Analyzing the Fork / Join Implementation of Parallel Streams

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR
@JosePaumard https://github.com/JosePaumard

Agenda

How does the Fork / Join framework work

How parallelism is implemented

Fork and Join steps

See how synchronization can affect performance

Forking and Joining Tasks

Parallel Streams are built on the Fork / Join framework:

- a task is split in 2 sub tasks

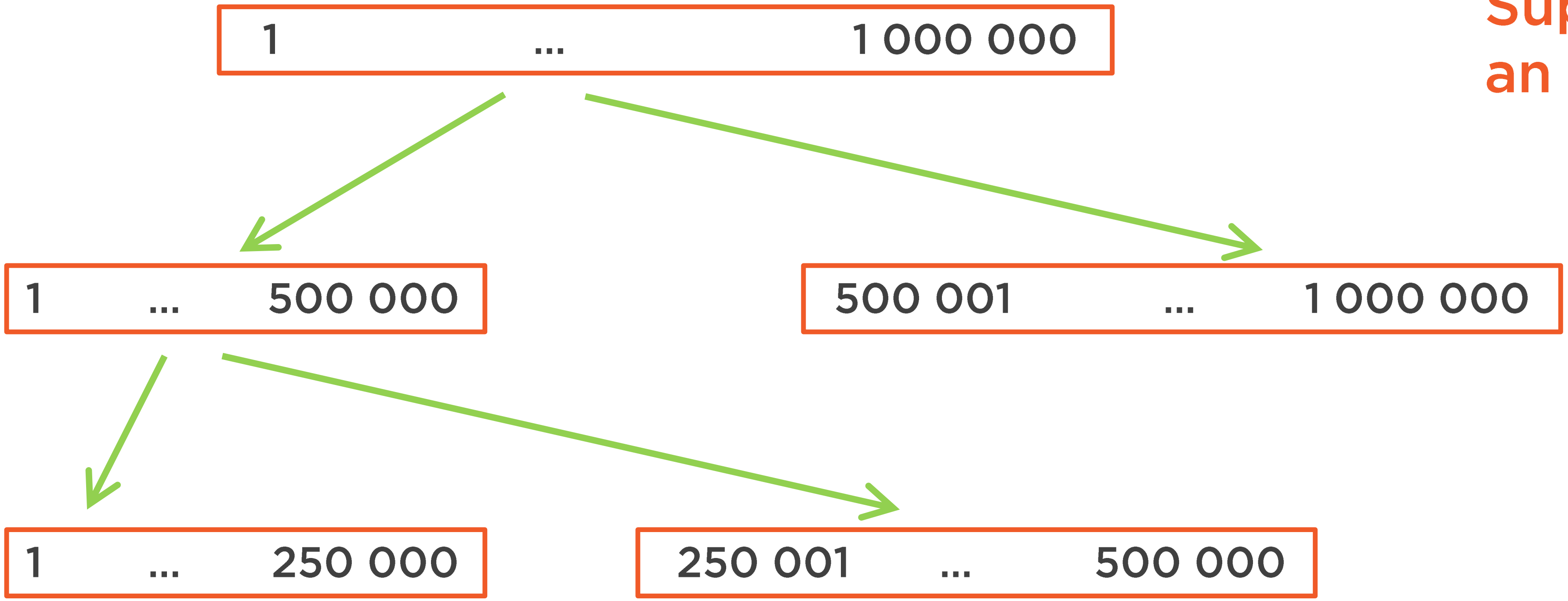
- sub tasks are sent to a pool of thread: "Fork / Join Pool"

- the results of each sub tasks are joined

- and the global result is computed

Forking Tasks

Suppose we have an array of 1M int



The splitting continues until each task is “small enough”

This is the Fork step

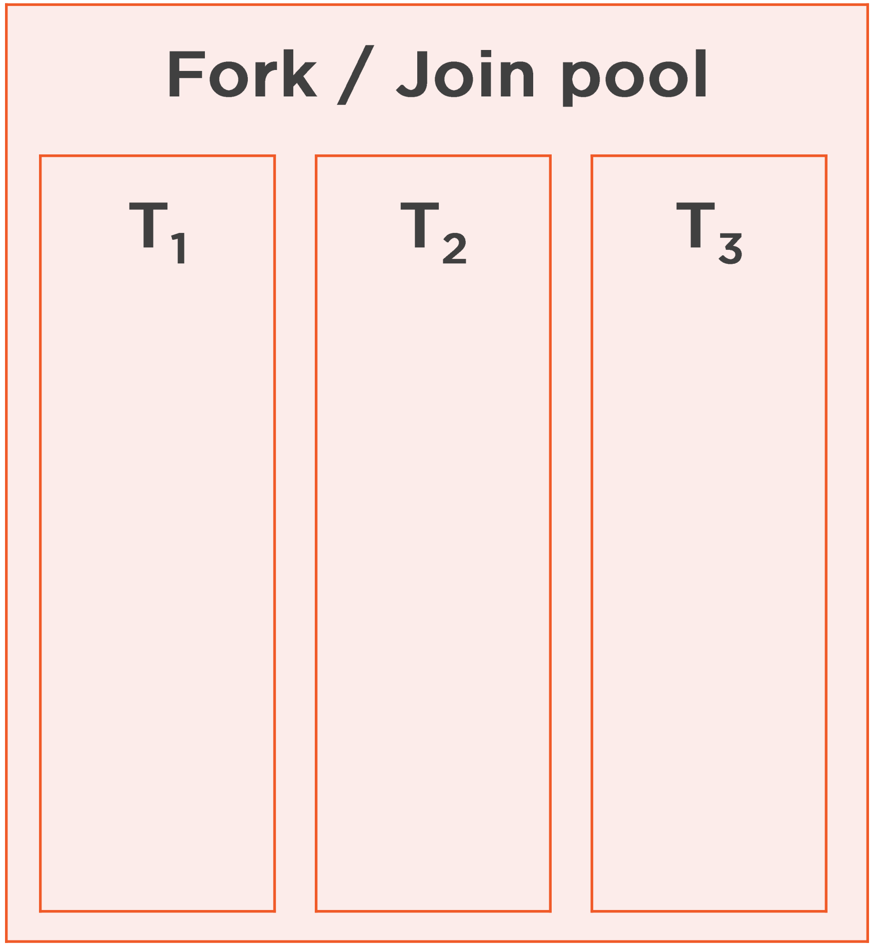
The Common Fork / Join pool:

