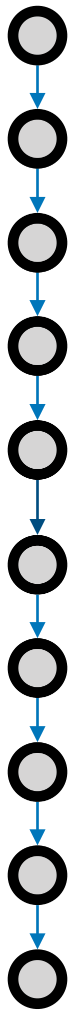
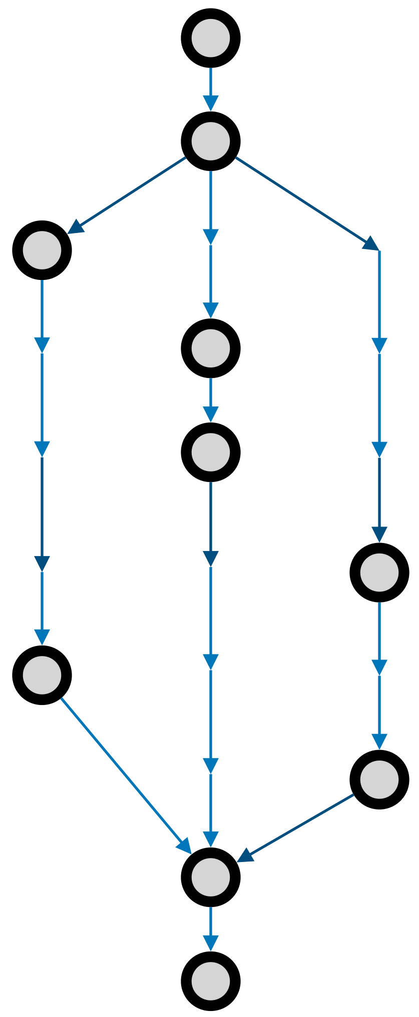
Lecture 10

