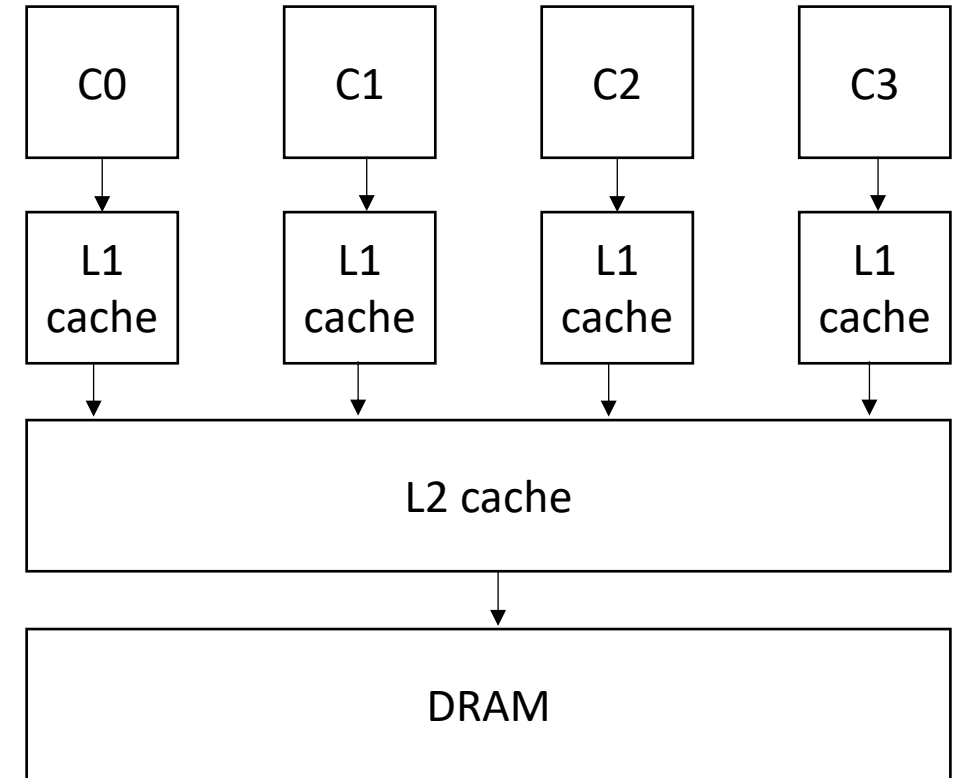


CSE211: Compiler Design

Nov. 3, 2022

- **Topic:** Making loops parallel
 - Equal amounts of work
 - Unequal sizes of work
- **Discussion questions:**
 - *How do you deal with load imbalance in workloads?*



Announcements

- Back to in person!

Announcements

- Homework 2 was due yesterday
 - hopefully you all got it in. If not, let me know ASAP!
- Putting the final touches on HW 1 grades, thanks for being patient!
- Homework 3 will be released on Monday or Tuesday. It will be about parallelism.

Announcements

- Midterm
 - due tomorrow (Friday Nov. 4) by midnight
 - Rules:
 - Open book, open internet, open notes.
 - Do not discuss the test with any other student while it is out.
 - Do not google (or otherwise search) for exact questions. It is fine to search for concepts.
 - Do not post questions to others on the internet (e.g. through discord or reddit)
 - Any question should be asked as a private post on Piazza. If it's a clarification that needs to be made to the whole class, I will do it in a public Piazza thread

Announcements

- Midterm
 - Designed to take about 2 hours (not including studying)
 - Students report taking longer because they study while taking the test.
 - Students also report taking longer because they double check their answers and make the test nicely formatted.
 - Please look over the test as soon as it is released so that you roughly know how long it will take you.
- LATE MIDTERMS WILL NOT BE ACCEPTED

Announcements

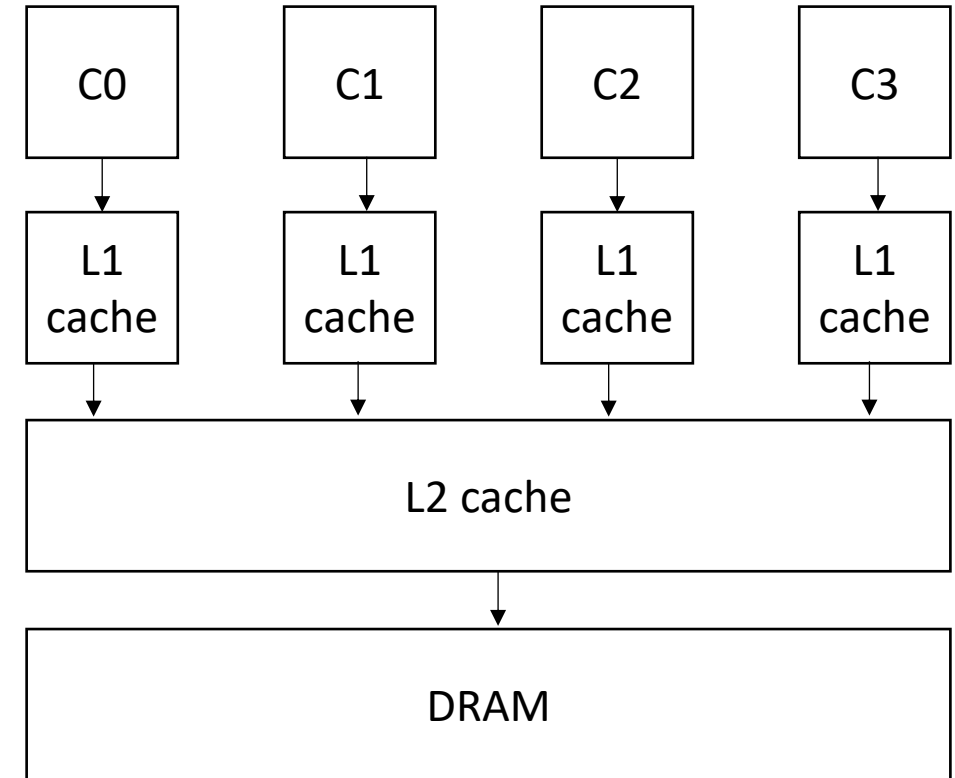
- Mark your attendance for today after you watch the recording (or if you are attending live)
 - Please try to keep on top of this.
 - We have more attendance put in, please let us know within 1 week if there are any issues

Review SMP parallelism

CSE211: Compiler Design

Nov. 3, 2022

- **Topic:** Making loops parallel
 - Equal amounts of work
 - Unequal sizes of work
- **Discussion questions:**
 - *How do you deal with load imbalance in workloads?*



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 compile down to, or link to C/C++: DNN libraries, Graph analytic DSLs, Numpy.

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 $SIZE / NUM_THREADS = 4$

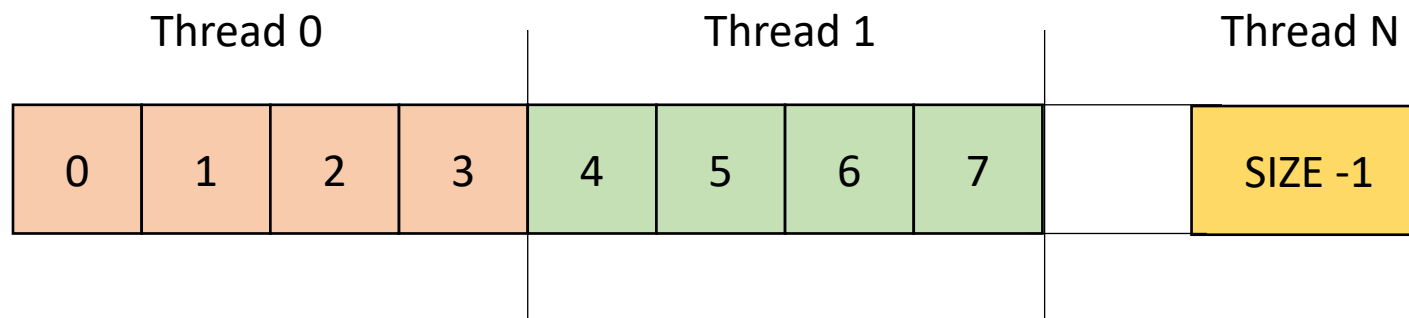


Regular Parallel Loops

- 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  
  }  
}
```

Regular Parallel Loops

- How to implement in a compiler:

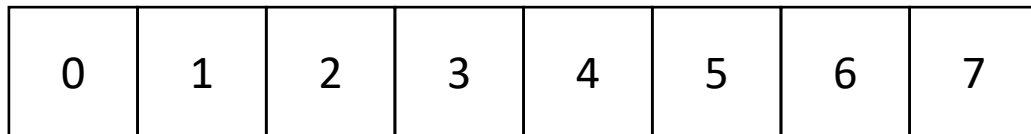
```
void foo() {  
    ...  
    for (int t = 0; t < NUM_THREADS; t++) {  
        spawn(parallel_loop(..., t))  
    }  
    join();  
    ...  
}
```

Spawn threads

```
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

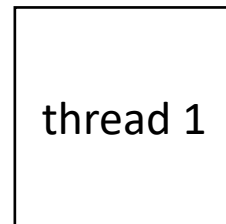
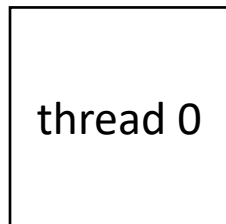
- Example, 2 threads/cores, array of size 8



chunk_size = ?

0: start=? 1: start=?

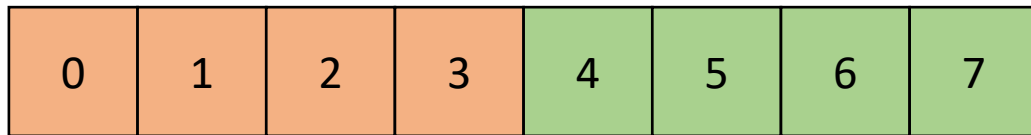
0: end=? 1: end=?



```
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

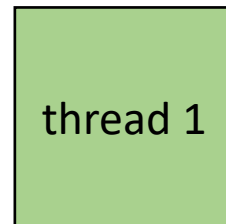
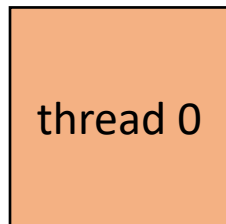
- Example, 2 threads/cores, array of size 8



`chunk_size = 4`

0: start = 0 1: start = 4

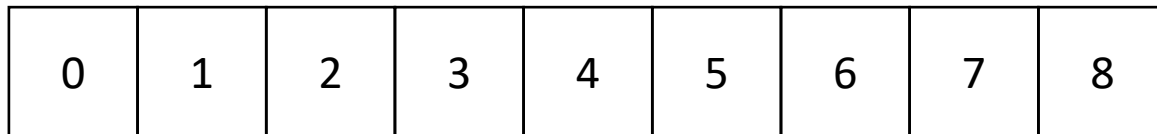
0: end = 4 1: end = 8



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

Regular Parallel Loops

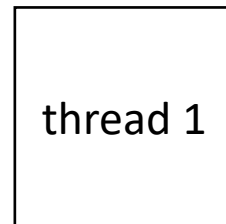
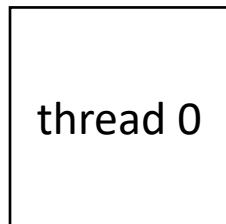
- Example, 2 threads/cores, array of size 9



chunk_size = ?

0: start= ? 1: start= ?

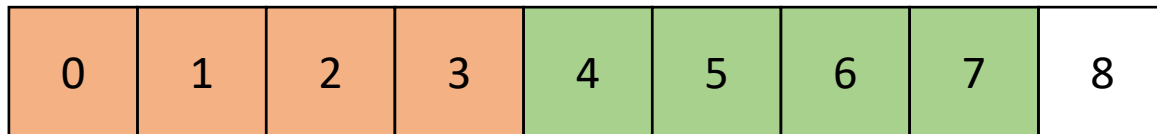
0: end= ? 1: end= ?



```
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

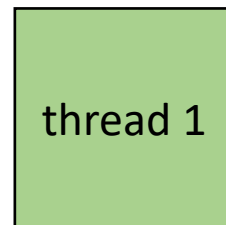
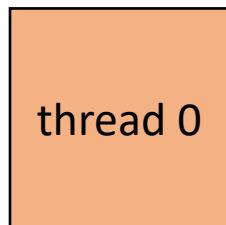
- Example, 2 threads/cores, array of size 9



chunk_size = 4

0: start = 0 1: start = 4

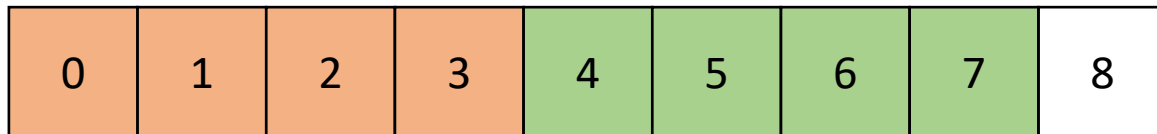
0: end = 4 1: end = 8



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

Regular Parallel Loops

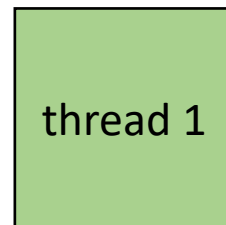
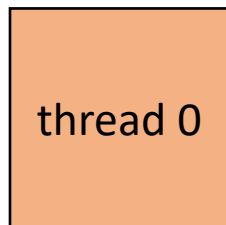
- Example, 2 threads/cores, array of size 9



chunk_size = 4

0: start = 0 1: start = 4

0: end = 4 1: end = 8

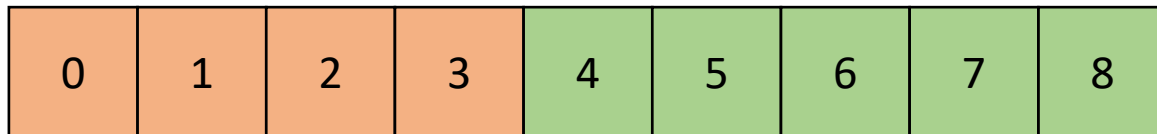


```
void parallel_loop(..., int tid) {  
  
    int chunk_size = SIZE / NUM_THREADS;  
    int start = chunk_size * tid;  
    int end = start + chunk_size;  
    if (tid == NUM_THREADS - 1) {  
        end = (SIZE);  
    }  
    for (x = start; x < end; x++) {  
        // work based on x  
    } 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

Regular Parallel Loops

last thread gets more work

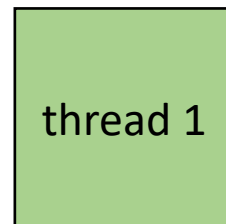
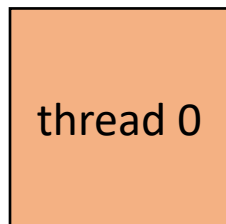
- Example, 2 threads/cores, array of size 9



chunk_size = 4

0: start = 0 1: start = 4

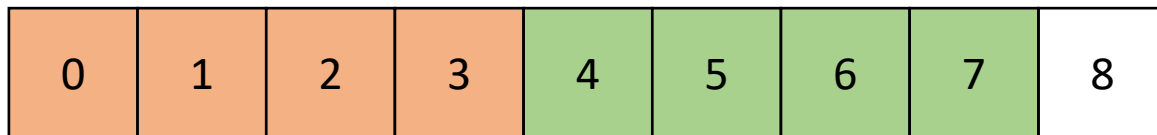
0: end = 4 1: end = 9



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

Regular Parallel Loops

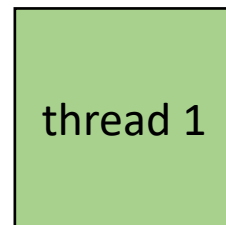
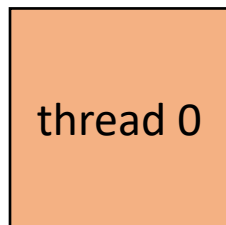
- Example, 2 threads/cores, array of size 9



`chunk_size = 4`

0: start = 0 1: start = 4

0: end = 4 1: end = 8



ceiling division

