Busy-Waiting, Mutex, Matrix Vector Thread multiplication times comparison

Juan Victor Alexander Amésquita Sierra

April 24, 2018

1 Introducción

Es importante conocer el porque programas como los que usan Mutex son más rápidos que aquellos que utilizan Busy-Wainting, y el porque la eficiencia en una multiplicación de matrices se ve afectada por los cache misses y como afecta el rendimiento de una lista enlazada dependiendo de donde se encuentre el mutex.

2 Busy-Waiting vs Mutex

Para realizar este experimento se utilizó dos programas similares cuya única diferencia era el uso de mutex y busy-waiting para determinar que thread utilizaría el recurso compartido. En la tabla 1 se ve que si bien a un inicio los tiempos del programa que usa Busy-Waiting es equiparable al del programa que usa Mutex, la diferencia se hace notoria cuando la cantidad de threads va aumentando. Como se puede ver a mayor cantidad de threads Busy-Waiting se estanca en casi 3 segundos debido al tiempo que consume esperando por el turno de la thread, mientras que con Mutex el tiempo disminuye a menos de 1 segundo.

Table 1: Run-Times (in Seconds) of Programs Using 1k Terms

Threads	***Busy-Wait***	***Mutex***
1	13.07837	9.778121
2	4.999971	4.696128
4	2.610453	2.388944
8	2.534595	1.579022
16	2.731977	1.154021
32	2.406745	0.9470050
64	2.652490	0.8818419

3 Multiplicación matriz-vector con pthread

Como se puede observar en la tabla 2 la eficiencia en la multiplicación con 8 mil columnas es la que tiene la peor eficiencia esto debido a que tiene una mayor cantidad de cache misses es decir, tiene que recorrer varias veces los niveles de la memoria cache para encontrar los datos que necesita de la matriz.

Table 2: Run-Times and Efficiencies of Matrix-Vector Multiplication (times are in seconds)

	Matrix Dimension					
	8,000,000 x 8		***8,000 x 8,000***		***8 x 8,000,000***	
Threads	***Time***	***Efficiency***	***Time***	***Efficiency***	***Time***	***Efficiency***
1	0,557229	1,000	0,4332408	1,000	0,6681400	1,000
2	0,2876348	0,968	0,2387841	0,907	0,389195	0,858
4	0,1798661	0,774	0,1390011	0,779	0,4282629	0,390

4 Operaciones en Lista enlazada usando pthreads

Como se puede ver en la tabla 3 la implementación mas rápida es la de read-write locks debido a que solo consume tiempo de espera cuando se realizan estas operaciones, la más lenta es la que usa un mutex por nodo es esta misma condición lo que hace que se demore mucho más que las otras implementaciones, lo mismo ocurre en la tabla 4 donde sus tiempos son mas que el doble que las otras implementaciones. Pienso que el porque de la lentitud al colocar un mutex por nodo es debido a que al, por ejemplo, insertar un elemento en paralelo al extremo de la lista pero otra thread esta utilizando algun nodo intermedio tendra que esperar a que la operación que se realiza en ese nodo termine para poder continuar con su ejecución.

Table 3: Linked List Times: 1000 Initial Keys, 100,000 ops, 99.9% Member, 0.05% Insert, 0.05% Delete

	Number of Threads			
Implementation	***1***	***2***	***4***	***8***
Read-Write Locks	0.163490	0.098344	0.097626	0.088546
One Mutex for Entire List	0.159704	0.251342	0.313504	0.322357
One Mutex per Node	0.903164	1.078087	0.741978	0.793247

Table 4: Linked List Times: 1000 Initial Keys, 100,000 ops, 80% Member, 10% Insert, 10% Delete

	Number of Threads			
Implementation	**1**	**2**	**4**	**8**
Read-Write Locks	2.146679	2.272500	2.245201	2.664563
One Mutex for Entire List	2.151260	2.405133	2.379499	2.840533
One Mutex per Node	5.976023	5.263990	3.045335	3.242442