CS2030 Lecture 11

Parallel and Concurrent Programming

Henry Chia (hchia@comp.nus.edu.sg)

Semester 2 2018 / 2019

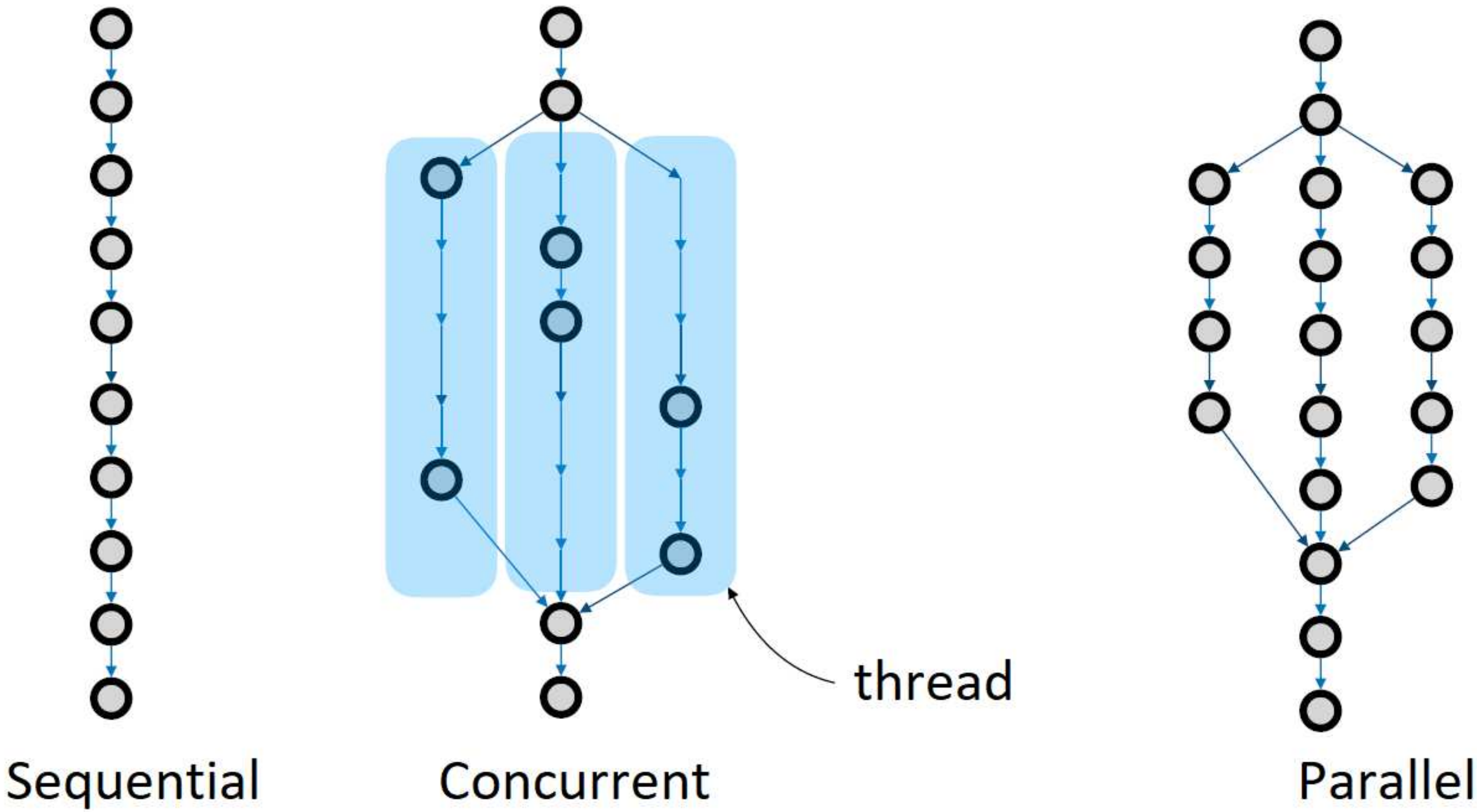
Lecture Outline

- Concurrency versus parallelism
- Parallel streams
  - Debugging parallel streams
  - Comparing sequential and parallel streams
- Correctness of parallel streams
  - `reduce` operator
  - Accumulator and combiner
- Fork and join in parallel streams
  - Overhead of parallelization

