Functional programming with Scala 3

Parallel Processing and Concurrency

Said BOUDJELDA

Senior Software Engineer @SCIAM
Email : mohamed-said.boudjelda@intervenants.efrei.net
Follow me on GitHub @bmscomp

Course, May 2025

Course Overview

- **Parallel vs Concurrent Programming**
- **Scala 3 Collections Parallel Processing**
- **Future and Promise**
- **Async/Await Pattern**
- **Actors with Akka**
- **Functional Reactive Programming**
- **ZIO for Effect Management**
- **Performance Optimization**
- **Best Practices and Patterns**

What is Parallel Processing?

Parallel Processing is the simultaneous execution of multiple computations to solve a problem faster.

Key Characteristics:

- Multiple CPU cores
- True simultaneity
- Data decomposition
- Independent tasks

Benefits:

- Faster execution
- Better resource utilization
- Scalability
- Throughput improvement

Parallel vs Concurrent

Aspect	**Parallel**	**Concurrent**
Execution	Simultaneous	Interleaved
Hardware	Multiple cores	Single/Multiple cores
Goal	Speed up computation	Manage multiple tasks
Example	Matrix multiplication	Web server handling requests
Complexity	Data decomposition	Synchronization

In Scala 3: We can achieve both through functional programming patterns!

Sequential vs Parallel Example

```scala
// Sequential processing
val numbers = (1 to 1000000).toList
val sequential = numbers.map(_ * 2).filter(_ > 100)

// Parallel processing
val parallel = numbers.par.map(_ * 2).filter(_ > 100)

// Measuring performance
import scala.util.Random
val data = List.fill(1000000)(Random.nextInt(100))

// Sequential: ~200ms
val start1 = System.currentTimeMillis()
val result1 = data.map(math.sqrt).sum
val time1 = System.currentTimeMillis() - start1

// Parallel: ~50ms (on 4-core machine)
val start2 = System.currentTimeMillis()
val result2 = data.par.map(math.sqrt).sum
```

Parallel Collections in Scala 3

Parallel Collections Overview

Scala provides parallel collections that automatically distribute work across available CPU cores.

```
import scala.collection.parallel.CollectionConverters.
    _

// Converting to parallel collections
val list = (1 to 1000).toList.par
val vector = (1 to 1000).toVector.par

// All standard collection operations work
val result = list.map(_ * 2).filter(_ > 100).sum
```

Parallel Collection Operations

```scala
val data = (1 to 1000).par

// Map and filter operations
val processed = data.map(_ * 2).filter(_ % 2 == 0)

// Reduce operations - require associative functions
val sum = data.reduce(_ + _)
val max = data.reduce(_ max _)
```

Custom Parallel Operations

```scala
// Parallel matrix multiplication
def multiply(a: Array[Array[Int]], b: Array[Array[Int
    ]]) = {
  (0 until a.length).par.map { i =>
    (0 until b(0).length).map { j =>
      (0 until a(0).length).map(k => a(i)(k) * b(k)(j)
          ).sum
    }.toArray
  }.toArray
}

// Parallel Monte Carlo Pi estimation
def estimatePi(n: Int) = (1 to n).par
  .count(_ => math.random*math.random + math.random*
     math.random <= 1) * 4.0 / n
```

Futures and Promises

Introduction to Futures

Future represents a computation that may complete in the future.

```
import scala.concurrent.{Future, ExecutionContext}
import scala.util.{Success, Failure}

// Implicit execution context for async operations
given ExecutionContext = ExecutionContext.global

// Creating futures
val future1: Future[Int] = Future {
  Thread.sleep(1000)
  42
}

val future2: Future[String] = Future {
  "Hello, World!"
}

// Handling completion
future1.onComplete {
```

Future Composition

```scala
// Chaining futures with for-comprehension
val computation = for {
  x <- Future(10)
  y <- Future(20)
  z <- Future(x + y)
} yield z * 2

// Parallel execution with Future.sequence
val futures = List(Future(1), Future(2), Future(3))
val allResults: Future[List[Int]] = Future.sequence(
   futures)
```

