Java Concurrency Evolution

It is becoming crucial to understand the tooling around how the available computing capacity can be best utilized.

by
Csaba Bejan

Dec. 02, 20 · Java Zone · Opinion

Since the early days of Java, Threads were available to support concurrent programming. Interestingly till Java 1.1, green threads (virtual threads) were supported by the JVM, but they were dropped in favor of native OS threads, however with Project Loom on the horizon (targeted for Java 16 or later?), virtual threads are on track to become mainstream again.

The goal of this article is to go through the main milestones for the evolution of thread/concurrency handling in Java. As the topic can easily fill a library full of books, the following aspects are out of scope (basically, the goal is to take a look at the happy path of Java concurrency):

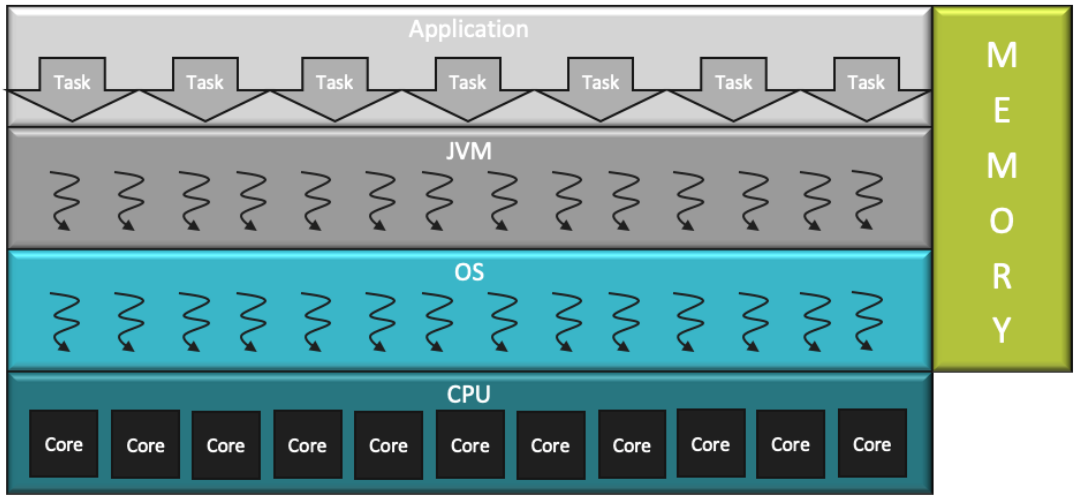
- Error handling – This is very limited, to say the least. Lombok's SneakyThrows is used to support readability in the code examples.
- Libraries and Frameworks supporting concurrency – There are endless alternative solutions for the JVM (e.g., Quasar, Akka, Guava, EA Async…)
- Sophisticated stop conditions – When kicking off a thread with a given task, it is not always clear how and how long to wait for its completion
- Synchronization between threads – The literature is endless about this.

So, what will we covering here, you may ask? A very valid question as it may seem that we excluded all the fun stuff. We will take an example task which we will solve multiple times using newer / different approaches, and we will compare their execution "behavior." By nature, this list can't be complete. I tried to collect the native solutions (the reactive approach is the outlier here, but it became mainstream enough not to leave it out).

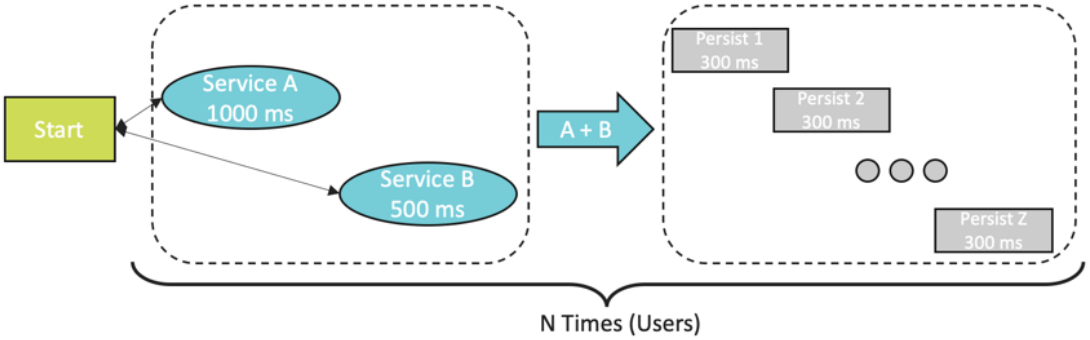
Threads in Java

Before we start, a few things to note about threads in Java.

