Edgar Andres Vargas Garcia - Juan Pablo Escobar Viveros

1. Informe de Corrección de Funciones Implementadas

multMatriz:

en esta función se ingresan dos elementos de tipo matriz y se retorna una matriz, se crea una variable val la cual es una versión transpuesta de la segunda matriz con la que se desea multiplicar la primera, luego se utiliza “Vector.tabulate” para general una matriz de tamaño adecuado, teniendo en cuenta el tamaño de las filas de la primer matriz con el número de columnas de la segunda matriz para finalmente definir una función “(i, j) => prodPunto(m1(i), m2Transpuesta(j))” la cual será aplicada a cada una de los elementos que tendrá la nueva matriz. Para cada par de índices “(i, j)”, se calcula el producto punto de la fila i de la primera matriz y la columna j de la segunda matriz transpuesta.

multMatrizPar:

se sigue la misma lógica de la primera función, con la diferencia de que el vector.tabulate se utiliza para crear una matriz de filas paralelas, utilizando “task{}”.Dentro de cada una de las tareas se utiliza la lógica anterior de prodpunto, la matriz resultados contiene Futures, que son promesas de resultados que se completarán en el futuro. La función “join()” se llama en cada Future para esperar y obtener el resultado de cada tarea paralela, para al finalizar realizar un mapeo de cada uno de los futures en unos valores reales, para así generar la matriz final.

multMatrizRec:

cómo se explicó en el documento la estructura de la multiplicación de matrices de forma recursiva, cada una de las dos matrices se dividió en 4 submatrices, para después en crear 4 variables la cuales contiene la multiplicación recursiva de la submatrices, estas al entrar en una posición con tal se multiplicara con la que se encuentra su misma posición. finalmente, la función construye la matriz resultante combinando las 4 variables creadas en la matriz completa. Esto se hace utilizando Vector.tabulate y seleccionando el elemento apropiado de las submatrices basado en los índices i y j.

multMatrizRecPar

en esta función se implementa la misma lógica descrita anteriormente en la función multMatrizrec, con la diferencia en que en la división de las matrices de ingreso como en el llamado recursivo de la función en las 4 variables nuevas creadas se utilizar parallel para que las tareas se hagan de manera paralela, sin contar este cambio, el código se mantiene similar en términos de la lógica.

multStrassen

en esta función se mantiene la división de matrices tanto de la primera como de la segunda para después generar 10 nuevas variables “s” las cuales son sumas y restas de matrices, luego se generaran 7 nuevas variables p las cuales son hechas por medio de llamado recursivos de la misma función con algunas de las submatrices iniciales y las variables s, para así al final obtener las variables c las cuales son producto de sumas y resta de la s variables p y se utiliza el método de combinar matrices dicho anteriormente .

multStrassenPar

se mantiene la misma lógica empleada en multStrassen con la diferencia en que se utiliza parallel en cada una de las partes en las que se generan las variables tanto s,p, y c, aparte de esta modificación no hay otro cambio significante .

2. Informe de Desempeño de las Funciones Secuenciales y Paralelas

A continuación se muestra el informe de desempeño el cual muestra las diferentes funciones evaluadas con matrices n x n donde n = 2^k, el desempeño se muestra en milisegundos y fue el tiempo de ejecución obtenido por las funciones :

	Mult. Concurrente	Mult. Paralela	Mult. Recursiva	M. Recursiva Paralela	Strassen	Strassen Paralela
2x2	0.1169 ms	0.1013 ms	0.0297 ms	0.1025 ms	0.0393 ms	0.3562 ms
4x4	0.1227 ms	0.1398 ms	0.1626 ms	0.157 ms	0.2211 ms	0.3489 ms
8x8	0.1345 ms	0.2378 ms	0.5315 ms	0.4911 ms	0.9847 ms	0.7064 ms
16x16	0.1777 ms	0.6251 ms	3.0781 ms	1.7006 ms	1.9178 ms	1.7453 ms
32x32	0.789 ms	0.5819 ms	11.0169 ms	5.0886 ms	6.7834 ms	6.3264 ms
64x64	7.8033 ms	0.9574 ms	53.8026 ms	37.6285 ms	51.3028 ms	38.5704 ms
128x128	41.9106 ms	12.5339 ms	498.593 ms	359.6721 ms	428.2134 ms	334.9084 ms
256x256	453.8281 ms	453.8281 ms	279.6621 ms	4707.3502 ms	3020.8184 ms	2349.9874 ms
512x512	2521.4157 ms	1094.7824 ms	33096.7256 ms	24110.0854 ms	22071.2259 ms	17156.3655 ms
1024x1024	26347.2139 ms	7182.6443 ms	326161.8223 ms	142312.3753 ms	112519.9078 ms	108969.2834 ms

3. Evaluación comparativa

-¿Cuál de las implementaciones es más rápida?

