CS443: Compiler Construction

Lecture 26: Parallelism and Concurrency

Stefan Muller

Concurrency: Interleave multiple threads

- Modularity
- Responsiveness
- Can be on multiple processors or time slicing

Parallelism: Run computations simultaneously on mult. processors

- Speed up computation
- Need multiple processors

Using concurrency for events

```
while(true) {
  if (can_accept(sock))
    conns[num_conns++] = accept(sock);
  for (int i = 0; i < num_conns; i++) {
    if (has_request(conns[i])) { … } } }
```

```
while(true) {
  conn = accept(sock);
  create_handling_thread(conn);
}
```

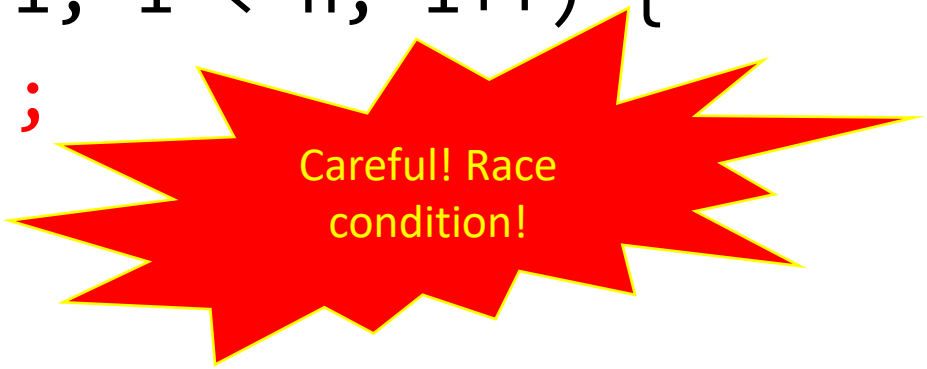
```
while(true) {
  req = recv(conn);
  …
}
```

```
while(true) {
  req = recv(conn);
  …
}
```

Using concurrency to implement parallelism

```
int sum;

void sum_array(int A[], int l, int h) {
  for (int i = l; i < h; i++) {
    sum += A[i];
  }
}
```



Race conditions: multiple threads accessing data simultaneously

```
int x = 0;
```

```
for (int i = 0; i < 1000; i++) {
  x++;
}
```

```
for (int i = 0; i < 1000; i++) {
  x++;
}
```

What are the possible values of x?

A: [1000, 2000]

```
      temp1 = x;
                  temp2 = x;
      x = temp1 + 1;
                  x = temp2 + 1;
```

OK, so what does this have to do with compilers?

Is this a safe optimization?

```
int x = 0;

for (int i = 0; i < 1000; i++) {
  x++;
}
```

```
for (int i = 0; i < 1000; i++) {
  x++;
}
```

```
x += 1000;
```

```
x += 1000;
```

Changes set of possible answers (now just 1000, 2000) but maybe?

Is this a safe optimization?

```
int num_conns;
while(true) {
  conn = accept(sock);
  create_handling_thread(conn);
  num_conns++;
}
```

```
while(true) {
  for (int i = 0; i < num_conns; i++) {
    …
  }
}
```

Don’t even have to explicitly intend this as an optimization: could just be the result of putting num_conns in a register!

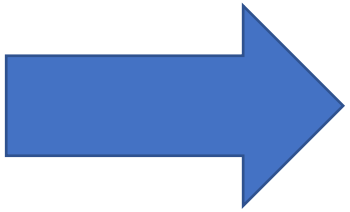
```
int num_conns;
while(true) {
  conn = accept(sock);
  create_handling_thread(conn);
  num_conns++;
}
```

```
int n = num_conns;
while(true) {
  for (int i = 0; i < n; i++) {
    …
  }
}
```

Is this a safe optimization?

```
int a;
int b;
int c;
int d;

int f() {
  c = a * b;
  d = a * b + a;
  return d;
}
```



```
int a;
int b;
int c;
int d;

int f() {
  c = a * b;
  d = c + a;
  return d;
}
```

No, under our previous def. (it can change the answer)!

```
int a;
int b;
int c;
int d;

int f() {
  c = a * b;
  d = a * b + a;
  return d;
}
```

```
int g() {
  c++;
  return c;
}
```

C's `volatile` keyword tells the compiler the value might change at any time

```
volatile int a;
volatile int b;
volatile int c;
volatile int d;

int f() {
  c = a * b;
  d = a * b + a;
  return d;
}
```

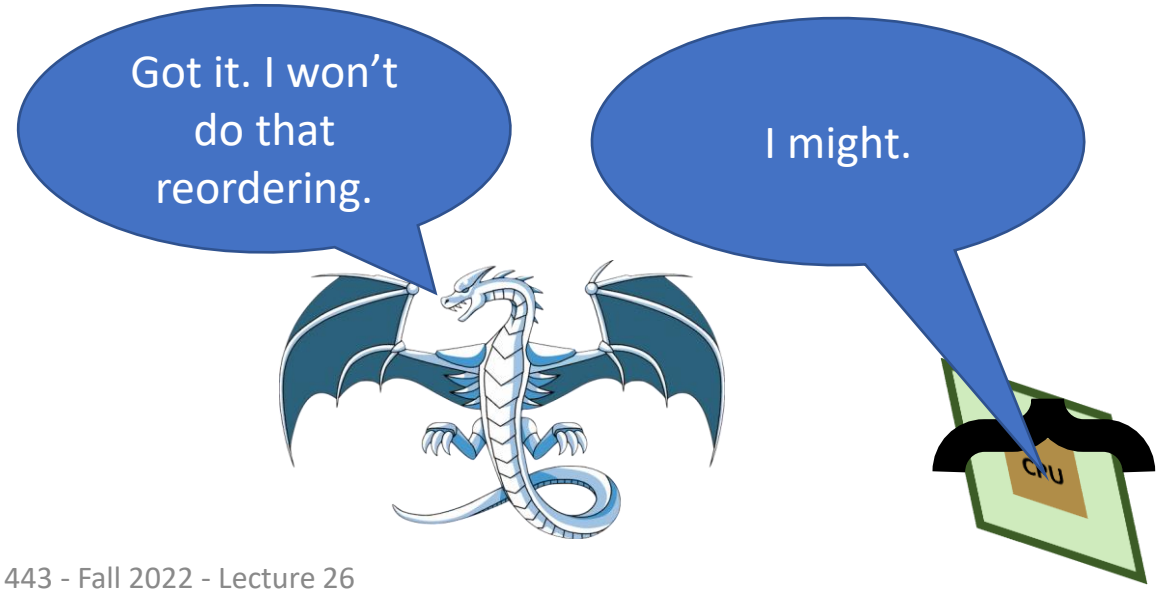
(Doesn't fix data races)

