“AÑO DE LA UNIVERSALIZACIÓN DE LA SALUD”.

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

ESCUELA PROFESIONAL DE CIENCIA DE LA COMPUTACIÓN

COMPUTACIÓN PARALELA Y DISTRIBUIDA

Tarea 03

Alumno:
Miguel Alexander Herrera Cooper

Docente:
Dr. Alvaro Henry Mamani Aliaga

9 de octubre de 2020

Índice

1. Ejercicio 1

Implemente y analice el problema de la lista enlazada multithread. Replique los cuadros y compare sus resultados.

Resolución

Table 4.3 Linked List Times : 1000 Initial Keys , 100000 ops				
99.9 % Member 0.05 % Insert 0.05% Delete				
Implementation	Threads			
	1	2	4	8
Read-White Locks	0.085115	0.109195	0.205993	0.720024
One Mutex for Entire List	0.117777	0.139951	0.296115	0.539064
One Mutex per Node	0.082969	0.107049	0.252008	0.611782

Figura 1: Caption

Table 4.4 Linked List Times : 1000 Initial Keys , 100000 ops				
80 % Member 10 % Insert 10% Delete				
Implementation	Threads			
	1	2	4	8
Read-White Locks	0.104904	0.092983	0.288009	0.649929
One Mutex for Entire List	0.098943	0.129938	0.277996	0.884056
One Mutex per Node	0.128030	0.133007	0.265836	0.550031

Figura 2: Caption

En las Tablas 4.3 y 4.4 se muestras los tiempos con las diferentes implementaciones, el número de hilos y los diferentes porcentajes para cada operación.

En todos los casos usar 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), sı se nota la diferencia, siendo la implementación con read-write locks mucho más rápida.

2. Ejercicio 2

Replicar la tabla de los resultados del problema de multiplicación Matriz-Vector y explicar los términos de cache-coherence, false sharing.

Resolución

Table 4.5 Run-Times and Efficiencies of Matrix-Vector Multiplication (times are in seconds)						
Implementation	**Threads**					
	8000000 x 8		**8000 x 8000**		**8 x 8000000**	
	Time	**Eff**	**Time**	**Eff**	**Time**	**Eff**
1	1.288247	1.149343	0.335460	0.983585	0.340752	0.988881
2	1.171835	0.629666	0.193625	0.883774	0.188476	0.900395
3	1.115986	0.327525	0.119788	0.723385	0.111910	0.784694