Si hablamos de todas la comparativas en general, la implementación en paralelo casi siempre es la ganadora de las pruebas, con la ligera diferencia de que cuando la matriz es suficientemente pequeña como de tamaño 8x8 para abajo la diferencia es tan minúscula que hasta en algunos casos la versión normal era más rápida que la paralela.

-¿De qué depende que la aceleración sea mejor?

Depende de el numero de tareas a realizar y la complejidad de la tarea .

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

Se recomienda usar el método en paralelo cuando hay una gran cantidad de tareas las cuales se pueden organizar y hacerse en paralelo, pero estas son tan pequeñas y que difícilmente se pueden dividir de manera paralela es mejor la versión secuencial.

3.1. Implementando el producto punto usando paralelismo de datos

(1) primera parte comparativa:

Tamaño del Vector	prodPunto	prodPuntoParD	Aceleración
10	0,1951	4,2552	0,0458
100	0,234	2,8344	0,0826
1000	1,7924	2,8512	0,6286
10000	1,6472	13,4768	0,1222
100000	18,4126	5,1595	3,5687
1000000	50,5611	41,3171	1,2237
10000000	683,8147	469,5542	1,4563

Como se puede apreciar en la tabla, entre menor sea el tamaño del vector es recomendable usar el prodPunto normal. En cambio, entre sea mayor el tamaño del vector es recomendable usar el prodPunto en paralelo.

(2)responder las siguientes preguntas :

-¿Será práctico construir versiones de los algoritmos de multiplicación de matrices del enunciado, para la colección ParVector y usar prodPuntoParD en lugar de prodPunto?

Teniendo en cuenta los resultados registrados en las tablas, se puede decir que si

-¿Por qué si o por qué no?

Debido a que los el tamaño de los vectores llegan a tener un tamaño demasiado grande, al usar vectores paralelos y una función que calcule el producto punto de forma paralelas, mejoraría la organización de las funciones al multiplicar las matrices, ya que en a comparación de un vector normal con uno en paralelo se nota una gran diferencia en la velocidad que podría tener un programa.

4. Análisis Comparativo de las Soluciones

	Mult. Concurrente vs Mult. Paralela	Mult. Recursiva vs M. Recursiva Paralela	Strassen vs Strassen Paralelo
2x2	1,153998026	0,289756098	0,110331275
4x4	0,877682403	1,03566879	0,633705933
8x8	0,565601346	1,082264305	1,393969422
16x16	0,284274516	1,810008232	1,098836876
32x32	1,355903076	2,165015918	1,072236975
64x64	8,150511803	1,429836427	1,330108062
128x128	3,34377967	1,386243192	1,278598566
256x256	1	0,059409665	1,285461531
512x512	2,303120419	1,372733653	1,286474452
1024x1024	3,668177457	2,291872521	1,032583718

¿Las paralelizaciones sirvieron?

Si, la paralelización sirvió en técnicamente casi todos los casos de ejemplo, eso teniendo en cuenta cuando es mejor usar paralelización o no .

¿Es realmente más eficiente el algoritmo de Strassen?

En la mayoría de casos el algoritmo de Strassen parece ser más efectivo que los demás algoritmos, ya sea el Strassen Paralelo o el normal, solamente se vio inferior en las pruebas con menor tamaño de matrices, fuera de eso fue el mejor algoritmo de la prueba.

¿No se puede concluir nada al respecto?

Se puede concluir que la paralelización es una practica bastante efectiva al momento de la mejora de la velocidad de las funciones, en los diferentes test pudimos ver una mejora significativa en casi todos los casos, y pues hablando de las funciones normales la más rápida en la mayoría de casos en la versión secuencial, seguida de la función con el algoritmo de Strauss y por último la versión recursiva.