```
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

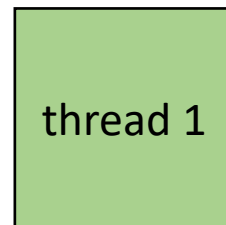
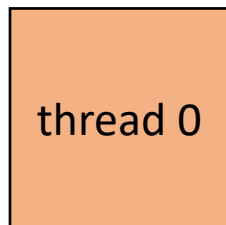


out of bounds

`chunk_size = 5`

0: start = 0 1: start = 5

0: end = 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 = start + chunk_size  
    for (x = start; x < end; x++) {  
        // work based on x  
    }  
}
```

Regular Parallel Loops

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

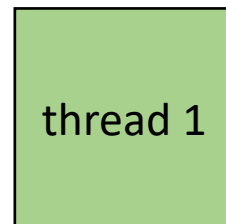
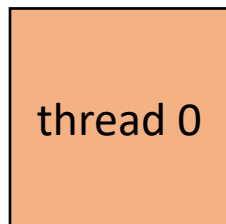


out of bounds

`chunk_size = 5`

0: start = 0 1: start = 5

0: end = 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
```

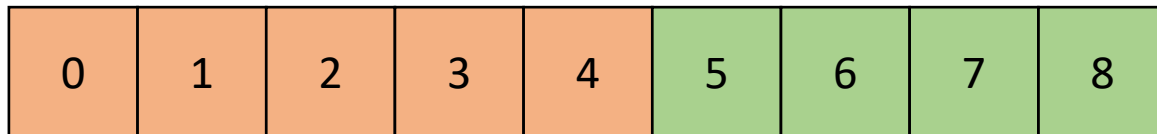
```
    }
```

```
}
```


Regular Parallel Loops

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

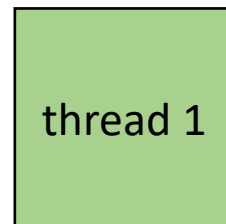
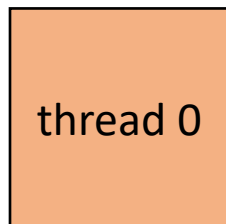
most threads do equal amounts of work, last thread may do less.



chunk_size = 5

0: start = 0 1: start = 5

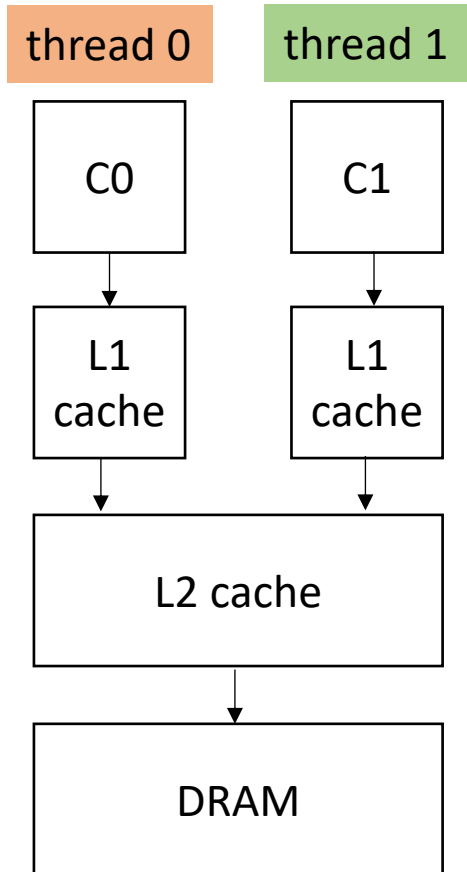
0: end = 5 1: end = 9



```
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  
    }  
}
```

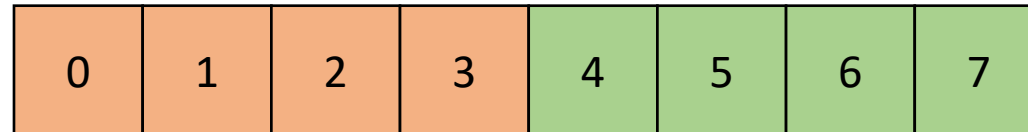
Good for SMP parallelism

SMP parallelism



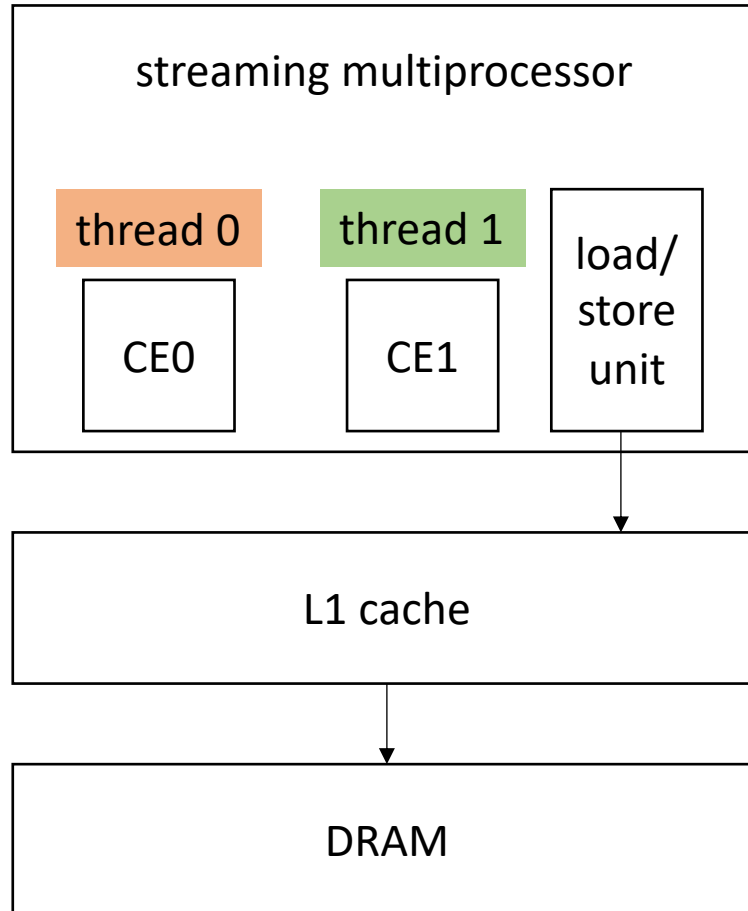
stays in thread 0's
L1 cache

stays in thread 1's
L1 cache



What about streaming multiprocessors (GPUs)?

one streaming multiprocessor contains many small Compute Elements (CE)



CEs Can load adjacent memory locations simultaneously.

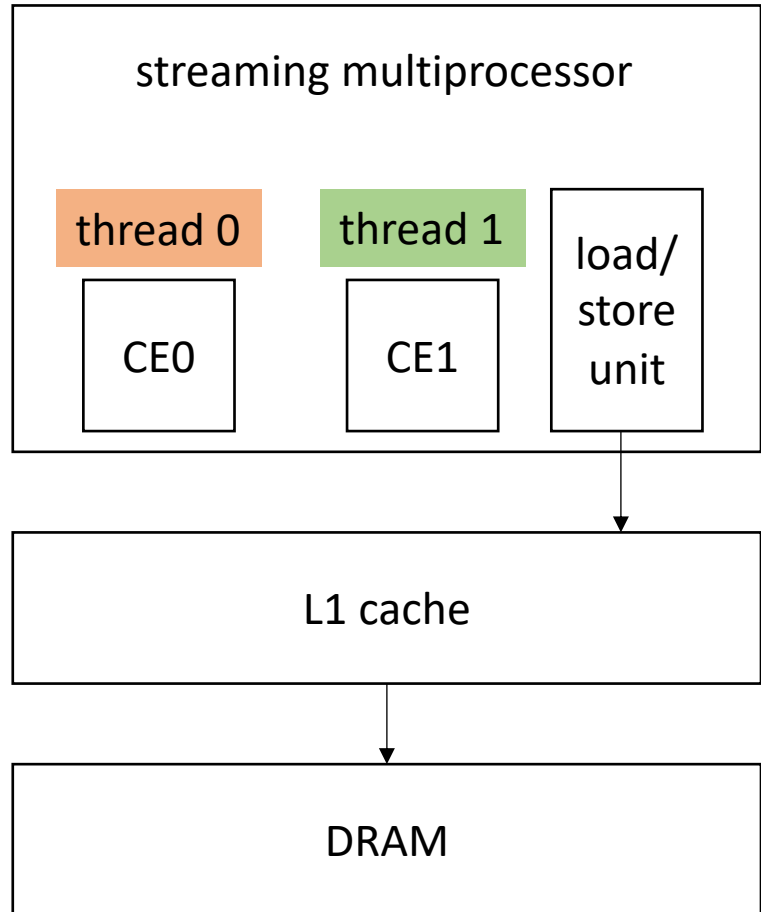
CEs execute iterations synchronously

is this partition good for GPUs??



What about streaming multiprocessors (GPUs)?

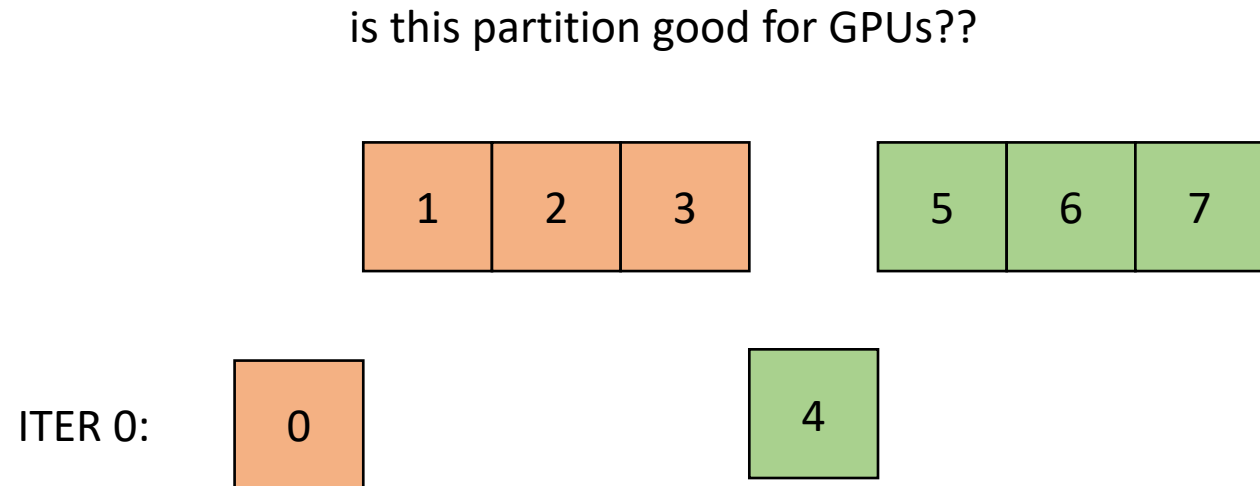
one streaming multiprocessor contains many small Compute Elements (CE)



CEs Can load adjacent memory locations simultaneously.

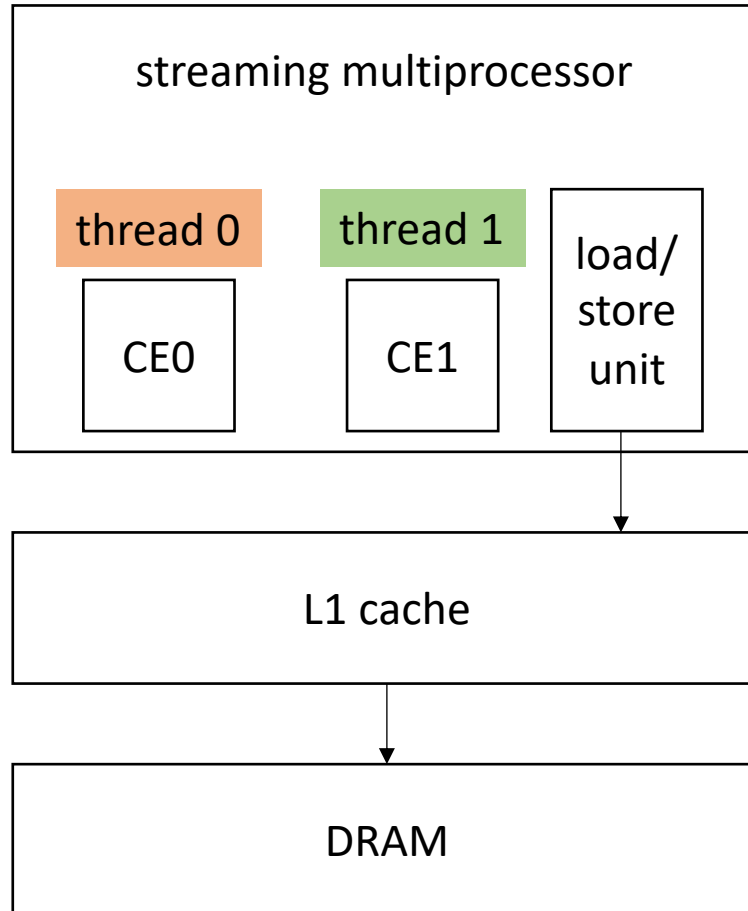
CEs execute iterations synchronously

is this partition good for GPUs??



What about streaming multiprocessors (GPUs)?

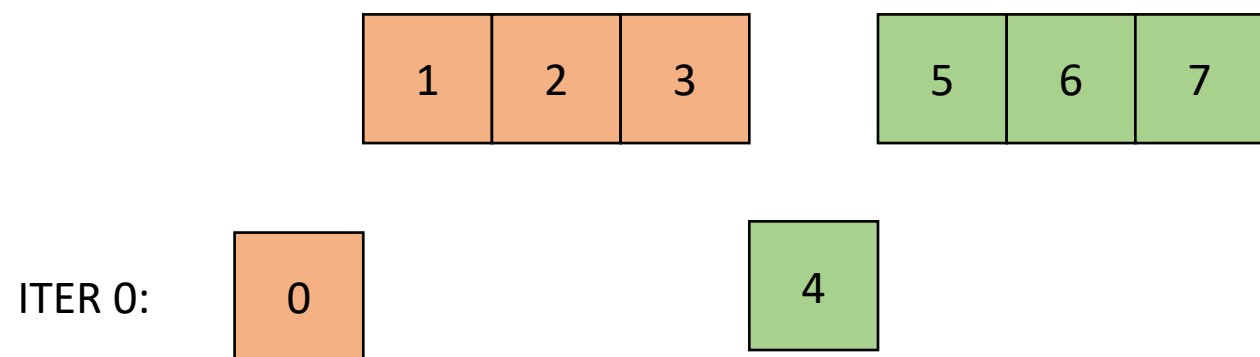
one streaming multiprocessor contains many small Compute Elements (CE)



CEs Can load adjacent memory locations simultaneously.

