Informe: paralelismo en memoria compartida usando pthreads

Fredy Álvarez Béjar
Algoritmos Paralelos,
Ciencia de la computación - Universidad Católica San Pablo

[1]

1. Introducción

En un programa en el cual existen múltiples threads (o subprocesos) es necesario tener alguna forma de bloquear el acceso de más de un thread a una sección crítica, es decir que un único thread pueda ejecutar el código contenido en la sección crítica a la vez, para evitar corrupción de datos y otros problemas relacionados al paralelismo

En este trabajo se analizarán distintas formas de sincronización entre threads, usando *busy waiting*, mutex y mutex de lectura y escritura. Todos los tiempos se tomaron usando un procesador AMD A10-4600M que cuenta con 4 núcleos y fueron compilados usando los compiladores de GNU: *gcc* y *g++* usando flags de optimizaciones: *-O3*

2. Comparación entre *busy waiting* y mutex aplicados al cálculo de valores aproximados a π

Para calcular el valor de π se usó la serie de Leibniz:

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \frac{\pi}{4}$$

Para paralelizar este algoritmo, se envía a cada thread una porción del total de iteraciones, luego se agregan todos los valores generados por cada thread par producir el resultado general.

Threads	Cantidad de iteraciones						
	1024	10240	102400	102400	10240000	102400000	1024000000
1	0.285	0.390	1.124	7.884	66.217	493.422	4713.495
2	0.248	0.372	1.080	7.063	67$\dot{7}$507	335.872	2427.462
4	0.299	0.320	0.643	3.685	39.816	263.100	2398.610
8	0.643	5.780	1.052	6.690	56.800	268.868	2487.603
16	1.197	1.066	84.345	5.573	81.211	283.049	2604.393
32	1$\dot{7}$526	2.104	330.660	248.095	438.372	548.448	2818.076

Cuadro 1. Busy waiting: Tiempos en milisegundos del algoritmo implementado en C, usando pthreads

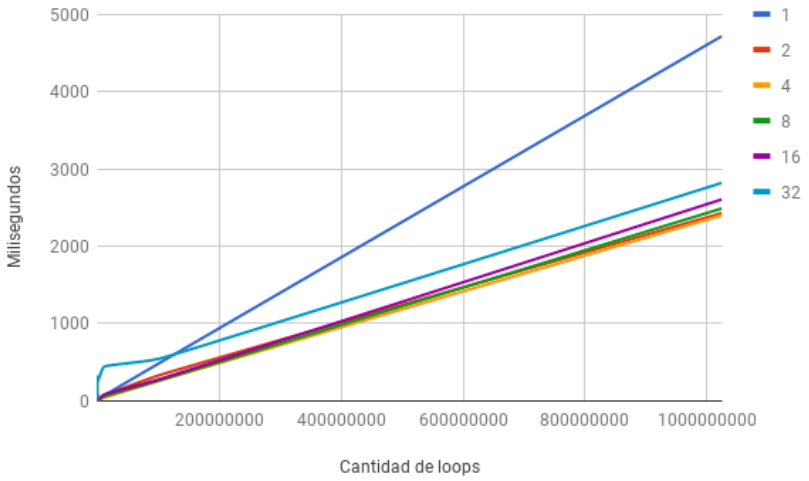

Figura 1. Busy waiting C: Gráfica del cuadro 1

Threads	Cantidad de iteraciones						
	1024	10240	102400	102400	10240000	102400000	1024000000
1	0.386	0.453	1.117	8.314	68.415	501.717	4824.106
2	0.256	0.414	0.651	8.064	397523	467.416	2588.266
4	0.43	0.411	07527	47528	31.402	247.056	2380.311
8	1.627	1.708	1.26	5.645	48.715	297.562	2393.768
16	3.736	4.371	27.104	14.8	131.318	274.221	2486.501
32	5.044	377.145	5.519	263.563	418.376	587.235	29257576

