Práctica 1 | Multithreading en C++

Autor: Nicolás Amigo Sañudo

Fecha: 13/01/2021

Prefacio

En general, la práctica no ha sido sencilla ya que el lenguaje de programación C no lo tenemos tan trabajado como Java. Además, emplear sentencias en C++ para conseguir el paralelismo ha aumentado la dificultad de la práctica. Por lo tanto, ha habido dos frentes. El primero, volver ha coger soltura con C/C++; y el segundo, aprender a paralelizar un programa. A pesar de ello, me ha resultado "divertido" tener que enfrentarme a hacer un programa desde cero debido a que hace tiempo que no lo hacía. Si que es verdad que la extensión de la práctica es demasiada a mi parecer. En mi opinión, se debería dividir en dos para poder realizar una práctica por semana y así no acumular trabajo. De esta forma, nos podemos centrar mejor en cada nuevo concepto.

Como punto positivo me llevo aprender a optimizar un programa. Hasta ahora no habíamos incidido mucho en ello en la carrera y considero que es de lo más importante ya que marca la diferencia.

Sistema

Resultados obtenidos mediante el comando lscpu:

Arquitectura:	x86_64
Modo(s) de operación de las CPUs:	32-bit, 64-bit
Orden de los bytes:	Little Endian
CPU(s):	8
Lista de la(s) CPU(s) en línea:	0-7
Hilo(s) de procesamiento por núcleo:	2
Núcleo(s) por «socket»:	4
«Socket(s)»	1
Modo(s) NUMA:	1
ID de fabricante:	GenuineIntel
Familia de CPU:	6
Modelo:	142
Nombre del modelo:	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
Revisión:	10
CPU MHz:	800.007
CPU MHz máx.:	4000,0000
CPU MHz mín.:	400,0000
BogoMIPS:	3984.00
Virtualización:	VT-x
Caché L1d:	32K
Caché L1i:	32K
Caché L2:	256K
Caché L3:	8192K
CPU(s) del nodo NUMA 0:	0-7

Diseño e Implementación del Software

Para llevar a cabo el apartado **Base**, se ha establecido en primer lugar el conjunto de enumerados y estructuras necesarias. Los enumerados se corresponden a los modos de funcionamiento, Single y Multi-thread; y a las operaciones que se pueden realizar, suma, resta y xor. Las estructuras almacenan las variables que necesitan los hilos para el desarrollo de las operaciones. Por un lado, la estructura datos contiene un puntero a la variable double relativa al array de los datos con los que se operará. Asimismo, alberga el puntero al futuro array con los datos restantes de *array que utilizará el primer hilo en terminar. La estructura de opciones posee información como el tamaño de los anteriores array, la operación junto con el modo de funcionamiento y el numero de threads en caso de multithreading.

Al inicio de la función main() se crean las distintas variables, entre ellas, las de los tipo de estructuras definidas anteriormente. Se inicializan según los datos que se obtienen de la consola. Se realiza una pequeña gestión de errores de la command-line interface para evitar ejecuciones erróneas. Además, se asigna un espacio de memoria a cada uno de los array mediante la sentencia malloc.

En segundo lugar, antes de la creación de los threads se inicializa el array con los datos de la operación. En caso de que sobren datos, también se inicializa el array_restante.

Los hilos que se encargan de la paralelización de la operación son creados y ejecutan el método funcion(). Esta función recibe como parámetros punteros a las estructuras explicadas anteriormente con los datos ya inicializados, un puntero a una estructura relativa al logger, otro al array con los resultados que se obtendrán y el identificador de cada hilo según su orden de creación. Dentro se realizan diferentes gestiones según el caso en concreto. Si el número de hilos es mayor que el tamaño del array entonces se asigna al array de resultados directamente la parte correspondiente del array de datos. Cuando el número de hilos es igual o menor cada uno de ellos recibe una parte proporcional del trabajo. Si no es posible la asignación igual para cada thread, el primero que llegue al final de la función se encarga del array restante. Esto se consigue mediante una sección crítica manejada mediante un mutex. De esta forma el primer hilo que tome dicho mutex cambia la variable global g_restante_operado evitando que el resto de hilos operen con el array restante.

Al terminar los hilos su trabajo se sincronizan mediante join(). Posteriormente, se obtienen las soluciones recorriendo el array en el cual cada hilo ha depositado sus resultados.

Para obtener el tiempo de ejecución, antes de la creación de los threads y una vez sincronizados se hace uso de la API que se indica para ello. El tiempo es mostrado en micro segundos.

*Cabe destacar que no existe diferencia entre el modo de funcionamiento single y el de multi-threading mediante un único hilo. Durante el desarrollo del programa también se implementó el modo single en el cual el main() realiza el trabajo sin crear un nuevo hilo. Se ha dejado así ya que se comentó en clase que no es necesario el cambio para evitar una sobrecarga de trabajo.

