Análisis

Jesús Rodríguez Heras

30 de octubre de 2018

Resumen

Desarrollo de la tabla con las conclusiones de usar varios hilos en la resolución de los problemas 1 y 2 de la práctica 4.

Índice

1. `matVector.java`

Para las pruebas realizadas, hemos considerado una matriz cuadrada con el mísmo número de columnsa que elementos tiene el vector por el cual se multiplica dicha matriz.

Elementos	**% CPU**	**Tiempo (s)**
500	9	0.003
1000	10	0.006
1500	12	0.009
2000	16	0.01
2500	18	0.012
3000	20	0.014
3500	19	0.017
4000	23	0.02
4500	25	0.024
5000	30	0.029
10000	35	0.097

Tabla 1: Valores de matVector.

Con valores superiores a los de la tabla, no obtenemos resultados debido a que excedemos la memoria de la máquina virtual de java.

Figura 1: Gráfica de matVector.

Figura 2: Gráfica de matVectorConcurrente.

2. `matVectorConcurrente.java`

Para las pruebas realizadas, hemos utilizado una matriz cuadrada con el mismo número de columnas que elementos tiene el vector por el cual se multiplica dicha matriz.

Elementos	**% CPU**	**Tiempo (s)**
500	25	0.019
1000	31	0.192
1500	35	0.154
2000	52	0.297
2500	49	0.216
3000	57	0.394
3500	68	0.374
4000	52	0.402
4500	75	0.427
5000	88	0.506
10000	100	0.906

Tabla 2: Valores de matVectorConcurrente.

Con valores superiores a los de la tabla obtendremos un 100 % de uso de CPU y el tiempo irá aumentando hasta el punto en el que no entre en la memoria de la máquina virtual de java.

Figura 3: Gráfica de prodMat.

3. `prodMat.java`

Para las pruebas realizadas, hemos usado matrices cuadradas.

Elementos	**% CPU**	**Tiempo (s)**
500	19	0.282
1000	34	10.13
1500	37	44.212
2000	40	112.759
2500	48	259.564

Tabla 3: Valores de prodMat.

Viendo como va evolucionando la tabla, poner más valores es innecesario, ya que sería inviable para valores más altos.

Figura 4: prodMatConcurrente.

4. `prodMatConcurrente.java`

Para las pruebas realizadas, hemos usado matrices cuadradas. Este algoritmo crea un hilo por cada elemento de la matriz resultante.

Elementos	**% CPU**	**Tiempo (s)**
25	33	0.052
50	63	0.19
100	99	0.839

Tabla 4: Valores de prodMatConcurrente.

Debido al uso de excesivos hilos, si aumentamos el numero de filas/columnas a más de 100, el programa no ejecuta debido a que crea un hilo por elemento de la matriz resultante, por lo que satura la memoria de la máquina virtual de java.

Figura 5: prodMatAHiloPorFila.

5. `prodMatAHiloPorFila.java`

Para las pruebas realizadas, hemos usado matrices cuadradas. Este algoritmo crea un hilo por cada fila de la matriz inicial, por lo que no genera tantísimos hilos como el algoritmo anterior.

Elementos	**% CPU**	**Tiempo (s)**
500	72	0.274
1000	88	1.756
1500	81	5.172
2000	92	12.37
2500	98	25.733
3000	100	49.419
3500	84	105.38
4000	94	191.47
4500	100	344.6

Tabla 5: Valores de prodMatAHiloPorFila.

Si aumentamos el número de filas/columnas a mas de 4500, el programa tarda demasiado tiempo como para ser viable. Aún así, tarda mucho menos y no satura la memoria de la máquina virtual de java en comparación con su versión de hilo por elemento de la matriz resultante.

6. Impresiones recabadas

Viendo los resultado obtenidos en las pruebas de los diversos algoritmos, podemos darnos cuenta de que el hecho de crear más hilos, no nos determina la velocidad con la que se ejecutará un programa.
Esto es así, debido a que la creación de hilos, nos genera un coste en tiempo y en memoria que puede ser crucial en otros aspectos.
El caso más evidente donde podemos contrastar lo descrito en este apartado lo tenemos a simple vista en los algoritmos `prodMatConcurrente.java` (donde creamos un hilo por elemento de la matriz resultante) y `prodMatAHiloPorFila.java` (donde creamos un hilo por fila de la primera matriz). En estos algoritmos que, hacen exáctamente lo mismo, podemos ver que tener un mayor número de hilos, no nos asegura una mayor velocidad ya que, aparte de la velocidad, la memoria, es un elemento a tener muy en cuenta, ya que en el primero se satura con algo más de 100 elementos, mientras que el segundo acepta más de 4500 y no da problemas respecto a la memoria.