Cuadro 2. Busy waiting: Tiempos en milisegundos del algoritmo implementado en C++, usando threads de la librería estándar

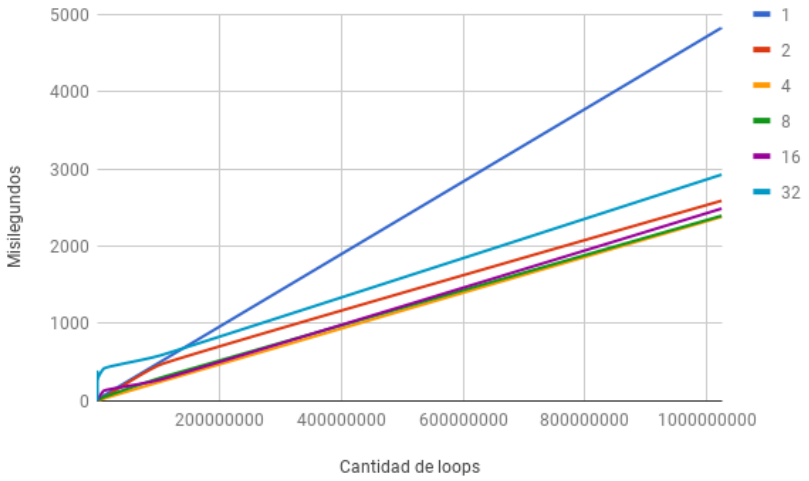

Figura 2. Busy waiting C++:Gráfica del cuadro 2

Threads	Cantidad de iteraciones						
	1024	10240	102400	102400	10240000	102400000	1024000000
1	0.302	0.202	0.766	6.156	67.402	465.668	4502.803
2	0.305	0.367	0̇7594	8.228	50.449	306.788	2330.412
4	0.294	0.312	0.667	4.214	35.249	240.862	2317.752
8	0.605	0.610	0̇7584	5̇7584	27̇7561	233.123	2230.705
16	0̇7538	1.021	1.326	6.016	32.512	229.225	2205.334
32	1̇7584	3.731	2.217	3.703	23.398	225.191	2279.745

Cuadro 3. Mutex: Tiempos en milisegundos del algoritmo implementado en C, usando pthreads

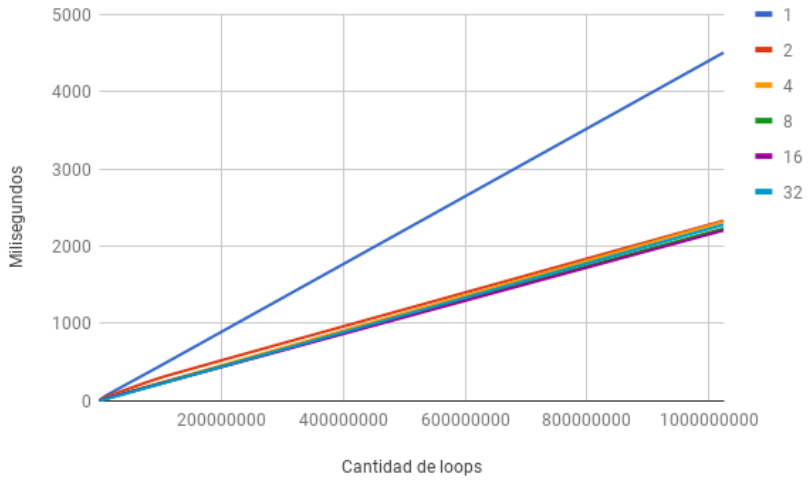

Figura 3. Mutex C: Gráfica del cuadro 3

Threads	Cantidad de iteraciones						
	1024	10240	102400	102400	10240000	102400000	1024000000
1	0.249	0.472	1.263	8.3	66.594	501.301	4815.233
2	0.356	0.522	1.271	9.285	40.011	312.488	2482.82
4	0.376	0.495	0.792	5.713	39.129	256.733	2369.683
8	0.771	0.852	1.225	4.83	29.397	251.747	2372.056
16	2.13	3.565	1̇7555	3.148	29.233	239.008	2368.244
32	3.25	4.785	6.863	4.815	28.873	239.695	2373.059

