Informe Proyecto Final Paralelización

Integrantes:

- Juan Esteban Torres Medina – 2266007
- Jesus Ediber Arenas Guzmán – 2266066

Docente:

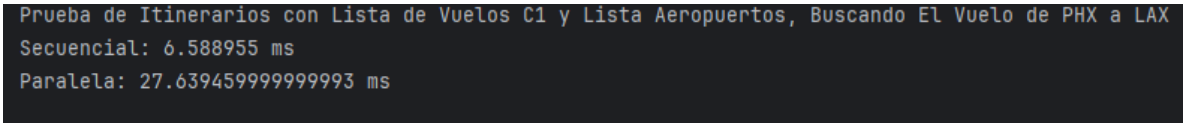
Carlos Andrés Delgado Saavedra

Universidad del valle, sede tuluá

2024

• ItinerariosPar:
Al intentar paralelizar la función Itinerariospar, observamos que su naturaleza

mayormente secuencial resulta en un empeoramiento de los tiempos de ejecución. Esto se evidencia claramente en la imagen adjunta.




```scala
def itinerariosPar(vuelos: List[Vuelo], aeropuertos: List[Aeropuerto]): (String, String) => List[List[Vuelo]] = {  JuanTowers14 +1
  // Recibe una lista de vuelos y aeropuertos
  // Retorna una función que recibe los códigos de dos aeropuertos
  // Retorna todos los itinerarios posibles de cod1 a cod2
  def encontrarRutas(origen: String, destino: String, visitados: Set[String], rutaActual: List[Vuelo]): Future[List[List[Vuelo]]] = {
    if (origen == destino) {
      Future.successful(List(rutaActual))
    } else {
      val futuros: List[Future[List[List[Vuelo]]]] = vuelos.filter(v => v.Org == origen && !visitados.contains(v.Dst)).map { vuelo =>
        encontrarRutas(vuelo.Dst, destino, visitados + origen, rutaActual :+ vuelo)
      }
      Future.sequence(futuros).map(_.flatten)
    }
  }

  (aeropuertoOrigen: String, aeropuertoDestino: String) => {
    // Espera el resultado de los futuros para retornarlos como una lista sin Future
    val futureResult = encontrarRutas(aeropuertoOrigen, aeropuertoDestino, Set(), List())
    Await.result(futureResult, Duration.Inf)
  }
}
```

Para abordar este problema, aplicamos la paralelización mediante el uso de Future en el llamado recursivo de la función encontrar_rutas. De este modo, intentamos ejecutar en paralelo cada invocación recursiva. Sin embargo, los resultados mostraron que no hubo una mejora en los tiempos de ejecución.

Probamos dos métodos de paralelización: .par y future. Ambos métodos resultaron en un aumento del tiempo de ejecución, como se muestra en la captura proporcionada. Dado que la paralelización no produjo los resultados esperados y, de hecho, empeoró el rendimiento, decidimos mantener la función itinerariospar sin paralelizar.

• ItinerariosTiempo:

Para la función ItinerariosTiempo, decidimos aplicar la paralelización utilizando future en la función auxiliar calculartiempototal.

Tuvimos que crear dos nuevas variables que representan las secuencias a ejecutar en paralelo. Al final, usamos await en estas variables para asegurarnos de que la ejecución espere indefinidamente hasta obtener los resultados.

