INFORME: TALLER 3

PROGRAMACIÓN FUNCIONAL Y CONCURRENTE

JUAN DIEGO ESCOBAR TRIVIÑO - 2359393

GABRIEL URAZA GARCIA - 2359594

INGENIERIA DE SISTEMAS – 3743

CARLOS ANDRES DELGADO SAAVEDRA

DOCENTE

UNIVERSIDAD DEL VALLE

9/12/2024

TULUA VALLE DEL CAUCA

ÍNDICE GEENERAL

1.INFORME DE PROCESOS:

Multiplicación standard secuencial (def multMatriz)

```
taller.App.main(App.scala)
java.base/java.lang.Thread.getStackTrace(Thread.java:1619)
taller.MatrixParallel.multMatriz(MatrixParallel.scala:42)
taller.MatrixParallel.StrassenParallel(MatrixParallel.scala:241)
taller.App$.$anonfun$benchmarking$6(App.scala:68)
taller.Benchmark.$anonfun$compararAlgoritmo$2(Benchmark.scala:22)
org.scalameter.MeasureBuilder.$anonfun$measured$1(MeasureBuilder.scala:66)
org.scalameter.MeasureBuilder.$anonfun$measuredWith$4(MeasureBuilder.scala:54)
org.scalameter.Measurer$Default.measure(Measurer.scala:133)
org.scalameter.MeasureBuilder.measuredWith(MeasureBuilder.scala:54)
org.scalameter.MeasureBuilder.measured(MeasureBuilder.scala:66)
org.scalameter.MeasureBuilder.measure(MeasureBuilder.scala:63)
taller.Benchmark.compararAlgoritmo(Benchmark.scala:22)
taller.App$.$anonfun$benchmarking$4(App.scala:68)
scala.runtime.java8.JFunction1$mcVI$sp.apply(JFunction1$mcVI$sp.scala:18)
scala.collection.immutable.Range.map(Range.scala:59)
taller.App$.benchmarking(App.scala:64)
taller.App$.main(App.scala:44)
```

taller.App.main(App.scala) -> Este es el punto de entrada del programa. La función main inicializa la ejecución de la aplicación.

Java.base/java.lang.Thread.getStackTrace(Thread.java:1619) -> Genera la pila de llamadas debido al getstackTrace

Taller.MatrixParallel.multMatriz(MatrixParallel.scala:42) -> La función multMatriz fue invocada desde otro contexto. En este caso, ejecuta la lógica de multiplicación estándar de matrices.

Taller.MatrixParallel.StrassenParallel(MatrixParallel.scala:241) -> La función StrassenParallel llamó a multMatriz. Esto ocurre porque probablemente está utilizando la versión estándar como parte del algoritmo de Strassen.

taller.App$.$anonfun$benchmarking$6(App.scala:68) -> Dentro de la función benchmarking, se ejecuta un bloque de código que incluye la llamada a StrassenParallel. Es parte de las pruebas de rendimiento.

taller.Benchmark.$anonfun$compararAlgoritmo$2(Benchmark.scala:22) -> función compararAlgoritmo compara la ejecución de dos algoritmos (en este caso, posiblemente la versión estándar y la paralela).

org.scalameter.MeasureBuilder.$anonfun$measured$1(MeasureBuilder.scala:66)
-> org.scalameter mide el tiempo de ejecución de los algoritmos en este punto.

org.scalameter.MeasureBuilder.$anonfun$measuredWith$4(MeasureBuilder.scala:54)

-> Otro nivel de medición y benchmarking, posiblemente asegurando la precisión al incluir múltiples ejecuciones.

org.scalameter.Measurer$Default.measure(Measurer.scala:133) -> Realiza una medición concreta del tiempo de ejecución del algoritmo.

org.scalameter.MeasureBuilder.measuredWith(MeasureBuilder.scala:54) -> Configura las mediciones de los algoritmos.

taller.Benchmark.compararAlgoritmo(Benchmark.scala:22) -> Esta función dirige las comparaciones entre algoritmos como parte de las pruebas de rendimiento. **taller.App$.benchmarking(App.scala:64)** -> Coordina la ejecución de las pruebas de rendimiento, incluyendo múltiples tamaños de entrada.

taller.App$.main(App.scala:44) -> Regresa al flujo principal de la aplicación luego de las pruebas y cálculos.

Multiplicación de matrices paralela: (def mulMatrizPar)

```
java.base/java.lang.Thread.getStackTrace(Thread.java:1619)
taller.MatrixParallel.mulMatrizPar(MatrixParallel.scala:69)
taller.App$.main(App.scala:25)
taller.App.main(App.scala)
```

java.base/java.lang.Thread.getStackTrace(Thread.java:1619) ->Se genera la pila de llamadas debido a la ejecución de getStackTrace, mostrando el estado del programa en este momento.

taller.MatrixParallel.mulMatrizPar(MatrixParallel.scala:69) ->Se está ejecutando la función mulMatrizPar, que implementa la multiplicación paralela de matrices. Probablemente, fue llamada desde el flujo principal del programa.

taller.App$.main(App.scala:25) ->La función main dentro de la clase App llama a mulMatrizPar como parte del flujo principal de la aplicación.

taller.App.main(App.scala) ->Este es el punto de entrada del programa. Se ejecuta la función main para iniciar toda la lógica del programa.

