Coroutine against Goroutine: comparison between ***Kotlin*** and ***GoLang*** concurrency

Activity Project in *Operating Systems M*

Luca Marchegiani

March 5, 2025

Contents

Abstract

In this paper we are going to make a comparison between `Kotlin` and `Go` concurrency that is the main focus of the activity project for the *Operating Systems M* course in the master degree in *Computer Science Engineering* at the *Alma Mater Studiorum* University of Bologna.

After a general description of the languages and their concurrency management, we will try to go into an experimental comparison between their implementation of the *coroutine* pattern using a concurrent version of the matrix multiplication algorithm.

1 Introduction

1.1 `Kotlin` general overview

`Kotlin` is a modern, multi-platform and blended programming language developed by Jetbrains that works on JVM such as `Scala`. `Kotlin` is completely interoperable with `Java`[1] and it is *object-oriented* with strong elements of functional programming that make it more powerful than his father `Java`. As specified in the main page of the official website, `Kotlin` has also the advantages to be *concise*, *safe in nullability*, *expressive*, *asynchronous*, *interoperable* and *multiplatform*.

Furthermore, since 2019, Google has declared `Kotlin` as the preferred language for developing `Android` applications, establishing it as the *de facto* official language. As anticipated, `Kotlin` supports also multiplatform allowing the developer to write `Kotlin` code that can be compiled for native platforms (including `Android` and `iOS`), `JVM` and `JavaScript`.

```kotlin
// Concise ************************************************************
val object = Object()

// Safe in nullability ****************************************************
var name: String = "myName"
var nullableName: String? = null
// NOT POSSIBLE TO ASSIGN 'nullableName = null'
// The compiler will say "Null can not be a value of a non-null type String"

// Expressive ************************************************************
fun List<Int>.evenSum() = filter { it % 2 == 0 }.sum()

fun user(block: UserBuilder.() -> Unit): User {
 val user = User();
 user.block()
 return user
}

fun main() {
 val numbers = listOf(1, 2, 3, 4, 5, 6)
 println("Sum of even numbers: ${numbers.evenSum()}")

 val user = user {
```

[1] All classes written in `Kotlin` can be callable from `Java` code and vice-versa.

```kotlin
    name = "luca"
    surname = "marchegiani"
  }
  println("User: $user")
}

// Asynchronous ****************************************************
suspend fun fetchData(): String {
  delay(2000) // Simulating network delay
  return "Data received"
}

fun main() = runBlocking {
  val result = async { fetchData() }
  println(result.await())
}

// Interoperable ***************************************************
public class JavaClass {
  public static String greet(String name) {
    return "Hello, " + name + "!";
  }
}

fun main() {
  val message = JavaClass.greet("Kotlin")
  println(message) // Calls the Java method from Kotlin
}
```

Listing 1: Exemples of `Kotlin` 's features

1.2 `Go` general overview

`Go` is an open source programming language developed and supported by `Google`. It's an *imperative* and *typed* language that is strongly designed for concurrency thanks to its very easy way to launch process and its efficiency. The idea of this language is to maintain the run-time efficiency of `C` but with more readability and usability. Differently from `C`, `Go` has *memory safety*, *garbage collection* and *structural typing* as said by Wikipedia.

In the last years, `Go` also supports mobile platform (`Android` and `iOS`) as described in the official wiki, by writing *all-Go native mobile applications* or *SDK applications* with bindings for `Java` or `Objective-C`. There is also a toolkit called Fyne that is free and open source that makes easy to build graphical application also for mobile using `Go` . Anyway, `Go` is not widely used for mobile applications since it is primarily designed for backend and system programming. Additionally, `Kotlin` is the official language for Android and is also multi-platform.

```go
// Designed for concurrency ******************************************
func asyncTask() {
  fmt.Println("Running asynchronously!")
}

func main() {
  go asyncTask()
```

```go
}

// Readability and usability ******************************************
func add(a, b int) int {
 return a + b
}

func main() {
 nums := []int{1, 2, 3, 4, 5}

 for _, num := range nums {
  fmt.Println("Number:", num)
 }

 result := add(3, 7)
 fmt.Println("Sum:", result)
}
```

Listing 2: Exemples of `Go` 's features

1.3 Fast comparison

`Go` feels like a modern reinterpretation of C, emphasizing simplicity and efficiency, while `Kotlin` embodies a contemporary approach to programming with expressive syntax and power features, with full support to functional programming.

From the concurrency point of view, **both of this language supports coroutines** as concurrent units of execution. *Coroutines* are lightweight processes that can run over multiple OS threads, allowing to save on thread management costs.

Coroutines and threads are very similar, but the main difference is that the firsts are *non-preemptive* (or *cooperatives*) differently from the seconds that are typically *preemptive* and scheduled by the OS. Indeed, the execution of a coroutine can be suspended and resumed by the developer, calling some operations, and not by the OS. We will go in the details for both of this language.

To conclude this fast overview, you can find the table below that summarizes the main differences between the two languages:

Feature	Go (Golang)	Kotlin
Typing	Statically typed	Statically typed
Null Safety	No built-in null safety	Built-in null safety with nullable and non-nullable types
Concurrency	Goroutines for lightweight concurrency	Coroutines for asynchronous programming
Interoperability	Limited interoperability with C/C++	Full interoperability with Java, multiplatform
Syntax	Simple and minimalistic	Concise with modern features (lambda, receivers, infix functions)
Use Cases	System programming, cloud services	Android development, server-side applications
Standard Library	Rich standard library with built-in concurrency support	Rich standard library with extensive collection utilities, all the Java libraries on `Kotlin` JVM
Tooling	Supported by editors like *VS Code* or *Goland*	Supported by *IntelliJ IDEA* and Android Studio
Community	Strong community with a focus on simplicity and performance	Growing community with a focus on modern development practices
Functional Programming	Limited support for functional programming	Strong support with higher-order functions, lambdas, and more

2 Overview of the concurrency in `Kotlin` and `Go`

As specified in the introduction, `Kotlin` and `Go` exposes concurrency thanks to **coroutines** and other tools that let the developer manage their synchronization. To be precise, while `Go` has only coroutines to implement concurrency, `Kotlin` has a more sophisticated and complete *framework*: indeed, lots of `Kotlin` application (including `Android` apps) run over a `JVM` (or on the `ART`), so all the standard `Java` threading packages are available.