- is a pool of threads
- created when the JVM is created
- that is used for all the parallel Stream
- the size is fixed by the number of virtual cores

```
java.util.concurrent.ForkJoinPool.common.parallelism
```

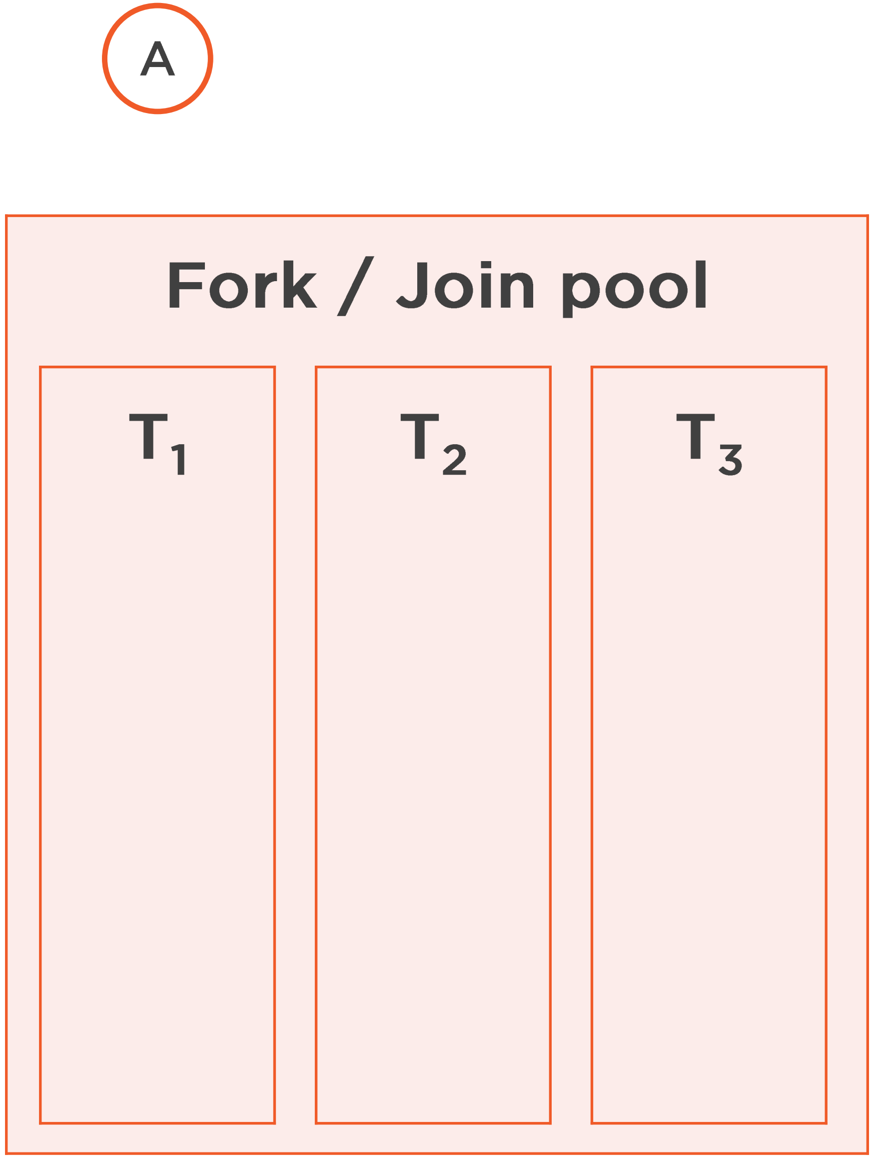
We have a first task

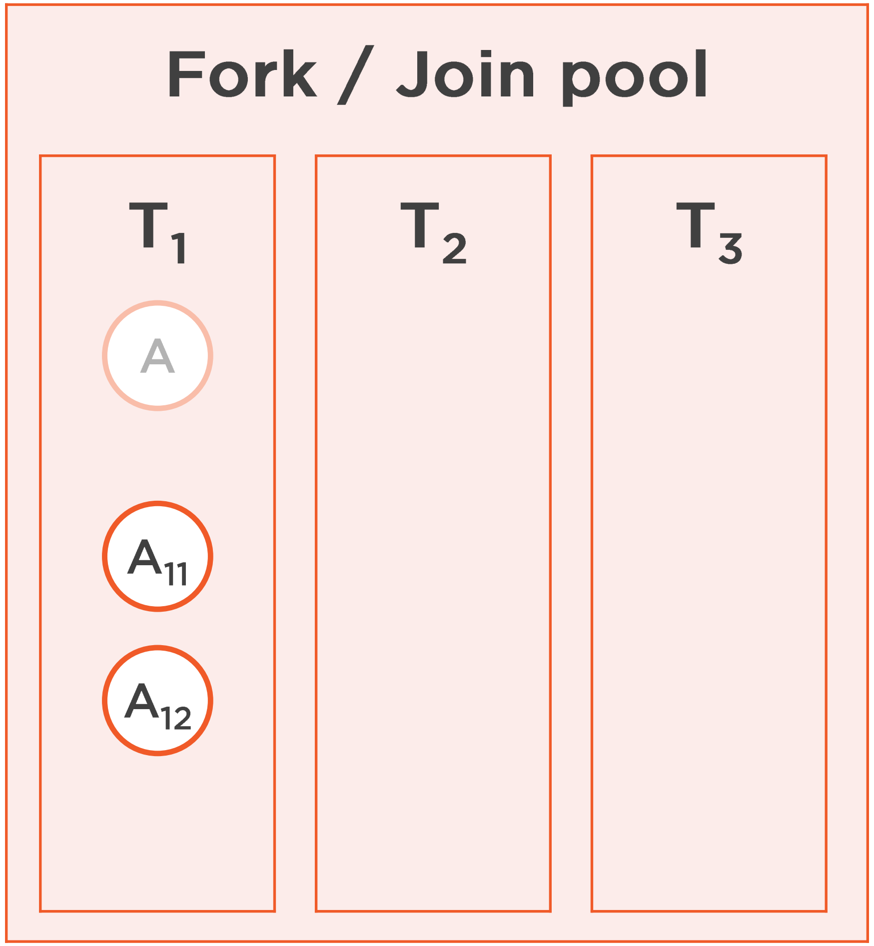
And a pool of threads



We have a first task

And a pool of threads

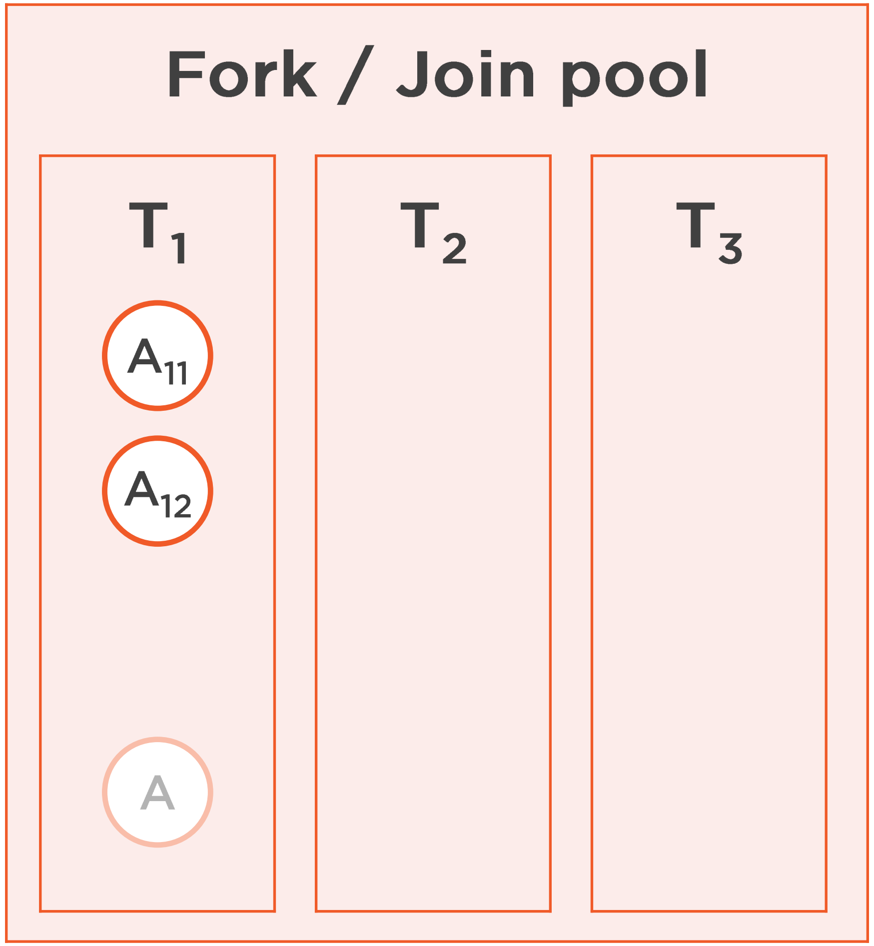




We have a first task

And a pool of threads

A is forked

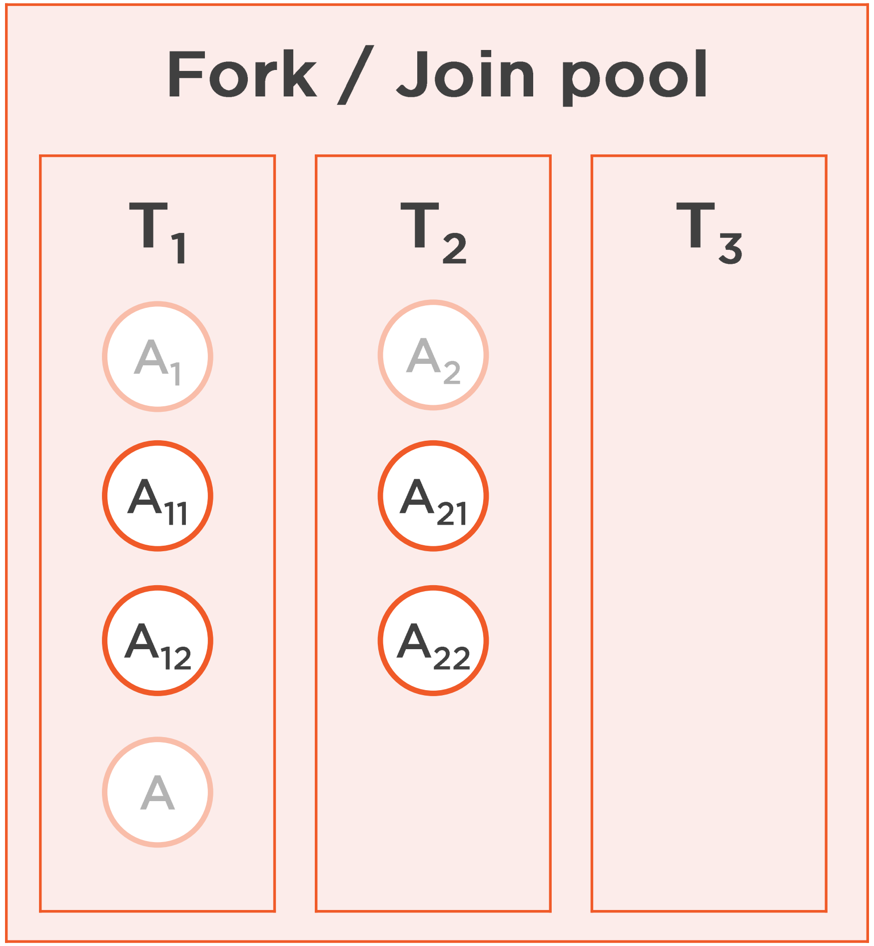


We have a first task

And a pool of threads

A is forked