Cuadro 4. Mutex: Tiempos en milisegundos del algoritmo implementado en C++, usando threads de la librería estándar

Se implementaros dos versiones del algoritmo una en C, usando pthreads y otra en C++ usando threads de la librería estándar. Ambas implementaciones mostraron un desempeño similar. La implementación usando *busy waiting* (en C y C++) empeora en desempeño cuando se ejecutan más threads que procesadores disponibles, en cambio la implementación con mutex, mantiene su desempeño aunque se usen más threads que procesadores disponibles. Como se observa en la figura 7, usar mutex es más eficiente que usar *busy waiting*.

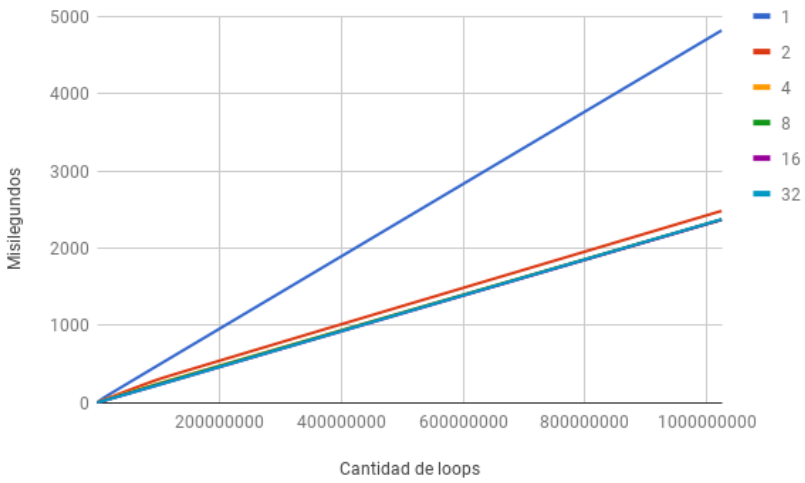

Figura 4. Mutex C++: Gráfica del cuadro 4

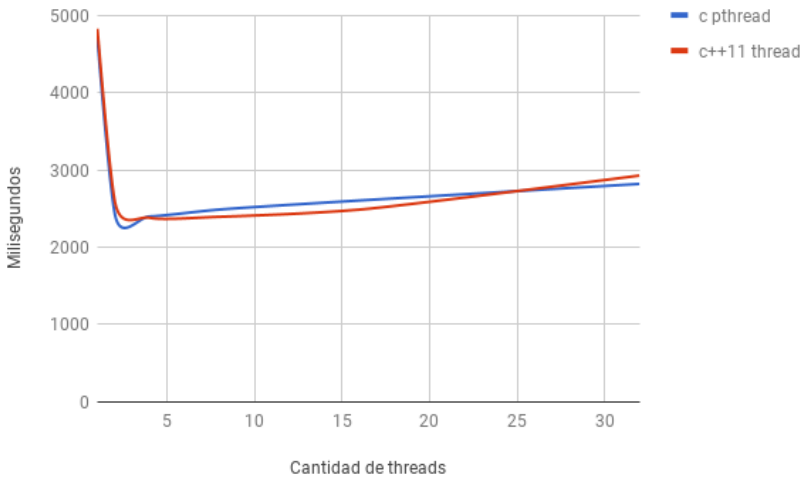

Figura 5. Comparación de tiempos usando busy waiting entre las implementación en C y C++