- There is 1:1 mapping between JVM and OS threads. On the JVM side, there is just a thin wrapper over the OS thread.
- OS has a very general (hence slower) scheduling. It doesn't know anything about JVM internals.
- Creating and switching between threads is also expensive (slow) as it must go through the kernel.
- OS implementation of continuations includes the native call stack along with Java's call stack; it results in a heavy footprint.
- OS threads are pinned to CPU core.s
- The stack memory for the thread is reserved outside the heap by the O.S



The Task



	No Concurrency	Full Concurrency
Execution Time	N * (SA + SB + Z*Persist)	SA + Persist
N=10, Z=3	24.000 ms	1.300 ms
N=1000, Z=30	10.500.000 ms (~3 hours)	~1.300 ms

Our task is centered around concurrent calls. You can imagine a web server where a flow for one user looks like this:

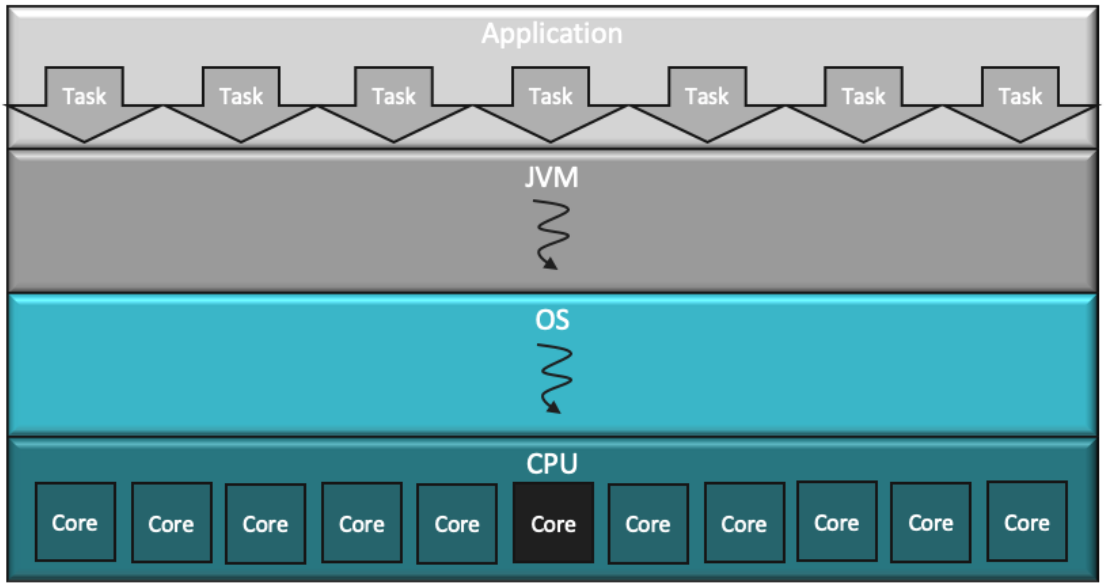
- Service A is called, and that call takes 1000ms
- Service B is called, and that call takes 500ms.
- The aggregated result of these service calls is persisted Z times each persistence (e.g,: File, DB, S3….) takes 300ms. All persistence calls take the same time which won't happen in reality but is much simpler to calculate with.

There could be an N number of users going through this flow concurrently. Naturally the first question could be: “How realistic is this workflow”? It is challengeable at best, but it provides a good tool to show the different approaches where we can just simply put the thread to sleep to emulate latencies. The long running computation can be considered to be a different approach and would result in different code.

