CONCURRENCIA EN EL MÉTODO DE ORDENAMIENTO MERGE SORT: ANÁLISIS COMPARATIVO

Mariano Alan Ramirez
Repositorio: https://github.com/xnanomarx/comparativa-mergesort-concurrencia
Video explicativo: https://youtu.be/AwXsRt-ANVM
E.mail: marianoramirez0714@gmail.com

RESUMEN

Este informe explora las ventajas y desafíos de la aplicación de la programación concurrente a un algoritmo secuencial. Para dicho objetivo, se realizará una comparativa entre las dos formas de gestión de procesos aplicadas en el método de ordenamiento Merge Sort. Se analizarán los escenarios de mejor y peor caso para su ejecución y se evaluarán los cambios notables en la curva de complejidad.

Se utilizará un algoritmo base secuencial tomado de Geek for Geeks (Geeks for Geeks, n.d.) y se comparará con otro que emplea el manejo de hilos, basado en una implementación descrita en un artículo de Medium (Codex, 2021). Este trabajo busca no solo comprender el impacto de la concurrencia en la complejidad algorítmica, sino también identificar los escenarios en los que resulta más conveniente recurrir a su uso.

1. INTRODUCCIÓN

El algoritmo Merge Sort es un método de ordenamiento popular, de los más eficaces en cuando a desempeño. Pero, ¿Cuál es el parámetro de eficacia? En este trabajo, nuestra medida será el tiempo. No necesariamente el tiempo reloj, o sí, pero no de la manera que uno imagina. El tiempo que nos importa para medir la eficacia de un proceso es el **tiempo máquina** o tiempo de la cpu.

Esta medida es la que cuantifica la cantidad de tiempo durante el cual la unidad central de procesamiento (CPU) de una computadora está dedicada a la ejecución de un programa o proceso específico (Manuales Tech, 2023).

En otras palabras, nos dedicaremos a observar cuánto tiempo mantendremos ocupada a nuestra computadora con las instrucciones que le damos y la forma de llevarlas a cabo.

Así, tenemos dos caminos: la forma **secuencial** y la **concurrente.**

Presentando a la primera, es la ejecución de instrucciones paso a paso, de manera lineal y respetando los turnos independientemente de prioridades o asignaciones. El código irá de un punto A al punto B en una dirección.
En cambio, de la manera concurrente, pasamos a paralelizar el proceso en distintos **hilos de ejecución**. Nuestro programa puede ejecutarse de manera lineal y llegar a un punto en el cual se disparen estos hilos, que se desprenden del **main** o rama principal, para poder realizar procesos de manera simultánea hasta el próximo punto de encuentro entre los mismos.

Volviendo al Merge Sort, su estructura clásica es de manera secuencial, con una complejidad O(n log n) (Geeks for Geeks, n.d.). Se basa en el criterio de "dividir y conquistar". El ordenamiento consiste en dividir la entrada en dos mitades, clasificándolas recursivamente hasta terminar fusionándolas en su forma final ordenada.

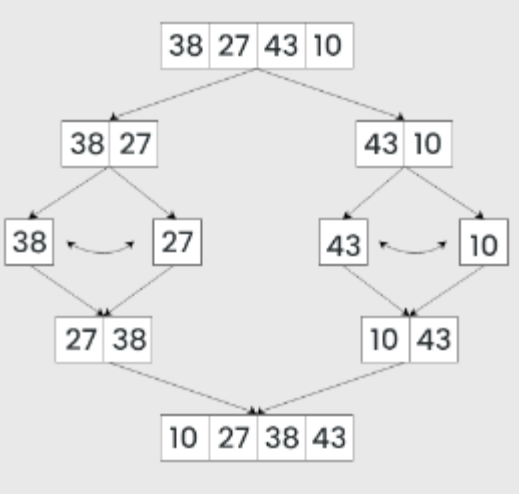

Figura 1. Visualización del algoritmo Merge Sort. Adaptado de "Merge Sort," por Geeks for Geeks, https://www.geeksforgeeks.org/merge-sort/.

Entre sus ventajas encontramos principalmente: su estabilidad, ya que mantiene el orden relativo de los elementos con valores iguales; su adecuación a grandes conjuntos de datos, ya que divide el problema en partes más pequeñas y las resuelve de manera independiente y su capacidad de paralelización, permitiendo aprovechar arquitecturas multinúcleo o multiprocesador para mejorar la velocidad en conjuntos de datos grandes.

2. IMPLEMENTACIÓN CONCURRENTE

En la paralelización del Merge Sort, apuntaremos a realizar de manera concurrente la división recursiva de las entradas, es decir, los subarreglos.