Concurrency vs Parallelism

- A single core processor executes one instruction at a time
  - Only one process can run at any one time
  - Context-switching allows multi-tasking on a single processor
- Concurrent programs run concurrently via threads
  - OS switches between threads
  - Separate unrelated tasks into separate threads
  - Improves processor utilization
- Parallel computing involves multiple subtasks running at the same time on multiple (possibly multi-core) processors
- Parallel programs are concurrent, but not all concurrent programs are parallel

Concurrency vs Parallelism



Parallel Streams

- Execute Streams in parallel to increase runtime performance
- Parallel streams use a common `ForkJoinPool` via the static `ForkJoinPool.commonPool()` method

```
System.out.println(ForkJoinPool.commonPool().getParallelism());
```

- The level of parallelism can be controlled by setting the following system property, either using

```
System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "4")
```

  or including the following flag when you running the program

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=4
```

- Collections support the method `parallelStream()` to create a parallel stream of elements
- Alternatively, the intermediate operation `parallel()` can be invoked on a given stream to parallelize a sequential stream

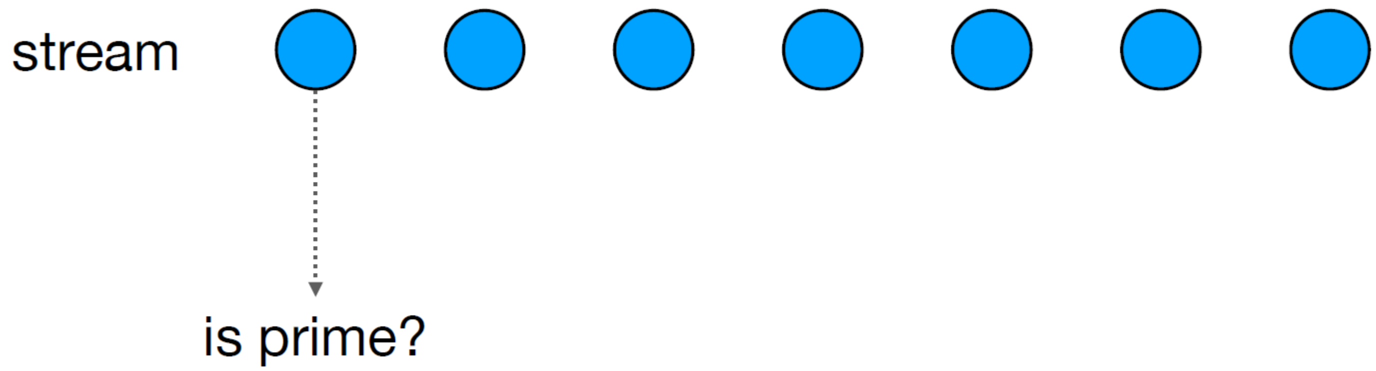
Parallel Streams

- Using prime number testing as an example

```java
static boolean isPrime(int n) {
    return IntStream
        .rangeClosed(2, (int) Math.sqrt(n))
        .noneMatch(x -> n % x == 0);
}
```

- Count number of primes between $2,000,000$ and $3,000,000$