You can find all the runnable code examples solving this problem here. Feel free to play around with them and set the N (users) and Z (persistence number) to different values to see how it reacts to it. As usual everything can be solved in many different ways these are just my solutions and there are cases where optimality was sacrificed for readability. We will go through the execution results and some interesting takeaways now.

No Concurrency

The simplest route to take. We can write a familiar code. There is great tooling around it. Easy and straightforward to debug and reason about it. However, it is easy to see that it results in less than optimal resource consumption. Our only JVM thread will be tied to one OS thread which will be pinned to one core so all the other cores will be idle (from our application's point of view). In the task description you can see that with the increasing number of users and persistence the running time explodes.



Code: The code itself is also straightforward. Short and to the point (there are some helper functions not detailed here to support readability. You can find it in the shared repository if you are interested.

Java

```
public void shouldBeNotConcurrent() {
    for (int user = 1; user <= USERS; user++) {
        String serviceA = serviceA(user);
        String serviceB = serviceB(user);
        for (int i = 1; i <= PERSISTENCE_FORK_FACTOR; i++) {
            persistence(i, serviceA, serviceB);
        }
    }
}
```

Native Multi-Threading

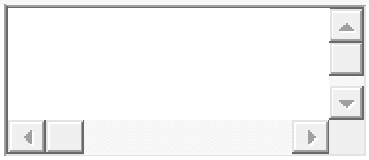
There are some challenges with multi-threading in general:

- Using the resources optimally (CPU Cores / Memory)

- Fine tuning thread number and thread management
- Lost control flow and context. Stack trace is bound to the thread not to the transaction so you will see only that small step assigned currently to the thread (debug, profile issues)
- Synchronizing the execution
- Debugging and Testing

Code: You can see that the size of the solution exploded from a few lines to well over a hundred. We had to implement Runnable(s), create and control threads manually. Also had to do some synchronization so callbacks are introduced. The logic itself is scattered and hard to follow. On the other hand, it performs quite well. It doesn't reach the 1.300ms "ideal" full concurrency execution time which we calculated above as creating threads and switching between them is expensive, but it gets close to it. Please note that for readability not all of the code is copied here. You can find everything in the shared git repository.

Java



```
public void shouldExecuteIterationsConcurrently() throws InterruptedException {
    List<Thread> threads = new ArrayList<>();
    for (int user = 1; user <= USERS; user++) {
        Thread thread = new Thread(new UserFlow(user));
        thread.start();
        threads.add(thread);
    }
    // Stop Condition - Not the most optimal but gets the work done
    for (Thread thread : threads) {
        thread.join();
    }
}
static class UserFlow implements Runnable {
    private final int user;
    private final List<String> serviceResult = new ArrayList<>();
    UserFlow(int user) {
        this.user = user;
    }
    @SneakyThrows
    @Override
    public void run() {
        Thread threadA = new Thread(new Service(this, "A", SERVICE_A_LATENCY,
user));
        Thread threadB = new Thread(new Service(this, "B", SERVICE_B_LATENCY,
user));
        threadA.start();
        threadB.start();
        threadA.join();
        threadB.join();
        List<Thread> threads = new ArrayList<>();
        for (int i = 1; i <= PERSISTENCE_FORK_FACTOR; i++) {
            Thread thread = new Thread(new Persistence(i, serviceResult.get(0),
serviceResult.get(1)));
            thread.start();
            threads.add(thread);
        }
        // Not the most optimal but gets the work done
```