Para este objetivo emplearemos dos métodos: **invokeAll()**, que ejecuta de manera concurrente múltiples tareas y espera a que todas finalicen antes de continuar, asegurando que los subprocesos trabajen en paralelo de manera eficiente; y **ForkJoinPool()**, que proporciona un entorno especializado para la ejecución de tareas paralelas, gestionando los recursos mediante técnicas como el trabajo robado para optimizar la concurrencia.

En el método compute(), dividimos el arreglo principal (array) en dos subarreglos: left y right. Esta operación no es concurrente, sino una preparación para el procesamiento paralelo. Luego creamos dos nuevas instancias de ParallelMergeSort, una para cada subarreglo (leftTask y rightTask).

La concurrencia se ejecuta cuando llamamos a invokeAll(leftTask, rightTask), las dos tareas (leftTask y rightTask) se ejecutan de manera concurrente en diferentes subprocesos del ForkJoinPool. Esto permite que las divisiones y las llamadas recursivas de clasificación en los subarreglos se realicen en paralelo, maximizando el uso de los núcleos del procesador.

Después de que ambas tareas concurrentes (leftTask y rightTask) hayan terminado, los subarreglos ordenados (left y right) se fusionan en el arreglo original (array). Esta operación de fusión no se realiza de manera concurrente, ya que el método merge() es secuencial.

```
int mid = n / 2;
T[] left = Arrays.copyOfRange(array, 0, mid);
T[] right = Arrays.copyOfRange(array, mid, n);

ParallelMergeSort<T> leftTask = new ParallelMergeSort<>(left, comp);
ParallelMergeSort<T> rightTask = new ParallelMergeSort<>(right, comp);

invokeAll(leftTask, rightTask);

merge(left, right, array, comp);
```

Figura 2. Visualización del algoritmo ParallelMergeSort. Adaptado de CodeX (2021). *Merge Sort: Divide and Conquer for Large Datasets*. Recuperado de https://medium.com/codex/merge-sort-divide-and-conquer-for-large-datasets-a33e2a6e58e2.

3. COMPARATIVA Y DESEMPEÑO

Generamos comparativas en base a 3 casos del Merge Sort (Geeks for Geeks, n.d.):

Mejor caso: O(n log n), cuando el arreglo ya está ordenado o casi ordenado.

Caso promedio: O(n log n), cuando el arreglo está desordenado de manera aleatoria.

Peor caso: O(n log n), cuando el arreglo está invertido.

Además, en la comparativa utilizamos arreglos de distintos tamaños (1000, 100000, 1000000 y 10000000) para identificar el rendimiento tanto en pequeños como grandes volúmenes de datos. Para no obtener resultados aislados ni intentos "sin calentamiento" del CPU, realizamos 100 intentos de cada caso y obtenemos el promedio de tiempo de ejecución en milisegundos.

Las pruebas fueron realizados en una PC con 16gb de memoria ram y un procesados Ryzen 7 5700g de 8 núcleos y 16 hilos.

Los resultados fueron:

Algoritmo	Tamaño del arreglo	Peor caso (En ms)	Caso promedio (En ms)	Mejor caso (En ms)
Secuencial	1.000	0	0	0
Concurrente	1.000	1	1	1
Secuencial	100.000	3	8	3
Concurrente	100.000	4	5	3
Secuencial	1.000.000	38	96	49
Concurrente	1.000.000	32	106	44
Secuencial	10.000.000	444	1004	486
Concurrente	10.000.000	447	1409	461

Figura 3. Resultados de la comparativa.

4. CONCLUSIÓN

Podemos apreciar que en arreglos de tamaño pequeño (1000), sin importar el caso, no hay diferencia significativa e incluso gana el secuencial por la mínima. Aunque incrementemos el tamaño a mediano (100,000), las diferencias siguen siendo mínimas. Esto se debe a que es más costoso en cuanto a tiempos de ejecución soltar y gestionar la sobrecarga de hilos de lo que se tarde en ordenar el arreglo en sí.

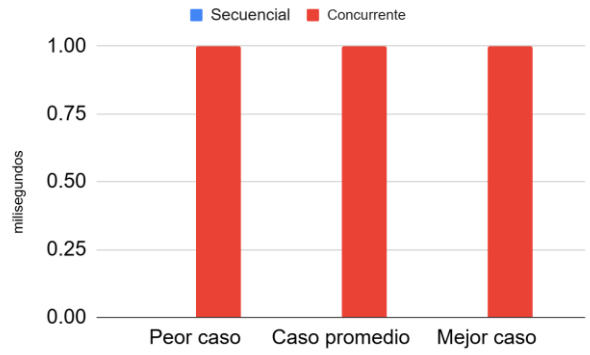

Figura 4. Arreglo de tamaño pequeño.

Aunque en el caso promedio y en el mejor caso nuestro algoritmo concurrente empieza a mostrarse competitivo, el problema de la gestión de hilos sigue pesando más en el peor caso.

