Final Project

Pi with Monte Carlo Method, Threads and Coroutines

Tecnológico de Monterrey

Professor:

Benjamín Valdés Aguirre

May 29th, 2020

Parallelization using Kotlin and Android

Carlos Miguel Negrete Barrientos A01208783

Table of Contents

Abstract

It is known that day by day manufacturers around the world are designing more and more powerful components for our computers. In addition, it can be seen that technological devices are part of our daily lives, it is not something only for experts or science anymore. Actually we have in our pockets really powerful personal computers, our mobile phones are not ordinary anymore. The physical components, known as hardware, of our devises are progressing, so there internal and logical way of using these pieces, also known as software, of computational power needs to evolve to. With the time some options for the hardware manipulation appeared. In this project it is explored one of the most basic, but very important way of programming; the parallelization. This way of thinking may be helpful in order to make some computationally expensive processes more efficient and quicker. Something called threads are in almost every application, even mobile phones use them. Threads are the basic tool for understanding parallelization. But, it is not the only way of dividing tasks. This project will focus on an alternative to regular threads; coroutines is the selected one. It was mentioned that mobile phones are portable computers and they have an Operative System, so they can handle threads and coroutines. Therefore, the exercise was tested on an Android device with Kotlin as languaje. This time the Monte Carlo Method was selected for the project demonstration.

1. Theorical Background

1.1 Parallelization

Some processes may be very expensive for actual computers, it does not matter how powerful they are. Actually, the final users of the different software out in the market want really quickly applications, they do not want to wait too much time in just waiting for something. Parallelization is the solution, this way of thinking when programming in union with the technical specifications of modern devices help programmers to divide huge and expensive processes in subtasks. Actually there are two different ways of approaching a situation and it will depend on the problem itself.

The first concept is named Data Parallelization. This concept consists in dividing the data of an iterative calculations in different processes.[1] For example, in a homework you need to solve 100 sums. Probably it is an easy task, but you will take some time. It would be quicker if a friend do 50 and you the other 50. It would be quicker if there are two, or three or even more friends. Since the exercises are the same, but with different values you can divide the homework only considering the data. In this analogy, friends represent a thread and each sum represent a portion of the data of the complete task.

The second concept is known as Task Parallelization. This idea is kind of similar to the previous one, but this applies with more complex and laborious tasks. Taking in consideration another homework as an example. This time it is not a simple homework, it is a huge project. For the project some materials must be bought. After that, a preliminary report must be done. Consequently the experiment is part of the protect and someone must take note of the results. At the end the final report must be

1 https://www.mcs.anl.gov/~itf/dbpp/text/node83.html
https://www.tutorialspoint.com/data-parallelism-vs-task-parallelism

printed and handed in. The entire task is the project, but it is divided in subtasks. There are some subtasks that can be done at the same time if there are more than one person. This time task parallelization is being used.

1.2 Threads and Coroutines

Threads can be explained for easy purposes just as the previously exposed examples. Something similar to a set of friends doing a particular homework or project. However, it is important to mention that threads are not necessarily related with computer processors. Threads is a simple way of dividing a task and share the same information with each other since they can be executed simultaneously in a single processor.

Now threads are more clear, but this work is not entirely focused on threads. This time an alternative called coroutines is being explored. However, it is important to mention that coroutines are supported in some programming languages. Int his case Kotlin is the selected one.

Coroutines are something similar to threads, but with some significant differences. At the beginning, it can be said that they can handle different subtasks asynchronously. In addition, coroutines runs in a single thread. Inside this thread there are a certain amount of coroutines executing different tasks. This alternative is something that some people prefer over threads, mainly because each thread require some resources from the device, including memory and cpu. Since coroutines are being executed in a single thread the required resources are minimum compared with threads. In addition, there is a limitation for the amount of created threads in each device. Meanwhile, coroutines does not have the same limitation because of their resources requirements.

Moreover, the most important concept in coroutines is that the programmer can decide when each coroutine may stop its process in order to wait for the other ones.

This is very helpful, since it usually makes easier to coordinate or synchronize different subtasks.

1.3 Getting Pi with Monte Carlo Method

There are several ways to estimate the value of Pi. However, some of the methods may be really expensive for some computers and the time taken by the computer can be large. One of the most known method is by using the Monte Carlo method. It is the perfect task for a test in an Android device since it is really simple to implement and it can be really expensive and the calculation can be really slow depending on the number of iterations.

First of all, how does Monte Carlo method works? In a graphical way, there is a circle inside a square. As it can be seen in Fig. 1.

