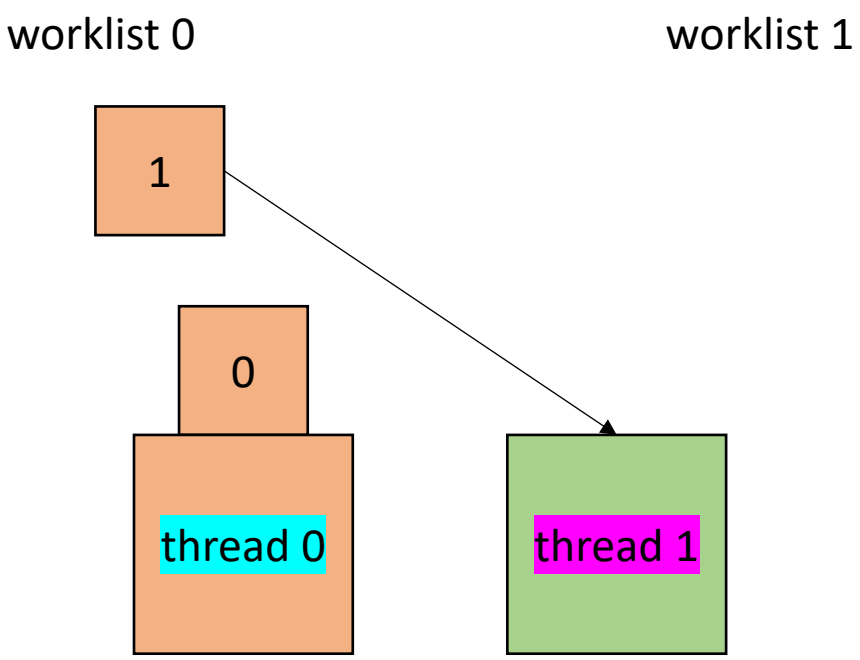

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists




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

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists



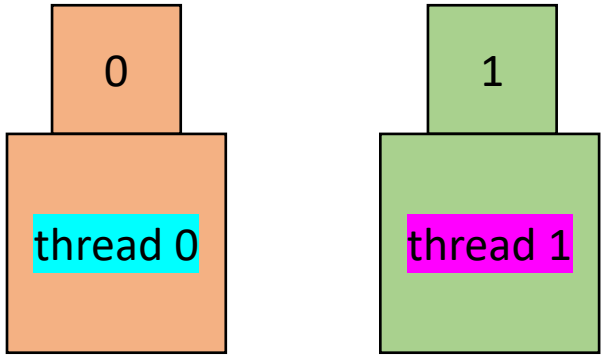

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists

finished_threads: 1

worklist 0

worklist 1



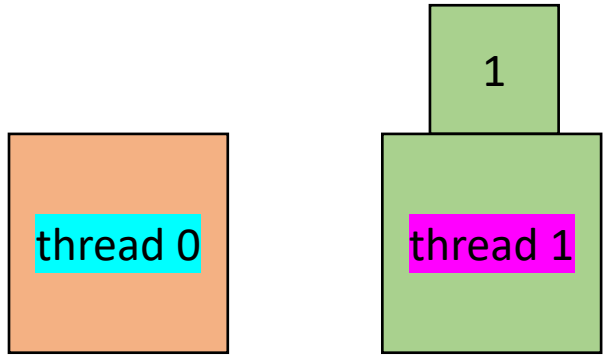

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists

finished_threads: 1

worklist 0

worklist 1



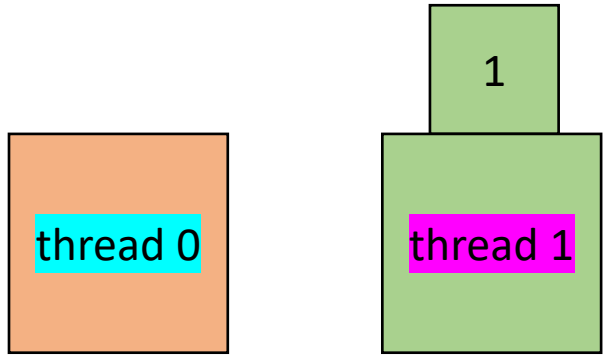

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists

finished_threads: 1

worklist 0

worklist 1



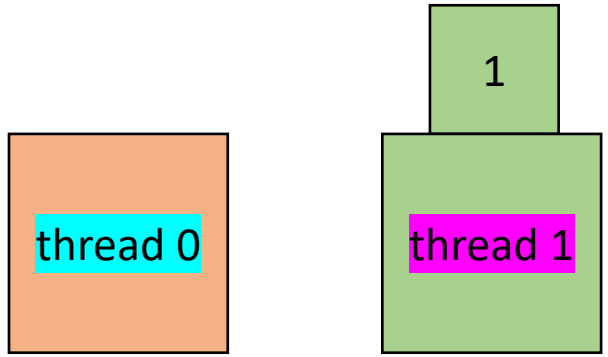

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists

finished_threads: 2

worklist 0

worklist 1



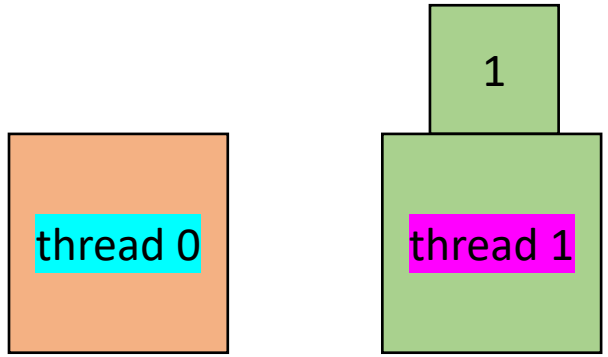

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists

finished_threads: 2

worklist 0

worklist 1



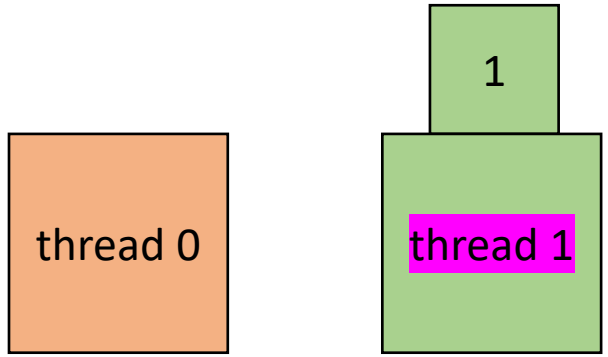

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists

finished_threads: 2

worklist 0

worklist 1



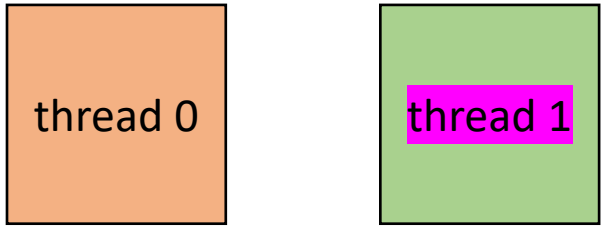

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists

finished_threads: 2

worklist 0

worklist 1



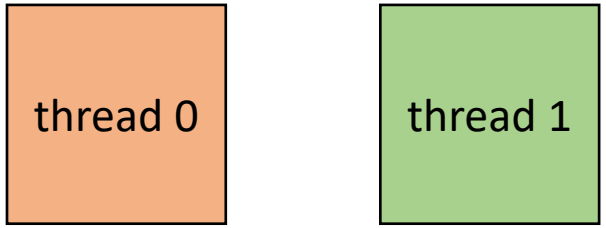

```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

  int task = 0;
  while (cq[tid].dequeue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].dequeue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists

finished_threads: 2

worklist 0

worklist 1




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

  int task = 0;
  while (cq[tid].enqueue(&task)) {
