Cuestionario sobre threads y concurrencia

Proyectos de arquitectura distribuida

La Salle - Universitat Ramón Llull

Nombres: Víctor Garrido Martínez y Matias Villarroel Hagemann

Data: 16.10.2019

Índice

1 Ejercicio 1

Explica cuáles son las ventajas y las desventajas que tiene el paradigma de la programación paralela (*shared memory*) frente al paradigma de la programación distribuida.

1.1 Ventajas

- No sufre de *network bottlenck* a diferencia de la distribuida
- La información no sufre de tanto retraso a la hora de ser actualizada, si se compara a una estructura distribuida con la misma cantidad de nodos
- Es más fácil diseñar e implementar una arquitecura shared memory ya que existen herramientas como los semáforos que ayudan a lidiar con la concurrencia. Por el contrario, en la programación distribuida, la concurrencia es mucho más complicada administrar.

1.2 Desventajas

- No se puede distribuir geográficamente debido a que los componentes que componen el sistema deben de permanecer conectados mediante un bus. En cambio, la distribuida si lo permite mediante la conexión a la red.

- Presenta un Single Point Of Failure (SPOF), en este caso la memoria. En cambio, la distribuida presenta una redundancia con cada una de las máquinas que forma el sistema por lo que es más fiable

- El acceso a la información es concurrente, por esa razón la disponibilidad de los datos es inferior que, en una programación distribuida, dónde los datos disponen de varios puntos de acceso

- No son igual de escalables debido a la *shared memory*, contra más "cpu + memoria local" se añaden, más peticiones recibe la misma y más lento se vuelven el acceso a los datos. Hay que tener en cuenta también, que la distribuida puede sufrir de *network bottlenck*, cosa que puede afectar a la escalabilidad también.

- No son heterogéneos ni modulares, por lo que carecen de flexibilidad para hacer modificaciones al sistema.

- No tienen la misma viabilidad económica que la distribuida, debido a la reducción de precio que ha sufrido la banda ancha y las *workstations*.

2 Ejercicio 2

Describe tres tipos de aplicaciones reales de programación paralela y tres tipos de aplicaciones reales de programación distribuida

2.1 Paralela

- Matemáticas
- Bases de Datos
- Diagnóstico médico por la imagen

Fuente : (Blaise Barney, 2019)

2.2 Distribuida

- Distribución de contenido Peer-To-Peer (P2P)
- Content Delivery Network (CDN)
- Banca online

3 Ejercicio 5

Modifica el ejercicio anterior para que los *threads* accedan al *array* vía memoria compartida. Compara los tiempos de búsqueda con el ejercicio anterior y justifica las diferencias. Experimenta con diferentes tamaños del *array* y diferentes números de *threads*.

Para la realización de las pruebas comparativas, se ha procurado de diseñarlas de manera que sean lo más justas posibles. Para hacerlo, hemos mantenido la misma estructura del código en los dos casos y sólo modificado la parte que los diferencian. El tiempo inicial, se toma en el instante en el que se lanza el primer thread, y en el momento en que se lanzan los threads, estos ya han estado creados y preparados previamente de modo que en el momento de realizar pruebas no se incluye tiempo gastado en la preparación y creación de estos. En definitiva, el tiempo medido en las pruebas corresponde únicamente con la ejecución de la función de búsqueda.