CEs execute iterations synchronously

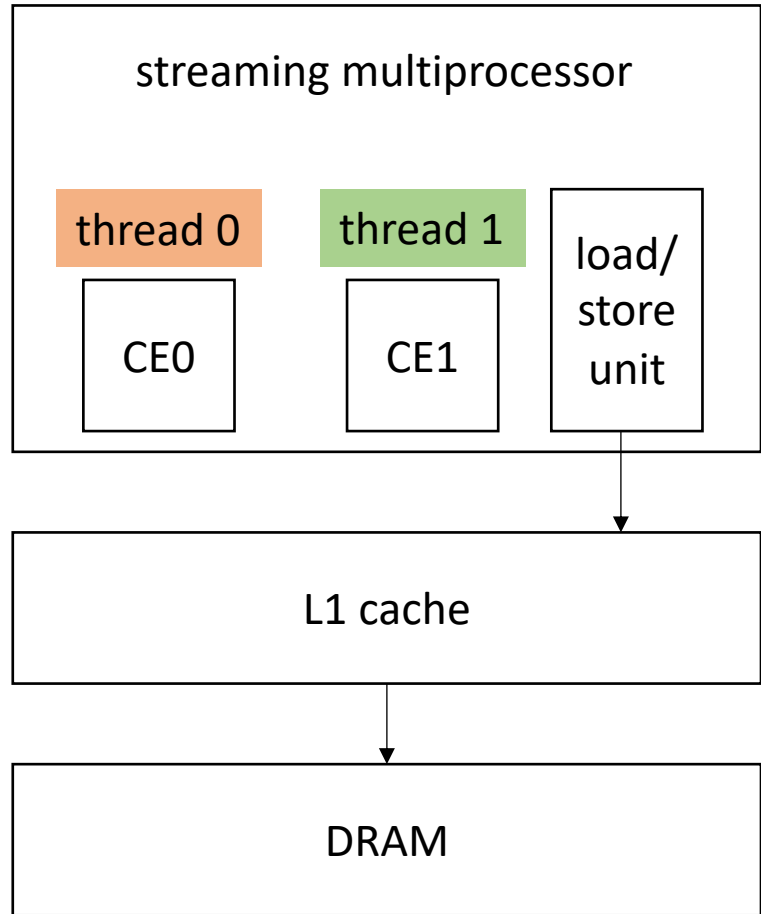
is this partition good for GPUs??



not adjacent, so the loads have to be serialized

What about streaming multiprocessors (GPUs)?

one streaming multiprocessor contains many small Compute Elements (CE)



ITER 0:

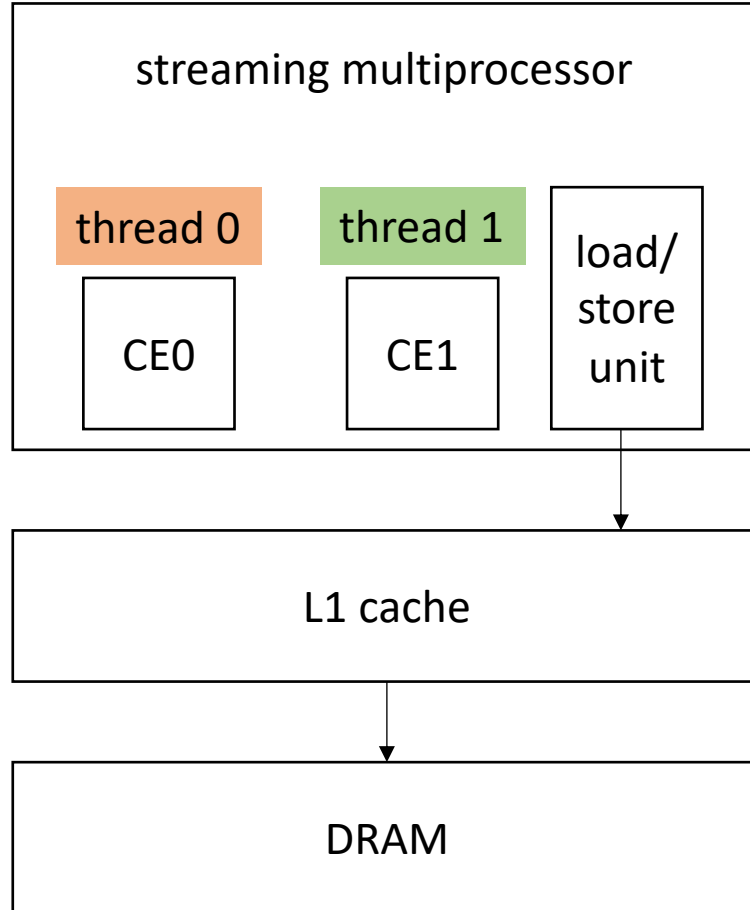
What about a striped pattern?



CEs Can load adjacent memory locations simultaneously

What about streaming multiprocessors (GPUs)?

one streaming multiprocessor contains many small Compute Elements (CE)

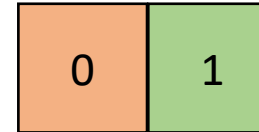


CEs Can load adjacent memory locations simultaneously

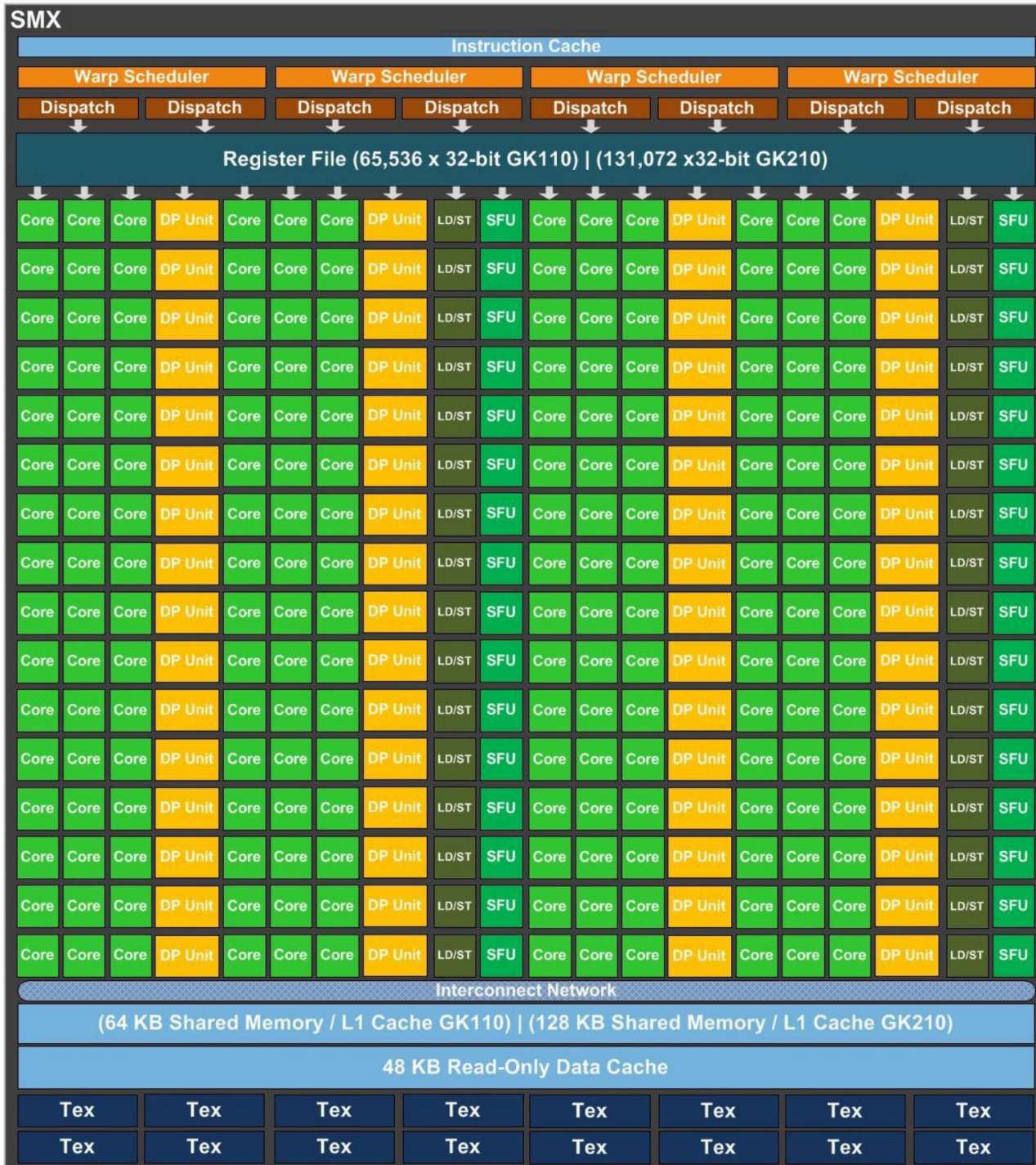
What about a striped pattern?



ITER 0:



adjacent memory locations can be loaded at the same time!

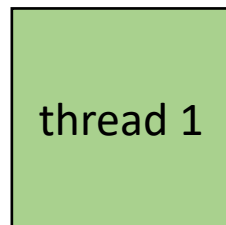
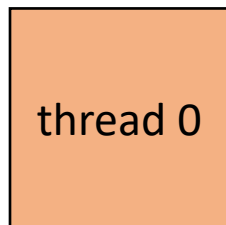
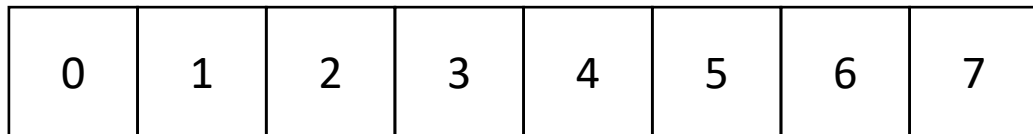


Kepler architecture

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?

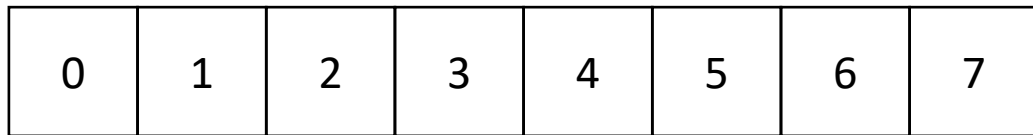
- Example, 2 threads/cores, array of size 8.
Change code for a GPU



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

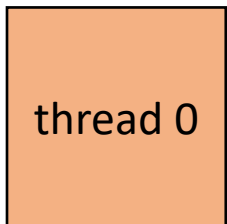
How to compiler for GPUs?

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

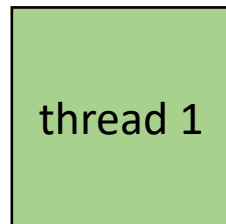


ITER 0

x: 0



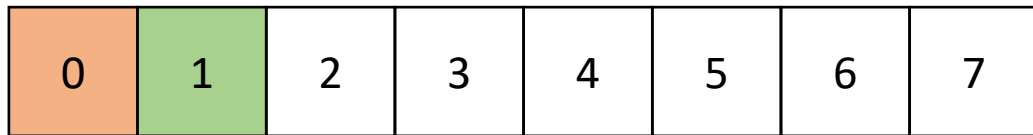
x: 1



```
void parallel_loop(..., int tid) {  
  
    for (x = tid; x < SIZE; x+=NUM_THREADS) {  
        // work based on x  
    }  
}
```

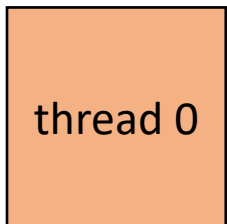
How to compiler for GPUs?

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

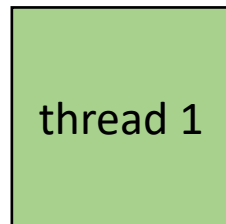


ITER 0

x: 2



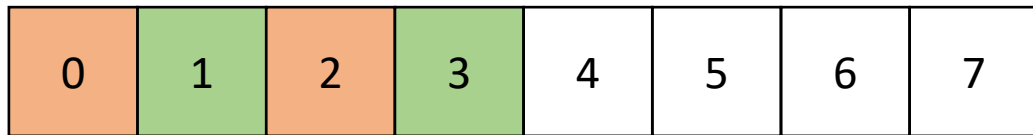
x: 3



```
void parallel_loop(..., int tid) {  
  
    for (x = tid; x < SIZE; x+=NUM_THREADS) {  
        // work based on x  
    }  
}
```

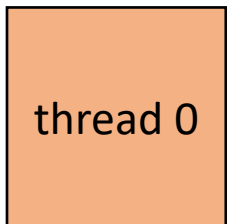
How to compiler for GPUs?

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

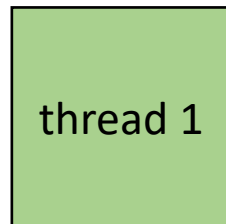


ITER 1

x: 2



x: 3



```
void parallel_loop(..., int tid) {  
  
    for (x = tid; x < SIZE; x+=NUM_THREADS) {  
        // work based on x  
    }  
}
```

Demo

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

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

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

example: regular (or embarrassingly)
parallelism:
each x iteration performs the same
amount of work

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 underutilized.

```
for (x = 0; x < SIZE; x++) {  
    for (y = 0; y < x; 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 $0 - \text{SIZE}/2$
 - Thread 2 takes iterations $\text{SIZE}/2 - \text{SIZE}$

```
for (x = 0; x < SIZE; x++) {  
    for (y = 0; y < x; 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 $0 - \text{SIZE}/2$
 - Thread 2 takes iterations $\text{SIZE}/2 - \text{SIZE}$

