Concurrent Programming in Java with Virtual Threads

Improving Reactive Programming with Virtual Threads

José Paumard

PhD, Java Champion, JavaOne Rockstar

@JosePaumard | https://github.com/JosePaumard

Why is it hard to increase the throughput of your application?

So far, there is one way: use reactive programming
Virtual threads are another solution, with a simpler and cleaner programming model

This is a Java course

Basic knowledge
of the language

And its main API: Collections, Streams

Elements of concurrent programming:

Threads and Executors
Future objects

This is a Java course

Based on Java 21

The IDE used is
IntelliJ IDEA

Ultimate or
Community version

Any other Java-compatible IDE will do

Version Check

This course was created by using:

Java 21

IntelliJ IDE Ultimate Edition

Not Applicable

This course is NOT applicable to:

Any Java version prior to 21

Relevant Notes

A note on frameworks and libraries:

Any version of Java after 21 is fine

Any other IDE (Eclipse, NetBeans, VS Code), will work fine

Reactive programming

Used with
callback-based patterns

Virtual Threads aim
to fix several problems:

The complexity of
the code you write

The observability of
your application

The maintenance cost of your application

Course Agenda

What is reactive programming, and how virtual threads can improve it

Introducing virtual threads

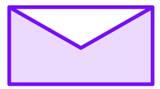
Using virtual threads to increase your throughput

Introducing structured concurrency

Replacing ThreadLocals with ScopedValues

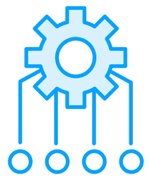
Throughput of an Application

Let Us Define a Few Words



Throughput: the number of items (request, messages, etc...) processed per second

Synchronous vs. asynchronous



Concurrent execution

Blocking vs. Non-blocking

Throughput

The number of elements you process per second, or millisecond

Your elements can be HTTP requests, DB requests, messages, etc...

Synchronous vs. Asynchronous

Synchronous vs. Asynchronous

A synchronous code is executed immediately

An asychronous code may be executed later

```
var number = 2d;
var sqrt = Math.sqrt(number);


var strings =
    List.of("one", "two", "three");

strings.forEach(System.out::println);


strings.sort(Comparator.naturalOrder());
```

- Most of the code you write is synchronous: it is executed as you write it

- Sometimes the code is executed later, or not at all

- Sometimes you don’t even know how many times it will be executed

```
var number = 2d;
var sqrt = Math.sqrt(number);


HTTPClient client = ...;
var response =
   client.get("http://mydata.com/data");
```

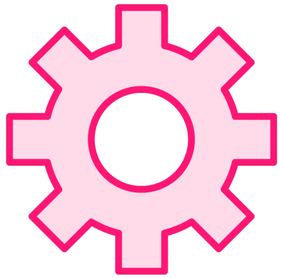
- Computing the square root blocks your code for a few nanoseconds
- This get() call blocks the execution of your code for a few milliseconds

Concurrent Execution

Concurrent Execution

A concurrent code is a code executed in another thread

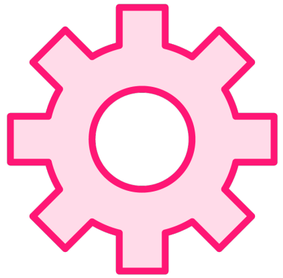
Concurrent Execution



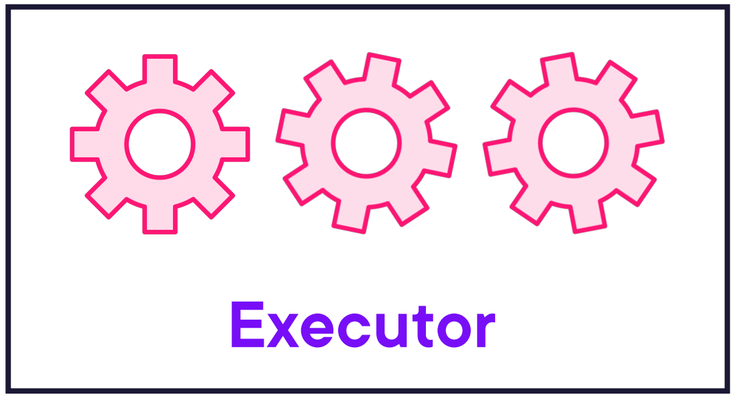
Main

```
var number = 2d;
var sqrt = Math.sqrt(number);
```

Concurrent Execution



Main



Executor

Task creation

```
Callable<Json> task = () -> {
   HTTPClient client = ...;
   var response =
      client.get("http://...");
}
```

Task submission

```
var executor = ...;
Future<Json> future =
   executor.submit(task);
```

```
// Execute the task
// in another thread
```

Result analysis

```
Json result =
   future.get();
```

Blocking vs. Non-Blocking

Blocking Code

Your core is blocking if it blocks the thread that is running it without using your CPU.