Error Handling with Futures

```scala
// Recovering from failures
val riskyComputation = Future {
  if (scala.util.Random.nextBoolean()) throw new
     Exception("Error")
  else 42
}

val recovered = riskyComputation.recover {
  case _: Exception => 0
}
```

Promises

Promise is a writable, single-assignment container that completes a Future.

```scala
import scala.concurrent.Promise

// Creating a promise
val promise = Promise[Int]()
val future = promise.future

// Completing the promise
promise.success(42)
// or promise.failure(new Exception("Failed"))

// Practical example: timeout
def withTimeout[T](future: Future[T], timeout:
    Duration): Future[T] = {
  val promise = Promise[T]()

  future.onComplete(promise.tryComplete)
```

Async/Await Pattern

Async/Await in Scala 3

```scala
import scala.async.Async.{async, await}

// Traditional future composition
def traditionalWay(): Future[String] = {
  fetchUser(1).flatMap { user =>
    fetchPosts(user.id).flatMap { posts =>
      fetchComments(posts.head.id).map { comments =>
        s"User ${user.name} has ${comments.length} comments"
      }
    }
  }
}

// With async/await - more readable
def asyncAwaitWay(): Future[String] = async {
  val user = await(fetchUser(1))
  val posts = await(fetchPosts(user.id))
  val comments = await(fetchComments(posts.head.id))
```

Actor Model with Akka

Actor Model provides a higher-level abstraction for concurrent and distributed programming.

```scala
import akka.actor.typed.{ActorRef, Behavior}
import akka.actor.typed.scaladsl.Behaviors

// Define actor messages and behavior
sealed trait CounterMessage
case object Increment extends CounterMessage
case class GetValue(replyTo: ActorRef[Int]) extends
   CounterMessage

def counter(value: Int): Behavior[CounterMessage] =
  Behaviors.receive { (_, message) =>
    message match {
      case Increment => counter(value + 1)
      case GetValue(replyTo) => replyTo ! value;
         Behaviors.same
    }
  }
```

Parallel Processing with Actors

```scala
// Worker actor for parallel computation
case class ProcessChunk(data: List[Int], replyTo:
    ActorRef[Int])

def worker(): Behavior[ProcessChunk] =
  Behaviors.receive { (_, ProcessChunk(data, replyTo))
      =>
    replyTo ! data.map(_ * 2).sum
    Behaviors.same
  }

// Distribute work to multiple workers
val workers = (1 to 4).map(_ => spawn(worker())).
    toList
```

ZIO for Effect Management

Introduction to ZIO

ZIO is a functional effect system for Scala that provides powerful abstractions for concurrent and parallel programming.

```
import zio._

// Basic ZIO effects
val simpleEffect: UIO[Int] = ZIO.succeed(42)
val failingEffect: IO[String, Nothing] = ZIO.fail("Error")

// Parallel collection processing
val numbers = (1 to 1000).toList
val result = ZIO.foreachPar(numbers)(n => ZIO.succeed(n * n))
```

ZIO Fibers - Lightweight Threads

```scala
// Creating and managing fibers
val fiber1 = ZIO.succeed(expensiveComputation()).fork
val fiber2 = ZIO.succeed(anotherComputation()).fork

val parallel = for {
  f1 <- fiber1
  f2 <- fiber2
  result1 <- f1.join
  result2 <- f2.join
} yield (result1, result2)
```

ZIO Parallel Combinators

```scala
// Racing effects - first to complete wins
val raced = ZIO.succeed(slowTask()) race ZIO.succeed(fastTask())

// Parallel tuple - both must succeed
val both = ZIO.succeed(task1()) <&> ZIO.succeed(task2())

// Parallel validation with error accumulation
val validation = ZIO.validatePar(
  validateEmail("user@test.com"),
  validateAge(25)
)((email, age) => User(email, age))
```

ZIO Error Handling

```scala
// Catching and recovering from errors
val recovered = riskyOperation()
  .catchAll(error => ZIO.succeed(defaultValue))

// Retry with exponential backoff
val retried = riskyOperation()
  .retry(Schedule.exponential(1.second) && Schedule.
     recurs(3))

// Timeout operations
val withTimeout = longRunningTask().timeout(30.seconds
  )
```