En cuanto al apartado **Logger**, se ha desarrollado una funcion_logger() la cual ejecuta el thread logger. Este es creado antes de los hilos del paralelismo si la variable FEATURE__LOGGER está definida. Por tanto, se utiliza la compilación condicional para su creación. El método que ejecuta recibe como parámetros un puntero a un tipo de estructura definida especialmente para el logger y otro a los parámetros de la operación. La estructura datoslogger contiene el número de creación de hilo con el que está trabajando, el número total que lleva trabajados, el resultado parcial del hilo siendo trabajado, un puntero a un array con todos los resultados y el resultado total.

La función está compuesta por un bucle while que se repite hasta que el logger haya obtenido los resultados de todos los hilos. Al comienzo de dicho bucle realiza un wait de forma que sea despertado por el hilo que pasa los resultados. Por tanto, en funcion() se utiliza un notify para avisar al logger. Una vez obtenido los resultados y realizado las operaciones pertinentes se notifica al siguiente hilo que realiza una espera wait con otra variable condicional. Cuando se hayan conseguido todos los valores se calcula el resultado total. Finalmente, el logger notifica al main() mediante una variable condicional.
Luego, antes del aviso el main() ha estado esperando mediante wait. Después se sincroniza el logger y se obtienen y comparan los resultados.

Para el apartado de **Optimización** se ha valorado la inclusión de variables atómicas que puedan mejorar los tiempos de ejecución. Tras pensar donde poder incluirlas se ha llegado a la conclusión que su uso en el programa desarrollado no es viable y no mejoraría el rendimiento. Al calcularse el resultado total en el main() a partir de los resultados parciales de los hilos no es posible el empleo de una variable atómica double correspondiente al resultado total. De todas formas, para demostrar el conocimiento de su uso se ha incluido una variable atómica booleana relacionada con la gestión del array restante. Su utilización no mejora en líneas generales el tiempo.

Metodología y desarrollo de las pruebas realizadas

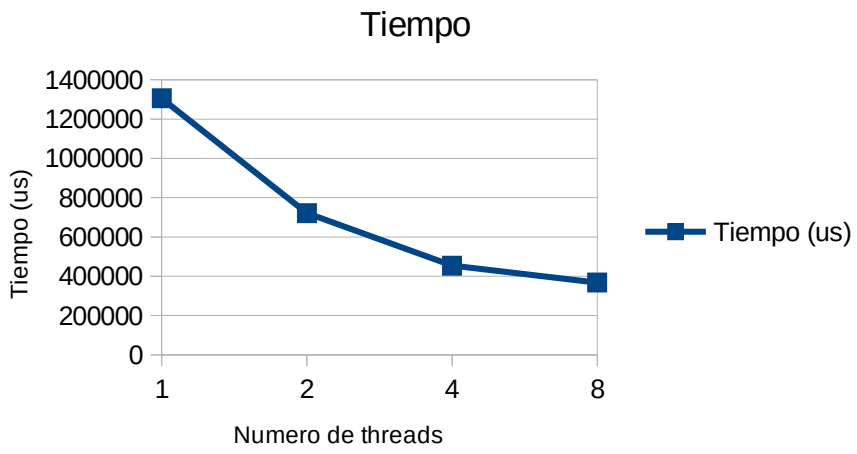
En el apartado **Base** se ha realizado un proceso de captura de tiempos de ejecución con el objetivo de comprobar el funcionamiento del paralelismo. Para ello, se han utilizado tres tamaños de array con la operación SUMA. Se emplean 1, 2, 4, y 8 hilos. Asimismo, se muestran tres gráficas por cada tamaño.

Tamaño: 99.999.999.999

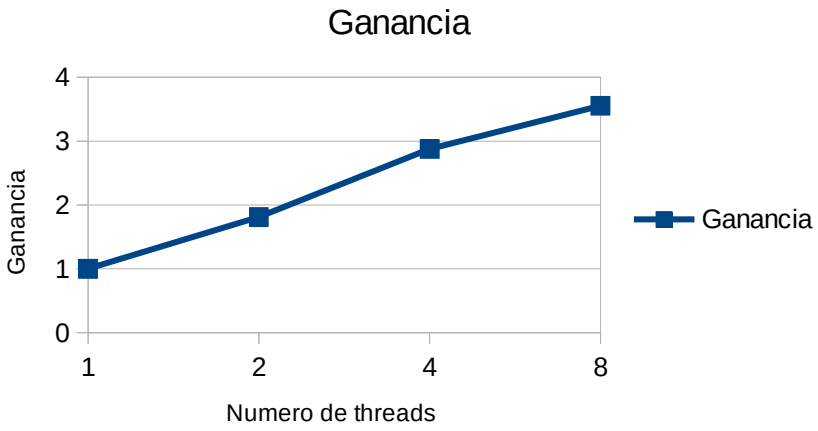
Tras la realización de dos conjuntos de 10 ejecuciones se obtienen los siguientes resultados con cada uno de los parámetros indicados. Siendo el tiempo la media de todas las ejecuciones realizadas en micro segundos.