Anyway, as we already said, **coroutines** are **lightweight processes for cooperation that execute over OS threads** and that can suspend at a certain point and resume later at the same point, but with the possibility to execute on a different thread. The main advantage of using them instead threads is that **switching between coroutines does not require any *system call***, ensuring lower management costs. This introduces great advantages, especially for *asynchronous* computation.

To conclude this general introduction, coroutines can use *shared memory* or *message passing*, based on what developer choose to use. Indeed, both `Kotlin` and `Go` provides supports for the two mechanisms: *semaphore* and *mutex* for shared memory and *channels* for *message passing*.

2.1 Kotlin concurrency overview

We said that Kotlin is based on the JVM (but can also compile JavaScript or native using LLVM) and is interoperable with Java. The main implementation of Kotlin is done in its compiler: for Kotlin on JVM, all classes are compiled as normal Java classes. This means that **Kotlin can access to all threading** packages exposed by Java (and this is also valid for Android). So, in Kotlin **it is possible to use the standard threads** that are provided by Java.

Even if there is the possibility to use the standard Java threads, as said, Kotlin introduces the new kotlinx.coroutines library to realize concurrency by adopting *coroutines*. Coroutines are *instances of suspendable computation* letting the developer to easily write **asynchronous and non-blocking code** that can run concurrently, without using *callback* or *promises*. The main mechanism that turns around Kotlin coroutines is the concept of **suspending function**: a special type of Kotlin method that can suspend the execution of the current coroutine without blocking the current thread. A function can be marked as suspend by simply adding this modifier to its signature.

```
suspend fun task() {
 // Asynchronous task
}
```

Listing 3: The first *suspend* function

A suspend function must be called from a coroutine or another suspend function, otherwise the compiler throws a compilation error.

2.1.1 Realization of coroutines in Kotlin

Before going into the details of coroutines in Kotlin , we have to introduce some basic concepts[2]:

- Job:
  The object that represents the *background job* of one coroutine. When a coroutine is launched, the launch method immediately returns the reference to the Job associated to the coroutine. **A job represents the lifecycle of a coroutine** and can be used to *cancel* its execution. It can have six possible states, each coded by a combination of the three properties of the Job class: isActive, isCompleted and isCancelled. The following table summarizes the possible states of a Job and the value of the three properties for each state:

State	Type	isActive	isCompleted	isCancelled
New	initial	false	false	false
Active	initial	true	false	false
Completing	transient	true	false	false
Cancelling	transient	false	false	true
Cancelled	final	false	true	true
Completed	final	false	true	false

Table 1: States of a Job

[2]See medium.com for additional details.

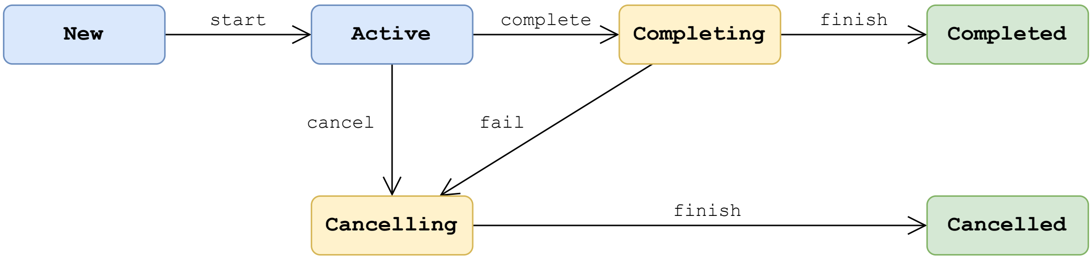

Figure 1: Lifecycle of `Kotlin` coroutine in `Job`

The graphs shown in the 1 represents the entire lifecycle of a `Job`, so it also represents the lifecycle of a `Kotlin` coroutine.

- CoroutineDispatcher:
  As already said, in their lifecycle coroutines can run among different threads. For example, suppose to have a coroutine C_1 that is started on the thread T_1 that executes its code:

  1. C_1 starts its execution on thread T_1;
  2. during its execution, C_1 encounter an instruction I_1 that suspends itself waiting for something (but the instruction does not suspend the thread);
  3. C_1 is suspended by I_1 and another coroutine C_2 starts to execute on T_1;
  4. C_2 is executing on T_1 while I_1 returns resuming C_1 from its suspension, but now T_1 is not available because it is executing the code of C_2;
  5. C_1 may execute on another available thread T_2 while C_2 continue to run in parallel on T_1 (if the configuration allows it).

  `CoroutineDispatcher` is the object that *dispatchs* coroutines between the different available threads. It is one of the most important component offered by the `Kotlin` concurrency framework because it determines among which thread a couroutine has to run: for example, in `Android`, using `Dispatchers.Main` means that the coroutine will be executed confined to the `Main` thread[3].

  By default, when a coroutine is created, it is used the `Dispatchers.Default` that uses *worker* threads: a shared pool of threads on the `JVM` in which coroutines can execute.

- CoroutineContext:
  Each coroutine in `Kotlin` has a *context* that is *immutable*. A context is simply a set of *elements* that realizes the concept of *context* the coroutine executes within. The main elements in a context are:

  – the `Job` that represents the coroutine;
  – the `CoroutineDispatcher` that dispatches the execution of coroutine over the threads;

[3]In this case, the coroutine can update the `UI`. There are also dispatchers for `JavaFX` or `Swing` for `Kotlin JVM` to force coroutines to be executed on the thread that can update the user interface.

- the `CoroutineName` that is the name associated to the coroutine (useful for debugging);
- the `CoroutineExceptionHandler` that is an handler for all the exception thrown during the execution of the coroutine;
- the `ContinuationInterceptor` that allows to define *how* the coroutine should continue after a resume (a sort of *callback* that is invoked on coroutine resume).

Notice that **`CoroutineContext` is immutable, but it is possible to add elements using the plus operator** that produces a new context instance[4]. In addition, **all of these elements extends `CoroutineContext`** itself, so using the `plus` operator lets to easily create a context that is a *join* of others. For example:

```
val newContext = CoroutineName("MyCoroutine") + Dispatchers.Main
```

creates a new context named *MyCoroutine* in which coroutine will be executed using `Disparchers.Main`.

A context can be passed to the coroutine builder before launching coroutines but, if the context has to be switched while the coroutine is running, there is the special suspend function withContext. `Kotlin` has also a default context for builders: `EmptyCoroutineContext`. It can also be used with the `plus` operator to create new contexts.

