CS2030/S Lecture 11

From Sequential to Parallel Programming Using Java Streams

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

Semester 1 2020 / 2021

Lecture Outline

- Stream concepts
  - stream elements and pipelines
  - intermediate and terminal operations
  - stateless vs stateful operations
- Parallel streams
  - debugging parallel streams
- Correctness of parallel streams
  - `reduce` operator
  - accumulator and combiner
- Overhead of parallelization

Streams and Pipelines

- A stream pipeline starts with a **data source**
- **Intermediate operations** specify tasks to perform
- **Terminal operation** initiates the processing of a stream pipeline so as to produce a result
- Example using primitive stream `IntStream`

```
int sum = IntStream
    .rangeClosed(1, 10)
    .filter(x -> x % 2 == 0)
    .map(x -> x * 2)
    .sum();
```

- Intermediate operations use **lazy evaluation**
  - does not perform any operations on stream's elements until a terminal operation is called

Lazy Evaluation in Streams

- The following illustrates the movement of stream elements

```
int sum = IntStream
    .rangeClosed(1, 10)
    .filter(
        x -> {
            System.out.println("filter: " + x);
            return x % 2 == 0;
        })
    .map(
        x -> {
            System.out.println("map: " + x);
            return 2 * x;
        })
    .sum();
System.out.println(sum);
```

```
filter: 1
filter: 2
map: 2
filter: 3
filter: 4
map: 4
filter: 5
filter: 6
map: 6
filter: 7
filter: 8
map: 8
filter: 9
filter: 10
map: 10
sum is 60
```

User-defined Reductions

- Terminal operations can be in the form of **reductions**
- For example, using `reduce` in place of `sum`

```
int[] values = {7, 9, 5, 2, 8, 4, 1, 6, 10, 3};
IntStream
    .of(values)
    .reduce(0, (x, y) -> x + y));
```

  - first argument to `reduce` is the operation's identity value
  - second argument is the lambda that receives two `int` values, adds them and returns the result; in the above
    - first calculation uses identity value 0 as left operand
    - subsequent calculations uses the result of the prior calculation as the left operand
    - if stream is empty, the identity value is returned