T_2 can steal tasks



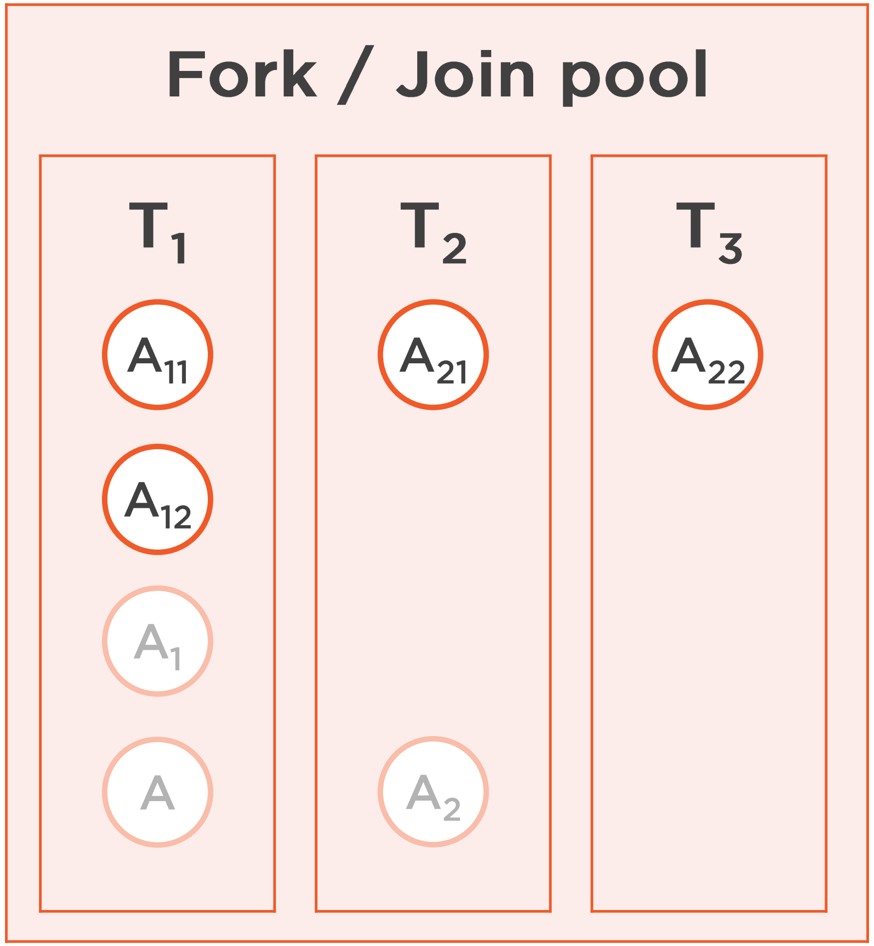
We have a first task

And a pool of threads

A is fork

T_2 can steal tasks

More forks



We have a first task

And a pool of threads

A is fork

T_2 can steal tasks

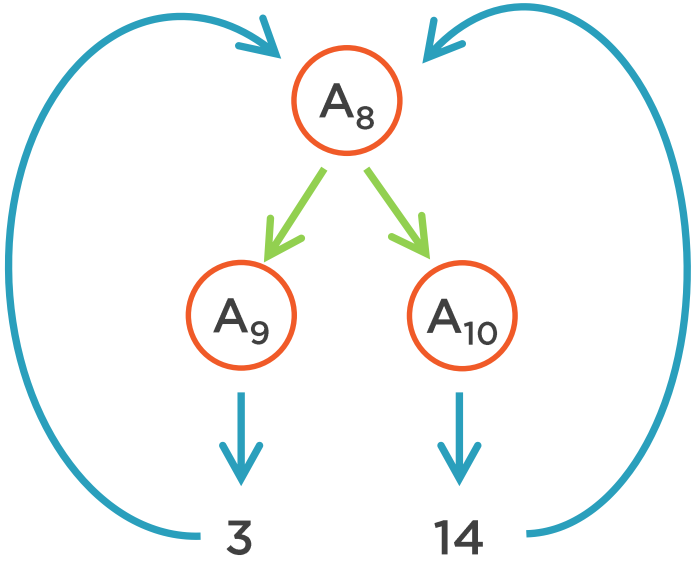
More forks

More work stealing

There are two kind of tasks:

- tasks that have been split
- terminal tasks

Joining Tasks

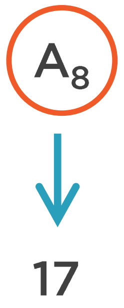


Suppose we compute a sum

A_8 has been split

A_9 produced the result 3
A_{10} produced the result 14

The results are sent to A_8



Suppose we compute a sum

A_8 has been split

A_9 produced the result 3
A_{10} produced the result 14

The results are sent to A_8

That can produce a result: 17

This is the Join step

The Fork and Join steps happen in parallel

As soon as tasks are generated, the computations begin

- The tasks are stored in waiting queues
- Each thread has its own waiting queue
- A non-active thread can steal tasks from another queue
- This is “work stealing”
- Each thread is kept busy

Forking and Joining is an overhead…

Are there cases where this overhead is greater than the gain given by parallelism?

