Calculating π concurrently

This article is a rewrite of a blogpost about calculating π concurrently using Scala's `Future` and that was published in 2014. Two major changes that are introduced are:

- Discuss alternative implementations based on Akka Streams.
- Use Scala 3 instead of Scala 2 and utilise the existing Akka 2.6.14 library that was compiled with Scala 2 in a transparant fashion.

All source code shown in this article can be found here.

Introduction

With the prevalence of multi-CPU computers and multi-core CPUs, the opportunity exists to speed-up the execution of applications in general and algorithms that lend themselves to parallelisation specifically.

If we look at the latter, we may find that the algorithm can be decomposed into subtasks which can be executed in parallel. Getting the final result then consists of combining the sub-results of the subtasks. There we can distinguish two cases. In the first case, the order in which the sub-results are composed doesn't matter, whereas in the second case, it does.

An example of the first case is when the sub-results are numbers and the final result is their sum. Obviously, the order in which the numbers are added doesn't change the result (to be complete, effects of rounding due to finite precision *can* impact the final result, but that's an interesting but entirely different topic). An example where order definitely matters is if we multiply a series of matrices: if we split the multiplication into groups, calculate the product of the matrices in each group, then the order in which we combine the sub-results *does* matter!

In Scala, we have different tools available to write applications that execute code concurrently, and if we run this code on a multi-CPU or multi-core computer, we can achieve parallelism in the execution of this code.

A first tool we have is Akka: Akka allows us to create so-called Actors that can execute code. This execution can be triggered by sending an Actor a message. In the context of concurrent execution of code, we can imagine a system where we create a pool of Actors that execute code concurrently. This execution model allows us to implement concurrent execution of sub-tasks that fall in the first category mentioned earlier. Unless we take specific measures, we have no control over the order in which sub-results are returned. Another potential issue with an Actor based implementation is that we don't have any flow control. What this means is that if we throw more work at an Actor than this Actor can process, work will get piled-up in a buffer (actually, the Actor's mailbox). The conclusion is that the Actor model can be applied in a straightforward way to problems of the first type, albeit that we need to be aware of potential ordering and flow control issues. We'll come back to this topic later when we discuss solutions to our sample problem based on Akka Streams.

A second tool we have at our disposal is Scala's `Future`. This API allows us to tackle both of the aforementioned cases.

Let's see how `Future` can be used to implement a parallelised computation of the number π.

Calculating π

Ok, this topic certainly has been beaten to death, but it's easy to understand, so we may just as well use it for demonstration purposes…

I chose the *Bailey—Borwein—Plouffe formula (BBP)* discovered in 1995 by *Simon Plouffe* :

$$\pi = \sum_{n=0}^{\infty} \frac{1}{16^n} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right)$$

There are more efficient ways to calculate π: for example, the Chudnovsky algorithm is a series that generates about 14 digits per term! I chose to not use this algorithm as it unnecessarily adds complexity to the topic of this article.

The *BBP* algorithm has a very interesting property (that we won't exploit in our case): if the calculation is performed in hexadecimal number base, the algorithm can produce any individual digit of π without calculating all the digits preceding it! In this way, it is often used as a mechanism to verify the correctness of the calculated value of π using a different algorithm.

On precision

Obviously, if we calculate an approximation of π by calculating the sum of the `N` first terms of the *BBP* formula shown earlier, we must calculate each term with sufficient precision. If we would perform the calculation using `Double`, we wouldn't be able to calculate many digits…

Let's use `BigDecimal` instead. `BigDecimal` allows us to create numbers of practically arbitrary precision specified in a so-called `MathContext`. Let's create some numbers with 100 digit precision and perform a calculation with them:

```
scala> import java.math.MathContext
import java.math.MathContext

scala> val mc = new MathContext(100)
mc: java.math.MathContext = precision=100 roundingMode=HALF_UP

scala> val one = BigDecimal(1)(mc)
one: scala.math.BigDecimal = 1

scala> val seven = BigDecimal(7)(mc)
seven: scala.math.BigDecimal = 7

scala> println((one/seven).toString)
0.14285714285714285714285714285714285714285714285714285714285714285714285714285714285714
1428571428571429
```

Each time we create a `BigDecimal` number we need to specify the required precision by supplying the corresponding MathContext. A bit clumsy, so let's use a Scala feature so that we use `BigDecimal` in a more straightforward way:

```
scala> import java.math.MathContext

scala> import scala.math.{BigDecimal as ScalaBigDecimal}

scala> object BigDecimal:
         def apply(d: Int)(using mc: MathContext): BigDecimal = ScalaBigDecimal(d, mc)
// defined object BigDecimal

scala> given MathContext = new MathContext(100)
lazy val given_MathContext: java.math.MathContext

scala> val one = BigDecimal(1)
one: scala.math.BigDecimal = 1

scala> val seven = BigDecimal(7)
seven: scala.math.BigDecimal = 7

scala> println((one/seven).toString)
0.14285714285714285714285714285714285714285714285714285714285714285714285714285714285714
```

If you're unfamiliar with Scala's contextual abstractions (formerly known as 'implicits'), the above may look a bit strange. In fact it works as follows: in object `BigDecimal`, we define an apply method that has two argument lists. The purpose of the first one is obvious. The second one takes a single argument, namely a `MathContext`. This argument (in fact, this applies to the argument list as a whole) is marked *using*. What this will do is that the compiler will, in the current scope, look for a *given* value of type `MathContext`. The declaration `given MathContext = new MathContext(100)` meets that criterium. Hence, when we invoke a call to the apply method in our `BigDecimal` object, value `mc` will automatically be passed as the second argument.

