Taller 4 - Multiplicación de Matrices

Juan Manuel Arango Rodas - 2259571 [Grupo 50]
Juan Camilo Valencia Rivas - 2259459 [Grupo 51]

1. Informe de Corrección de Funciones

A continuación se expone de manera formal el proceso de las funciones implementadas durante la realización del taller.

1.1. Función *multMatriz*

En esta función se utiliza el proceso matemático estándar usado para la multiplicación de matrices, realizando de manera secuencial el producto entre los elementos de las filas de la matriz A con los elementos de la matriz transpuesta de B. Para este fin, se utiliza la función *tabulate*, perteneciente a la librería Vector para aplicar el producto punto entre las filas de A y las columnas de B, de forma que:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \times B_{kj}$$

Es importante recordar que la función tabulate devuelve un vector, por lo que el resultado de la función multMatriz es una nueva matriz con la función aplicada.

1.2. Función *multMatrizPar*

El proceso matemático utilizado es el mismo que en la función multMatriz; no obstante, en esta instancia se hace uso de *task* con el fin de paralelizar las tareas y obtener un mejor rendimiento. Es decir, en este caso se divide el cálculo de los elementos individuales y se lanzan hilos que los ejecutan de manera pseudo-simultánea. En este caso, y con el ejemplo de una matriz de dimensión 2, uno de los hilos realizaría la operación

$$C_{11} = \sum_{k=1}^{n} A_{1k} \times B_{k1},$$

mientras que otro de los hilos se encargaría de solucionar el siguiente fragmento:

$$C_{12} = \sum_{k=1}^{n} A_{1k} \times B_{k2}.$$

Gracias a la ejecución paralela de estos hilos, es posible reducir de manera sustancial el tiempo requerido para finalizar el cálculo.

1.3. Función *multMatrizRec*

En esta función se da solución al producto entre las matrices tomando como apoyo la generación de submatrices y la suma de matrices. Para lograr esta implementación se divide cada matriz en cuatro submatrices, las cuales se suman de manera recursiva, obteniendo el resultado de forma similar a la del algoritmo Strassen.

1.4. Función *multMatrizRecPar*

Esta implementación realiza el mismo procedimiento que la función multMatrizRec; sin embargo, en este caso se utiliza *parallel* para lanzar hilos encargados del cálculo de las submatrices y de las sumas parciales recursivas de las mismas.

1.5. Función *multStrassen*

En este algoritmo se calcula la multiplicación utilizando submatrices, suma y resta de matrices. En primera instancia, se generan cuatro submatrices de tamaño $\frac{n}{2}$ a partir de cada matriz de entrada, las cuales se denominarán A_{ij} y B_{ij}, donde $j, i = 1, 2$. Posterior a esto, se hace uso de las funciones restaMatriz y sumMatriz para calcular las sumas y restas de submatrices necesarias para el término de la operación:

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} - B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} - B_{12}
\end{aligned}
$$

Una vez terminado este proceso, se calculan los resultados de las siguientes multiplicaciones de matrices:

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 \\
P_2 &= S_2 \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 \\
P_5 &= S_5 \cdot S_6 \\
P_6 &= S_7 \cdot S_8 \\
P_7 &= S_9 \cdot S_{10}
\end{aligned}
$$

Como penúltimo paso, se usan nuevamente las funciones restaMatriz y sumMatriz para obtener las submatrices que posteriormente conformarán nuestra nueva matriz resultado:

$$
\begin{aligned}
C_{11} &= P_5 + P_4 - P_2 + P_6 \\
C_{12} &= P_1 + P_2 \\
C_{21} &= P_3 + P_4 \\
C_{22} &= P_5 + P_1 - P_3 - P_7
\end{aligned}
$$

Para finalizar, se hace uso de la función tabulate para combinar las submatrices $C_{11}, C_{12}, C_{21}, C_{22}$.

1.6. Función *multStrassenPar*

La implementación con paralelismo del algoritmo de Strassen lleva a cabo el mismo proceso lógico que la implementación secuencial. Fue posible paralelizar el cálculo de las submatrices A_{ij} y B_{ij}. En el caso de las submatrices S, fue necesario calcularlas en tres grupos de submatrices debido a la limitación de hilos impuesta por parallel, al igual que con las multiplicaciones P, que fueron divididas en dos grupos. El cálculo de las submatrices finales C también se paralelizó.

2. Informe de Desempeño

Tras ejecutar repetidamente las pruebas de rendimiento, fue posible obtener algunas conclusiones a partir de nuestras implementaciones. En las siguientes tabla puede verse reflejado el promedio de tiempo que tomó la ejecución de cada implementación según el tamaño de la matriz:

2.1. Matriz 2x2

Cuadro 1: Promedio de tiempos de ejecución por función para matriz 2x2

	Implementación secuencial (ms)	Implementación paralela (ms)
Multiplicación	0.1109	0.096
Multiplicación Recursiva	0.0281	0.097
Algoritmo de Strassen	0.0372	0.3368

2.2. Matriz 4x4

Cuadro 2: Tiempos de ejecución por función para matriz 4x4

	Implementación secuencial (ms)	Implementación paralela (ms)
Multiplicación	0.1164	0.1326
Multiplicación Recursiva	0.1536	0.1486
Algoritmo de Strassen	0.2091	0.3302