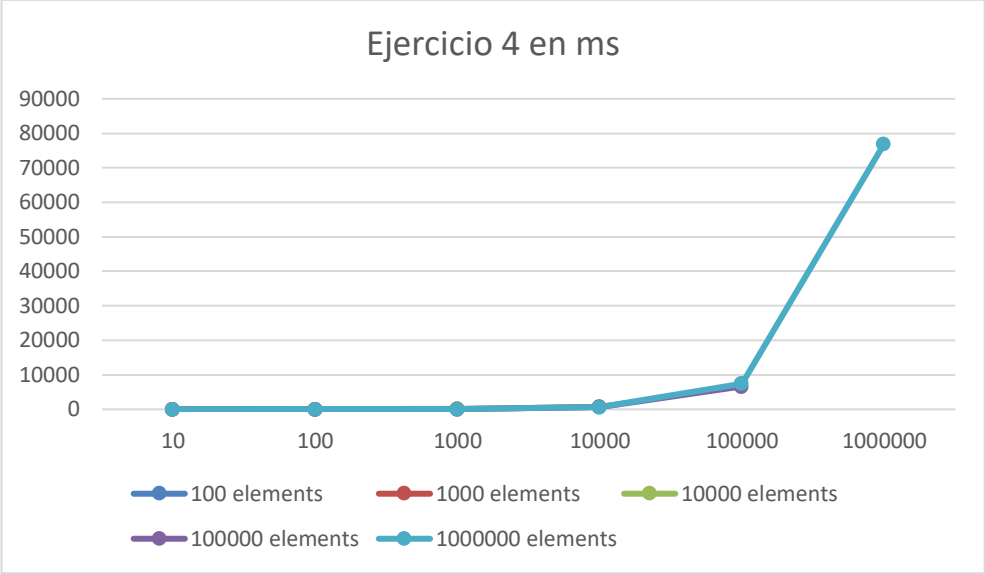
También, hemos tenido en cuenta la cache, puesto que la primera ejecución de la función de búsqueda era más costosa que las siguientes, dado que esta copiaba el array de búsqueda en la cache, haciendo subir su tiempo considerablemente. Naturalmente, esto no hacía las comparativas justas. La primera ejecución de la función de búsqueda partía de una clara desventaja que no tenían las siguientes, dado que se aprovechaban de esa cache que tanto había penalizado a la primera ejecución de la función de búsqueda. Para solucionar dicho fenómeno, tomamos la decisión de, por cada nuevo array de tamaño n de búsqueda generado, se realiza una ejecución previa de la función de búsqueda, con el único propósito de forzar que se guarde el array en la cache. De esta manera, las pruebas se realizan en *hot* y no en *cold*.

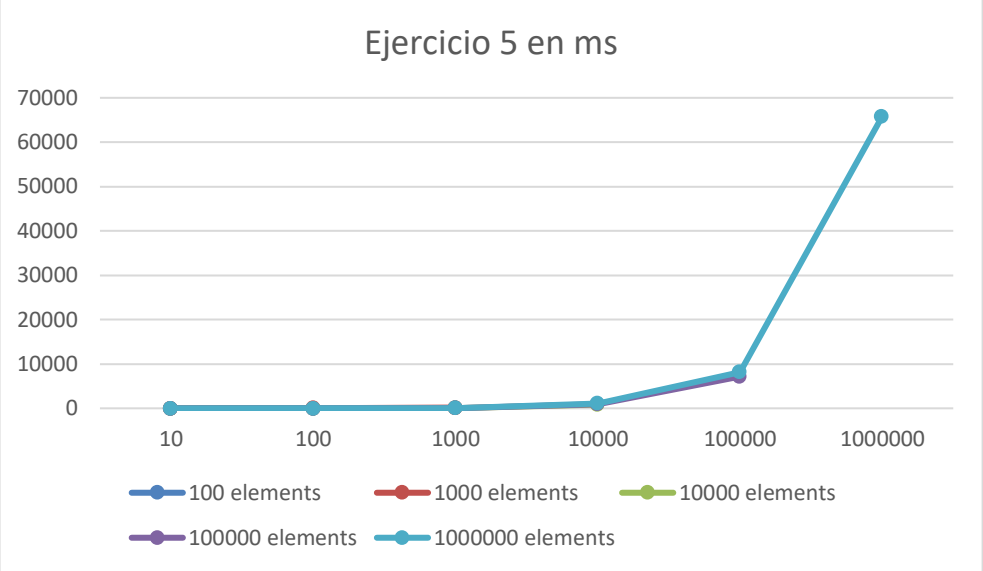
Antes de llegar a la solución, decidimos experimentar haciendo un seguido de pruebas en cold, es decir, ejecutar en el main una única vez la función de búsqueda y modificar sus parámetros manualmente. Los resultados eran muy variantes y no nos generaban confianza. Esto podría ser debido a como interactúa la máquina virtual de Java, con el sistema operativo, el scheduler y otros elementos del hardware para realizar la copia en la cache.