Circle Inside a Square for Monte Carlo Method

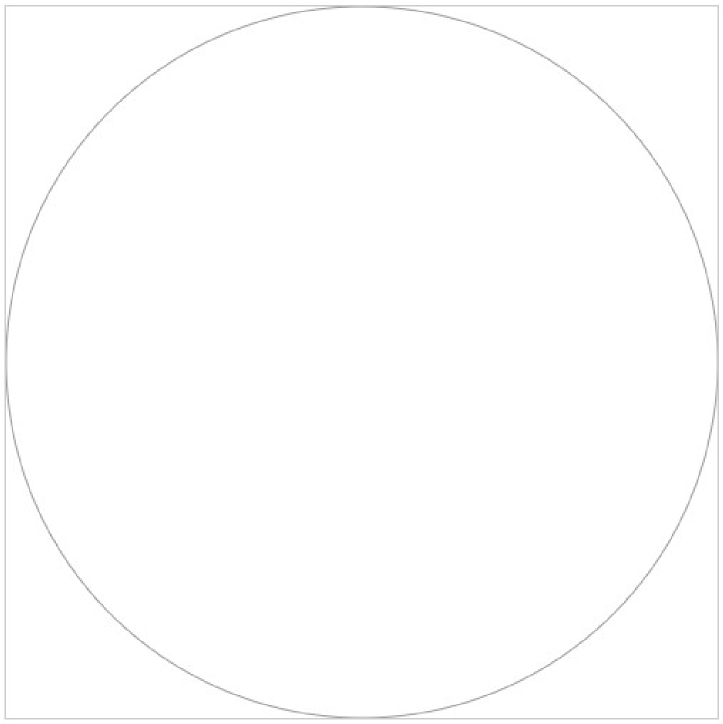


Fig. 1. Empty circle for Monte Carlo Method

Somehow, some points are being throwed randomly on the canvas where the circle and the square are drawn. As it can be seen in Fig. 2.

Circle Inside a Square with Random Points for Monte Carlo Method

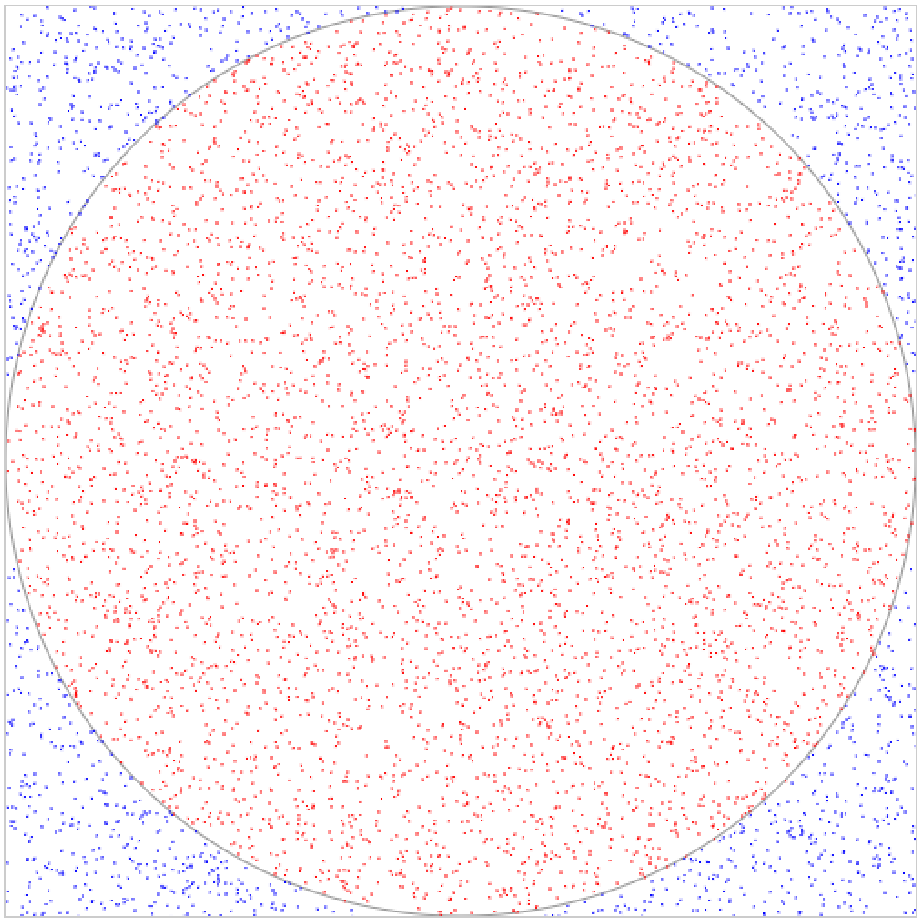


Fig. 2. Random dots in the canvas for Monte Carlo Method

There is still missing the mathematical part of the calculation. Usually this method is approached with a circle of radius of 0.5 units. Therefor, taking in consideration the formula for getting the area of a circle, which is $\pi r^2 = area$

Since there are a square, its area is also important: $(2r)^2 = area$

If the area of the circle is divided by the area of the square the final formula is $\frac{\pi}{4}$

Therefor the final formula to estimate pi is $\pi = 4 * number\ of\ points inseide\ the ¿ / _{total\ points} ¿$

The amount of total random drawn points is the same as the number of iterations done by the program.

2. Implementation

2.1 Threads

First of all, it is important to know how threads work with kotlin on an Android device. It is really similar to the way it works on Java. And it is important to consider everything that is different from using a program on a PC in CLI and on an Android device.