Max hilos		8		
Operacion	Suma			
Tamaño	99999999999			
nThreads	Tiempo (us)	Ganancia	Eficiencia	
1	1306098	1	0,125	
2	721374	1,8105698292	0,2263212287	
4	454023	2,8767221044	0,359590263	
8	367575	3,5532830035	0,4441603754	

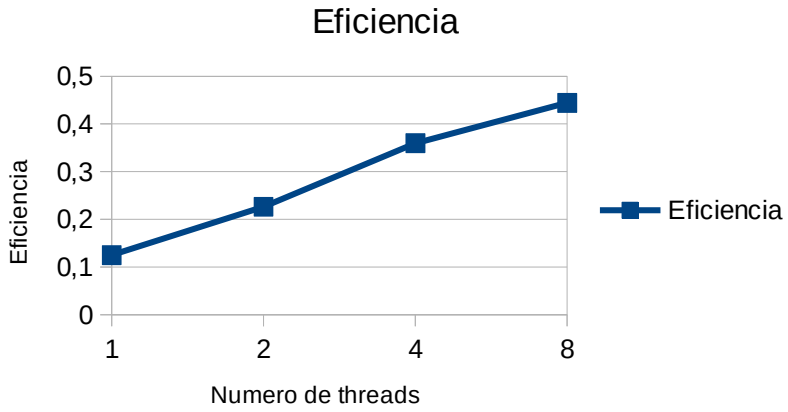
La gráfica correspondiente al tiempo en micro segundos refleja un descenso del tiempo de ejecución a medida que aumenta el número de hilos utilizados. Se puede observar una fuerte caída al pasar de una ejecución mediante un único hilo a otra con dos. Este decremento se suaviza a medida que se llega al límite de hilos establecido por el computador.



En cuanto a la ganancia o speedup, se observan mejoras en el rendimiento del programa con el aumento del número de hilos utilizados produciéndose una mejora lineal.

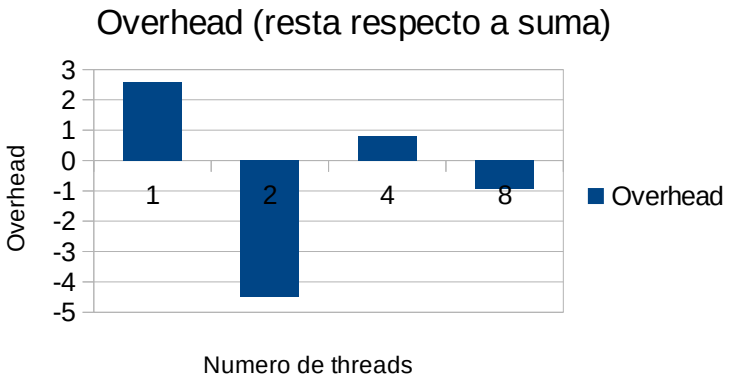


La eficiencia del paralelismo aumenta a media que lo hace el número de hilos ejecutándose. Por lo tanto, en este caso es prácticamente proporcional a dicho número. Con el máximo de hilos del computador llega a situarse cerca de 0.5.



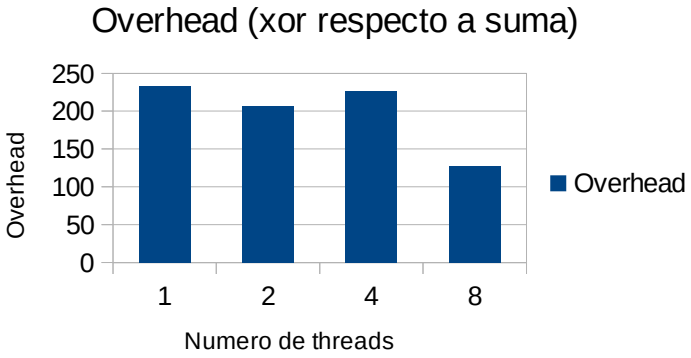
En cuanto a los overheads de la resta respecto a la suma, se puede observar que no hay un patrón que determine si una u otra operación es más rápida. Se producen el mismo número de casos, según el número de hilos, en el que una gana ala otra y viceversa. Esto se debe a que la resta presenta prácticamente el mismo coste al ser una suma de números negativos.

Tamaño	99999999999		
nThreads	Suma	Resta	Overhead
1	1306098	1340109	2,6040159314
2	721374	688880	-4,504459545
4	454023	457689	0,8074480808
8	367575	364120	-0,939944229



Mientras, la comparativa entre la operación xor y la suma muestra grandes overheads. Estos son más predominantes en las ejecuciones con número de hilos más bajos. Es coherente ya que xor es una operación más costosa al ser bit a bit.

