Cornell Bowers C·IS
Computer Science

CS3410: Computer Systems and Organization

LEC27: Parallel Programming with Maps and Reductions

Dr. Kevin Laeufer
Wednesday, December 3, 2025

Credits: Garcia, Laeufer

The Map Function

- Apply a custom function to each element.
- Square each number:

```
fn square(n: i32) -> i32 { n * n }
[1, 2, 3].into_iter().map(square)
Result: [1, 4, 9]
```

- Using anonymous functions:

```
[1, 2, 3].into_iter().map(|n| n * n)
Result: [1, 4, 9]
```

The Map Function: Chaining

- We can chain calls to map:

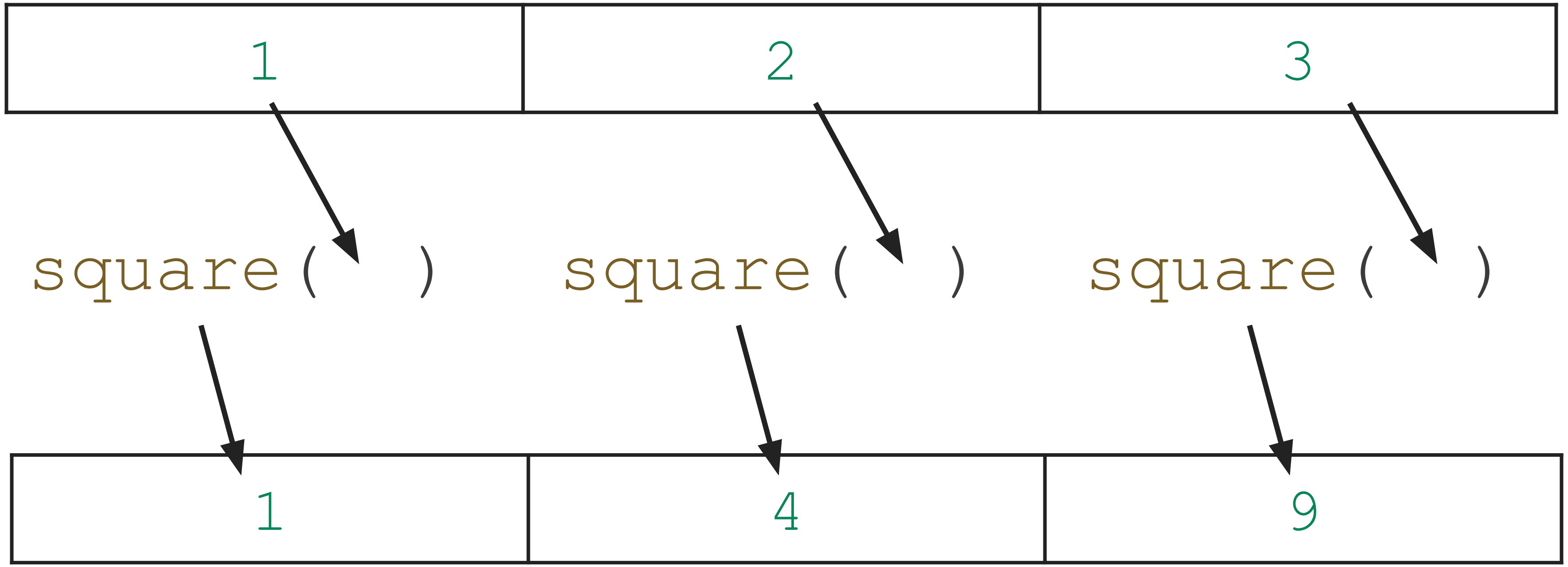
```
[1, 2, 3].into_iter().map(|n| n * n).map(|n| n + 1)
Result: [2, 5, 10]
```

- This is equivalent to:

```
[1, 2, 3].into_iter().map(|n| (n * n) + 1)
Result: [2, 5, 10]
```

