PROYECTO FINAL - FUNDAMENTOS PROGRAMACIÓN FUNCIONAL Y CONCURRENTE: PLANIFICACIÓN DE VUELOS

Johan Sebastian Laverde Pineda -2266278
Santiago Useche Tascon - 2266200 César
David Peñaranda Melo -2266265

Mayo 2024.

Carlos Andres Delgado

Universidad del valle
Facultad de
Ingeniería
Fundamentos de
Programación
Funcional y
Concurrente

Tabla de Contenidos

Introducción

El cálculo de los itinerarios de vuelos entre distintos aeropuertos es un trabajo de computación intenso, especialmente cuando se trata con grandes conjuntos de datos que incluyen varios vuelos y aeropuertos. En el contexto de la programación funcional y concurrente, Scala nos ofrece grandes herramientas para poder abordar estos desafíos.

En este documento se mostrará como se paraleliza un código para darle un mayor rendimiento y disminuir el tiempo de ejecución. La paralelización, en particular, es una técnica muy eficaz para mejorar el rendimiento de aplicaciones que requieren un procesamiento intensivo de datos, el uso de esta permite que múltiples tareas se ejecuten simultáneamente. Se explicará el proceso de modificación del código original, detallando técnicas y herramientas utilizadas para implementar la paralelización.

1. Descripción del código original

El código original utilizado para calcular los itinerarios de vuelo se describe en esta sección. Este código se puede usar en Scala y cuenta con varias funciones clave que permiten encontrar y ordenar itinerarios basados en varios criterios, como la hora de salida, el número de escalas y el tiempo total de vuelo.

- **Estructura del Código**

  El código se organiza en una clase principal llamada ***Itinerario***, que contiene varios métodos para calcular los itinerarios de vuelos.

- **Definiciones de Tipos**

  El código utiliza alias para simplificar la lectura y manejo de listas de aeropuertos y vuelos

```
type Aeropuertos = List[Aeropuerto]
type Vuelos = List[Vuelo]
type Itinerario = List[Vuelo]
```

- **Función para Encontrar Itinerarios**

  La función ***itinerarios*** toma una lista de vuelos y una lista de aeropuertos, y devuelve una función que, a su vez, toma dos códigos de aeropuerto y devuelve una lista de posibles itinerarios entre ellos

```
def itinerarios(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String) => List[Itinerario] = { c

  // Función interna recursiva para buscar itinerarios
  def buscarItinerarios(origen: String, destino: String, visitados: Set[String]): List[Itinerario] = {
    if (origen == destino) {
      List(List())  // Caso base: si el origen es igual al destino, retornar lista vacía
    } else {
      // Filtrar vuelos que salen del aeropuerto de origen y aún no han sido visitados
      vuelos.filter(v => v.Org == origen && !visitados.contains(v.Dst)).flatMap { vuelo =>
        // Llamada recursiva para continuar buscando itinerarios desde el destino actual del vuelo
        buscarItinerarios(vuelo.Dst, destino, visitados + origen).map(vuelo :: _)
      }
    }
  }

  // Retornar función que toma dos códigos de aeropuerto y retorna la lista de itinerarios
  (cod1: String, cod2: String) => buscarItinerarios(cod1, cod2, Set.empty)
}
```

- **Funciones para Ordenar Itinerarios**

  El código también incluye varias funciones para ordenar los itinerarios según diferentes criterios:

  - **Ordenar por Tiempo Total de Vuelo**

```
def itinerariosTiempo(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String) => List[List[Vuelo]] = {

  def gmtOffset(cod: String): Double = {
    aeropuertos.find(_.Cod.equalsIgnoreCase(cod)).map(_.GMT).getOrElse(0.0)
  }

  def calcularTiempoVuelo(vuelo: Vuelo, aeropuerto: List[Aeropuerto]): Int = {
    val gmtOrg = gmtOffset(vuelo.Org)
    val gmtDst = gmtOffset(vuelo.Dst)
    val horaSalidaGMT = vuelo.HS * 60 + vuelo.MS - ((gmtOrg / 100.0) * 60).toInt
    val horaLlegadaGMT = vuelo.HL * 60 + vuelo.ML - ((gmtDst / 100.0) * 60).toInt
    val duracionVuelo = horaLlegadaGMT - horaSalidaGMT
    if (duracionVuelo < 0) duracionVuelo + 1440 else duracionVuelo
  }

  def calcularTiempoTotal(itinerario: List[Vuelo], aeropuerto: List[Aeropuerto]): Int = {
    itinerario match {
      case Nil => 0
      case vuelo :: Nil => calcularTiempoVuelo(vuelo, aeropuerto)
      case vuelo1 :: vuelo2 :: tail =>
        calcularTiempoVuelo(vuelo1, aeropuerto) +
          calcularTiempoEspera(vuelo1, vuelo2, aeropuerto) +
          calcularTiempoTotal(vuelo2 :: tail, aeropuerto)
```

```
    }
  }

  def calcularTiempoEspera(vuelo1: Vuelo, vuelo2: Vuelo, aeropuerto: List[Aeropuerto]): Int = {
    val horaLlegada = vuelo1.HL * 60 + vuelo1.ML
    val horaSalida = vuelo2.HS * 60 + vuelo2.MS
    if (horaSalida < horaLlegada) (24 * 60 - horaLlegada) + horaSalida
    else horaSalida - horaLlegada
  }

  (cod1: String, cod2: String) => {
    val allItineraries = itinerarios(vuelos, aeropuertos)(cod1, cod2)
    val allItinerariesTime = allItineraries.map(itinerary => (itinerary, calcularTiempoTotal(itinerary, aeropuertos)))
    allItinerariesTime.sortBy(_._2).map(_._1).take(3)
  }
}
```

- **Ordenar por Número de Escalas**

```
def itinerariosEscalas(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String) => List[List[Vuelo]] = {

  val aeropuertosMap = aeropuertos.map(a => (a.Cod, a)).toMap

  def encontrarItinerariosEscalas(cod1: String, cod2: String): List[List[Vuelo]] = {
    def contarEscalas(itinerario: List[Vuelo]): Int = {
      itinerario.map(_.Esc).sum + (itinerario.size - 1)
    }

    val todosItinerarios = itinerarios(vuelos, aeropuertos)(cod1, cod2)
    todosItinerarios.sortBy(contarEscalas).take(3)
  }

  (cod1: String, cod2: String) => {
    if (!aeropuertosMap.contains(cod1) || !aeropuertosMap.contains(cod2)) {
      List()
    } else {
      encontrarItinerariosEscalas(cod1, cod2)
    }
  }
}
```

- **Ordenar por Tiempo en el Aire.**

```
def itinerariosAire(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String) => List[List[Vuelo]] = {

  def calcularDuracionVuelo(vuelo: Vuelo): Int = {
    val aeropuertoOrigen = aeropuertos.find(_.Cod == vuelo.Org).get
    val aeropuertoDestino = aeropuertos.find(_.Cod == vuelo.Dst).get

    val salidaEnMinutos = vuelo.HS * 60 + vuelo.MS
    val llegadaEnMinutos = vuelo.HL * 60 + vuelo.ML

    val diferenciaGMT = (aeropuertoDestino.GMT - aeropuertoOrigen.GMT) / 100
    val diferenciaGMTEnMinutos = (diferenciaGMT * 60).toInt

    val duracionEnMinutos = llegadaEnMinutos - salidaEnMinutos - diferenciaGMTEnMinutos

    if (duracionEnMinutos < 0) duracionEnMinutos + 1440 else duracionEnMinutos
  }

  def calcularTiempoTotal(itinerario: List[Vuelo]): Int = {
    itinerario.map(calcularDuracionVuelo).sum
  }
```

```
  (cod1: String, cod2: String) => {
    def minimoAire(cod1: String, cod2: String): List[List[Vuelo]] = {
      val allItineraries = itinerarios(vuelos, aeropuertos)(cod1, cod2)
      val allItinerariesTime = allItineraries.map(it => (it, calcularTiempoTotal(it)))
      allItinerariesTime.sortBy(_._2).map(_._1).take(3)
    }

    minimoAire(cod1, cod2)
  }
}
```