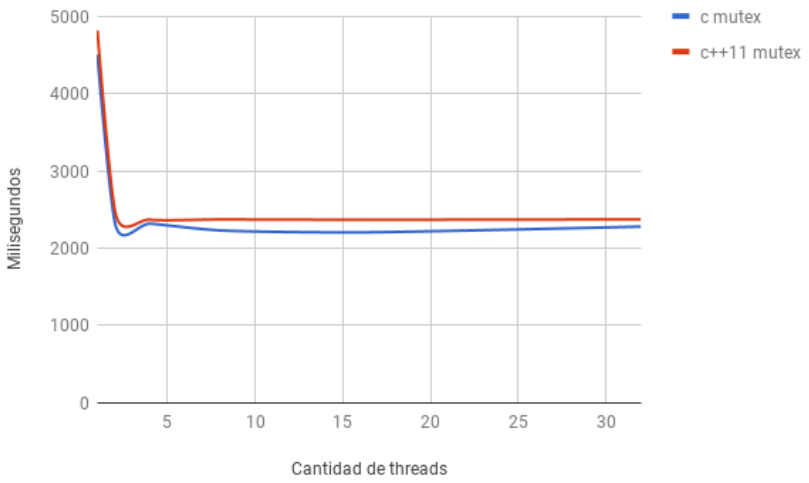

Figura 6. Comparación de tiempos usando mutex entre las implementaciones en C y C++

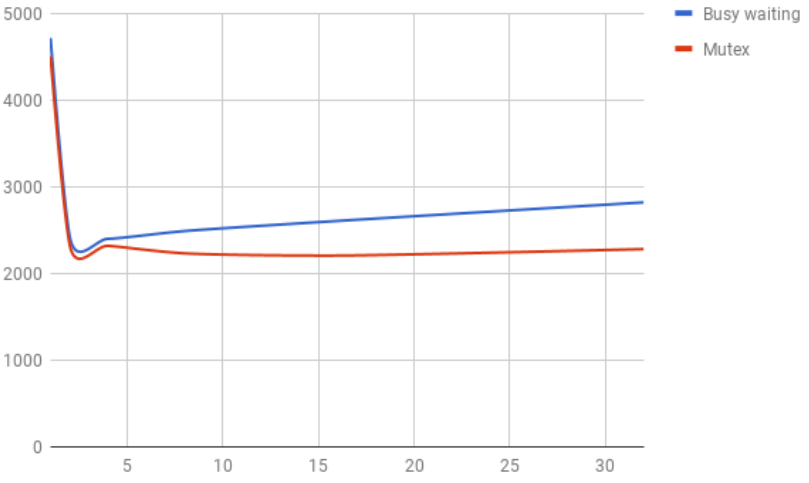

Figura 7. Comparación entre busy waiting y mutex en C

3. Comparación de distintos mecanismos de bloqueo en listas enlazadas

Se implementaron mecanismos de bloqueo en una lista enlazada que permite tres operaciones:

- Buscar un elemento
- Borrar un elemento
- Insertar un elemento

Método	Cantidad de threads			
	1	2	4	8
Read-Write Locks	1.507	1.058	1.561	1.617
Un mutex para toda la lista	1.529	2.329	2.534	2.597
Un mutex por nodo	11.182	15.735	10.421	15.144

Cuadro 5. Cantidad de operaciones: 100000, 90 % búsquedas, 5 % eliminaciones, 5 % inserciones. Tiempo en segundos