- CoroutineScope:
  Each coroutine in `Kotlin` must have a *scope* that delimits the lifetime of the coroutine. The `CoroutineScope` consists in only one property: `coroutineContext`, an instance of `CoroutineContext`. In additio, `CoroutineScope` has also some *extension functions* such as `launch` that is a builder for coroutines.

  When `launch` is invoked using a `CoroutineScope`, a new coroutine is launched and its context is *inherited* from those of the scope. In this way, all the elements of the parents and its cancellation are propagated to the children; then, if a scope is cancelled, all the coroutine launched starting from it will be cancelled.

In `Kotlin` **the concept of *coroutine* can be summarized by the formula**:
$$\textit{Coroutine} = \texttt{CoroutineContext} + \texttt{Job}$$

In order to launch a coroutine, the developer has to:

1. create an instance of `CoroutineScope`, for example using the `runBlocking` scope builder;
2. call a coroutine builder starting from the created scope, such as `launch`, that returns the `Job` associated to the coroutine.

Here there is an example of the creation of a simple coroutine taken from the official documentation on kotlinlang.org:

```
fun main() = runBlocking { // this: CoroutineScope
 launch { // launch a new coroutine and continue
  delay(1000L) // non-blocking delay for 1 second
  println("World!") // print after delay
 }
```

[4] This way to *compute by composition* comes from the functional programming.

```
  println("Hello") // main coroutine continues while a previous one is delayed
}
```

that produces this result on the console:

```
Hello
World!
```

To fully understand this snippet, the reader should know something about *higher-order functions* and *receivers* which are concepts that come from the *functional programming* available in `Kotlin` .

Notice that `runBlocking` has also an optional argument `CoroutineContext` that can be used to pass elements that will be added to the context of the scope. All of these elements are inherited by the children except for the `Job` that is created by the coroutine builder instead.

For example:

```
runBlocking(CoroutineName("MyCoroutine")) {
  val parentScope = this
  println("parent  : $coroutineContext")
  val job1 = launch {
   println("launch1 : $coroutineContext," +
   " childScope == parentScope : ${this == parentScope}")
  }
  val job2 = launch {
   println("launch2 : $coroutineContext, " +
   "childScope == parentScope : ${this == parentScope}")
  }
  joinAll(job1, job2)
}
```

produces an output similar to:

```
parent  : [CoroutineName(MyCoroutine),
          BlockingCoroutine{Active}@68f7aae2, BlockingEventLoop@4f47d241]
launch1 : [CoroutineName(MyCoroutine),
          StandaloneCoroutine{Active}@d70c109, BlockingEventLoop@4f47d241],
          childScope == parentScope : false
launch2 : [CoroutineName(MyCoroutine),
          StandaloneCoroutine{Active}@1bc6a36e, BlockingEventLoop@4f47d241],
          childScope == parentScope : false
```

As you can see, both child scopes are different from parent even if they are related: indeed, cancelling the parent scope means to cancel those of the children, but the reverse is not true. About the context, it's clear that children contexts are completely inherited from the parent except for the `Job` instances[5] that are different.

[5] `BlockingCoroutine` and `StandaloneCoroutine` are `Job` extensions.

2.1.2 Synchronization between coroutines in `Kotlin`

We highlight that **coroutines in `Kotlin` can use shared memory or *messages*** to synchronize themselves. In particular:

- The package `kotlinx.coroutines.sync` exposes the classical tools for synchronization in a shared memory environment (*mutex* and *semaphore*).

  Notice that this type of synchronization is very basic if compared with the standard `Java` tools for concurrency, such as `Lock` and `Condition`; at this moment, `Kotlin` does not define any mechanism similar to `Java` condition, but, however, it's very easy to implement it (for example, we have an implementation made by the author called `CoroutineCondition` that uses the `Continuation` object of a coroutine).

  [See `MutexPiCalculation.kt` for a basic example]

- The package `kotlinx.coroutines.channels` exposes modern tools for synchronization with *message-passing* in a non-shared memory environment (*channels*). The main entity of this package is `Channel`, that is very close to `BlockingQueue`, but with suspending operations instead of the `Java` blocking methods.

  Two coroutines can use a channel in order to transfer a single value that comes from the *producer* (the coroutine that invokes the `send` operation) and is transferred to the *consumer* (the coroutine that invokes the `receive` operation); originally, in `Kotlin` channels were **bidirectional** and **symmetric** (one-to-one), but in the last updates of the language it is possible to have *asymmetric* behavior thanks to *Fan-Out* and *Fan-In* mechanisms. Nevertheless, `Kotlin` has more sophisticated tools to make multiple coroutines able to have an asymmetric communication and we will see them below.

  The semantic of *send/receive* operations depends on the nature of the channel that is determinated by its capacity, but the communication can be **synchronous** or **asynchronous** with also some little variations of these (for example, **rendez-vous**). Notice that, as anticipated, a channel can be safely shared between coroutines, but the developer has to pay attention because the `receive` operation can quickly lead to competition problems if invoked from two or more coroutines in parallel.

  [See `ChannelPiCalculation.kt` for a basic example]

- The package `kotlinx.coroutines.flow` exposes the tools for using *flows*, defined by the documentation as *asynchronous cold stream of elements* that can safely be used to synchronize multiple coroutine at the same time.

  Flow can be more formally defined as **mono-directional, one-to-many and asynchronous** channels with the possibility to be **buffered** for replay strategies. In the latest versions of `Kotlin` , flows replaced `BroadcastChannel`.

2.1.3 Suspending functions

Suspending functions are normal `Kotlin` methods but with a special feature: **they can *suspend* the execution of a coroutine** they run within. The main example of suspending function it's `delay` that suspends the execution of the coroutine which calls it for a specified time.