- **Ordenar por Hora de Salida**

```
def itinerariosSalida(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String, Int, Int) => List[Vuelo] =

  val buscarItinerariosFn = itinerarios(vuelos, aeropuertos)

  def convertirAMinutos(hora: Int, minutos: Int): Int = {
    hora * 60 + minutos
  }

  def calcularLapsoTiempo(horaLlegada: Int, horaCita: Int): Int = {
    val diferencia = horaCita - horaLlegada
    if (diferencia >= 0) diferencia else 1440 + diferencia
  }

  def esValido(itinerario: List[Vuelo], tiempoCita: Int): Boolean = {
    val horaLlegada = convertirAMinutos(itinerario.last.HL, itinerario.last.ML)
    horaLlegada <= tiempoCita || (horaLlegada < 1440 && tiempoCita < horaLlegada)
  }

  (origen: String, destino: String, horaCita: Int, minCita: Int) => {
    val tiempoCita = convertirAMinutos(horaCita, minCita)
    val todosItinerarios = buscarItinerariosFn(origen, destino)
    val itinerariosValidos = todosItinerarios.filter(it => esValido(it, tiempoCita))
```

```
    val itinerariosOrdenados = itinerariosValidos.sortBy { it =>
      val horaLlegada = convertirAMinutos(it.last.HL, it.last.ML)
      val lapsoTiempo = calcularLapsoTiempo(horaLlegada, tiempoCita)
      val horaSalida = convertirAMinutos(it.head.HS, it.head.MS)
      (lapsoTiempo, horaSalida)
    }

    itinerariosOrdenados.headOption.getOrElse(List.empty)
  }
 }
}
```

- **Conclusión**

  El código original proporciona una base sólida para el cálculo de itinerarios de vuelos utilizando diferentes criterios de ordenación. Sin embargo, al ser una implementación secuencial, puede no ser eficiente para grandes volúmenes de datos. En la siguiente sección, se presentará cómo se implementaron técnicas de paralelización para mejorar el rendimiento de este código.

2. Paralelización del Código

En esta sección se explorará cómo se implementó la paralelización del código original para mejorar el rendimiento y reducir el tiempo de ejecución. La paralelización se llevó a cabo principalmente en las operaciones que involucran el procesamiento intensivo de datos, como la búsqueda de itinerarios y el cálculo de tiempos.

- ### Modificaciones para la Paralelización

  - #### Uso de Colecciones Paralelas

    En lugar de utilizar colecciones estándar como ***List***, se utilizó ***ParSeq*** de Scala para aprovechar el procesamiento paralelo:

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

    Las listas de vuelos y aeropuertos se convirtieron en ***ParSeq*** para habilitar la paralelización en las operaciones de filtrado y mapeo.

  - #### Paralelización de la Búsqueda de Itinerarios

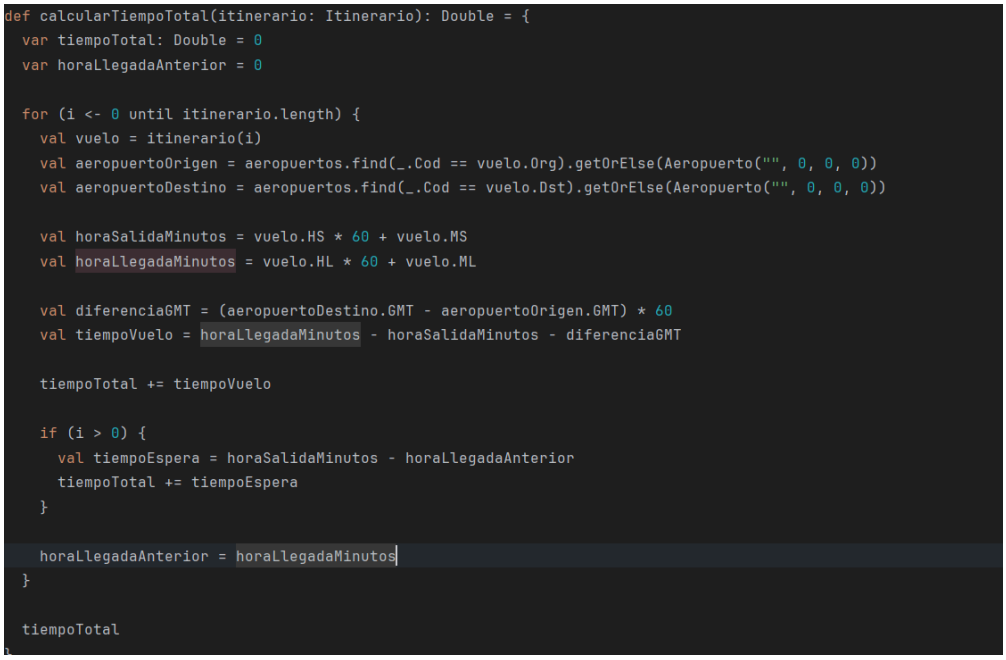
    La función ***buscarItinerarios*** fue modificada para utilizar paralelismo en la búsqueda de itinerarios entre dos aeropuertos:

```
def buscarItinerarios(origen: String, destino: String, visitados: Set[String]): List[Itinerario] = {
  if (origen == destino) {
    List(List())
  } else {
    vuelos.par.filter(v => v.Org == origen && !visitados.contains(v.Dst)).flatMap { vuelo =>
      buscarItinerarios(vuelo.Dst, destino, visitados + origen).map(vuelo :: _)
    }.toList
  }
}
buscarItinerarios(cod1, cod2, Set())
```

