Theory: Parallel streams

20 minutes 0 / 4 problems solved

One of the greatest features of the Stream API and the functional programming in general is the ability to easily write clear code for parallel data processing. There is no need to create threads manually, check whether the code is well-synchronized and invoke `wait` / `notify` methods. All these things are performed inside the parallel streams automatically.

> Under the hood, parallel streams use `ForkJoinPool` introduced in Java 7 to manage parallel executions.

In this topic, you will learn how to create and use parallel streams, as well as learn about the performance and some caveats associated with them.

§1. Creating parallel streams

There are several simple ways to create parallel streams:

- to invoke the `parallelStream()` method of a collection instead of `stream()`:

```
List<String> languages = List.of("java", "scala", "kotlin", "C#");

List<String> jvmLanguages = languages.parallelStream()
        .filter(lang -> !Objects.equals(lang, "C#"))
        .collect(Collectors.toList());

System.out.print(jvmLanguages); // [java, scala, kotlin]
```

- to transform an existing stream into a parallel stream using the `parallel()` method:

```
long sum = LongStream
        .rangeClosed(1, 1_000_000)
        .parallel()
        .sum();

System.out.println(sum); // 500000500000
```

> The example shows not the best way to calculate the sum of this sequence, but this is just an example of parallel streams. It is better to use the formula for the sum of n terms in the arithmetic progression.

As you can see, despite the fact that we use parallel streams, all the code for working with them remains the same as before.

There are two additional methods for working with parallel streams:

- `isParallel()` returns `true` if the stream is parallel and `false` otherwise;
- `sequential()` returns an equivalent sequential stream.

> It is important that any stream is either **parallel** or **sequential**. A mixed mode is impossible. If a stream pipeline calls both `parallel()` and `sequential()` methods, the last call wins.

§2. Performance of parallel streams

It's really easy to make a stream parallel. But should we always do this? Not really.

> A **parallel stream** is not always faster than the equivalent sequential stream.

Current topic:

Parallel streams

Topic depends on:

× Stream pipelines

Table of contents:

There are a number of factors which significantly affect the performance of a parallel stream.

- **Size of data.** The bigger size of data → the greater speedup.
- **Boxing/Unboxing.** Primitive values are processed faster than boxed values.
- **The number of cores are available at runtime.** The more available cores → the greater speedup.
- **Cost per element processing.** The longer each element is processed → the more efficient parallelization. But it is not recommended to use parallel stream for performing too long operations (for example, network connections). So it's a kind of a trade off.
- **The type of data source.** Usually initial data source is a collection. The easier it's split into parts → the greater speedup. For example, regular arrays, `ArrayList`, and `IntStream.range` are good sources for data splitting since they support random access. Others, such as `LinkedList`, `Stream.iterate` are bad sources for data splitting.
- **Type of operations: stateless or stateful**. Stateless operations (for example, `filter` and `map`) are better for parallel processing than stateful operations (for example, `distinct`, `sorted`, `limit`). Operations that are based on the order of elements are especially hard for parallelizing. But it's not always possible to avoid them.

Of course, the listed factors give us only an approximate estimate, i.e. some reference points. When developing a real application, you need to perform some measurements to decide whether to use parallel streams. Another important question is "*Are you ready to spend additional server's threads on the streams?*".

§3. Some caveats with parallel streams

In addition to the fact that parallel streams do not always increase the performance (and sometimes, decrease), there are some differences in the behavior of parallel and sequential streams which you need to understand before using them.

If you know additional caveats with parallel streams, write them into comments, please.

§4. The reduce method and its initial value

Let's assume, you'd like to calculate the sum of numbers and add 100 to the result. When using a sequential stream, you can just set 100 as the initial value (seed) of the `reduce()` method:

```
1 int result = numbers.stream().reduce(100, Integer::sum);
```

This code produces the same result as following:

```
1 int result = numbers.stream().reduce(0, Integer::sum) + 100;
```

But when using a parallel stream, the first code will produce a *strange result*. The reason is your dataset will be split into some parts and the value 100 will be added to each of them.

> When using a parallel stream, use only a neutral element (0 for summing, 1 for multiplication, and so on) as the initial value in the `reduce` method. It is better to do the same with sequential streams.

§5. The forEach method and the order of elements

Given a sorted list of numbers `1, 2, ..., 10`. We'd like to process and print each number from the list using streams.

Here is a sequential stream:

```
sortedNumbers.stream()
        .map(Function.identity()) // some processing
        .forEach(n -> System.out.print(n + " "));
```

The output is:

```
1 2 3 4 5 6 7 8 9 10
```

Here is a parallel stream:

```
sortedNumbers.parallelStream()
        .map(Function.identity()) // some processing
        .forEach(n -> System.out.print(n + " "));
```

The output:

```
6 7 9 10 8 3 4 1 5 2
```

Oops..! The `forEach` method breaks the order when used with parallel streams.

If we rewrite this using the `forEachOrdered` method, the code will work as we expected:

```
sortedNumbers.parallelStream()
        .map(Function.identity()) // some processing
        .forEachOrdered(n -> System.out.print(n + " "));
```

The output:

```
1 2 3 4 5 6 7 8 9 10
```

That's OK.

> When using a parallel stream, use `forEachOrdered` rather than `forEach` if the order of elements matters to you. But, at the same time, this will reduce the speedup from the parallelization.

§6. Empty lists and the order of elements

You may have thought that there are no more problems with the order of elements. But this is not true: the order and parallelization are not friends.

Let's assume, we'd like to get the first three even numbers from a parallel stream of two concatenated streams.

```
// create a filled list of integers
List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

// create an empty list
List<Integer> emptyList = List.of();
```

Here is a concatenation, and processing of two lists.

```
Stream.concat(numbers.stream(), emptyList.stream())
        .parallel()
        .filter(x -> x % 2 == 0)
        .limit(3)
        .forEachOrdered((n) -> System.out.print(n + " "));
```

The output:

```
2 4 6
```

That's OK.

But If we create an empty list using `Collections.emptyList()`, then we will always have a different output.

The output:

```
2 4 10
```

Oops..!

The reason is `Collections.emptyList()` doesn't report about its ordering and the stream cannot use the `forEachOrdered` method correctly.

> Just a general, but an important advice: be careful with the order of elements in parallel streams. You can encounter troubles in the most unpredictable places.

§7. Should I use parallel streams?

Stream API makes it very easy to start using parallel streams. But they are not always faster than equivalent sequential streams since their performance depends on many factors including the volume of data, the hardware, and the used operations. At the same time, it is quite difficult to predict the speedup without performing measurements in the realities. In addition, there are some possible caveats when using a parallel stream especially related to the order of its elements.

Thus, you must be absolutely sure why you need parallel streams in your case. If there are enough resources or the performed operations are simple, it may be better to use sequential streams. But if you've achieved a measurable stable speedup, you can try using parallel streams.

Report a typo

52 users liked this theory. **1** didn't like it. **What about you?**

Start practicing

Comments (4) Hints (0) Useful links (0) Show discussion