También cabe destacar, que pese a ser unas pruebas en *hot*, los resultados también variaban con cada una de las ejecuciones, y esto también podría deberse a elementos ya mencionados, como la máquina virtual de Java, el scheudler (dependiendo de si hay o no otros programas abiertos utilizando recursos del sistema en el background) o el estado del sistema en el momento.

	SN	SM	SN	SM	SN	SM	SN	SM	SN	SM
ArraySize / Threads	**100**		**1000**		**10000**		**100000**		**1000000**	
10	**2**	**3**	**1**	**1**	**2**	**8**	**1**	**1**	**1**	**1**
100	**17**	**19**	**24**	**36**	**15**	**21**	**8**	**6**	**7**	**6**
1000			**120**	**178**	**120**	**105**	**61**	**78**	**64**	**85**
10000					**712**	**852**	**682**	**1006**	**642**	**1139**
100000							**6647**	**7158**	**7485**	**8210**
1000000									**76958**	**65778**

SN: Shared Nothing (Ejercicio 4)
SM: Shared Memory (Ejercicio 5)





En ambos casos, podemos observar que contra más threads, más tiempo tarda en realizar la búsqueda. Esto es debido a las limitaciones de una CPU de un ordenador de 13 pulgadas dirigido para el público general. Tiene sólo 2 núcleos y gracias a la tecnología HyperThreading dispone de dos subprocesos por núcleo, sumando un total de 4.

A partir de ahí, añadir más threads en esta máquina, en vez de ayudar complica, dado que debe de mantener todos los demás threads en espera y hacer todos los cambios de contexto necesarios para cada thread por lo que añade tiempo.

Por regla general, *Shared Memory* tarda más tiempo que *Shared Nothing,* y en ejecuciones en arrays relativamente pequeños, 10, 100, 1000, 10000 se puede observar como la teoría se confirma.

Ahora bien, si nos fijamos en lo que ocurre cuando tenemos un array de un millón y tenemos un millón de threads, hay una clara diferencia entre Shared Nothing y Shared Memory. Por primera vez, Shared Memory tarda substancialmente menos que Shared Nothing.

Esto es debido a que las pequeñas copias del array de búsqueda que tienen los threads de Shared Nothing superan en tamaño al tamaño de la cache, por la que esta debe de ser constantemente reemplazada, bajando el rendimiento.
En Shared Memory, la cache sólo tiene una copia del array, por lo que cada búsqueda tiende a generar un cache hit.

4 Ejercicio 6

Justifica cuál es la diferencia entre usar el método `run`, el método `start` de la clase `thread`

Cuando se llama al método start(), este crea un thread nuevo y luego ejecuta el método run() en este nuevo thread. Por el contrario, al llamar al método run() directamente, no se crea ningún thread nuevo, por lo que el método se ejecuta en el thread actual.

Otra diferencia es que el método start() NO puede ser llamado más de una vez, mientras que el método run() si. Si se vuelve a llamar al método start() salta una excepción.

