Using Optionals ← Parallelizing Streams → That's the end of the series!

Parallelizing Streams

Optimizing Streams Computations

A very exciting feature of the Stream API is the fact that a stream is capable of processing data in parallel. Processing data in parallel with the Stream API is as simple as calling the `parallel()` method on any existing stream.

```
int parallelSum =
    IntStream.range(0, 10)
             .parallel()
             .sum();
```

Running this code gives you the following result.

```
parallelSum = 45
```

This sum has in fact been computed in parallel. You may not notice any performance gain on such a small example though.

Why would you want to compute your data in parallel? Probably to get the result of your computations faster. Will a parallel stream give you a result faster than a sequential stream? Well, the answer to this question is not as easy as it sounds. In some cases yes, but in some other cases, unfortunately, no. As disappointing as it may sound, a parallel stream is not always faster than a sequential stream.

With that in mind, you should be careful: choosing to use parallel stream is not a decision to be taken lightly. There are several questions you need to ask yourself before even considering going parallel.

First, ask yourself, do you need it? Do you have performance requirements in your application that are not met? Are you sure that your performance problems are coming from the stream processing you are thinking of computing in parallel? How do you plan to measure your performance gain to make sure that going parallel for this particular computation improved the performance of your application?

Going parallel consumes more computing power. Do you have spare CPUs or CPU cores to give to this computation? Can you give more CPU cycles to your computation without slowing down the rest of your application?

Going parallel consumes threads. Do you have spare threads to give to your computation? If you are working on an application running in a webserver, then your thread are used to serve HTTP requests. Are you OK with taking them for other things?

Once you have chosen to go parallel, then you need to make sure the performances of your stream computations have indeed improved. You should measure this performance gain in a context that is as close as possible to your production environment.

In this tutorial, we are covering several key elements that will help you evaluate the gains you may expect from going parallel and some other elements that should make you wary about going parallel. But in the end, the only thing that should tell you if going parallel is worth it or not is testing and measuring execution times.

Parallelization Implementation

Parallelization is implemented in the Stream API by using recursive decomposition of the data the stream is processing. It is build on top of the Fork/Join Framework, added in JDK 7.

The decomposition consists of dividing the data your stream is processing in two parts. Each part is then processed by its own CPU core that may decide to divide it again, recursively.

At some point, the framework will decide that the amount of data in a given part is small enough to be processed normally. This subset of data will then be processed, and a partial result will be computed. This partial result will then be merged with the other partial results computed from the other parts on other CPU cores.

Going parallel does come with an overhead. This overhead has to be small compared to the gains of distributing the computations on several CPU cores. If not, going parallel will worsen the performances of your computations instead of improving them.

Let us examine all these steps on by one and see what can prevent you from getting better performance gains.

Understanding Data Locality

Data locality has an impact on how fast your data can be processed, whether it is processed sequentially or in parallel. The better the locality, the faster your computation will be.

To be made available to the CPU, your data must be transferred from the main memory of your computer to the cache of your CPU. Physically speaking, the main memory is a specific component of your computer, separated from your

CPU. On the other hand, the cache share the same silicon die as the core computing elements of your CPU. They are connected together through your motherboard and different communication buses. Transferring data from the main memory to the cache of your CPU is very slow compared to speed at which the core of a CPU can access data from its cache.

When your CPU needs some data, it first checks if this data is available in its cache. If it is, then it can use it right away. If it is not, then this data must be fetched from the main memory and copied to the cache. This situation is called a *cache miss*. Cache misses are costly because during this time your CPU is waiting for your data. You want to avoid this situation.

The way the data is transferred between the main memory and the cache of your CPU plays an important role in avoiding cache misses. The memory is organized in lines. Typically, a line is 64 bytes long, that is, eight `long` values (it can vary from one CPU to another). All the transfers between the main memory and the CPU cache are made line by line. So even if your CPU only needs a single `int` value, the full line that contains this value is transferred to the cache.

Iterating Over an Array of Primitive Types

Suppose that your code is iterating over an array of type `int[]`. A line of 64 bytes can hold 16 `int` values. Suppose that accessing the first element of your array is a cache miss. The CPU will then load the line that contains this element into its cache to start the iteration. Because it loaded a full line, the 15 next values may have also been transferred. Accessing the next values will be very fast.

In that case, *data locality* is excellent: your data is physically stored in contiguous zones of the main memory. It is desirable because transferring your data from the main memory to the CPU cache will be much faster.

Iterating Over an Array of Integer Instances

Suppose now that your code is iterating over an array of type `Integer[]`. What you really have is not an array of primitive types anymore but an array of references. Each cell of this array contains a reference to an object of type `Integer` that can be anywhere in the memory.