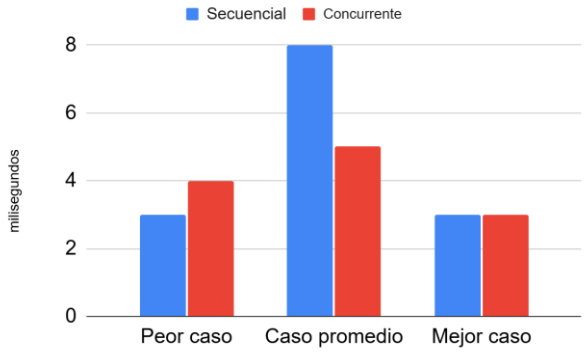

Figura 5. Arreglo de tamaño mediano.

Cuando pasamos a los arreglos grandes (1000000), el concurrente logra ser más eficiente en el peor caso por la ventaja de paralelizar la complejidad del ordenamiento. Incluso gana en el mejor caso, aunque se queda atrás en el caso promedio.

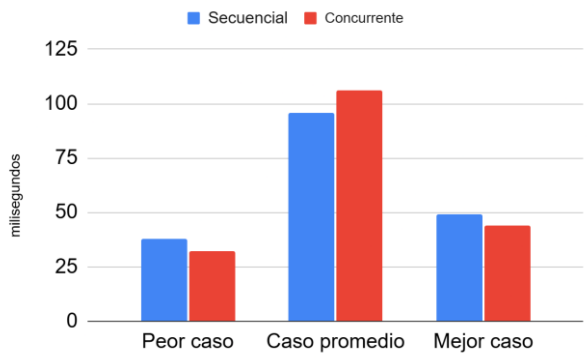

Figura 6. Arreglo de tamaño grande.

Finalizamos con arreglos muy grandes (10000000) y si bien logramos un resultado muy aproximado en el peor caso, es notoria la diferencia en tiempo para el caso promedio. La complejidad en el manejo de hilos para la concurrencia resulta muy alta como para poder optimizar el procesamiento. Sí podemos notar que, en el mejor caso, el algoritmo secuencial perdió bastante ventaja.

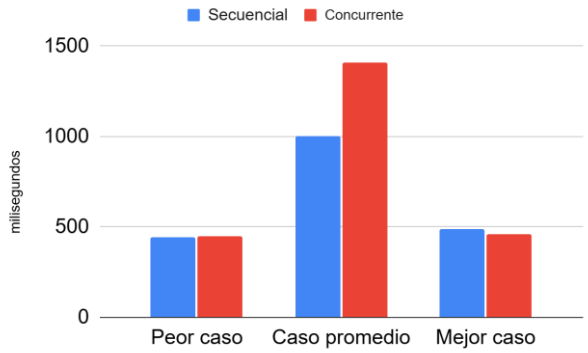

Figura 7. Arreglo de tamaño muy grande.

El análisis que podemos obtener de estos resultados es que el uso de un algoritmo concurrente es más beneficioso cuando el tamaño del conjunto de datos justifica el costo de la concurrencia.

Por ejemplo, para arreglos pequeños, la sobrecarga asociada con la creación, gestión y sincronización de hilos en el algoritmo supera cualquier ventaja potencial de la concurrencia, haciendo que el algoritmo secuencial sea más eficiente.

A medida que crece el tamaño del arreglo, los beneficios de dividir el trabajo en tareas concurrentes comienzan a superar la sobrecarga inicial: El manejo eficiente de la concurrencia puede aprovechar la capacidad de múltiples núcleos de CPU para mejorar los tiempos de procesamiento, especialmente en escenarios donde las divisiones del problema son equilibradas y las operaciones son suficientemente complejas para justificar la paralelización.

La conclusión final de este proyecto es que el uso de algoritmos concurrentes debe ser cuidadosamente evaluado dependiendo del tamaño de los datos y la arquitectura del sistema. Para aplicaciones con conjuntos de datos pequeños o medianos, un algoritmo secuencial puede ser suficiente Incluso la concurrencia puede llegar a ser contraproducente por la cantidad de recursos que debemos gestionar. Sin embargo, en aplicaciones que procesan grandes volúmenes de datos, especialmente en sistemas modernos de múltiples núcleos, la concurrencia puede ofrecer mejoras significativas en el rendimiento.

REFERENCIAS

Codex. (2021). *Merge Sort: Divide and conquer for large datasets*. Medium. https://medium.com/codex/merge-sort-divide-and-conquer-for-large-datasets-a33e2a6e58e2

Geeks for Geeks. (n.d.). *Merge Sort*. https://www.geeksforgeeks.org/merge-sort/

Manuales Tech. (2023). ¿Qué es Tiempo de CPU o CPU Time? https://manualestech.com/definicion/cpu-o-cpu-time/