CS2030 Lecture 10

Parallel and Concurrent Programming

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

Semester 1 2018 / 2019

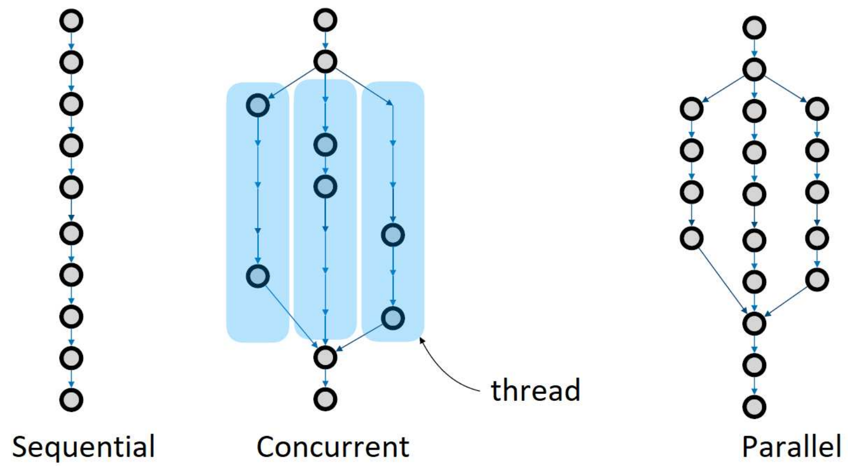
Lecture Outline

- Concurrency versus parallelism
- Parallel streams
- Fork/join framework
- Thread pools

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

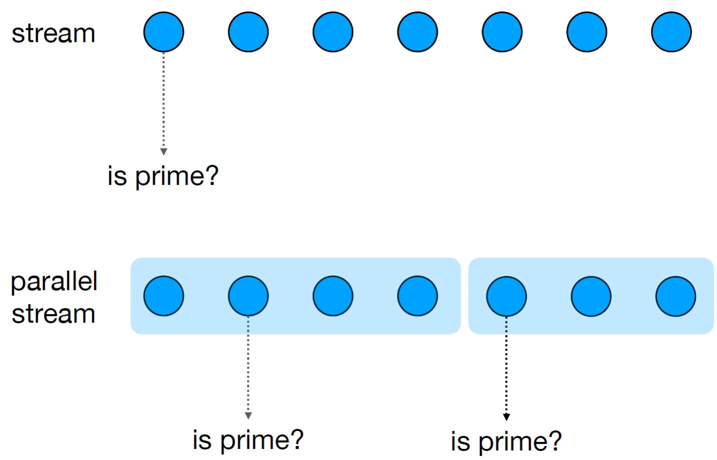
- Streams can be executed in parallel to increase runtime performance
- Parallel streams use a common `ForkJoinPool` via the static `ForkJoinPool.commonPool()` method

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

- 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:



Parallel Streams

```
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(9, (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, preferably stateless and have no side effects
- Example:

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

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

- Inteference is not allowed in both sequential and parallel streams

Correctness of Parallel Streams

- Another example:

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

- Use `.collect` instead

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

- Side effects are a problem in parallel streams
- Use 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:

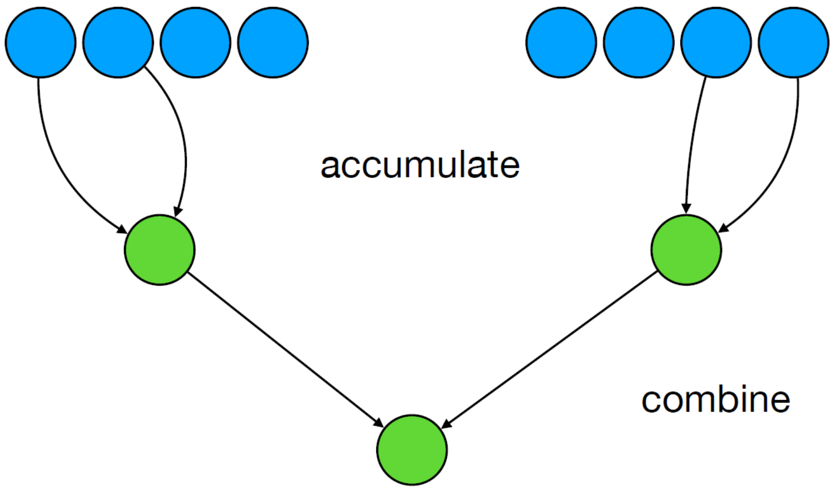
```
Stream.of(1,2,3,4)
    .parallel()
    .reduce(1, (x,y) -> x * y, (x,y) -> x * y)
```


Accumulator and Combiner

- Accumulator and combiner functions are executed in parallel