At this moment the user must type the number of repetitions or the number of dots that will be randomly generated on the imaginary canvas.

Input Area for the User

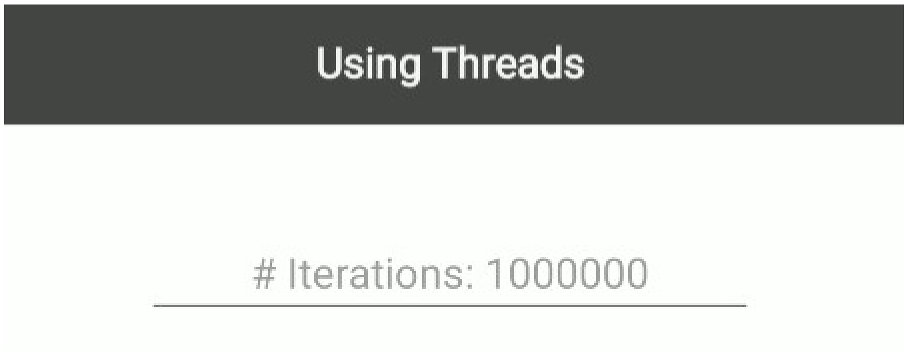
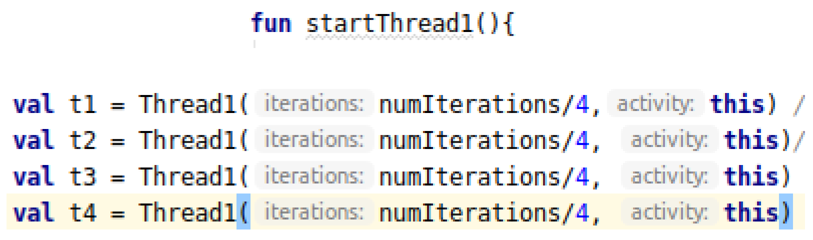


Fig. 3. The user must enter a number of iterations

Always remember to validate that the field is not empty.

As in Java, in Kotlin it is necessary to create a function that will be in charge of the creation of the threads. At this moment this is how some threads are created just by a static amount of them. Just 4 for the first tests.

Function and Creation of the Threads

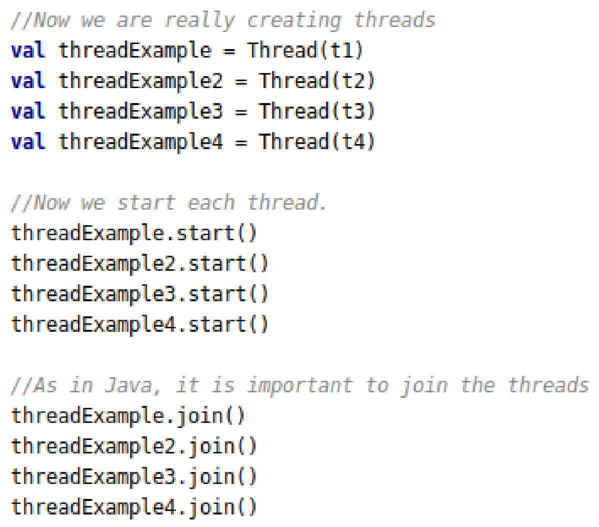


```
                fun startThread1(){

val t1 = Thread1( iterations: numIterations/4, activity: this) /
val t2 = Thread1( iterations: numIterations/4,  activity: this)/
val t3 = Thread1( iterations: numIterations/4,  activity: this)
val t4 = Thread1( iterations: numIterations/4,  activity: this)
```

Fig. 4. Using the class of the threads for the creation of the thread

After that, there is the proper creation of the threads. In addition, it is important to start them all and join them, as in Java.

Function and Creation of the Threads

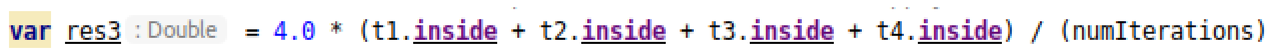


```
//Now we are really creating threads
val threadExample = Thread(t1)
val threadExample2 = Thread(t2)
val threadExample3 = Thread(t3)
val threadExample4 = Thread(t4)

//Now we start each thread.
threadExample.start()
threadExample2.start()
threadExample3.start()
threadExample4.start()

//As in Java, it is important to join the threads
threadExample.join()
threadExample2.join()
threadExample3.join()
threadExample4.join()
```

Fig. 5. Creation of the threads

Inside the same function there appears the final calculation for Pi, using the formula. In order to get access to the total amount of points inside the circle, it most be done by using the object t1, t2, t3, etc...