Is this a valid compilation?

```
x = 42;                lw a0, 0(t0)        # a0 = y
z = y;                 addi t1, zero, 42 # t1 = 42
return z;              sw t1, 0(t2)        # x = 42
```

```
x = 42;
z = y;
return z;
```

```
y = x;
return x;
```



When designing a language, we can offer a more abstract version of parallelism

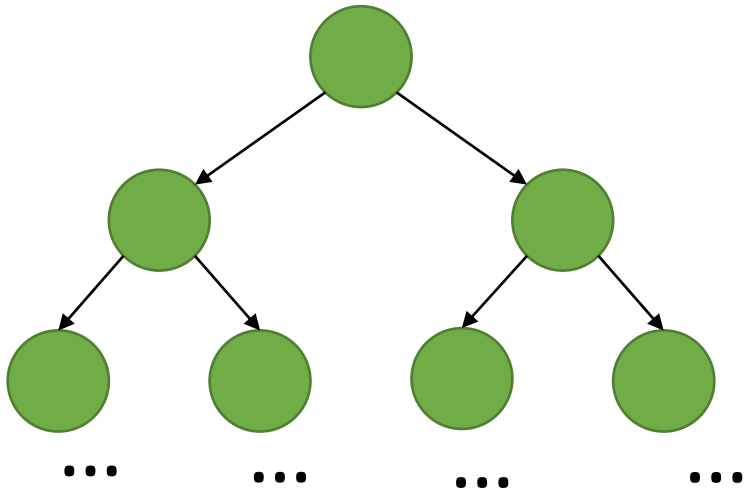
- Allowing OCaml programmers to call `pthread_create` is likely to cause all hell to break loose

“Implicit” parallelism

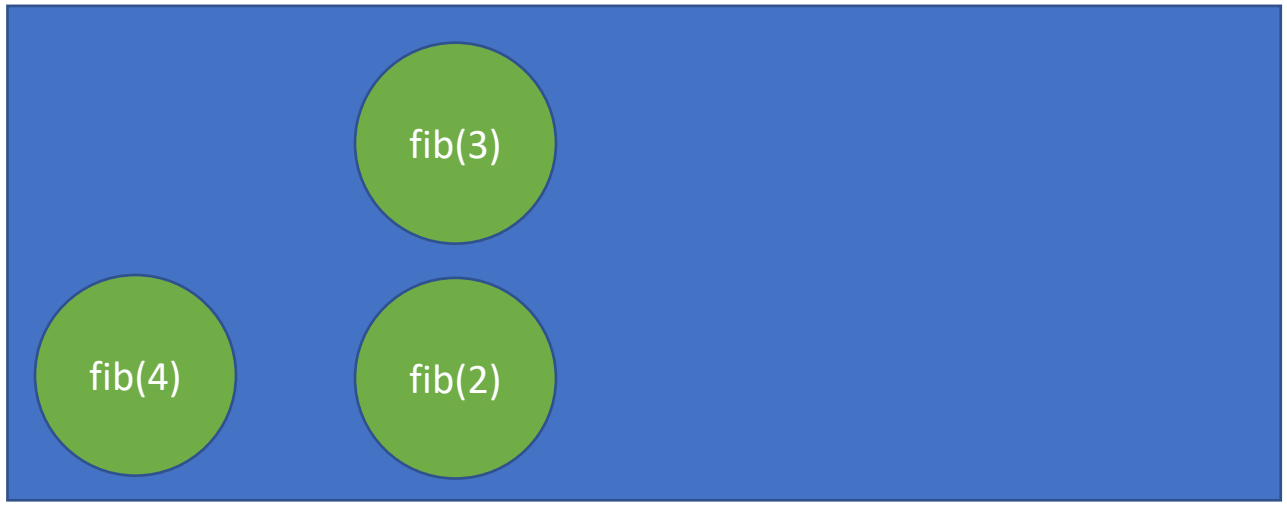
```
let rec fib (n: int) =
  if n <= 1 then n
  else
    let (a, b) = par (fib (n – 2), fib (n – 1))
    in
      a + b
```

How to implement par?

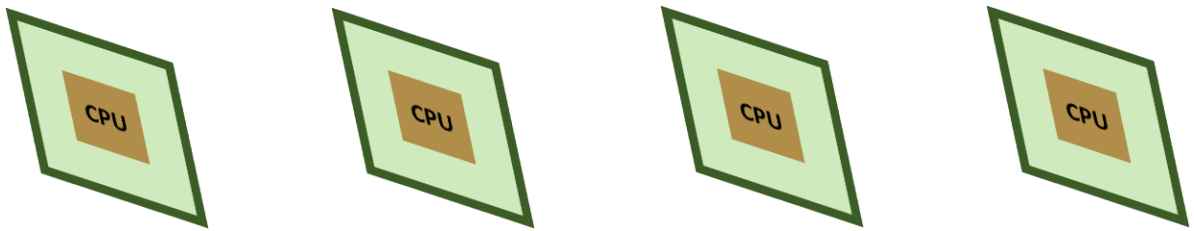
- pthread_create, pthread_join
  - WAY better off just running sequentially-overhead of pthreads is huge