Calculate how much total work:

$$\text{total_work} = \sum_{n=0}^{\text{SIZE}} n$$

```
for (x = 0; x < SIZE; x++) {  
    for (y = 0; y < x; 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 $0 - \text{SIZE}/2$
 - Thread 2 takes iterations $\text{SIZE}/2 - \text{SIZE}$

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

Calculate how much total work:

$$\text{total_work} = \sum_{n=0}^{\text{SIZE}} n$$

Calculate work done by first thread:

$$\text{t1_work} = \sum_{n=0}^{\text{SIZE}/2} n$$

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

Calculate how much total work:

$$\text{total_work} = \sum_{n=0}^{\text{SIZE}} n$$

Calculate work done by second thread:

$$t1_work = \sum_{n=0}^{\text{SIZE}/2} n$$

Calculate work done by first thread:

$$t2_work = \text{total_work} - t1_work$$

Irregular parallelism in loops

Example: SIZE = 64

total_work = 2016

t1_work = 496

t2_work = 1520

t2 does ~3x more work than t1

Only provides ~1.3x speedup

Potential solution:

Have T1 do only ¼ of the iterations

Gives a better speedup of 1.77x

Doesn't always work as loop bounds are not always statically known!

Calculate how much total work:

$$\text{total_work} = \sum_{n=0}^{\text{SIZE}} n$$

Calculate work done by second thread:

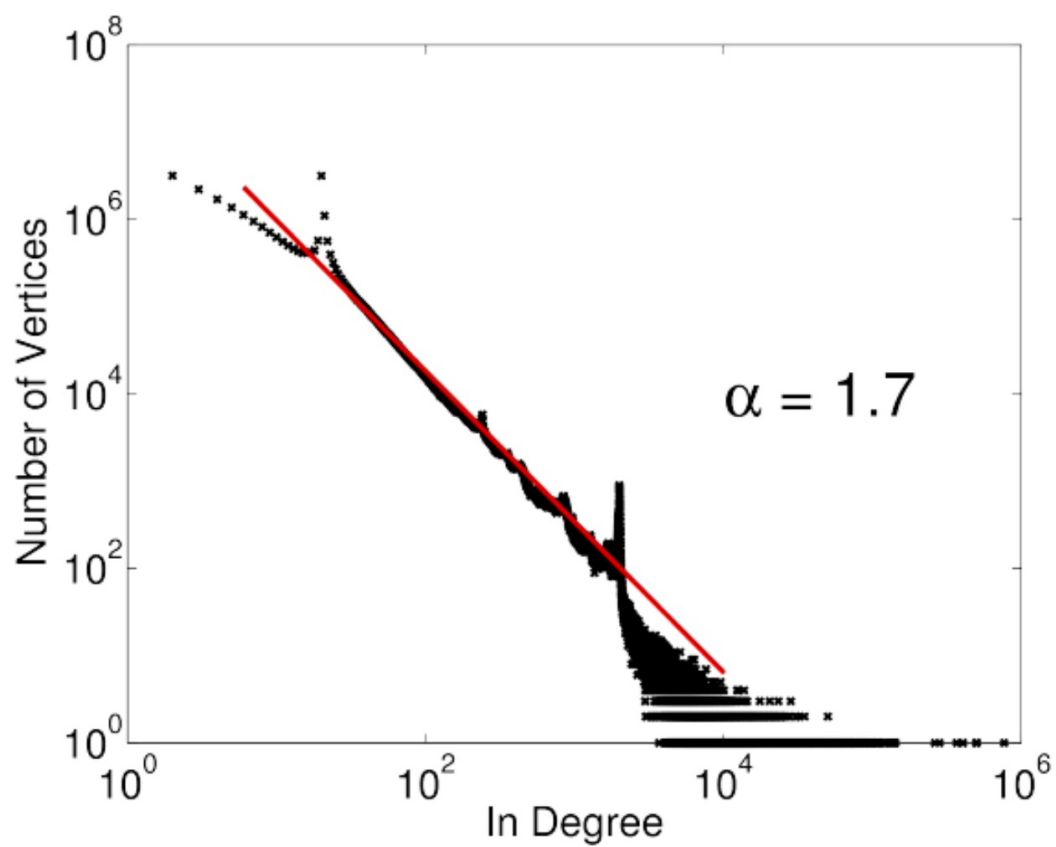
$$\text{t1_work} = \sum_{n=0}^{\text{SIZE}/2} n$$

Calculate work done by first thread:

$$\text{t2_work} = \text{total_work} - \text{t1_work}$$

Demo

Where does irregular parallelism show up?



(a) Twitter In-Degree

from "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012

Tyler Sorensen
@Tyler_UCSC

Computer Science Researcher
Assistant Professor at UC Santa Cruz in 2020

📍 Santa Cruz, CA 🌐 users.soe.ucsc.edu/~tsorensen/
📅 Joined September 2019

303 Following 332 Followers

Lindsey Kuper
@lindsey Follows you

Mommy; blogger. CS professor at @ucsc (PL, distributed systems, verification).
"Permit yourself to open a book and start reading from anywhere." pronoun.is/she

🌐 composition.al 📅 Joined October 2006

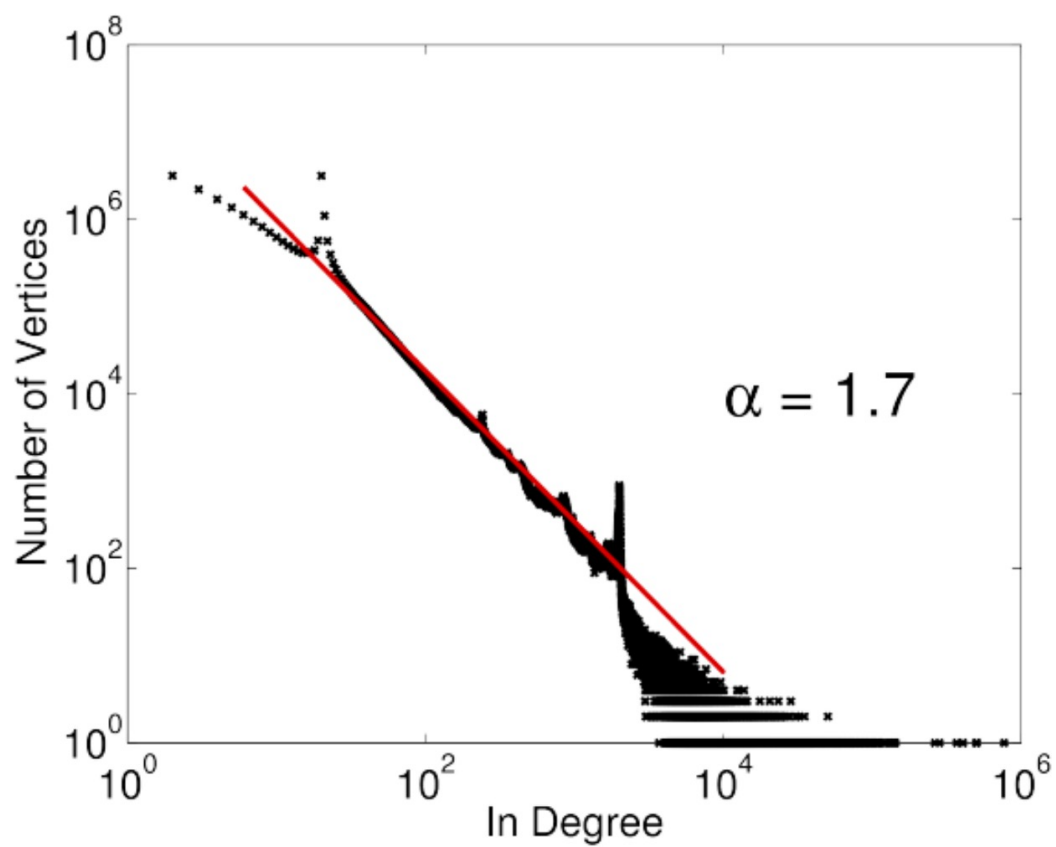
751 Following 6,840 Followers

Barack Obama ✓
@BarackObama

Dad, husband, President, citizen.

📍 Washington, DC 🌐 obama.org 🗓️ Born August 4, 1961
📅 Joined March 2007

598.7K Following 126.2M Followers



(a) Twitter In-Degree

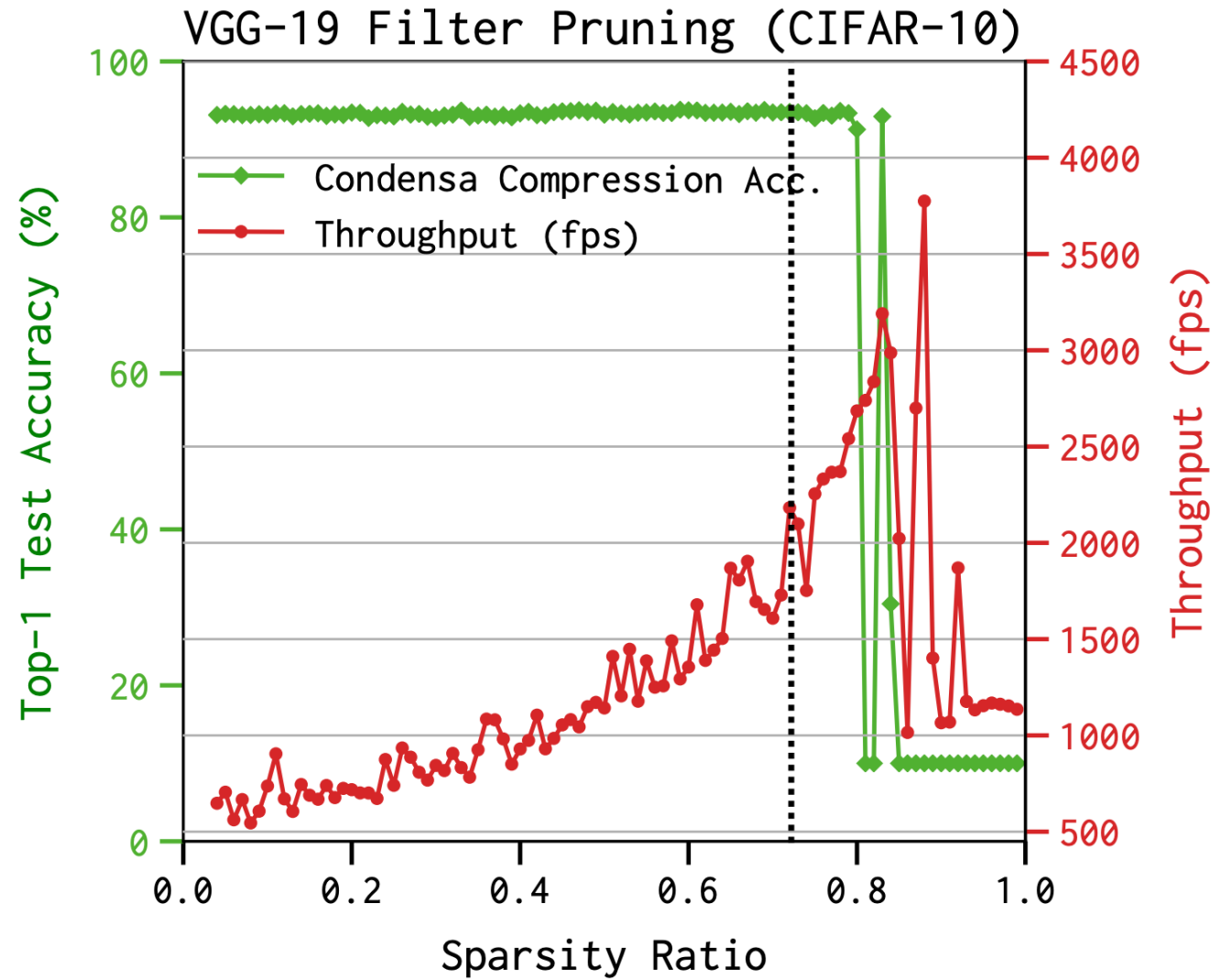
- Vertex programming model iterates over each node in parallel.
- Each node pulls in values from neighbors
- Similar to flow analysis!

from "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs", OSDI 2012



Sparse Neural Nets

from: "A PROGRAMMABLE APPROACH TO MODEL COMPRESSION". arxiv 2019.