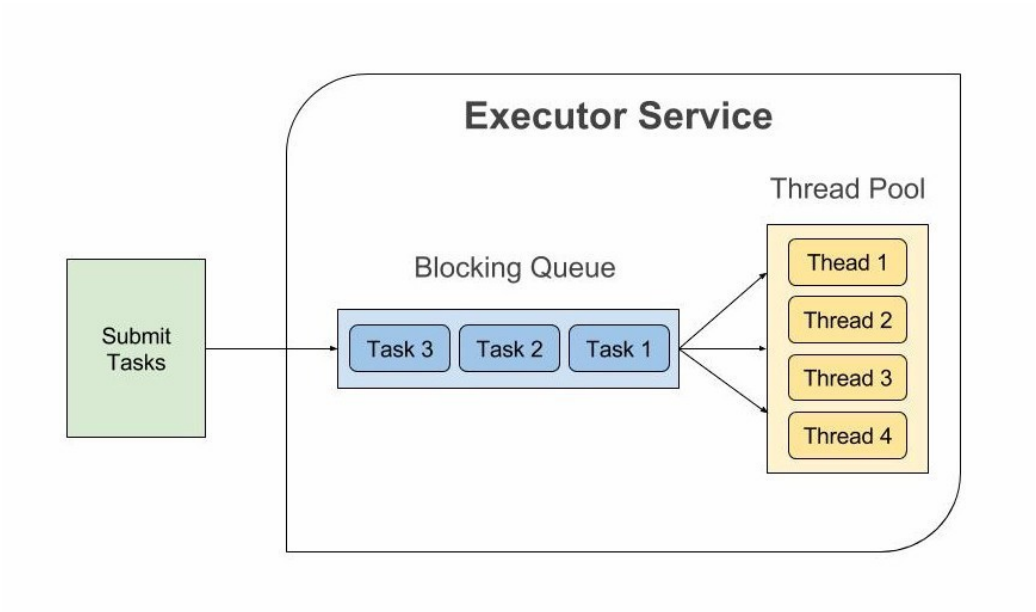
```
        for (Thread thread : threads) {
            thread.join();
        }
    }
    public synchronized void addToResult(String result) {
        serviceResult.add(result);
    }
}
// Service and Persistence implementations are omitted
```

Fun Fact: You can't create endless number of threads. It is OS dependent but for my 64 bit system each thread created takes up 1MB of memory (the allocated memory for the thread stack). When executing with 1000 users and 30 persistence I got out of memory consistently as it tries to create 33.000 threads which is more than the system memory available.

Iteration / Persistence Fork Factor	Threads	Total Time (target 1300ms)
1000 / 3	6000	~4800ms
1000 / 30	33000	Out of Memory

ExecutorService

In Java 1.5 the ExecutorService was introduced. The main goal is to support thread pooling this way mitigating the burden of creating new threads and dealing with them on a low level. The tasks are submitted to the ExecutorService and they are queued there. The available threads are picking up tasks from the queue.



Notes:

- It is still bounded by the operating system thread count
- If you take a thread even if you are not doing computation on it that thread is not available for others (wasted)
- Future is returned which sounds great however it is not compose-able and as soon as we try to get the returned value it blocks until it is not fulfilled

Code: In general, it is very similar to the native multi-threading approach. The main difference is that we are not creating the threads we just submit tasks (Runnable) to the ExecutorService. It takes care of the rest. The other difference is that we don't use callbacks here for synchronization between Service A and Service B we just simply block on the returned Futures. In our example, we use an ExecutorService with a pool of 2000 threads.