User-level lightweight threads

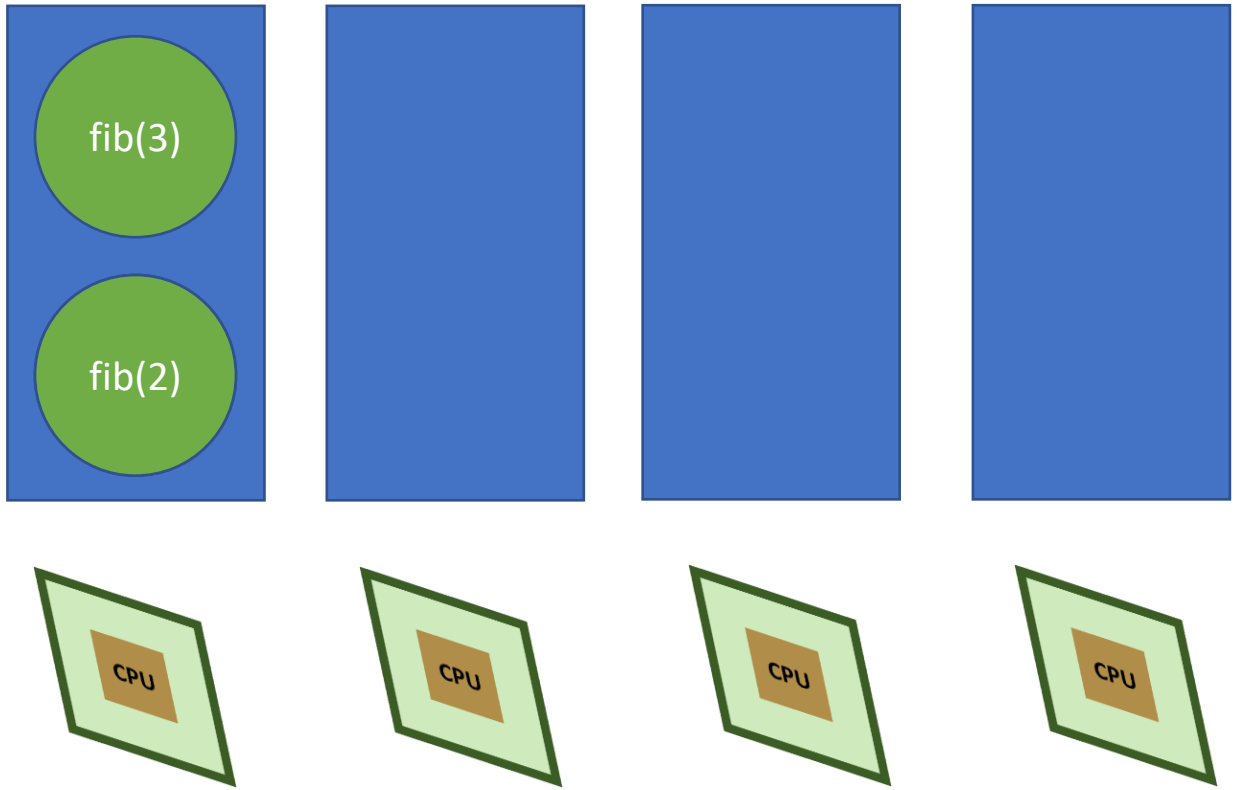


One global thread pool: too much contention



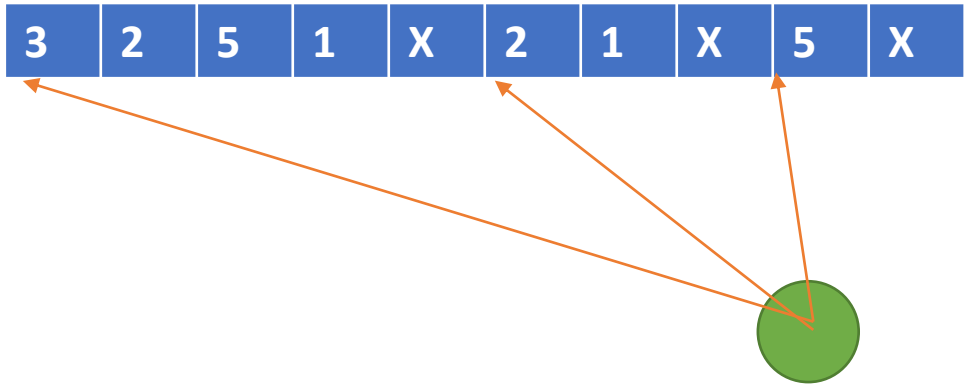
One pthread per processor

Work stealing: one queue of tasks per processor



One pthread per processor

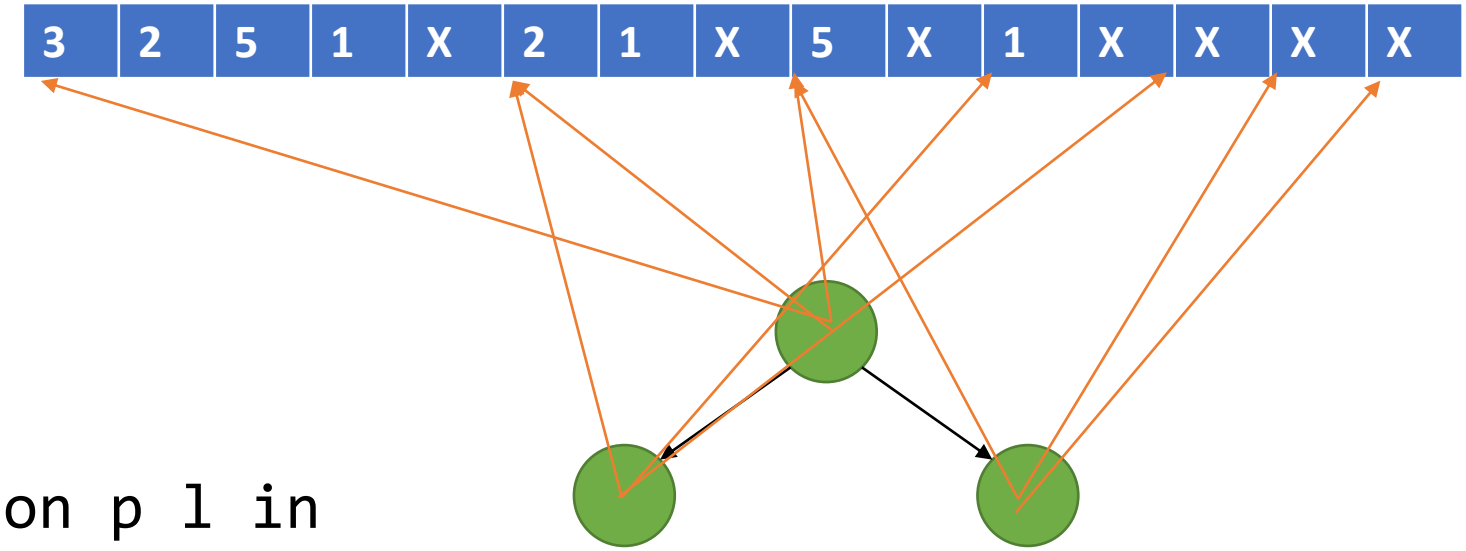
Each thread gets its own environment, but share a heap




```
let rec qsort l =
  match l with
  | [] -> []
  | [x] -> [x]
  | p::l ->
     let (a, b) = partition p l in
     let (a_sort, b_sort) =
       par (qsort a, qsort b)
     in
     a_sort @ [p] @ b_sort
```