Final Calculation for Pi Using the Values of each Thread



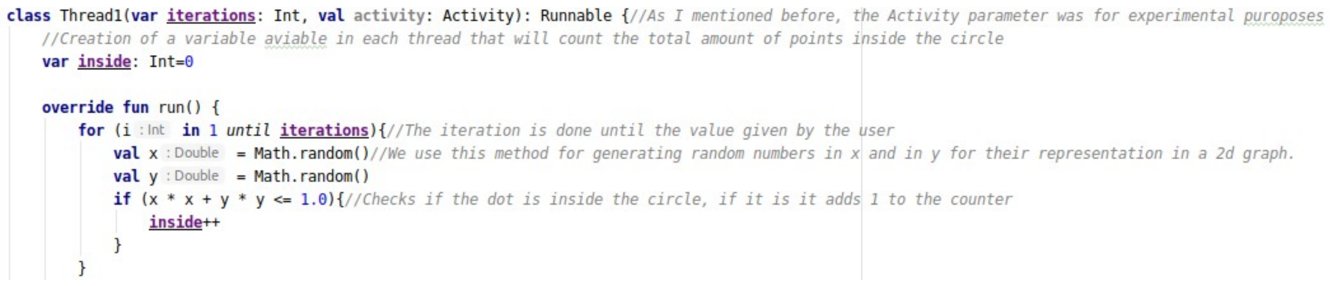
```
var res3 : Double  = 4.0 * (t1.inside + t2.inside + t3.inside + t4.inside) / (numIterations)
```

Fig. 6. Final calculation of Pi

Inside the function there are also the code for printing in User Interface the results.

Now it is the turn of the thread class, where the action happens. As in Java, the interface Runnable was implemented. Inside the overriden function is the code for generating random pints and the calculation of the points inside the circle.

Class for the threads



```
class Thread1(var iterations: Int, val activity: Activity): Runnable {//As I mentioned before, the Activity parameter was for experimental puroposes
    //Creation of a variable aviable in each thread that will count the total amount of points inside the circle
    var inside: Int=0

    override fun run() {
        for (i :Int  in 1 until iterations){//The iteration is done until the value given by the user
            val x :Double  = Math.random()//We use this method for generating random numbers in x and in y for their representation in a 2d graph.
            val y :Double  = Math.random()
            if (x * x + y * y <= 1.0){//Checks if the dot is inside the circle, if it is it adds 1 to the counter
                inside++
            }
        }
```

Fig. 7. Thread class for Pi

Basically that is the most important parts for Java threads.

2.2 Coroutines

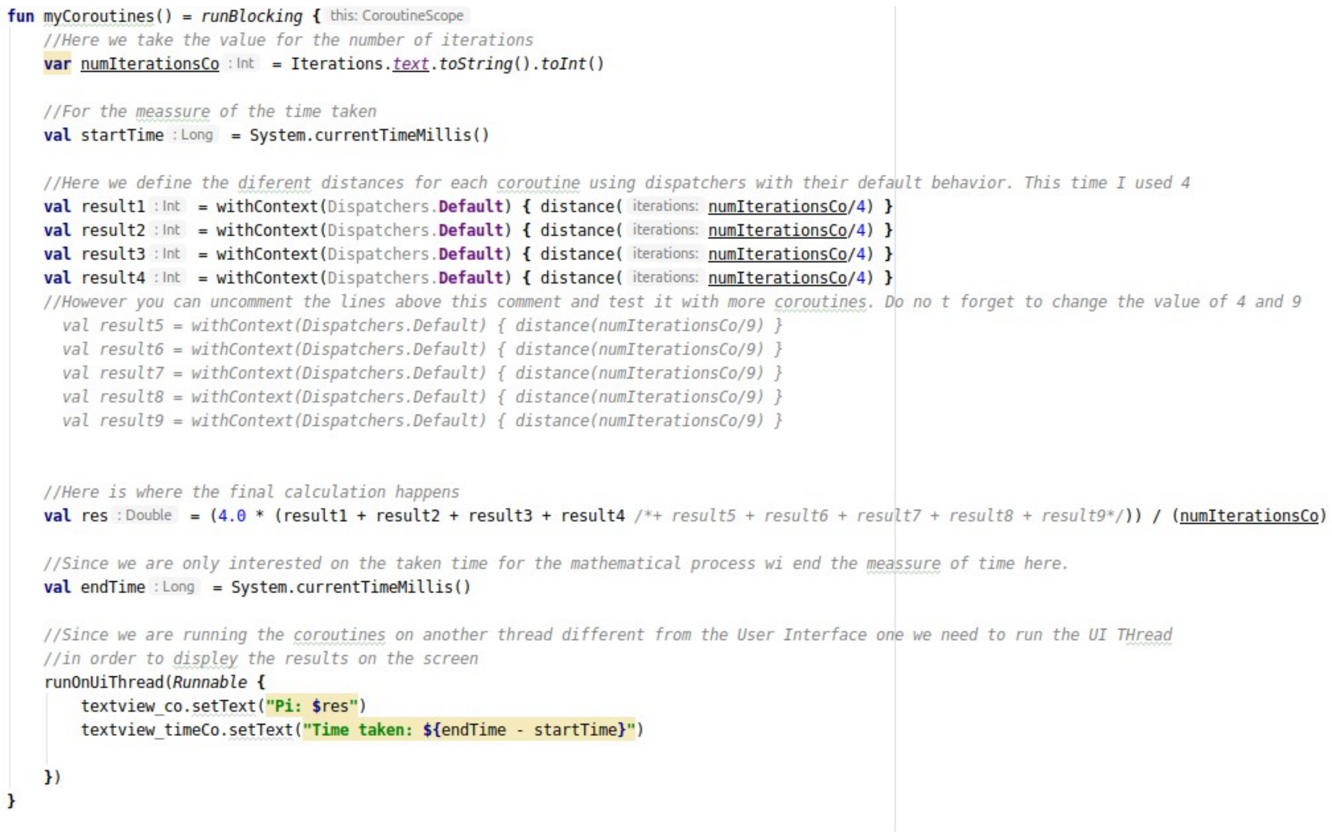
This time there is also an entry input by the user, so it is important to validate that the field is not empty. However, the most interesting part is the creation of the coroutines.