Demo

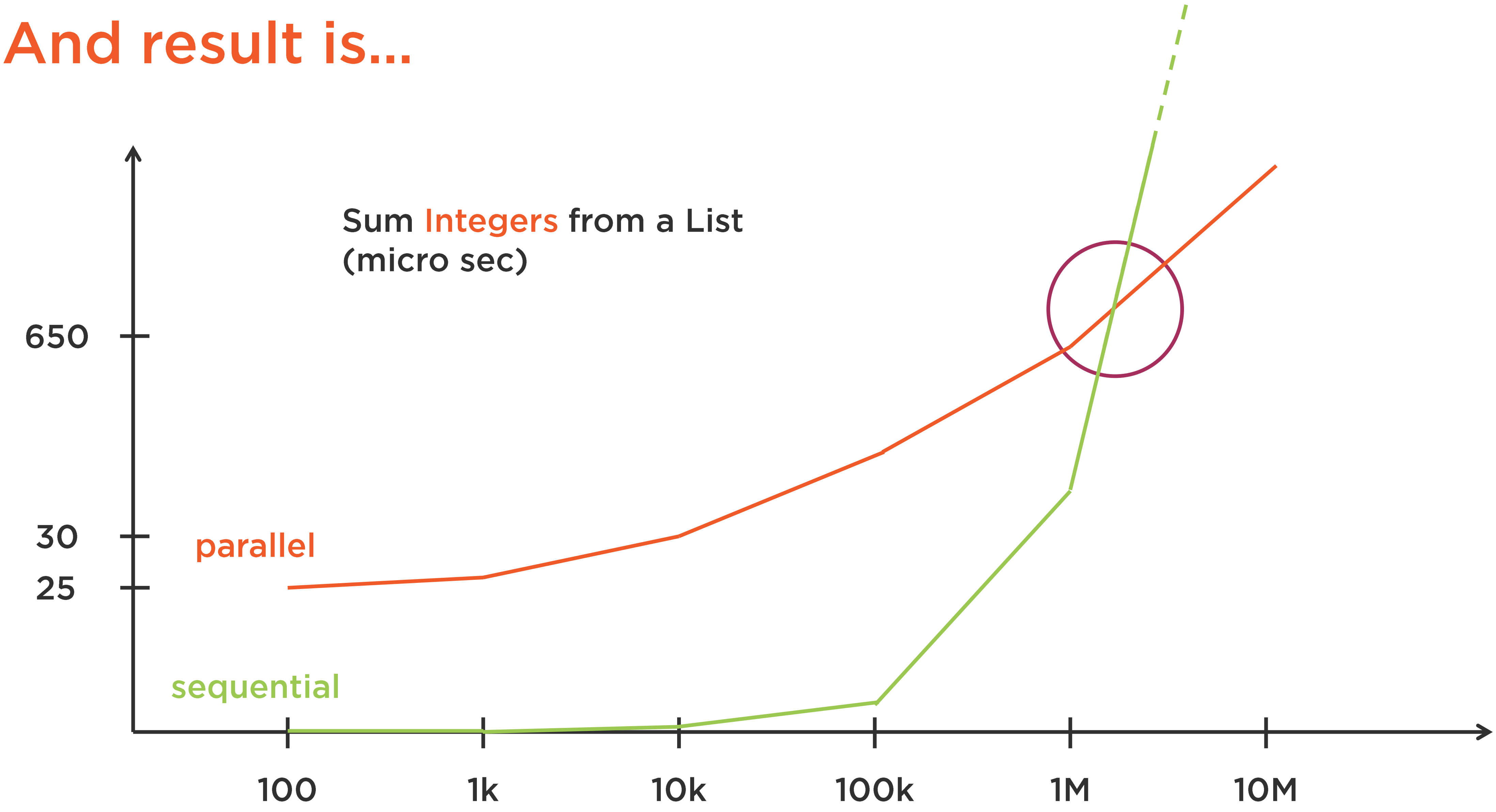
Let us write some code!

Let us compute parallel streams

Some with simple tasks!

And see if there is any difference

And result is...



And result is...

Benchmark	(N100)	Mode	Cnt	Score		Error	Units
M04_Sum.sum	100	avgt	15	0,078	±	0,002	us/op
M04_Sum.sum_parallel	100	avgt	15	24,116	±	0,127	us/op
M04_Sum.sum	1000	avgt	15	0,518	±	0,005	us/op
M04_Sum.sum_parallel	1000	avgt	15	25,744	±	0,392	us/op
M04_Sum.sum	10000	avgt	15	4,907	±	0,052	us/op
M04_Sum.sum_parallel	10000	avgt	15	32,676	±	0,245	us/op
M04_Sum.sum	10000000	avgt	15	6507,270	±	55,670	us/op
M04_Sum.sum_parallel	10000000	avgt	15	1083,358	±	42,048	us/op

How Can Things Go Wrong?

Things that can go wrong:

1) Hidden synchronizations
2) Faulty non-associative reductions

Hidden Synchronizations

```java
stream.filter(number -> number % 7 == 0)
      .findFirst(); // find the FIRST element
```

How can the API track that first element?

```
stream.limit(100) // takes the FIRST 100 elements
      .sum();
```

How can the API count the first 100 elements?

Demo

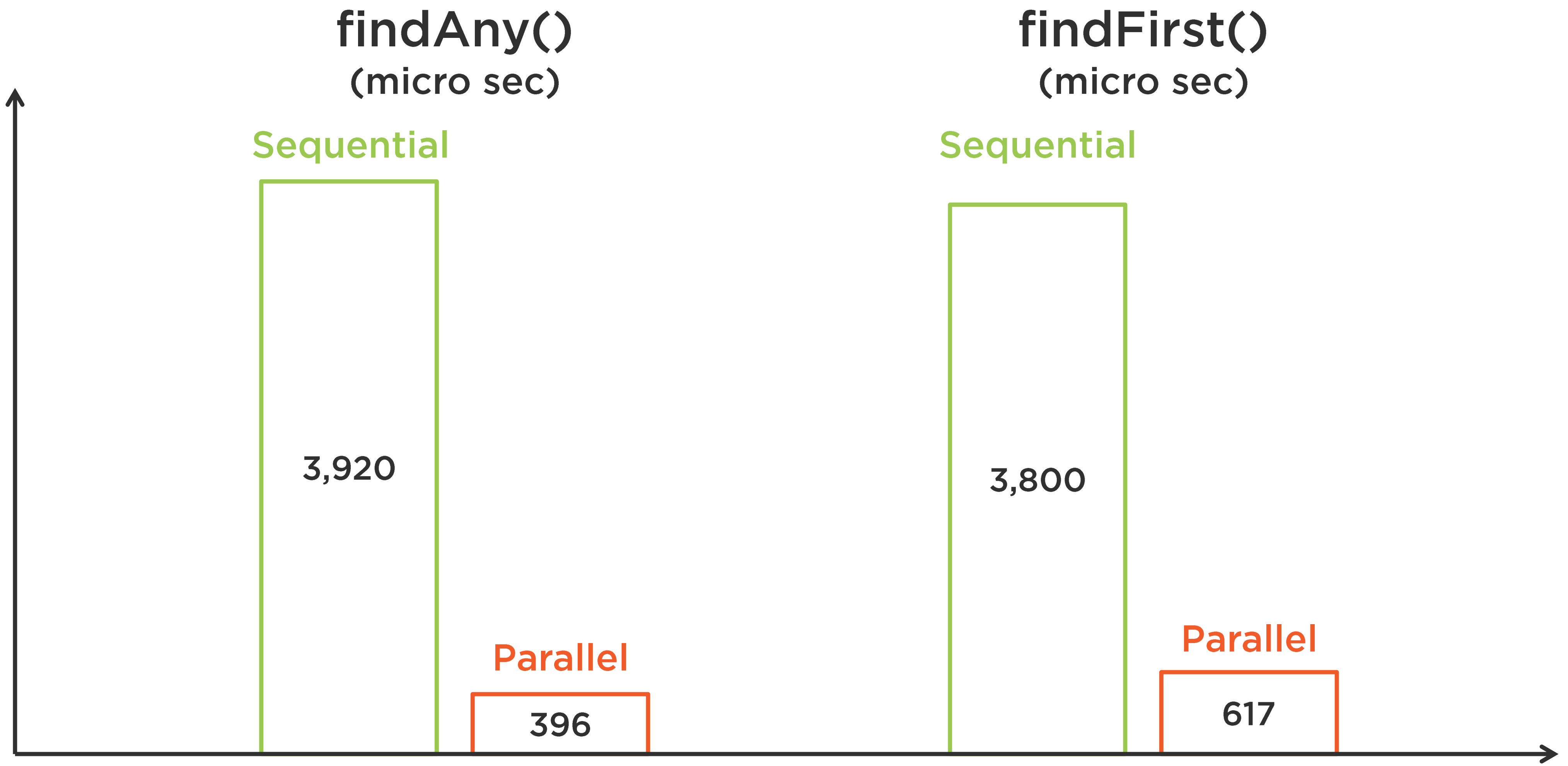
Let us write some code!