ZIO Resource Management

```scala
// Automatic resource cleanup with ZIO.scoped
val fileProcessing = ZIO.scoped {
  for {
    file <- ZIO.fromAutoCloseable(ZIO.attempt(openFile("data.txt")))
    content <- ZIO.attempt(file.readAll())
    processed <- ZIO.succeed(content.toUpperCase)
  } yield processed
}

// Resources are automatically closed even on failure
```

ZIO Concurrent Data Structures

```scala
// Ref for safe concurrent state
val counter = for {
  ref <- Ref.make(0)
  _ <- ZIO.foreachParDiscard(1 to 100)(_ => ref.update
     (_ + 1))
  result <- ref.get
} yield result

// Queue for producer-consumer patterns
val queue = Queue.bounded[String](100)
val producer = queue.offer("message")
val consumer = queue.take
```

ZIO STM (Software Transactional Memory)

```scala
import zio.stm._

// Atomic transactions with STM
val transfer = for {
  from <- TRef.make(1000)
  to <- TRef.make(0)
  _ <- STM.atomically {
    from.update(_ - 100) *> to.update(_ + 100)
  }
} yield ()

// Composable and deadlock-free
```

ZIO Scheduling

```scala
// Repeat operations with schedules
val repeated = task()
  .repeat(Schedule.fixed(1.second) && Schedule.recurs
     (10))

// Complex scheduling patterns
val schedule = Schedule.exponential(100.millis)
  && Schedule.recurs(5)
  && Schedule.elapsed.whileOutput(_ < 30.seconds)

val scheduled = operation().retry(schedule)
```

ZIO Interruption and Cancellation

```scala
// Graceful interruption
val interruptible = longRunningTask()
  .onInterrupt(ZIO.succeed(println("Cleaning up...")))

// Racing with timeout
val raceWithTimeout = task() race ZIO.sleep(5.seconds)

// Uninterruptible critical sections
val critical = criticalOperation().uninterruptible
```

ZIO Testing

```scala
import zio.test._

// ZIO Test framework
val spec = test("parallel processing") {
  for {
    result <- ZIO.foreachPar(1 to 100)(n => ZIO.
        succeed(n * 2))
    expected = (1 to 100).map(_ * 2).toList
  } yield assertTrue(result == expected)
}

// Built-in test aspects for timeouts, retries, etc.
```

ZIO Layers and Dependency Injection

```scala
// Service definition
trait UserService {
  def getUser(id: Int): UIO[User]
}

// Layer providing the service
val userServiceLayer = ZLayer.succeed(new UserService
    {
  def getUser(id: Int) = ZIO.succeed(User(id, "John"))
})

// Using the service
val program = ZIO.serviceWithZIO[UserService](_.
    getUser(1))
  .provide(userServiceLayer)
```

ZIO Streaming

```scala
import zio.stream._

// Creating and processing streams
val stream = ZStream.fromIterable(1 to 1000)
  .mapZIOPar(8)(n => ZIO.succeed(n * n))
  .take(100)

// Merging streams
val merged = stream1.merge(stream2)

// Error handling in streams
val resilient = stream.catchAll(_ => ZStream.empty)
```

Functional Reactive Programming

```
import akka.stream._
import akka.stream.scaladsl._
import akka.NotUsed

// Simple stream processing
val source: Source[Int, NotUsed] = Source(1 to
    1000000)
val flow: Flow[Int, Int, NotUsed] = Flow[Int].map(_ *
    2)
val sink: Sink[Int, Future[Done]] = Sink.foreach(
    println)

val graph = source.via(flow).to(sink)

// Parallel processing with streams
val parallelFlow = Flow[Int].mapAsyncUnordered(
    parallelism = 4) { n =>
  Future {
    Thread.sleep(100) // Simulate work
```

ZIO Streams

```scala
import zio.stream._

// Creating streams
val stream1 = ZStream.fromIterable(1 to 1000000)
val stream2 = ZStream.repeatEffect(ZIO.succeed(scala.
    util.Random.nextInt()))

// Parallel processing
val parallelStream = stream1
  .mapZIOPar(8)(n => ZIO.succeed(n * n))
  .take(1000)

// Merging streams
val merged = stream1.merge(stream2)

// Grouping and windowing
val grouped = stream1
  .groupedWithin(100, 1.second)
  .map(_.sum)
```

