INFORME DEL PROYECTO FINAL (PARALELIZACIÓN)

ANDERSON GÓMEZ GARCÍA - 202266242
KEVIN ALEXIS LORZA RAMÍREZ - 202266098
JUAN DAVID PÉREZ VALENCIA - 202266289

FUNDAMENTOS DE PROGRAMACIÓN FUNCIONAL Y CONCURRENTE
UNIVERSIDAD DEL VALLE
TULUÁ - VALLE DEL CAUCA
21 DE JUNIO DE 2024

Índice

Función de itinerarios “paralelizada”

La función no es viable aplicarle una paralelización:

```
def itinerariosPar(vuelos: List[Vuelo], aeropuertos:
List[Aeropuerto]): (String, String) => List[List[Vuelo]] = {
    def buscarItinerarios(origen: String, destino: String,
visitados: Set[String] = Set()): List[List[Vuelo]] = {
      origen match {
        case o if o == destino => List(List())
        case o if visitados.contains(o) => List()
        case _ =>
          val vuelosDesdeOrigen = vuelos.filter(_.Org ==
origen)
          val nuevasVisitados = visitados + origen

          vuelosDesdeOrigen.flatMap { vuelo =>
            val rutasDesdeDestino =
buscarItinerarios(vuelo.Dst, destino, nuevasVisitados)
            rutasDesdeDestino.map(ruta => vuelo :: ruta)
          }
      }
    }
    (cod1: String, cod2: String) => buscarItinerarios(cod1,
cod2)
  }
```

Se debe a que su funcionamiento base, es puramente secuencial, al ir recorriendo una lista y validando casos una y otra vez, la mayoría de los pasos son dependientes del anterior, por lo que una paralelización no es viable en costos computacionales, por esta razón tomamos exactamente el mismo código del sequencial.

Función de itinerarios por tiempo paralelizada

Inicio de la función:

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

Creamos una Función para obtener el atributo GMT de un aeropuerto:

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

Declaramos una función para calcular el tiempo de vuelo en minutos, considerando la diferencia de GMT:

```
  def calcularTiempoVuelo(vuelo: Vuelo, aeropuertos:
List[Aeropuerto]): Int = {
    val gmtOrg = gmtOffset(vuelo.Org)
    val gmtDst = gmtOffset(vuelo.Dst)
    val horaSalidaGMT = vuelo.HS * 60 + vuelo.MS - ((gmtOrg /
100) * 60).toInt
    val horaLlegadaGMT = vuelo.HL * 60 + vuelo.ML - ((gmtDst
/ 100) * 60).toInt
    val duracionVuelo = horaLlegadaGMT - horaSalidaGMT
    if (duracionVuelo < 0) duracionVuelo + 1440 else
duracionVuelo
  }
```

Definimos una función que calcula el tiempo de espera entre dos vuelos en minutos:

```
def calcularTiempoEspera(vuelo1: Vuelo, vuelo2: Vuelo,
aeropuertos: 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)
  }
```

Creamos una función para calcular el tiempo total de un itinerario de vuelos:

```
def calcularTiempoTotal(itinerario: List[Vuelo], aeropuertos:
List[Aeropuerto]): Int = {
    itinerario.zipWithIndex.map { case (vuelo, index) =>
      calcularTiempoVuelo(vuelo, aeropuertos) + (if (index <
itinerario.length - 1) calcularTiempoEspera(vuelo,
itinerario(index + 1), aeropuertos) else 0)
    }.sum
  }
```

Declaramos una función que retorna los tres itinerarios con menor tiempo total de viaje:

```
  (cod1: String, cod2: String) => {
    val allItineraries = itinerariosPar(vuelos,
aeropuertos)(cod1, cod2)
```

Convertimos la colección a una colección paralela para procesamiento en paralelo:

```
  val allItinerariesPar = allItineraries.par
```

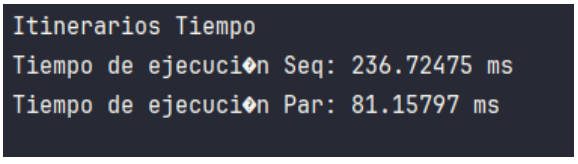
Calculamos el tiempo total de cada itinerario de forma paralelizada:

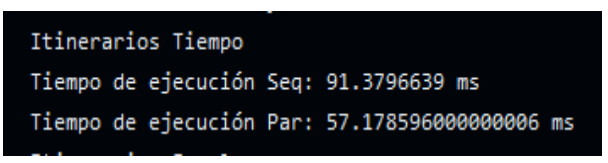
```
 val allItinerariesTime = allItinerariesPar.map(itinerary =>
(itinerary, calcularTiempoTotal(itinerary, aeropuertos)))
```