```
int result = Stream.of(1, 2, 3, 4)
    .parallel()
    .reduce(
            1,
            (x,y) -> {
                System.out.println("accumulator: " +
                    x + " * " + y);
                return x * y;
            },
            (x,y) -> {
                    System.out.println("combiner: " +
                        x + " * " + y);
                    return x * y;
            });
System.out.println(result);
```


Accumulator and Combiner



Accumulator and Combiner

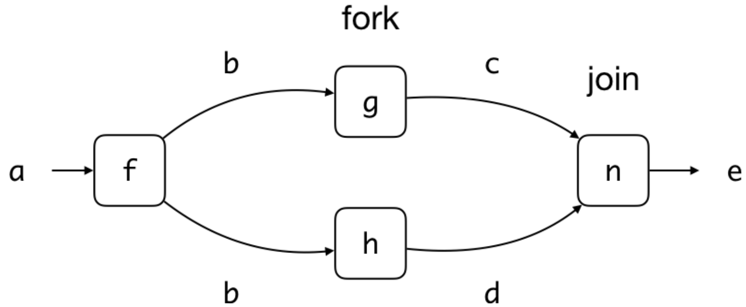
- Errneous examples:

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

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

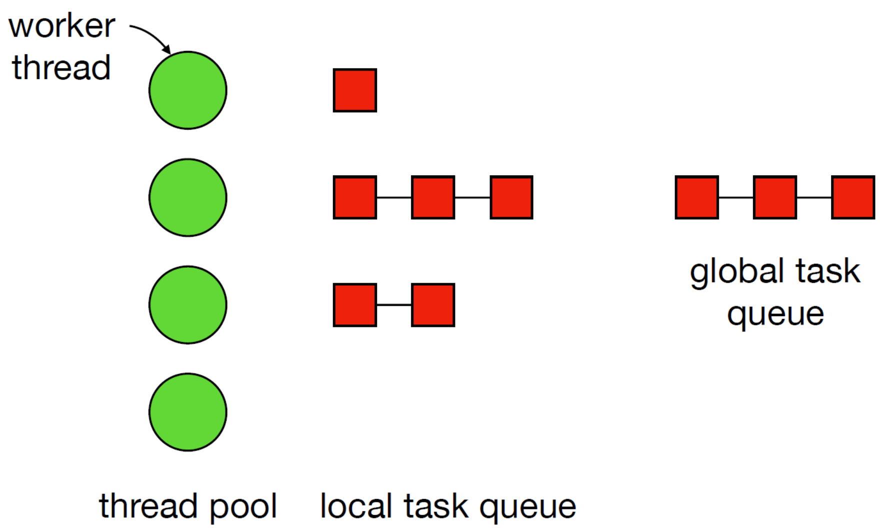
- Given the following program fragment and *computation* graph

```
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

Thread Pools



Thread Pools

- Java maintains a pool of *worker threads*
  - Each thread is an abstraction of a running task
  - Task submitted to the pool for execution, and joins a queue (global queue or worker queue)
  - Worker thread picks a task from the queue to execute
- `ForkJoinPool` is the class that implements the thread pool for `RecursiveTask` (a sub-class of `ForkJoinTask`)
- To submit a task to the thread pool:

```
int sum = ForkJoinPool.commonPool().invoke(task);
```

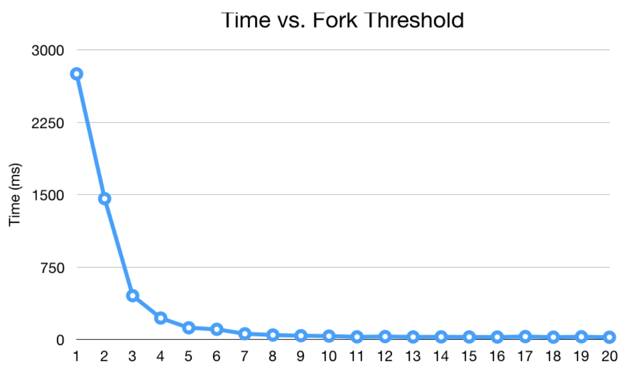
- `invoke(task)` versus `task.compute()`
  - `task.compute()` invokes task immediately; may result in stack overflow if too many recursive tasks
  - `invoke(task)` gets the task to join the queue, waiting to be carried out by a worker (recommended)

Fork/Join Framework

```java
import java.util.concurrent.RecursiveTask;

class SumLeftRight extends RecursiveTask<Integer> {
    int low;
    int high;
    int[] array;

    SumLeftRight(int low, int high, int[] array) {
        this.low = low;
        this.high = high;
        this.array = array;
    }

    @Override
    protected Integer compute() {
        if (high - low < 2) {
            int sum = 0;
            for (int i = low; i < high; i++) {
                sum += array[i];
            }
            return sum;
        } else {
            int middle = (low + high) / 2;
            SumLeftRight left = new SumLeftRight(low, middle, array);
            SumLeftRight right = new SumLeftRight(middle, high, array);
            left.fork();
            return right.compute() + left.join();
        }
    }
}
```


Overhead of Fork/Join

- Forking and joining creates additional overhead
  - wrap the computation in an object
  - submit object to a queue of tasks
  - workers go though the queue to execute tasks



Fork/Join in Parallel Streams

- `parallel()` runs `fork` to create sub-tasks running the same chain of operations on sub-streams
- `combiner` in `reduce` runs `join` to combine the results
- Parallelizing a trivial task actually creates more work in terms of parallelizing overhead

```java
IntStream.range(2, (int) Math.sqrt(n) + 1)
    .parallel()
    .noneMatch(x -> n % x == 0);
```

- Parallelization is worthwhile if the task is complex enough that the benefit of parallelization outweights the overhead
- In the following example, what happens when we parallelize `isPrime`?

Fork/Join in Parallel Streams

```java
public static boolean isPrime(int n) {
    return IntStream.range(2, (int) Math.sqrt(n) + 1)
        .noneMatch(x -> n % x == 0);
}

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");
}
```

Comparing Sequential and Parallel Streams

- Suppose given the following task unit

```
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

```
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

```
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
```


Lecture Summary

- Familiarity with the use of parallel streams
- Adherence to rules for parallelizing streams
- Appreciate fork and join
  - Thread pools
  - Fork/join overhead