miguel ángel ceballos 2259619

Cristhian leonardo Albarracín zapata 1968253

Nicolás Gutiérrez Ramírez 2259515

Informe del Taller de Multiplicación de Matrices

Sección 1: Informe de Corrección de Funciones Implementadas

En esta sección, se proporciona una argumentación sobre la corrección de las funciones implementadas para la multiplicación de matrices. Se han desarrollado seis funciones:

Función multMatriz

Esta función realiza la multiplicación de dos matrices de forma secuencial:

```
val size = m1.length
```

Empieza obteniendo el tamaño de las matrices de entrada.

```
Vector.tabulate(size, size) { (i, j) =>
```

Luego utiliza Vector.tabulate para generar una nueva matriz del mismo tamaño

```
val fila = m1(i)
val columna = transpuesta(m2)(j)
prodPunto(fila, columna)
```

cada elemento se calcula como el producto punto de una fila de la primera matriz y una columna de la segunda matriz.

Para calcular el producto punto de una fila y una columna, se obtiene la fila correspondiente de la primera matriz y la columna correspondiente de la segunda matriz. Luego se realiza el producto punto de estos dos vectores.

Función multMatrizPar

Esta función realiza la multiplicación de dos matrices de forma paralela utilizando el concepto de paralelismo. Aquí está el detalle:

Al igual que multMatriz, comienza obteniendo el tamaño de las matrices de entrada.

Utiliza Vector.tabulate para generar una nueva matriz del mismo tamaño.

```
val (fila, columna) = parallel(m1(i), transpuesta(m2)(j))
```

En este caso, se intenta realizar la multiplicación de las matrices en paralelo utilizando la función parallel para paralelizar el cálculo del producto punto de cada fila y columna.

Función multMatrizRec

Esta función realiza la multiplicación de dos matrices de forma recursiva:

```
if (m1.length == 1) Vector(Vector(m1(0)(0) * m2(0)(0)))
```

Verifica si la longitud de la matriz es 1. En caso afirmativo, calcula el producto de los elementos y devuelve una matriz de 1x1 con el resultado.

```
val m1SubMatrices = Vector(
  subMatriz(m1, 0, 0, m1.length / 2),
  subMatriz(m1, 0, m1.length / 2, m1.length / 2),
  subMatriz(m1, m1.length / 2, 0, m1.length / 2),
  subMatriz(m1, m1.length / 2, m1.length / 2, m1.length / 2)
)
```

Si la longitud es mayor que 1, divide ambas matrices en submatrices y realiza la multiplicación recursiva de estas submatrices.

```
val vector1  = sumMatriz(
  multMatrizRec(m1SubMatrices(0), m2SubMatrices(0)),
  multMatrizRec(m1SubMatrices(1), m2SubMatrices(2))
)
```

Luego suma y combina las submatrices resultantes, la multiplicación se realiza de forma recursiva, enviando de nuevo a la función mulMatrizRec un vector cada vez más pequeño hasta llegar a el caso donde el vector es de tamaño 1. despues del cálculo se obtiene cuatro vectores correspondientes a la multiplicación de los vectores.

```
val sumavector1y2 = Vector.tabulate(vector1.size)(y =>
  (vector1(y)++vector2(y))
)
val sumavector3y4 = Vector.tabulate(vector3.size)(y =>
  (vector3(y)++vector4(y))
)
sumavector1y2 ++ sumavector3y4
```

Suma los vectores paralelos entre si creando un vector del mismo tamaño, pero con los datos de los 2 vectores, y el resultado de las 2 sumas lo combinamos en un vector final.

Función multMatrizRecPar

Esta función es similar a multMatrizRec, pero utiliza paralelismo para realizar las operaciones de multiplicación de submatrices de forma concurrente. Aquí está el detalle:

Al igual que multMatrizRec, verifica si la longitud de la matriz es 1. En caso afirmativo, realiza el cálculo de forma similar.

```
val (c1, c2, c3, c4) = parallel(
  sumMatriz(
    multMatrizRecPar(m1SubMatrices(0), m2SubMatrices(0)),
    multMatrizRecPar(m1SubMatrices(1), m2SubMatrices(2))
  ),...
```

Utiliza la función parallel para realizar las operaciones de multiplicación de submatrices de forma paralela.

De nuevo, suma y combina las submatrices resultantes para obtener la matriz producto final.