Parallel Streams

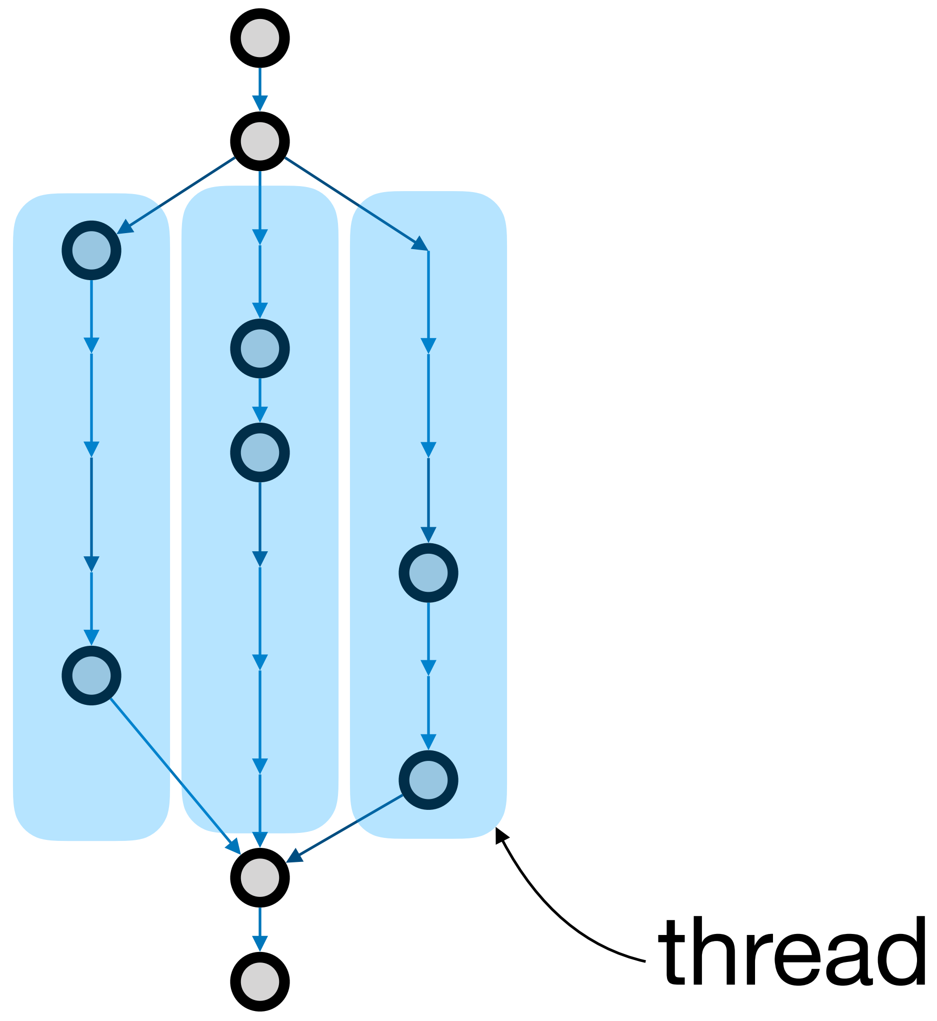
Sequential Program



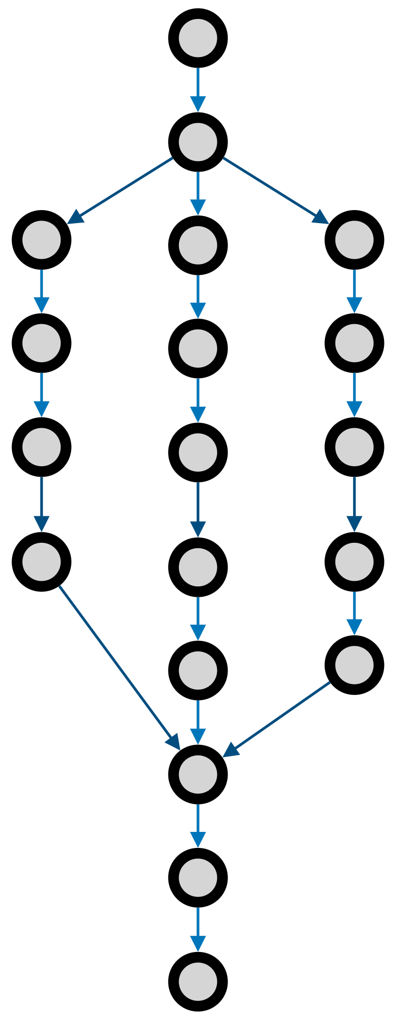
Concurrent Program



Concurrent Program

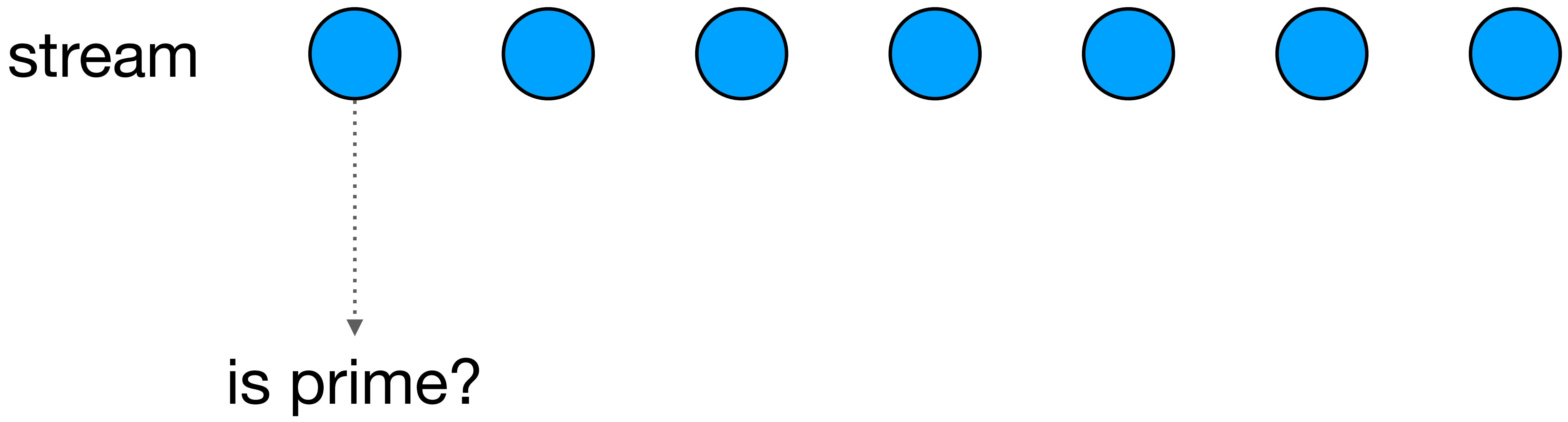


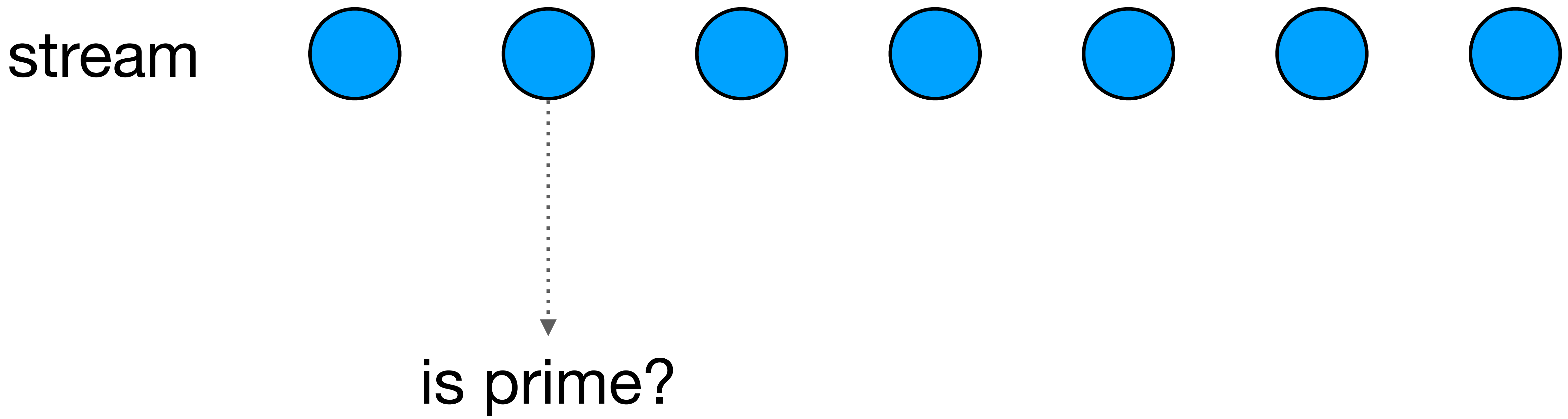
Parallel Program

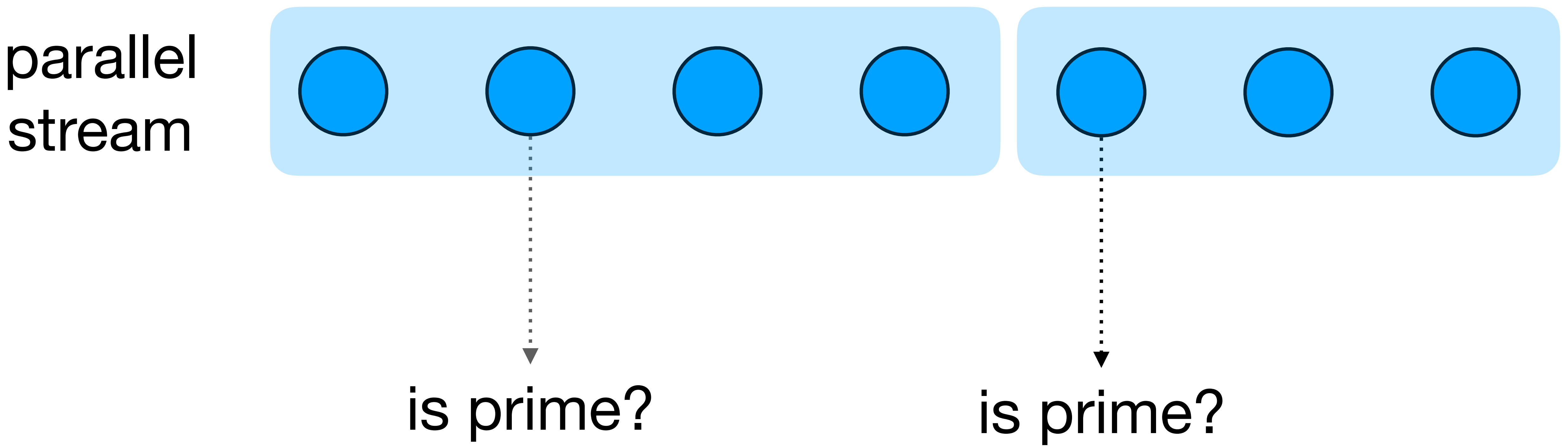


In modern computing devices
with multiple processors/cores,
the lines between parallelism and
concurrency is blurred.

```java
IntStream.range(1, 1_000_000)
     .parallel()
     .filter(x -> isPrime(x))
     .forEach(System.out::println);
```


stream
is prime?


stream
is prime?