```java
long count = IntStream.range(2_000_000, 3_000_000)
    .filter(x -> isPrime(x))
    .count();
```

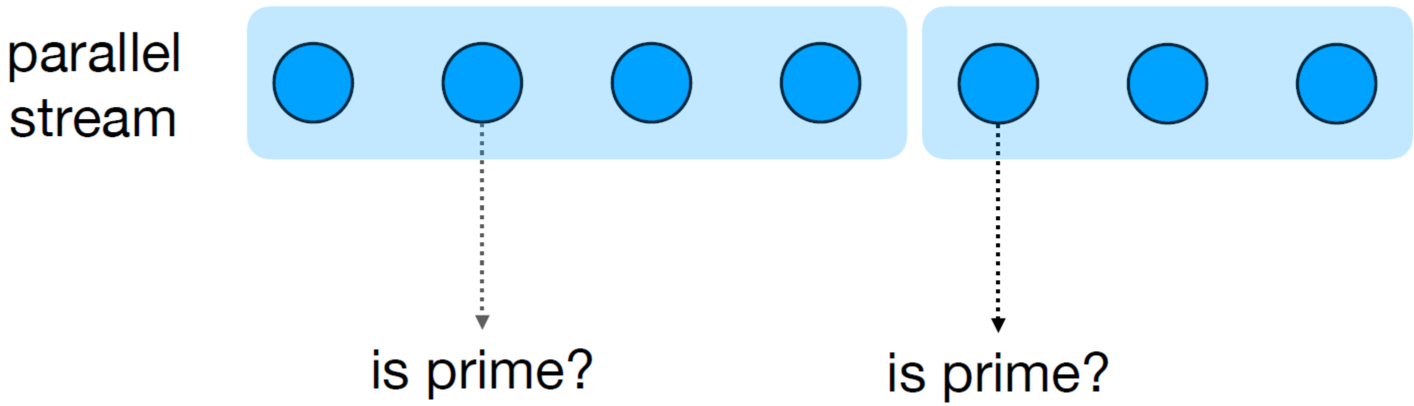


Parallel Streams

- Parallelizing the seach for primes

```java
long count = IntStream.range(2_000_000, 3_000_000)
        .parallel()
        .filter(x -> isPrime(x))
        .count();
```

- The `parallel()` intermediate operation turns on a boolean flag that switches the stream pipeline to be parallel
  - Invoked anywhere between the data source and terminal
  - The counter operation is `sequential()`



Parallel Streams

```
public static void main(String[] args) {
    if (args.length != 0) {
    System.setProperty(
        "java.util.concurrent.ForkJoinPool.common.parallelism",
        args[0]);
    }
    System.out.println("Number of worker threads: " +
            ForkJoinPool.commonPool().getParallelism());

    Instant start = Instant.now();
    long howMany = IntStream.range(2_000_000, 3_000_000)
        .parallel()
        .filter(x -> isPrime(x))
        .count();
    Instant stop = Instant.now();

    System.out.println(howMany + " : " +
            Duration.between(start, stop).toMillis() + "ms");
}
```

Debugging Parallel Streams

- To time the execution of a process,
  - `java.time.Instant`'s `now()` method returns the current `Instant` from the system clock
  - `java.time.Duration`'s `between()` returns the `Duration` of two `Instances` (an implementation of `Temporal`)
  - `Duration`'s `toMillis()`/`toNanos()`/... extracts the desired representation of the duration

```
java.util.Instant;
java.util.Duration;

Instant start, stop;
start = Instant.now();
/* perform some task */
stop = Instant.now();

long timeInMillis = Duration.between(start, stop).toMillis();
```

Debugging Parallel Streams

- To debug and manage each execution thread
  - Thread.currentT~~a~~hread() (or `Thread.currentThread().getName()`) to retrieve the identity of the thread
  - `Thread.sleep(long millis)` causes the currently executing thread to sleep (i.e. temporarily cease execution) for the specified number of milliseconds
    - Used within a **try.. catch** block
    - Example, letting a thread sleep for one second

```
try {
    ...
    Thread.sleep(1000);
    ...
} catch (InterruptedException e) { }
```

Debugging Parallel Streams

- Effect of parallelizing a stream

```java
int sum = IntStream.of(1, 2, 3, 4, 5)
    .parallel()
    .filter(x -> {
        System.out.println("filter:  " + x + " "
            + Thread.currentThread().getName());
        return true;
    })
    .map(x -> {
        System.out.println("map:     " + x + " "
            + Thread.currentThread().getName());
        return x;
    })
    .reduce(0, (x, y) -> {
        System.out.println("reduce:  " + x + " + " + y + " "
            + Thread.currentThread().getName());
        return x + y;
    });
    System.out.println(sum);
```

Comparing Sequential and Parallel Streams

- Suppose given the following task unit

```java
class OneSecondTask {
    int ID;

    public OneSecondTask(int ID) {
        this.ID = ID;
    }

    public int compute() {
        System.out.println(Thread.currentThread().getName());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        return ID;
    }
}
```

Comparing Sequential and Parallel Streams

```java
public static void sequentialRun(List<OneSecondTask> tasks) {
    Instant start = Instant.now();
    List<Integer> result = tasks.stream()
        .map(x -> x.compute())
        .collect(Collectors.toList());
    Instant stop = Instant.now();
    System.out.print(result + " ");
    System.out.println(Duration.between(start,stop).toMillis() + "ms")
}
```

- Sequential stream on 4 worker threads:

```
main
main
main
main
main
main
main
main
main
main
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] 10003ms
```

Comparing Sequential and Parallel Streams

```java
public static void parallelStreamRun(List<OneSecondTask> tasks) {
    Instant start = Instant.now();
    List<Integer> result = tasks.parallelStream()
        .map(x -> x.compute())
        .collect(Collectors.toList());
    Instant stop = Instant.now();
    System.out.print(result + " ");
    System.out.println(Duration.between(start,stop).toMillis() + "ms")
}
```

- Parallel stream on 4 worker threads:

```
main
ForkJoinPool.commonPool-worker-1
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-2
ForkJoinPool.commonPool-worker-4
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-2
ForkJoinPool.commonPool-worker-4
ForkJoinPool.commonPool-worker-1
ForkJoinPool.commonPool-worker-3
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] 3006ms
```

Correctness of Parallel Streams

- To ensure correct parallel execution, stream operations must not interfere with stream data, preferably stateless and have no side effects
- Example of interference:

```
List<String> list = new ArrayList<>(
        List.of("abc", "def", "xyz"));

list.stream()
    .peek(str -> {
        if (str.equals("xyz")) {
            list.add("pqr");
        }
    })
    .forEach(x -> {});
```

- Interference is not allowed in both sequential and parallel streams

Correctness of Parallel Streams

- Another example:

```java
List<Integer> list = new ArrayList<>(
        Arrays.asList(1, 3, 5, 7, 9, 11, 13, 15, 17, 19));
List<Integer> result = new ArrayList<>();
```

- The following is erroneous due to side effects

```java
list.parallelStream() // list.stream().parallel()
    .filter(x -> isPrime(x))
    .forEach(x -> result.add(x));
```

- Use `.collect` instead

```java
result = list.parallelStream()
    .filter(x -> isPrime(x))
    .collect(Collectors.toList());
```

- Side effects are a problem in parallel streams
- Consider using a thread-safe list, e.g. `CopyOnWriteArrayList`

Inherently Parallelizable `reduce`

- Consider `Stream`'s three-argument `reduce` method:

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

- Rules to follow when parallelizing
  - `combiner.apply(identity, i)` must be equal to `i`
  - `combiner` and `accumulator` must be associative, i.e. order of application does not matter
  - `combiner` and `accumulator` must be compatible, i.e. `combiner.apply(u, accumulator.apply(identity, t))` must be equal to `accumulator.apply(u, t)`
  - The following example compiles with the above rules:

```
Stream.of(1,2,3,4)
    .parallel()
    .reduce(1, (x,y) -> x * y, (x,y) -> x * y)
```

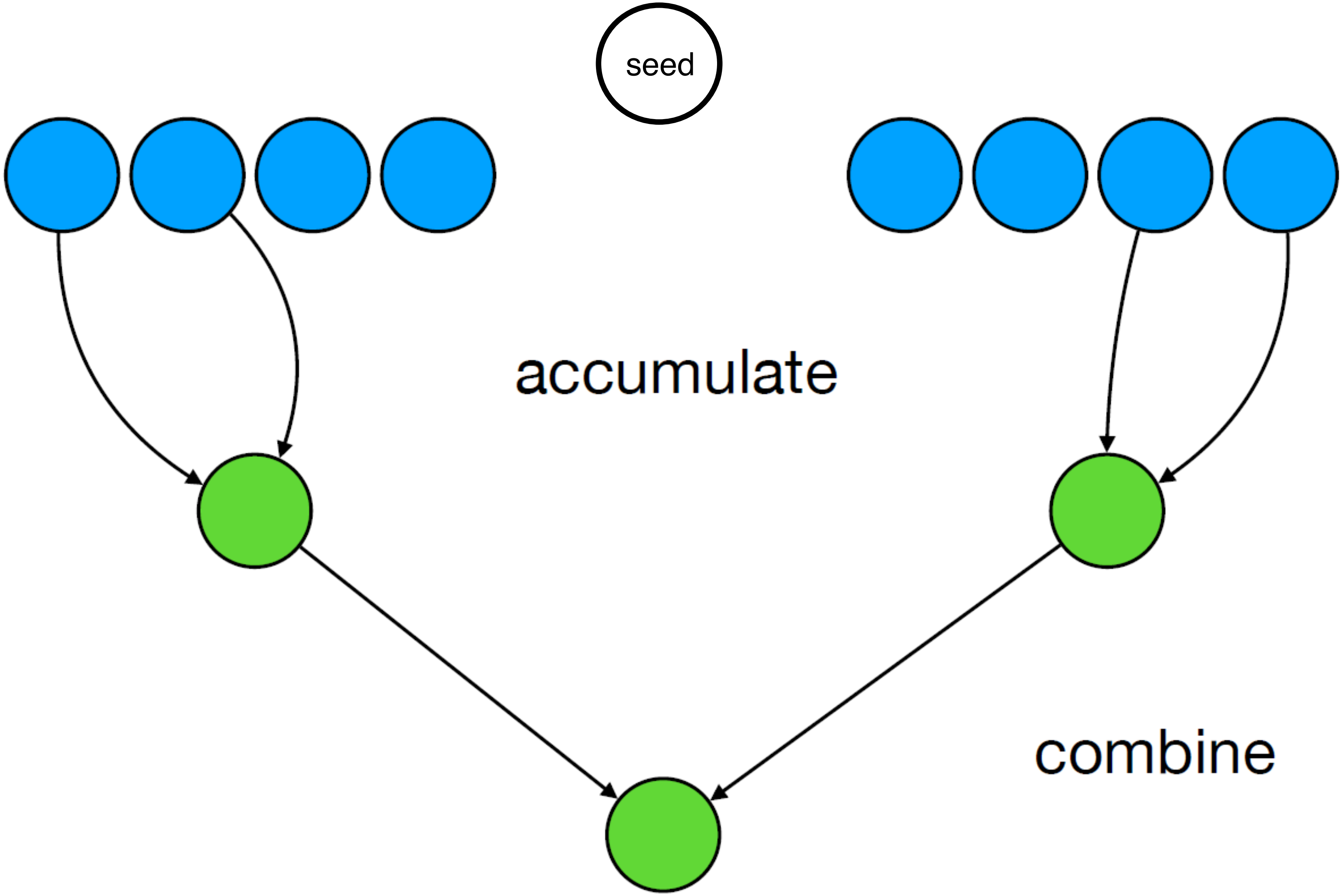
Accumulator and Combiner

- Accumulator and combiner functions are executed in parallel

NEVER System.out.println 2 times when using thread

```
String s = Stream.of(1, 2, 3, 4, 5)
    .parallel()
    .filter(x -> {
        System.out.println("filter:  " + x + " "
            + Thread.currentThread().getName());
        return true;
    })
    .map(x -> {
        System.out.println("map:     " + x + " "
            + Thread.currentThread().getName());
        return x;
    })
    .reduce("",
        (x, y) -> {
            System.out.println("accumulate: " + x + " + " + y + " " +
                    Thread.currentThread().getName());
            return x + y;
            },
        (x, y) -> {
            System.out.println("combine: " + x + " + " + y + " "
                    + Thread.currentThread().getName());
            return x + y;
            }
    );
```

Accumulator and Combiner



Accumulator and Combiner

- Erroneous examples where rules are violated
  - `combiner.apply(identity, i)` not equal to `i`