2.3. Matriz 8x8

Cuadro 3: Tiempos de ejecución por función para matriz 8x8

	Implementación secuencial (ms)	Implementación paralela (ms)
Multiplicación	0.1275	0.2253
Multiplicación Recursiva	0.503	0.4641
Algoritmo de Strassen	0.9306	0.6681

2.4. Matriz 16x16

Cuadro 4: Tiempos de ejecución por función para matriz 16x16

	Implementación secuencial (ms)	Implementación paralela (ms)
Multiplicación	0.1686	0.5913
Multiplicación Recursiva	2.9135	1.6089
Algoritmo de Strassen	1.8136	1.6485

2.5. Matriz 32x32

Cuadro 5: Tiempos de ejecución por función para matriz 32x32

	Implementación secuencial (ms)	Implementación paralela (ms)
Multiplicación	0.7496	0.5503
Multiplicación Recursiva	10.4387	4.8107
Algoritmo de Strassen	6.4162	5.9783

2.6. Matriz 64x64

Cuadro 6: Tiempos de ejecución por función para matriz 64x64

	Implementación secuencial (ms)	Implementación paralela (ms)
Multiplicación	7.3961	0.9052
Multiplicación Recursiva	50.9572	35.5761
Algoritmo de Strassen	48.4619	36.4461

2.7. Matriz 128x128

Cuadro 7: Tiempos de ejecución por función para matriz 128x128

	Implementación secuencial (ms)	Implementación paralela (ms)
Multiplicación	39.7697	11.8485
Multiplicación Recursiva	471.0226	339.0093
Algoritmo de Strassen	404.2618	316.2028

2.8. Matriz 256x256

Cuadro 8: Tiempos de ejecución por función para matriz 256x256

	Implementación secuencial (ms)	Implementación paralela (ms)
Multiplicación	431.1357	264.0919
Multiplicación Recursiva	4453.6314	2850.847
Algoritmo de Strassen	3131.7058	2214.5879

2.9. Matriz 512x512

Cuadro 9: Tiempos de ejecución por función para matriz 512x512

	Implementación secuencial (ms)	Implementación paralela (ms)
Multiplicación	2394.336	1035.0214
Multiplicación Recursiva	31271.7477	22736.8473
Algoritmo de Strassen	20819.0052	16146.2078

2.10. Matriz 1024x1024

Cuadro 10: Tiempos de ejecución por función para matriz 1024x1024

	Implementación secuencial (ms)	Implementación paralela (ms)
Multiplicación	24898.5241	6693.4891
Multiplicación Recursiva	309418.0789	133017.3628
Algoritmo de Strassen	101866.5879	105613.5906

El análisis de estos resultados nos otorga bastante información respecto a la ejecución de nuestro código. En primera instancia, podemos observar que los resultados positivos de la paralelización se ven reducidos a medida que aumenta la dimensión de la matriz y nos permite responder algunas preguntas:

- **¿Cuál de las implementaciones es más rápida?** La implementación que nos dio mejores resultados fue la multiplicación de matrices estándar en modo paralelo.

- **¿Puede caracterizar los casos en que es mejor utilizar la versión secuencial/paralela de cada algoritmo de multiplicación de matrices?** En el caso del algoritmo de Strassen, es preferible utilizar la implementación secuencial en las ocasiones que la dimensión de la matriz sea demasiado grande. En las otras implementaciones, a pesar de que existe una deceleración, las pruebas realizadas lograron hallar el punto en el que la implementación paralela resulte menos eficiente que la secuencial.

3. Análisis Comparativo

Para efectos de brevedad, se utilizarán únicamente los resultados de las matrices 1024x1024 durante el desarrollo de esta sección.

3.1. Tiempos de ejecución en implementaciones secuenciales

Implementación	**Tiempo (ms)**
Multiplicación	24898.5241
Multiplicación Recursiva	309418.0789
Algoritmo de Strassen	101866.5879

A partir de estos resultados es posible inferir que, en caso de utilizar una implementación secuencial, la de la multiplicación estándar nos daría los mejores resultados. Por otro lado, la multiplicación recursiva se queda rezagada en este aspecto.

3.2. Tiempos de ejecución en implementaciones paralelas

Implementación	**Tiempo (ms)**
Multiplicación	6693.4891
Multiplicación Recursiva	133017.3628
Algoritmo de Strassen	105613.5906

En el caso de las implementaciones paralelas obtenemos los mismos resultados. La multiplicación estándar se resuelve sustancialmente más rápido que las otras dos; sin embargo, notamos una diferencia menor entre la multiplicación recursiva y el algoritmo de Strassen.

3.3. Aceleración

Implementación	**Secuencial (ms)**	**Paralelo (ms)**	**Aceleración**
Multiplicación	24898.5241	6693.4891	3.7198
Multiplicación Recursiva	309418.0789	133017.3628	2.3261
Algoritmo de Strassen	101866.5879	105613.5906	0.9645

Es posible evidenciar una gran aceleración entre la implementación paralela y la secuencial de la multiplicación estándar. Asimismo, en la multiplicación recursiva observamos que reducimos el tiempo a menos de la mitad. Por el contrario, en el algoritmo de Strassen se observa una ralentización del proceso, indicando que esta implementación es la menos recomendable de utilizar.