Let's make an example (`SuspendingFunctionExample.kt`):

```kotlin
suspend fun sleep(who : String, timeMillis : Long) {
 println("$who: I'm going to sleep for $timeMillis milliseconds...")
 delay(timeMillis)
 println("$who: Good morning, I wake up!")
}

suspend fun pollAlive(who : String, pollingTime : Long) {
 while (true) {
  delay(pollingTime)
  println("$who: i'm alive [thread=${Thread.currentThread()}]")
 }
}

suspend fun sayHello(who : String) {
 println("$who : Hello... I'm a coroutine " +
   "[thread=${Thread.currentThread()}]")
 println("$who : My context: $coroutineContext")
}

fun main() {
 @OptIn(DelicateCoroutinesApi::class)
 val ctx = newSingleThreadContext("CoroutineSingleThread")

 runBlocking(ctx) {
  println("parent: [thread=${Thread.currentThread()}]")
  val job1 = launch {
   val who = "job1"
   sayHello(who)
   sleep(who, 3000)
   sayHello(who)
  }
  val job2 = launch {
   val who = "job2"
   sayHello(who)
   pollAlive("job2", 500)
   sayHello(who)
  }
  job1.join()
  println("parent: job1 = $job1, job2 = $job2")
  job2.cancelAndJoin()
  println("parent: job1 = $job1, job2 = $job2")
 }
}
```

it produces:

```
parent: [thread=Thread[CoroutineSingleThread,5,main]]
job1 : Hello... I'm a coroutine [
                thread=Thread[CoroutineSingleThread,5,main]]
job1 : My context: [StandaloneCoroutine{Active}@739c17c3,
                java.util.concurrent.ScheduledThreadPoolExecutor@4289a013
                [Running, pool size = 1, active threads = 1,
                        queued tasks = 1, completed tasks = 1]]
```

```
job1: I'm going to sleep for 3000 milliseconds...
job2 : Hello... I'm a coroutine [
                thread=Thread[CoroutineSingleThread,5,main]]
job2 : My context: [StandaloneCoroutine{Active}@7ce21fc2,
                java.util.concurrent.ScheduledThreadPoolExecutor@4289a013
                [Running, pool size = 1, active threads = 1,
                        queued tasks = 1, completed tasks = 2]]
job2: i'm alive [thread=Thread[CoroutineSingleThread,5,main]]
job2: i'm alive [thread=Thread[CoroutineSingleThread,5,main]]
job2: i'm alive [thread=Thread[CoroutineSingleThread,5,main]]
job2: i'm alive [thread=Thread[CoroutineSingleThread,5,main]]
job2: i'm alive [thread=Thread[CoroutineSingleThread,5,main]]
job1: Good morning, I wake up!
job1 : Hello... I'm a coroutine [
                thread=Thread[CoroutineSingleThread,5,main]]
job1 : My context: [StandaloneCoroutine{Active}@739c17c3,
                java.util.concurrent.ScheduledThreadPoolExecutor@4289a013
                [Running, pool size = 1, active threads = 1,
                        queued tasks = 1, completed tasks = 8]]
parent: job1 = StandaloneCoroutine{Completed}@739c17c3,
                job2 = StandaloneCoroutine{Active}@7ce21fc2
parent: job1 = StandaloneCoroutine{Completed}@739c17c3,
                job2 = StandaloneCoroutine{Cancelled}@7ce21fc2
```

In this significant example we used the `newSingleThreadContext` to create a context with one single thread T_x dedicated for the execution of the coroutine: each coroutine that inherits this context executes on T_x.

The example shows some important characteristic of `Kotlin` coroutines and suspending function:

1. When `job1` calls the *suspend* function `sleep(who, 3000)` and encounters the `delay` instruction at the line 3, coroutine goes into suspension for 3 seconds but, once resumed, the execution restarts exactly by the end of `delay` at line 3;

2. Since we forced a single thread for the two coroutines, this snippet shows that even if `job1` is suspended on the `delay` (line 3), the thread is however active (not paused or suspended) and it continues to run the `job2`. This is shown by all the *alive* prints of `job2` in the resulting command windows.

3. The instruction `cancelAndJoin()` let the developer easy to cancel a coroutine, waiting for its end.

To conclude, a **suspending function** is a `Kotlin` **method able to suspend the coroutine it is running within, without blocking the executing thread**. From the implementation point of view, a suspending function is a normal method with an *hidden* parameter of type `Continuation` that is automatically added when the code is compiled. The implementation of this class is built-in provided and is used by coroutines to save their state before a suspension point.

To understand better, suppose to compile the example `SuspendingFunctionExample.kt` shown before on a desktop machine with JVM. After the normal Kotlin compilation we will have the executable SuspendingFunctionExampleKt.class, and if we decompile[6], we will see that all the suspending functions have this Java signature:

```java
public static final Object sleep(@NotNull String who, long timeMillis,
 @NotNull Continuation<? super Unit> paramContinuation)

public static final Object pollAlive(@NotNull String who,
 long pollingTime,
 @NotNull Continuation<? super Unit> paramContinuation)

public static final Object sayHello(@NotNull String who,
 @NotNull Continuation $completion)
```

So, a suspending function is simply compiled into a Java method with a Continuation as last parameter that can be used to suspend the coroutine that calls the function.

Finally, we conclude the Kotlin concurrency overview by informing about the possibility to wait for multiple channels to receive data thanks to the select statement.

2.2 Go concurrency overview

As said in the official page, Go has a *built-in concurrency and a robust standard library* which is one of the central features of the language.

Go is designed to make concurrency easier, but this simplicity limits the functionalities offered by the concurrency framework: indeed, Go 's coroutines are not structured as the Kotlin implementation.

There is no Job, no Scope or Context. The main entities of the concurrency in Go are:

- Goroutines
  They are *lightweight* thread directly managed by the Go runtime, with low memory overhead and fast creation. Goroutines are quickly launched by the keyword go and their code is simply specified as a normal function:

```go
func asyncTask() {
 fmt.Println("Running asynchronously!")
}

func main() {
 go asyncTask()
}
```

- Channels
  Built-in data structures that realize the communication between goroutines, through which it is possibile to **send** and **receive** values with the channel operator <-. The make instruction allows to easily create a channel:

```go
channel := make(chan int, 10) // nCreate a channel
channel <- 100 // Send the value 100
receivedValue := <- channel //Receive a value
close(channel) // Close the channel
```

[6] For example using *JD-GUI*.

They are **bidirection** but they can be anyway declared in three ways: *bidirectional* (`chan T`), *receive-only* (`<-chan T`) or *send-only* (`chan<- T`).

Based on their capacity, they can be **buffered** or **unbuffered**.

```
  unbufferedChannel := make(chan int)
  bufferedChannel := make(chan int, 10)
```

As in `Kotlin` , `Go` 's channel also support *Fan-In* and *Fan-Out*:

```
fun worker1(channel chan<- int) {
  channel <- 1
}

fun worker2(channel chan<- int) {
  channel <- 2
}

fun consumer(channel <-chan int, ack chan<- bool) {
  for value := range channel { // listen until channel is not closed
    fmt.Println("received value: " + value)
    ack <- true
  }
}

ack := make(chan int, 1)
channel := make(chan int, 10)
go worker1(channel)
go worker2(channel)

for i := 0; i < 2; i++ {
  fmt.Println("ack " + i)
}
close(channel)
close(ack)
```

Finally, `Go` also allows to listen multiple channels thanks to the `select` statements.