5 Ejercicio 7

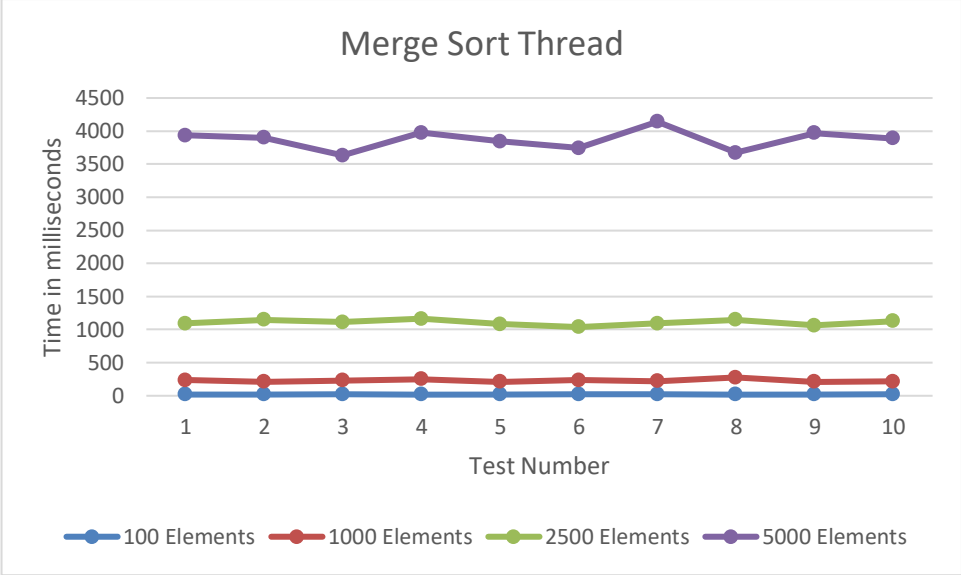
Escribe un programa *multithread* en Java que ordena un *array* recursivamente utilizando *merge sort*. Así, el *thread* principal crea 2 *threads* y cada uno va creando dos nuevos *threads* para ir ordenando la mitad del *array*

Para testear el merge con threads se crearon 4 arrays distintos de forma aleatoria. El primero contiene 100 elementos, el siguiente 1000, el que le sigue 2500 y el tercero 5000. Con estos números se puede apreciar la diferencia entre cuanto se tarda el algoritmo en procesar arrays más grandes y a su vez la diferencia entre usar el algoritmo de forma paralela o de forma secuencial.

El número máximo de elementos del array que se pudo ordenar con este método fue de aproximadamente 6000 elementos. Esto se debe a que como el algoritmo crea tantos threads, hay un punto en el que el sistema operativo ya no permite que la máquina virtual cree más threads y por ende sale el siguiente error “java.lang.OutOfMemoryError: unable to create new native thread”. En las pruebas se testó con 5000 ya que con este número ya se puede establecer la diferencia entre hacerlo paralelo o secuencial.

Merge Sort Thread

Tamaño/tiempo	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Avg
100 Elements	18	18	23	20	18	23	23	19	18	23	20,3
1000 Elements	234	210	230	250	213	237	221	277	212	216	230
2500 Elements	1092	1150	1114	1165	1081	1038	1095	1147	1062	1128	1107,2
5000 Elements	3935	3898	3633	3975	3845	3743	4141	3674	3968	3890	3870,2



Como se puede ver en la gráfica a medida que se agregan más elementos al array se tarda más en ordenarse. Sin embargo, se puede ver que el tiempo no incrementa de forma proporcional. Esto se debe a que cuando se van creando los threads, no sólo se añade el sobrecosto de crear los threds, si no también de tener que esperar a que estos terminen de ejecutarse, utilizando el join(), para poder hacer el merge. Este sobrecosto no es proporcional al número de threads.