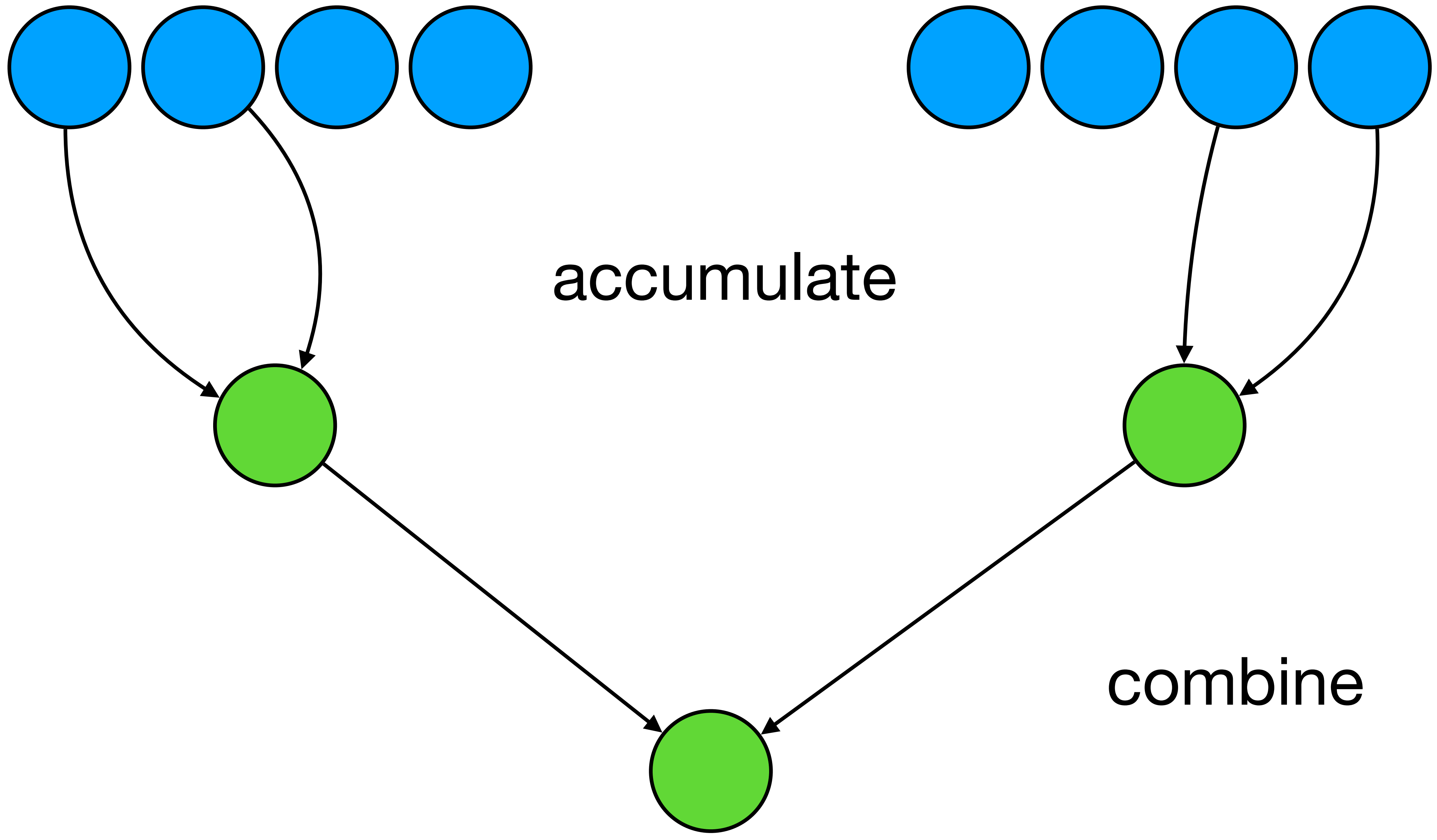

parallel stream
is prime?
is prime?

```
IntStream.range(1, 1_000_000)
     .filter(x -> isPrime(x))
     .forEach(System.out::println);
```

Good for Parallelization

- Non-interference
- Stateless
- No side-effect
- Associative Reduction

```
<U> U reduce(
    U identity,
    BiFunction<U,? super T,U> accumulator,