Cool, and now that we've got that out of the way, let's look at how we can perform the calculation.

Implementing the BBP digit extraction algorithm

An obvious way to parallelize the calculation of the BPP formula shown above up to N terms is to split the terms into `nChunks` chunks and to calculate the sum of the terms in the chunks in parallel. When that's done, the final result (π) is the sum of the partial sums.

For each chunk, we need to know its offset in the sequence of terms. Let's generate a sequence that contains the offsets of the respective chunks for a given `N` and `nChunks`. Note that the latter may not divide entirely into the former, so we do a bit of rounding that may result in calculating extra terms:

```
scala> val N = 3000
N: Int = 3000

scala> val nChunks = 64
nChunks: Int = 64

scala> val chunkSize = (N + nChunks - 1) / nChunks
chunkSize: Int = 47

scala> val offsets = 0 until N by chunkSize
val offsets: Range = inexact Range 0 until 3000 by 47

scala> println(s"Calculating π with ${nChunks*chunkSize} terms in $nChunks chunks of
$chunkSize terms each")
Calculating π with 3008 terms in 64 chunks of 47 terms each
```

Next we define a method `piBBPdeaPart` that will calculate the sum of `n` terms in the BBPDEA formula, starting at term `offset`.

```
  def piBBPdeaPart(offset: Int, n: Int): BigDecimal =
    def piBBPdeaTermI(i: Int): BigDecimal =
      BigDecimal(1) / BigDecimal(16).pow(i) * (
        BigDecimal(4) / (8 * i + 1) -
        BigDecimal(2) / (8 * i + 4) -
        BigDecimal(1) / (8 * i + 5) -
        BigDecimal(1) / (8 * i + 6)
      )
    println(s"Started @ offset: $offset ")
    (offset until offset + n).foldLeft((BigDecimal(0))) {
      case (acc, i) => acc + piBBPdeaTermI(i)
    }
```

Relatively straightforward, and time to tie everything together. Note the presence of a println statement that prints some text just before the calculation of a partial sum starts. Let's start by launching the calculation of the sum of the chunks:

```
  val piChunks: Future[Seq[BigDecimal]] =
    Future.sequence(
      for  offset <- offsets
        yield Future(piBBPdeaPart(offset, chunkSize))
    )
```

Two things are important to note. First we map each offset in `offsets` to a `Future[BigDecimal]`; it is here that we introduce concurrency and each part of the calculation will be scheduled for execution within an execution context (that we haven't provided yet). What we end up with is a sequence of `Future`s. Secondly, `Future.sequence` converts the `Seq[Future[BigDecimal]]` into a `Future[Seq[BigDecimal]]`. Pretty awesome.

What remains to be done is to calculate the sum of the partial sums. We can do this as follows:

```
  val piF: Future[BigDecimal] = piChunks.map(_.sum)
```

If the previous was awesome, this certainly is awesome++. Think about it: we're performing a calculation on a Future, but it sure looks as if we're working on the concrete thing: `piChunks` is a `Future[Seq[BigDecimal]]`.

When we apply map on this future, we can work with a lambda that works on a `Seq[BigDecimal]`.

The relevant (simplified) part in the source code of `Future` is as follows:

```
trait Future[+T] extends Awaitable[T] {
  ...
  def map[S](f: T => S): Future[S] = {
    ...
  }
  ...
}
```

Variable `piF` is still a `Future[BigDecimal]`. So, if we want to do something with the final result, we can do this by registering a callback via `Future.onComplete`.

This is done as follows:

```
  piF.onComplete {
    case Success(pi) =>
      val stopTime = System.currentTimeMillis
      println(s"Pi:       ${pi}")
      val delta = pi - Helpers.piReference
      Helpers.printMsg(s"|Delta|: ${delta(new MathContext(8)).abs}")
      Helpers.printCalculationTime(startTime, stopTime)
      fjPool.shutdown()
    case Failure(e) =>
      println(s"An error occurred: ${e}")
      fjPool.shutdown()
  }
```

Note that we are using a few helper functions such as `printMsg` and `printCalculationTime` (defined in object `Helpers`) to print out the difference between the calculated value and a reference value of π (with the latter being read from a file).

Execution context and thread pools

The above code contains almost everything that is needed. However, if we compile it, we get the following error:

```
[error] -- Error: /Users/ericloots/Trainingen/LBT/calculating-pi-
concurrently/step_001_calculating_pi_with_futures/src/main/scala/com/lunatech/pi/Futur
es.scala:52:55
[error] 52 |          yield Future(piBBPdeaPart(offset, chunkSize))
[error]    |                                                       ^
[error]    |Cannot find an implicit ExecutionContext. You might add
[error]    |an (implicit ec: ExecutionContext) parameter to your method.
[error]    |
[error]    |The ExecutionContext is used to configure how and on which
[error]    |thread pools asynchronous tasks (such as Futures) will run,
[error]    |so the specific ExecutionContext that is selected is important.
[error]    |
[error]    |If your application does not define an ExecutionContext elsewhere,
[error]    |consider using Scala's global ExecutionContext by defining
[error]    |the following:
[error]    |
[error]    |implicit val ec: scala.concurrent.ExecutionContext =
scala.concurrent.ExecutionContext.global
[error]    |
[error]    |The following import might fix the problem:
[error]    |
[error]    |  import concurrent.ExecutionContext.Implicits.global
                                                              ^
```

Looking at the (simplified - Scala 2) signature of `Future` we see the following:

```
object Future {
  ...
  def apply[T](body: =>T)(implicit executor: ExecutionContext): Future[T] = ...
  ...
}
```

So, we need to provide a so-called ExecutionContext. An ExecutionContext will provide the machinery (Threads) on which the Future code (body in the signature) will be run.