Tamaño	99999999999		
nThreads	Suma	Xor	Overhead
1	1306098	4341814	232,42635698
2	721374	2215775	207,16036342
4	454023	1482007	226,41672338
8	367575	833513	126,75998096

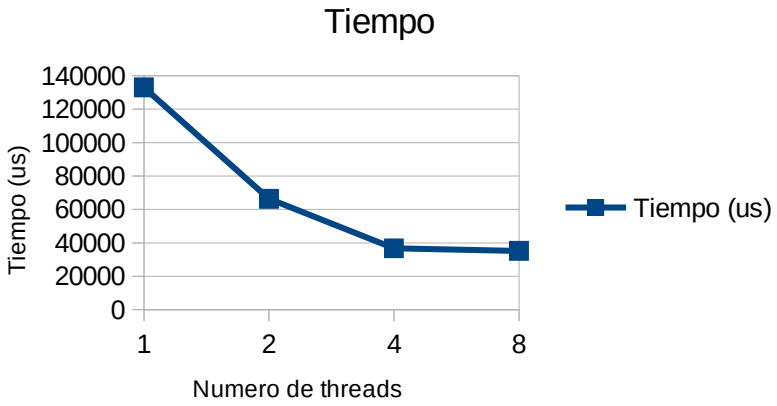


Tamaño: 99.999.999

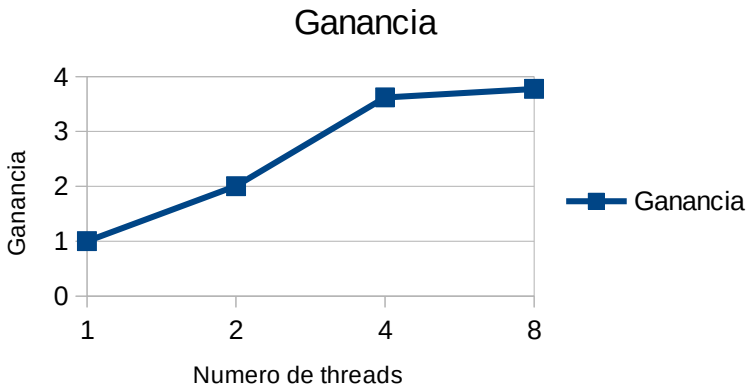
Con el nuevo tamaño se puede observar una caída de los tiempos debido a que es menor que el utilizado anteriormente.

Max hilos	8		
Operacion	Suma		
Tamaño	99999999		
nThreads	Tiempo (us)	Ganancia	Eficiencia
1	133000	1	0,125
2	66345	2,004672545	0,2505840681
4	36732	3,6208210824	0,4526026353
8	35225	3,7757274663	0,4719659333

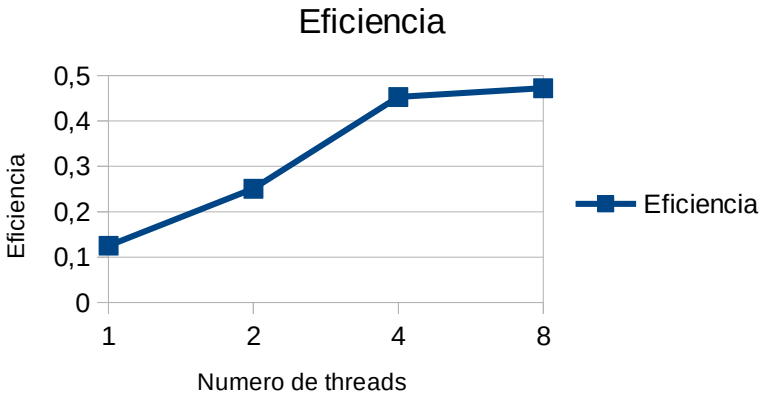
El tiempo de ejecución disminuye a medida que se aumenta el número de threads al igual que en el anterior caso. Sin embargo, el tiempo correspondiente al de 8 hilos presenta una mejora mucho menor. Esto es debido a que el tamaño de array no es lo suficientemente grande para aprovechar dicho número de hilos. Al aumentar el número de threads también lo hace los overheads derivados de su procesamiento.



Por tanto, como el tiempo con 8 hilos prácticamente no mejora, la ganancia con este número de threads se suaviza respecto a la anterior.

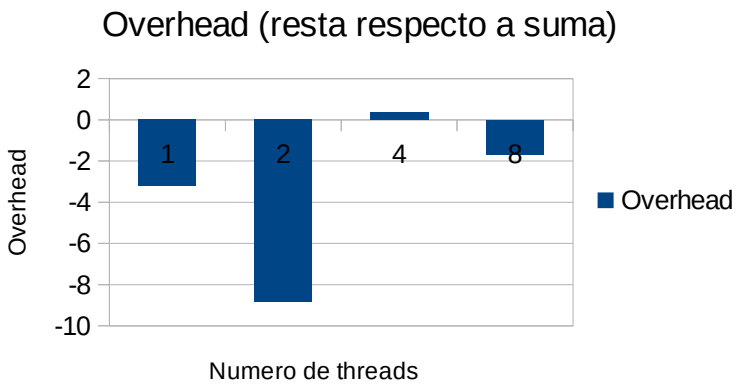


Comparando la eficiencia de este tamaño de array con el anterior, se puede observar que en este caso la diferencia entre 4 y 8 hilos es mínima. Sólo se produce una ligera subida. Esto se debe a que el tamaño no es lo suficientemente grande para que sea eficiente emplear 8 hilos.