    BinaryOperator<U> combiner)
```


accumulate
combine

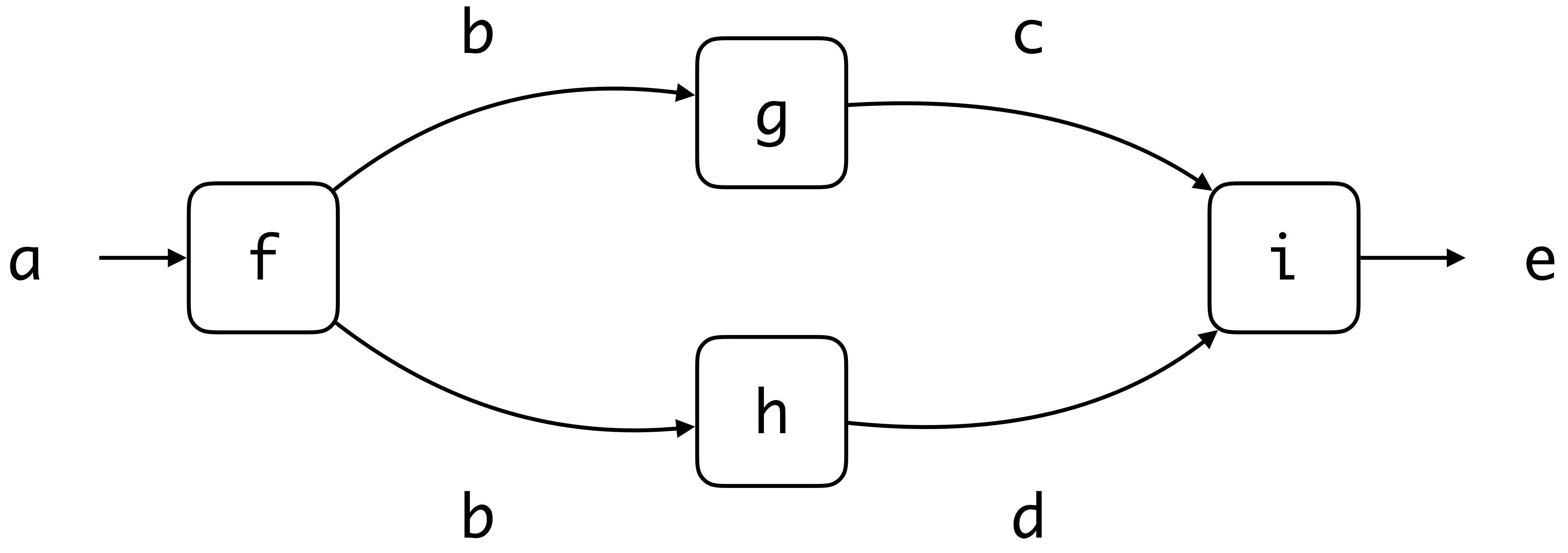
- `combiner.apply(identity, i)` must be equal to i.
- The `combiner` and the `accumulator` must be associative -- the order of applying must not matter.
- The `combiner` and the `accumulator` must be compatible:

  ```
    combiner.apply(u,
       accumulator.apply(identity, t)
    )
  ```