First of all, as threads, it is necessary to create a function that will create the coroutines. This time it is a little bit different. There are several ways for implementing this alternative to threads. Therefor, here is the method that uses *runBlocking*, the method that may be the most similar to threads but with more efficiency.

With *runBlocking* the code can use structured concurrency. Moreover, instead of creating global scopes, which are almost the same as creating regular threads, *runBlocking* create coroutines inside a global scope. This means that there is only one thread. Additionally, with this method it is not necessary to join the coroutines because an outer coroutine (*runBuilder*) does not complete until all the courutines launched in its main coroutine (thread) are complete.

Inside the function the functionthe creation of the coroutines may be done with a dispatcher that, as its name indicates, dispatches the different threads or coroutines that arrive. There are different configurations for the dispatchers. At the moment the default one is enough.

Function that Generates Coroutines

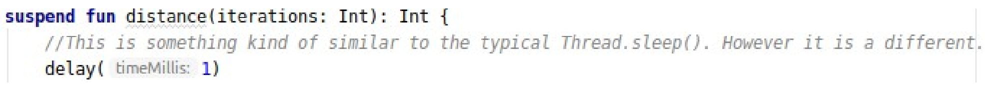


```kotlin
fun myCoroutines() = runBlocking { this: CoroutineScope
    //Here we take the value for the number of iterations
    var numIterationsCo : Int  = Iterations.text.toString().toInt()

    //For the meassure of the time taken
    val startTime : Long  = System.currentTimeMillis()

    //Here we define the diferent distances for each coroutine using dispatchers with their default behavior. This time I used 4
    val result1 : Int  = withContext(Dispatchers.Default) { distance( iterations: numIterationsCo/4) }
    val result2 : Int  = withContext(Dispatchers.Default) { distance( iterations: numIterationsCo/4) }
    val result3 : Int  = withContext(Dispatchers.Default) { distance( iterations: numIterationsCo/4) }
    val result4 : Int  = withContext(Dispatchers.Default) { distance( iterations: numIterationsCo/4) }
    //However you can uncomment the lines above this comment and test it with more coroutines. Do no t forget to change the value of 4 and 9
      val result5 = withContext(Dispatchers.Default) { distance(numIterationsCo/9) }
      val result6 = withContext(Dispatchers.Default) { distance(numIterationsCo/9) }
      val result7 = withContext(Dispatchers.Default) { distance(numIterationsCo/9) }
      val result8 = withContext(Dispatchers.Default) { distance(numIterationsCo/9) }
      val result9 = withContext(Dispatchers.Default) { distance(numIterationsCo/9) }


    //Here is where the final calculation happens
    val res : Double  = (4.0 * (result1 + result2 + result3 + result4 /*+ result5 + result6 + result7 + result8 + result9*/)) / (numIterationsCo)

    //Since we are only interested on the taken time for the mathematical process wi end the meassure of time here.
    val endTime : Long  = System.currentTimeMillis()

    //Since we are running the coroutines on another thread different from the User Interface one we need to run the UI THread
    //in order to displey the results on the screen
    runOnUiThread(Runnable {
        textview_co.setText("Pi: $res")
        textview_timeCo.setText("Time taken: ${endTime - startTime}")

    })
}
```

Fig. 8. Thread class for Pi

Finally there need to be a class for the coroutines. Also, this is a little bit different in its definition. But, inside it it is the same as the one that can be found in the threads explanation.

Function that Generates Coroutines



```kotlin
suspend fun distance(iterations: Int): Int {
    //This is something kind of similar to the typical Thread.sleep(). However it is a different.
    delay( timeMillis: 1)
```

Fig. 9. Class for coroutines

It is noticeable that there is the word “suspend” in the definition of the function. The reason is that here is the place where the coroutine switches between other coroutines in order to achieve the parallelization.

Also, there is the function *delay() .* This function is a special way of suspending the function. This is exclusive for coroutines and it will not block a thread. It suspends the coroutine.

3. Tests

It has been told that coroutines are way more efficient than threads, so it is the moment for some tests.

3.1 First Case

For the first set of test the main variable is the amount of iterations. However, there may be some differences because of the random generation of the points. In consequence, there are eight different measures for each test case and then the mean is taken in consideration for the final results.