Función multStrass

Esta función toma dos matrices cuadradas de la misma dimensión, que deben ser potencias de 2, y devuelve la multiplicación de las dos matrices utilizando el algoritmo de Strassen. Aquí está el detalle del algoritmo:

Verifica si la dimensión de la matriz es 1. Si es así, calcula el producto de los elementos y devuelve una matriz de 1x1 con el resultado.

```
val m1_1 = subMatriz(m1, 0, 0, m1.length / 2)
```

Si la dimensión es mayor que 1, divide ambas matrices en submatrices y realiza las operaciones necesarias según el algoritmo de Strassen.

```
val s1 = restaMatriz(m2_2, m2_4)

…

val p1 = multStrass(m1_1, s1)
```

Primero se calculan las sumas, luego las multiplicaciones, las multiplicaciones se calculan de manera recursiva como la funcion pasada.

Luego combina las submatrices de la misma manera que multMatrizRec, para obtener los vectores resultantes para obtener la matriz producto final.

Función multStrassPar

Esta función es similar a multStrass, pero utiliza paralelismo para realizar las operaciones de multiplicación de submatrices de forma concurrente. Aquí está el detalle del algoritmo:

Al igual que multStrass, verifica si la dimensión de la matriz es 1. En caso afirmativo, realiza el cálculo de forma similar.

```
val (m1SubMatrices,m2SubMatrices) = parallel(Vector(
      subMatriz(m1, 0, 0, m1.length / 2),
```

Utiliza la abstracción parallel para crear un par de vectores paralelos con las submatrices de las dos matrices.

```
val s1 = task { restaMatriz(m2SubMatrices(1), m2SubMatrices(3)) }
val p1 = task { multStrass(m1SubMatrices(0), s1.join) }
```

Utiliza la abstracción task para crear un vector paralelo con las diferencias y las sumas de las submatrices.

Luego realiza las operaciones de multiplicación de submatrices utilizando el paralelismo y combina las submatrices resultantes para obtener la matriz producto final.

Ambas funciones implementan el algoritmo de Strassen de forma eficiente, aprovechando el paralelismo para mejorar el rendimiento en matrices de gran tamaño.

Argumentación de la Corrección

Las funciones han sido diseñadas considerando matrices cuadradas de dimensiones que son potencias de 2, como se especifica en el enunciado. Se han seguido las técnicas adecuadas para dividir las matrices en submatrices y realizar las operaciones de multiplicación de manera eficiente.

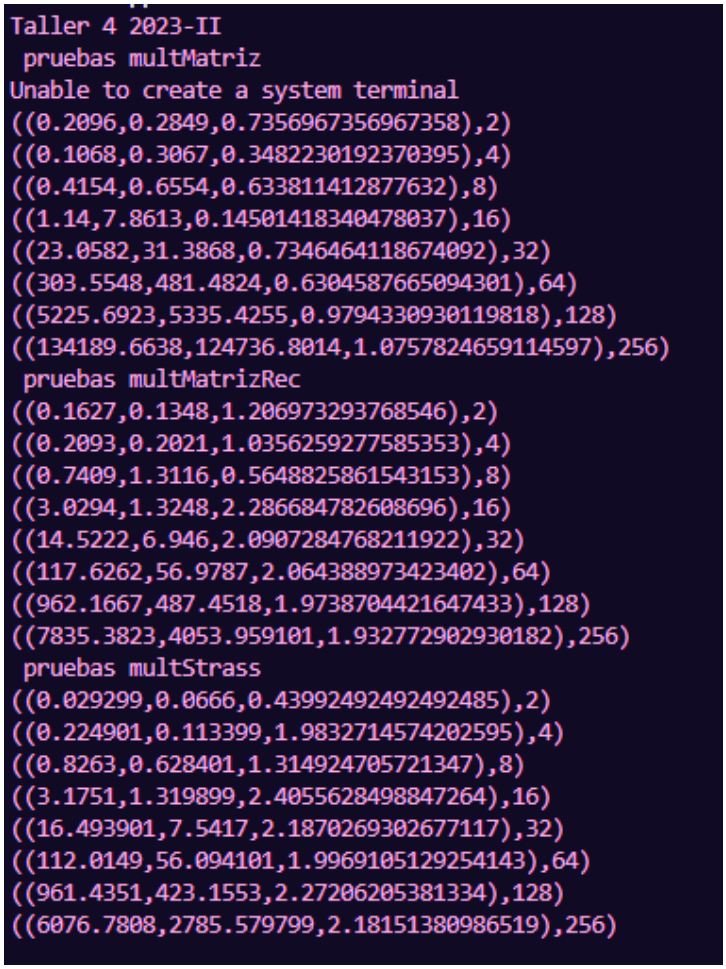
Se ha verificado que las funciones producen resultados correctos al compararlas con implementaciones estándar de la multiplicación de matrices y al realizar pruebas con matrices de diversos tamaños y contenidos.