Suma De Matrices (def sumMatriz)

```
taller.App$.main(App.scala:44)
taller.App.main(App.scala)
java.base/java.lang.Thread.getStackTrace(Thread.java:1619)
taller.MatrixParallel.SumMatriz(MatrixParallel.scala:74)
taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:107)
taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:107)
taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:105)
taller.App$.$anonfun$benchmarking$2(App.scala:58)
taller.Benchmark.$anonfun$compararAlgoritmo$1(Benchmark.scala:16)
org.scalameter.MeasureBuilder.$anonfun$measured$1(MeasureBuilder.scala:66)
org.scalameter.MeasureBuilder.$anonfun$measuredWith$3(MeasureBuilder.scala:47)
org.scalameter.MeasureBuilder.$anonfun$measuredWith$3$adapted(MeasureBuilder.s
org.scalameter.Warmer$Default$$anon$2.$anonfun$foreach$5(Warmer.scala:56)
scala.runtime.java8.JFunction0$mcV$sp.apply(JFunction0$mcV$sp.scala:18)
org.scalameter.utils.package$.withGCNotification(package.scala:24)
org.scalameter.Warmer$Default$$anon$2.foreach(Warmer.scala:49)
org.scalameter.MeasureBuilder.measuredWith(MeasureBuilder.scala:47)
org.scalameter.MeasureBuilder.measured(MeasureBuilder.scala:66)
org.scalameter.MeasureBuilder.measure(MeasureBuilder.scala:63)
taller.Benchmark.compararAlgoritmo(Benchmark.scala:16)
taller.App$.$anonfun$benchmarking$1(App.scala:58)
scala.runtime.java8.JFunction1$mcVI$sp.apply(JFunction1$mcVI$sp.scala:18)
scala.collection.immutable.Range.map(Range.scala:59)
taller.App$.benchmarking(App.scala:54)
taller.App$.main(App.scala:44)
```

taller.App.main(App.scala) -> El punto de entrada del programa llama a la función principal main.

java.base/java.lang.Thread.getStackTrace(Thread.java:1619) -> Se captura la pila de llamadas en este punto con getStackTrace.

taller.MatrixParallel.SumMatriz(MatrixParallel.scala:74) -> Se ejecuta la función SumMatriz, que suma matrices. Es invocada como parte de la lógica de MultMatrizRec.

taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:107) ->La función recursiva MultMatrizRec se llama a sí misma para resolver la multiplicación de matrices. Esta es una de las llamadas recursivas.

taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:107) ->Otra llamada recursiva de MultMatrizRec, indicando que el problema se sigue descomponiendo en subproblemas más pequeños.

taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:105) ->Aquí comienza otra rama de la recursión, posiblemente alcanzando un caso base o resolviendo subproblemas de menor tamaño.

taller.App$.$anonfun$benchmarking$2(App.scala:58) -> Dentro de la función benchmarking, se ejecuta una operación que involucra MultMatrizRec como parte del análisis de rendimiento.

taller.Benchmark.$anonfun$compararAlgoritmo$1(Benchmark.scala:16) -> La función compararAlgoritmo se encarga de comparar el rendimiento de MultMatrizRec y otros algoritmos de multiplicación.

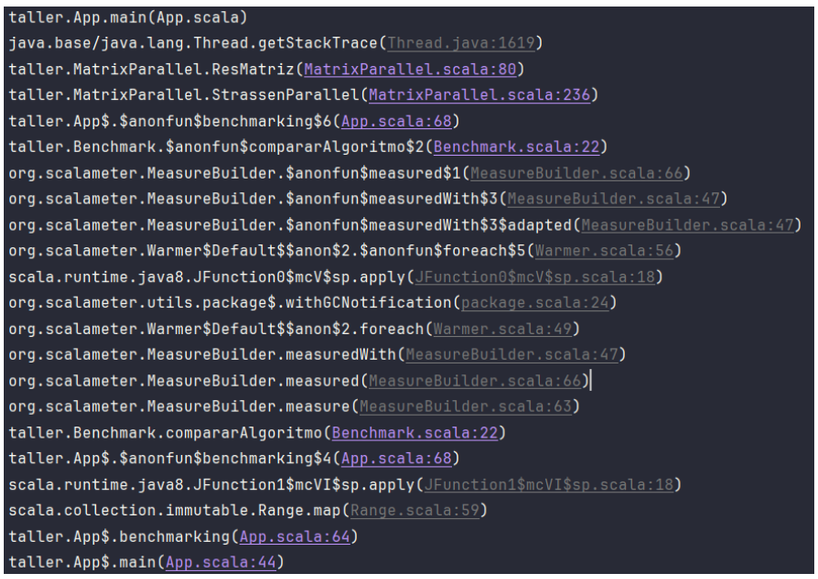
org.scalameter.MeasureBuilder.$anonfun$measured$1(MeasureBuilder.scala:66) -> El tiempo de ejecución se mide como parte de las pruebas de rendimiento.

org.scalameter.MeasureBuilder.$anonfun$measuredWith$3(MeasureBuilder.scala:47 -> Otra capa de medición, configurando cómo se registran las métricas.

taller.App$.benchmarking(App.scala:54) -> Coordina las pruebas y llama a compararAlgoritmo para evaluar la eficiencia de los algoritmos implementados.

taller.App$.main(App.scala:44) -> Regresa al flujo principal del programa después de completar las pruebas.