Finalmente, Convertimos a lista, ordenamos por tiempo total y seleccionamos los primeros 3:

```
 allItinerariesTime
      .toList
      .sortBy(_._2)
      .map(_._1)
      .take(3)
  }
}
```

Ganancia de la paralelización Itinerarios Tiempo :





Función de itinerarios por escalas paralelizada

Inicio de la función

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

Creamos una función que calcula el número de escalas en un itinerario:

```
  def escalas(itinerario: List[Vuelo]): Int = {
    (itinerario.map(_.Esc).sum) + (itinerario.size - 1)
  }
```

Declaramos un función que toma los códigos de origen y destino, y retorna una lista de itinerarios con menos escalas:

```
  (origen: String, destino: String) => {
```

Llamamos a la función itinerariosPar para encontrar todos los itinerarios posibles entre el origen y el destino:

```
 val itinerariosEncontrados = itinerariosPar(vuelos,
aeropuertos)(origen, destino)
```

Convierte la colección de itinerarios a una colección paralela para procesamiento en paralelo

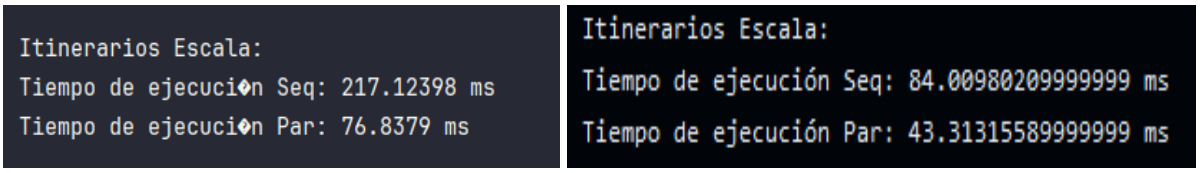
```
val itinerariosParalelos = itinerariosEncontrados.par
```

Usamos map para calcular las escalas en paralelo, y luego convertimos a una lista

```
    itinerariosParalelos.map(itinerario => (itinerario,
escalas(itinerario))) // Para cada itinerario, calculamos el
número de escalas
      .toList // Convertimos la colección paralela a una
lista para usar sortBy
      .sortBy(_._2) // Ordenamos la lista por el número de
escalas
      .map(_._1) // Extraemos solo los itinerarios,
descartando el número de escalas
      .take(3) // Tomamos los primeros 3 itinerarios (con
menos escalas)
  }
```

```
}
```

Ganancia de la paralelización Itinerarios Escala:



Función de itinerarios por Aire paralelizada

Inicio de la función

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

Devuelve el GMT del aeropuerto con el código dado, o 0.0 si no se encuentra.

```
    def gmtOffset(cod: String): Double = {

aeropuertos.find(_.Cod.equalsIgnoreCase(cod)).map(_.GMT).getO
rElse(0.0)
    }
```

Calcula la duración de un vuelo ajustando por diferencias horarias GMT.

```
   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) * 60).toInt
        val horaLlegadaGMT = (vuelo.HL * 60) + vuelo.ML -
((gmtDst / 100) * 60).toInt
        val duracionVuelo = horaLlegadaGMT - horaSalidaGMT
        if (duracionVuelo < 0) duracionVuelo + 1440 else
duracionVuelo
    }
```

Calcula el tiempo total de un itinerario sumando la duración de todos los vuelos.

```
    def calcularTiempoTotal(itinerario: List[Vuelo],
aeropuerto: List[Aeropuerto]): Int = {
```

```
        itinerario.map(vuelo => calcularTiempoVuelo(vuelo,
aeropuerto)).sum
    }
```

Encuentra todos los itinerarios posibles entre dos aeropuertos

```
    (cod1: String, cod2: String) => {
        .
        val allItineraries = Await.result(Future {
itinerariosPar(vuelos, aeropuertos)(cod1, cod2) },
Duration.Inf)
```

Calcula los tiempos totales de los itinerarios en paralelo.

```
        val calcularTiemposFuturas =
Future.traverse(allItineraries) { itinerario =>
            Future { calcularTiempoTotal(itinerario,
aeropuertos) }
        }
```

Asocia cada itinerario con su tiempo total y los ordena por tiempo.

```
        val allItinerariesWithTimes =
Await.result(calcularTiemposFuturas,
Duration.Inf).zip(allItineraries)
```

Devuelve los tres itinerarios más cortos en tiempo.

```

allItinerariesWithTimes.sortBy(_._1).map(_._2).take(3)
    }
}
```

Ganancia de paralelización Itinerarios Aire:

```
Itinerarios Aire:
Tiempo de ejecuci◆n Seq: 631.9499300000001 ms
Tiempo de ejecuci◆n Par: 125.28050500000002 ms
```

```
Itinerarios Aire:
Tiempo de ejecución Seq: 532.5986762 ms
Tiempo de ejecución Par: 54.86846865 ms
```