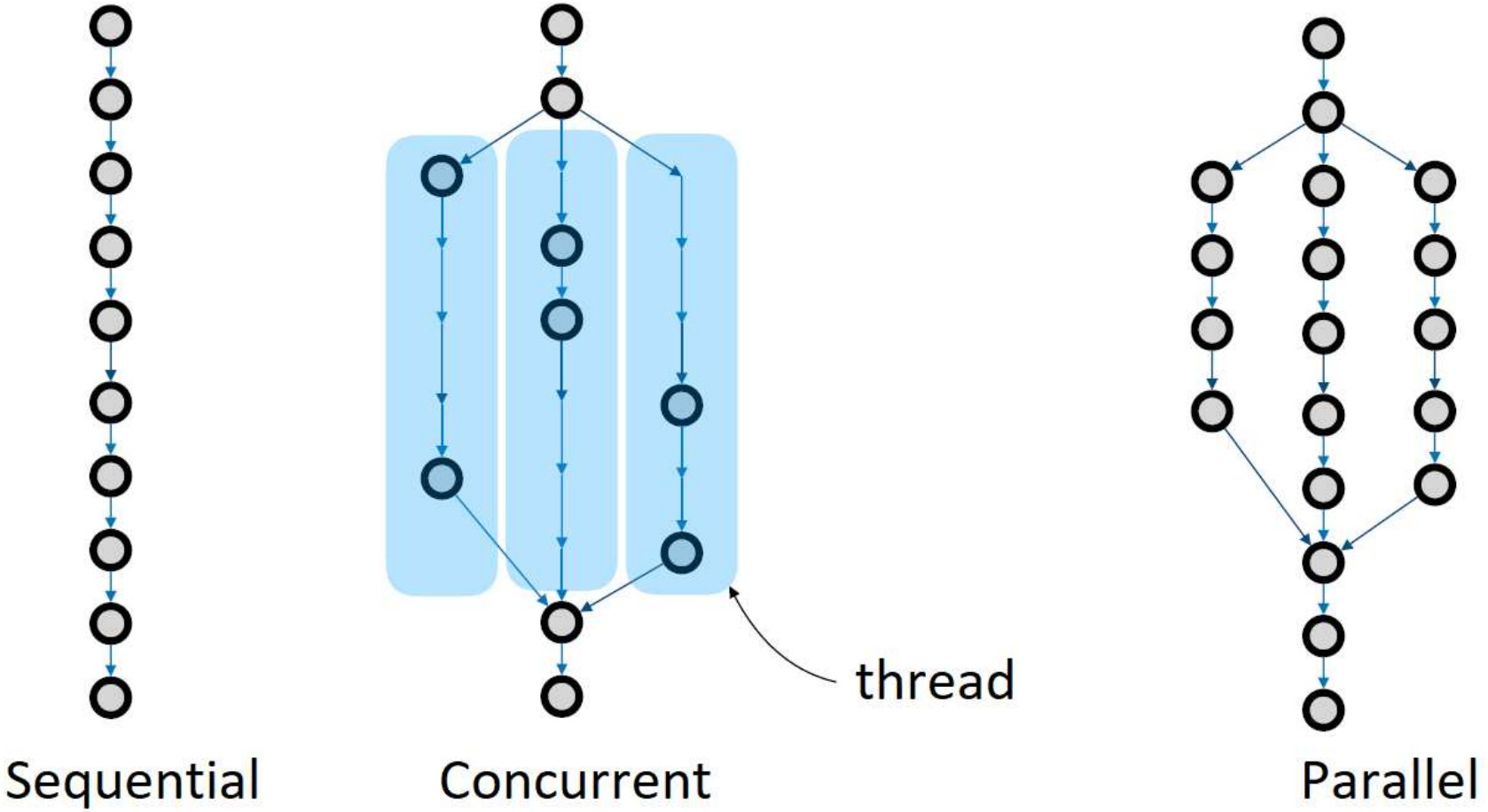
Stateless vs Stateful Operations

- Intermediate stream operations like `filter` and `map` are **stateless**, i.e. processing one stream element does not depend on other stream elements
- There are, however, **stateful** intermediate operations that depend on the current state
- E.g. stateful operations: `sorted`, `limit`, `distinct`, etc.

```
IntStream
    .of(7, 9, 5, 2, 8, 4, 1, 6, 10, 3)
    .sorted()
    .forEach(System.out::println);

IntStream
    .of(1, 1, 1, 0, 0, 0, 1, 0, 0, 1)
    .distinct()
    .forEach(System.out::println);
```

Concurrency vs Parallelism



Parallel Streams

- Parallel streams use a common `ForkJoinPool` via the static `ForkJoinPool.commonPool()` method

```
jshell> ForkJoinPool.commonPool().getParallelism()
$1 ==> 23
```

- The level of parallelism can be controlled by setting the system property during program run

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

  or including the following flag when running the program

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

- Similar to the method `stream()`, Java `Collection`(s) also support the method `parallelStream()` to create a parallel stream of elements