How is it possible?

Two ways:

1) Your thread is waiting for some I/O data

2) Or waiting to execute a synchronized block

Analyzing the Performance of a Request

```
URI uri =
URI.create("https://mydata.com/data");
HttpClient client =
   HttpClient.newBuilder().build();
HttpRequest request =
   HttpRequest.newBuilder(uri)
              .GET().build();

var response =
   client.send(request,
               HttpResponse.BodyHandlers
                  .ofString());

if (response.statusCode() == 200) {
   String body = response.body();
   ...
}
```

- You choose the URI of your request
- Then create a HttpClient objects
- Then a HttpRequest object
- Then you send this request on the Internet, and get a HttpResponse object in return
- That you can analyze for its status, and its content

The first step is about building objects in memory

Then you send your request on the Internet

Then you get back a response, that you process in memory

Concurrent Execution

```
URI uri = URI.create("https://mydata.com/data");
HttpClient client =
   HttpClient.newBuilder().build();

HttpRequest request =
   HttpRequest.newBuilder(uri)
              .GET().build();
```

Concurrent Execution

```
URI uri = URI.create("https://mydata.com/data");
HttpClient client =
   HttpClient.newBuilder().build();

HttpRequest request =
   HttpRequest.newBuilder(uri)
              .GET().build();

var response =
   client.send(request,
               HttpResponse.BodyHandlers
                  .ofString());
```

Concurrent Execution

```
URI uri = URI.create("https://mydata.com/data");
HttpClient client =
   HttpClient.newBuilder().build();

HttpRequest request =
   HttpRequest.newBuilder(uri)
              .GET().build();

var response =
   client.send(request,
               HttpResponse.BodyHandlers
                  .ofString());

if (response.statusCode() == 200) {
   String body = response.body();
   ...
}
```