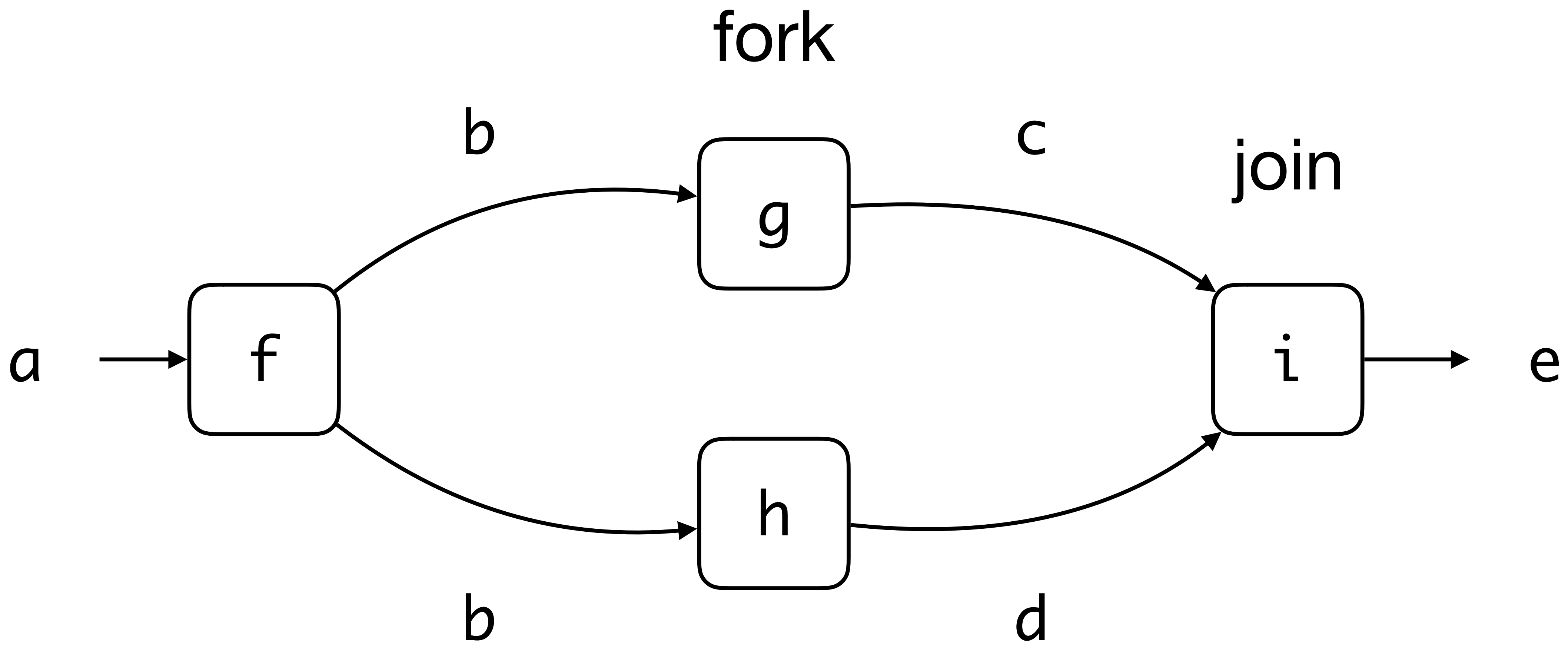
  must equal to

  ```
   accumulator.apply(u, t)
  ```