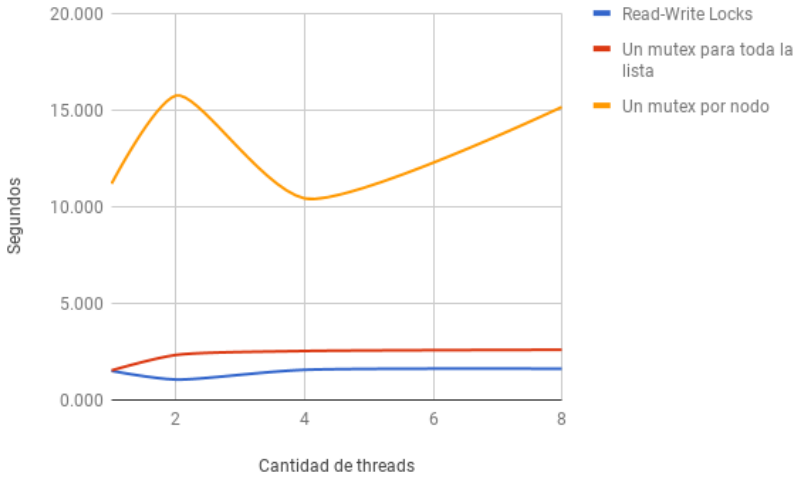

Figura 8. Gráfico de la tabla 5, 90 % búsquedas, 5 % eliminaciones, 5 % inserciones

Método	Cantidad de threads			
	1	2	4	8
Read-Write Locks	2.875	2.430	3.759	3.702
Un mutex para toda la lista	2.871	4.074	4.101	4.398
Un mutex por nodo	27.096	33.589	23.552	26.605

Cuadro 6. Cantidad de operaciones: 100000, 80 % búsquedas, 10 % eliminaciones, 10 % inserciones. Tiempo en segundos

Todas las comparaciones se realizaron en una lista con 1000 elementos. En las tres comparaciones realizadas, el uso de un mutex por nodo es el menos eficiente de todos. El mutex de lectura y escritura resulta ser más eficiente cuando se realizan más operaciones de lectura (búsqueda) que operaciones de escritura (borrar e insertar). Un mutex por lista

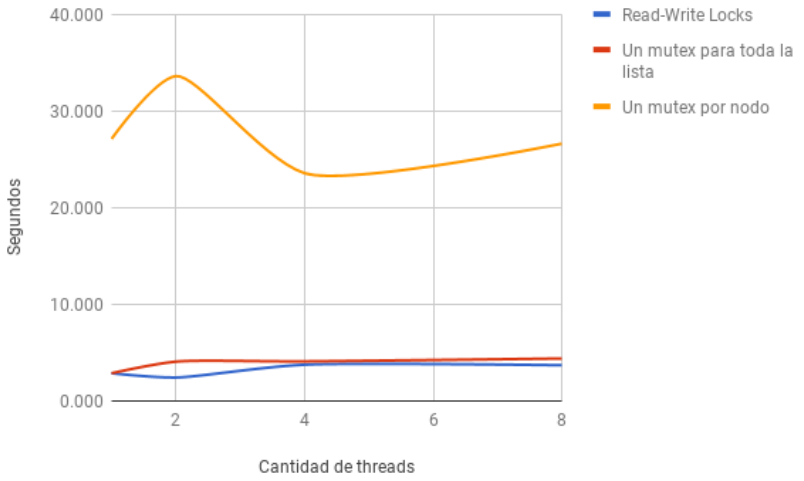

Figura 9. Gráfico de la tabla 6, 80 % búsquedas, 10 % eliminaciones, 10 % inserciones

Método	Cantidad de threads			
	1	2	4	8
Read-Write Locks	8.800	19.451	25.366	25.537
Un mutex para toda la lista	8.797	11.123	11.805	11.623
Un mutex por nodo	79.045	85.409	58.162	59.215

Cuadro 7. Cantidad de operaciones: 100000, 50 % búsquedas, 25 % eliminaciones, 25 % inserciones. Tiempo en segundos