Java

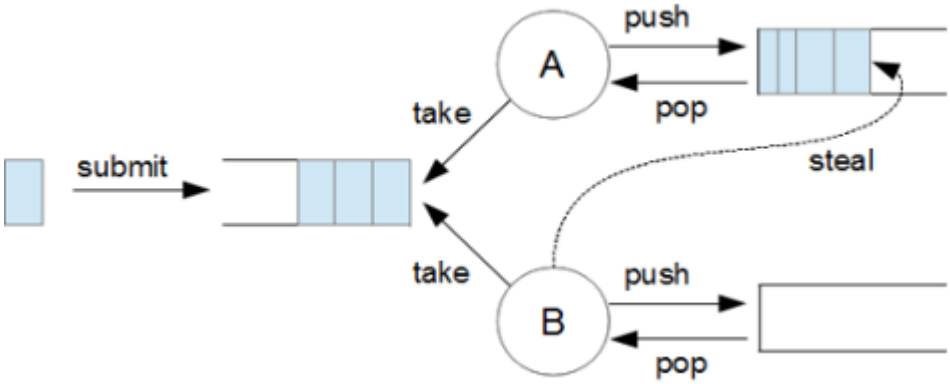
```
public void shouldExecuteIterationsConcurrently() throws InterruptedException {
    for (int user = 1; user <= USERS; user++) {
        executor.execute(new UserFlow(user));
    }
    // Stop Condition
    latch.await();
    executor.shutdown();
    executor.awaitTermination(60, TimeUnit.SECONDS);
}
static class UserFlow implements Runnable {
    private final int user;
    UserFlow(int user) {
        this.user = user;
    }
    @SneakyThrows
    @Override
    public void run() {
        Future<String> serviceA = executor.submit(new Service("A",
SERVICE_A_LATENCY, user));
        Future<String> serviceB = executor.submit(new Service("B",
SERVICE_B_LATENCY, user));
        for (int i = 1; i <= PERSISTENCE_FORK_FACTOR; i++) {
            executor.execute(new Persistence(i, serviceA.get(),
serviceB.get()));
        }
        latch.countDown();
    }
}
// Service and Persistence implementations are omitted
```

Fun Fact: Incorrectly configured pool size can lead to deadlock. If you set the pool size to a low number (e.g.: 10) and increase the user and the persistence number, you will see that nothing is happening. This is because in our solution one UserFlow requires multiple threads and multiple flows are taking up all of the threads from the pool however they need additional threads to complete which they will never get as the pool is empty at this point. In the example we use fixedThreadPool but there are other solutions which are more dynamic.

Iteration / Persistence Fork Factor	Threads	Total Time (target 1300ms)
1000 / 3	2000	~2383ms
1000 / 30	2000	~4968ms

Fork/Join Framework

Introduced in Java 1.7 at the core it builds on the ExecutorService. It was intended for smaller recursive tasks. There were expectations that it will replace the ExecutorService however there is also a reduction in the amount of developer control over concurrent execution so usage of ExecutorService could be still justified. The big new enhancement is the introduction of work steal. If a thread is getting overwhelmed and its internal queue fills up another thread instead of picking up task from the main queue can “steal” task from another threads internal queue.



Code: The code is getting shorter and more compact (partly because of usage of streams and lambdas of course). Now we implement RecursiveAction for the main user flow which we can submit to the Fork / Join Pool. As I said the usage of the framework for this task is not optimal, the problem itself is not really recursive by nature, there could be much better approaches, just wanted to demonstrate how it would work.

Java

```
public void shouldExecuteIterationsConcurrently() throws InterruptedException {
    commonPool.submit(new UserFlowRecursiveAction(IntStream.rangeClosed(1,
USERS)
            .boxed()
            .collect(Collectors.toList())));
    // Stop Condition
    commonPool.shutdown();
    commonPool.awaitTermination(60, TimeUnit.SECONDS);
}
public static class UserFlowRecursiveAction extends RecursiveAction {
```

```
    private final List<Integer> workload;
    public UserFlowRecursiveAction(List<Integer> workload) {
        this.workload = workload;
    }
    @Override
    protected void compute() {
        if (workload.size() > 1) {
            commonPool.submit(new UserFlowRecursiveAction(workload.subList(1,
workload.size())));
        }
        int user = workload.get(0);
        ForkJoinTask<String> taskA = commonPool.submit(() -> service("A",
SERVICE_A_LATENCY, user));
        ForkJoinTask<String> taskB = commonPool.submit(() -> service("B",
SERVICE_B_LATENCY, user));
        IntStream.rangeClosed(1, PERSISTENCE_FORK_FACTOR)
                .forEach(i -> commonPool.submit(() -> persistence(i,
taskA.join(), taskB.join())));
    }
}
```

Fun Fact: Given our example with the same pool size when the ExecutorService is getting stuck due to work steal mechanism the Fork / Join Framework still completes but please note it is very use case specific Fork / Join can also deadlock it just depends on how the main stack is broken up to smaller ones recursively. Our task is not optimal for recursive decomposition and as you can see due to the additional logic Fork / Join Framework performs much worse compared to the ExecutorService however it is more stable.