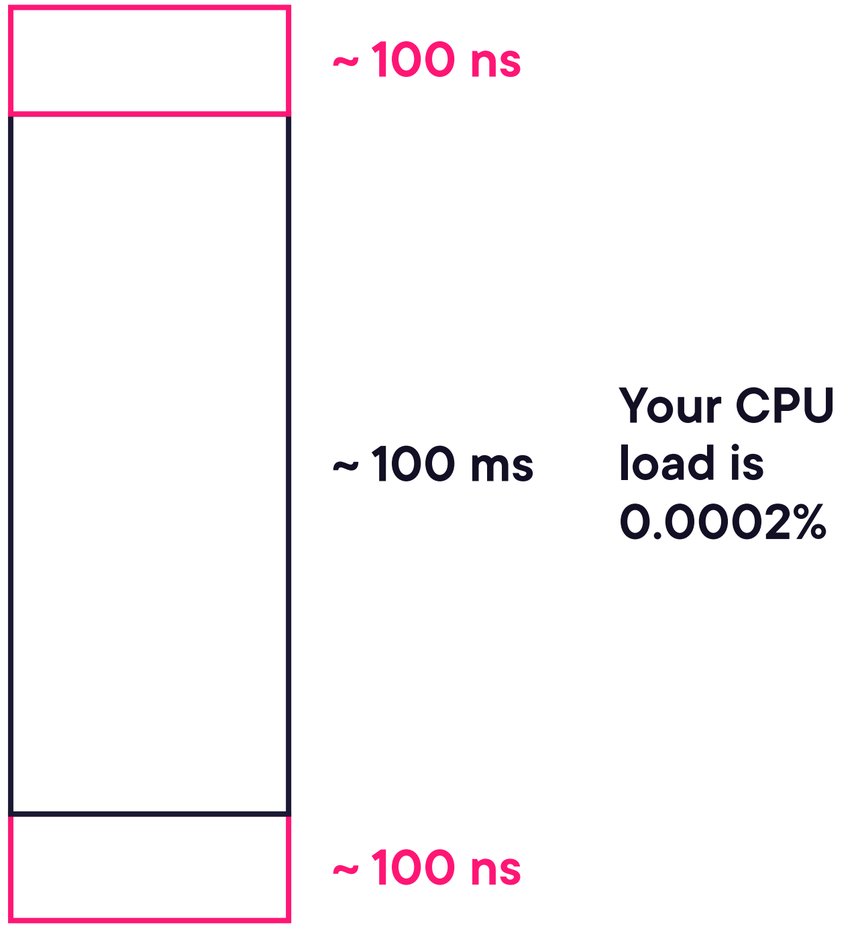
Concurrent Execution

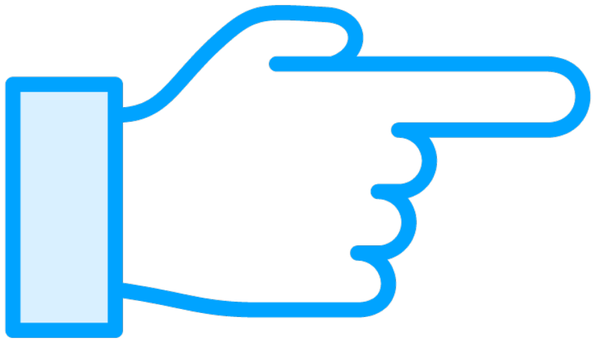
```
URI uri = URI.create("https://mydata.com/data");
HttpClient client =
   HttpClient.newBuilder().build();

HttpRequest request =
   HttpRequest.newBuilder(uri)
              .GET().build();

var response =
   client.send(request,
               HttpResponse.BodyHandlers
                  .ofString());

if (response.statusCode() == 200) {
   String body = response.body();
   ...
}
```





Because in-memory processing is so fast

Compared to a network request

During a request / response cycle

The CPU that runs your thread
is doing almost nothing

How can you fix that?

To keep your CPU core busy

The first solution would be

Send your network requests in parallel

One request per thread

There is a factor of 1 million

Between in-memory computation

And a request / response cycle time

So: how many threads do you need for one CPU core?

Answer: 1 million

Sending Concurrent Requests

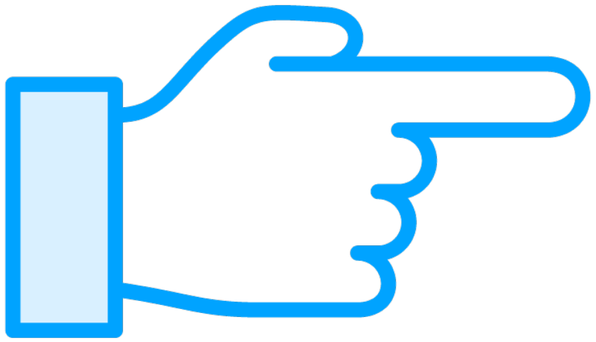
There is a factor of 1 million

Between in-memory computation

And a request / response cycle time

So: how many threads do you need for one CPU core?

Answer: 1 million



A java thread is a wrapper

On an operating system thread

Also called a kernel thread

Or a platform thread

So the cost of a Java thread

Is bound by the cost of a platform thread

Analyzing the Cost of One Thread

Cost of a single platform thread

Memory:	1 MB
Start-up time:	1 ms
Context switching:	100 µs

Analyzing the Cost of One Thread

Cost of a single platform thread

Memory: 1 MB
Start-up time: 1 ms
Context switching: 100 µs

Cost of 1 million platform threads

Memory: 1 TB
Start-up time: 1,000 s
Context switching: 100 s

The cost of a platform thread is way too high

This is why you need to pool them

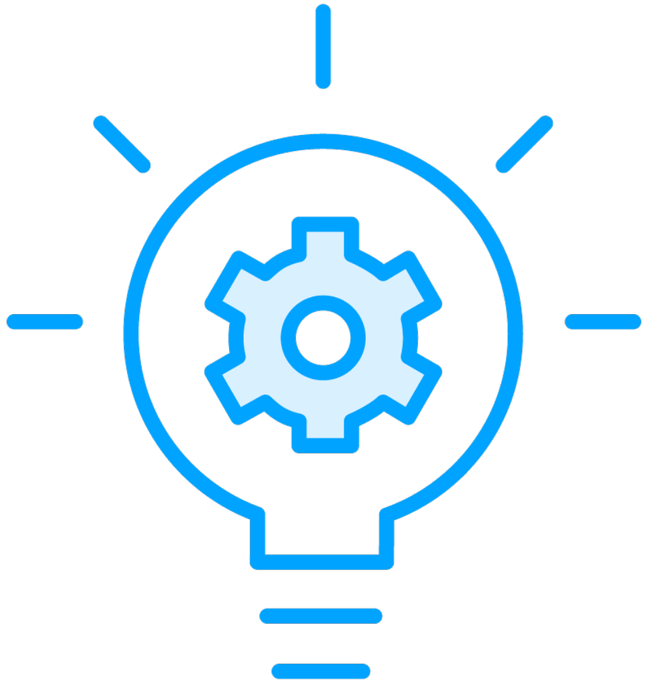
You cannot have 1 million of them on a regular machine

You need to find another solution

Because platform threads
are too expensive

The solution that consists of giving
one request to one thread

Does not work!



How can you fix that?

You need to keep your CPU core busy

So two solutions can come to mind:

First, give several requests to one thread

Second, build a thread that is lighter than a platform thread

Module Wrap Up

The problem of keeping your CPU busy when doing network requests

How Java threads are working

The cost of a platform thread

Why the Concurrent programming model cannot solve this problem

Up Next:

Introducing Virtual Threads