Resta de Matrices (def ResMatriz)



taller.App.main(App.scala) -> El programa inicia desde el punto de entrada en la función main.

java.base/java.lang.Thread.getStackTrace(Thread.java:1619) -> Se genera y captura la pila de llamadas al ejecutar getStackTrace.

taller.MatrixParallel.ResMatriz(MatrixParallel.scala:80) ->La función ResMatriz, que implementa la resta de matrices, está siendo ejecutada. Probablemente es una parte del algoritmo de Strassen.

taller.MatrixParallel.StrassenParallel(MatrixParallel.scala:236) ->El algoritmo paralelo de Strassen utiliza ResMatriz para calcular subproblemas como parte de su lógica de descomposición.

taller.App$.$anonfun$benchmarking$6(App.scala:68) ->Dentro de la función benchmarking, se invoca StrassenParallel como parte de las pruebas de rendimiento.

taller.Benchmark.$anonfun$compararAlgoritmo$2(Benchmark.scala:22) -> La función compararAlgoritmo compara el rendimiento del algoritmo de Strassen con otro algoritmo, ejecutando ambos y registrando los tiempos.

org.scalameter.MeasureBuilder.$anonfun$measured$1(MeasureBuilder.scala:66) -> org.scalameter mide el tiempo de ejecución del algoritmo en este punto.

org.scalameter.MeasureBuilder.$anonfun$measuredWith$3(MeasureBuilder.scala:47) -> Configura cómo se registran las mediciones para las pruebas de rendimiento.

taller.App$.benchmarking(App.scala:64) -> Esta función organiza las pruebas de rendimiento y gestiona la ejecución de los algoritmos a comparar.

taller.App$.main(App.scala:44) -> Regresa al flujo principal del programa tras completar las pruebas.

Multiplicación de matrices recursivas (def MultMatrizRec)

```
taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:85)
taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:107)
taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:108)
taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:108)
taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:106)
taller.App$.$anonfun$benchmarking$2(App.scala:58)
taller.Benchmark.$anonfun$compararAlgoritmo$1(Benchmark.scala:16)
org.scalameter.MeasureBuilder.$anonfun$measured$1(MeasureBuilder.scala:66)
org.scalameter.MeasureBuilder.$anonfun$measuredWith$3(MeasureBuilder.scala:47)
org.scalameter.MeasureBuilder.$anonfun$measuredWith$3$adapted(MeasureBuilder.scala:47)
org.scalameter.Warmer$Default$$anon$2.$anonfun$foreach$5(Warmer.scala:56)
scala.runtime.java8.JFunction0$mcV$sp.apply(JFunction0$mcV$sp.scala:18)
org.scalameter.utils.package$.withGCNotification(package.scala:24)
org.scalameter.Warmer$Default$$anon$2.foreach(Warmer.scala:49)
org.scalameter.MeasureBuilder.measuredWith(MeasureBuilder.scala:47)
org.scalameter.MeasureBuilder.measured(MeasureBuilder.scala:66)
org.scalameter.MeasureBuilder.measure(MeasureBuilder.scala:63)
taller.Benchmark.compararAlgoritmo(Benchmark.scala:16)
taller.App$.$anonfun$benchmarking$1(App.scala:58)
scala.runtime.java8.JFunction1$mcVI$sp.apply(JFunction1$mcVI$sp.scala:18)
scala.collection.immutable.Range.map(Range.scala:59)
taller.App$.benchmarking(App.scala:54)
taller.App$.main(App.scala:44)
```

taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:85) ->Se invoca la función MultMatrizRec, que implementa la multiplicación recursiva de matrices. Aquí comienza el proceso de descomposición.

taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:107) -> Una llamada recursiva de MultMatrizRec descompone las matrices en submatrices más pequeñas, posiblemente hacia el caso base.

taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:108) -> Otra rama recursiva de MultMatrizRec. Esto indica que la recursión está avanzando en paralelo para calcular las submatrices.

taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:106) -> Se ejecuta otra rama de la recursión. El algoritmo continúa resolviendo subproblemas antes de combinar los resultados.

taller.App$.$anonfun$benchmarking$2(App.scala:58) -> La función benchmarking ejecuta MultMatrizRec como parte de una prueba de rendimiento, pasando matrices de prueba.

taller.Benchmark.$anonfun$compararAlgoritmo$1(Benchmark.scala:16) -> La función compararAlgoritmo mide el rendimiento de MultMatrizRec y lo compara con otro algoritmo.

org.scalameter.MeasureBuilder.$anonfun$measured$1(MeasureBuilder.scala:66) -> Herramientas de org.scalameter miden el tiempo de ejecución del algoritmo recursivo. **org.scalameter.MeasureBuilder.$anonfun$measuredWith$3(MeasureBuilder.scala:47)** -> Configura las pruebas de rendimiento y asegura la precisión de las mediciones.

taller.App$.benchmarking(App.scala:54) -> Coordina las pruebas, comparando MultMatrizRec con otros métodos implementados.

taller.App$.main(App.scala:44) -> El programa regresa al flujo principal tras completar las pruebas y mediciones.

SubMatriz (def subMatriz)

```
taller.App.main(App.scala)
java.base/java.lang.Thread.getStackTrace(Thread.java:1619)
taller.MatrixParallel.SubMatriz(MatrixParallel.scala:118)
taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:102)
taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:105)
taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:108)
taller.App$.$anonfun$benchmarking$2(App.scala:58)
taller.Benchmark.$anonfun$compararAlgoritmo$1(Benchmark.scala:16)
org.scalameter.MeasureBuilder.$anonfun$measured$1(MeasureBuilder.scala:66)
org.scalameter.MeasureBuilder.$anonfun$measuredWith$3(MeasureBuilder.scala:47)
org.scalameter.MeasureBuilder.$anonfun$measuredWith$3$adapted(MeasureBuilder.scala:47)
org.scalameter.Warmer$Default$$anon$2.$anonfun$foreach$5(Warmer.scala:56)
scala.runtime.java8.JFunction0$mcV$sp.apply(JFunction0$mcV$sp.scala:18)
org.scalameter.utils.package$.withGCNotification(package.scala:24)
org.scalameter.Warmer$Default$$anon$2.foreach(Warmer.scala:49)
org.scalameter.MeasureBuilder.measuredWith(MeasureBuilder.scala:47)
org.scalameter.MeasureBuilder.measured(MeasureBuilder.scala:66)
org.scalameter.MeasureBuilder.measure(MeasureBuilder.scala:63)
taller.Benchmark.compararAlgoritmo(Benchmark.scala:16)
taller.App$.$anonfun$benchmarking$1(App.scala:58)
scala.runtime.java8.JFunction1$mcVI$sp.apply(JFunction1$mcVI$sp.scala:18)
scala.collection.immutable.Range.map(Range.scala:59)
taller.App$.benchmarking(App.scala:54)
taller.App$.main(App.scala:44)
```