2.3 Comparison between the concurrency frameworks of `Kotlin` and `Go`

As shown, `Go` concurrency is really easy and efficient but less structured if compared with the `Kotlin` 's one. The following table highlight the key differences and similarities:

Feature	Kotlin Coroutines/Channels	Go Goroutines/Channels
Concurrency Model	Structured concurrency with coroutines	Lightweight threads with goroutines (OS bound)
Channel Type	Channels for communication between coroutines	Channels for communication between goroutines
Syntax	Uses `suspend` functions and builders like `launch` and `async` (needs to create a context)	Uses `go` keyword to spawn goroutines
Error Handling	Structured error handling with coroutine scopes	Error handling is done manually, often with `defer`, `panic`, and `recover`
Cancellation	Built-in cancellation support with coroutine scopes	Context package used for cancellation
Performance	Efficient with cooperative multitasking	Efficient with preemptive multitasking
Tooling	Supported by *IntelliJ IDEA* with *coroutine debugging tools*	Supported by *Go tools* with *goroutine profiling*

3 Performance test

3.1 Parallel matrix multiplication algorithms

We will use the matrix multiplication as algorithm to test the performance of `Kotlin` and `Go` coroutine implementations. There will be three different versions of the algorithm:

- **FAN**:
  Uses 2 channels: one to distribute the cells to be computed (`taskChannel`), the other to collect the partial results (`resultChannel`); thanks to the `fan`, all the workers wait on the same channel and send their result on the other. A coordinator is in charge to open the channels, launch the workers, distribute the work, collect the results and finally close the channels, automatically notifying the end of the computation.

- **COORDINATOR**:
  Requires a coordinator *server* like the previous, but a greater amount of channels: one channel shared by the workers to notify their availability (`requestWorkChannel`), one channel the workers can use to notify their termination (`ackChannel`), one channel to send the partial results (`resultChannel`), and one channel for each worker the coordinator uses to send the cell to be computed (`workerChannels[]`).

  The matrices are still on the shared memory and their reference is passed to the workers. The coordinator has to listen both of the `requestChannel` and the `resultChannel` to react to the availability of a worker or to the computation of a partial result: when all the cells are computed, the coordinator notifies each worker channel, then wait for their termination and returns the final result.

- **PURE**
  This algorithm is the same of the previous, but the matrices are not in the shared memory anymore, so the coordinator send the row and the column the worker has to use to produce the result for the requested cell.

  This is a pure implementation of a message-passing algorithm, without any shared memory but only using channels and messages.

Algorithm	**Shared Memory Usage**	**Number of Channels**
FAN	Input Matrices	2
COORDINATOR	Input Matrices	$3 + N_{WORKERS}$
PURE	None	$3 + N_{WORKERS}$

While `Kotlin` offers a solid *object-oriented* paradigm, `Go` has only a minimal support for it: while for the first language we have one interface in its file and the implementation in their own files, all the `Go` code is enclosed in a single file following the language's conventions:

Language	**Implementations**

`Kotlin`	`MatrixProduct.kt`
	`FanChanneledMatrixProductImpl.kt`
	`CoordinatorChanneledMatrixProductImpl.kt`
	`PureChanneledMatrixProductImpl.kt`

`Go`	`matrix.go`

3.2 Main and launchers for performance analysis

In order to use the algorithm that have just been presented, we need two main application:

Language	**Implementations**
`Kotlin`	`KAposMatrixApp.kt`
`Go`	`main.go`

Both of the `main` functions, accept that series of parameters in order to **perform the computation of the product of two matrices with the specified size, using the given number of coroutines**. In addition, the programs have been implemented with the possibility to store the results in a `csv`, specifying a file to go in append mode.

Notice that:

- the `Kotlin` application must be built using `Gradle`, for example via the task `distZip` that creates a convenient launcher to execute the jar
- the `Go` application has to be compiled using `go build`

Once the executable are created, they can be invoked using the arguments specified in the following table:

Option	Default Value	Description
`-s, -size`	3	Size of the matrices (NxN)
`-c, -coroutines`	4	Number of coroutines to use
`-o, -output`	`false`	Store results in CSV file
`-f, -file`	`null`	CSV filename for storing results
`-r, -repeat`	1	Number of times to repeat the calculation
`-l, -log`	false	Enable detailed logging
`-m, -mode`	`COORDINATOR`	Concurrency mode: `COORDINATOR`, `FAN`, or `PURE`

The argument `-r` allows to repeat the multiplication for the specified number of times with the given parameters: each run is then associated to the same **workspace**. A workspace is a set of execution with the same parameters and it's used in order to perform average logic and enhance the precision of the analysis.

To easily produce the executables and launch them at once with suitable sets of values for the matrix size and the workers, a convenient launcher for `Linux` has been developed at `launcher.sh`.

That launcher executes two kind of analysys:

- **FIXED SIZE**
  The size of the matrices is fixed while the number of the workers is increased execution by execution.
- **FIXED WORKER**
  The number of the workers is fixed but the size of the matrix varies execution by execution.

Each set of $(size, workers)$ is executed for 10 times. This way, then the `csv` is produced, just the average of the execution times of the repetitions of the same set will be considered.

3.3 Native compilation for `Kotlin`

3.3.1 `Kotlin` Native

Kotlin supports **native compilation** realized from the same `Jetbrains` by involving LLVM.

We want to add also `Kotlin Native` in the comparison to analyze the difference between `Kotlin JVM` and `Go` .

Notice that when `Kotlin` is configured to work in the native mode (see `build.gradle.kts`), the set of usable library is reduced and the **Java** standard libraries are not supported anymore (even the `Java IO/NIO`). For this reason, some modifications to the existing `Kotlin` code were needed: you can find here the `kotlin-native` project with the right configuration and a thinner code that allows the compilation.

A convenient launcher as also be developed to launch the native compilation and the execution with the same parameters of the launcher shown in the paragraph above:

launcher_ktntv.sh

3.3.2 Kotlin compiled via GraalVM

Java itself supports native compilation thanks to an external and advanced JVM with innovative technologies that supports a process called *ahead-of-time Native Image compilation*.