We can provide an ExecutionContext in the following way:

```
  val fjPool = new ForkJoinPool(12)

    given ExecutionContext = ExecutionContext.fromExecutor(fjPool)
```

Here, we create a ForkJoinPool with 12 threads and create an ExecutionContext from it. This 'given' value will now be picked-up by our calls to Future.apply...

Wrap-up

Following is the complete code:

```
1  package com.lunatech.pi
2
3  import java.math.MathContext
4  import java.util.concurrent.ForkJoinPool
5
6  import scala.concurrent.*
7  import scala.math.{BigDecimal as ScalaBigDecimal}
8  import scala.util.{Failure, Success}
9
10 object Main:
11   def main(args: Array[String]): Unit =
12
13     val RunParams(iterationCount, precision) = Helpers.getRunParams(args)
14
15     Helpers.printMsg(s"Iteration count = $iterationCount - Precision = $precision")
16
17     given MathContext = new MathContext(precision)
18
19     object BigDecimal:
20       def apply(d: Int)(using mc: MathContext): BigDecimal = ScalaBigDecimal(d, mc)
21
22     def piBBPdeaPart(offset: Int, n: Int): BigDecimal =
23       def piBBPdeaTermI(i: Int): BigDecimal =
24         BigDecimal(1) / BigDecimal(16).pow(i) * (
25           BigDecimal(4) / (8 * i + 1) -
```

```
26            BigDecimal(2) / (8 * i + 4) -
27            BigDecimal(1) / (8 * i + 5) -
28            BigDecimal(1) / (8 * i + 6)
29          )
30        println(s"Started @ offset: $offset ")
31        (offset until offset + n).foldLeft((BigDecimal(0))) {
32          case (acc, i) => acc + piBBPdeaTermI(i)
33        }
34
35      val fjPool = new ForkJoinPool(Settings.parallelism)
36
37      given ExecutionContext = ExecutionContext.fromExecutor(fjPool)
38
39      val N = iterationCount
40      val nChunks = Settings.BPP_chunks
41      val chunkSize = (N + nChunks - 1) / nChunks
42      val offsets = 0 to N by chunkSize
43      Helpers.printMsg(s"Calculating π with ${nChunks * chunkSize} terms in $nChunks
    chunks of $chunkSize terms each")
44      Helpers.printMsg(s"Threadpool size: ${Settings.parallelism}")
45      Helpers.printMsg(s"BigDecimal precision settings: ${summon[MathContext]}")
46
47      val startTime = System.currentTimeMillis
48
49      val piChunks: Future[Seq[BigDecimal]] =
50        Future.sequence(
51          for  offset <- offsets
52            yield Future(piBBPdeaPart(offset, chunkSize))
53        )
54
55      val piF: Future[BigDecimal] = piChunks.map(_.sum)
56
57      piF.onComplete {
58        case Success(pi) =>
59          val stopTime = System.currentTimeMillis
60          println(s"Pi:      ${pi}")
61          val delta = pi - Helpers.piReference
62          Helpers.printMsg(s"|Delta|: ${delta(new MathContext(8)).abs}")
63          Helpers.printCalculationTime(startTime, stopTime)
64          fjPool.shutdown()
65        case Failure(e) =>
66          println(s"An error occurred: ${e}")
67          fjPool.shutdown()
68      }
```

Note that the number of threads in the `ForkJoinPool` and the number of *chunks* are obtained from settings in object `Settings`. The actual value can be set as a configuration setting (`calculating-pi.parallelism` and `calculating-pi.bpp-chunks` respectively).

When this program is executed on my laptop (a MacBook Pro with a 2,6 GHz 6-Core Intel Core i7

processor), it produces the following output (values of π truncated):

