Informe de Pthreads

Fabrizio Paul Rosado Málaga
Universidad Católica San Pablo
Arequipa, Perú
Email: fabrizio.rosado@ucsp.edu.pe

I. Introducción

A continuación se muestra el desarrollo de la ejecución de varios programas utilizando la librería pthreads en el lenguaje C.

II. Desarrollo

A. Multiplicación de Matriz Vector

El algoritmo consiste en una multiplicación de una matriz por un vector, dividiendo el número de filas de la matriz entre el número de threads para distribuir los datos, cada thread realiza el producto punto con sus filas asignadas y el vector.

Matrix Vector Multiplication					
Threads	8000000 x 8	80000 800	8000 x 8000	800 x 80000	8 x 8000000
1	3.27E-01	2.67E-01	2.65E-01	2.61E-01	2.52E-01
2	1.68E-01	1.39E-01	1.40E-01	1.31E-01	3.28E-01
4	1.01E-01	8.55E-02	9.13E-02	8.14E-02	2.69E-01
8	7.68E-02	7.82E-02	6.12E-02	7.60E-02	2.60E-01

Fig. 1. Comparación de Matriz Vector Multiplication

La figura 1 muestra la comparación en tiempo de ejecución teniendo en cuenta el número de threads utilizados y las dimensiones de la matriz. Mostrando que mientras más threads se utilicen, menos tiempo requiere la ejecución.

B. Pthread Sorted Linked List

Esta estructura de datos consiste de tres funciones, buscar, insertar y eliminar nodos en la lista.

Para paralelizar la estructura, estas funciones deben ser adaptadas. La función buscar no presenta bastantes cambios, pues varios threads pueden utilizar esa función simultáneamente sin ningún problema. En cambio las otras dos funciones al ser de escritura, es necesario un mecanismo de exclusión como mutex y write-read locks.

A continuación se realiza una comparación de los tiempos ejecutando varios mecanismos de Wait and Signal para evitar problemas de lectura y escritura.

El código puede ser encontrado en el repositorio de Github al cual se accede cliqueando en el hipervínculo: Github/FPRM2021/Comp_Paralela

Linked List One Mutex, 1000 keys, 10000 operations, 0.6 searches, 0.2 inserts, 0.2 deletes			
Threads	One Mutex	Mutex Per Node	Write-Read Lock
1	0.3	2.75	6.00E-02
2	0.511	1.9	6.12E-02
4	1.004	1.501	6.49E-02
8	1.968	1.024	6.30E-02

Fig. 2. Comparación de lectura al 60% y escritura al 40%

Linked List One Mutex, 1000 keys, 10000 operations, 0.2 searches, 0.6 inserts, 0.2 deletes			
Threads	One Mutex	Mutex Per Node	Write-Read Lock
1	0.453	5.22	0.121
2	0.594	4.12	0.137
4	1.127	2.75	0.137
8	2.111	2.09	0.141

Fig. 3. Comparación de lectura al 40% y escritura al 60%

C. Threat Safety

Thread Safety se refiere a si un bloque de código puede ser simultáneamente ser ejecutado por múltiples threads sin ningún problema.

El caso de la función strtok no es thread-safe debido a que cuando es usado simultáneamente por varios threads, el string que utiliza como entrada, no es un dato privado, sino que es compartido con los demás threads, por lo que este se puede ver sobreescrito si es que otro thread realiza una llamada a esta función.

Un ejemplo del mal funcionamiento de esta función es al utilizarla para realizar la tokenización de strings, es decir separar las palabras en una oración.

Al querer tokenizar las oraciones:

Fuente de lasaña en el horno Fuente de Lasaña caliente Fuente de Lasaña fría

Utilizando dos threads con la función strtok, el primer thread tokenizaría la primera oración, luego el segundo thread pondría en caché la segunda oración. El primer thread al haber acabado de tokenizar la primera oración, pondría en caché la tercera oración en caché sobre-escribiendo la oración del segundo thread. Dando así un mal resultado.

Afortunadamente existe una función strtok diseñada para ser utilizada en un ambiente multithreading llamada strtok_r este en cambio utiliza tres parametros de entrada, los dos primeros son iguales al de su anterior versión, el string a tokenizar, los separadores, el tercer parámetro es un puntero doble de tipo char que mantiene al tanto de donde se encuentra la función en el string de entrada. Su propósito es el de un puntero en caché dentro de la función strtok.

III. Conclusiones

En los algoritmos desarrollados como la multiplicación matriz vector, es posible darse cuenta que el uso de threads para ejecutar un programa y paralelizarlo, lo hace más eficiente al repartir el trabajo a pesar de las dimensiones de la matriz.

La lista enlazada ordenada muestra algunos aspectos interesantes al ser probada con diferentes tipos de operaciones, diferentes números de threads y diferentes mecanismos de Wait and Signal. En la figura 2 los tiempos de ejecución en general son menores a los de la figura **??**, debe ser por el número de operaciones de lectura y escritura que difieren en las dos pruebas. También la diferencia entre el uso de los mecanismos de Wait and Signal en ambas pruebas es notable, el orden en cuanto a mejores tiempos de peor a mejor sería: Mutex per Node, One Mutex, Write-Read Lock. Esto se debe a como funciona cada uno de estos mecanismos, el Mutex per Node obviamente es el más lento debido a que para acceder a cualquier nodo, este tiene que pasar por mutex lock y un mutex unlock. Mientras que usando un solo mutex para toda la lista, serializará las operaciones así sean de lectura o escritura, pero manteniendo un rendimiento mejor que al de un mutex para cada nodo. En el caso de los Read-Write locks, estos ofrecen un acceso más flexible para las operaciones de lectura, evitando cortar el acceso a múltiples threads a leer al mismo tiempo. Por lo que cuando hay menos operaciones de escritura, los Read-Write locks tienen un mejor desempeño, en cambio cuando las operaciones de escritura son mayores, su rendimiento sería parecido al de usar un mutex en toda la lista.