The used values and the results can be seen in the following table.

Remember to see Apendixes for some photographs of the tests.

Test 1 Results Table Threads

Run	Threads used	Number of iterations	Pi Value	Time Taken (ms)
1	4	10000	3.14	70
2	4	10000	3.164	9
3	4	10000	3.1216	10
4	4	10000	3.1568	11
5	4	10000	3.12.88	8
6	4	10000	3.134	14
7	4	10000	3.1344	8
8	4	10000	3.116	12
MEAN	-	-	3.138114286	17.75

Fig. 10. Results with four threads and 10000 iterations

Test 1 Results Table Coroutine

Run	Coroutines used	Number of iterations	Pi Value	Time Taken (ms)
1	4	10000	3.14	70
2	4	10000	3.164	9
3	4	10000	3.1216	10
4	4	10000	3.1568	11
5	4	10000	3.12.88	8
6	4	10000	3.134	14
7	4	10000	3.1344	8
8	4	10000	3.116	12
MEAN	-	-	3.15045	31.125

Fig. 11. Results with four coroutines and 10000 iterations

At the moment the thread solution seems to be more efficient than the coroutine method.

3.2 Second Case

Used values and results can be seen in the following tables.

Test 2 Results Table Threads

Run	Threads used	Number of iterations	Pi Value	Time Taken (ms)
1	4	1000000	3.137772	383
2	4	1000000	3.144296	366
3	4	1000000	3.141796	390
4	4	1000000	3.140736	358
5	4	1000000	3.14058	347
6	4	1000000	3.142832	344
7	4	1000000	3.140884	381
8	4	1000000	3.14152	345
MEAN	-	-	3.141302	364.25

Fig. 12. Results with four threads and 1000000 iterations

Test 2 Results Table Coroutine

Run	Coroutines used	Number of iterations	Pi Value	Time Taken (ms)
1	4	1000000	3.139848	230
2	4	1000000	3.1415	121
3	4	1000000	3.143132	303
4	4	1000000	3.143164	294
5	4	1000000	3.143016	251
6	4	1000000	3.138188	322
7	4	1000000	3.143712	246
8	4	1000000	3.142552	247
MEAN	-	-	3.141889	251.75

Fig. 13. Results with four coroutines and 10000000iterations

At this point some difference can be seen between coroutines and threads. This time Threads were more efficient.

3.3 Third Case

In this case only 5 examples are presented. The amount of taken time is increasing.

Test 3 Results Table Threads

Run	Threads used	Number of iterations	Pi Value	Time Taken (ms)
1	4	100000000	3.14174872	38372
2	4	100000000	3.14174548	38482
3	4	100000000	3.151683	39866
4	4	100000000	3.14181832	35838
5	4	100000000	3.14164844	36957
MEAN	-	-	3.143728792	37903

Fig. 14. Results with four threads and 100000000 iterations

Test 3 Results Table Coroutine

Run	Coroutines used	Number of iterations	Pi Value	Time Taken (ms)
1	4	100000000	3.14167788	10935
2	4	100000000	3.1417012	11149
3	4	100000000	3.14143864	11097
4	4	100000000	3.1415384	10930
5	4	100000000	3.141427	11009
MEAN	-	-	3.141556624	11024

Fig. 15. Results with four coroutines and 100000000 iterations

Now it is really appreciable that coroutines are very efficient compared with threads, especially when the amount of iterations are huge.

4. Conclusion

There are a lot of usable and important tool for software development. Nowadays, every single person with a computer wants everything very quickly, programmers can use threads or coroutines, if available, in order to have a great performance. Parallelization is a common tool nowadays, it must be learn t by almost everyone who knows basic coding. Many of the most simple, but common examples can take advantage of this paradigm.

Regarding with threads and it alternative, it can be seen that coroutines may be, in efficiency terms, really superior compared with regular threads. However, there are some cases when threads can be better, it all depends on the task and the objective platform. Also the language is really important, since there are some languages out there that does not support this tool or does not behave as it should. In android it seems to be the better option when anyone wants to prallelize a task.

5. Sources

[1] Ian Fooster. “7.1 Data Parallelism“. In URL:

https://www.mcs.anl.gov/~itf/dbpp/text/node83.html

[2] W. Richard Stevens, Stephen A. Rago. *Advanced Programming in the UNIX Environment: Second Edition, 2015*

[3]Academo, “Estimating Pi using the Monte Carlo Method”. In URL:

https://academo.org/demos/estimating-pi-monte-carlo/

[4]Kotlinlang, “Language Guide”. In URL:

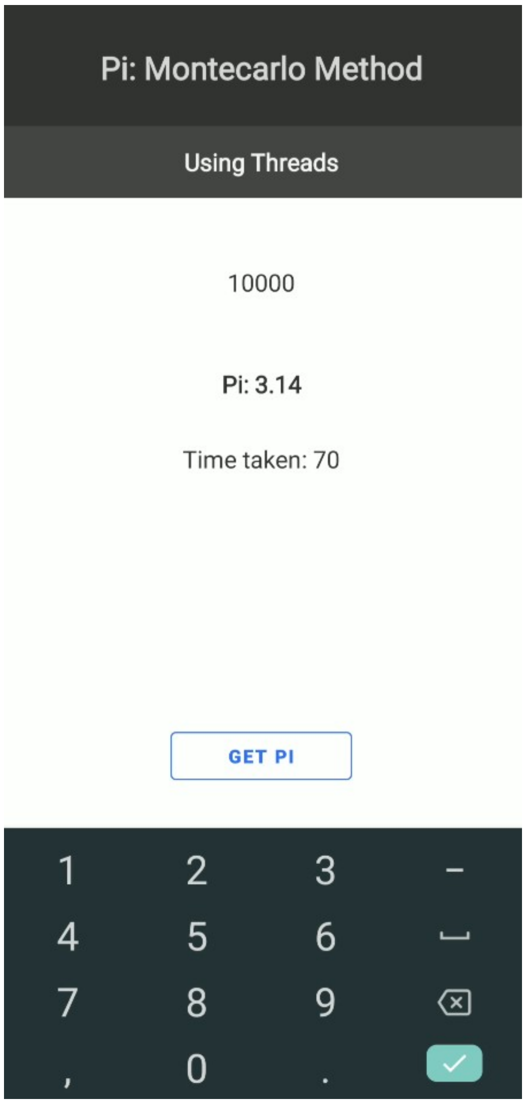
https://kotlinlang.org/docs/reference/coroutines/basics.html