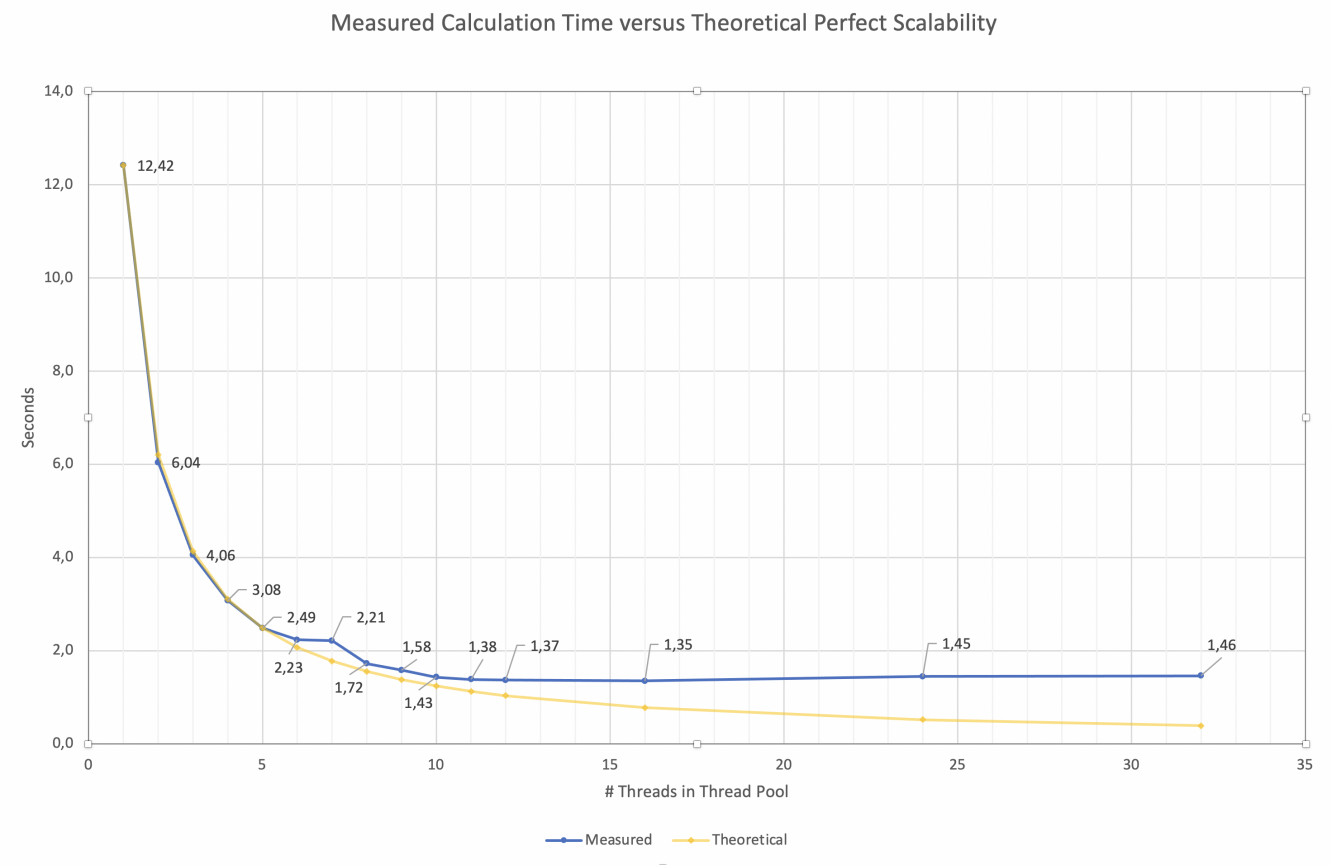
```
man [e] > calculating-pi-concurrently > calculating pi with futures > run 3000 4000
[info] running com.lunatech.pi.Main 3000 4000
Iteration count = 3000 - Precision = 4000
Calculating π with 3008 terms in 64 chunks of 47 terms each
Threadpool size: 12
BigDecimal precision settings: precision=4000 roundingMode=HALF_UP
Started @ offset: 0
Started @ offset: 94
Started @ offset: 141
Started @ offset: 47
Started @ offset: 188
Started @ offset: 235
Started @ offset: 282
Started @ offset: 329
Started @ offset: 423
Started @ offset: 376
Started @ offset: 470
Started @ offset: 517
[success] Total time: 1 s, completed 19 Apr 2021, 18:51:35
Started @ offset: 564
Started @ offset: 611
Started @ offset: 658
Started @ offset: 705
Started @ offset: 752
Started @ offset: 799
Started @ offset: 846
Started @ offset: 893
Started @ offset: 940
Started @ offset: 987
Started @ offset: 1034
Started @ offset: 1081
Started @ offset: 1128
Started @ offset: 1175
Started @ offset: 1222
Started @ offset: 1269
Started @ offset: 1316
Started @ offset: 1363
Started @ offset: 1410
Started @ offset: 1457
Started @ offset: 1504
Started @ offset: 1551
Started @ offset: 1598
Started @ offset: 1645
Started @ offset: 1692
Started @ offset: 1739
Started @ offset: 1786
Started @ offset: 1833
Started @ offset: 1880
```

```
Started @ offset: 1927
Started @ offset: 1974
Started @ offset: 2021
Started @ offset: 2068
Started @ offset: 2115
Started @ offset: 2162
Started @ offset: 2209
Started @ offset: 2256
Started @ offset: 2303
Started @ offset: 2350
Started @ offset: 2397
Started @ offset: 2444
Started @ offset: 2491
Started @ offset: 2538
Started @ offset: 2585
Started @ offset: 2632
Started @ offset: 2679
Started @ offset: 2726
Started @ offset: 2773
Started @ offset: 2820
Started @ offset: 2867
Started @ offset: 2914
Started @ offset: 2961
Pi:      3.14159265358979323846264338327950288419716 9...
|Delta|: 2.8076968E-3630
Calculation time: 1.345
```

Note that we set the number of *chunks* to 64 in this run.

What we can observe is that, with 3,008 terms, we have correctly calculated more than 3,600 digits accurately.