Iteration / Persistence Fork Factor	Threads	Total Time (target 1300ms)
1000 / 3	2000	~3581ms
1000 / 30	2000	~26124ms

CompletableFuture

Introduced in Java 8 and under the hood it is implemented on the top of the Fork / Join Framework. It is the long awaited “evolution” of the Future interface which did not have any methods to combine computation results (addresses callback hell) or handle possible errors.

Notes:

- Introduces a more functional programming style
- Has about 50 different methods for composing, combining, and executing asynchronous computation steps and handling errors.
- Most methods of the fluent API in CompletableFuture class have two additional variants with the Async postfix. These methods are usually intended for running a corresponding step of execution in another thread.

Code: The code is getting even more shorter and compact however in the coding style there is a noticeable paradigm shift. Combining async execution results is very natural. No need for additional boilerplate. However for someone who is new to functional programming style the code can look very foreign and needs some getting used to.

Java

```
public void shouldExecuteIterationsConcurrently() throws InterruptedException,
ExecutionException {
    CompletableFuture.allOf(IntStream.rangeClosed(1, USERS)
            .boxed()
            .map(this::userFlow)
            .toArray(CompletableFuture[]::new)
    ).get();
}
@SneakyThrows
private CompletableFuture<String> userFlow(int user) {
    return CompletableFuture.supplyAsync(() -> serviceA(user), commonPool)
            .thenCombine(CompletableFuture.supplyAsync(() -> serviceB(user),
commonPool), this::persist);
}
@SneakyThrows
private String persist(String serviceA, String serviceB) {
    CompletableFuture.allOf(IntStream.rangeClosed(1, PERSISTENCE_FORK_FACTOR)
            .boxed()
            .map(iteration -> CompletableFuture.runAsync(() ->
persistence(iteration, serviceA, serviceB), commonPool))
            .toArray(CompletableFuture[]::new)
    ).join();
    return "";
}
```

Fun Fact: While it builds on top of the Fork / Join Framework the execution results are showing much better performance. By the way CompletableFuture is one of the few monads in Java but that is a completely different rabbit hole.

Iteration / Persistence Fork Factor	Threads	Total Time (target 1300ms)
1000 / 3	2000	~2170ms
1000 / 30	2000	~8600ms

Reactive

There is a very important distinction we have to make first. Reactive architecture and Reactive programming are two different things. We are talking about Reactive programming here which is basically oriented around asynchronous data streams and has very robust support around error handling and backpressure (however we don't deal with that here). It can be considered as the evolution of the CompletableFuture but it is much more than that of course. The thread management is happening on the library/framework side and the main goal is to structure your program as an asynchronous event stream.

Notes:

- The trinity is Observable: Emits the data, Observer: Consumes the data, Scheduler: Thread management
- It is very "viral" meaning if you start to use it somewhere it pops up all around your codebase as a reactive solution has to be end to end, blocking somewhere throws away all the benefits.
- There are a lot of implementations, the original for Java was the RxJava library, however nowadays Spring's Rector is more dominant. A more "radical" approach for reactive programming is the Akka framework which implements the actor model.

Code:

This is the only non naive Java solution. We are taking a look at Spring's Reactor but RxJava is very similar. At this point for someone who is more familiar with the imperative coding style the code can be really strange. Actually, working with reactive code requires a mindset shift not just in coding but in testing and debugging too (which can be very painful or rewarding depending on where you stand but it is an investment nonetheless). However, if someone masters this coding style he/she can write very performant code as the support and libraries are constantly growing around it.