How can we deal with load imbalance?

- Great research question! Changes per domain/architecture/input etc.

Work stealing

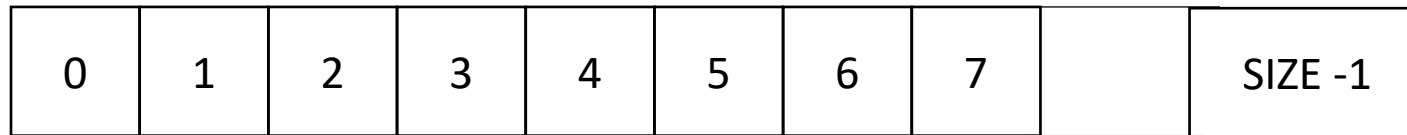
- Tasks are dynamically assigned to threads.

Work stealing - global implicit worklist

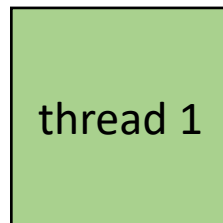
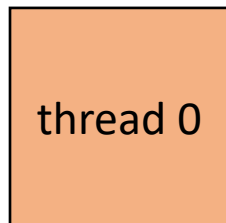
- Pros
 - Simple to implement
- Cons:
 - High contention on global counter
 - Potentially bad memory locality.

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

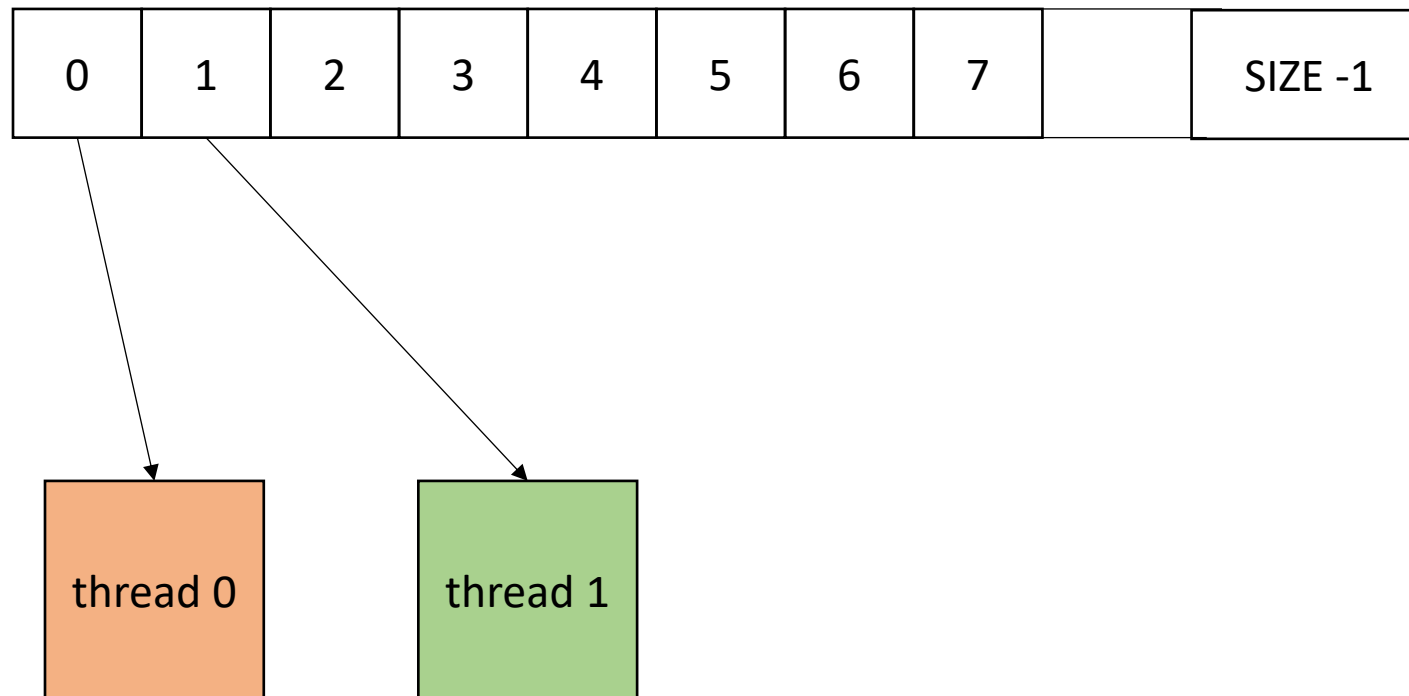


cannot color initially!



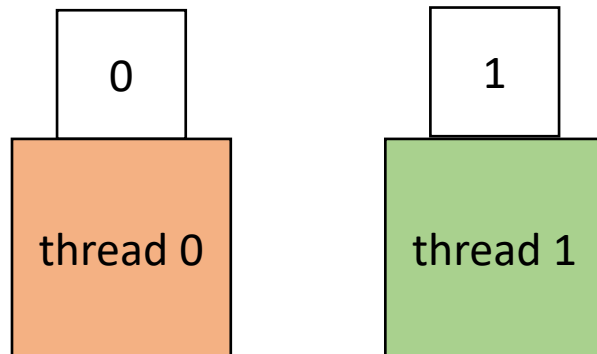
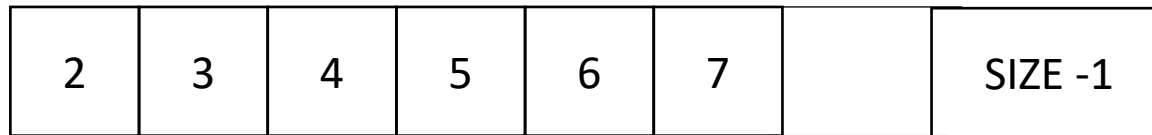
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



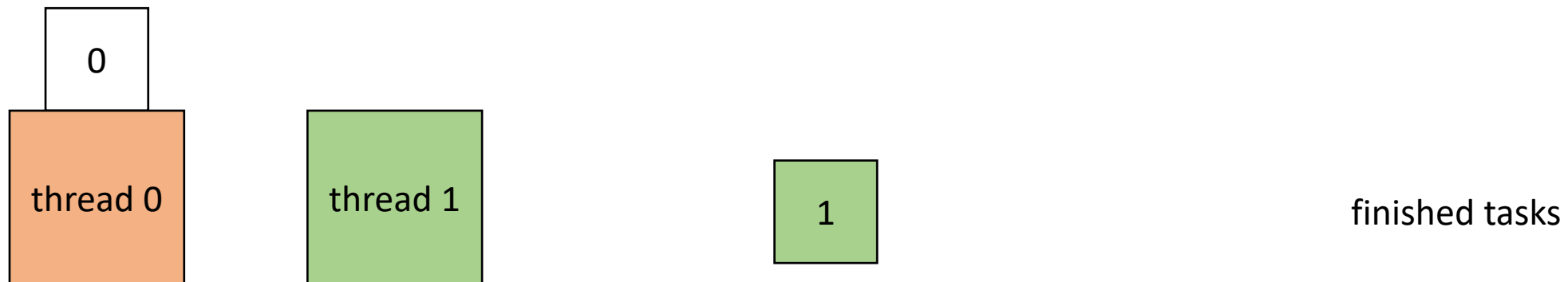
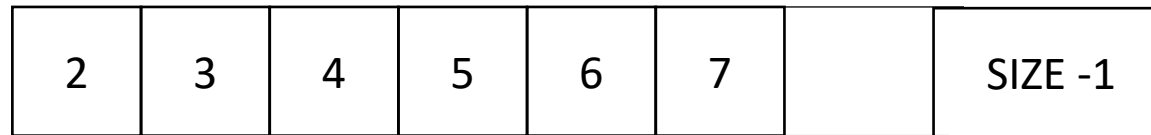
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



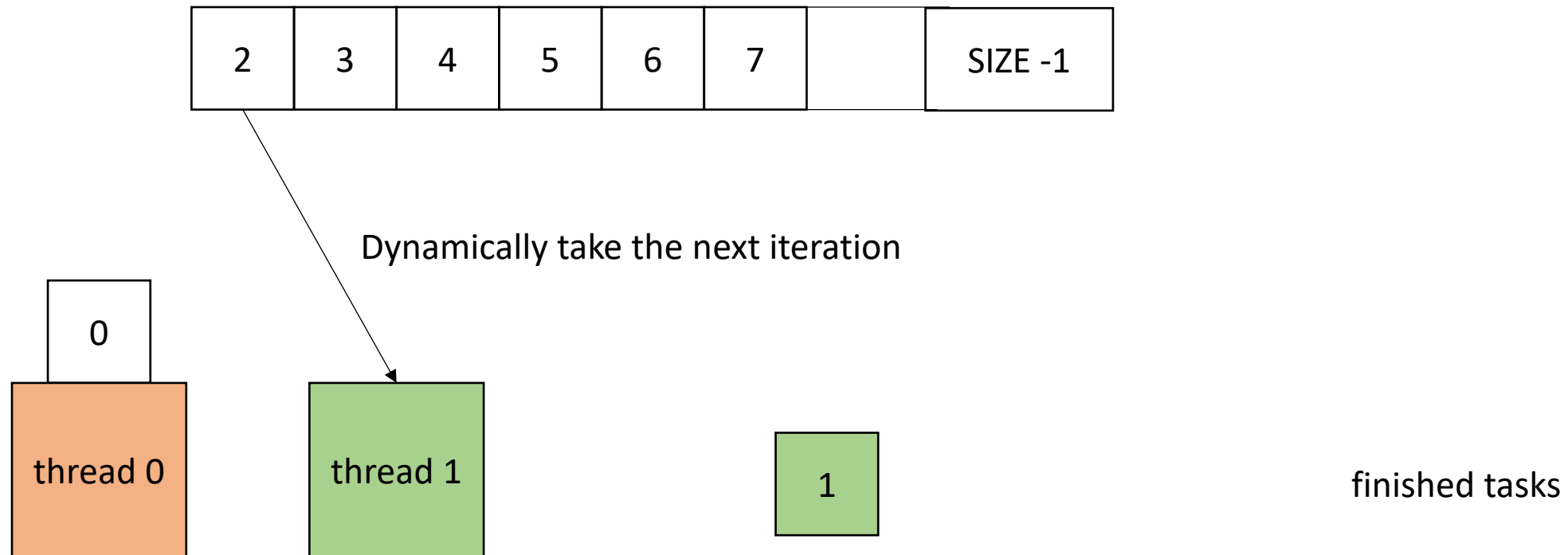
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



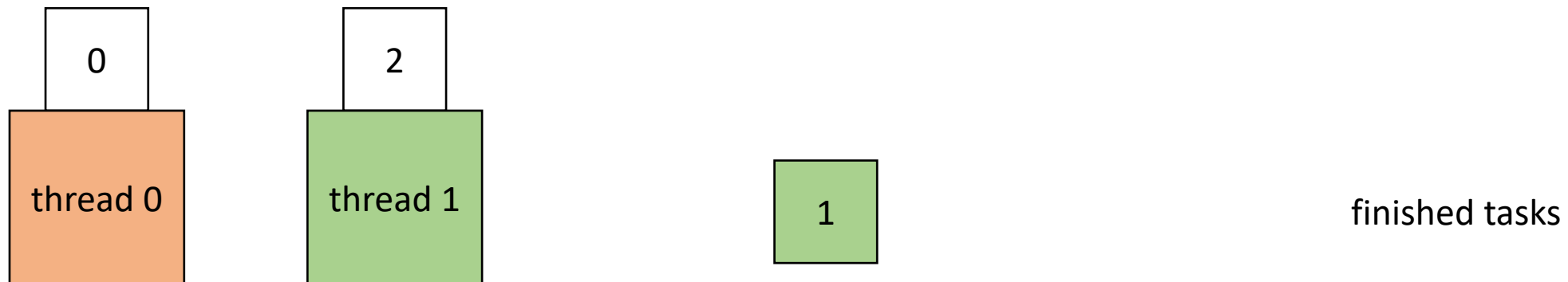
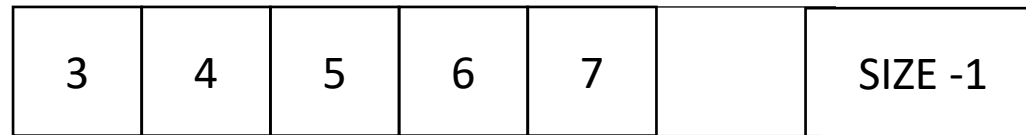
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



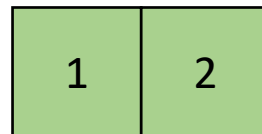
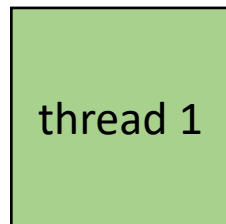
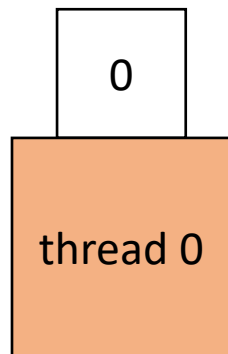
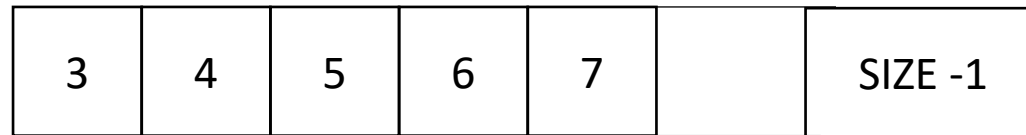
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



Work stealing - global implicit worklist

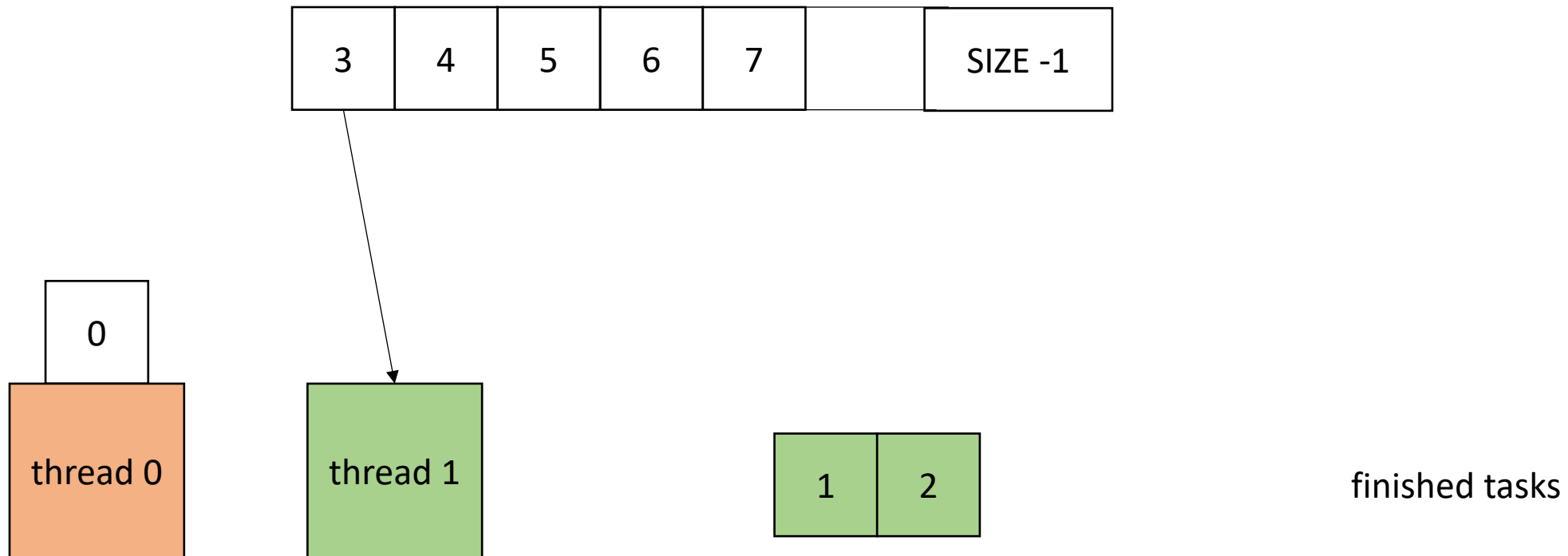
- Global worklist: threads take tasks (iterations) dynamically



finished tasks

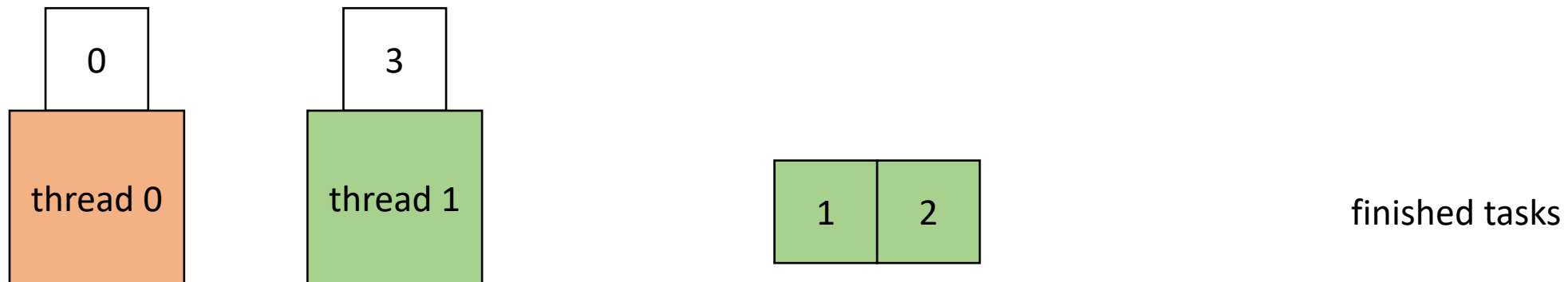
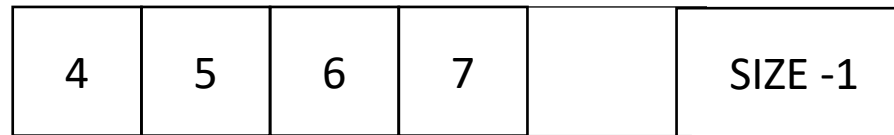
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



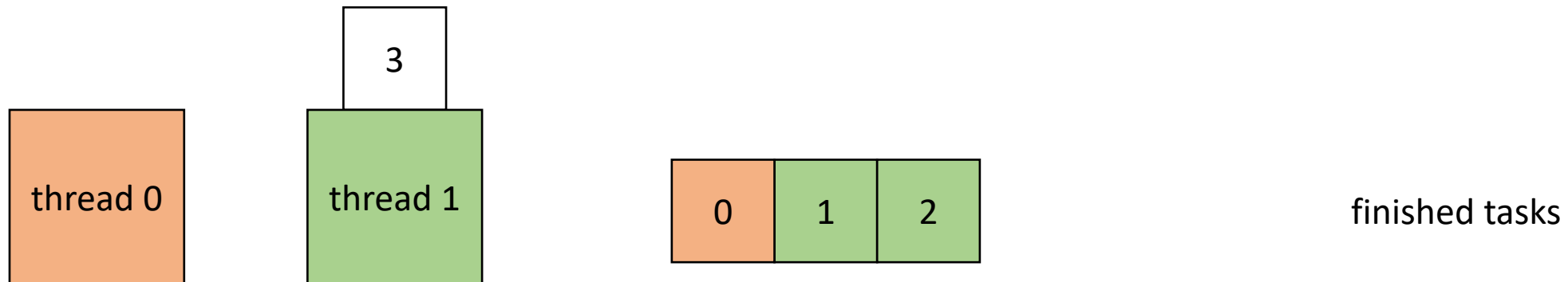
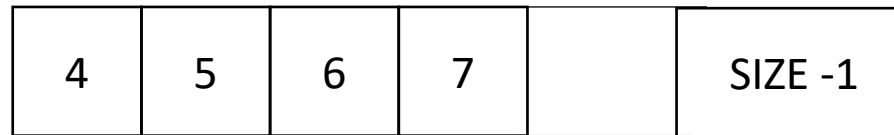
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



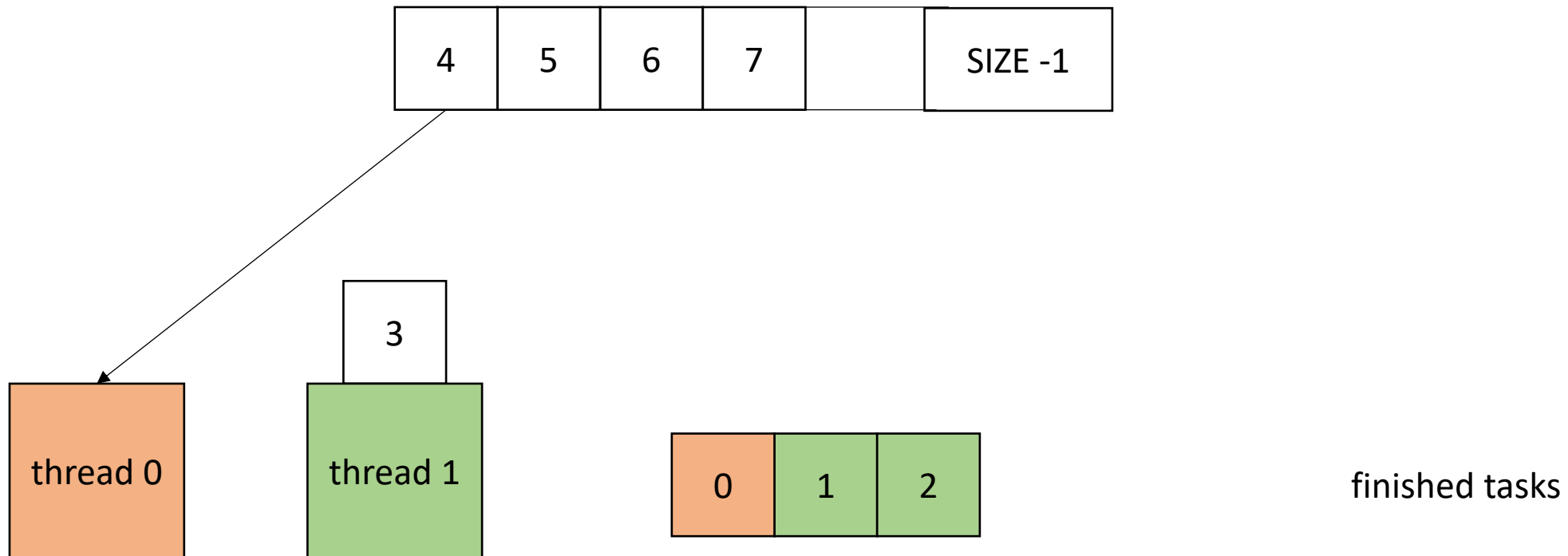
Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



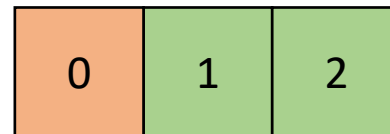
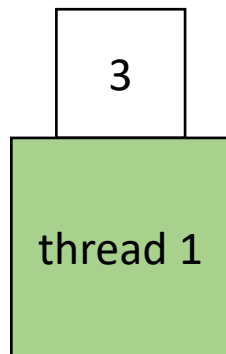
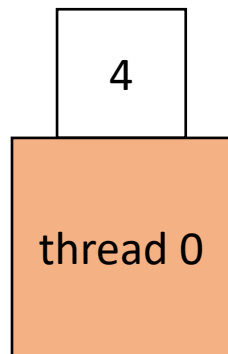
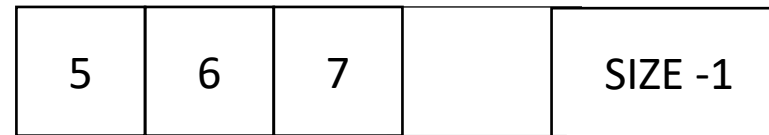
Work stealing - global implicit worklist

- 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

- 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:

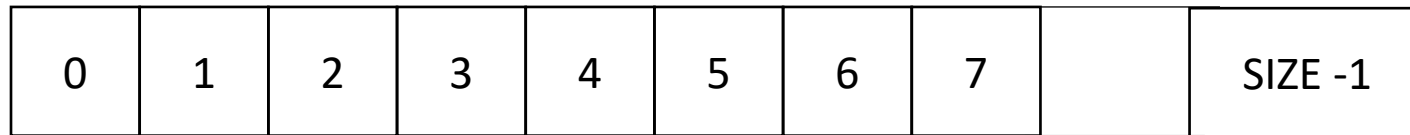
```
void foo() {  
    ...  
    for (t = 0; x < THREADS; t++) {  
        spawn(parallel_loop);  
    }  
    join();  
    x = 0;  
    ...  
}
```

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

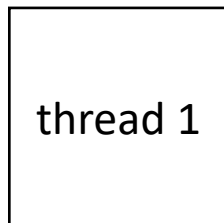
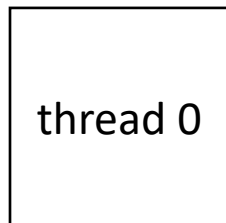
Are we finished?

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



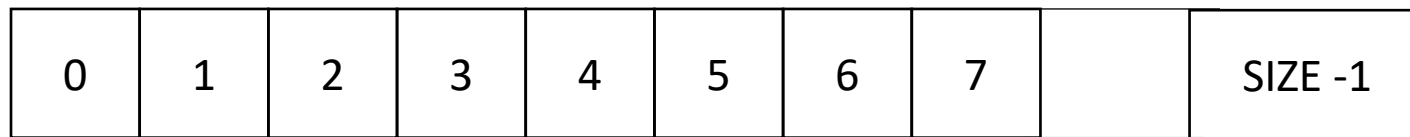
x: 0
0 - local_x - UNDEF
1 - local_x - UNDEF