Función de itinerarios por salida paralelizada

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

Creamos una función que encuentra todos los itinerarios posibles entre dos aeropuertos:

```
  val buscarItinerariosFn = itinerariosPar(vuelos,
aeropuertos)
```

Definimos una función que convierte horas y minutos a unívocamente minutos:

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

Declaramos una función que calcula el tiempo que hay entre la hora de llegada y la hora de la cita:

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

Creamos una función que verifica si un itinerario es válido en función de la hora de llegada y la hora de la cita:

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

Definimos una función que toma los códigos de origen y destino, y la hora de la cita, y devuelve un itinerario

```
  (origen: String, destino: String, horaCita: Int, minCita:
Int) => {
```

declaramos una variable que convierte la hora de la cita a minutos totales:

```
val tiempoCita = convertirAMinutos(horaCita, minCita)
```

Encuentra todos los itinerarios posibles en un futuro:

```
 val todosItinerariosFuturo = Future {
buscarItinerariosFn(origen, destino) }
```

Filtramos los itinerarios válidos en función de la hora de la cita:

```
 val itinerariosValidosFuturo =
todosItinerariosFuturo.flatMap { todosItinerarios =>
      Future.sequence(todosItinerarios.map(it => Future {
        if (esValido(it, tiempoCita)) Some(it) else None
      })).map(_.flatten)
    }
```

Ordenamos los itinerarios válidos por el tiempo hasta la cita y la hora de salida:

```
 val itinerariosOrdenadosFuturo =
itinerariosValidosFuturo.flatMap { itinerariosValidos =>
      Future.sequence(itinerariosValidos.map { it =>
        Future {
          val horaLlegada = convertirAMinutos(it.last.HL,
it.last.ML)
          val lapsoTiempo = calcularLapsoTiempo(horaLlegada,
tiempoCita)
          val horaSalida = convertirAMinutos(it.head.HS,
it.head.MS)
          (it, lapsoTiempo, horaSalida)
        }
      })
    }.map { itinerariosOrdenados =>
      itinerariosOrdenados.sortBy { case (_, lapsoTiempo,
horaSalida) =>
```

```
          (lapsoTiempo, horaSalida)
        }
      }
```

Espera a que se completen los futuros y devuelve el itinerario con la mejor combinación:

```
Await.result(itinerariosOrdenadosFuturo.map(_.headOption.map(
_._1).getOrElse(List.empty)), Duration.Inf)
  }
}
```

Ganancia de la paralelización de Itinerarios Salida:

```
Itinerarios Salida:
Tiempo de ejecuci◆n Seq: 185.84660000000002 ms
Tiempo de ejecuci◆n Par: 108.061945 ms
```

```
Itinerarios Salida:
Tiempo de ejecución Seq: 78.72394915 ms
Tiempo de ejecución Par: 68.1889572 ms
```

Ejecución del benchmark de forma local:

```
Itinerarios Tiempo
Tiempo de ejecuci◆n Seq: 236.72475 ms
Tiempo de ejecuci◆n Par: 81.15797 ms

Itinerarios Escala:
Tiempo de ejecuci◆n Seq: 217.12398 ms
Tiempo de ejecuci◆n Par: 76.8379 ms

Itinerarios Aire:
Tiempo de ejecuci◆n Seq: 631.9499300000001 ms
Tiempo de ejecuci◆n Par: 125.28050500000002 ms

Itinerarios Salida:
Tiempo de ejecuci◆n Seq: 185.84660000000002 ms
Tiempo de ejecuci◆n Par: 108.061945 ms

Itinerarios base:
Tiempo de ejecuci◆n Seq: 84.45916500000001 ms
Tiempo de ejecuci◆n Par: 89.128015 ms
```

Ejecución del benchmark de forma virtual en github:

```
16  Itinerarios Tiempo
17  Tiempo de ejecución Seq: 91.3796639 ms
18  Tiempo de ejecución Par: 57.178596000000006 ms
20  Itinerarios Escala:
21  Tiempo de ejecución Seq: 84.00980209999999 ms
22  Tiempo de ejecución Par: 43.31315589999999 ms
24  Itinerarios Aire:
25  Tiempo de ejecución Seq: 532.5986762 ms
26  Tiempo de ejecución Par: 54.86846865 ms
28  Itinerarios Salida:
29  Tiempo de ejecución Seq: 78.72394915 ms
30  Tiempo de ejecución Par: 68.1889572 ms
32  Itinerarios base:
33  Tiempo de ejecución Seq: 47.1506627 ms
34  Tiempo de ejecución Par: 50.15712095000001 ms
```