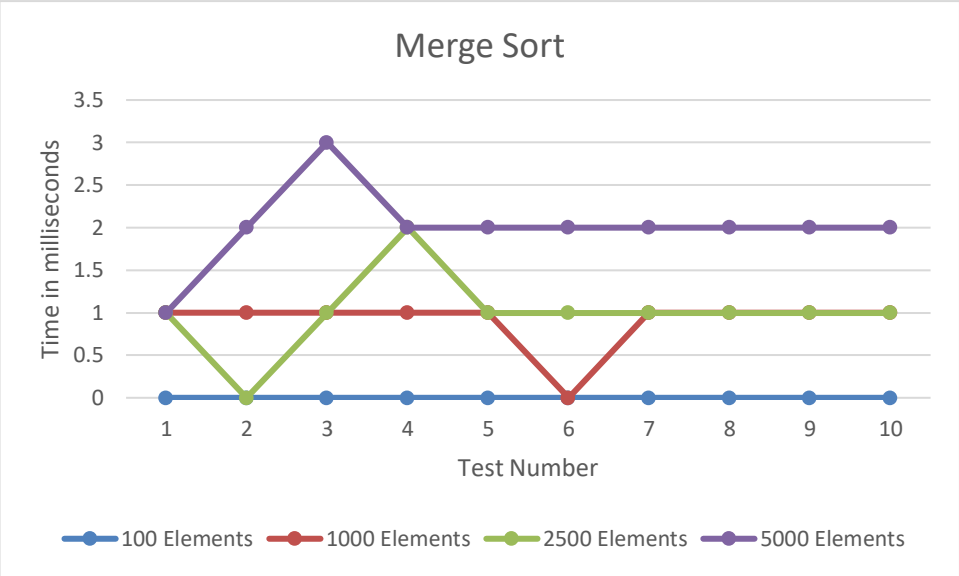
En la siguiente sección se muestra la misma tabla con los resultados del merge secuencial y se profundiza más en estos sobrecostos de utilizar threads

6 Ejercicio 8

Compara el tiempo de ejecución del ejercicio anterior con una ordenación secuencial. Justifica los resultados

Merge Sort

Tamaño/tiempo	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Avg
100 Elements	0	0	0	0	0	0	0	0	0	0	0
1000 Elements	1	1	1	1	1	0	1	1	1	1	0,9
2500 Elements	1	0	1	2	1	1	1	1	1	1	1
5000 Elements	1	2	3	2	2	2	2	2	2	2	2,2



Como se puede apreciar en la gráfica, al igual que antes, a medida que el número de elementos incrementa el tiempo que se tarda el algoritmo en ordenarlo también lo hace. A diferencia del algoritmo paralelo, el aumento del tiempo es mucho menor. Entre 1000 elementos y 5000 la diferencia no suele pasar de 1 mili, mientras que en el caso anterior son más de 3500 milisegundos.

En la gráfica también se puede ver que hay un pico inusual que llega a 3 milis en vez de 2. Sin embargo, esto no importa a la hora de mostrar la gran diferencia entre hacerlo secuencial o en paralelo.

Se cree que la diferencia en el tiempo en que se tarda se debe al método join(). Para poder ordenar de forma correcta usando threads hay que utilizar join() para esperar a que los dos threads hayan terminado para poder proseguir al merge para que se ordenen los arrays. Esto estipula una sobre carga en rendimiento al algoritmo. En la documentación de Oracle se especifica que el join() al igual que un sleep() dependen del sistema operativo, por lo tanto, por más que se le especifique el tiempo que se desea esperar, este tiempo no se puede cumplir de forma exacta. Esto ayuda a indicar que los join() agregan una sobrecarga ya que no es posible determinar por cuánto tiempo se va quedar el thread esperando. Como los threads dependen uno del otro, hasta que no se terminen de ejecutar, no se puede continuar. Si bien el tiempo perdido es poco, se puede evidenciar en la gráfica de "Merge Sort Thread" que esta espera impacta en el rendimiento, ya que el tiempo no escala proporcionalmente.

Además si se le especifica al join() un tiempo de espera, esto no garantiza que las operaciones dentro del thread hayan terminado y que se pueda proseguir al merge, por lo tanto no es una opción de mejora.
Para comprobar que los joins agregan un sobrecosto en el rendimiento se quitaron del código. En la siguiente tabla se pueden ver los resultados.

Merge Sort Thread SIN Joins

Tamaño/tiempo	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Avg
100 Elements	1	1	1	0	1	1	1	1	1	1	0,9
1000 Elements	1	0	1	1	1	1	1	1	1	0	0,8
2500 Elements	1	1	1	1	0	0	1	1	1	1	0,8
5000 Elements	1	1	1	2	1	1	0	2	1	1	1,1

Como se puede ver, el tiempo que toma el código en ejecutarse siempre tiende a 1 milisegundo. Esto indica que el sobre costo que se apreciaba en el rendimiento puede estar causado por el join(). Sin embargo, si no se utilizan los joins, el array no se ordena. Teniendo esto en cuenta se quiso ver que pasa si se hace una mezcla entre los dos algoritmos. Crear uno que empiece la ejecución de forma paralela usando los threads y que en cierto punto pase a ser un algoritmo secuencial. De esta forma se podría tener varios threads actuando de forma secuencial al mismo tiempo y optimizar la búsqueda.