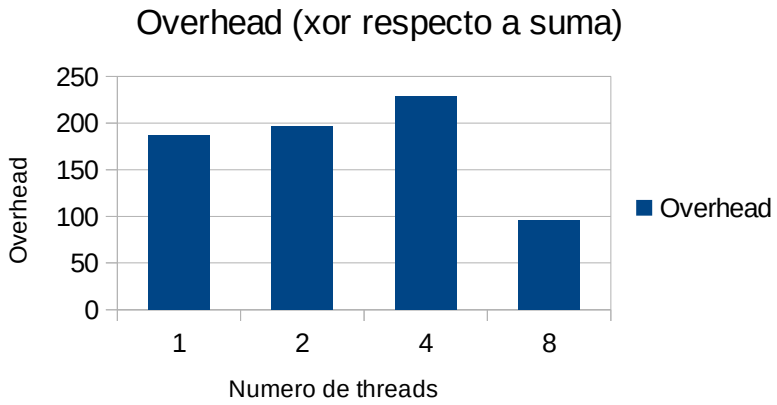
Al igual que en el tamaño anterior, la resta no muestra signos de presentar mayores overheads que la suma. En este caso, incluso las ejecuciones de la resta son más rápidas. De todos modos, no se puede concluir que esta operación sea más rápida debido a que no hay un modelo que lo determine.

Tamaño	99999999		
nThreads	Suma	Resta	Overhead
1	133000	128721	-3,217293233
2	66345	60472	-8,852211923
4	36732	36858	0,3430251552
8	35225	34626	-1,700496806



Como en el tamaño anterior, se muestran grandes overheads de la operación xor. El menor se corresponde con el mayor número de hilos (8).

Tamaño	99999999		
nThreads	Suma	Xor	Overhead
1	133000	381613	186,92706767
2	66345	196884	196,75785666
4	36732	120798	228,86311663
8	35225	68791	95,290276792