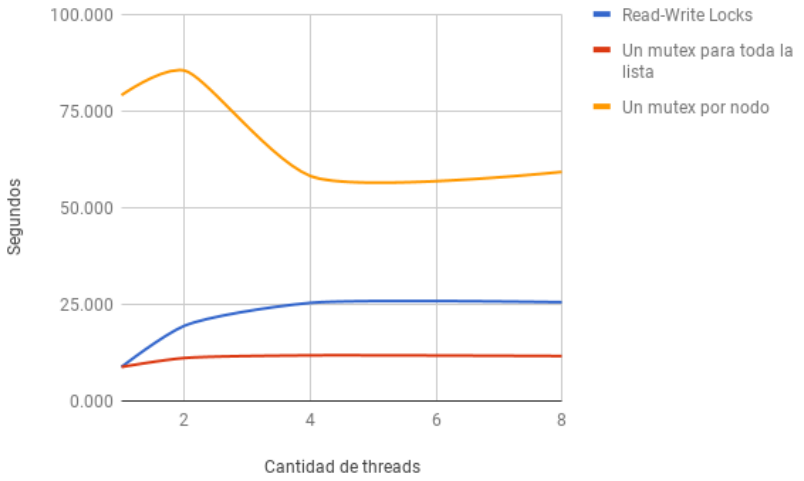

Figura 10. Gráfico de la tabla 7, 50 % búsquedas, 25 % eliminaciones, 25 % inserciones

es más eficiente cuando se realizan muchas operaciones de escritura. En todos los casos, el tiempo de ejecución del programa secuencial resulta ser mejor o similiar al tiempo de ejecución del programa paralelizado.

4. Multiplicación de matriz por vector

Este algoritmo consiste en dividir el total de filas de matriz en los distintos threads y multiplicarlas por el vector, luego agregar todos los resultados para obtener el resultado general.

Threads	Tamaño de la matriz				
	8000000 x 8	80000 x 800	8000 x 8000	800 x 80000	8 x 8000000
1	179.806	176.245	169.549	157.802	199.179
2	110.393	89.402	99.283	97.190	234.738
4	76.014	52.665	60.121	58.654	177.853
8	79.251	52.995	53.363	60.019	195.720

Cuadro 8. Tiempos de ejecución del algoritmo, usando distintos tamaños de matriz y distintas cantidades de threads. Tiempo en milisegundos

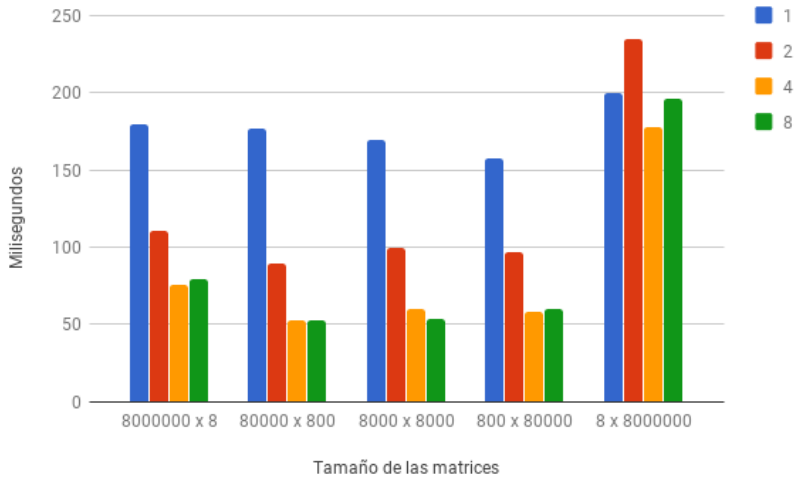

Figura 11. Gráfico de la tabla 8

En esta comparación se aprecia el impacto del uso de la memoria caché en este algoritmo, todas las operaciones se realizan con la misma cantidad de elementos (64000000), lo que cambia son las dimensiones de la matriz. El menor tiempo de ejecución se encuentra en el vector cuya cantidad de elementos, que es igual a la cantidad de columnas de la matriz, pueda entrar y permanecer más tiempo en memoria caché. En la imagen 11 se observa que los tiempos de ejecución del algoritmo con una matriz de tamaño 8 x 80000000 es el peor de todos, debido a que el vector, con 8000000 es demasiado grande como para poder entrar completamente en memoria caché, cada vez que un thread requiere accesarlo, necesita buscarlo en memoria principal, lo que ocasiona un deterioro en el desempeño.