TABLA COMPARATIVA				
	GITHUB		**COMPUTADOR LOCAL**	
Itinerario	**Sequencial (ms)**	**Paralelo (ms)**	**Sequencial(ms)**	**Paralelo (ms)**
Itinerario Tiempo:	91.38	57.18	236.724	81.158
Itinerario Escala:	84.009	43.313	217.124	76.838
Itinerario Aire:	532.598	54.868	631.950	125.280
Itinerario Salida:	78.723	68.188	185.846	108.061
Itinerario base:	47.150	50.157	84.460	89.128

La tabla anterior tiene datos aproximados, estos tienen un margen de variación que se puede observar en el itinerario base

Grafico Comparativo Computador local:

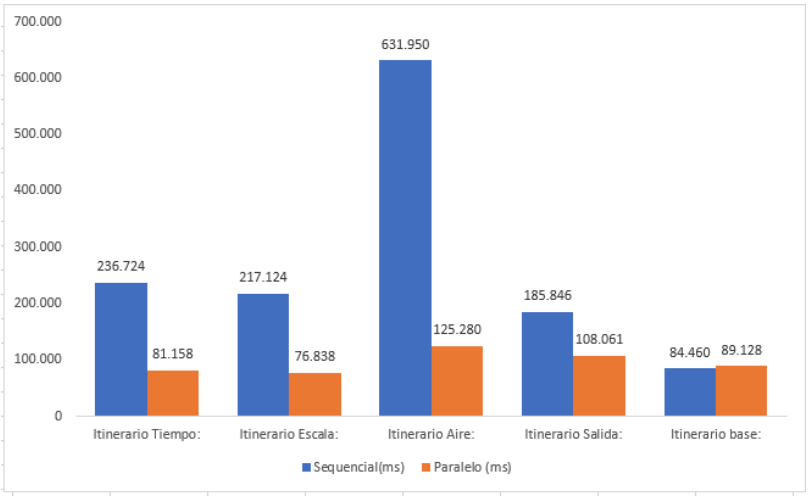
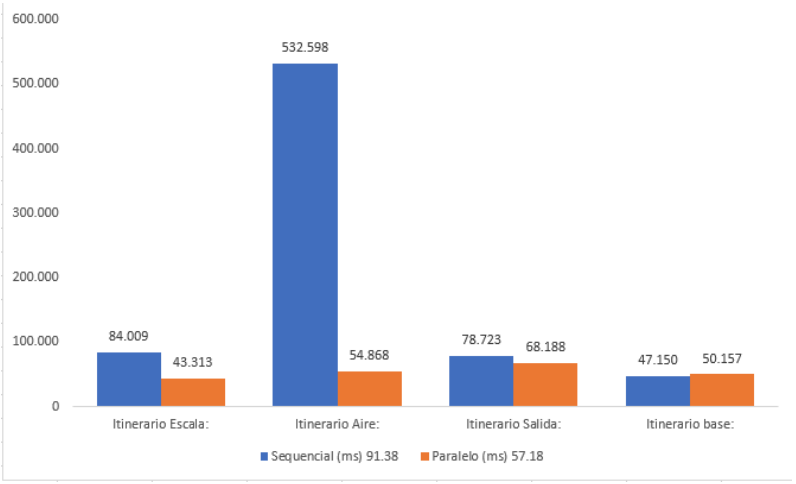


Grafico comparativo Github



Conclusiones

Aunque en general hubo una mejora en el tiempo de ejecución con la paralelización, la magnitud de esta mejora varió entre las diferentes funciones de itinerarios (tiempo, escalas, aire, salida). Algunas funciones mostraron una reducción significativa en los tiempos, mientras que otras tuvieron mejoras más modestas. Esto resalta la importancia de evaluar la idoneidad de la paralelización para cada tarea específica y contexto de datos.

Los resultados obtenidos en un entorno local versus un entorno virtual (GitHub) mostraron variaciones en los tiempos de ejecución paralelos. Esto puede atribuirse a diferencias en la infraestructura de hardware, la gestión de recursos y la carga de red. Es crucial considerar estas variaciones al implementar y evaluar la paralelización en diferentes entornos.

La función de itinerarios por salida mostró una mayor variabilidad en los tiempos paralelos entre los dos entornos evaluados. Esto puede indicar que algunas operaciones paralelizadas pueden ser más sensibles a factores externos como la latencia de red o la gestión de recursos, lo cual es importante tener en cuenta al planificar implementaciones en entornos distribuidos o en la nube.

A pesar de las mejoras observadas con la paralelización, es esencial realizar una evaluación continua y ajustes finos en la implementación para optimizar aún más el rendimiento. Esto incluye monitorear y ajustar parámetros como el número de hilos paralelos, el tamaño de los datos procesados en paralelo y la gestión de la concurrencia para maximizar los beneficios de la paralelización.