Let us compute parallel streams

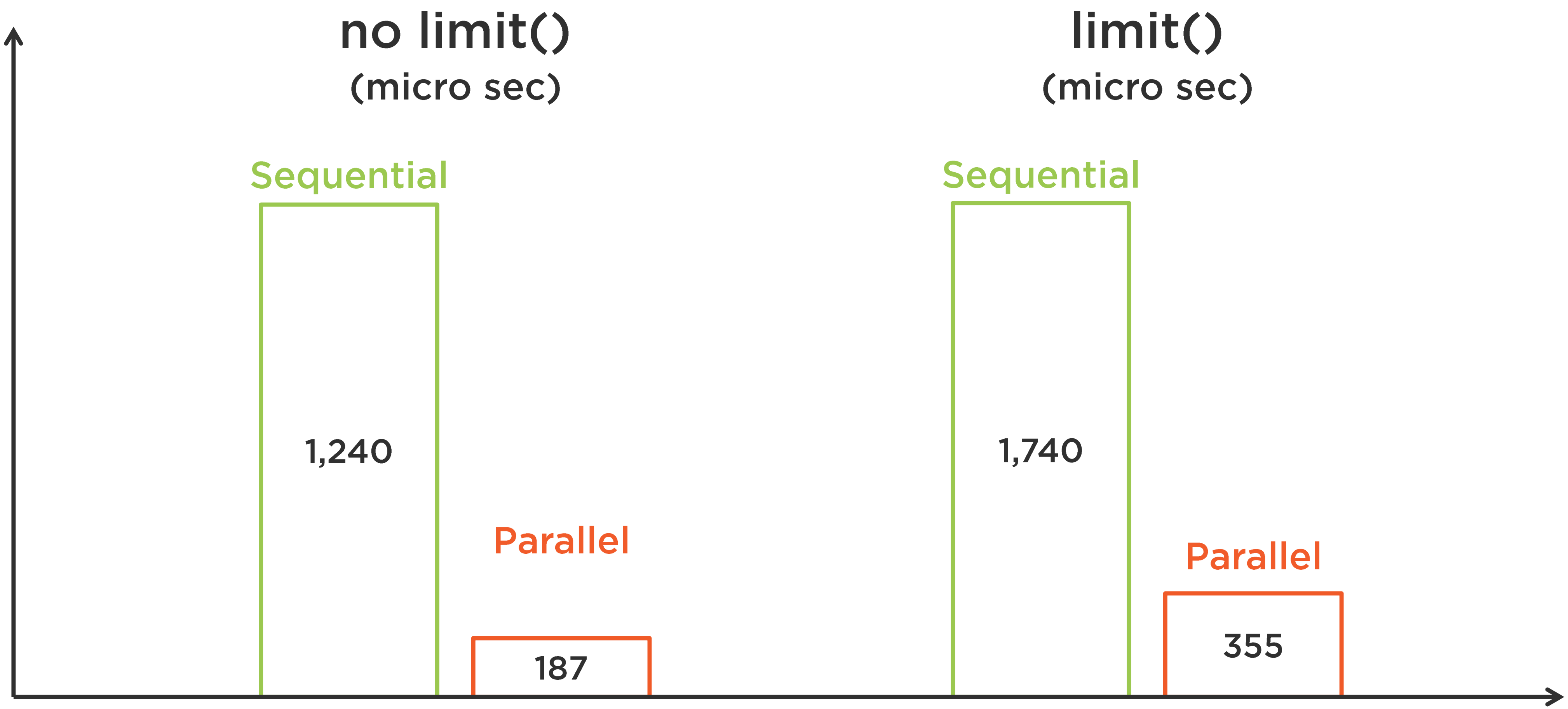
With hidden synchronization

And see if there is any difference

And result is...



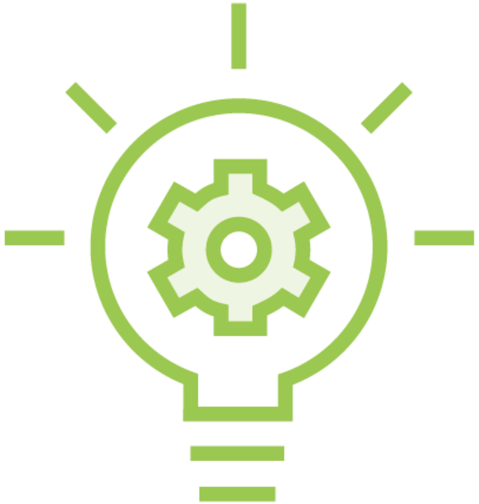
And result is...