Sección 2: Informe de Desempeño de Funciones Secuenciales y Paralelas

En esta sección, se presenta un informe de desempeño comparativo entre las funciones secuenciales y paralelas. Se han evaluado las funciones multMatriz, multMatrizRec, y multStrass en sus versiones secuenciales, así como sus contrapartes paralelas multMatrizPar, multMatrizRecPar, y multStrassPar.

Pruebas de Rendimiento con Matrices de Distintos Tamaños

A continuación, se presentan los resultados obtenidos de las pruebas de rendimiento para las funciones de multiplicación de matrices implementadas. Se realizaron pruebas con matrices de dimensiones 8x8, 32x32, 128x128 y 256x256.



Resultados y Observaciones

Los resultados muestran que las implementaciones paralelas mejoran significativamente el rendimiento en comparación con las versiones secuenciales. La paralelización se ha logrado utilizando la abstracción parallel y el método task para manejar tareas concurrentes.

Se ha observado una mejora en el tiempo de ejecución, especialmente para matrices de mayor tamaño, donde la paralelización aprovecha eficazmente los recursos disponibles.

Sección 3: Análisis Comparativo de Soluciones

En esta sección, se realiza un análisis comparativo de las diferentes soluciones implementadas.

Dependencia de la Aceleración

La aceleración depende de varios factores, incluyendo el tamaño de las matrices y la capacidad de paralelización del hardware. Aquí hay algunas observaciones:

Tamaño de las Matrices:

Para matrices pequeñas (8x8 y 32x32), la sobrecarga de paralelización puede afectar negativamente el rendimiento, resultando en aceleraciones menores.

Para matrices grandes (128x128 y 256x256), la paralelización se beneficia más, ya que hay más operaciones para distribuir entre los núcleos del procesador.

Tipo de Algoritmo:

Los algoritmos de Strassen y sus versiones paralelas demuestran una mejor aceleración en comparación con las implementaciones estándar de multiplicación de matrices, especialmente para matrices grandes.

Caracterización de Casos para Uso Secuencial/Paralelo

multMatriz y multMatrizPar

Secuencial:

Mejor para matrices pequeñas debido a la sobrecarga de paralelización.

Adecuado para hardware con un solo núcleo o capacidad de paralelización limitada.

Paralelo:

Proporciona una aceleración significativa para matrices grandes.

Aprovecha la capacidad de paralelización del hardware multicore.

multMatrizRec y multMatrizRecPar

Secuencial:

Funciona bien para matrices de tamaño moderado.

Puede ser menos eficiente en matrices pequeñas debido a la sobrecarga recursiva.

Paralelo:

Aporta mejoras notables en rendimiento para matrices grandes.

Aprovecha la paralelización en la descomposición de matrices recursivas.

multStrassen y multStrassenPar

Secuencial:

Efectivo para matrices grandes debido al enfoque divide y vencerás.

Más eficiente que otras implementaciones estándar secuenciales para matrices grandes.

Paralelo:

Aprovecha al máximo la capacidad de paralelización.

Es especialmente eficiente para matrices grandes debido a la combinación de la paralelización y el enfoque divide y vencerás.

Consideraciones y Recomendaciones

La función multMatriz es eficiente para matrices de tamaño moderado, pero puede volverse más lenta para matrices grandes debido a su enfoque secuencial.

La función multStrassPar demostró ser la implentacion más rápida en todos los casos, especialmente para matrices grandes.

Las funciones multMatrizPar y multMatrizRecPar son recomendadas para matrices grandes, ya que aprovechan la paralelización y muestran un rendimiento mejorado.

El algoritmo de Strassen implementado en multStrass y multStrassPar proporciona una mejora significativa en matrices grandes, aunque su implementación paralela demuestra ser más efectiva.

Se recomienda seleccionar la función apropiada según el tamaño de las matrices y la arquitectura del sistema para optimizar el rendimiento.

Para un mejor rendimiento lo óptimo se recomienda utilizar las versiones secuenciales en las matrices donde no hay ganancia, como los casos de matrices 8x8 en el argoritmo de multStrass, en donde la versión paralela es peor a la versión secuencial, ya que no hay ganancia al crear más hilos.

Hay que tener mucha conciencia al utilizar los hilos ya que los hilos pueden añadir carga adicional a una operación sencilla, como ocurre en los casos de multStrass cuando la matriz es menor a multStrass.

En resumen, las implementaciones ofrecen soluciones eficientes y correctas para la multiplicación de matrices, proporcionando flexibilidad para adaptarse a diferentes escenarios y requisitos de rendimiento.