    // dynamic work based on task
  }
  finished_threads++;
  while (finished_threads != NUM_THREADS) {
    target = //select a random thread
    if (cq[target].enqueue(&task))
        // dynamic work based on task
  }
}
```

Work stealing - local worklists

- How to implement in a compiler:

```
concurrent_queues cq[NUM_THREADS];
void foo() {
   ...
   int chunk = ceil(SIZE/NUM_THREADS);
   for (x = 0; x < SIZE; x++) {
      int tid = x / chunk;
      cq[tid].enqueue(x);
   }
   for (t = 0; t < NUM_THREADS; t++) {
      spawn(parallel_loop(..., t)
   }
   join();
   finished_threads = 0;
   ...
}
```

Final note: initializing the worklists may become a bottleneck. Amdahl's law

Work stealing - local worklists

- How to implement in a compiler:

```
concurrent_queues cq[NUM_THREADS];
void foo() {
   ...
   int chunk = ceil(SIZE/NUM_THREADS);
   for (x = 0; x < SIZE; x++) {
      int tid = x / chunk;
      cq[tid].enqueue(x);
   }
   for (t = 0; t < NUM_THREADS; t++) {
      spawn(parallel_loop(..., t)
   }
   join();
   finished_threads = 0;
   ...
}
```

Final note: initializing the worklists may become a bottleneck. e.g. Amdahl's law

Can be made parallel using regular parallelism constructs

Summary

- Many ways to parallelize DOALL loops
  - Independent iterations are key to giving us this freedom!
- Some are more complicated than others.
  - Local worklists require concurrent data structures
  - Global worklist requires read-modify-write
- Compiler implementation can enable rapid exploration and experimentation.

Next week

- Guest lecture about types

- This will take us through the Thanksgiving break
  - Paper and project proposals will be due!