Java

```
public void shouldExecuteIterationsConcurrently() {
    Flux.range(1, USERS)
            .flatMap(i -> Mono.defer(() ->
userFlow(i)).subscribeOn(Schedulers.parallel()))
            .blockLast();
}
private Mono<String> userFlow(int user) {
    Mono<String> serviceA = Mono.defer(() ->
Mono.just(serviceA(user))).subscribeOn(Schedulers.elastic());
    Mono<String> serviceB = Mono.defer(() ->
Mono.just(serviceB(user))).subscribeOn(Schedulers.elastic());
    return serviceA.zipWith(serviceB, (sA, sB) -> Flux.range(1,
PERSISTENCE_FORK_FACTOR)
            .flatMap(i ->
                    Mono.defer(() -> Mono.just(persistence(i, sA,
sB))).subscribeOn(Schedulers.elastic())
            )
            .blockLast()
    );
}
```

Fun Fact: While you certainly have more control you don't need to deal with thread pool fine tuning, that happens transparently under the hood. In our example we didn't explore the real strength of this approach mainly the error handling and backpressure.

Iteration / Persistence Fork Factor	Threads	Total Time (target 1300ms)
1000 / 3	elastic	~5635ms
1000 / 30	elastic	~10070ms

Project Loom

Will be released at some point Currently what we know is that it is available as EA in Java 16 so we can play around with it but there are no promises that it will be actually included in Java 16. It will be released when it is ready. Project Loom is not one thing, but many interconnecting features related to virtual threads. For our specific case we are most interested in Virtual Threads and Structured Concurrency. With Virtual Threads the 1:1 mapping between JVM and OS threads goes away. This way making the JVM threads comparably extremely cheap. With structured concurrency a threads lifetime correlates to its code block so synchronization between them is clear and the coding style resembles the well know imperative one.

Notes:

- Tooling around it is not that great. I had issues with IntelliJ, Gradle, Lombok, JProfiler…
- Metadata and stack footprint with context switch time is an order of magnitude smaller compared to using native OS threads
- Due to the problems above I created a dedicated repository for the Loom code

Code: The code itself while is not the same as the original nonconcurrent one maybe this is the closest to it. It feels somewhat familiar. There is a new type of ExecutorService to support virtual threads and as it implements the AutoCloseable interface we can use it in try with resources. This way supporting structured concurrency (the parent thread will wait for all threads created inside the try block to terminate)

Java

```
@SneakyThrows
private void startConcurrency() {
    try (var e = Executors.newVirtualThreadExecutor()) {
        IntStream.rangeClosed(1, USERS).forEach(i -> e.submit(() ->
userFlow(i)));
    }
}
@SneakyThrows
private void userFlow(int user) {
    List<Future<String>> result;
```

```
    try (var e = Executors.newVirtualThreadExecutor()) {
        result = e.invokeAll(List.of(() -> serviceA(user),() ->
serviceB(user)));
    }
    persist(result.get(0).get(), result.get(1).get());
}
private void persist(String serviceA, String serviceB) {
    try (var e = Executors.newVirtualThreadExecutor()) {
        IntStream.rangeClosed(1, PERSISTENCE_FORK_FACTOR)
                .forEach(i -> e.submit(() -> persistence(i, serviceA,
serviceB)));
    }
}
```

Fun Fact: As it is a JVM level enhancement when Loom will be introduced the performance improvements will be inherited by many existing implementations/libraries.

As you can see, we don't know actually how many OS threads were created under the hood but it doesn't really matter. However it is clear that this is the solution where we could get closest to our theoretical optimum of 1.300ms. Increasing the user or the persistence number doesn't explode the execution time like with the other solutions so we can say it scales really well.

Iteration / Persistence Fork Factor	Threads	Total Time (target 1300ms)
1000 / 3	???	~1448ms
1000 / 30	???	~1941ms

Conclusion

Concurrency is clearly complicated. As we showed our very simple and clear imperative code became really complicated, hard to read and reason about while almost impossible to debug. However, there is light at the end of the tunnel for those who don't want or simply don't have the time to become experts in the field of concurrency. With Loom on the horizon performant Java code should be available for everyone.

There is a debate around whether after Loom is released reactive programming becomes obsolete. I think there will be a coexistence as there are no silver bullets. There are problems where Reactive Programming is a better fit with the currently available advanced tooling however Loom can be considered a more natural and more familiar next step in the Java Concurrency Evolution.