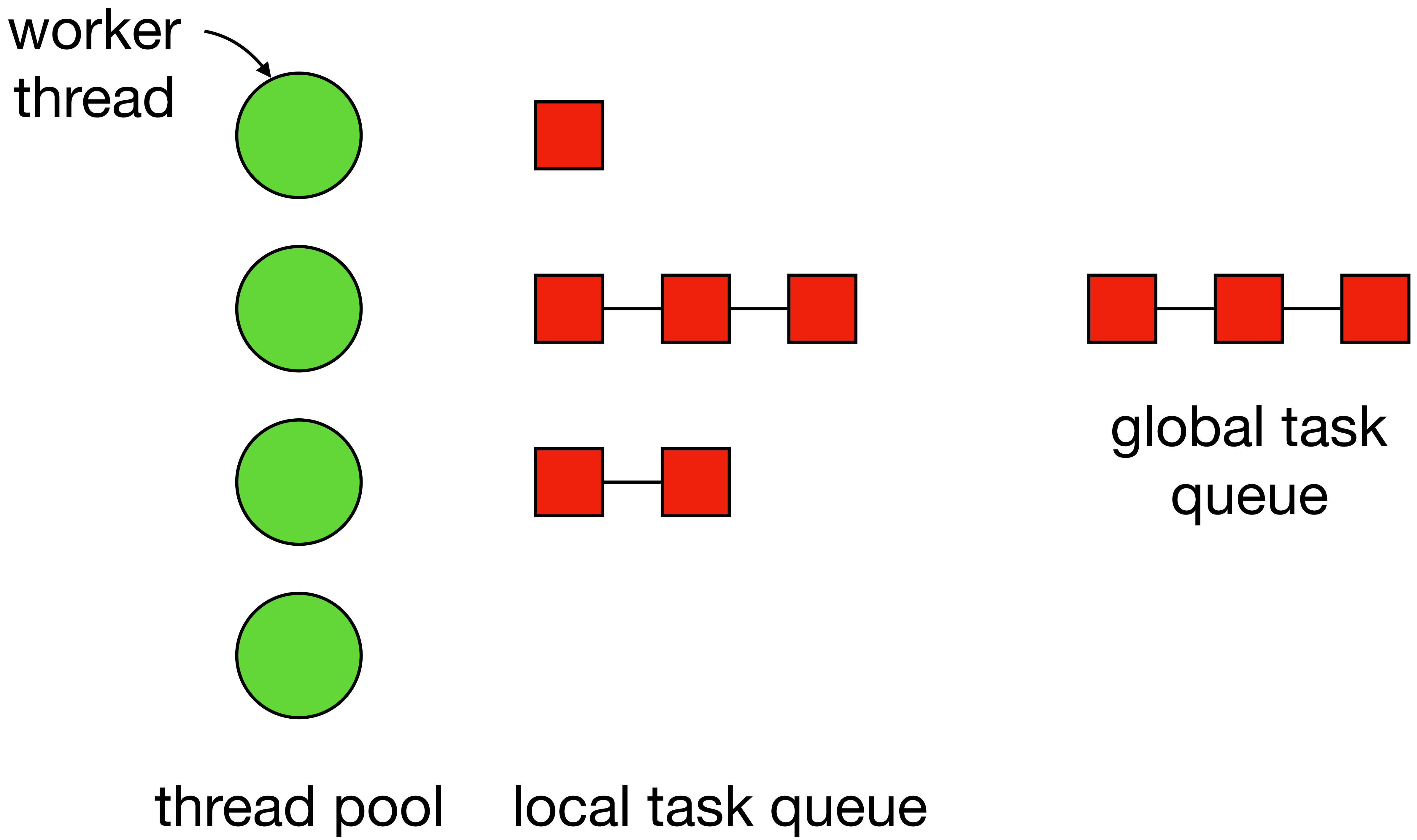
$$b = f(a)$$
$$c = g(b)$$
$$d = h(b)$$
$$e = i(c,d)$$



$$\begin{tikzcd}
 & \boxed{g} \arrow[dr, bend left, "c"] & & \\
a \arrow[r] & \boxed{f} \arrow[u, bend left, "b"] \arrow[d, bend right, "b"'] & & \boxed{i} \arrow[r] & e \\
 & \boxed{h} \arrow[ur, bend right, "d"'] & &
\end{tikzcd}$$


