Parallel Streams

Monday, April 2, 2018 10:52 AM

Parallel and Concurrent Programming

Concurrency

- OS switching between different processes -> looks like multiple processes running
- Divide computation subtasks into threads (multi-thread)
    - Programmers write threads separately
    - Processor better utilised

Parallelism

- Processor capable of multiple instructions/ multiple cores
- Parallel -> Concurrent

Parallel Stream

- Stream allows for parallel operations on elements of stream in one single line of code

`.parallel()`

- Breaks stream into subsequences running in parallel
- Ordering is lost because of a lack of coordination between parallel tasks (e.g. printing numbers)
- `forEachOrdered`
    - Maintains order, but trade-offs
- Lazy operation: can be placed anywhere in the chain
- Can also call from parallelStream() instead of stream() from Collector class

`.sequential()`

- Opposite of parallel

Embarrassingly Parallel

- Stateless, no side effects
- Each element processed independently of other elements
- Only communication = combine results (e.g. in count())

Stream operations should not interfere with stream data and are usually stateless, with minimum side effects

Interference

- Stream operation modifies source of stream during execution of terminal operation
- `ConcurrentModificationException`
- Non-interference applies to stream() and parallelStream()

Stateless

- Result does not depend on any state that might change during execution of stream
- Have to ensure that state updates are visible to all parallel subtasks

```
Stream.generate(this.events::poll)
    .takeWhile(event -> event != null)
    .filter(event -> event.happensBefore(sim.expireTime()))
    .peek(event -> event.log())
    .map(event -> sim.handle(event))
    .forEach(eventStream -> this.schedule(eventStream));
```

- generate and map depend on the states of the queue and shop

Side Effects

- ArrayList = non-thread-safe data structure
  - Multiple threads manipulate it at the same time -> incorrect result
- `.collect()`
- Use thread-safe data structures
  - `java.util.concurrent`
    - `CopyOnWriteArrayList`

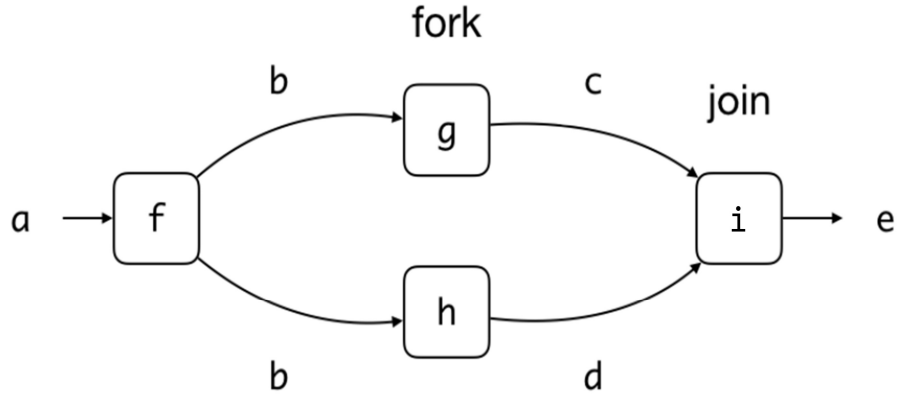
Associativity

- reduce
  - Inherently parallelisable
- Properties:
  - `combiner.apply(identity, i) == i`
  - combiner and accumulator are associative and compatible
    - `combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t)`
    - `u * (1 * t) == u * t`

Performance

- Parallelising does not always improve performance
  - E.g. isPrime()

Fork and Join



- g() and h() can be forked and then joined
- Join operations causes wait for g() to complete for i(c, d)
- Parallel programming: fork/ join framework recursively

```
abstract class ForkJoinTask<V>
  - fork(), join()

abstract class RecursiveTask<V>
  - extends ForkJoinTask<V>
```

- V compute()

```
left.fork();
return right.compute() + left.join();
```

`abstract class RecursiveAction<V>`

- Does not return a value

Thread Pools

- Pool of worker threads (abstraction for running task)
- Global queue for newly submitted task, queue for each worker
    - Task spawn will join queue of the same worker (deque behind each worker)
    - New tasks put at the front of the queue since bigger tasks tend to depend on smaller tasks
- Worker thread picks task to execute
- Idle threads -> work stealing from end of queue, the bigger tasks are split
- Blocked workers -> compensation threads (limit to this, cannot have too many blocked workers)

`ForkJoinPool`

- Implements a thread pool for ForkJoinTask

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

- invoke(task) vs. task.compute()
    - task.compute() is invoked immediately and directly
    - invoke(task) gets task to join a queue
    - Too many recursive tasks: task.compute() will result in a stack overflow
        - Too many compensation threads

Overhead of Fork/ Join

- At small fork thresholds, not worth to parallelise
- isPrime() - task is trivial, more efficient to work sequentially

Ordered vs. Unordered Source

- Streams can define an encounter order
    - Ordered = iterate, ordered collections, of
    - Unordered = generate, set
- Preserved ordering = stable
- unordered()
    - Order not important, parallel operations more efficient

Exercises

2. Parallel streams are unordered and not executed sequentially
    - The reduce function is not associative