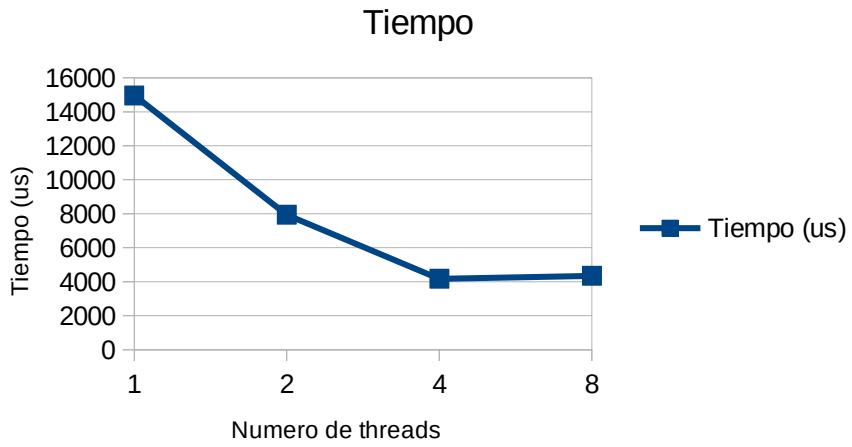


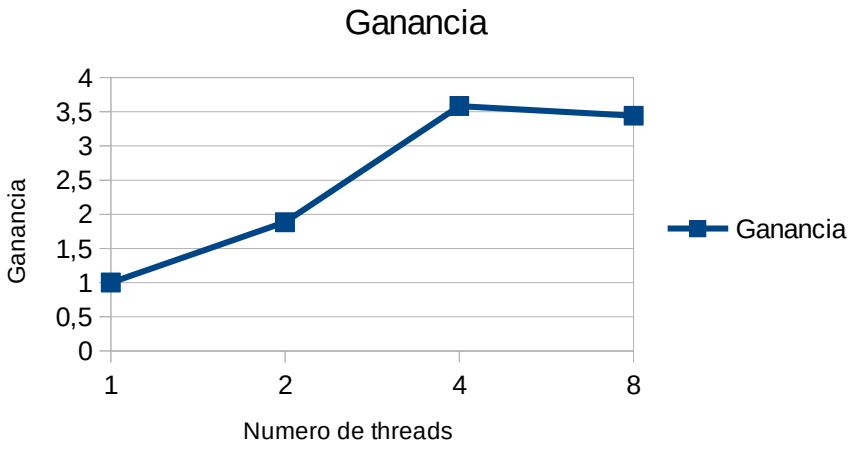
<u>Tamaño: 9.999.999</u>

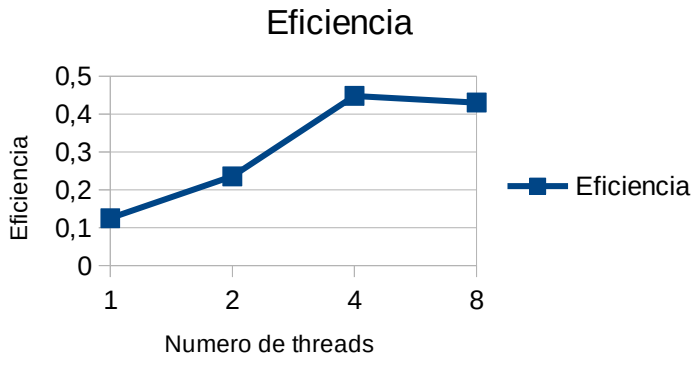
Como en el último caso, los tiempos vuelven a descender con el tamaño del array. En este caso, se puede observar que con 8 hilos se produce un ligero aumento respecto a la ejecución con 4 hilos.
En consecuencia, tanto la ganancia como la eficiencia disminuyen con este número de hilos.

Se puede concluir que con tamaños menores al empleado, el aumento en el número de threads del paralelismo podrá dar lugar a tiempos peores.

Max hilos	8		
Operacion	Suma		
Tamaño	99999999		
nThreads	Tiempo (us)	Ganancia	Eficiencia
1	14964	1	0,125
2	7941	1,8843974311	0,2355496789
4	4176	3,5833333333	0,4479166667
8	4348	3,4415823367	0,4301977921

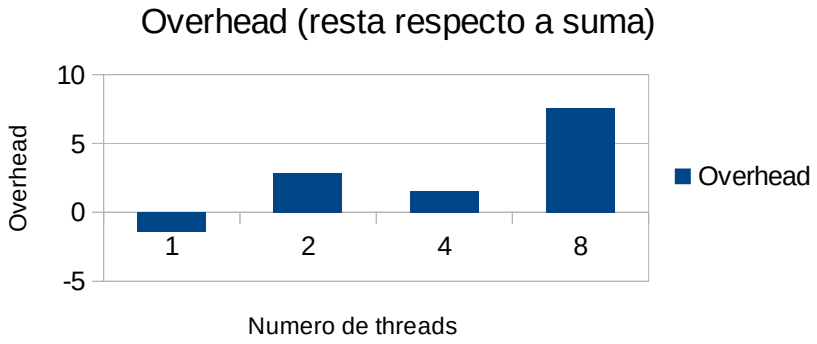






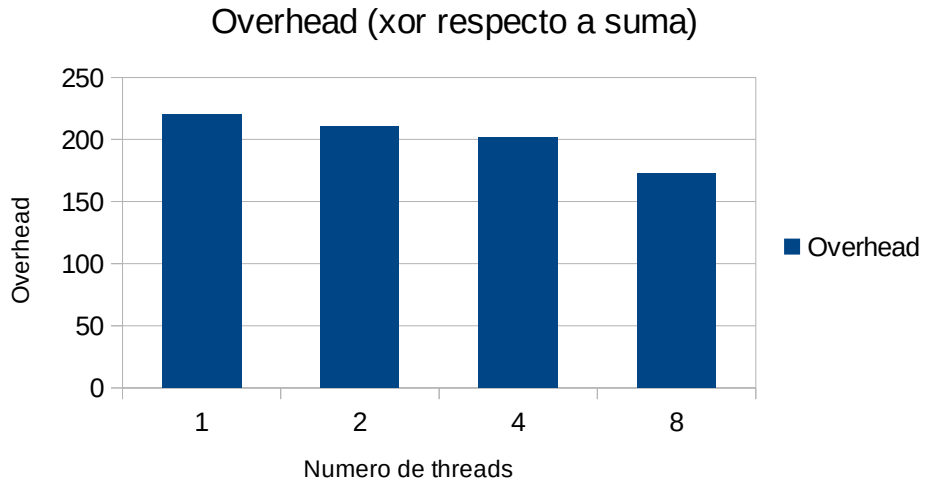
En este tamaño, se puede ver que la resta da resultados peores en comparación con la suma. En conclusión, vistos los resultados anteriores, se puede decir que no existe una operación más rápida que la otra ya que sus diferencias son mínimas. Por tanto, a la hora de realizar benchmarking, en estas condiciones, no es posible demostrar que la suma vaya a arrojar mejores tiempos.

Tamaño	9999999		
nThreads	Suma	Resta	Overhead
1	14964	14756	-1,390002673
2	7941	8169	2,871174915
4	4176	4241	1,55651341
8	4348	4677	7,5666973321



En este tamaño se vuelven a mostrar los mismos resultados que en los anteriores. Por tanto, se puede concluir estableciendo que la operación xor es mucho más costosa que la suma.