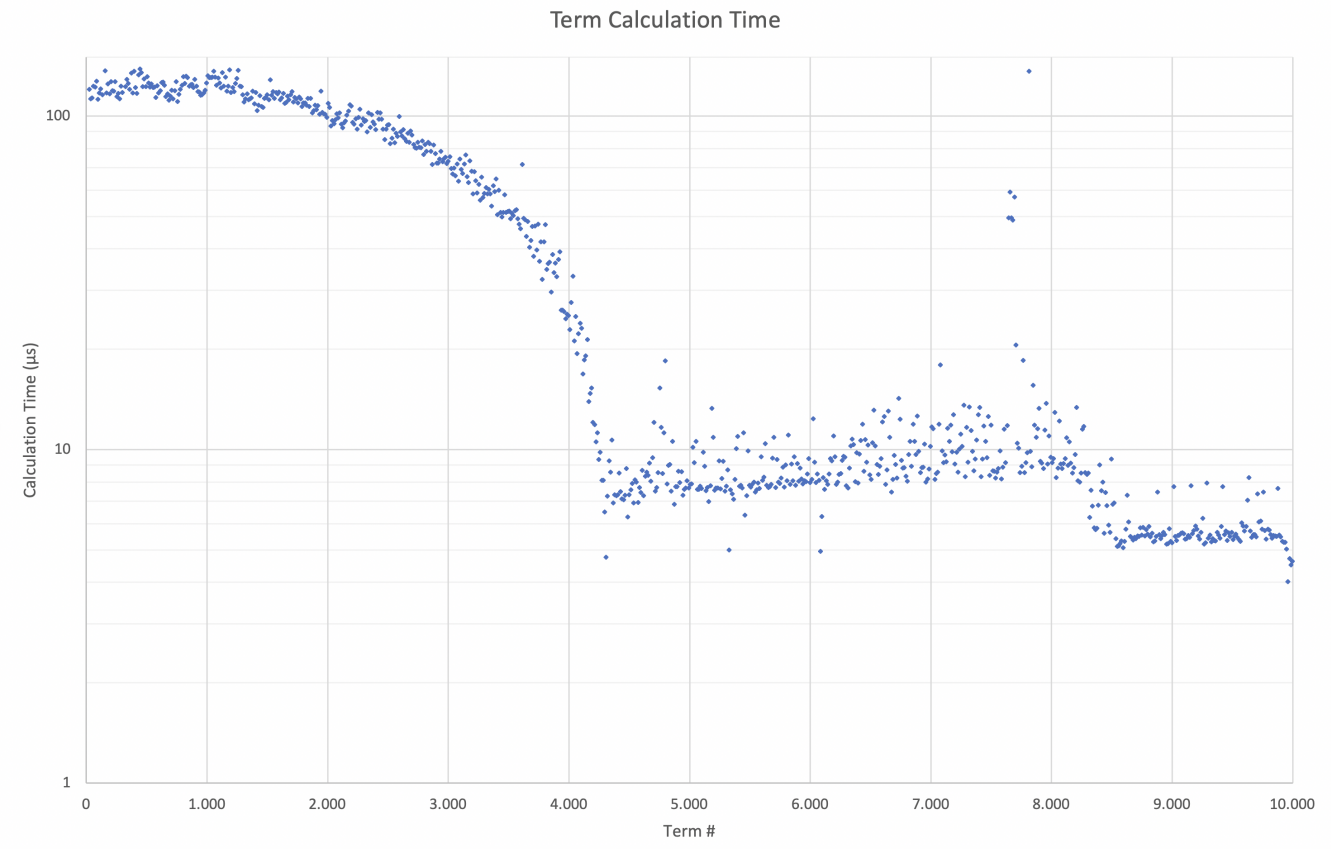
We can measure the effect of changing the Thread pool size on the total calculation time. The following graphs shows the relation between the two and it also shown another (orange) curve that plots the case when the system scales perfectly and unbounded.



So, we see a nearly linear speed-up by increasing the number of threads from 1 to 12. A further increase of the thread-count doesn't yield a further linear speed-up: this may be caused by different factors, not in the least by the fact that we have a single chip processor with a shared on-chip cache. Of course, since it's a six core CPU (with hyper-threads that don't yield the same performance as the regular CPU threads), we don't get a speed-up beyond 12 threads in the ForkJoinPool.

We can do another experiment by setting the number of *chunks* and the number of threads in the thread pool to 12. One would expect that this should be no impact. In practice however, the total execution time increases from 1.37s to 3.31s! How can that be?

The explanation can be found by looking at the following graph which plots the time to calculate each term in the series (and this for a run with 10,000 calculated terms).



We see that the time to calculate terms decreases significantly between the first term and term 4,300 at which point it reaches kind of a plateau.

As we divide the calculation into chunks, the first few chunks will determine the overall speed at which the calculation is performed.

It turns out that the choice of chunking the calculation is not optimal. Another approach is to spread the calculation across different threads in a round robin fashion as illustrated in an alternative implementation in the source code.

Conclusion on the `Future` approach

Scala's `Future` API presents a very powerful way to perform asynchronous and concurrent execution of code. Even though it may take some time to wrap one's head around it, once you grasp it, it's pretty cool and very powerful.

We've also seen that we should *"know our data"* as demonstrated with the *chunking* versus round robin implementation.

Now, as for π, is the approach used in this article to calculate π a realistic way to calculate this number to say multi-billion digit precision? Not really for multiple reasons.