The Map Function: Dataflow View

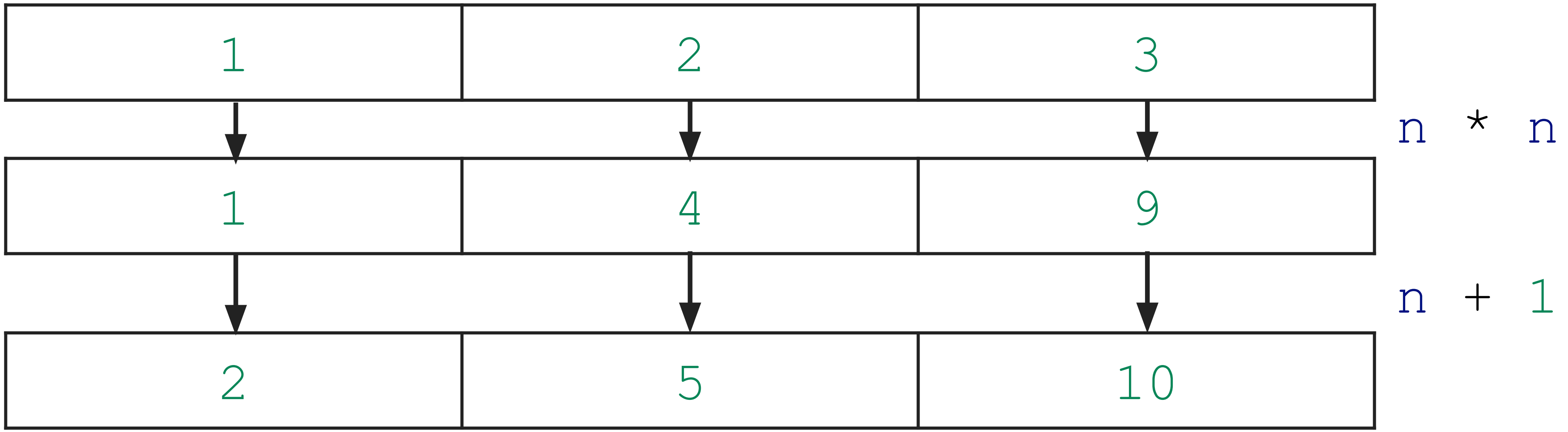
- `[1, 2, 3].into_iter().map(square)`



- The <u>order</u> in which we compute the individual squares <u>does not matter</u>.

The Map Function: Dataflow View

- `[1, 2, 3].into_iter().map(|n| n * n).map(|n| n + 1)`



- We can interleave the computation of the map calls to save memory bandwidth.

From Lectrue 25: Computing Primes

```
bool is_prime(int n) {
 for (int i = 2; i < n; ++i) {
   if (n % i == 0) { return false; }
 }
 return true;
}

// ...
static const int NUMBERS = 1024;

for(int i = 1; i < NUMBERS; i += 1) {
 is_prime(i);
}
```

- We can rewrite the loop using map:

```
(0..NUMBERS).into_iter()
             .map(is_prime)
```

- Can we use `map` inside `is_prime`?
- The loop in `is_prime` is meant to exit early, does not fit the standard `map` pattern.

From Lectrue 25: Computing Primes

- Calculating whether a number is prime using `map`:

```
(0..NUMBERS).into_iter().map(is_prime)
```

- The Rust library **rayon** can parallelize calls to `map` (and much more!). All we have to do is:

```
(0..NUMBERS).into_par_iter().map(is_prime)
```

- **Demo**: measure speed with hyperfine
- *Note* how we did not have to decide how we want to partition the work across threads. Rayon uses as many threads as CPU cores and work stealing for balance.

Reductions

- How can we calculate the number of primes?

```
let mut count = 0;
(0..NUMBERS).into_iter()
            .map(is_prime)
            .map(|p| { count += p as u32; p });
println!("count={count}");
```

- This prints: `count=0`

Reductions

- How can we calculate the number of primes?

```
let mut count = 0;
let _: Vec<_> = (0..NUMBERS).into_iter()
                .map(is_prime)
                .map(|p| { count += p as u32; p })
                .collect();
println!("count={count}");
```

This tells Rust to actually perform the computation and *collect* the results in an array.