```
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

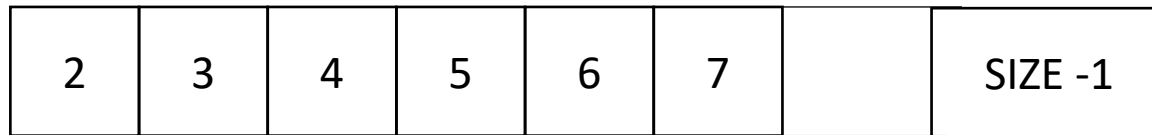
thread 0

thread 1

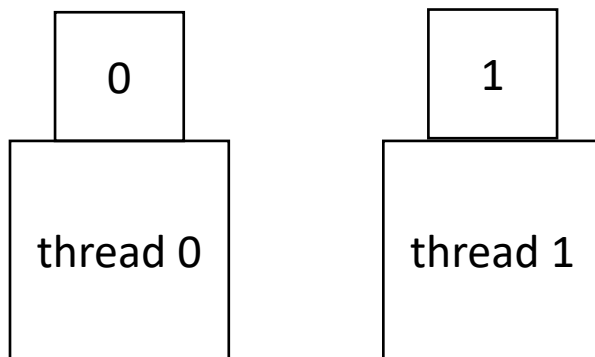
```
atomic_int x = 0;
void parallel_loop(...) {
    for (int local_x = x++;
         local_x < SIZE;
         local_x = x++) {
        // dynamic work based on x
    }
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



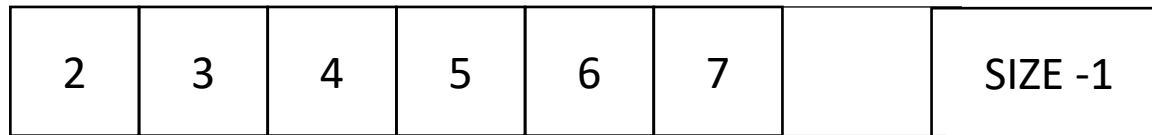
x: 2
0 - local_x - 0
1 - local_x - 1



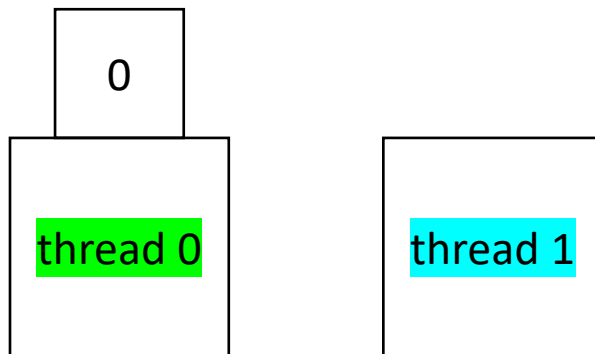
```
atomic_int x = 0;
void parallel_loop(...) {
    for (int local_x = x++;
         local_x < SIZE;
         local_x = x++) {
        // dynamic work based on x
    }
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



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

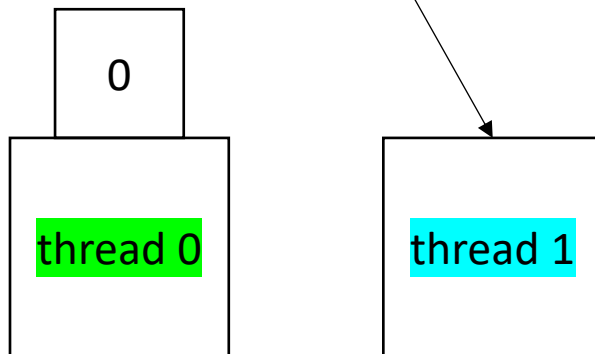
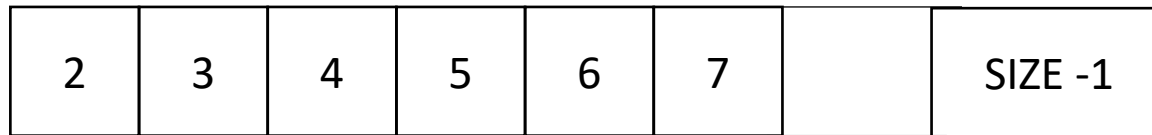


```
atomic_int x = 0;  
void parallel_loop(...) {  
  
    for (int local_x = x++;  
         local_x < SIZE;  
         local_x = x++) {  
  
        // dynamic work based on x  
  
    }  
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

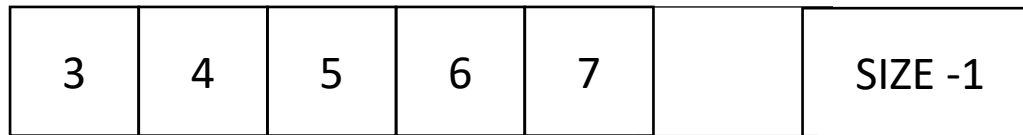
x: 3
0 - local_x - 0
1 - local_x - 2



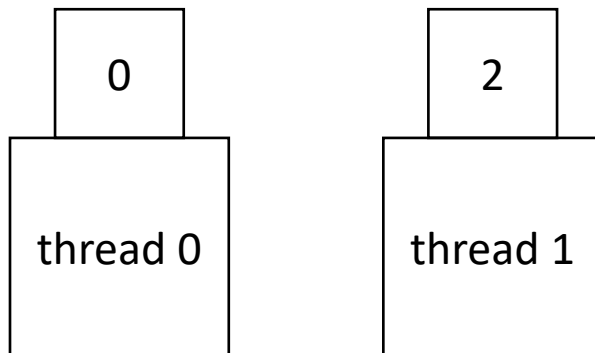
```
atomic_int x = 0;
void parallel_loop(...) {
    for (int local_x = x++;
         local_x < SIZE;
         local_x = x++) {
        // dynamic work based on x
    }
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



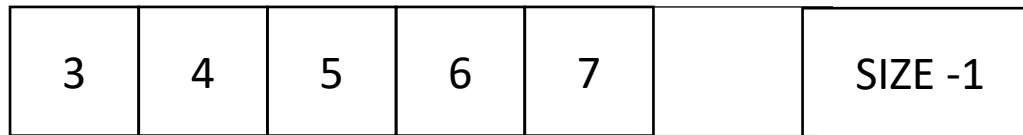
x: 3
0 - local_x - 0
1 - local_x - 2



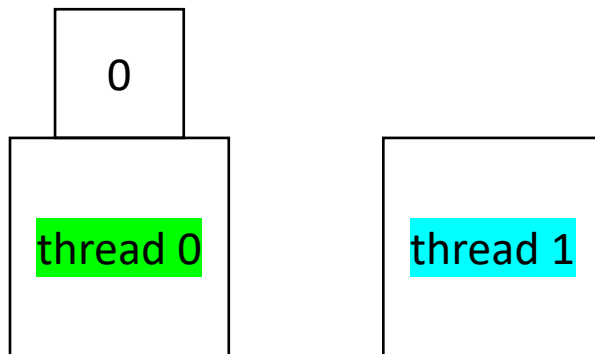
```
atomic_int x = 0;
void parallel_loop(...) {
    for (int local_x = x++;
         local_x < SIZE;
         local_x = x++) {
        // dynamic work based on x
    }
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



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

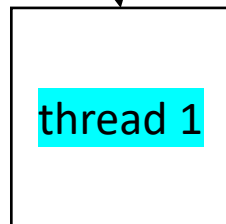
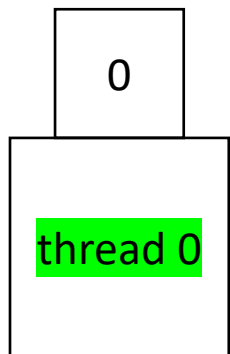
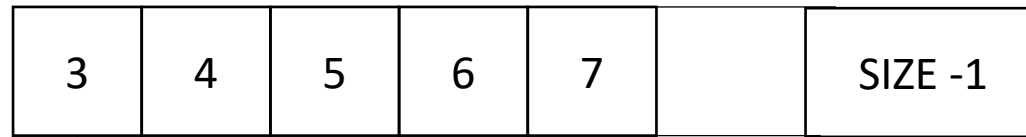


```
atomic_int x = 0;  
void parallel_loop(...) {  
  
    for (int local_x = x++;  
         local_x < SIZE;  
         local_x = x++) {  
  
        // dynamic work based on x  
  
    }  
}
```


Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

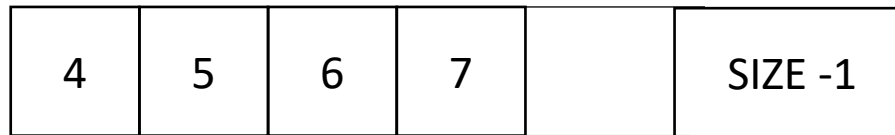
x: 4
0 - local_x - 0
1 - local_x - 3



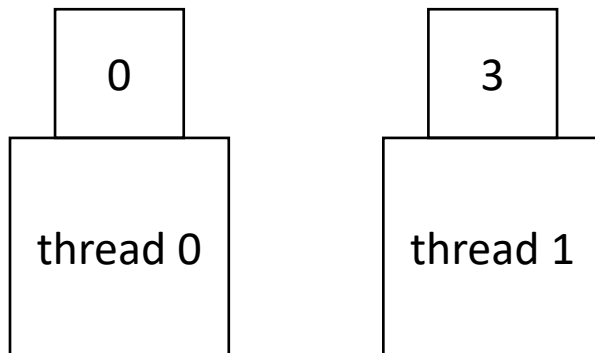
```
atomic_int x = 0;
void parallel_loop(...) {
    for (int local_x = x++;
         local_x < SIZE;
         local_x = x++) {
        // dynamic work based on x
    }
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