Para probar la viabilidad del nuevo algoritmo se empezó con un array de 10.000 elementos, luego 100.000, 1.000.000 y 10.000.000. Uno de los puntos importantes en esta versión es que también depende del umbral que se decida usar para pasar de paralelo a secuencial. Esto se debe a que si es muy pequeño va a seguir sufriendo de los problemas que tiene el merge paralelo.

Merge Sort

Tamaño/tiempo	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Avg
10.000 Elements	5	5	4	4	5	4	4	5	4	3	4,3
100.000 Elements	27	28	27	29	28	33	27	24	28	27	27,8
1.000.000 Elements	203	208	230	215	207	205	206	203	206	215	209,8
10.000.000 Elements	2028	2138	2092	2051	2149	2065	2083	2050	2204	2155	2101,5

Merge Sort Mixed (50.000)

Tamaño/tiempo	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Avg
10.000 Elements	4	4	4	3	4	4	4	3	4	4	3,8
100.000 Elements	39	36	37	37	38	34	33	41	39	37	37,1
1.000.000 Elements	193	340	182	199	178	197	173	197	190	185	203,4
10.000.000 Elements	1367	1373	1345	2098	3553	983	3901	1518	1436	1380	1895,4

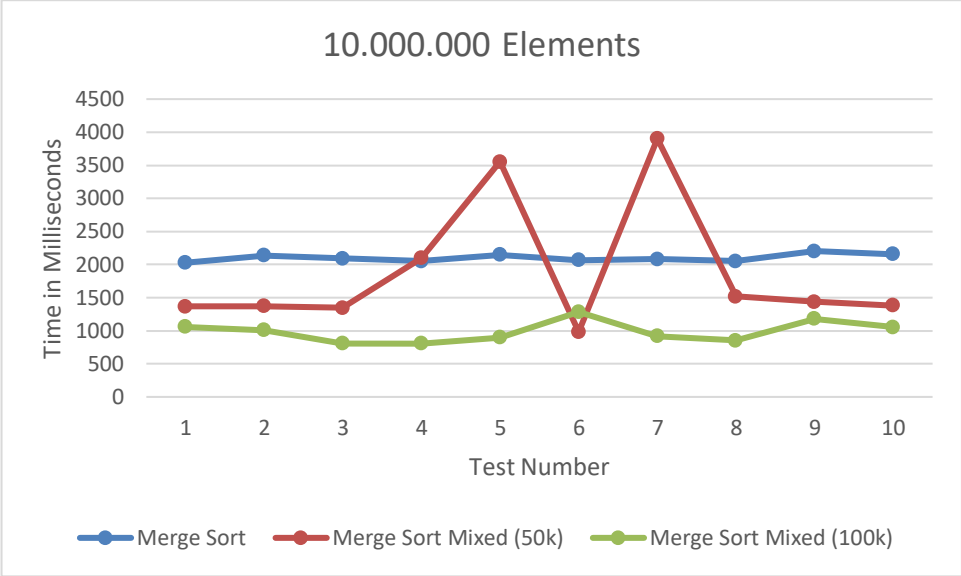
Merge Sort Mixed (100.000)

Tamaño/tiempo	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9	Test 10	Avg
10.000.000 Elements	1057	1006	806	807	899	1283	916	853	1182	1056	986,5

Como se puede ver en las tablas anteriores, la diferencia entre usar el merge mixto, con umbral de 50.000, o el merge secuencial es poca. Y esta diferencia puede llegar a ser más borrosa o peor para el merge mixto ya que sufre constantemente de picos en ejecución. Por lo tanto, si hay muchos picos seguidos, el promedio de ejecución puede llegar a ser peor que el merge secuencial.