    Aquí, ***vuelos.par*** permite que el filtrado y el mapeo se realicen en paralelo, mejorando así la eficiencia para grandes conjuntos de datos de vuelos.

- **Paralelización en el Cálculo de Tiempos**

  Las funciones que calculan tiempos, como ***calcularTiempoTota***l y ***calcularDuracionVuelo***, operan eficientemente con ***ParSeq*** para reducir el tiempo de cálculo:



```
def calcularTiempoTotal(itinerario: Itinerario): Double = {
  var tiempoTotal: Double = 0
  var horaLlegadaAnterior = 0

  for (i <- 0 until itinerario.length) {
    val vuelo = itinerario(i)
    val aeropuertoOrigen = aeropuertos.find(_.Cod == vuelo.Org).getOrElse(Aeropuerto("", 0, 0, 0))
    val aeropuertoDestino = aeropuertos.find(_.Cod == vuelo.Dst).getOrElse(Aeropuerto("", 0, 0, 0))

    val horaSalidaMinutos = vuelo.HS * 60 + vuelo.MS
    val horaLlegadaMinutos = vuelo.HL * 60 + vuelo.ML

    val diferenciaGMT = (aeropuertoDestino.GMT - aeropuertoOrigen.GMT) * 60
    val tiempoVuelo = horaLlegadaMinutos - horaSalidaMinutos - diferenciaGMT

    tiempoTotal += tiempoVuelo

    if (i > 0) {
      val tiempoEspera = horaSalidaMinutos - horaLlegadaAnterior
      tiempoTotal += tiempoEspera
    }

    horaLlegadaAnterior = horaLlegadaMinutos
  }

  tiempoTotal
}
```

- **Beneficios de la Paralelización**

  La paralelización de estas operaciones críticas en el código original proporcionó varios beneficios significativos:

  - **Reducción del Tiempo de Ejecución**

    Las operaciones paralelizadas ejecutan tareas simultáneamente en múltiples núcleos de CPU, lo que lleva a una reducción en el tiempo total de ejecución del código.

  - **Mejora del Rendimiento**

    El código puede manejar grandes volúmenes de datos de manera más eficiente al distribuir las tareas en paralelo, aprovechando mejor los recursos computacionales disponibles.

- **Escalabilidad**

  La implementación paralela es más escalable, permitiendo que el código mantenga un buen rendimiento a medida que crece el tamaño de los conjuntos de datos.

3. Conclusión

La paralelización del código proporciona una estrategia efectiva para mejorar el rendimiento y reducir significativamente el tiempo de ejecución en aplicaciones que manejan grandes volúmenes de datos. En este documento, hemos explorado cómo se implementó la paralelización en un sistema de gestión de itinerarios aéreos utilizando Scala.

Al revisar el código original y su versión paralelizada, se observa claramente cómo la introducción de colecciones paralelas (ParSeq) y el uso de operaciones paralelas en las tareas intensivas en computación, como la búsqueda de itinerarios y el cálculo de tiempos de vuelo, han optimizado el rendimiento del sistema..

Los beneficios obtenidos son notables: una reducción significativa en el tiempo de ejecución, una mejor utilización de recursos computacionales al aprovechar múltiples núcleos de CPU simultáneamente, y una capacidad mejorada para manejar grandes conjuntos de datos de manera eficiente y escalable..

Es crucial destacar que la paralelización no solo mejora el rendimiento, sino que también implica consideraciones cuidadosas en términos de diseño, optimización y pruebas exhaustivas para garantizar la correcta funcionalidad y el comportamiento consistente del sistema en diferentes condiciones.