If the access to the first element of your array is a cache miss, then the CPU will have to load the line that contains this element to its cache. What it really loads is the first 16 references of your array, assuming that this first reference is at the beginning of the line. It then has to load the first `Integer` object, that may be somewhere else in the main memory, leading to another cache miss. In fact, odds are that each reading of every `Integer` object of your array will also be a cache miss.

In that case, *data locality* is not as good as in the previous example: the references to your data is physically stored in contiguous zones of the main memory, but the values you need to conduct your computations is not. It is not desirable because transferring the values you need from the main memory to the CPU cache is much slower than in the case of an array of primitive types.

Iterating Over a Linked List of Integer Instances

Let us examine one last situation. Suppose now that your code is iterating over a list of type `LinkedList<Integer>`. If the access to the first element is a cache miss, then the CPU will load the first node of your linked list to its cache. That node contains two references: the first to the value you need for your computations, the second one to the next node of the list. This situation is worse than the previous one: odds are that accessing the next value of your list will generate two cache misses.

In this case, *data locality* is terrible: neither your data nor the references to them are stored in contiguous zones of the main memory. Accessing the elements you need will be much slower than the first case we examined.

Avoiding Pointer Chasing

Having to follow references, or pointers, to reach the right element that carries the data your need is called *pointer chasing*. Pointer chasing is something you want to avoid in your application and is the source of many performance hits. Pointer chasing does not exist when iterating over an array of `int` values. It constitutes your main performance hit when iterating over a linked list of `Integer` instances.

Splitting a Source of Data

If you decide to process a stream in parallel, the first step will consist of splitting your source of data. For the splitting to be efficient, it should have several properties.

- Splitting the data structure should be easy and fast.
- The splitting should be even: the two substreams you get should have the same amount of data to process.

Splitting an Instance of Collection

An `ArrayList` is a perfect data structure to split. You can get the middle element easily, and if you split an array by its middle, you know exactly how many elements you will have on both subarrays.

On the other hand, a `LinkedList` is not a good structure to split. Getting to the middle element requires going through half of the elements of the list, one by one, which is costly because of pointer chasing. Once you are there, you can get two sublists with the right number of elements.

A `HashSet` is built on an array of buckets, so splitting this array is the same as splitting the internal array of an array list. But the way the data is stored in this array is different. It is harder to split this array in a ways that guarantees that you will have the same amount of elements in both parts. You may even end up with an empty subpart.

A `TreeSet` is based on a red-black tree implementation. It guarantees that all the nodes have the same amount of elements on their right and left child nodes. So splitting an instance of `TreeSet` in two even subtrees is easy. You still need to chase pointers to reach your data, though.

All these structures are used in the Collection Frameworks, and you can get the number of elements each of them carry.

This is not the case for all the structures from which you can create a stream.

Splitting the Lines of a Text File

This is the case for the `Files.lines(path)` pattern, which was covered earlier in this tutorial. It creates a stream that processed the lines of the text file denoted by this `path` object. It is not possible to get the number of lines of a text file without analyzing it.

The same goes for the `Pattern.splitAsStream(line)` pattern that we also covered. It creates a stream from the splitting of the `line` using the provided pattern. Again, you cannot tell in advance the number of elements you wil process in such a stream.

Splitting a Range or a Generated Stream

The specialized streams of numbers also give you patterns to create streams.

The `IntStream.range(0, 10)` stream is easy to split. In fact, it looks like an array of numbers that you can split by the middle. The amount of elements in each part is predictable, which is desirable.

On the other hand, the `Stream.generate()` and `Stream.iterate()` methods does not give you an easily splittable source of data. In fact, this source may be infinite and only limited by the way it is processed in your stream.

Let us compare the two following patterns.

```java
List<Integer> list1 =
    IntStream.range(0, 10).boxed()
             .toList();

List<Integer> list2 =
    IntStream.iterate(0, i -> i + 1)
             .limit(10).boxed()
             .toList();
```

Both lists `list1` and `list2` are the same, created with different patterns. The first one is easily splittable, while the second one is not. The main reason is that, in the second pattern, knowing the value of the fifth element requires the computation of all the previous elements. In that sense, this second pattern looks like a linked list where you need to visit the first four elements to reach the fifth one.

Splitting and Dispatching Work

Once your source of data has been split, then the two substreams have to be processed on different cores of your CPU for the parallelization to be effective.

This is done by the Fork/Join Framework. The Fork/Join Framework handles a pool of threads, created when your application is launched, called the Common Fork/Join Pool. The number of threads in this pool is aligned with the number of cores your CPU has. Each thread in this pool has a waiting queue, in which the thread can store tasks.

1. The first thread of the pool creates a first task. The execution of this task decides if the computation is small enough to be computed sequentially or is too big and should be split.

2. If it is split, then two subtasks are created and stored in the queue of that thread. The main task then waits for the two sub-tasks to complete. While waiting, it is also stored in this waiting queue.
3. If the computation is conducted, then a result is produced. This result is a partial result of the whole computation. This task then returns the result to the main task that created it.
4. Once a task has the two results of the two subtasks it created, it can merge them to produce a result and return it to the main task that created it.

