Outline

Parallel Data Processing

Summary and References
Outline

Parallel Data Processing
  Parallel Streams
    Fork/join Framework
    Spliterator

Summary and References
Parallel Data Processing  Parallel Streams

Thread Logic

Parallelizing a processing code requires to:

1. split the data structure into chunks;
2. assign each chunk to a different thread;
3. synchronize the executions and combine the results.

Parallel streams hide this logic but each step can be customized.
parallel and sequential

A stream is either:
- sequential (by default or with the method `sequential`)
- or parallel (with the method `parallel`).

The last call in the stream construction wins.

```java
LongStream.rangeClosed(1, N)
    .parallel()
    .sum();
```
Setting the Number of Threads

In the common pool, there is by default one less threads as there are processors (to let room for the main thread).
With the command line:

```
java -Djava.util.concurrent.ForkJoinPool.common.parallelism=3
```

In the code:

```java
System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "3");
ForkJoinPool forkJoinPool = new ForkJoinPool(3);
forkJoinPool.submit(() -> streamterminalOp()).get();
```
Pitfalls

Necessary to understand the semantics to predict how the stream will be parallelized.

Performance issue:

```java
Stream.iterate(1L, i -> i + 1)
   .limit(n)
   .parallel()
   .reduce(0L, Long::sum);
```

Correctness issue:

```java
Accumulator accumulator = new Accumulator();
LongStream.rangeClosed(1, n)
   .parallel()
   .forEach(accumulator::add);
```
Measuring Performance

- Critical to test performance (even if code is well understood).
- Speed-up: ratio of sequential execution time over parallel execution time. With $n$ cores, the ideal speed-up is $n$.
- Micro-benchmarking is difficult:
  - Cold-start (warm-up).
  - Variability (multiple iterations).
  - Java Microbenchmark Harness (JMH).
General Advices

▶ Avoid boxing (prefer primitive stream specialization).
▶ `findFirst` and `limit` rely on the order and require synchronization. `findAny` and `unordered` may solve this issue.
▶ Avoid parallelizing small streams with few data or processing.
▶ Prefer underlying data source that can be split efficiently (e.g. `ArrayList`, `range/rangeClosed`, and streams that convey their sizes).
▶ Be cautious of costly `combiner` method.
▶ Measure and test.
Outline

Parallel Data Processing
  Parallel Streams
  Fork/join Framework
  Spliterator

Summary and References
ForkJoinPool

- Parallel streams rely on the class ForkJoinPool, which is an implementation of ExecutorService.
- Split tasks and distribute them on a pool of threads.
Fork recursively a task in smaller subtask until each subtask is small enough.

Sequential evaluation
Sequential evaluation
Sequential evaluation
Sequential evaluation

Evaluate all subtasks in parallel

Recombine the partial results

join
join
join

fork
fork
fork
Work-Stealing

- Creation of more tasks than available threads because tasks may be heterogeneous.
- Each thread starts by splitting tasks, putting them in their execution queue.
- Once a task is small enough, it is executed.
- When a thread finishes all tasks in its execution queue, it steals waiting tasks from the other queues.
Outline

Parallel Data Processing
  Parallel Streams
  Fork/join Framework
  Spliterator

Summary and References
Parallel Data Processing  Spliterator

## Interface

Splitable iterator: define how to split the stream that is iterated.

```java
public interface Spliterator<T> {
    boolean tryAdvance(Consumer<T> action);
    Spliterator<T> trySplit();
    long estimateSize();
    int characteristics();
}
```
trySplit

Logic for splitting data stream.

```java
public Spliterator trySplit() {
    if (currentSize < MIN_SIZE)
        return null;
    long half = currentSize / 2;
    currentSize -= half;
    return new Spliterator(half);
}
```
tryAdvance

Feeds the consumer with the next stream element.

```java
public boolean tryAdvance(Consumer action) {
    if (currentSize == 0)
        return false;
    action.accept(nextValue());
    currentSize--;
    return true;
}
```
estimateSize

The size can be estimated (used for balancing work).

```java
public long estimateSize() {
    return currentSize;
}
```
Characteristics

Information to control or optimize the usage of the spliterator.

```java
public int characteristics() {
    return ORDERED + DISTINCT + SORTED + CONCURRENT
    + Sized + SUBSIZED + NONNULL + IMMUTABLE;
}
```

- **ORDERED** the order of the elements must be enforced
- **DISTINCT** all elements are distinct
- **SORTED** elements are sorted
- **CONCURRENT** the modification of the stream source is thread-safe
- **SIZE** the estimated size is precise
- **SUBSIZED** all created spliterators are SIZED
- **NON-NULL** all elements are non-null
- **IMMUTABLE** the stream source cannot be modified
Demonstration

Compute the sum of all square numbers between 1 and $n$.

```java
IntStream.rangeClosed(1, n)
    .filter(i -> {
        var x = Math.floor(Math.sqrt(i));
        return x * x == i;
    })
    .sum()
```
Outline

Parallel Data Processing

Summary and References
Parallel Data Processing

- Internal iteration allows you to process a stream in parallel without the need to explicitly use and coordinate different threads in your code.
- Behavior and performance of parallel code can be counterintuitive, it is always necessary to check and measure them.
- Parallel execution can improve performance, especially when the number of elements is large or the processing is time consuming.
- From a performance point of view, using the right data structure (e.g. primitive streams) is more important than trying parallelization.
- The fork/join framework is used to recursively split streams and distribute them with a work-stealing approach.
- A Spliterator defines how a parallel stream can split the data it traverses.
Official Documentation

- Documentation of interface Spliterator