- This prints: `count=1902`

Reductions

```
error[E0594]: cannot assign to `count`, as it is a captured
variable in a `Fn` closure
  --> src/main.rs:87:74
   |
87 | ...s_prime).map(|p| { count += p as u32; p }).collect();
   |                       ^^^^^^^^^^^^^^^^^ cannot assign
   |
```

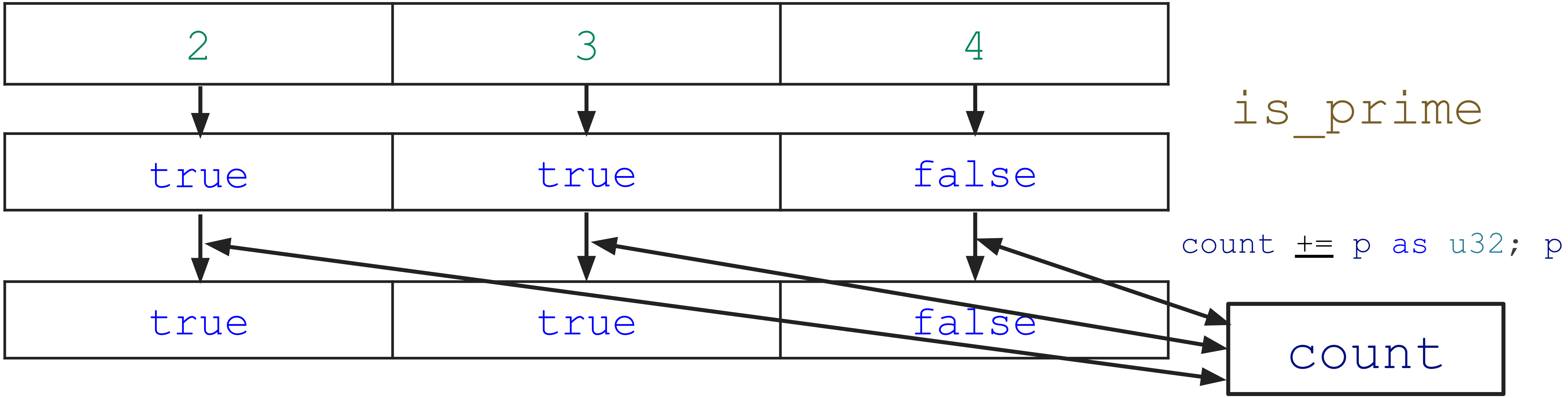
- Now in parallel?

```
let mut count = 0;
let _: Vec<_> = (0..NUMBERS).into_par_iter()
                .map(is_prime)
                .map(|p| { count += p as u32; p })
                .collect();
println!("count={count}");
```

- Does not compile! Cannot assign shared variable!

Reductions

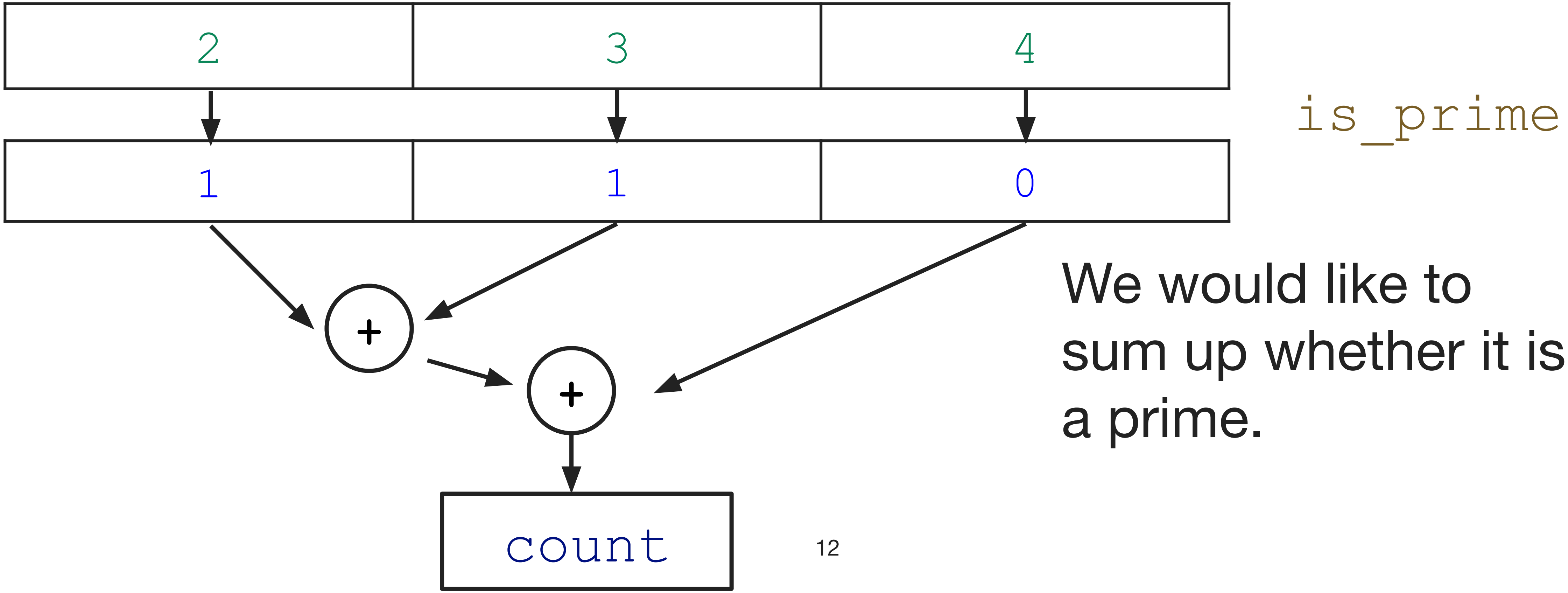
- `[2, 3, 4].into_iter()`
  `.map(is_prime)`
  `.map(|p| { count += p as u32; p })`