Además, utilizamos Future para ejecutar la función en paralelo para cada ruta en la lista. De esta forma, al final, usamos await en cada resultado para recibirlos y procesarlos correctamente.

```
def calcularTiempoTotal(ruta: List[Vuelo]): Future[Int] = Future {
  val tiemposDeVueloFut = Future.sequence(ruta.map(vuelo => Future(calcularDuracionVuelo(vuelo))))
  val tiemposDeEsperaFut = Future.sequence(ruta.zip(ruta.tail).map { case (v1, v2) => Future(calcularTiempoEspera(v1, v2)) })

  val tiemposDeVuelo = Await.result(tiemposDeVueloFut, Duration.Inf)
  val tiemposDeEspera = Await.result(tiemposDeEsperaFut, Duration.Inf)

  tiemposDeVuelo.sum + tiemposDeEspera.sum
}
```

```
(aeropuertoOrigen: String, aeropuertoDestino: String) => {
  val itinerarios = buscarItinerarios(aeropuertoOrigen, aeropuertoDestino)
  val itinerariosConTiempoFut = Future.sequence(itinerarios.map { ruta =>
    calcularTiempoTotal(ruta).map(tiempo => (ruta, tiempo))
  })

  val itinerariosConTiempo = Await.result(itinerariosConTiempoFut, Duration.Inf)
  itinerariosConTiempo.sortBy(_._2).take(3).map(_._1)
}
```

Prueba lista vuelos tamaño 100:

```
Prueba de ItinerariosTiempo con Lista de Vuelos C1 y Lista Aeropuertos, Buscando El Vuelo de PHX a LAX
Secuencial: 47.833825 ms
Paralela: 18.527459999999998 ms
```

Podemos evidenciar una mejora significativa en el rendimiento en comparación con su versión secuencial.

•ItinerariosEscala:

Para itinerarios tiempo decidimos usar para paralelizar “future” directamente en el filtro final

```
(aeropuertoOrigen: String, aeropuertoDestino: String) => {
  val futureItinerarios = Future {
    buscarItinerarios(aeropuertoOrigen, aeropuertoDestino)
  }
  val futureResultados = futureItinerarios.flatMap { itinerarios =>
    val futureRutas = itinerarios.map { ruta =>
      Future {
        (ruta, calcularEscalas(ruta))
      }
    }
    Future.sequence(futureRutas).map { rutasConEscalas =>
      rutasConEscalas.sortBy(_._2).take(3).map(_._1)
    }
  }
  Await.result(futureResultados, Duration.Inf)
}
```

Se crea el future para aplicar el map a las rutas de manera paralelas para así poder tener una ganancia Se crea el future para aplicar el map a las rutas de manera paralela, logrando así una mejora significativa.

Prueba lista vuelos tamaño 100:

```
Prueba de ItinerariosEscalas con Lista de Vuelos C1 y Lista Aeropuertos, Buscando El Vuelo de PHX a LAX
Secuencial: 9.770935 ms
Paralela: 9.192865 ms
```

Podemos evidenciar una pequeña mejora en comparación con su parte secuencial. Esto se debe a que su funcionamiento no tiene un costo computacional muy alto.

• ItinerarioEscalas:

Para itinerarios tiempo decidimos usar para paralelizar “future” directamente en el filtro final.

```
  def calcularTiempoTotalDeVuelo(ruta: List[Vuelo], aeropuertos: List[Aeropuerto]): Int = {
    ruta.map(vuelo => calcularDuracionVuelo(vuelo, aeropuertos)).sum
  }
  (aeropuertoOrigen: String, aeropuertoDestino: String) => {
    val futureItinerarios = Future {
      buscarItinerarios(aeropuertoOrigen, aeropuertoDestino)
    }
    val futureResultados = futureItinerarios.flatMap { itinerarios =>
      val futureRutas = itinerarios.map { ruta =>
        Future {
          (ruta, calcularTiempoTotalDeVuelo(ruta, aeropuertos))
        }
      }
      Future.sequence(futureRutas).map { rutasConDuracion =>
        rutasConDuracion.sortBy(_._2).take(3).map(_._1)
      }
    }
    Await.result(futureResultados, Duration.Inf)
  }
}
```

Se crea el future para aplicar el map a las rutas de manera paralelas para así poder tener una ganancia

Prueba lista vuelos tamaño 100:

```
Prueba de ItinerariosAire con Lista de Vuelos C1 y Lista Aeropuertos, Buscando El Vuelo de PHX a LAX
Secuencial: 51.736715000000004 ms
Paralela: 14.272915000000001 ms
```

Podemos evidenciar una mejora muy alta en comparación de su parte secuencial.