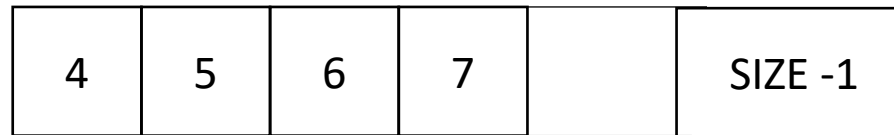
x: 4
0 - local_x - 0
1 - local_x - 3



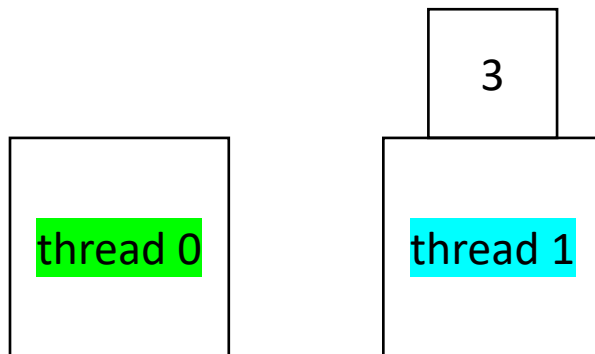
```
atomic_int x = 0;  
void parallel_loop(...) {  
    for (int local_x = x++;  
         local_x < SIZE;  
         local_x = x++) {  
        // dynamic work based on x  
    }  
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically



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

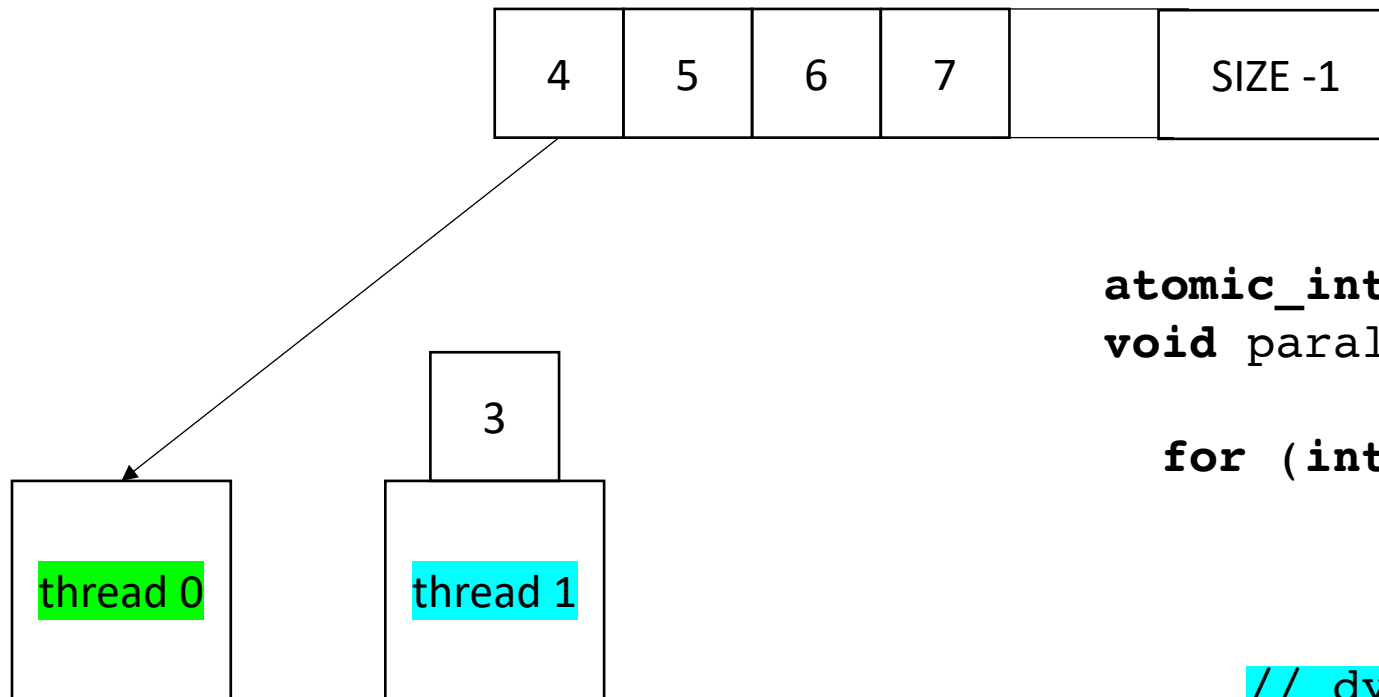


```
atomic_int x = 0;  
void parallel_loop(...) {  
    for (int local_x = x++;  
         local_x < SIZE;  
         local_x = x++) {  
        // dynamic work based on x  
    }  
}
```

Work stealing - global implicit worklist

- Global worklist: threads take tasks (iterations) dynamically

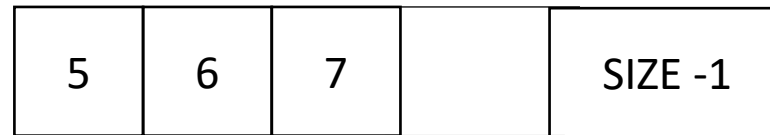
x: 5
0 - local_x - 4
1 - local_x - 3



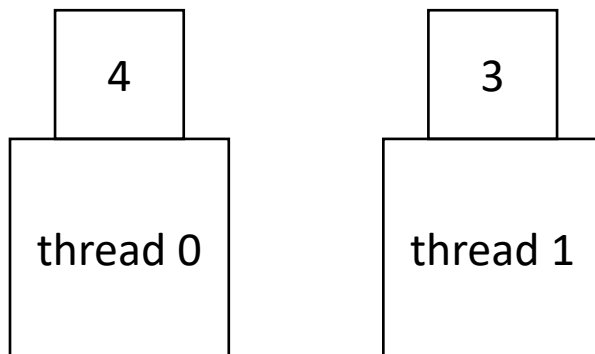
```
atomic_int x = 0;
void parallel_loop(...) {
    for (int local_x = x++;
         local_x < SIZE;
         local_x = x++) {
        // dynamic work based on x
    }
}
```

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

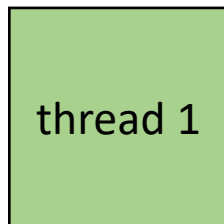
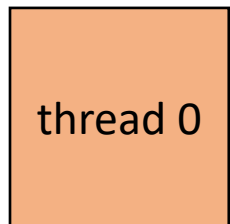
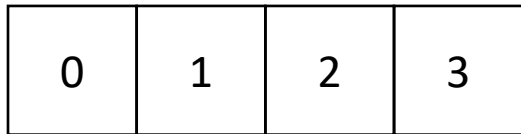
Next implementation

Work stealing - local worklists

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

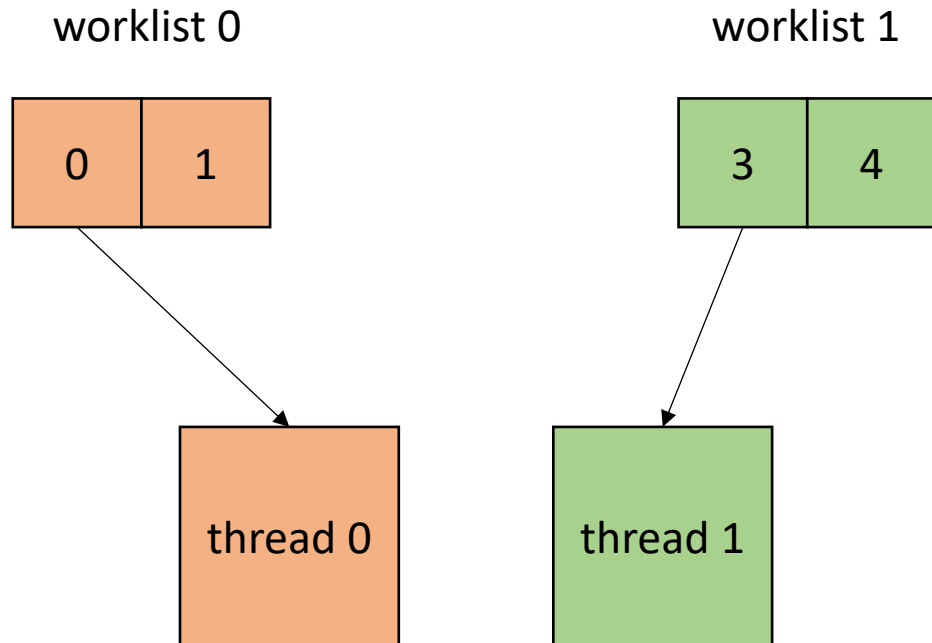
Work stealing - local worklists

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



Work stealing - local worklists

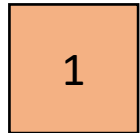
- local worklists: divide tasks into different worklists for each thread



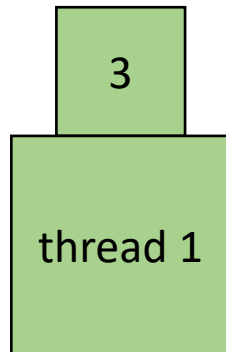
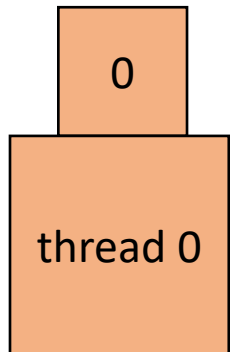
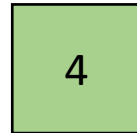
Work stealing - local worklists

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

worklist 0



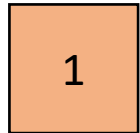
worklist 1



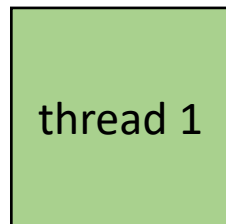
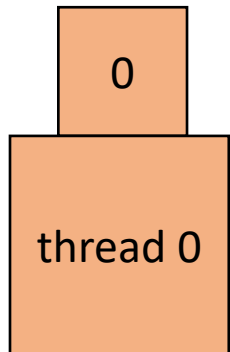
Work stealing - local worklists

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

worklist 0



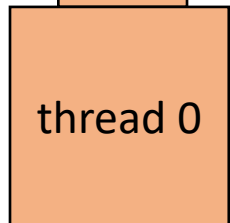
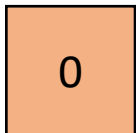
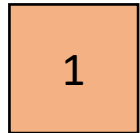
worklist 1



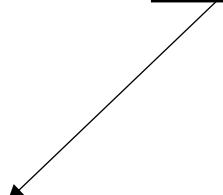
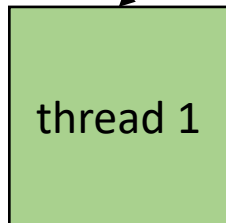
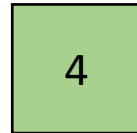
Work stealing - local worklists

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

worklist 0



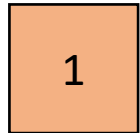
worklist 1



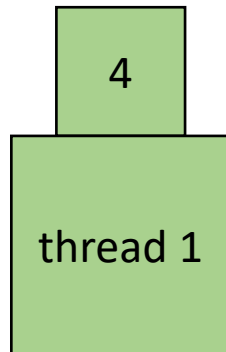
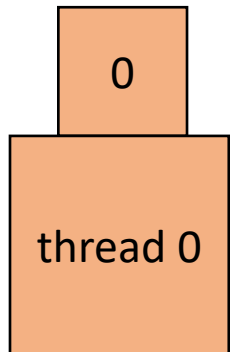
Work stealing - local worklists

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

worklist 0



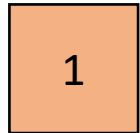
worklist 1



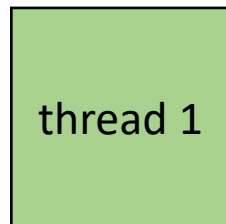
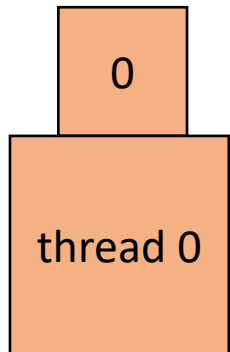
Work stealing - local worklists

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

worklist 0

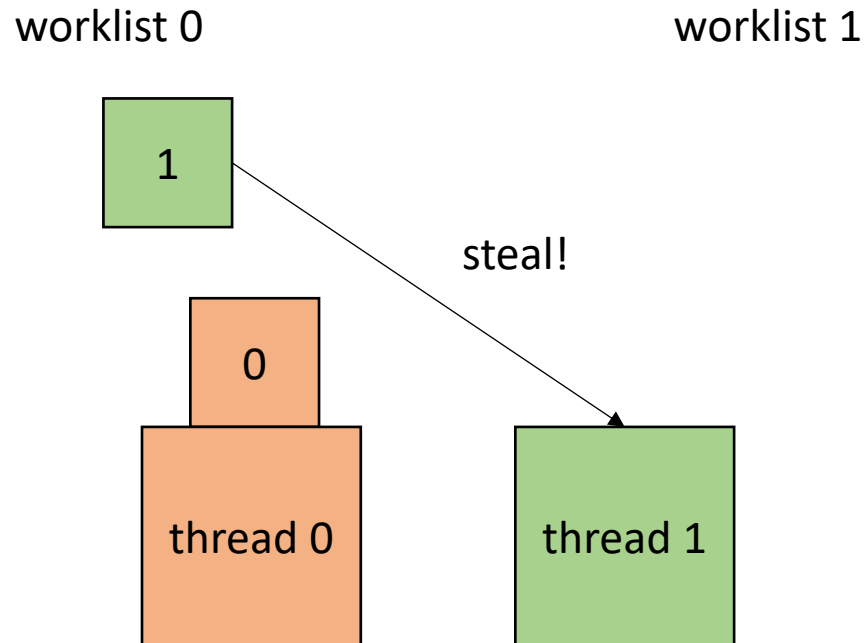


worklist 1



Work stealing - local worklists

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

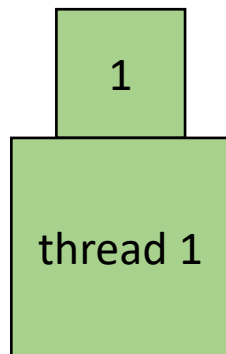
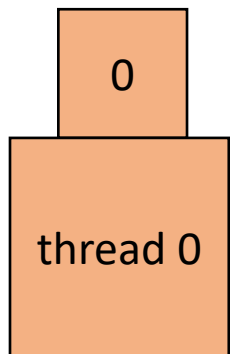


Work stealing - local worklists

- 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);  
    }  
    ...  
}
```

initialize queues in main thread

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

x	0	1	2	3
---	---	---	---	---

tid	0	0	1	1
-----	---	---	---	---

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;
    ...
}
```

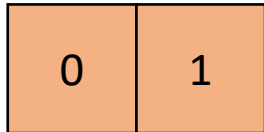
```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