All applications of the second map read and write the same global variable!

Reductions

- ```
  [2, 3, 4].into_iter()
            .map(is_prime)
            .sum()
  ```



We would like to sum up whether it is a prime.
  ```

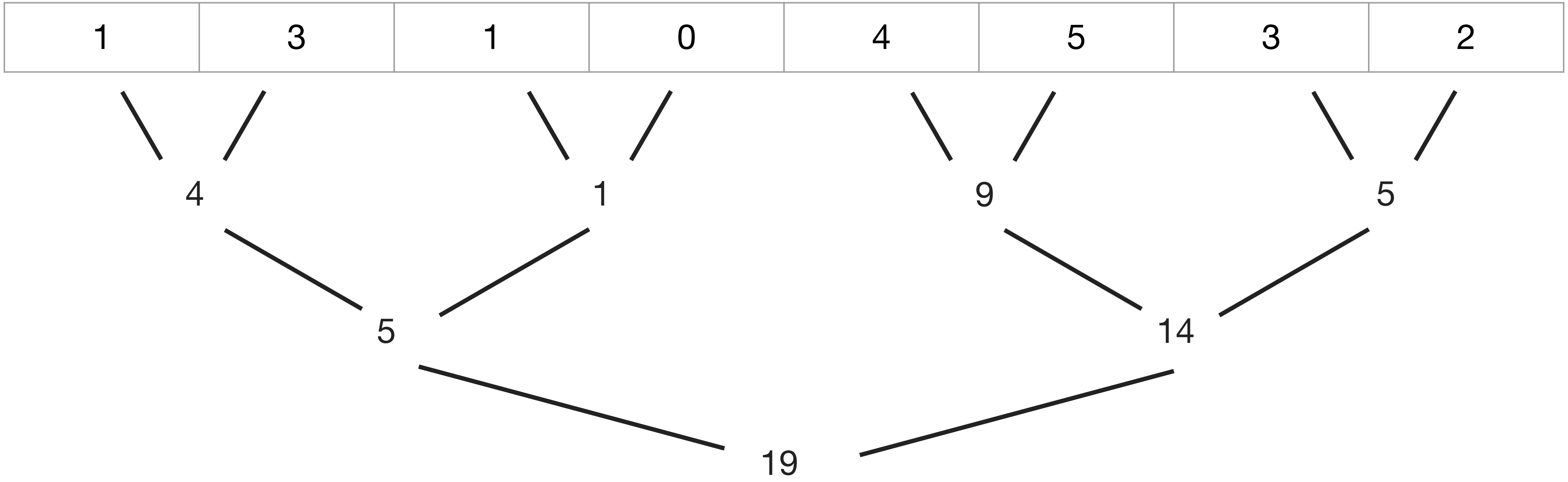
Reductions: from specialized to generic

- Three different ways to get the same result:
- `.sum()`
- `.reduce(|a, b| a + b).unwrap()`
- `.fold(0, |a, b| a + b)`
- And now in parallel:
-