taller.App.main(App.scala) -> Punto de entrada del programa, inicia la ejecución de la aplicación.

java.base/java.lang.Thread.getStackTrace(Thread.java:1619) -> Se genera la pila de llamadas debido a la ejecución de getStackTrace, capturando el estado actual del programa.

taller.MatrixParallel.SubMatriz(MatrixParallel.scala:118) -> Se ejecuta la función SubMatriz, que extrae una submatriz de una matriz principal. Esta operación es fundamental para la función recursiva de multiplicación de matrices.

taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:102) -> Se llama a la función recursiva MultMatrizRec, que implementa la multiplicación recursiva de matrices. Este es un paso clave donde se descompone el problema.

taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:105) -> Otra llamada recursiva de MultMatrizRec, descomponiendo aún más las submatrices.

taller.MatrixParallel.MultMatrizRec(MatrixParallel.scala:108) -> Otra rama de la recursión, avanzando hacia el caso base para continuar con el cálculo de productos.

taller.App$.$anonfun$benchmarking$2(App.scala:58) -> La función benchmarking ejecuta MultMatrizRec como parte de la evaluación de rendimiento del algoritmo.

taller.Benchmark.$anonfun$compararAlgoritmo$1(Benchmark.scala:16) -> La función compararAlgoritmo gestiona la comparación del rendimiento de MultMatrizRec con otros algoritmos.

org.scalameter.MeasureBuilder.$anonfun$measured$1(MeasureBuilder.scala:66) -> La herramienta org.scalameter mide el tiempo de ejecución del algoritmo.

org.scalameter.MeasureBuilder.$anonfun$measuredWith$3(MeasureBuilder.scala:47 -> Configura la medición de la ejecución para asegurar que los resultados sean precisos y confiables.

taller.App$.benchmarking(App.scala:54) -> Orquesta las pruebas de rendimiento y realiza la comparación de los algoritmos.

taller.App$.main(App.scala:44) -> Retorna al flujo principal del programa después de finalizar las pruebas y mediciones.

Multiplicación de raíces recursiva y paralela (def MultMatrizRecPar)

```
java.base/java.util.concurrent.ForkJoinTask.awaitDone(ForkJoinTask.java:436)
java.base/java.util.concurrent.ForkJoinTask.join(ForkJoinTask.java:670)
common.package$.parallel(package.scala:51)
taller.MatrixParallel.MultMatrizRecPar(MatrixParallel.scala:147)
taller.MatrixParallel.$anonfun$MultMatrizRecPar$4(MatrixParallel.scala:147)
common.package$.parallel(package.scala:50)
taller.MatrixParallel.MultMatrizRecPar(MatrixParallel.scala:147)
taller.MatrixParallel.$anonfun$MultMatrizRecPar$1(MatrixParallel.scala:144)
common.package$DefaultTaskScheduler$$anon$1.compute(package.scala:23)
java.base/java.util.concurrent.RecursiveTask.exec(RecursiveTask.java:100)
java.base/java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:373)
java.base/java.util.concurrent.ForkJoinPool$WorkQueue.topLevelExec(ForkJoinPool.java:1182)
java.base/java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:1655)
java.base/java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1622)
java.base/java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:165)
java.base/java.lang.Thread.getStackTrace(Thread.java:1619)
taller.MatrixParallel.MultMatrizRecPar(MatrixParallel.scala:124)
```

taller.MatrixParallel.MultMatrizRecPar(MatrixParallel.scala:147) ->Se ejecuta la función MultMatrizRecPar, que es la versión paralela de la multiplicación recursiva de matrices. Aquí, el problema se divide en subproblemas que se procesan de manera concurrente.

taller.MatrixParallel.$anonfun$MultMatrizRecPar$3(MatrixParallel.scala:146) ->Una función anónima dentro de MultMatrizRecPar se encarga de manejar la ejecución paralela de una de las submatrices. Es probable que se utilice para ejecutar tareas en paralelo con ForkJoinPool.

common.package$DefaultTaskScheduler$$anon$1.compute(package.scala:23) -> El planificador de tareas de common inicia la computación de la tarea paralela. Esto indica el uso de la programación concurrente y la gestión de tareas.

java.base/java.util.concurrent.RecursiveTask.exec(RecursiveTask.java:100) -> Se ejecuta una tarea recursiva utilizando RecursiveTask, que es parte de la biblioteca ForkJoinPool para la ejecución concurrente.

java.base/java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:373) -> La tarea se ejecuta de manera interna, siguiendo el mecanismo de trabajo del ForkJoinPool.

java.base/java.util.concurrent.ForkJoinPool$WorkQueue.topLevelExec(ForkJoinPool.java:1182) -> El ForkJoinPool asigna la tarea a un hilo de trabajo que ejecuta el proceso paralelo en el nivel superior.