And result is…

Benchmark	(N100)	Mode	Cnt	Score		Error	Units
M04_FindFirst.find_any_100	1000000	avgt	15	3917,334	±	80,895	us/op
M04_FindFirst.find_any_100_parallel	1000000	avgt	15	396,544	±	26,670	us/op
M04_FindFirst.find_first_100	1000000	avgt	15	3811,748	±	71,246	us/op
M04_FindFirst.find_first_100_parallel	1000000	avgt	15	617,370	±	33,394	us/op
M04_Max.max	1000000	avgt	15	1237,272	±	16,014	us/op
M04_Max.max_parallel	1000000	avgt	15	187,894	±	5,589	us/op
M04_Max.max_limit	1000000	avgt	15	1739,637	±	55,401	us/op
M04_Max.max_limit_parallel	1000000	avgt	15	355,791	±	18,332	us/op

Faulty Reduction



If you use the reduce() method

This operation is used to reduce the stream

And to join the partial result

This operation has to be associative

```java
BinaryOperator<T> reduction = ...;

T t1     = reduction.apply(a, b);
T result = reduction.apply(t1, c);

T t2     = reduction.apply(b, c);
T result = reduction.apply(t2, a);
```

Associative?

```
stream.reduce(0,
              (i1, i2) -> i1 + i2));
```

In that case the computed result is correct

```
stream.reduce(0,
              (i1, i2) -> i1*i1 + i2*i2));
```