Tamaño	9999999		
nThreads	Suma	Xor	Overhead
1	14964	47911	220,17508688
2	7941	24660	210,54023423
4	4176	12594	201,58045977
8	4348	11849	172,51609936



Como se ha explicado anteriormente, la **Optimización** del paralelismo mediante el empleo de variables atómicas no se produce. Esto es debido a que situaciones en las que podría ser beneficioso su uso no existen. Por ejemplo, en el paralelismo una variable que almacene el resultado total obtenido entre los hilos.

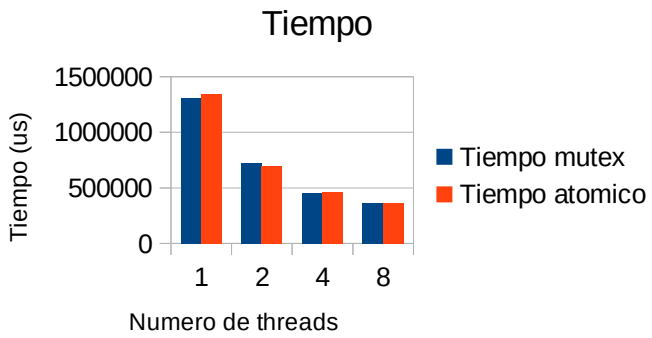
En este caso, se utiliza en la gestión de la parte restante a realizar por el hilo más rápido, la cual en la parte normal es protegida por un mutex.

Para llevar a cabo la comparación entre la programación mediante el mutex y la variable atómica obtenemos sus tiempos utilizando tres tamaños diferentes de array. En cada uno de ellos se ha usado de 1 a 8 hilos.

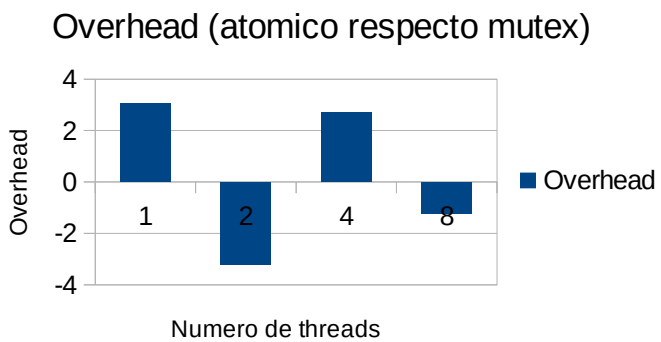
Tamaño: 99.999.999.999

Para este primer tamaño, se pueden ver tiempos muy similares entre el programa que emplea mutex y el supuestamente optimizado con una variable atómica. En función del número de hilos, tendrá lugar una mejora o no.

Operacion	Suma	
Tamaño	99999999999	
nThreads	Tiempo mutex	Tiempo atomico
1	1306098	1346097
2	721374	698145
4	454023	466368
8	367575	363075



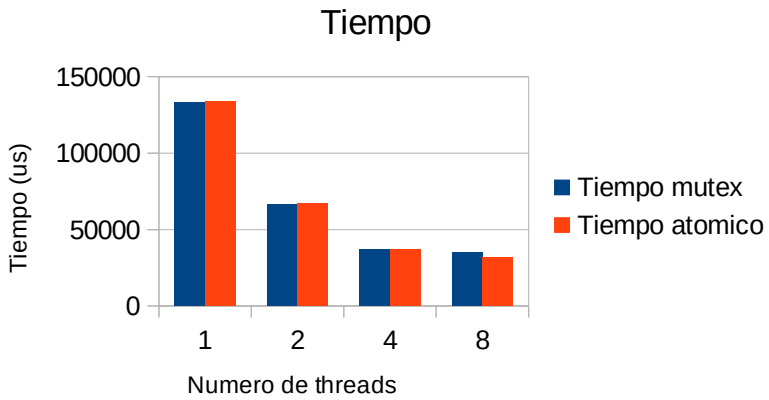
En cuanto a los overheads, no existe un patrón que defina que se produce una optimización al emplear una variable atómica. Con 1 y 4 hilos tiene lugar un claro overhead producido por dicha variable. Mientras que con 2 y 8 hilos se desarrolla una mejora.