java.base/java.util.concurrent.ForkJoinPool.scan(ForkJoinPool.java:1655) -> El ForkJoinPool escanea y gestiona la cola de trabajo, buscando tareas que puedan ejecutarse en paralelo.

java.base/java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1622)

-> Un hilo de trabajo se ejecuta en el ForkJoinPool, procesando tareas asignadas.

java.base/java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:165) -> Un hilo de trabajador específico del ForkJoinPool ejecuta su tarea hasta que termina.

java.base/java.lang.Thread.getStackTrace(Thread.java:1619) -> Se captura la pila de llamadas, mostrando el estado actual del programa cuando se ejecutó getStackTrace.

taller.MatrixParallel.MultMatrizRecPar(MatrixParallel.scala:124) -> Otra instancia de MultMatrizRecPar, posiblemente una llamada interna para dividir el trabajo en subproblemas más pequeños de manera paralela.

Multiplicacion de matriz Strassen (def MultMatrizStrassen)

taller.App.main(App.scala) ->Punto de entrada del programa donde se inicializa la ejecución de la aplicación.

java.base/java.lang.Thread.getStackTrace(Thread.java:1619) ->Se genera y captura la pila de llamadas con getStackTrace, reflejando el estado del programa en este momento.

taller.MatrixParallel.MultMatrizStrassen(MatrixParallel.scala:160) ->Se ejecuta la función MultMatrizStrassen, que implementa el algoritmo de Strassen para la multiplicación eficiente de matrices. Aquí se combinan operaciones de suma, resta y submatrices.

taller.App$.$anonfun$benchmarking$5(App.scala:68) -> Dentro de la función benchmarking, se invoca MultMatrizStrassen como parte de las pruebas de rendimiento del algoritmo.

taller.Benchmark.$anonfun$compararAlgoritmo$1(Benchmark.scala:16) ->compararAlgoritmo evalúa el rendimiento de MultMatrizStrassen, comparándolo con otro algoritmo.

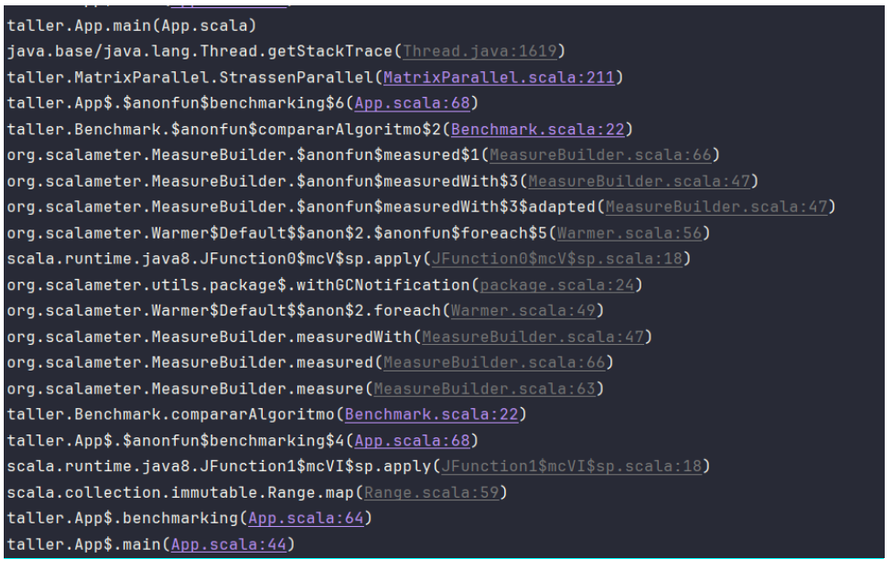
org.scalameter.MeasureBuilder.$anonfun$measured$1(MeasureBuilder.scala:66) ->org.scalameter mide el tiempo de ejecución del algoritmo para registrar datos de rendimiento.

org.scalameter.MeasureBuilder.$anonfun$measuredWith$3(MeasureBuilder.scala:47)-> Configura cómo se realizan y registran las mediciones de tiempo para las pruebas.

taller.App$.benchmarking(App.scala:64) -> Coordina las pruebas de rendimiento, ejecutando MultMatrizStrassen y otros algoritmos con diferentes tamaños de entrada.

taller.App$.main(App.scala:44) -> Regresa al flujo principal del programa tras finalizar las pruebas y mediciones.

StrassenParalela(defStrassenParallel)



taller.App.main(App.scala) ->La ejecución del programa comienza en el punto de entrada main.

java.base/java.lang.Thread.getStackTrace(Thread.java:1619) ->Se captura la pila de llamadas actual usando getStackTrace, mostrando el estado del programa en este momento.

taller.MatrixParallel.StrassenParallel(MatrixParallel.scala:211) -> Se ejecuta la versión paralela del algoritmo de Strassen, StrassenParallel, que optimiza el cálculo de la multiplicación de matrices dividiendo y resolviendo tareas en paralelo.

taller.App$.$anonfun$benchmarking$6(App.scala:68) -> Dentro de la función benchmarking, se invoca StrassenParallel como parte de las pruebas de rendimiento.

taller.Benchmark.$anonfun$compararAlgoritmo$2(Benchmark.scala:22) -> La función compararAlgoritmo compara el rendimiento de StrassenParallel con otro algoritmo.

org.scalameter.MeasureBuilder.$anonfun$measured$1(MeasureBuilder.scala:66)
->org.scalameter mide el tiempo de ejecución del algoritmo paralelo para registrar su desempeño.

org.scalameter.MeasureBuilder.$anonfun$measuredWith$3(MeasureBuilder.scala:47)-> Configura y realiza mediciones detalladas para las pruebas de rendimiento.

taller.App$.benchmarking(App.scala:64) -> Esta función coordina las pruebas de rendimiento de diferentes algoritmos, incluyendo StrassenParallel.

taller.App$.main(App.scala:44) -> Regresa al flujo principal del programa después de completar las pruebas y análisis.