Para probar en que tanto afecta el umbral, se hiso una última prueba en la que se usó un array con 10.000.000 de datos. Esta vez se usó un umbral de 100.000 y los resultados cambiaron drásticamente.

Para mostrar el contraste se graficaron los resultados.



Como se puede ver a en la gráfica, el merge mixto (con un umbral de 100k) es aproximadamente el doble de rápido que el merge secuencial. Además, este merge mixto (100k) no tiene picos tan extremos como el anterior(50k), por lo tanto, su valor es bastante constante y confiable. Esto demuestra como el umbral es bastante importante para que este algoritmo tenga una ventaja por sobre el merge secuencial. Finalmente se puede ver que para que esta ventaja ocurra, el tamaño del array tiene que ser considerablemente más grande, por lo que no es un algoritmo que sirva en todas las circunstancias.

Si bien a primera vista, dado los resultados de las pruebas anteriores, es posible pensar que la mayor causa del sobrecosto relacionado al utilizar threads son los join(), es importante seguir investigando de los sobrecostos relacionados con los threads. Al hacerlo aparece el concepto de “context switching”.

El mayor sobrecosto que existe al usar threads es el “context switching”. Cuando la CPU ejecuta un thread y luego pasa a ejecutar otro tiene que realizar varias operaciones. Estas incluyen “guardar los datos locales, el puntero de programa, etc. del hilo actual, y cargar los datos locales, el puntero de programa, etc. del siguiente hilo para ejecutar.” (Jenkov, Multithreading Costs, 2014) El cambio de contexto es una operación costosa, por lo tanto, hay que evitarla lo más posible al usar threads.

Una recomendación de Christian Hujer es no utilizar muchos más threads que el número de cores que tiene la CPU disponible. Esto se debe al sobre costo añadido por el thread-switching (context switching) y thread-synchronization. En este caso las pruebas de hicieron en un ordenardor de 4 cores.

El uso de threads también puede llegar a tener un coste en cuanto a la memoria. Esta memoria extra utilizada por los threads es para guardar su stack local. Si bien en este caso esto no afecta directamente al tiempo en que se tardan los algoritmos en ejecutarse, es bueno tenerlo presente a la hora de programar.

Si bien este sobre coste del uso de threads existe, no se cree que pueda justificar la penalización en el tiempo de ejecución que se visualiza en las gráficas. Esto se debe a que la diferencia del tiempo de ejecución no escala de forma proporcional. El incremento es considerablemente mayor. Por lo tanto, hay que seguir pensando en que cosas pueden estar pasando para que se vea este incremento en el tiempo.

Teniendo en cuenta el video “Scott Meyers - CPU Caches and Why You care” en el que se explicaron los conceptos de cache lines y false sharing se ha vuelto a analizar el comportamiento del algoritmo y el tiempo que toma en ejecutarse.

En ambos casos, el array se divide en dos y se pasa una copia a la siguiente iteración del mergerSort. Entonces, si en los dos casos se pasan copias, se podría pensar que la cache se debería comportar de la misma forma, que las copias se van añadiendo a la cache de forma secuencial y que todo queda relativamente bien aliñado en las cache lines.

En el caso de que sea secuencial, esto puede llegar a pasar, ya que al ejecutarse instrucción tras instrucción es posible que todo se aliñé de la mejor manera. Sin embargo, al tener en cuenta los threads, estos se ejecutan de forma simultanea y no se puede predecir el orden en que se van a guardar los datos en la cache, por lo tanto, el posible asumir que algunas o la mayoría de las copias se van a ir guardando en cache lines distintas. Esto quiere decir que se van a producir muchos más cache misses, los cuales añaden un overhead importante, que en la ejecución secuencial.

Por esta razón se puede concluir que la causa más importante en la diferencia del tiempo que se toman los algoritmos se debe a como se guarda y obtiene la información de las cache lines.

7 Conclusiones

Los ejercicios anteriores sirven para poder ver como la forma en la que se programa o diseña un programa puede tener un impacto positivo o negativo en su rendimiento. La diferencia entre estos suele ser sutil, y la forma más clara para ver su impacto es haciendo pruebas.