Figura 3: Tiempos de Ejecucion y Eficiencias de la Matriz-Vector

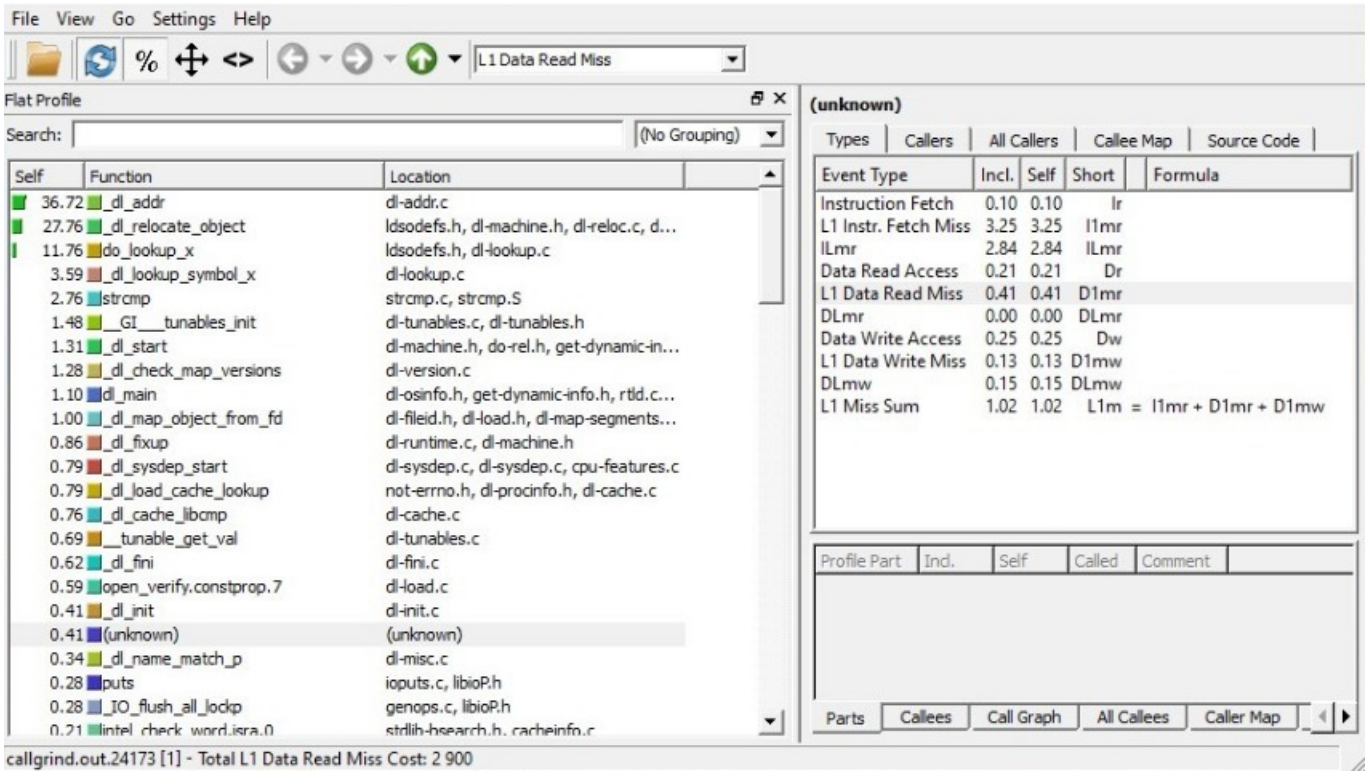

Figura 4: analisis de Cache en Valgrind

2.0.1. Cache-Coherence

Las arquitecturas modernas de microprocesadores usan cachés para reducir los tiempos de acceso a la memoria, por lo que las arquitecturas típicas tienen hardware especial para asegurar que los cachés en los diferentes chips sean coherentes.

Dado que la unidad de $cache\_coherence$, una línea de caché o bloque de caché, suele ser más grande que una sola palabra de memoria, esto puede tener el desafortunado efecto secundario de que dos subprocesos pueden estar accediendo a diferentes ubicaciones de memoria, pero cuando las dos ubicaciones pertenecen a la misma línea de caché, el hardware $cache\_coherence$ actúa como si los subprocesos estuvieran accediendo a la misma ubicación de memoria; si uno de los subprocesos actualiza su ubicación de memoria y luego el otro subproceso intenta leer su ubicación de memoria, tendrá que recuperar el valor de la memoria principal.

2.0.2. False sharing

Este termino es usado en la situación de que el hardware está obligando al hilo a actuar como si realmente estuviera compartiendo la ubicación de la memoria. Por lo tanto, esto se llama $false\_sharing$ y puede degradar seriamente el rendimiento de un programa de memoria compartida.

Algunas funciones de C almacenan datos en caché entre llamadas al declarar variables estáticas, esto puede provocar errores cuando varios subprocesos llaman a la función; dado que el almacenamiento estático se comparte entre los subprocesos, un subproceso puede sobrescribir los datos de otro subproceso. Esta función no es segura para subprocesos y, desafortunadamente, hay varias funciones de este tipo en la libreria C.

3. Ejercicio 3

THREAD-SAFETY. Análizar porque strtok no es thread-safe y mostrar un ejemplo de su mal funcionamiento. Explicar porque strtok_r si funciona en un ambiente multitithread.

Resolución

```
cooper@cooper-legion-y545:~/Desktop/4to/Computacion_Paralela/Practica03$ mpirun -np 1 ./token 2
Enter text
Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 0 > After tokenizing, my_line = Pease
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 0 > After tokenizing, my_line = Pease
Thread 1 > my line = Pease porridge cold.
Thread 1 > string 1 = Pease
Thread 1 > string 2 = porridge
Thread 1 > string 3 = cold.
Thread 1 > After tokenizing, my_line = Pease
```

Figura 5: Resultados con Strtock_r

Análisis :
La función strtock no es seguro para subprocesos ya que si varios subprocesos llaman al mismo tiempo , la salida puede no ser correcta . La manera de solucionarlo es usar **strtock_r** y agregarle un puntero en loss parámetros, este argumento se utiliza para almacenar el estado de una llamada en lugar de utilizar una variable global.

En el siguiente enlace se puede ver el código fuente del trabajo.