We will not go deep into the details of this project, but we are going to use it to test this other way to compile native Kotlin code. Notice that, differently from the previous solution, this one is not developed from Jetbrains, but from an external community of developers. Indeed, GraalVM was initially developed for Java but, since Kotlin compiles itself Java, it is also compatible without problems.

The configuration of GraalVM's compilation does not involve complex processes than modifying the build.gradle with the proper plugin conveniently configured.

The setup is carefully explained at this page but, as done for the others, we developed a launcher also for the GraalVM version at launcher_graal.sh.

3.4 Overview of the considered versions and plotting

Based on what has been presented, the following table summarizes all the algorithms and the version of the languages we are going to use to perform the analysis:

	COORDINATOR	**FAN**	**PURE**
Kotlin	kt_coordinator	kt_fan	kt_pure
Kotlin Native	kt_ntv_coordinator	kt_ntv_fan	kt_ntv_pure
Kotlin Graal	kt_graal_coordinator	kt_graal_fan	kt_graal_pure
Go	go_coordinator	go_fan	go_pure

The results of the executions will be placed into proper csv:

- fixed_size_sel.csv for the results of the *fixed size* analysis
- fixed_workers_sel.csv for the results of the *fixed workers* analysis

Another component server has been developed in order to accept the csv files and produce the graph we are going to show.

Again, we developed a script to easily call the server to produce the graphs: graphs.sh

3.5 All-in-one graphs

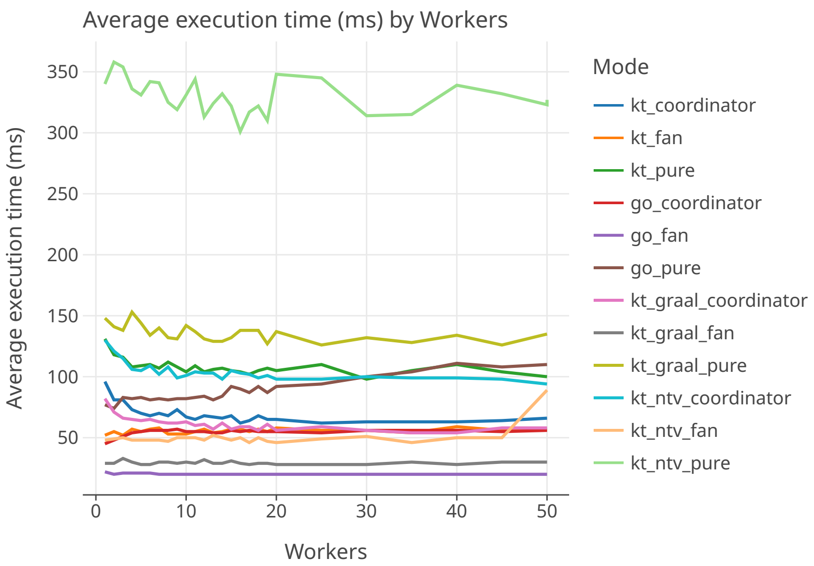

Figure 2: Fixed Size Analysis

The graph clearly shows a huge variety of performance between the different variants. We can see that `Go FAN` has the best performance, followed by `Kotlin` Graal at the same variant. Without any doubt, the worse is `Kotlin` Native in pure mode, meaning that pure message-passing communication could not be optimized in the native compiled sources.

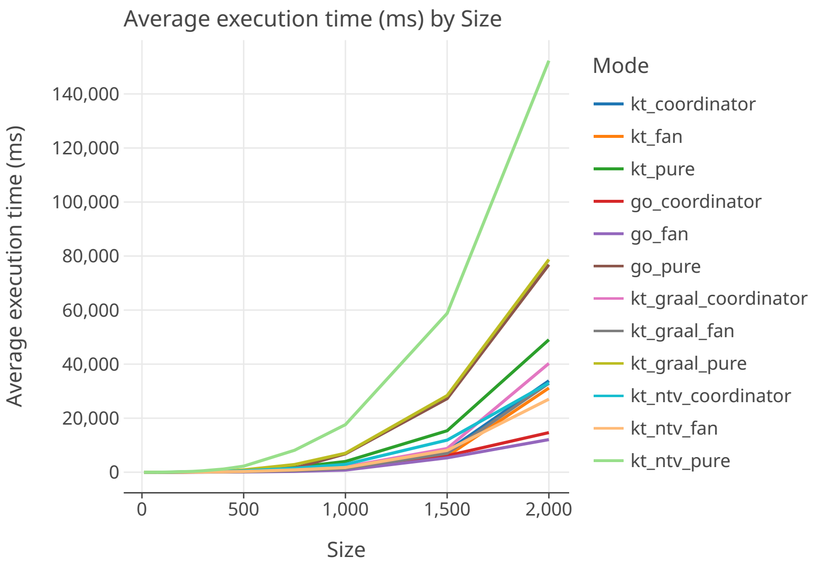

Figure 3: Fixed Worker Analysis

Also this graph seems to confirm the impressions above, with `Kotlin` Native having the worst performance. Anyway, `Kotlin` Native in the `FAN` mode performs slightly well, but `Go` confirms to be less time-consuming both in the `FAN` and `COORDINATOR` modes. Anyway, the `PURE` modes have the worst times while the `FAN` have the bests.

3.6 Graphs by modes

3.6.1 Fixed size

Figure 4: Fixed Size Analysis `FAN`

`Go` and `Kotlin Graal` still are the bests while `Kotlin` Native initially seems to be winning on `JVM`, but it definitively looses with a huge number of workers. In addition, `Go` seems to resist really well to the increase of the workers, keeping the execution time almost constant and demonstrating a huge ability to handle a significant amount of concurrent units.