Producto Punto (def prodPuntoParD)

taller.App.main(App.scala) -> El programa inicia su ejecución en el punto de entrada main.

java.base/java.lang.Thread.getStackTrace(Thread.java:1619) -> Se captura la pila de llamadas en este momento mediante getStackTrace, mostrando el flujo de ejecución del programa.

taller.MatrixParallel.prodPuntoParD(MatrixParallel.scala:266) -> Se ejecuta la función prodPuntoParD, que implementa el cálculo del producto punto utilizando paralelismo de datos con colecciones paralelas (ParVector).

taller.App$.$anonfun$benchmarking$8(App.scala:78) -> Dentro de la función benchmarking, se invoca prodPuntoParD como parte de la evaluación de rendimiento.

scala.runtime.java8.JFunction0mcIsp.apply(JFunction0mcIsp.scala:17) -> Un wrapper de Scala Runtime ejecuta la operación como una función de alto orden que retorna un valor entero.

org.scalameter.MeasureBuilder.$anonfun$measured$1(MeasureBuilder.scala:66) ->org.scalameter mide el tiempo de ejecución del algoritmo prodPuntoParD.

taller.Benchmark.CompararProductoPunto(Benchmark.scala:42) -> La función CompararProductoPunto compara el rendimiento de prodPuntoParD con su versión secuencial (prodPunto).

taller.App$.benchmarking(App.scala:78) ->Coordina las pruebas de rendimiento específicas del producto punto.

taller.App$.main(App.scala:44) ->Regresa al flujo principal del programa tras completar las pruebas.

INFORME DE PARALELIZACIÓN

Para la paralelización de las tareas del programa en las funciones de MultMatrizRecPar() e StrassenParallel() se utilizó la abstracción Parallel para no necesariamente tener que planificar la forma en como se resuelven los hilos del programa paralelo en ambos se utilizó Parallel de 4 para subdividir las operaciones del programa, lo cual permite que ser ejecuten las 4 tareas complejas al mismo tiempo.

En la forma que se obtiene la paralelización hace que en algunos momentos en que las entradas son muy pequeñas el costo de la paralelización es mayor al de iterar de manera normal o de forma recursiva, la paralelización toma mayor sentido con entradas grandes, entre más grande mejor es el tiempo de ejecución.

Para poder medir que tanto se gana al paralelizar se utilizó tanto calentamiento de máquina y valores para medir el tiempo en nanosegundos.

```
val timeA1 = config(
  KeyValue(Key.exec.minWarmupRuns -> 20),
  KeyValue(Key.exec.maxWarmupRuns -> 60),
  KeyValue(Key.verbose -> false)
) withWarmer(new Warmer.Default) measure {a1(m1,m2)}

val timeA2 = config(
  KeyValue(Key.exec.minWarmupRuns -> 20),
  KeyValue(Key.exec.maxWarmupRuns -> 60),
  KeyValue(Key.verbose -> false)
) withWarmer(new Warmer.Default) measure {a2(m1,m2)}
```

Y se mide el tiempo de mejoramiento con la siguiente línea.

```
val speedUp= timeA1.value - timeA2.value
```

Por último, durante la comparación con el Benchmark se realizó expresiones for con únicamente de 1 hasta 8 dado que más allá de eso nuestras maquinar no aguantaron la ejecución dado que las matrices se calculan en potencias de 2 y los tiempos de ejecución eran masivos.

```
def benchmarking(): Unit = {
  val Bench = new Benchmark()
  val matriz = new MatrixParallel()
  //Bench MultRec vs MultRecPar
  println("Multiplicacion de matrices de manera recursiva y paralela")
  for  {
    i <- 1 to 8
  } yield {
    val A = matriz.MatrizAlAzar(pow(2, i).toInt, 10)
    val B = matriz.MatrizAlAzar(pow(2, i).toInt, 10)
    val (time1, time2, speedUp) = Bench.compararAlgoritmo(matriz.MultMatrizRec, matriz.MultMatrizRecPar)(A, B)
    println(s"Time1: $time1, Time2: $time2, SpeedUp: $speedUp")
  }
  //Bench Strassen vs StrassenPar
  println("Multiplicacion de matrices por strassen y paralela")
  for {
    i <- 1 to 8
  }yield{
    val A = matriz.MatrizAlAzar(pow(2, i).toInt, 10)
    val B = matriz.MatrizAlAzar(pow(2, i).toInt, 10)
    val (time1, time2, speedUp) = Bench.compararAlgoritmo(matriz.MultMatrizStrassen, matriz.StrassenParallel)(A, B)
    println(s"Time1: $time1, Time2: $time2, SpeedUp: $speedUp")
  }
```

```
  //Comparacion ProductoPunto vs ProductoPuntoPar
  println("Producto punto de vectores de manera iterativa y paralela")
  val t = GenerarVec(1000)
  val u = GenerarVec(1000)
  val (time1, time2, speedUp) = Bench.CompararProductoPunto(
    () => matriz.ProductoPunto(t, u),
    () => matriz.prodPuntoParD(t.par, u.par)
  )
  println(s"Time1: $time1, Time2: $time2, SpeedUp: $speedUp")


}
```

