Introduction

Version 2.0.5 introduced the `ParallelFlowable` API that allows parallel execution of a few select operators such as `map`, `filter`, `concatMap`, `flatMap`, `collect`, `reduce` and so on. Note that is a **parallel mode** for `Flowable` (a sub-domain specific language) instead of a new reactive base type.

Consequently, several typical operators such as `take`, `skip` and many others are not available and there is no `ParallelObservable` because **backpressure** is essential in not flooding the internal queues of the parallel operators as by expectation, we want to go parallel because the processing of the data is slow on one thread.

The easiest way of entering the parallel world is by using `Flowable.parallel`:

```
ParallelFlowable<Integer> source = Flowable.range(1, 1000).parallel();
```

By default, the parallelism level is set to the number of available CPUs (`Runtime.getRuntime().availableProcessors()`) and the prefetch amount from the sequential source is set to `Flowable.bufferSize()` (128). Both can be specified via overloads of `parallel()`.

`ParallelFlowable` follows the same principles of parametric asynchrony as `Flowable` does, therefore, `parallel()` on itself doesn't introduce the asynchronous consumption of the sequential source but only prepares the parallel flow; the asynchrony is defined via the `runOn(Scheduler)` operator.

```
ParallelFlowable<Integer> psource = source.runOn(Schedulers.io());
```

The parallelism level (`ParallelFlowable.parallelism()`) doesn't have to match the parallelism level of the `Scheduler`. The `runOn` operator will use as many `Scheduler.Worker` instances as defined by the parallelized source. This allows `ParallelFlowable` to work for CPU intensive tasks via `Schedulers.computation()`, blocking/IO bound tasks through `Schedulers.io()` and unit testing via `TestScheduler`. You can specify the prefetch amount on `runOn` as well.

Once the necessary parallel operations have been applied, you can return to the sequential `Flowable` via the `ParallelFlowable.sequential()` operator.

```
Flowable<Integer> result = psource.filter(v -> v % 3 == 0).map(v -> v * v).sequential();
```

Note that `sequential` doesn't guarantee any ordering between values flowing through the parallel operators.

Parallel operators

TBD