Each thread gets its own environment, but share a heap

```
let rec qsort l =
  match l with
  | [] -> []
  | [x] -> [x]
  | p::l ->
     let (a, b) = partition p l in
     let (a_sort, b_sort) =
       par (qsort a, qsort b)
     in
     a_sort @ [p] @ b_sort
```

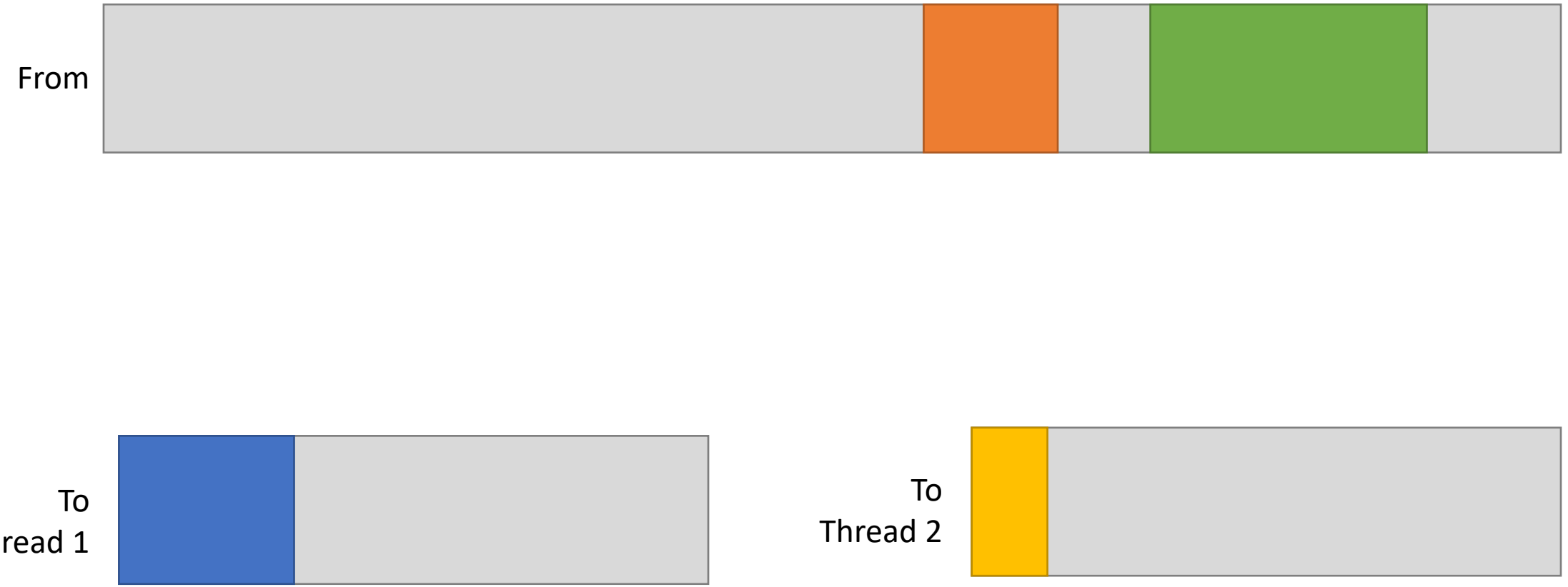


Problems with shared heap

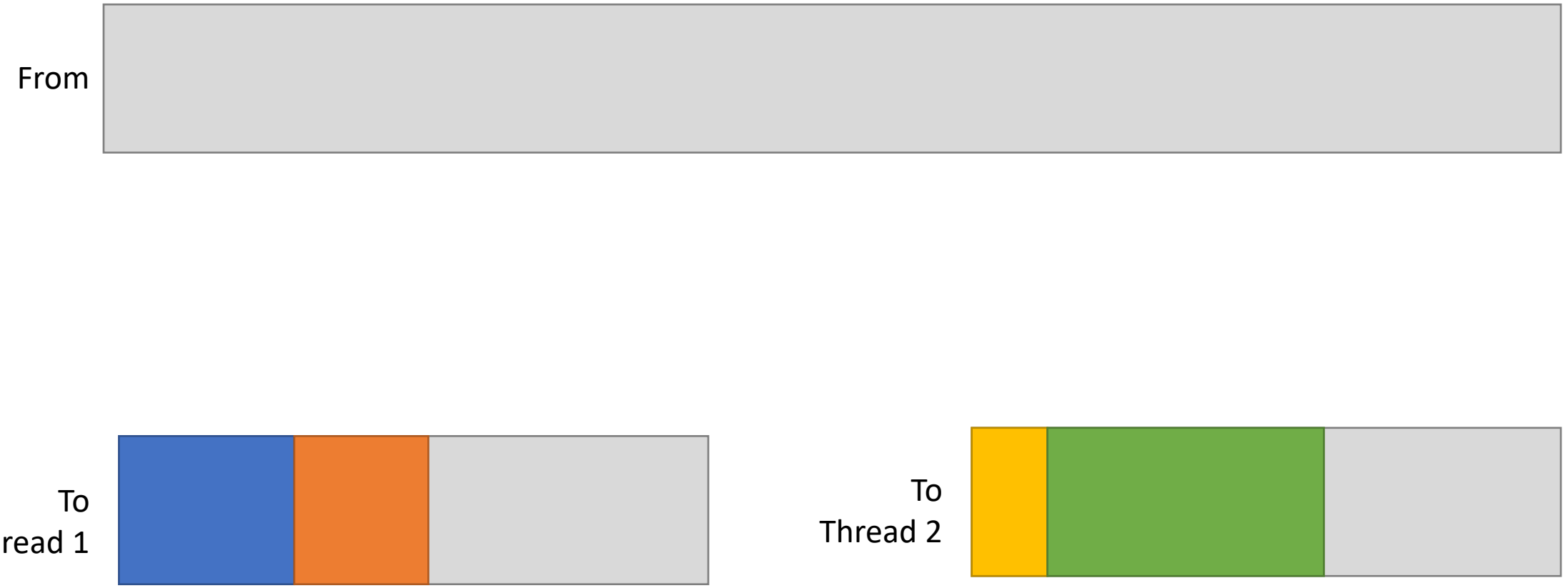
- Contention on allocation
  - Can give each thread a separate heap pointer
- Need stop-the-world GC
  - All threads need to synchronize