La salida de ello es lo siguiente:

```
Benchmarking
Multiplicacion de matrices de manera recursiva y paralela
Unable to create a system terminal
Time1: 0.0879, Time2: 0.1504, SpeedUp: -0.0625
Time1: 0.168, Time2: 0.2265, SpeedUp: -0.058499999999999996
Time1: 0.5069, Time2: 0.4803, SpeedUp: 0.026600000000000013
Time1: 1.1756, Time2: 0.6566, SpeedUp: 0.519
Time1: 7.0398, Time2: 5.3417, SpeedUp: 1.6980999999999993
Time1: 52.3793, Time2: 42.8981, SpeedUp: 9.481200000000001
Time1: 467.4074, Time2: 335.4723, SpeedUp: 131.93509999999998
Time1: 3821.8257, Time2: 2832.3591, SpeedUp: 989.4665999999997
Multiplicacion de matrices por strassen y paralela
Time1: 0.0793, Time2: 0.1019, SpeedUp: -0.022600000000000001
Time1: 0.0615, Time2: 0.1263, SpeedUp: -0.0648
Time1: 0.1362, Time2: 0.2159, SpeedUp: -0.07970000000000002
Time1: 0.3713, Time2: 0.2803, SpeedUp: 0.09100000000000003
Time1: 0.8359, Time2: 0.7375, SpeedUp: 0.09839999999999993
Time1: 4.2307, Time2: 3.8571, SpeedUp: 0.3735999999999997
Time1: 35.7628, Time2: 37.7625, SpeedUp: -1.9997000000000043
Time1: 252.9383, Time2: 255.4306, SpeedUp: -2.4923
Time1: 0.3713, Time2: 0.2803, SpeedUp: 0.09100000000000003
Time1: 0.8359, Time2: 0.7375, SpeedUp: 0.09839999999999993
Time1: 4.2307, Time2: 3.8571, SpeedUp: 0.3735999999999997
Time1: 35.7628, Time2: 37.7625, SpeedUp: -1.9997000000000043
Time1: 252.9383, Time2: 255.4306, SpeedUp: -2.4923
Producto punto de vectores de manera iterativa y paralela
Time1: 0.0178, Time2: 1.1757, SpeedUp: -1.1579
```

Por cómo se puede ver entre mayor es la entrada mejor es la aceleración de la forma paralela, en cambio, la aceleración es negativa al momento de que las entradas no son lo suficientemente grandes como para aprovechar la paralelización.

Por ley de Ahmdal la aceleración en la paralelización en todos los casos debería que ser superior.

INFORME DE CORRECCIÓN

Función mulMatriz:

Consideramos que la implementación de la función es correcta ya que sea m1 una matriz de tamaño n x k y m2 una matriz de tamaño k x m. El resultado C debe ser una matriz n x m, donde cada elemento C_ij está definido por:

$$c_{ij} = \sum_{p=1}^{k} m1[i][p] \cdot m2[p][j], \quad \forall i \in [0, n-1], \forall j \in [0, m-1].$$

Dentro de la implementación la transposición de m2 asegura que sus columnas sean accesibles como filas para calcular C_ij mediante el producto punto. Además, el uso “Vector.tabulate” asegura que se calcula C_ij para todas las posiciones validas (i,j)

Función mulMatrizPar

Creemos que la implementación de esta función no esta del todo bien y pudo haber sido mejor ya que tiene algunas inconsistencias a la hora de manejar los índices y la concatenación de resultados parciales.

Entonces, sea m1 una matriz de tamaño n x k y m2 una matriz de tamaño k x m, el resultado esperado es una matriz C de tamaño n x m, donde:

$$c_{ij} = \sum_{p=1}^{k} m1[i][p] \cdot m2[p][j], \quad \forall i \in [0, n-1], \forall j \in [0, m-1].$$

En la función auxiliar "auxPar", se calcula toda la matriz C llamando una sola vez a Vector.tabulate, incluso en casos donde la matriz es subdividida, por ese motivo creemos que esta mal implementado ya que va en contra del "divide y vencerás". Además, en algunos casos la notación matemática se ve afectada ya que los índices i y j no son manejados de la mejor manera para la división entre matrices.

Función sumMatriz

Siendo m1 = [a_ij] y m2 = [b_ij], dos matrices de tamaño n x n. La suma C = [C_ij] se define como:

$$c_{ij} = a_{ij} + b_{ij}, \quad \forall i, j \in [0, n-1].$$

La implementación de la función usa "Vector.tabulate" para calcular C_ij iterando sobre los índices i y j, y sumando los elementos correspondientes de m1 y m2. Esto asegura que la operación de suma es válida en cada posición

Entonces, teniendo esto en cuenta, la implementación es correcta bajo la condición de que las matrices m1 y m2 tienen el mismo tamaño n x n.

Función ResMatriz

Sea m1 = [a_ij] y m2 = [b_ij], dos matrices de tamaño n x n. La resta C = [C_ij] se define como:

$$c_{ij} = a_{ij} - b_{ij}, \quad \forall i, j \in [0, n-1].$$

La implementación usa Vector.tabulate para calcular C_ij iterando sobre los índices i y j, y restando los elementos correspondientes de m1 y m2. Por tanto, es correcta bajo la condición de que las matrices m1 y m2 tienen el mismo tamaño n x n.

Función MultMatrizRec

Sea m1 y m2 matrices cuadradas de tamaño n x n, lo que hace el algoritmo es dividir las matrices en submatrices cuadradas de tamaño m x m (m = n/2) y calcula la multiplicación mediante la fórmula recursiva

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}, \quad C_{12} = A_{11}B_{12} + A_{12}B_{22},$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}, \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

Las submatrices resultantes se combinan para formar C. Entonces la implementación es correcta ya que la descomposición y combinación preservan las propiedades del producto matriz standard, y la base n = 1 asegura que el algoritmo es válido.