Performance Optimization

Measuring Performance

```scala
import scala.concurrent.duration._

// Simple timing function
def time[T](block: => T): (T, Duration) = {
  val start = System.nanoTime()
  val result = block
  val end = System.nanoTime()
  (result, (end - start).nanos)
}

// Benchmarking parallel vs sequential
val data = (1 to 1000000).toList

val (seqResult, seqTime) = time {
  data.map(math.sqrt).sum
}

val (parResult, parTime) = time {
  data.par.map(math.sqrt).sum
```

Optimization Strategies

```
// 1. Choose appropriate data structures
val vector = Vector.fill(1000000)(scala.util.Random.
    nextInt())
val array = Array.fill(1000000)(scala.util.Random.
    nextInt())

// Arrays are faster for parallel operations
val vectorResult = vector.par.map(_ * 2).sum
val arrayResult = array.par.map(_ * 2).sum

// 2. Minimize object allocation
def inefficient(data: List[Int]): List[Int] =
  data.map(_ * 2).map(_ + 1).map(_ / 2)

def efficient(data: List[Int]): List[Int] =
  data.map(x => (x * 2 + 1) / 2)

// 3. Use appropriate parallelism level
val customParallelism = (1 to 1000000).par
```

Best Practices and Patterns

Common Pitfalls

```scala
// 1. Shared mutable state - AVOID
var counter = 0
(1 to 1000).par.foreach(_ => counter += 1)
// Result is unpredictable due to race conditions

// Better: Use atomic operations
import java.util.concurrent.atomic.AtomicInteger
val atomicCounter = new AtomicInteger(0)
(1 to 1000).par.foreach(_ => atomicCounter.
    incrementAndGet())

// 2. Side effects in parallel operations - AVOID
val results = mutable.ListBuffer[Int]()
(1 to 1000).par.foreach(x => results += x * 2)
// Non-thread-safe collection

// Better: Use functional approach
val results = (1 to 1000).par.map(_ * 2).toList
```

Design Patterns for Parallel Processing

```scala
// 1. Fork-Join Pattern
def parallelQuickSort[T: Ordering](arr: Array[T]):
    Array[T] = {
  if (arr.length <= 1000) {
    arr.sorted
  } else {
    val pivot = arr(arr.length / 2)
    val (left, right) = arr.partition(implicitly[
        Ordering[T]].lt(_, pivot))

    val leftFuture = Future(parallelQuickSort(left))
    val rightFuture = Future(parallelQuickSort(right))

    for {
      leftSorted <- leftFuture
      rightSorted <- rightFuture
    } yield leftSorted ++ Array(pivot) ++ rightSorted
  }
}
```

Thread Safety Patterns

```scala
// 1. Immutable Data Structures
case class ImmutableCounter(value: Int) {
  def increment: ImmutableCounter = copy(value = value
      + 1)
  def decrement: ImmutableCounter = copy(value = value
      - 1)
}

// 2. Thread-Safe Collections
import scala.collection.concurrent.TrieMap
val threadSafeMap = TrieMap[String, Int]()

// 3. Functional State Management
import cats.effect.Ref

def functionalCounter(): IO[Unit] = {
  for {
    counter <- Ref.of[IO, Int](0)
    _ <- (1 to 1000).toList.parTraverse(_ => counter.
```

Resource Management

```scala
// 1. Proper ExecutionContext management
val customEC = ExecutionContext.fromExecutor(
  java.util.concurrent.Executors.newFixedThreadPool(8)
)

// Always shutdown when done
Runtime.getRuntime.addShutdownHook(new Thread(() => {
  customEC.shutdown()
}))

// 2. Using ZIO for automatic resource management
def processFilesConcurrently(files: List[String]): ZIO
    [Any, Throwable, List[String]] = {
  ZIO.foreachPar(files) { filename =>
    ZIO.scoped {
      for {
        source <- ZIO.fromAutoCloseable(ZIO.attempt(
            scala.io.Source.fromFile(filename)))
        content <- ZIO.attempt(source.mkString)
```