Copying GC can be parallelized

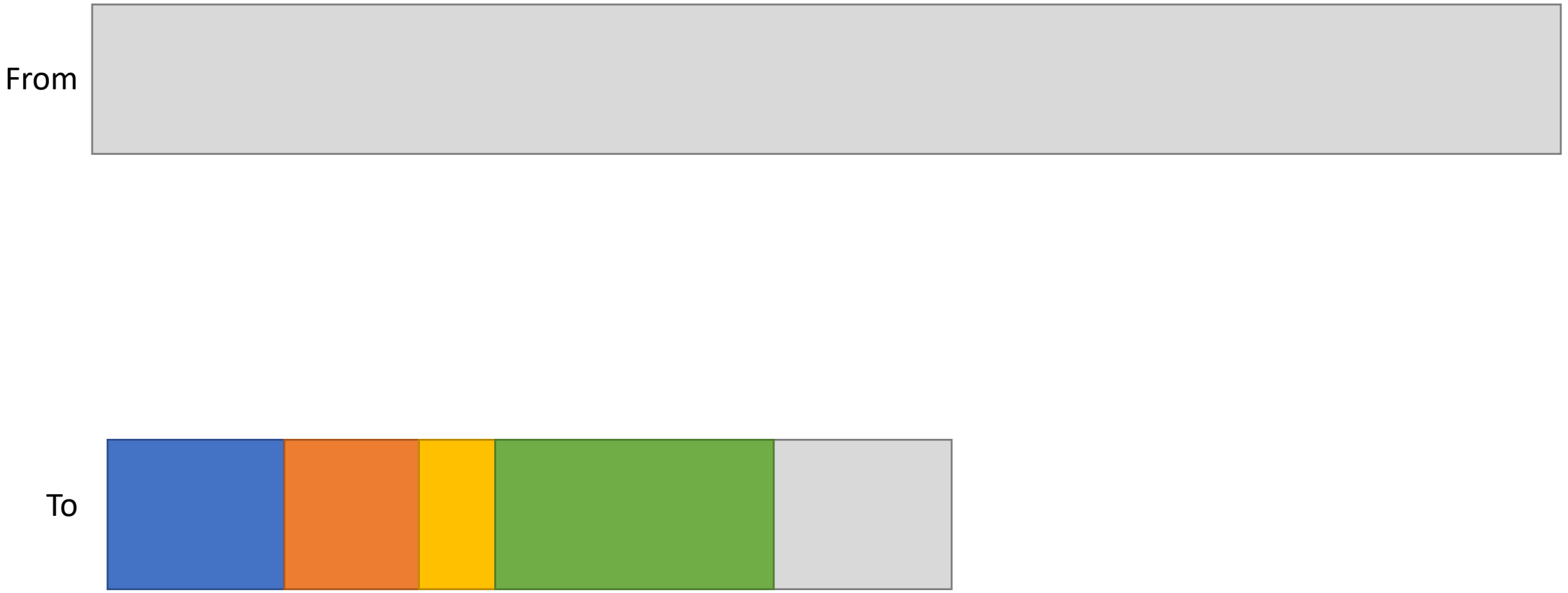
Copying GC can be parallelized



Copying GC can be parallelized



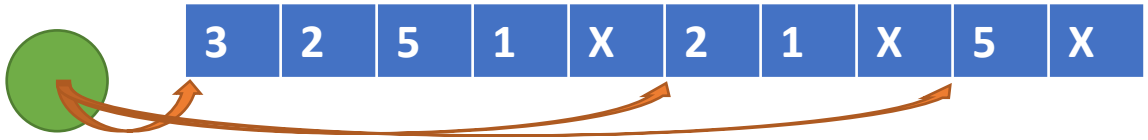
Copying GC can be parallelized



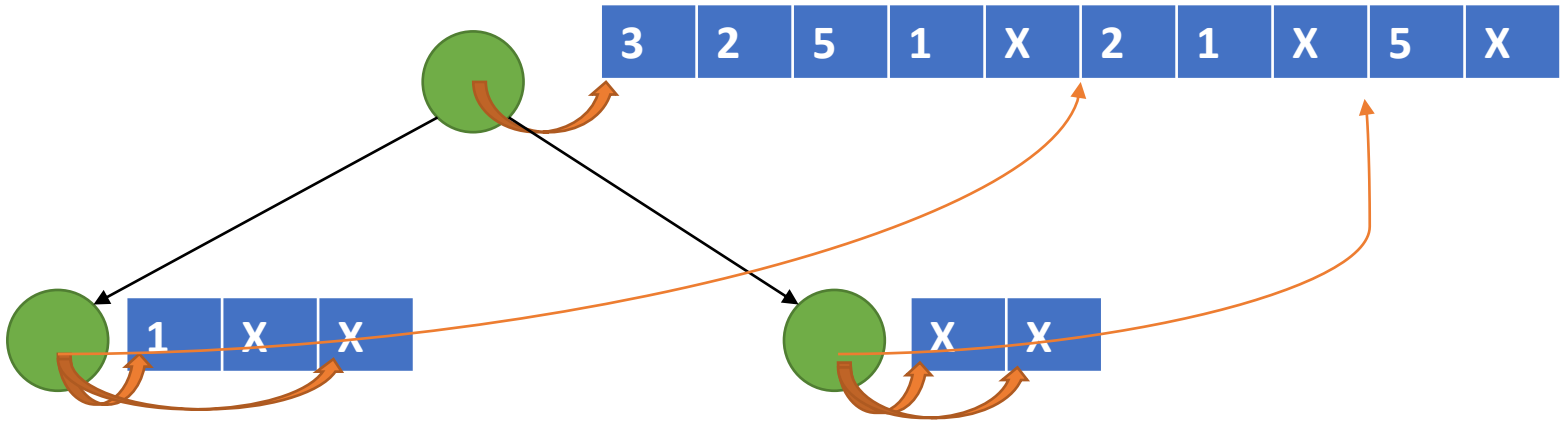
Copying GC can be parallelized

- That’s (roughly) what Haskell does
- Still doesn’t solve the problem of stopping, synchronizing all threads