```
let count: u32 =
(0..NUMBERS).into_par_iter().map(is_prime)
            .map(|p| p as u32)
            .reduce(|| 0, |a, b| a + b);
```

Parallel Reductions

- Tree reduction for maximum parallelism:



- Number of additions: O(n) in the serial implementation, O(n log(n)) in the tree reduction.

Recap: map and reduce for parallel execution.

- **Map and reduce** style computation provides a lot of freedom on how exactly the result is computed.
- The runtime can schedule small user provided functions across lots of data and **many CPU cores**.
- As long as we use *pure* functions that **do not modify shared variables**, the user will get the same result, no matter whether we execute in parallel or not.
- The **Rayon library** allows you to easily parallelize computation, as long as you can express it using map and reduce.

Cluster Scale MapReduce

- So far: all our data is in memory; we use standard map and reduce on a single CPU.
- Google's problems in the early 2000s:
    - TBs of data from logs, web crawling, etc.
    - Machines only have a 2 core CPU + 4GB of RAM
    - Files are distributed across machines
    - Sending data from one machine to another is slow.

Dean, Jeffrey, and Sanjay Ghemawat.
"Mapreduce: Simplified Data Processing on Large Clusters."
In OSDI 2004.

Example from the paper: Word count.

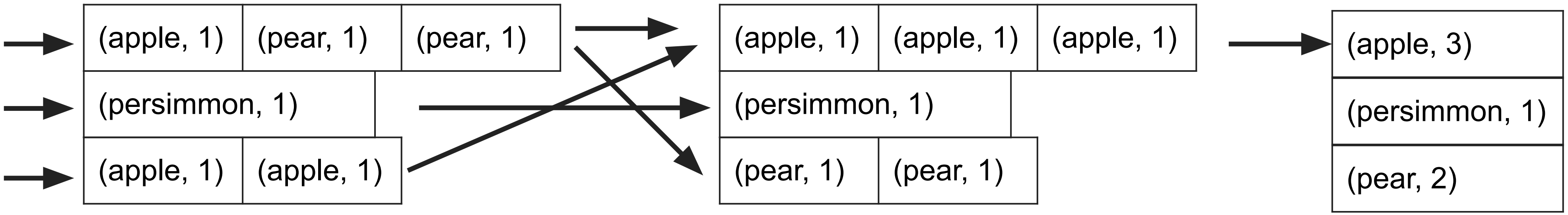
Actually a **flat_map** since we produce multiple outputs per input.

Implicit **grouping** by key**.**

“map”

“reduce”

file	content
a.txt	apple pear pear
b.txt	persimmon
c.txt	apple apple




```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

System Setup

- From the start, files are distributed across machines.
- **Map** can be computed on a machine that has the file already available.
- **Grouping** requires global synchronization through a **coordinator** machine.
- **Reduction** happens locally, since all values with the **same key** end up on the **same machine**.

Advantages

- Computation can be **distributed across machines**.
- Implicit grouping by key ensures **locality**!
- Tasks can be **load balanced** across machines.
- **Failure Resistance:** If a machine fails, a map or reduction task can be re-run on a different machine.

Disadvantages

- Only two operators: reduced flexibility.
- Explicit loops in reduction and map implementation cannot be parallelized by the system.
- Cannot easily chain multiple map and reduce calls without writing to disk.

(slightly more) Modern Alternative: Apache Spark

- Allows fine grained and more standard map/reduce style operations.
- Most of the work is done in-memory, and results can be reused.
- Word count in Spark:

```
val wordCounts = textFile
          .flatMap(line => line.split(" "))
          .groupByKey(identity)
          .count()
```

Takeaways

- Expressing computation in a functional style with map and reduce operators allows for **flexible compute** schedules.
- **Map** can safely be **parallelized** as long as the function we are applying does not modify shared state (like global variables).
- **Reduce** can be **parallelized** efficiently as long as the reduction function is **associative**.
- **Map** can work across CPU cores and across machines.
- Global **grouping by key** allows for efficient parallel “reductions” across machines.
- **Spark** supports more fine grained parallelism and reuse of intermediate results without going to disk.

Best of luck on the exam!

- This was my final lecture.
- Please keep exploring:
  - Compilers: CS 4120
  - Operating Systems: CS 4410
  - Floating-Point Math: CS 4210, CS 4220
  - Computer Architecture: CS 4420
  - Self-study Rust