But in that case it will be random…

```
(i1, i2) -> i1*i1 + i2*i2)
```

1	1	1	1	1	1	1	1

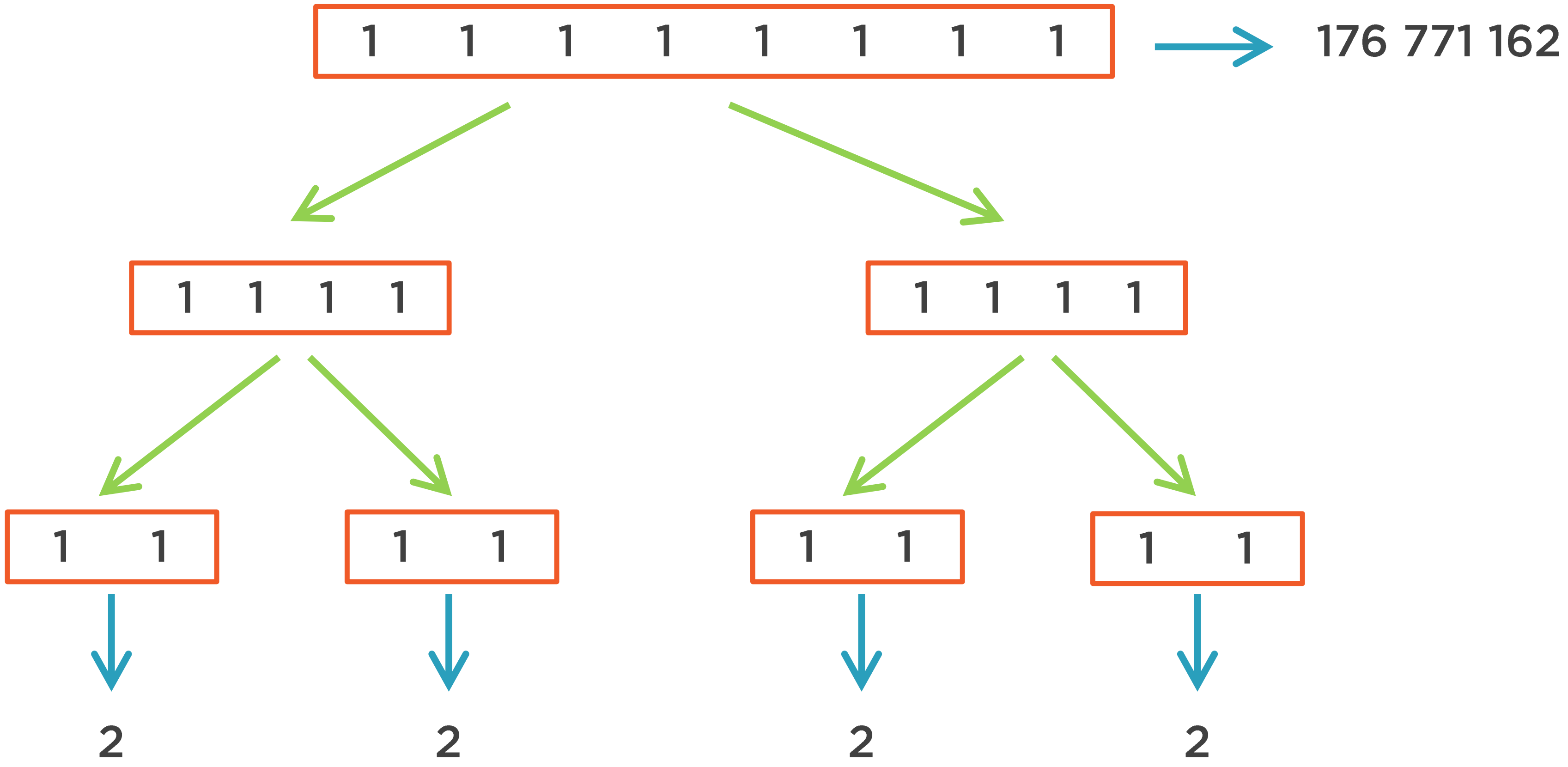
2

5

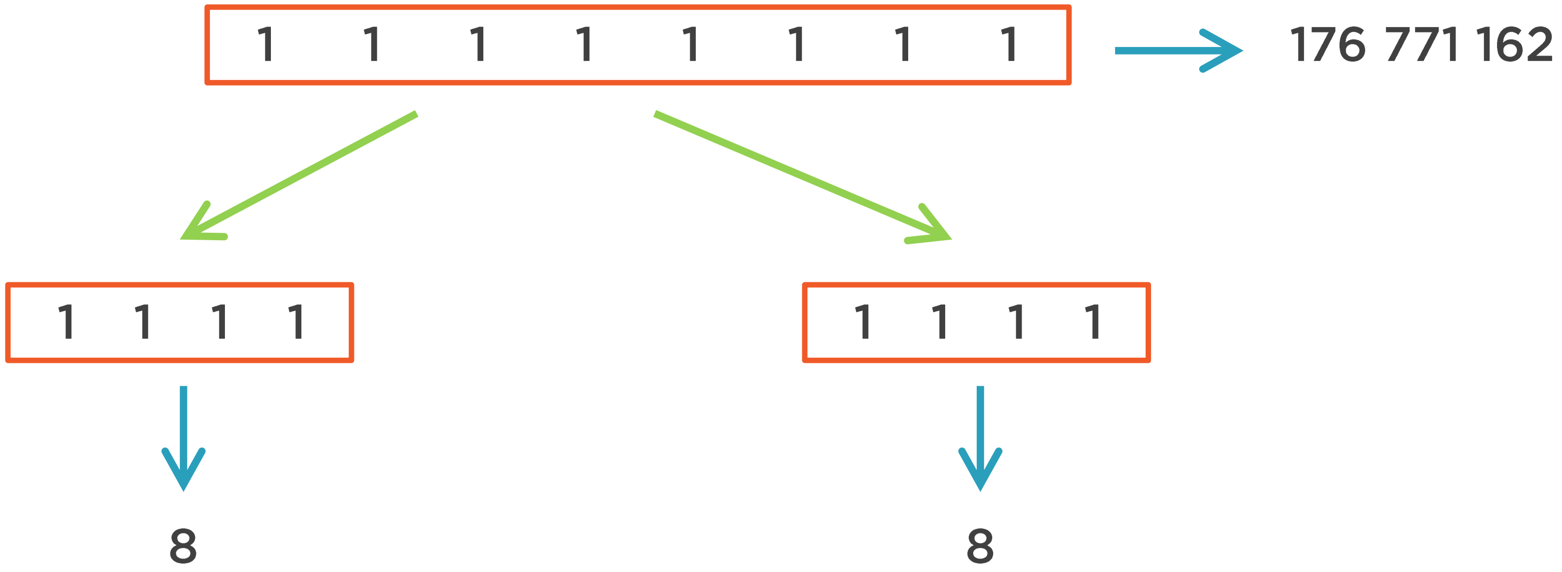
26

→ 176 771 162

```
(i1, i2) -> i1*i1 + i2*i2)
```



```
(i1, i2) -> i1*i1 + i2*i2)
```



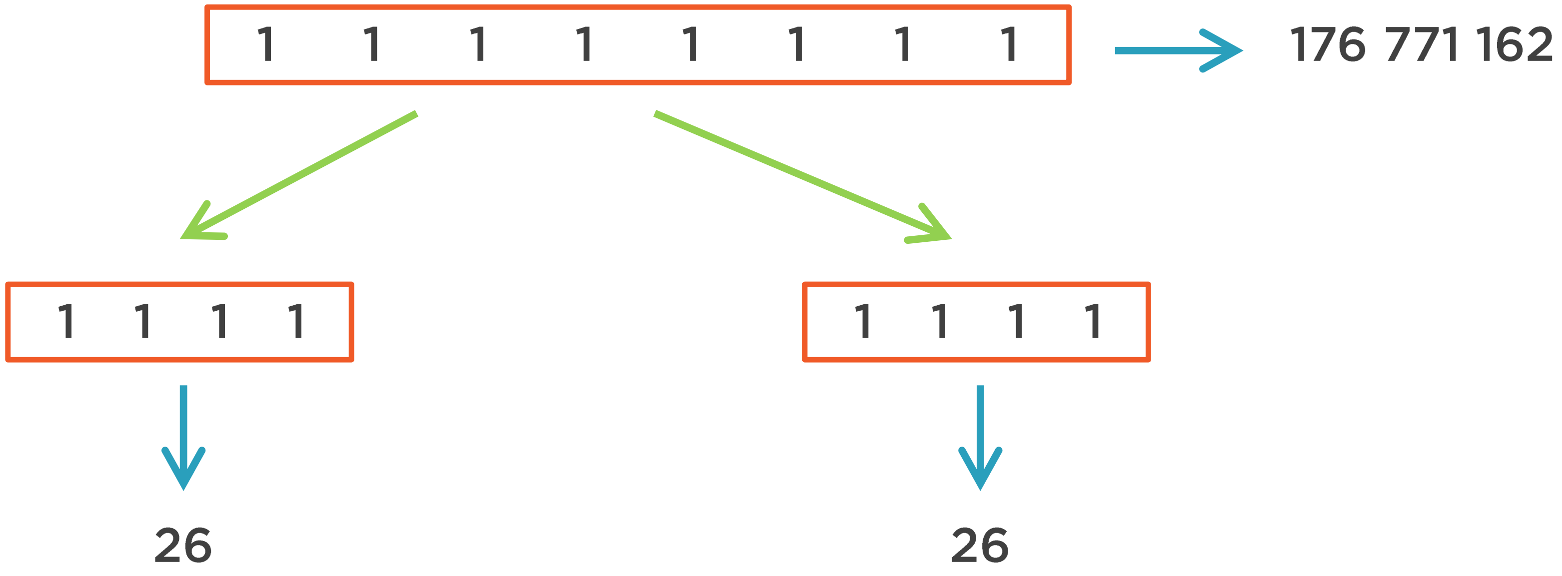
```
(i1, i2) -> i1*i1 + i2*i2)
```

1	1	1	1	1	1	1	1

→ 176 771 162

→ 128

`(i1, i2) -> i1*i1 + i2*i2)`



```
(i1, i2) -> i1*i1 + i2*i2)
```

1	1	1	1	1	1	1	1

→ 176 771 162

→ 1 352

The provided reduction operation is used to join partial results and has to be associative

Parallelism is not suited
for any kind of computations

Module Wrap Up

What did you learn?

Parallelism is implemented with the Fork / Join framework

It uses threads

Avoid hidden synchronization

It requires the reduction to be associative

Parallelism is not suited for everything...