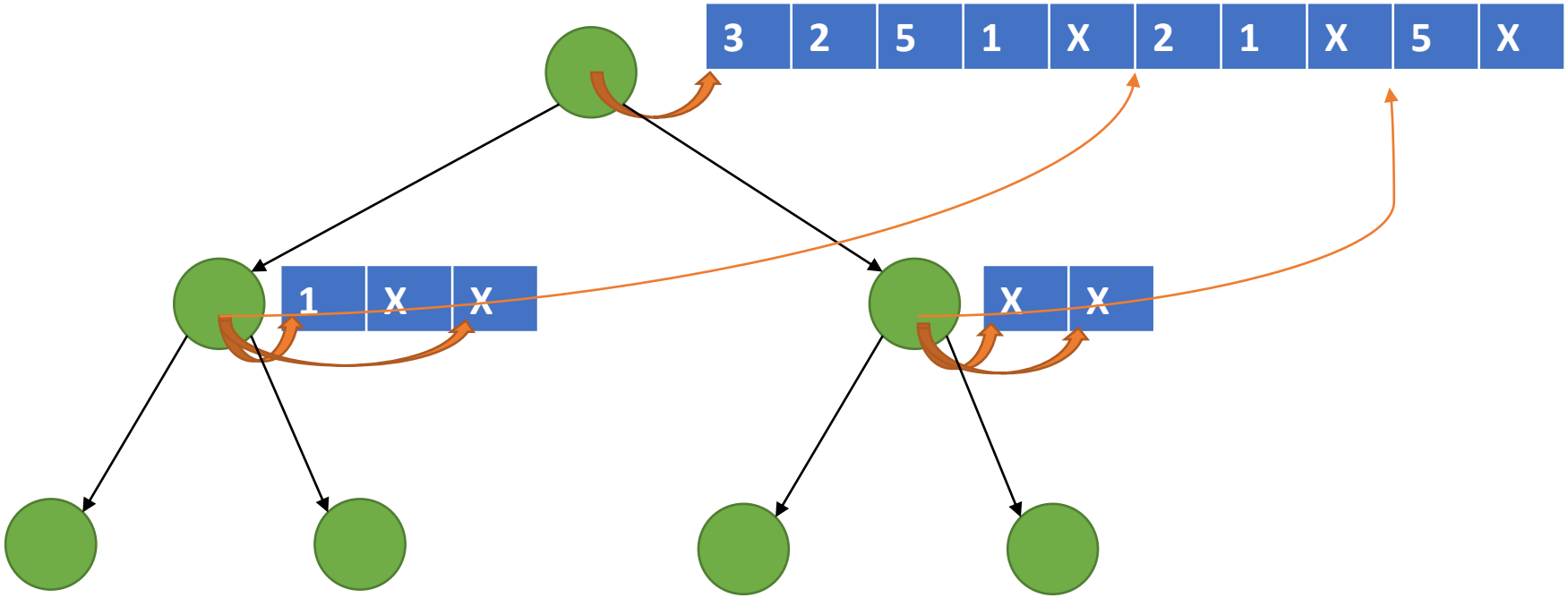
Idea: Give each thread its own heap



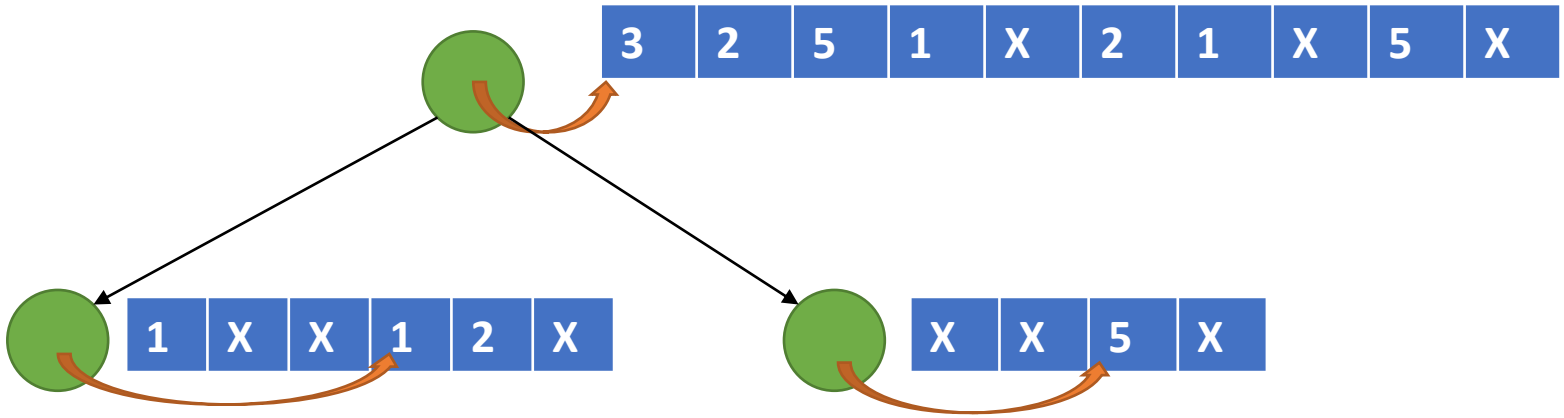
Idea: Give each thread its own heap



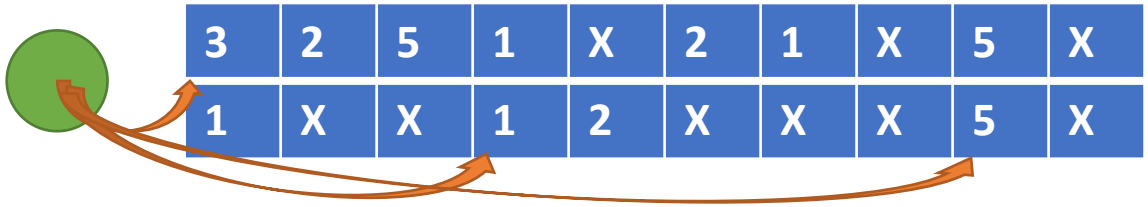
Idea: Give each thread its own heap



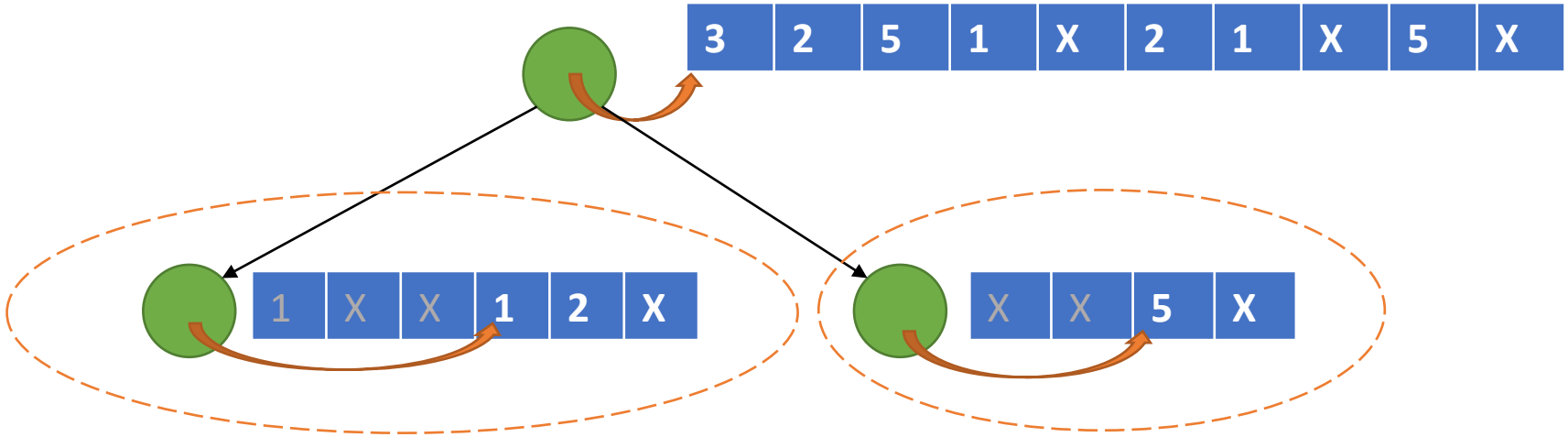