• ItinerarioSalida:

Para itinerarios tiempo decidimos usar para paralelizar “future” directamente en el filtro final.

```
(origen: String, destino: String, horaCita: Int, minCita: Int) => {
  val tiempoCita = convertirAMinutos(horaCita, minCita)
  val todosItinerariosFuturo = Future { buscarItinerariosFn(origen, destino) }

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

  val itinerariosOrdenadosFuturo = itinerariosValidosFuturo.flatMap { itinerariosValidos =>
    Future.sequence(itinerariosValidos.map { it =>
      Future {
        val horaLlegada = calcularHoraLlegadaTotal(it)
        val lapsoTiempo = calcularLapsoTiempo(horaLlegada, tiempoCita)
        (it, lapsoTiempo, calcularHoraSalidaTotal(it))
      }
    })
  }.map { itinerariosOrdenados =>
    itinerariosOrdenados.sortBy { case (_, lapsoTiempo, horaSalida) =>
      (lapsoTiempo, horaSalida)
    }
  }

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

Prueba lista de vuelo tamaño 100:

```
Prueba de ItinerariosAire con Lista de Vuelos C1 y Lista Aeropuertos, Buscando El Vuelo de ATL hasta DFW con una cita a las 15:10
Secuencial: 1.5940950000000005 ms
Paralela: 0.7912250000000001 ms
```

Este código implementa el paralelismo al utilizar la clase Future para realizar operaciones asíncronas. En particular, usa Future.sequence para manejar múltiples futuros y flatMap para encadenar las operaciones, lo que permite procesar itinerarios en paralelo. Este enfoque mejora la eficiencia al permitir que múltiples cálculos se realicen simultáneamente, en lugar de secuencialmente, lo que resulta en una reducción significativa del tiempo total de ejecución para la búsqueda y cálculo de itinerarios válidos y ordenados.

Conclusiones

- Itinerariostiempopar:

  la paralelización aquí se logra utilizando Futures para ejecutar cálculos intensivos (duración de vuelos y tiempos de espera) de manera concurrente, mejorando así la eficiencia y el tiempo de respuesta de la función itinerariosTiempoPar al encontrar itinerarios rápidos entre aeropuertos.

- ItinerariosEscalaspar:

  En el código de ItinerariosEscalaspar, la utilización de paralelización se enfoca en optimizar el cálculo del número de escalas en múltiples rutas de vuelo entre dos aeropuertos específicos. Al emplear Futures y future.sequence, se ejecutan los cálculos de manera concurrente para calcular las escalas en cada ruta de vuelo de forma eficiente. Esto permite obtener rápidamente los itinerarios con menos escalas, mejorando significativamente el tiempo de respuesta y la eficiencia del proceso de búsqueda de itinerarios basado en escalas.

- itinerariosAirePar:

  En la función itinerariosAirePar, la paralelización se utiliza para optimizar el cálculo de los itinerarios de vuelo que minimizan el tiempo entre dos aeropuertos específicos. Al emplear Futures y Future.sequence, se ejecutan simultáneamente los cálculos para determinar la duración total de vuelo de cada ruta posible. Esto permite obtener rápidamente los tres mejores itinerarios

basados en el tiempo total de vuelo, mejorando considerablemente la eficiencia y el tiempo de respuesta del proceso de búsqueda de itinerarios aéreos.

- ItinerariosSalidaPar:

  La implementación de paralelización en la función itinerariosSalidaPar ha sido crucial para mejorar la eficiencia y el rendimiento en la búsqueda de itinerarios de vuelo entre aeropuertos, enfocándose en criterios específicos de tiempo de salida y llegada. Mediante el uso de Futures y Future.sequence, se ha logrado procesar simultáneamente múltiples itinerarios, reduciendo significativamente el tiempo total requerido para obtener resultados válidos. Esta estrategia no solo optimiza la respuesta ante solicitudes de búsqueda, sino que también asegura una mejor experiencia de usuario al proporcionar resultados precisos y relevantes de manera más rápida.