First of all, this algorithm runs in-memory. If we consider that the current record for calculating π digits is at 50 trillion digits, there's no computer that can hold the required size of numbers in memory.

Secondly, the algorithm is too slow compared to the algorithm that is currently used to set the

record(s).

Consider that the current record holder is *Timothy Mullican* who calculated the 50.000.000.000.000 first digits of π. It took 303 days to complete the calculation.

What should be mentioned though is that the code that was used to perform the calculation is y-cruncher. This C++ program has been written by Alexander Yee. Between 2010 and 2013, he and Shigeru Kondo set various records for calculating π. If you're interested, have a look at this website.

y-cruncher utilises a different formula than the one used in this article, namely the Chudnovsky formula. Interesting to note is that earlier work by the famous, self-taught, and brilliant Indian mathematician Srinivasa Ramanujan inspired the Chudnovsky brothers to come up with their formula.

Using Akka Streams instead of `Future`

In the first part of the article, we implemented a concurrent version of the calculation of π using the BBP formula using Scala's `Future`. You may also remember that, in the introduction section of the article, we mentioned Actors as way to introduce concurrency in code and we cited a few difficulties with this approach such as issues with ordering and lack of flow control. It turns out however that we can work with Actors without having to deal with the nitty-gritty details of coding them up while at the same time obtaining ordering and flow control: for this we will use the Akka Streams API. In the remainder of the article, we'll explore alternative approaches to implement the calculation of the number π using this API.

We are not going to explain Akka Streams in detail, as there are plenty of articles that do this very well. Let's limit ourselves to the following:

- An [Akka] Stream processes a (possibly infinite) sequence of elements of a certain type.
- A Stream is built from base components:
  - Sources: a source of elements in a Stream. The elements are of a certain type `T`. A `Source` is a component that has a single output (and no input)
  - Sinks: a termination point for elements that have run through the Stream processing chain. A `Sink` has a single input (and no output)
  - Flows: an element processor with one input and one output. A `Flow` component processes elements one by one. It may filter elements out of a streams or transform them.
- An Akka Stream definition that ties a `Source` to a `Sink`, potentially running through one or several `Flow` components is called a `Runnable Graph`. It can be considered to be a blueprint of our stream processing. In itself, this blueprint doesn't process any elements.
- The Blueprint (a `Runnable Graph`) can be executed by running it on a so-called *Materialiser*. The latter is just a fancy term for one (or more than one – we'll come back on this later in this article) Actor. The Blueprint is optimised by a process called *fusing*. A fused Blueprint is executed as a single entity, something to keep in mind when reading the remainder of this article.
- Akka Streams has built-in flow-control commonly referred to as providing a *back pressure*

mechanism. The latter is a bit of a misnomer as the flow control mechanism is actually implemented as a downstream component signalling demand (i.e. being ready to process the next element) to upstream components.

- Finally, there's the concept of *materialised value* (which has been the source of a lot of confusion). In a nutshell: every Streams component, when run, has a materialised value associated with it. When a `Runnable Graph` is executed, each component (`Source`, `Flow`, `Sink`) will have a *materialised value* and in the most common case, the materialised value of the `Source` component passes through all downstream components and hence, it will be the materialised value of the complete stream. In some cases, it is desirable to make a specific "selection" of a materialised value of one of the Stream components and Akka Streams give one control over this.

With that behind us, let's start coding!

Base Akka Streams implementation

Instead of using Scala's `Future` to calculate terms in the BPP formula, we will use a `Flow` component that will, upon receiving an index `i` of a term, calculate the corresponding term.

What we need is a series of indexes (as a `Source`), a `Flow` component to calculate a term and finally a `Sync` that will sum-up all the terms.

Let's have a look at these in turn. The `Source` that produces the indexes looks as follows:

```
val indexes = Source(0 to iterationCount)
```

In between the `Source` and the `Sink`, we need to have a Stream component that transforms an index to a term. For this, we can apply the `map` combinator defined on `Source`. `map` takes a function as argument and we will supply the `piBBPdeaTermI` function for this.

At the other end, we need to calculate the sum of all the calculated terms. We can do this with the following `Sink`:

```
val sumOfTerms: Sink[BigDecimal, Future[BigDecimal]] =
  Sink.fold[BigDecimal, BigDecimal](BigDecimal(0)) {
    case (acc, term) => acc + term
  }
```

There are a few things to point out here:

- When looking at the type of `sumOfTerms` (annotated explicitly to point this out), we see that it accepts elements of type `BigDecimal`, but there's a second type parameter of type `Future[BigDecimal]`. This is the type of the materialised value of the `Sink`.
- `sumOfTerms` is implemented with `Sink.fold`. This particular fold is very similar to `foldLeft` on most of Scala collections. It takes an initial value for an accumulator (the `acc` variable in the code) and a function that generates a new value of the accumulator from the old value and an

element (the `term` variable in the code).

The following code will build and run the complete stream:

```
val piF: Future[BigDecimal] = indexes
  .map(piBBPdeaTermI)
  .runWith(sumOfTerms)
```

So, we start from a stream of indexes and *map* every index to a corresponding term value. Then, we apply `runWith(sumOfTerms)`, which will actually run the complete blueprint on a single Actor (which is invisible to us).