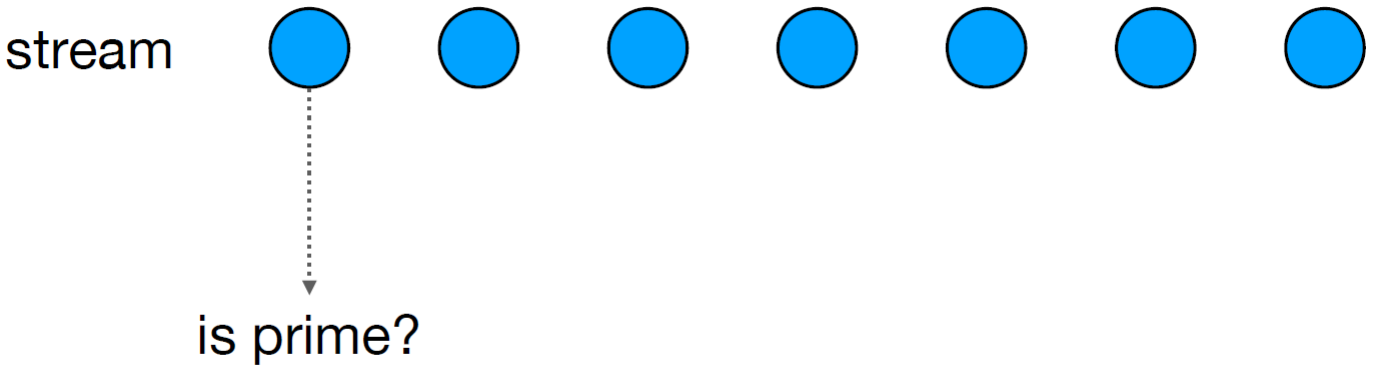
Parallel Streams

- Using prime number testing as an example

```
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 $3,000,000$

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

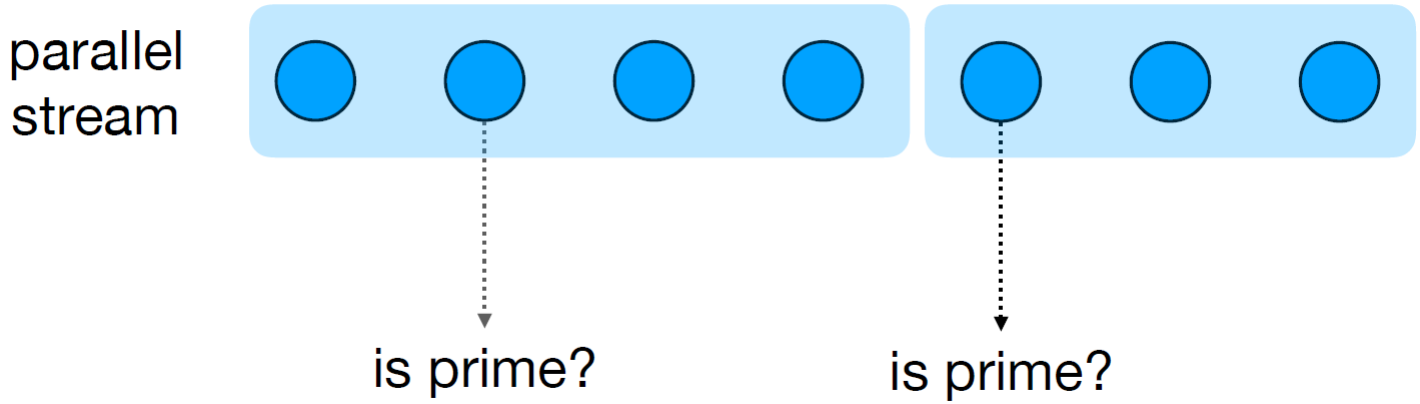


Parallel Streams

- Parallelizing the seach for primes

```
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

- 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();
```

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 debug and manage each execution thread
  - `Thread.currentThread()` (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

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

Correctness of Parallel Streams

- To ensure correct parallel execution, stream operations
  - must not interfere with stream data (true for sequential streams also)
  - preferably stateless with 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 -> {});
```

Correctness of Parallel Streams

- Example of side-effects:

```
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

```
list.parallelStream()
    .filter(x -> isPrime(x))
    .forEach(x -> result.add(x));
```

- Use `.collect` instead

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

- Can also consider using `forEachOrdered()`, or a thread-safe list `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)`

Accumulator and Combiner

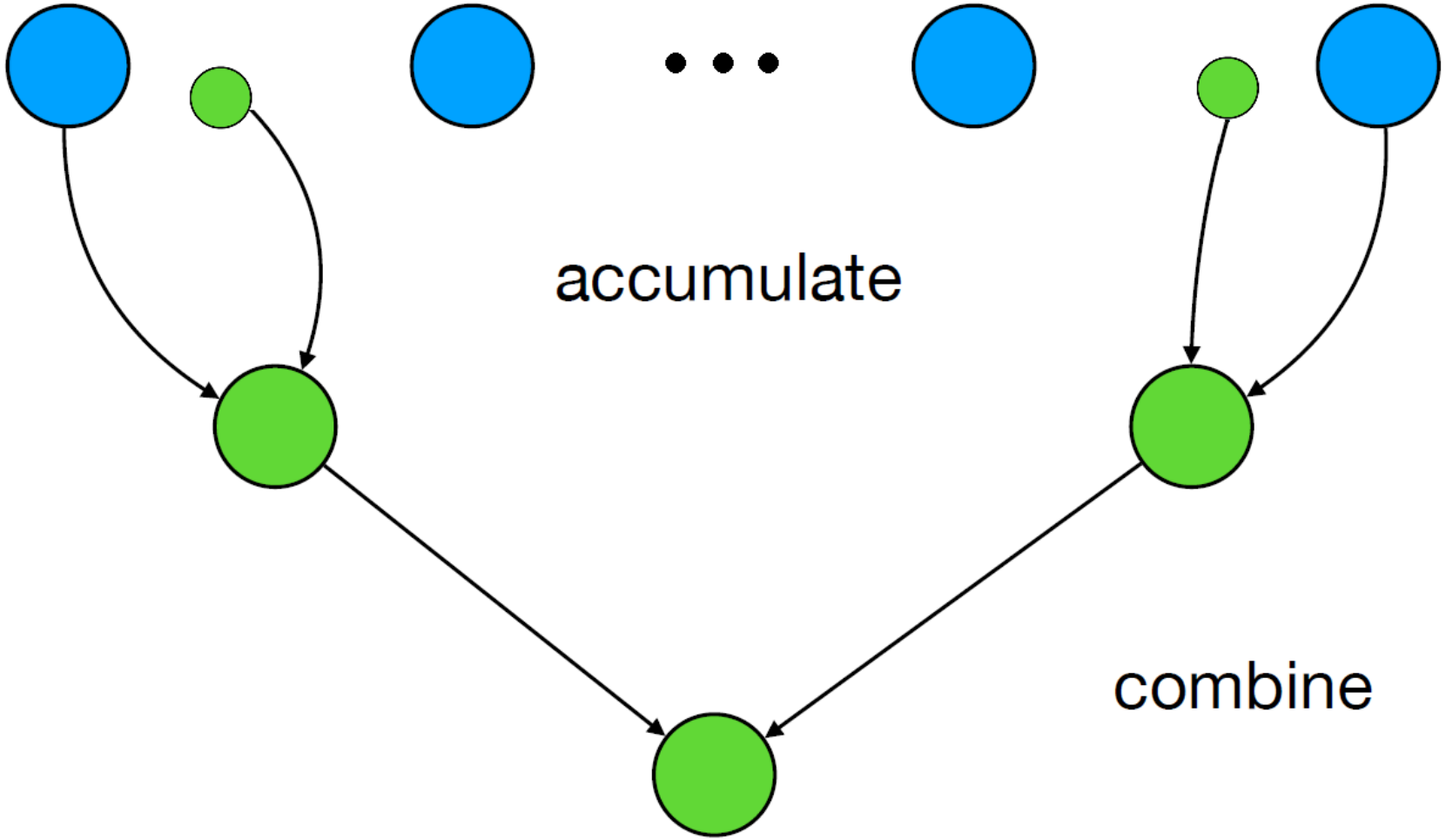
- The following example complies with the above rules:

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

- To see the effects of accumulator and combiner:

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

- Output from a sample run:

```
filter:  5 ForkJoinPool.commonPool-worker-1
filter:  4 ForkJoinPool.commonPool-worker-3
filter:  1 ForkJoinPool.commonPool-worker-3
filter:  3 main
filter:  2 ForkJoinPool.commonPool-worker-2
accumulate: 0 + 5 ForkJoinPool.commonPool-worker-1 // (A)
accumulate: 0 + 1 ForkJoinPool.commonPool-worker-3
combine: 1 + 0 ForkJoinPool.commonPool-worker-3    // (C)
accumulate: 0 + 3 main
combine: 0 + 5 ForkJoinPool.commonPool-worker-1
combine: 3 + 5 ForkJoinPool.commonPool-worker-1    // (B)
combine: 1 + 8 ForkJoinPool.commonPool-worker-1
9
```

(A) Accumulation with identity

(B) Combining results from accumulation

(C) Combining with identity

Accumulator and Combiner

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

```
double result = Stream
    .of(1, 2, 3, 4)
    .parallel()
    .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);
```

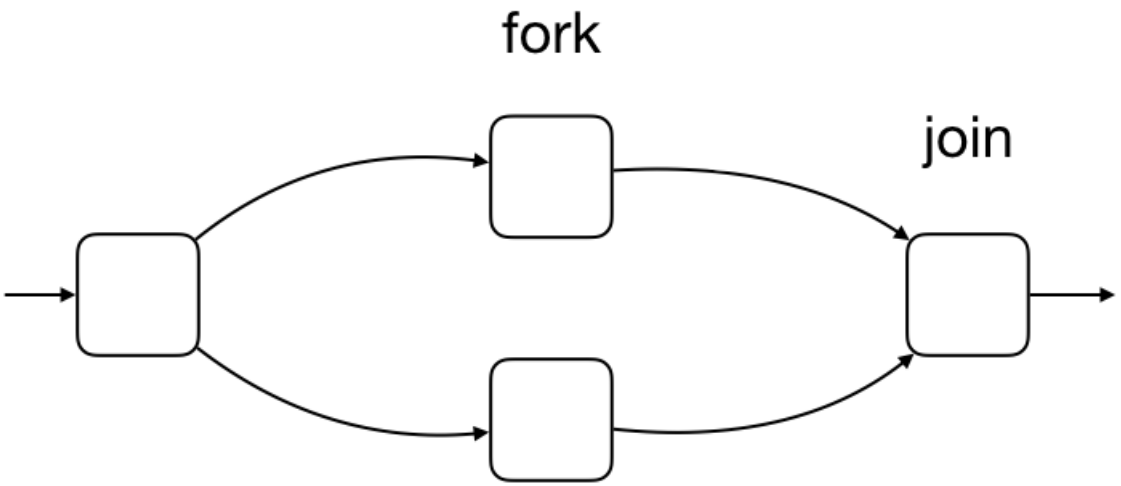
Fork and Join in Parallel Streams

- Should we exploit parallelism to the fullest?

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

- 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 outweighs 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

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

Lecture Summary

- Appreciate the declarative style of programming using streams
- Appreciate lazy evaluation in streams and how it supports streams of infinite elements
- Familiarity with the use of sequential and parallel streams
- Able to compare performances between sequential and parallel streams by timing the execution
- Able to debug parallel streams
- Adherence to rules for parallelizing streams
- Appreciate fork and join in parallel streams
- Appreciate fork/join overhead