```
double result = Stream
        .parallel()
        .of(1, 2, 3, 4)
        .reduce(1.0,
                (x, y) -> x + y,
                (x, y) -> x + y);
```

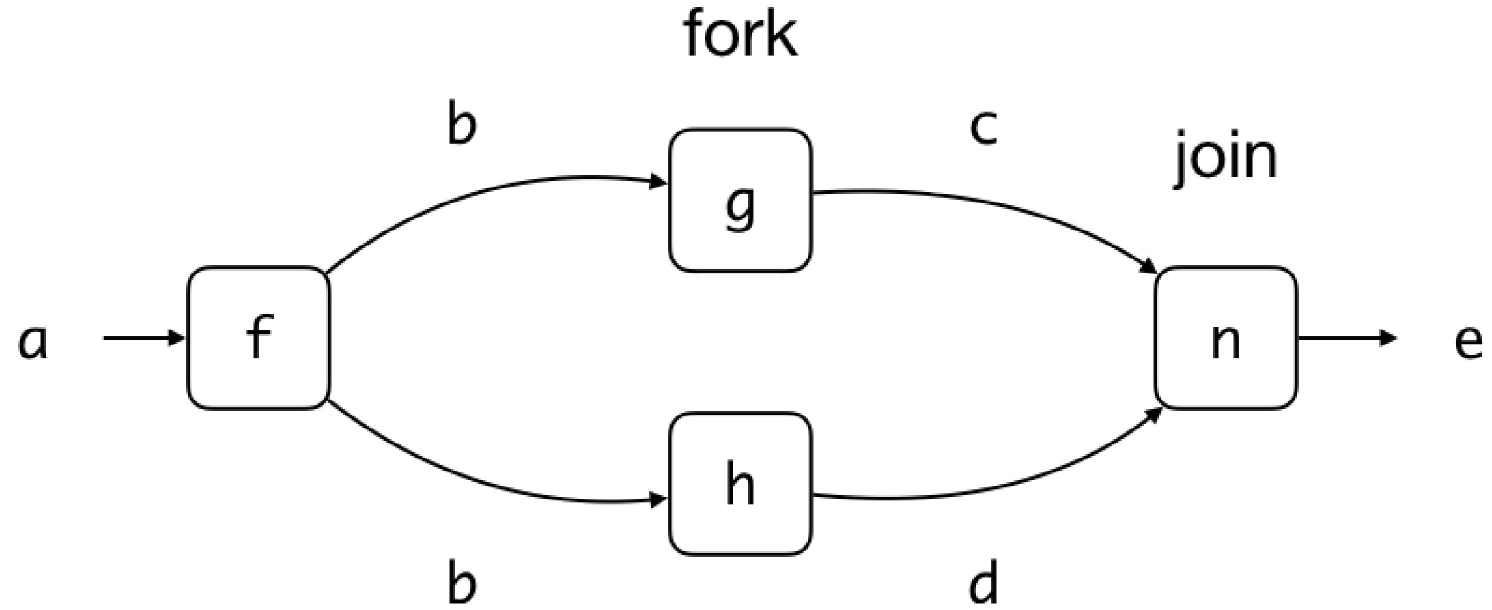
  - for division, the order of application **does** matter

```
double result = Stream
        .of(1, 2, 3, 4)
        .parallel()
        .reduce(24.0,
                (x, y) -> 1.0 * x / y,
                (x, y) -> 1.0 * x / y);
```

Fork and Join

- Given the following program fragment and *computation* graph

```
b = f(a);
c = g(b);
d = h(b);
e = n(c,d);
```



- `f(a)` invoked before `g(b)` and `h(b)`; `n(c,d)` invoked after
- How about the order of `g(b)` and `h(b)`?
  - If `g` and `h` does not produce side effects, then parallelize
  - **Fork** task `g` to execute at the same time as `h`, and **join** back task `g` later

Fork and Join in Parallel Streams

- `parallel()` runs `fork` to create sub-tasks running the same chain of operations on sub-streams
  - Processes for sub-tasks are run in multiple threads when appropriate
  - Threads are shared from a common **Fork Join Pool**
- `combiner` in `reduce` runs `join` to combine the results
- Should we exploit parallelism to the fullest?
  - Parallelizing primality testing

```java
return IntStream
    .rangeClosed(2, (int) Math.sqrt(n))
    .parallel()
    .noneMatch(x -> n % x == 0);
```

Fork and Join in Parallel Streams

- Parallelizing a trivial task actually creates more work in terms of parallelizing overhead
- Parallelization is worthwhile only if the task is complex enough that the benefit of parallelization outweights the overhead
  - In primality testing, checking `(n % x == 0)` is trivial;
  - Parallelizing it induces more overhead in terms of processing the forks and joins
- Holds true for all parallel and concurrent programs, either in the context of parallel streams or otherwise

Lecture Summary

- Familiarity with the use of sequential and parallel streams
- Able to compare performances between sequential and parallel streams
- Able to debug parallel streams
- Adherence to rules for parallelizing streams
- Appreciate fork and join in parallel streams
- Appreciate fork/join overhead