Variable `piF`, which has an explicit type annotation for documentation purposes, is a `Future[BigDecimal]`. It is the materialised value of the `Sink`.

We can now print the value of π and some other stats in the same way as in the `Future` based solution.

Let's run this version and compare execution times between this and the `Future` based version for a calculation using 10.000 terms calculated at a precision of 10.000 digits.

```
man [e] > calculating-pi-concurrently > calculating pi with akka streams base > run
10000 10000
[info] running com.lunatech.pi.MainAkkaStreams 10000 10000
19:28:24 INFO  [] - Slf4jLogger started
Iteration count = 10000 - Precision = 10000
Calculating π with 10000 terms
BigDecimal precision settings: precision=10000 roundingMode=HALF_UP
Memory size to encode BigDecimal at precision=10000 = 2915.0 bytes
Pi:      3.14159265358979323846264338327950288419716939937510582097494459230781640...
|Delta|: 2.5143328E-9998
Calculation time: 189,784

man [e] > calculating-pi-concurrently > calculating pi with akka streams base >
project step_001_calculating_pi_with_futures
[info] Set current project to step_001_calculating_pi_with_futures (in build
file:/Users/ericloots/Trainingen/LBT/calculating-pi-concurrently/)
man [e] > calculating-pi-concurrently > calculating pi with futures > run 10000 10000
[info] running com.lunatech.pi.Main 10000 10000
Iteration count = 10000 - Precision = 10000
Calculating π with 10048 terms in 64 chunks of 157 terms each
Threadpool size: 12
BigDecimal precision settings: precision=10000 roundingMode=HALF_UP
Pi:      3.14159265358979323846264338327950288419716939937510582097494459230781640...
|Delta|: 3.8566723E-9999
Calculation time: 20,698
```

We see that our Akka Streams based version is about 9 times slower than the `Future` based one. That's a lot slower, but this shouldn't be surprising: as I mentioned, the blueprint runs on a single

actor and it effectively is a single, fused piece of code. The materialiser, an Actor to be precise, processes stream elements sequentially. If you run the Akka Streams version, have a look at the CPU usage during the execution. Making the fair assumption that your system has multiple CPU cores, you will see that this program only uses one core (at 100%). What we need is to utilise the power of the remaining cores. How do we go about that?

One approach is to try to pipeline stages in our overall flow and running these on more than a single Actor. One way to do this is to introduce so-called asynchronous boundaries. Introducing an asynchronous boundary will lead to considering the parts of the Blueprint on either side of the boundary as separate components that are no longer fused together and that will be run on separate Actors when run.

```
val piF: Future[BigDecimal] = indexes
  .map(piBBPdeaTermI).async
  .runWith(sumOfTerms)
```

With this modification, the term calculation and the folding over the terms to calculate the sum will be run on different Actors.

Does this have a measurable effect? It does: the calculation time is reduced by ±4%, which is a minor gain. The reason for this is that the calculation of a term from its index takes a lot more time than adding it to the accumulated value. In such a case, pipelining the two stages will only have a minor impact. For the sake of completeness, pipelining two stages has a maximum effect when the stages take the same time to process an element.

On thing that can be said about this Akka Streams based implementation is that it's as concise as it gets: 3 lines of code to encode the algorithm (not counting the method that calculates a term).

So, how can we exploit the presence of multiple cores in our system? Let's look at two alternatives in the following sections.

Using Akka Stream's `mapAsync` to introduce concurrency

One thing we can learn from the first Akka Streams based implementation is that the term calculation and the summing of the different terms differ a lot in terms of computational complexity with the former being the most complex.

If we want to speed up the calculation, we need to focus on calculating the terms in parallel (just like in the `Future` based version).

A first way to do this is to use the `mapAsync` combinator on `Source` (or `Flow`). `mapAsync` has two argument lists. The first one takes a single argument named `parallelism`: this is a number which will introduce parallel execution of a function that transforms an element in the stream. This function is passed as an argument in the second argument list. There's one catch: the transformed element value has to be wrapped in a `Future`. Our calculation now looks as follows:

```
val piF: Future[BigDecimal] = indexes
    .mapAsync(Settings.parallelism)(i => Future(piBBPdeaTermI(i)))
    .runWith(sumOfTerms)
```

Do we need to introduce any asynchronous barriers to see a significant impact? Let's just give the code a spin to check if we have a positive return from the change we made...

```
man [e] > calculating-pi-concurrently > calculating pi with akka streams mapAsync >
run 10000 10000
[info] running com.lunatech.pi.MainAkkaStreamsMapAsync 10000 10000
21:04:12 INFO  [] - Slf4jLogger started
Iteration count = 10000 - Precision = 10000
Calculating π with 10000 terms
BigDecimal precision settings: precision=10000 roundingMode=HALF_UP
Memory size to encode BigDecimal at precision=10000 = 2915.0 bytes
Pi:      3.14159265358979323846264338327950288419716939937510582097494459230781640...
|Delta|: 1.4332772E-10000
Calculation time: 21,733
```

We're basically on par with the original solution. Compare this implementation with the `Future` based one. I think we can agree that this one is way simpler and easier to understand...

What `mapAsync` does is to asynchronously execute the element transformation in with up-to `parallelism` actors. The order in which transformations end is non-deterministic but the implementation of `mapAsync` retains ordering of the transformed elements. One important thing to note is that this ordering means that implementations based on `mapAsync` are subject to head-of-line blocking: when a number of transformations are in flight, if one of them is much slower than the other ones, [subsequent] transformations will be delayed.