Función SubMatriz:

Dada una matriz m de tamaño n x n, la función extrae una submatriz de tamaño l x l comenzando en la posición (i, j). La submatriz se obtiene mediante:

$$\text{SubMatriz}(m, i, j, l) = [m(i + x, j + y)] \quad \text{para} \quad 0 \leq x, y < l.$$

La implementación es correcta ya que extrae correctamente la submatriz al iterar sobre las posiciones correspondientes y acceder a los elementos de la matriz original.

Función MultMatrizRecPar

Esta es una versión paralelizada del algoritmo anterior, donde las operaciones para calcula C11, C12, C21 y C22 se ejecutan en paralelo utilizando parallel. Esto mantiene las mismas fórmulas matemáticas:

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}, \quad C_{12} = A_{11}B_{12} + A_{12}B_{22},$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}, \quad C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

Al final se combinan los resultados de manera correcta. Entonces la función está bien implementada porque descompone y paraleliza coherentemente con el producto matriz estándar y hace uso de concurrencia para mejorar el rendimiento.

Función MultMatrizStrassen

Sea m1 y m2 matrices cuadradas de tamaño n x n. El método Strassen reduce el número de multiplicaciones al usar las fórmulas:

$$P_1 = A_{11}(B_{12} - B_{22}), \quad P_2 = (A_{11} + A_{12})B_{22},$$
$$P_3 = (A_{21} + A_{22})B_{11}, \quad P_4 = A_{22}(B_{21} - B_{11}),$$
$$P_5 = (A_{11} + A_{22})(B_{11} + B_{22}),$$
$$P_6 = (A_{12} - A_{22})(B_{21} + B_{22}), \quad P_7 = (A_{11} - A_{21})(B_{11} + B_{12}).$$

Las submatrices del resultado se obtienen combinando P1, P2, …, P7. En si la implementación está bien porque se sigue el algoritmo Strassen y asegura que las operaciones de descomposición y combinación respeten las definciones del producto matriz.

Función StrassenParallel

Esta es la versión paralelizada de la función anterior. Las operaciones para calcular P1,P2, …, P7 y combinar los resultados de C11, C12, C21 y C22 se realizan en paralelo usando parallel.

Función prodPuntoParD

Sea v1 y v2 vectores paralelos. El producto punto se define como:

$$s = \sum_{i=0}^{n-1} v1[i] \cdot v2[i].$$

La implementación utiliza (v1 zip v2).map para calcular los productos parciales, y luego sum para agregar los resultados. En general está bien implementado ya que realiza el cálculo del producto punto de manera concurrente usando ParVector, lo que distribuye operaciones de mapeo entre hilos.

Preguntas punto 1.4

1.¿Cuál es la implementación más rápida?

R: La strassen paralela, por la forma en la que se efectúan las operaciones, al dividirse en operaciones más senciallas hace que el programa haga las operaciones más rápidas

2.¿De qué depende que la aceleración sea mejor?

R:Depende de la forma en la que se tratan los datos al entrar, y el proceso que se hace para las operaciones

3.¿Puede caracterizar los casos en que es mejor usar la versión secuencial/paralela de cada algoritmo de multiplicación de matrices?

R:Entre más grande sea la entrada, más vale la pena paralelizar, después de matrices de 8x8, vale totalmente la pena paralelizar por el tiempo que se gana, en cambio, en casos anteriores es mucho mejor la forma secuencial.

Preguntas 1.5

¿Será práctico construir versiones de los algoritmos de multiplicación de matrices del enunciado, para la colección ParVector y usar prodPuntoParD en lugar de prodPunto? ¿Por qué si o por qué no?

R: Solo si los datos son excesivamente grandes valdría la pena, de lo contrario no porque se consumen más recursos paralelizando y tarda más tiempo que hacerlo de forma secuencial en casos no tan excesivamente grandes.

CONCLUSIONES:

La implementación del proyecto utilizando paralelización con la abstracción `parallel` de Scala permitió aprovechar eficientemente los recursos computacionales, distribuyendo las tareas de manera concurrente para lograr un desempeño óptimo en operaciones intensivas como la multiplicación de matrices. El algoritmo de Strassen, al ser una mejora sobre el enfoque tradicional, destacó por su capacidad de reducir la complejidad computacional mediante una estrategia recursiva que minimiza las multiplicaciones necesarias, lo que resultó en una mayor eficiencia, especialmente para matrices de gran tamaño. La implementación recursiva de la multiplicación estándar también se evaluó para comprender mejor las diferencias de rendimiento y escalabilidad entre ambos enfoques.

Las comparaciones de los algoritmos realizadas a través de *benchmarking* proporcionaron datos valiosos para analizar el impacto de la paralelización y el uso de algoritmos avanzados. Los resultados mostraron que, aunque el algoritmo de Strassen presenta ventajas significativas en términos de velocidad en matrices grandes, su complejidad de implementación y sobrecarga de operaciones adicionales hacen que en algunos casos no sea ideal para tamaños pequeños. Por su parte, la multiplicación recursiva estándar demostró ser más adecuada para estos casos, aunque su rendimiento decrece con el aumento del tamaño

de las matrices. Este análisis subraya la importancia de elegir el enfoque más adecuado según el contexto y los recursos disponibles, destacando las capacidades de Scala para facilitar experimentos con programación paralela y análisis de rendimiento en aplicaciones computacionalmente intensivas.