At a given point, the first main task gets the two partial results from it two sub-tasks. It is then able to merge them and return the final result of the computation.

For now, the only thread that is working is the first thread of the pool, which was invoked by the Fork/Join Framework. The Fork/Join Framework implements another pattern of concurrent programming called *work stealing*. An idle thread of the pool can examine the waiting queue of the other threads of the same pool to take a task and process it.

This is what happens in this case. As soon as the number of tasks grows in the first waiting queue, the other threads are going to steal some of them, process them, split the work further, and populating their own waiting queues with more tasks. This feature keeps all the threads of the pool busy.

This work stealing feature works well but it has a drawback: depending on how your source has been split and how tasks are moved from one thread to another, your data may be processed in any order. This may be a problem in some cases.

Processing a Substream

Processing a sub-stream can be different from processing a full stream. Two elements can make the processing of a sub-stream different: accessing an external state, and carrying a state from the processing of an element to another. These two elements will hit the performance of your parallel streams.

Accessing an External State

The Fork/Join Framework splits your computation into many sub-tasks, each processed by a thread from the pool.

If you process your stream sequentially, all the elements are processed in the thread that runs your method. If you process the same stream in parallel, the elements are processed by a thread from the Common Fork/Join pool.

Accessing a state external to your stream is then made from another thread and may lead to race conditions.

Let us run the following code in a classic `main` method.

```
Set<String> threadNames = 

IntStream.range(0, 100)
        // .parallel()
        .mapToObj(index -> Thread.currentThread().getName())
        .collect(Collectors.toSet());

System.out.println("Thread names:");
threadNames.forEach(System.out::println);
```

The result it produces is the following.

```
Thread names:
main
```

If you uncomment the `parallel()` call then this stream is executed in parallel. The result becomes the following, and may vary on your own machine.

```
Thread names:
ForkJoinPool.commonPool-worker-3
ForkJoinPool.commonPool-worker-4
ForkJoinPool.commonPool-worker-2
ForkJoinPool.commonPool-worker-4
main
ForkJoinPool.commonPool-worker-5
```

Any access to a nonconcurrent, external element may lead to race conditions and data inconsistency. Let us run the following code.

```java
List<Integer> ints = new ArrayList<>();

IntStream.range(0, 1_000_000)
         .parallel()
         .forEach(ints::add);

System.out.println("ints.size() = " + ints.size());
```

Running this code several times may lead to different results because all the threads of the Common Fork/Join Pool are trying to add data concurrently in an instance of `ArrayList`, which is not a thread-safe structure. There is little chance to see the right result, and you can even get an `ArrayIndexOutOfBoundsException`. Running this kind of code with any non-concurrent collection or map leads to unpredictable results, including exceptions.

It is an antipattern for a stream to mutate a state external to this stream.

Encounter Order

There are cases where the order in which the data is processed is significant in the Stream API. This is the case for the following methods.

- `limit(n)`: limits the processing to the `n` *first* elements of this stream.
- `skip(n)`: skips the processing for the `n` *first* elements of this stream.

- `findFirst()`: find the *first* element of the stream.

These three methods need to remember in what order the elements of the stream are processed and need to count the elements to produce a correct result.

There are called *stateful* operations, because they need to carry an internal state to work.

Having such stateful operations leads to overheads in parallel streams. For instance, `limit()` needs an internal counter to work correctly. In parallel, this internal counter is shared among different threads. Sharing a mutable state between threads is costly and should be avoided.

Understanding the Overhead of Computing a Stream in Parallel

Computing a stream in parallel adds some computations to handle parallelism. These elements have a cost, and you need to know them to make sure that this cost will not be too high compared to the benefits of going parallel.

- Your data needs to be split. Splitting can be cheap, and it can be expensive, depending on the data you process. Bad locality for your data will make the splitting expensive.
- Splitting needs to be efficient. It needs to create evenly split sub-streams. Some sources can be evenly split easily, some others can not.
- Once split, the implementation processes your data concurrently. You should avoid any access to any external mutable state, and also avoid having an internal shared mutable state.
- Then the partial results have to be merged. There are results that can be easily merged. Merging a sum of integers is easy and cheap. Merging collections is also easy. Merging hashmaps is more costly.

Stating Some Rules to Use Parallel Streams Properly

Rule #1 Do not optimize because it's fun; optimize because you have requirements and you do not meet them.

Rule #2 Choose your source of data with caution.

Rule #3 Do not modify external state, and do not share mutable state.

Rule #4 Do not guess; measure the performance of your code.

Last update: *September 14, 2021*

Using Optionals ← **Parallelizing Streams** → That's the end of the series!

Home > Tutorials > The Stream API > Parallelizing Streams