Idea: Give each thread its own heap



Merge heaps with parent when threads finish

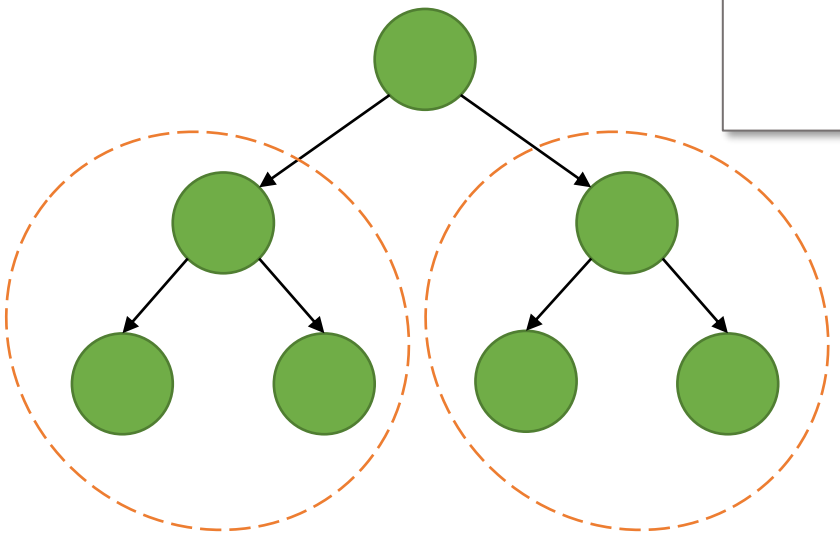


Key point: In FP, pointers only go up or down in the heap hierarchy ("disentanglement")



Can GC any leaf heap!