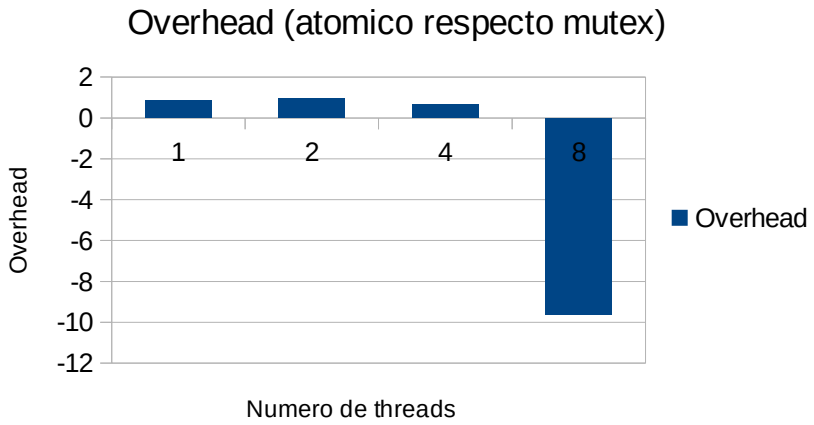


Tamaño: 99.999.999

Para este nuevo tamaño de array, el tiempo de ejecución empleando la variable atómica unicamente es mejor con 8 hilos. En el resto de casos, se produce un ligero aumento del tiempo. De nuevo, se puede observar que no existe un patrón que certifique que la optimización.

Operacion	Suma	
Tamaño	99999999	
nThreads	Tiempo mutex	Tiempo atomico
1	133000	134121
2	66345	66985
4	36732	36981
8	35225	31837

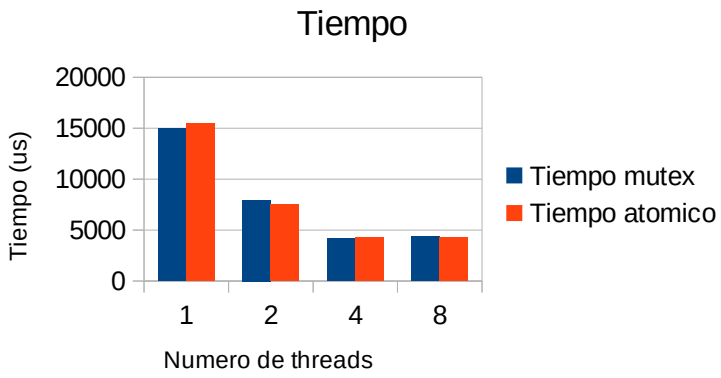


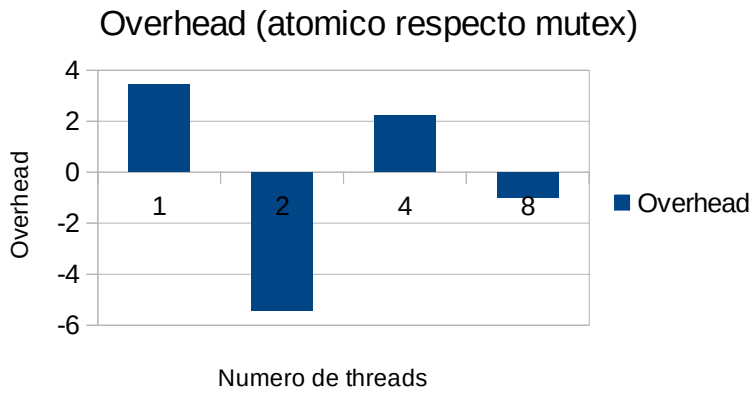


Tamaño: 9.999.999

Por último, se puede comprobar de nuevo que solo en la mitad de los casos se produce una mejora en el tiempo correspondiente a la variable atómica. Por tanto, no es posible concluir que dichos decrementos sean debido a su uso, ya que en el resto de casos el tiempo aumenta notablemente.

Operacion	Suma	
Tamaño	9999999	
nThreads	Tiempo mutex	Tiempo atomico
1	14964	15480
2	7941	7509
4	4176	4269
8	4348	4305





Discusión

Una de las cuestiones que me planteé durante la práctica fue cómo obtener el resultado total de la operación. Por un lado, medité la posibilidad de emplear una variable que recogiera los resultados de todos los hilos. De forma que cada uno de ellos realizara la operación sobre dicha variable. Esto daría lugar a la utilización de un mutex para el valor que está siendo operado por los threads. En consecuencia, todos los hilos tendrían que tomar el mutex para poder sumar sus resultados al terminar de obtenerlos. Eso daría lugar a que en caso de que un hilo estuviera realizando dicho proceso los demás tuvieran que esperar. Finalmente, decidí calcular el resultado total en el main() para así evitar emplear un mutex o una variable atómica que protegiera la variable.

t

En este apartado se aportará una valoración personal de la práctica, en la que se destacarán aspectos como las dificultades encontradas en su realización y cómo se han resuelto.

Preguntas

1. Explica cómo has hecho los benchmarks y qué metodología de medición has usado (ROI+gettimeofday, externo).

   El proceso de benchmarking es explicado en el apartado *Metodología y desarrollo de las pruebas realizadas*.

   Se utilizado como metodología de medición ROI + gettimeofday. Se ha definido como la región de interés el bloque de código que comprende desde la creación de los hilos del paralelismo hasta su sincronización. Luego, abarca todo el proceso de las operaciones que se realizan en paralelo. Como la obtención del resultado final se realiza en el main(), esta parte queda excluida de la ROI. Los tiempos se obtienen mediante la API nombrada y son mostrados al final de la función principal como micro segundos.