fork
b
c
join
a
f
g
i
e
h
b
d

ForkJoinPool



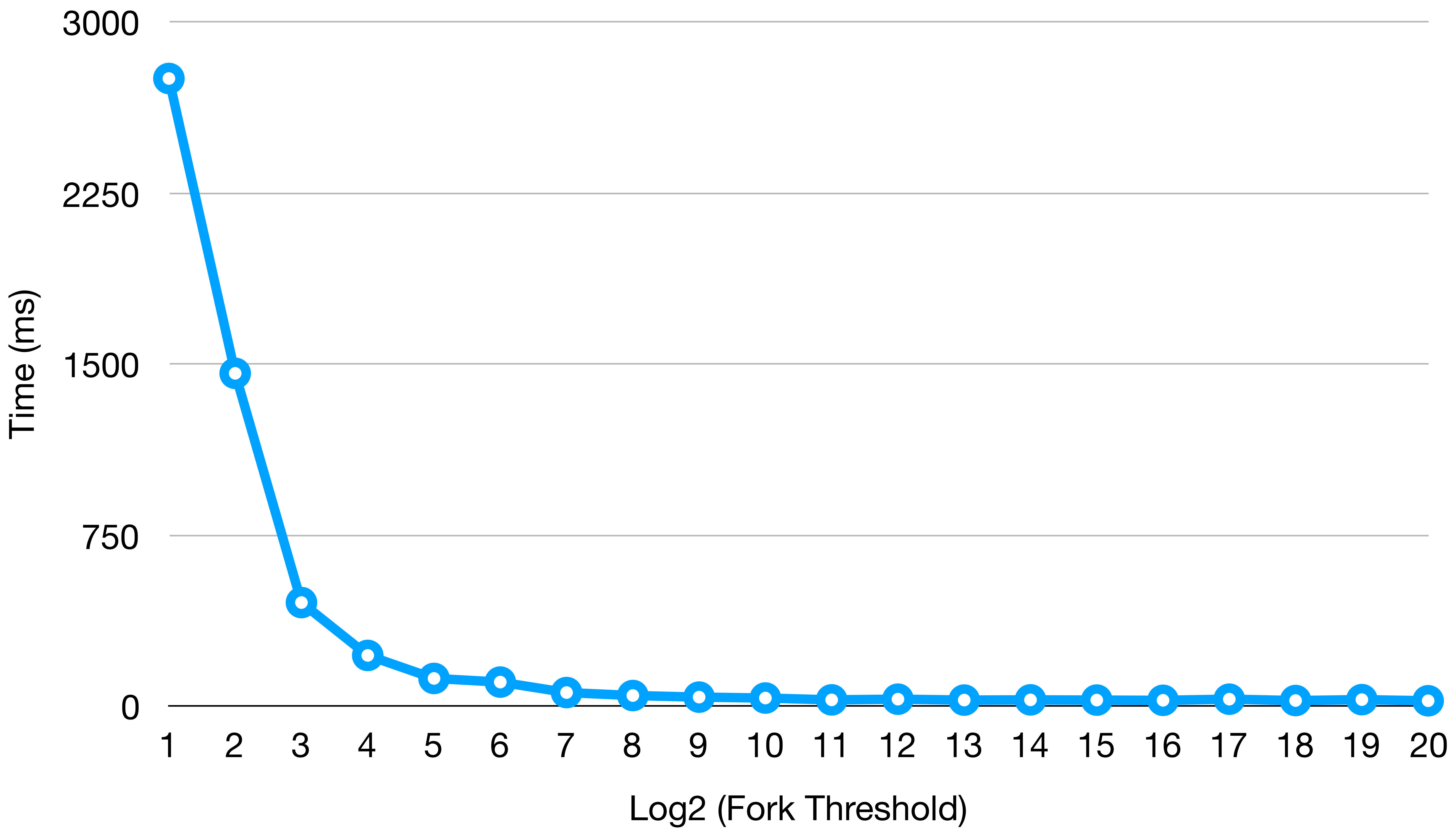
Implementing Tasks

- Inherit from `RecursiveTask<T>` or `RecursiveAction`
- Override the `compute` method
- `fork` to further subdivide
- `join` to wait for result

Submit Task for Execution

- Use default thread pool: `ForkJoinPool.commonPool()`
- Invoke using `invoke(task)`

Time vs. Fork Threshold



Parallelization

- Parallelism is not free
- Need to parallelise carefully

Ordered vs Unordered

- **Ordered:**
  - `Stream.of(..)`
  - `Stream.iterate(..)`
  - `List.stream()`
- **Unordered:**
  - `Stream.generate(..)`
  - `Set.stream()`

Order is important:

- `limit(n)`
- `skip(n)`
- `findFirst()`
- `distinct()`
- `sorted()`