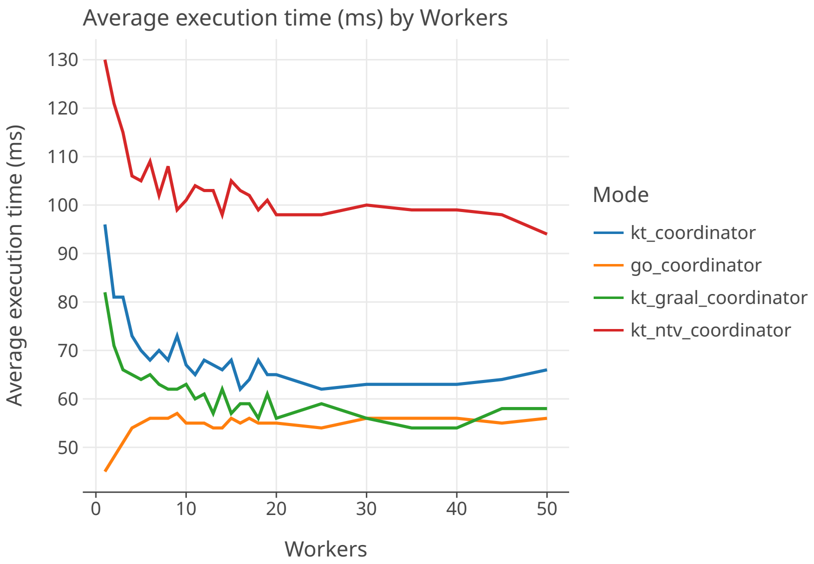

Figure 5: Fixed Size Analysis `COORDINATOR`

`Go` still confirms as the one that takes less time while `Kotlin` Native keeps being the worst. `GraalVM` demonstrate to be a competitive solution.

Figure 6: Fixed Size Analysis `PURE`

We can see that `Kotlin` native incredibly suffers with the pure variant, showing that the native compilation is not optimized with channels communication. Instead, the `JVM` seems to have a good optimization of the message-passing technology.

3.6.2 Fixed workers

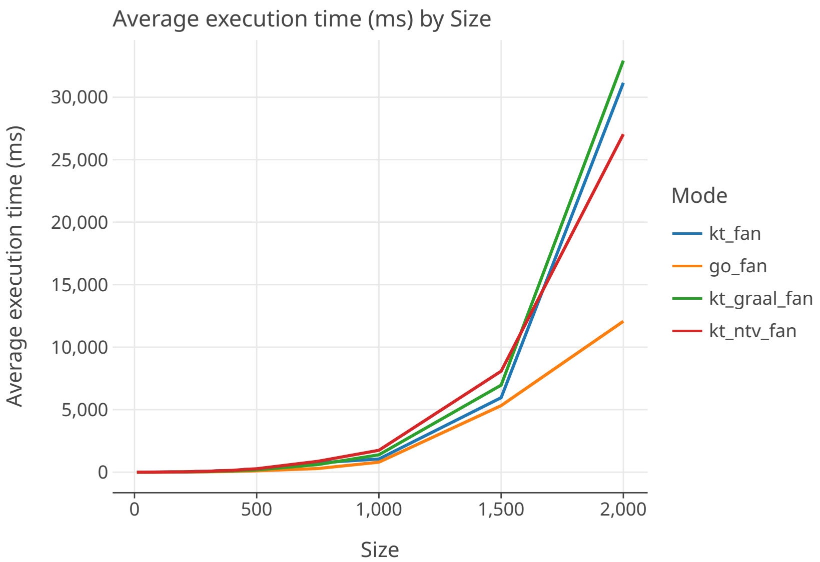

Figure 7: Fixed Workers Analysis `FAN`

`Go` keeps to be the one with the minimum elapsed times, while `Graal` shows some worsening especially with the increasing of the size.

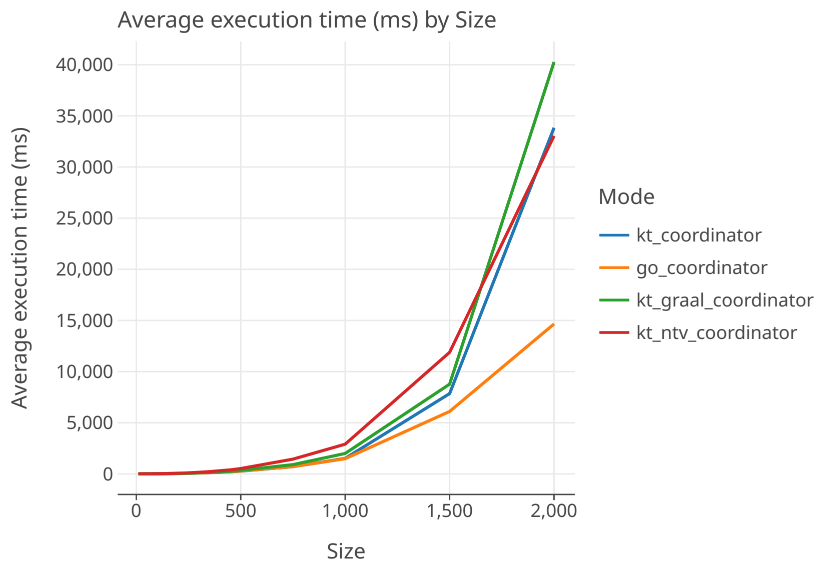

Figure 8: Fixed Workers Analysis `COORDINATOR`

The graphs is really similar to the previous one, with `Go` as the best, `Graal` that suffers when the size increases while `Kotlin` Native seems to recover some points at the end.

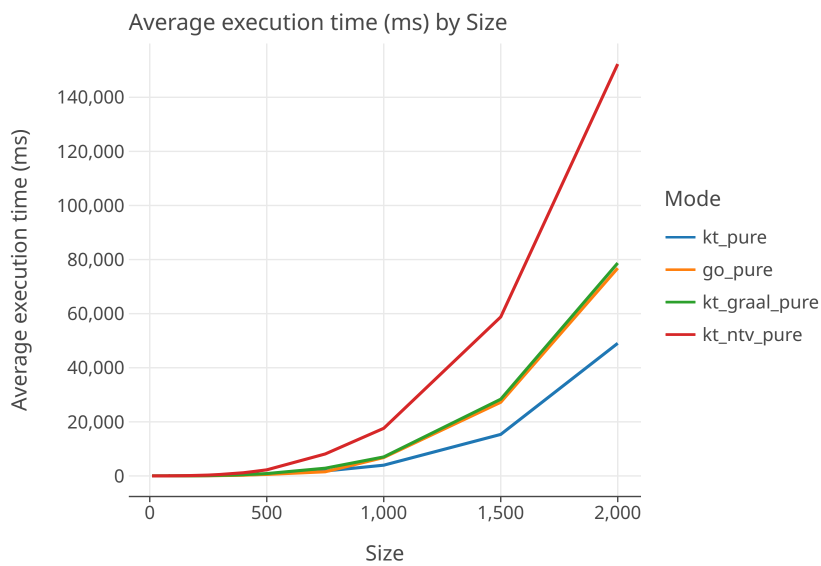

Figure 9: Fixed Workers Analysis `PURE`

This graph shows some improvements for the `JVM` while `Graal` and `Go` behave really similar. `Kotlin` Native keeps to be the worst.