    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
    }
}
```

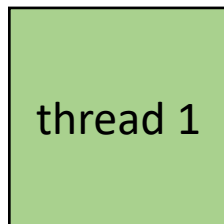
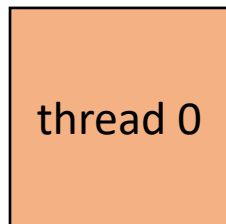
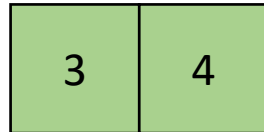
launch threads, join, reinitialize

Work stealing - local worklists

worklist 0



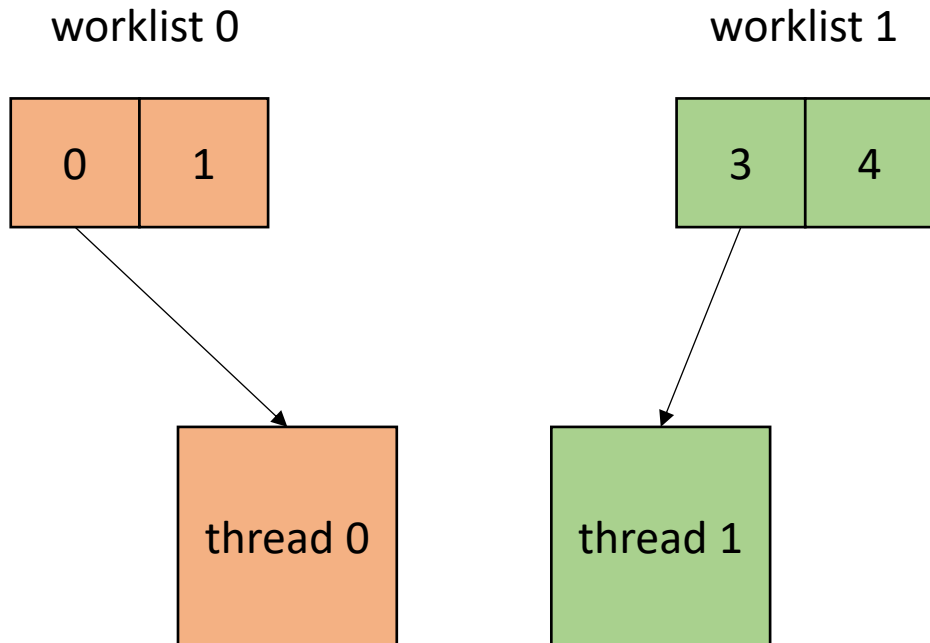
worklist 1



```
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

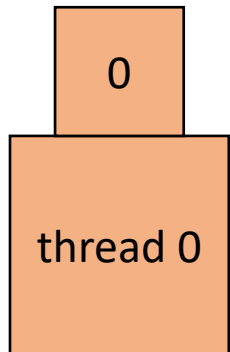
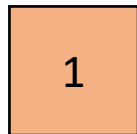


```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

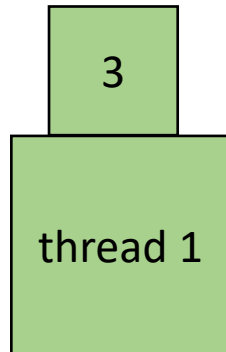
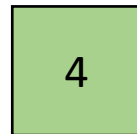
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```

Work stealing - local worklists

worklist 0



worklist 1

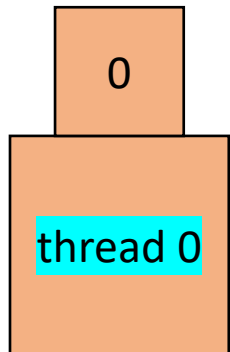
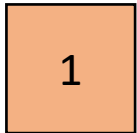


```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

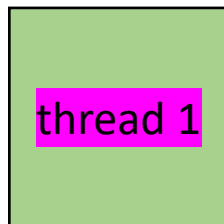
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```


Work stealing - local worklists

worklist 0



worklist 1

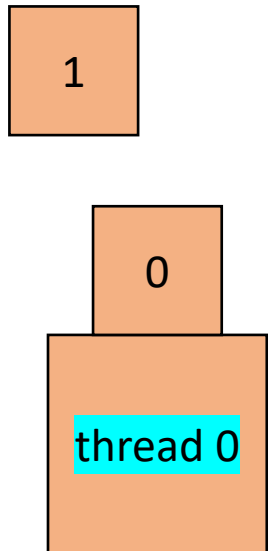


```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

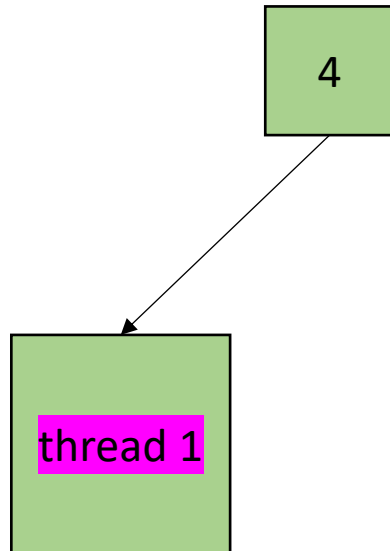
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```

Work stealing - local worklists

worklist 0



worklist 1

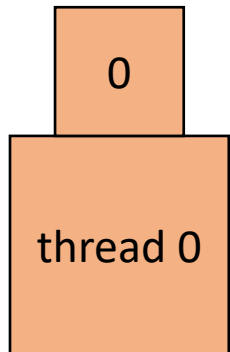
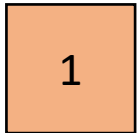


```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

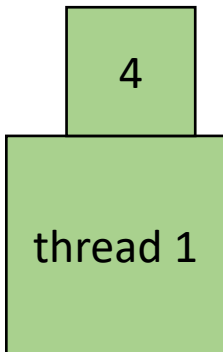
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```

Work stealing - local worklists

worklist 0



worklist 1

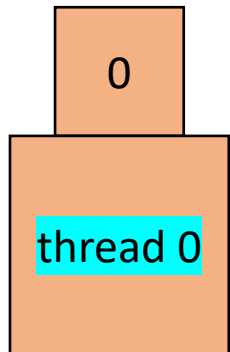
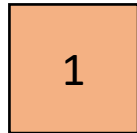


```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

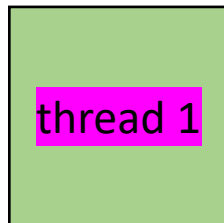
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```

Work stealing - local worklists

worklist 0



worklist 1



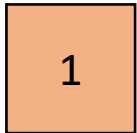
```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

    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
    }
}
```

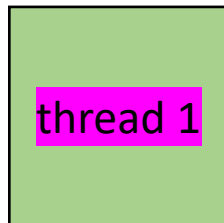
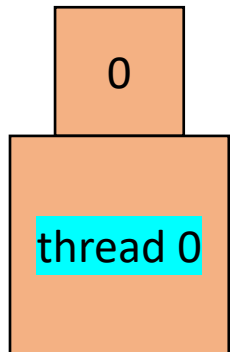
Work stealing - local worklists

finished_threads: 1

worklist 0



worklist 1



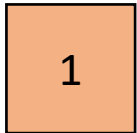
```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

    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
    }
}
```

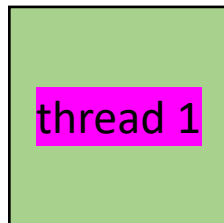
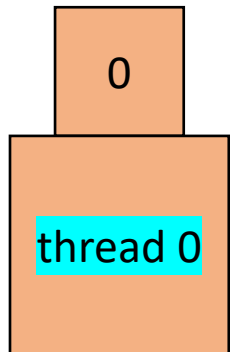
Work stealing - local worklists

finished_threads: 1

worklist 0



worklist 1



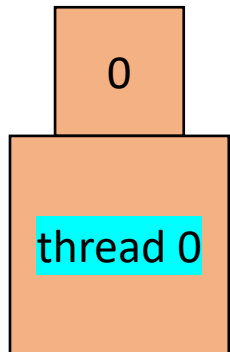
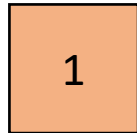
```
atomic_int finished_threads = 0;
void parallel_loop(..., int tid) {

    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
    }
}
```

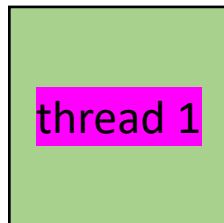
Work stealing - local worklists

finished_threads: 1

worklist 0



worklist 1



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

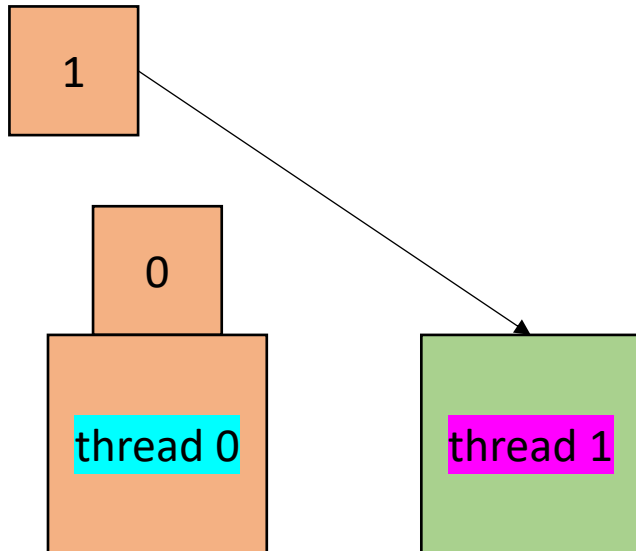
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```

Work stealing - local worklists

finished_threads: 1

worklist 0

worklist 1



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

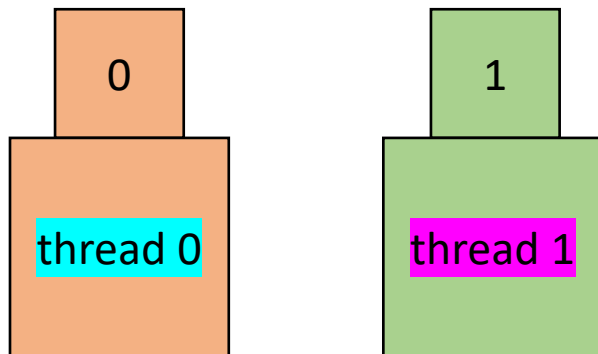
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```


Work stealing - local worklists

finished_threads: 1

worklist 0

worklist 1



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

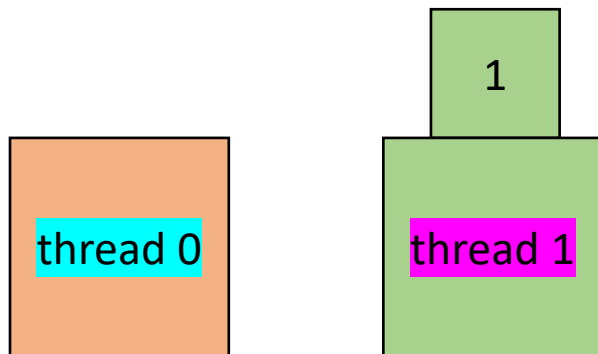
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```

Work stealing - local worklists

finished_threads: 1

worklist 0

worklist 1



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

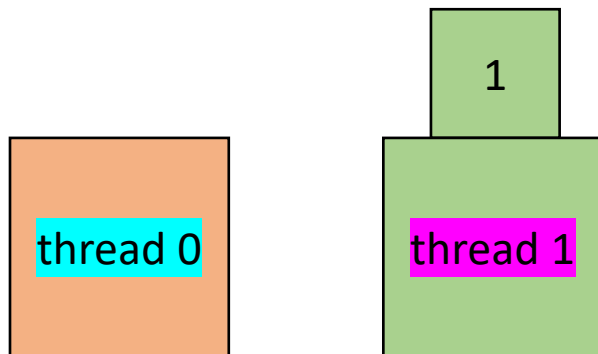
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```

Work stealing - local worklists

finished_threads: 1

worklist 0

worklist 1



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

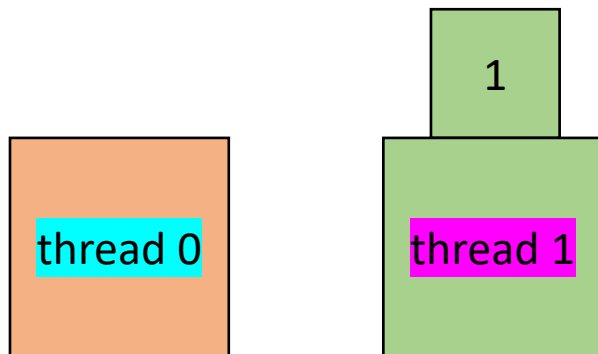
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```

Work stealing - local worklists

finished_threads: 2

worklist 0

worklist 1



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

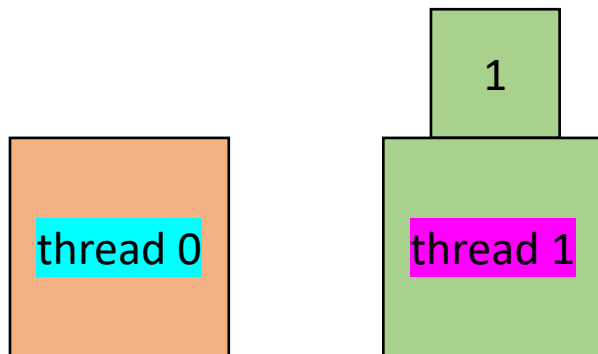
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```

Work stealing - local worklists

finished_threads: 2

worklist 0

worklist 1



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

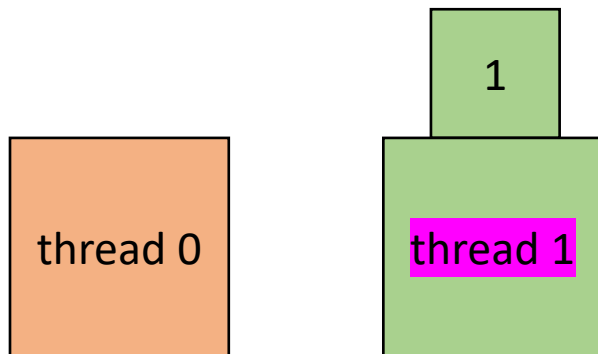
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```

Work stealing - local worklists

finished_threads: 2

worklist 0

worklist 1



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

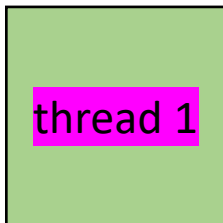
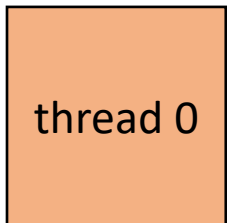
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```

Work stealing - local worklists

finished_threads: 2

worklist 0

worklist 1



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

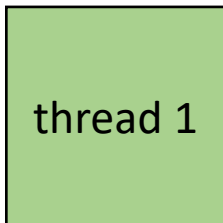
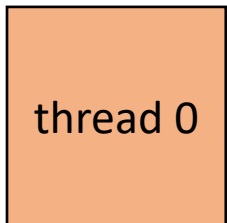
    int task = 0;
    while (cq[tid].dequeue(&task)) {
        // dynamic work based on task
    }
    finished_threads++;
    while (finished_threads != NUM_THREADS) {
        target = //select a random thread
        if (cq[target].dequeue(&task))
            // dynamic work based on task
    }
}
```

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].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

- 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

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 class

- Topics:
 - Restructuring loops
- Remember:
 - Homework 2 due tomorrow
 - Midterm due on Friday
 - Office hours tomorrow 3-5