Análisis y Resultados

Christofer Fabián Chávez Carazas
Universidad Nacional de San Agustín
Algoritmos Paralelos

16 de abril de 2017

1. Descripción del Problema

Usando *Pthreads* implementar tres algoritmos paralelos: Multiplicación matriz-vector, Cálculo de PI y una lista enlazada multihilo. Para la Multiplicación matriz-vector, probar y analizar el algoritmo con diferentes tipos de tamaños. Para el cálculo de PI, probar y analizar el algoritmo implementado con *Busy-Wait* y con *Mutex*. Para la lista enlazada, probar y analizar el algoritmo implementado con los diferentes tipos de mutex.

2. Resultados

2.1. Set-Up

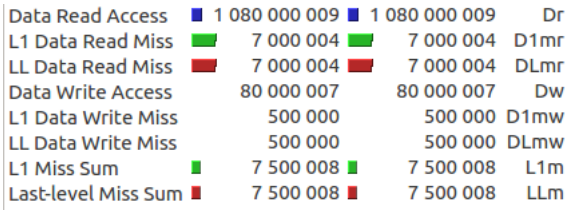
- Procesador: Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz.
- Memoria: 6 Gb.
- L2 Cache: 3072 KB.

2.2. Multiplicación matriz-vector

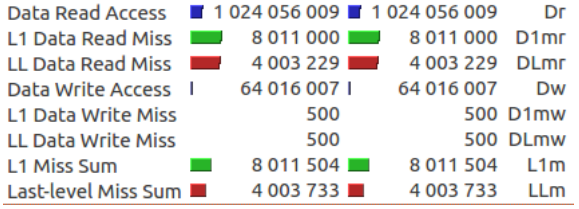
Para implementar la multiplicación matriz-vector se repartió de manera equitativa las filas de la matriz entre los hilos. En la Tabla 1 se muestran los tiempos y la eficiencia con los diferentes tamaños de matrices y el número de hilos. Como se puede observar la matriz cuadrada de 8000x8000 tiene los tiempos más bajos, seguido por la matriz de 8x8,000,000 y luego la más lenta 8,000,000x8.
En la Figura 1 se muestra los caché misses de las tres pruebas con un hilo. Con una matriz 8,000,000x8 se tienen más `write miss` (Figura 1(a)) lo que lo hace más lento. Con una matriz 8x8,000,000 y 8000x8000 se tienen el mismo número de *read*

	8,000,000x8		8000x8000		8x8,000,000	
Treads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.851185	1	0.298461	1	0.401316	1
2	0.596006	0.7140741872	0.176037	0.8477223538	0.326044	0.6154322729
4	0.534235	0.3983195597	0.16622	0.4488945374	0.285785	0.3510646115

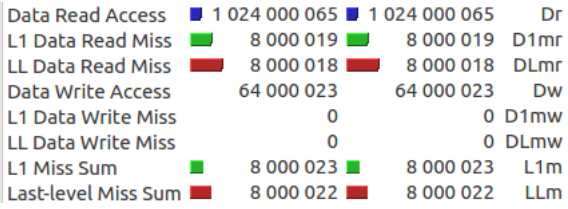
Tabla 1: Run-Times y eficiencia de la Multiplicación Matriz-Vector



(a) Caché misses con matriz 8,000,000x8



(b) Caché misses con matriz 8000x8000



(c) Caché misses con matriz 8x8,000,000

Figura 1: Resultados del Valgrind para la Multiplicación Matrix-Vector con un hilo

miss en el primer nivel de la cache (Figuras 1(c) y **??**), pero la diferencia se ve en el último nivel de la cache, donde la matriz 8000x8000 tiene menos *read miss*.

2.3. Cálculo de PI

Se probó el programa con dos implementaciones, uno con espera activa y otro utilizando mutex, los dos colocados donde el hilo actualiza la variable global. El experimento se ejecutó con un $n = 10^8$. Los resultados se muestran en la Tabla 2. Como se puede ver, con pocos hilos no hay mucha diferencia en el tiempo, pero cuando se va incrementando el número de hilos, la espera activa demora mucho más. Esto se debe a que el mutex, cuando se desbloquea, deja que cualquier hilo que este esperando entre, pero la espera activa sólo deja entrar al hilo con el id siguiente.

Threads	Busy-Wait	Mutex
1	1.417573	1.410313
2	0.845606	0.840515
4	0.758986	0.75207
8	0.845326	0.724456
16	0.889439	0.703118
32	1.031882	0.728032
64	1.678438	0.715238

Tabla 2: Run-Times (en segundos) del cálculo de PI

	Number of Threads			
Implementation	1	2	4	8
Read-Write Locks	0.606194	0.917242	0.974216	1.031685
One Mutex for Entire List	0.641805	1.151787	1.094305	1.197174
One Mutex per Node	2.004661	5.108905	6.368233	4.163973

Tabla 3: Tiempos de la lista enlazada con 80 % Member, 10 % Insert, 10 % Delete

2.4. Lista enlazada

Se probó el programa con tres implementaciones: un mutex para toda lista, un mutex para cada nodo de la lista, y un *read-write lock*. Los experimentos se hicieron con una lista enlazada con 1000 elemento iniciales generados aleatoriamente. Para cada ejecución se computó un total de 100,000 operaciones de Member, Insert y Delete, variando el porcentaje de operaciones a ejecutar. Todos los hilos se reparten equitativamente las operaciones; todos los hilos hacen las tres operaciones en porciones iguales.
En las Tablas 3 y 4 se muestras los tiempos con las diferentes implementaciones, el número de hilos y los diferentes porcentajes para cada operación. En todos los casos poner un mutex por nodo demora más y es más costoso, por lo que no es muy recomendable usarlo. Cuando hay un número considerable de operaciones de lectura (Tabla 3), la implementación con read-write lock es un poco más rapida que la implementación con un sólo mutex para toda la lista, casi no habiendo mucha diferencia. Pero cuando el número de operaciones de lectura es muy pequeño (Tabla 4), sí se nota la diferencia, siendo la implementación con read-write locks mucho más rápida.

	Number of Threads			
Implementation	1	2	4	8
Read-Write Locks	0.195334	0.257124	0.302961	0.296626
One Mutex for Entire List	0.192306	0.539524	0.530687	0.549751
One Mutex per Node	1.67227	2.546439	3.03843	3.428606

Tabla 4: Tiempos de la lista enlazada con 99.9 % Member, 0.05 % Insert, 0.05 % Delete