Testing Parallel Code

```
import org.scalatest.flatspec.AnyFlatSpec
import org.scalatest.matchers.should.Matchers
import scala.concurrent.duration._

class ParallelProcessingSpec extends AnyFlatSpec with
    Matchers {

  "Parallel processing" should "produce correct
      results" in {
    val data = (1 to 1000).toList
    val sequential = data.map(_ * 2).sum
    val parallel = data.par.map(_ * 2).sum

    parallel shouldEqual sequential
  }

  "Future composition" should "handle errors properly"
      in {
    val future = for {
```

Performance Guidelines

Scenario	Recommended Approach	Reason
CPU-intensive tasks	Parallel collections	Utilizes all cores
I/O operations	Futures/ZIO	Non-blocking
Stream processing	Akka Streams/ZIO Streams	Backpressure
Actor systems	Akka Typed	Message passing
Complex workflows	ZIO	Composable effects
Simple parallelism	.par collections	Easy to use

Key Metrics to Monitor:

- CPU utilization
- Memory usage
- Thread pool saturation
- Garbage collection pressure
- Latency percentiles

Summary

Key Takeaways:

- **Choose the right tool:** Collections.par for simple cases, Futures for async, ZIO for complex effects
- **Embrace immutability:** Avoid shared mutable state
- **Understand your workload:** CPU-bound vs I/O-bound
- **Measure performance:** Use proper benchmarking tools
- **Handle errors gracefully:** Parallel code can fail in complex ways
- **Test thoroughly:** Parallel code has subtle bugs

ZIO Benefits:

- Composable effects and resource management
- Built-in error handling and retry mechanisms
- Powerful concurrency primitives (fibers, STM, queues)
- Excellent testing support

References and Further Reading I

[Hewitt, 1973] Hewitt, C., Bishop, P., Steiger, R. *A Universal Modular ACTOR Formalism for Artificial Intelligence*. IJCAI 1973.
Original actor model paper

[Valiant, 1990] Valiant, L. *A Bridging Model for Parallel Computation*. Communications of the ACM, 1990.
BSP model for parallel algorithms

[Prokopec, 2011] Prokopec, A., et al. *A Generic Parallel Collection Framework*. Euro-Par 2011.
Design of Scala parallel collections

[Odersky, 2014] Odersky, M., Spoon, L., Venners, B. *Programming in Scala, 3rd Edition*. Artima Press, 2014.
Comprehensive Scala parallel programming guide

[Karmani, 2009] Karmani, R., et al. *Actor Frameworks for the JVM Platform*. PPPJ 2009.
Comparison of JVM actor implementations

PARIS PANTHÉON-ASSAS UNIVERSITÉ

References and Further Reading II

[Bernstein, 2014] Bernstein, D. *Akka in Action*. Manning Publications, 2014.
Practical actor programming with Akka

[De Goes, 2019] De Goes, J. *ZIO: A Type-Safe, Composable Library for Async and Concurrent Programming*. Scala Days 2019.
Introduction to ZIO effect system

[Spiewak, 2020] Spiewak, D. *Cats Effect 3: Toward Fearless Concurrency*. Typelevel Summit 2020.
Modern functional concurrency in Scala

[Kuhn, 2015] Kuhn, R., et al. *Reactive Streams Specification*. 2015.
Standard for asynchronous stream processing

[Bonér, 2016] Bonér, J., et al. *Akka Streams Documentation*. Lightbend, 2016.
Stream processing with backpressure

[Lea, 2000] Lea, D. *Concurrent Programming in Java, 2nd Edition*. Addison-Wesley, 2000.
JVM concurrency fundamentals

References and Further Reading III

[Goetz, 2006] Goetz, B., et al. *Java Concurrency in Practice*. Addison-Wesley, 2006.
Practical concurrent programming patterns

[Shipilev, 2014] Shipilev, A. *JMH: Java Microbenchmark Harness*. Oracle, 2014.
Accurate performance measurement

[Loom, 2021] Project Loom Team *Project Loom: Fibers and Continuations for the Java Platform*. OpenJDK, 2021.
Virtual threads for massive concurrency

[GraalVM, 2022] Oracle Labs *GraalVM Native Image for Scala Applications*. Oracle, 2022.
Ahead-of-time compilation for better startup