In general, can GC any subtree without stopping other threads



Hierarchical Memory Management for Parallel Programs

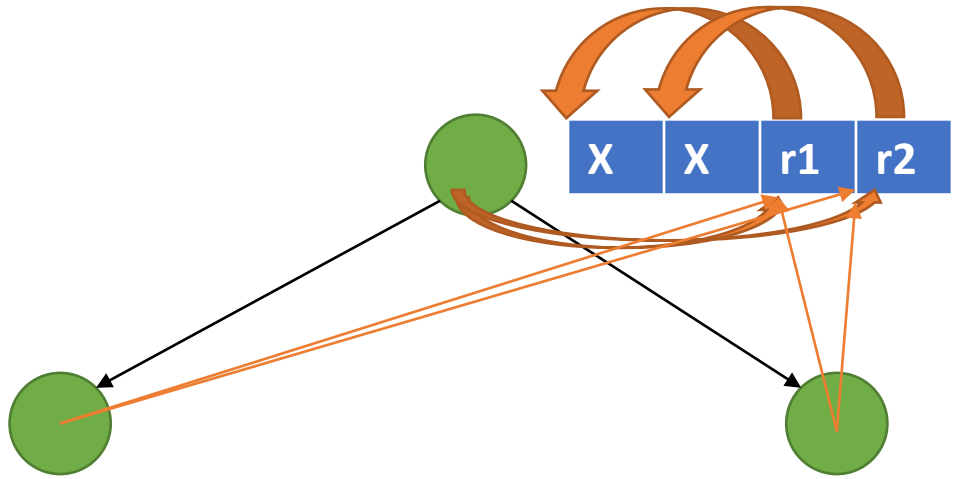
Ram Raghunathan* Stefan K. Muller* Umut A. Acar*† Guy Blelloch*

*Carnegie Mellon University, USA †Inria, France

{ram.r, smuller, umut, blelloch}@cs.cmu.edu

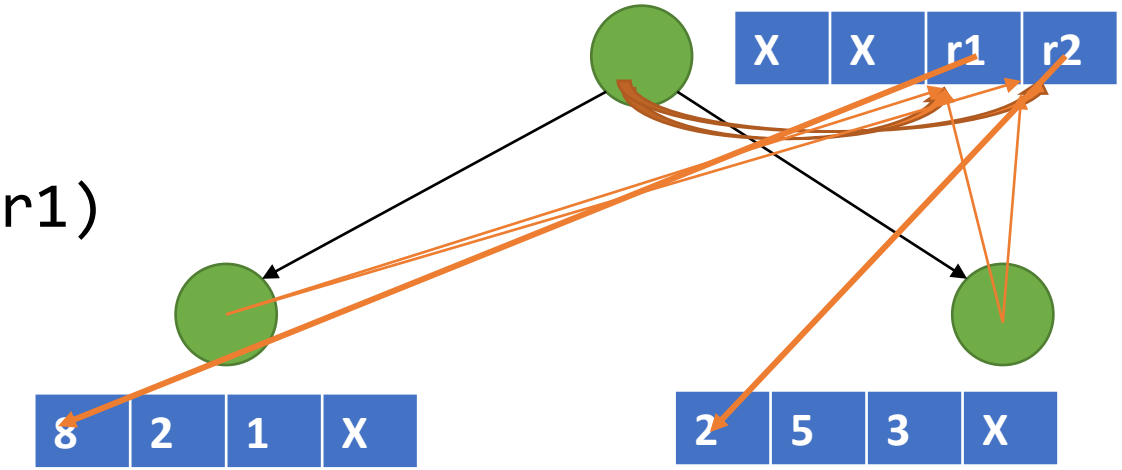
Disentanglement isn't guaranteed with side effects

```
let set_rand (mine: int list ref) (other: int list ref) =
  lr := random_list ();
  (!mine) @ (!other)
in
let r1: int list ref = ref [] in
let r2: int list ref = ref [] in
par (set_rand r1 r2, set_rand r2 r1)
```



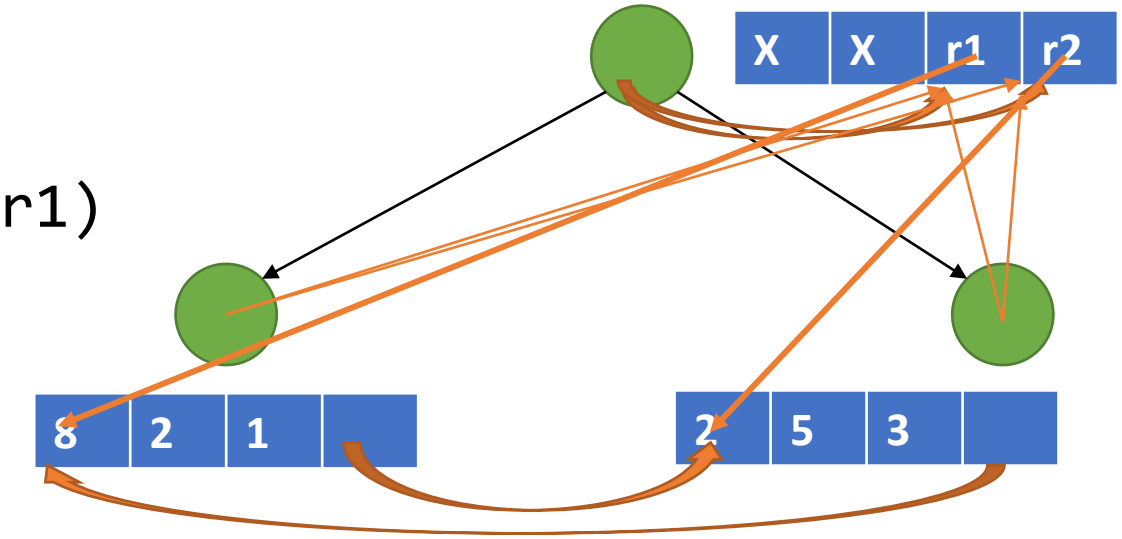
Disentanglement isn't guaranteed with side effects

```
let set_rand (mine: int list ref) (other: int list ref) =
  lr := random_list ();
  (!mine) @ (!other)
in
let r1: int list ref = ref [] in
let r2: int list ref = ref [] in
par (set_rand r1 r2, set_rand r2 r1)
```



Disentanglement isn't guaranteed with side effects

```
let set_rand (mine: int list ref) (other: int list ref) =
  lr := random_list ();
  (!mine) @ (!other)
in
let r1: int list ref = ref [] in
let r2: int list ref = ref [] in
par (set_rand r1 r2, set_rand r2 r1)
```



Actually, disentanglement is guaranteed as long as there are no data races

Disentanglement in Nested-Parallel Programs

SAM WESTRICK, Carnegie Mellon University, USA
ROHAN YADAV, Carnegie Mellon University, USA
MATTHEW FLUET, Rochester Institute of Technology, USA
UMUT A. ACAR, Carnegie Mellon University, USA