6. Apendixes

Test 1

Threads

Coroutines



Pi: Montecarlo Method

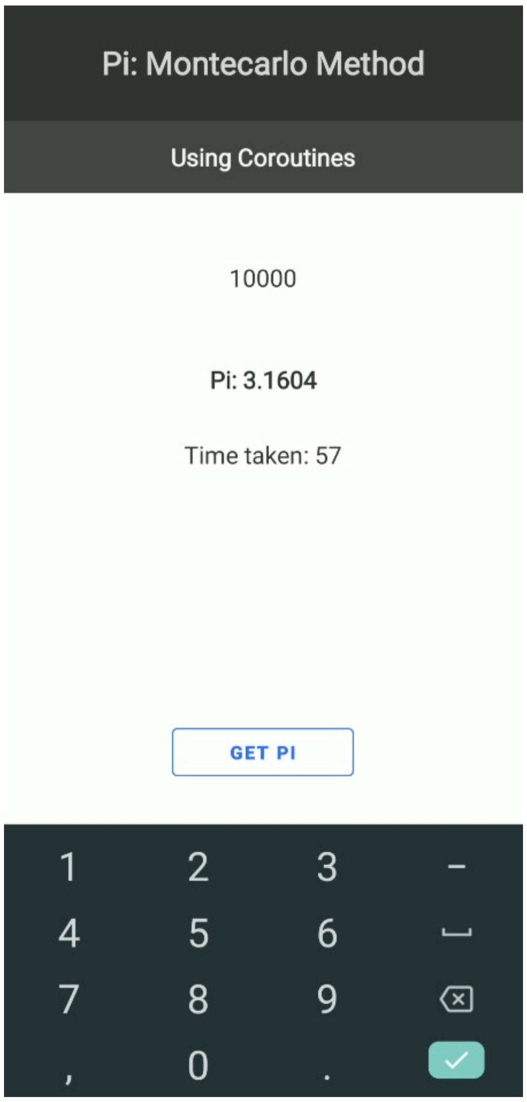
Using Threads

10000

Pi: 3.14

Time taken: 70

1. Case 1, Test 1



Pi: Montecarlo Method

Using Coroutines

10000

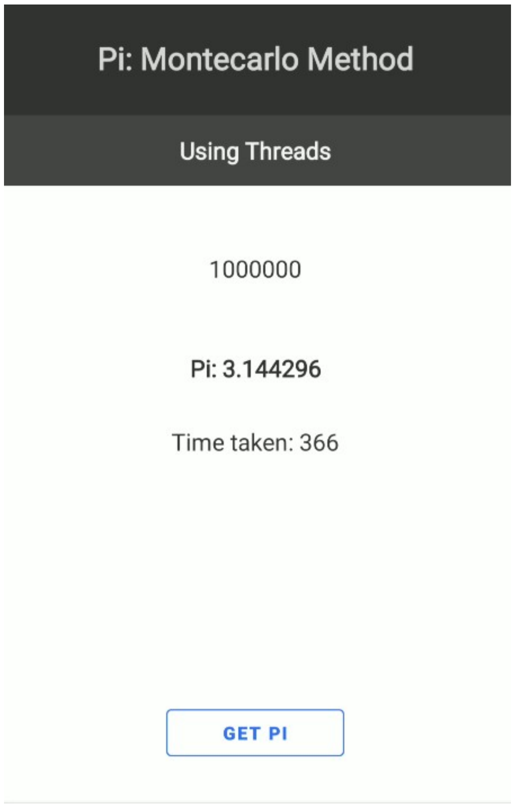
Pi: 3.1604

Time taken: 57

2. Case 1, Test 1

Test 2

Threads



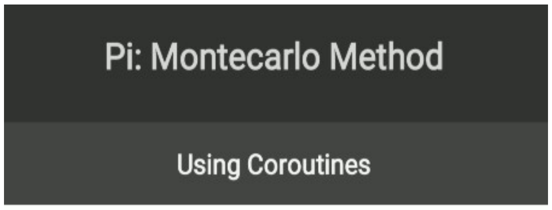
1000000

Pi: 3.144296

Time taken: 366

3. Case 2, Test 2

Coroutines



1000000

Pi: 3.1415

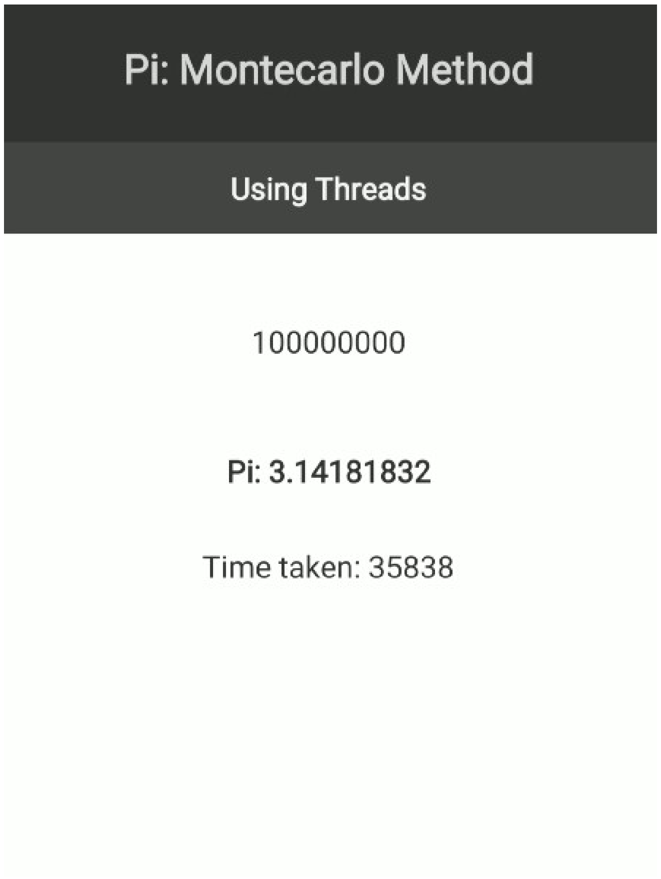
Time taken: 121

4. Case 2, Test 2

Test 3

Threads

Coroutines

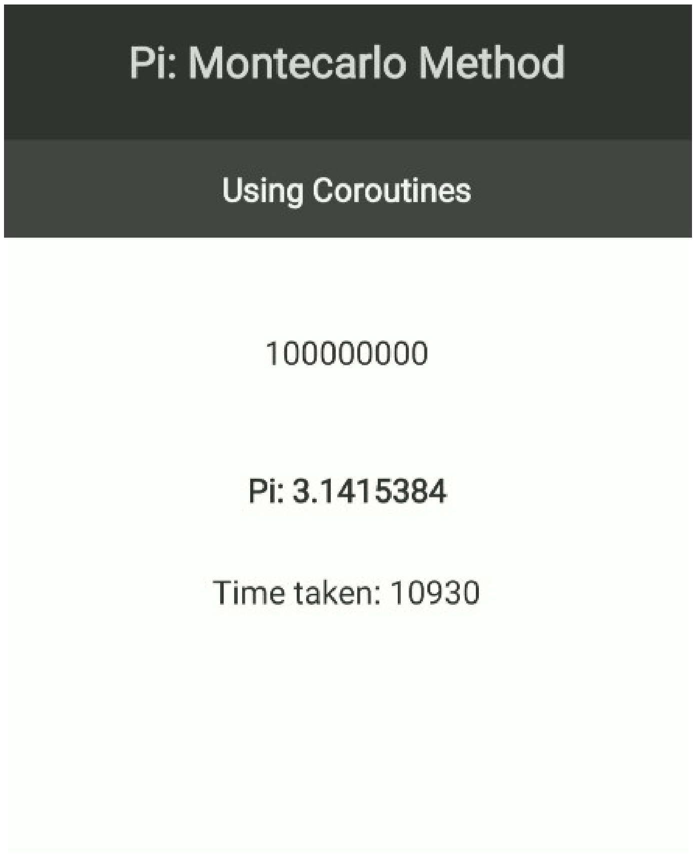

Pi: Montecarlo Method

Using Threads

100000000

Pi: 3.14181832

Time taken: 35838


Pi: Montecarlo Method

Using Coroutines

100000000

Pi: 3.1415384

Time taken: 10930

5. Case 4, Test 3

6. Case 4, Test 3