Después de analizar los resultados obtenidos se determinó que lo que más influenció en el rendimiento de los dos casos analizados fue el concepto de las “cache lines”.

En el primer caso, Shared Nothing vs Shared Memory, en casi todos los casos Shared Nothing fue más rápido en encontrar el número en el array. Sin embargo, en el último caso de un array con 1 millón de datos y un millón de threads, es más rápido Shared Memory. Esto es interesante porque se puede ver que el impacto de las cache lines también está ligado a la cantidad de datos que tiene que procesar el algoritmo.

En el segundo caso, el Merge Sort, se puede apreciar que el uso de threads no siempre mejora el rendimiento del programa. Los threads van generando sus copias en cache lines distintas y esto ocasiona que se generen más chache misses al cambiar de cache lines. En este ejemplo también se determinó que un algortimo mixto puede llegar a ser mejor que el secuencial cuando la cantidad de datos aumenta. Por lo tanto, se puede evidenciar nuevamente como combinar estrategias puede llegar a mejorar el rendimiento del programa.

En conclusión, la forma en la que funciona la cache tiene un impacto más grande del que se espera en la ejecución de los programas. No sólo por el hecho de que programarlo de una forma u otra puede hacer que un programa se ejecute más rápido que el otro, si no que a partir de cierto punto un programa que tenía un mejor rendimiento pasa a tener uno peor. Esto es interesante, ya que se vuelve a ver que el rendimiento también es variable según la escala. Puede ser que un algoritmo con pocos datos sea mejor, pero que a gran escala no lo sea. Por lo tanto, es indispensable hacer pruebas como las que se hicieron en los ejercicios para poder obtener una mejor idea de que está pasando y como se puede optimizar más.

8 Bibliografía

Apple. (2019). *Mac OS X Server v10.6: Understanding process limits*. Recuperado el October de 2019, de Support Apple: https://support.apple.com/en-gb/HT3854

Blaise Barney, L. L. (9 de October de 2019). *Introduction to Parallel Computing*. Recuperado el October de 2019, de Livermore National Laboratory: https://computing.llnl.gov/tutorials/parallel_comp/

Hujer, C. (20 de December de 2014). *does multi threading improve performance? scenario java [duplicate]*. Recuperado el October de 2019, de StackOverflow: https://stackoverflow.com/questions/27578208/does-multi-threading-improve-performance-scenario-java

Jenkov, J. (18 de December de 2014). *Java Memory Model*. Recuperado el October de 2019, de Jenkov.com: http://tutorials.jenkov.com/java-concurrency/java-memory-model.html

Jenkov, J. (29 de September de 2014). *Multithreading Costs*. Recuperado el October de 2019, de Jenkov.com: http://tutorials.jenkov.com/java-concurrency/costs.html

Narang, M. (2019). *Multithreading in Java*. Obtenido de GeeksforGeeks: https://www.geeksforgeeks.org/multithreading-in-java/

Oracle. (2010). *Multithreading Concepts*. Recuperado el October de 2019, de Oracle Multithreaded Programming Guide: https://docs.oracle.com/cd/E18752_01/html/816-5137/mtintro-25092.html

Oracle. (2019). *Defining and Starting a Thread*. Recuperado el October de 2019, de Oracle Java Documentation: https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html

Oracle. (2019). *Joins*. Recuperado el October de 2019, de Oracle Java Documentation: https://docs.oracle.com/javase/tutorial/essential/concurrency/join.html

Pankaj. (2013). *Java Multithreading Concurrency Interview Questions and Answers*. Recuperado el October de 2019, de JournalDev: https://www.journaldev.com/1162/java-multithreading-concurrency-interview-questions-answers

raman_257. (s.f.). *Difference between Thread.start() and Thread.run() in Java*. Recuperado el October de 2019, de GeeksforGeeks: https://www.geeksforgeeks.org/difference-between-thread-start-and-thread-run-in-java/