Before we move to another Akka Streams based solution (based on so-called sub-streams), it's worth to make a short detour to talk about the facilities that Akka Streams provides for logging stuff in a running stream.

Logging facilities in Akka Streams

Akka Streams has a nice way to log elements (or transformations thereof).

Let's look at how this is done using a simple example:

```
    val piF: Future[BigDecimal] = indexes
      .log("pi-index", identity)
      .withAttributes(
        Attributes.logLevels(
          onElement = Attributes.LogLevels.Info,
          onFinish = Attributes.LogLevels.Info,
          onFailure = Attributes.LogLevels.Error)
      )
      .mapAsync(Settings.parallelism)(i => Future(piBBPdeaTermI(i)))
      .runWith(sumOfTerms)
```

Assuming you have configured an SLF4J provider (such as logback in the code sample repository), logging stuff is done by inserting a `log` combinator which takes two arguments: the first is the log name, the second is a function that transforms the element in whatever form you're interested in logging. Using the `withAttributes` combinator, we can tweak the level at which things are logged. As can be seen from the code, we can tweak this level for individual elements as well as for [normal] stream completion or stream failure.

In the source code for this example, you will notice that I created an extension `logAtInfo` that allows one to add logging in a less verbose manner.

When this extension is applied, the code thus becomes:

```
  val piF: Future[BigDecimal] = indexes
    .logAtInfo("pi-index")
    .mapAsync(Settings.parallelism)(piBBPdeaTermIF)
    .runWith(sumOfTerms)
```

Using Substreams to introduce concurrency

An alternative way to speed-up our calculation is to utilise Akka Substreams. One can consider Substreams as a way to de-multiplex a stream of elements. Substreams can be created in different ways, but we'll focus on the `groupBy` combinator. `groupBy` takes two arguments, let's start with the second one: this is a function `f` that takes an element and which returns a key. The key will determine to which Substream the element will be sent. The idea is that `f` returns a finite number of unique key values. The first parameter `maxSubstreams` is the number of distinct key values generated by the function. Note that if `f` generates more distinct key values than `maxSubstreams`, the stream will fail. The following code segment shows the splitting of our main stream into Substreams:

```
  // A key generator which cycles through the sequence
  // 0, 1, ..., Settings.parallelism
  val genKey: Int => Int = (index: Int) => index % Settings.parallelism

  val piF: Future[BigDecimal] = indexes
    .groupBy(maxSubstreams = Settings.parallelism, genKey)
```

Next, we can perform the calculation of the terms in each Substream by mapping over each index. Also, we can calculate the sum of all the terms in each Substream:

```
val calculateSum =
  Flow[BigDecimal].fold(BigDecimal(0)){
    case (acc, term) => acc + term
  }

val piF: Future[BigDecimal] = indexes
  .groupBy(maxSubstreams = Settings.parallelism, genKey)
  .map(index => piBBPdeaTermI(index))
  .via(calculateSum)
```

With this, we will have `Settings.parallelism` Substreams that each generate one `BigDecimal` value. In order to calculate the total sum of the terms, we should merge the Substreams into one Stream. We can do this with the `mergeSubstreams` combinator. We can complete the calculation in the same manner as in the previous solution.

The stream processing now looks as follows:

```
val piF: Future[BigDecimal] = indexes
  .groupBy(maxSubstreams = Settings.parallelism, genKey)
  .map(index => piBBPdeaTermI(index))
  .via(calculateSum)
  .mergeSubstreams
  .via(calculateSum)
  .runWith(Sink.head)
```

An important note about the `mergeSubstreams` combinator is that it takes elements from the Substreams as they arrive. This means in our case is that the order in which the subtotals are added isn't deterministic.

When we run this code, we observe that, in terms of performance, we're back to square one. This is because our Blueprint is optimised, fused and run on a single actor. We can fix this by adding an asynchronous boundary in the right location like shown in the final version of the stream processing:

```
val piF: Future[BigDecimal] = indexes
  .groupBy(maxSubstreams = Settings.parallelism, genKey)
  .map(index => piBBPdeaTermI(index)).async
  .via(calculateSum)
  .mergeSubstreams
  .via(calculateSum)
  .runWith(Sink.head)
```

If we compare the different implementations of the calculation, we see that the `Future` based ones (the *Chunk* based one, with a sufficiently high number of chunks, and the round-robin one) and the Akka Streams based ones, we can conclude that their performance is on par.

The source code repository shows two more Akka Streams based implementations. These make use of the so-called Akka Streams Graph API and the balancer/merge approach respectively.

Conclusions

With this, we come to the end of this article. We have implemented a CPU-bound computation and managed to exploit our computer's multi-processing (or multi-core processing) capabilities using two very different approaches. First we used Scala's `Future` API followed by an Akka Streams based approach.

We have seen that the `Future` based implementation is more low level than the Akka Streams based one. Also, the latter produces more concise code and has more 'knobs' to control the concurrency of the execution.

Another important feature of Akka Streams is that it implements flow control (aka back-pressure) which is important when we have to process a very large number (or even an indeterminate number) of elements: the slowest stream processing component in the overall flow will dictate the rate at which elements are produced upstream.