3.7 The behavior of the `JVM` with multiple executions

One of the reason why the coroutines have success on the `JVM`, is because they let to pay the cost of the Thread's creation only once. Then, since the Threads remains active and the coroutines are scheduled across them, the `JVM` does not have to loose time managin activation or de-activation of the them, saving lots of time.

This can be seen zooming into one of the `csv`, focusing on `Kotlin JVM`:

```
workspaceId,size,coroutine,mode,timeMillis
612f53a2-3b8b-4e39-84ee-0ee64bf1c89e,250,1,kt_coordinator,272
612f53a2-3b8b-4e39-84ee-0ee64bf1c89e,250,1,kt_coordinator,102
612f53a2-3b8b-4e39-84ee-0ee64bf1c89e,250,1,kt_coordinator,96
612f53a2-3b8b-4e39-84ee-0ee64bf1c89e,250,1,kt_coordinator,89
612f53a2-3b8b-4e39-84ee-0ee64bf1c89e,250,1,kt_coordinator,86
```

```
612f53a2-3b8b-4e39-84ee-0ee64bf1c89e,250,1,kt_coordinator,80
612f53a2-3b8b-4e39-84ee-0ee64bf1c89e,250,1,kt_coordinator,61
612f53a2-3b8b-4e39-84ee-0ee64bf1c89e,250,1,kt_coordinator,61
612f53a2-3b8b-4e39-84ee-0ee64bf1c89e,250,1,kt_coordinator,61
612f53a2-3b8b-4e39-84ee-0ee64bf1c89e,250,1,kt_coordinator,61
```

As we can see, the time of the first execution is almost 4.45 times the last one. This is not happening on the compiled versions of `Go` and `Kotlin` Native, while the `Graal` show a significant improvement:

```
workspaceId,size,coroutine,mode,timeMillis
324fbed9-a3a9-43af-b9be-add5ae409379,250,1,go_coordinator,48
324fbed9-a3a9-43af-b9be-add5ae409379,250,1,go_coordinator,42
324fbed9-a3a9-43af-b9be-add5ae409379,250,1,go_coordinator,48
324fbed9-a3a9-43af-b9be-add5ae409379,250,1,go_coordinator,49
324fbed9-a3a9-43af-b9be-add5ae409379,250,1,go_coordinator,47
324fbed9-a3a9-43af-b9be-add5ae409379,250,1,go_coordinator,45
324fbed9-a3a9-43af-b9be-add5ae409379,250,1,go_coordinator,44
324fbed9-a3a9-43af-b9be-add5ae409379,250,1,go_coordinator,49
324fbed9-a3a9-43af-b9be-add5ae409379,250,1,go_coordinator,42
324fbed9-a3a9-43af-b9be-add5ae409379,250,1,go_coordinator,42
ce8edcc7-1d94-427b-9d55-af4dbdd1d515,250,1,kt_graal_coordinator,132
ce8edcc7-1d94-427b-9d55-af4dbdd1d515,250,1,kt_graal_coordinator,74
ce8edcc7-1d94-427b-9d55-af4dbdd1d515,250,1,kt_graal_coordinator,76
ce8edcc7-1d94-427b-9d55-af4dbdd1d515,250,1,kt_graal_coordinator,76
ce8edcc7-1d94-427b-9d55-af4dbdd1d515,250,1,kt_graal_coordinator,76
ce8edcc7-1d94-427b-9d55-af4dbdd1d515,250,1,kt_graal_coordinator,81
ce8edcc7-1d94-427b-9d55-af4dbdd1d515,250,1,kt_graal_coordinator,76
ce8edcc7-1d94-427b-9d55-af4dbdd1d515,250,1,kt_graal_coordinator,76
ce8edcc7-1d94-427b-9d55-af4dbdd1d515,250,1,kt_graal_coordinator,81
ce8edcc7-1d94-427b-9d55-af4dbdd1d515,250,1,kt_graal_coordinator,77
727f-3326-faf8-efb0,250,1,kt_ntv_coordinator,132
727f-3326-faf8-efb0,250,1,kt_ntv_coordinator,130
727f-3326-faf8-efb0,250,1,kt_ntv_coordinator,169
727f-3326-faf8-efb0,250,1,kt_ntv_coordinator,123
727f-3326-faf8-efb0,250,1,kt_ntv_coordinator,124
727f-3326-faf8-efb0,250,1,kt_ntv_coordinator,125
727f-3326-faf8-efb0,250,1,kt_ntv_coordinator,126
727f-3326-faf8-efb0,250,1,kt_ntv_coordinator,128
727f-3326-faf8-efb0,250,1,kt_ntv_coordinator,126
727f-3326-faf8-efb0,250,1,kt_ntv_coordinator,125
```

4 Conclusion

We compared three different matrix multiplication algorithms with different levels of coroutine communication: from `FAN`, that has the lower number of messages, to `PURE` that stressed the communication by sending all the parts of the matrices to the workers via channels. `COORDINATOR` is in the middle, with matrices on the shared memory and smaller messages.

We also used two different languages: `Go` , that has lightweight concurrent units mapped on the OS threads and managed by the `Go` runtime, and `Kotlin` in its three different versions `JVM`, Native and `Graal`.

`Go` seems to be <u>less time consuming</u> than all of the other in every scenario, showing how its effectively designed for efficiency. It manages coroutines without problems, handling also a large amount of workers keeping the time almost constant.

`Kotlin` *JVM* shows that `JVM` is able to perform some optimizations on the communication when the channels are stressed, but actually shows some lacks during its "warm-up" phase, in which has to perform lots of operations and to create the threads the coroutines will be dispatched within.

`Kotlin Graal` provides an excellent compromise between the efficiency of `Go` and the rich functionality of `Kotlin` . Unlike `Kotlin` Native, `GraalVM` does not impose significant limitations on Kotlin's dynamic features, such as reflection and runtime libraries, thanks to its compatibility with the `JVM` ecosystem and its advanced ahead-of-time (AOT) compilation capabilities.

`Kotlin` Native, instead, had the worst performance in almost every situation. Moreover, it significantly limits the feature offered by `Kotlin` , allowing the developer to use only built-in `Kotlin` libraries. `Java` libraries are denied.

All the code can be found in the GitHub repository at:
`Activity-Project-Operating-Systems-M`