IMPACTO DEL PARALELISMO EN MERGESORT: COMPARACIÓN ENTRE VERSION SECUENCIAL Y CONCURRENTE EN JAVA

Datos del autor: Luca De La Via
Repositorio con el trabajo: https://github.com/lucadelavia/MergeSort-Concurrente-vs-Secuencial
Video explicativo (<10min): https://youtu.be/fm9HVbJc2pE
E.mail: lucadelavia@gmail.com

RESUMEN (ABSTRACT)

Este trabajo comparó las implementaciones secuencial y concurrente del algoritmo de ordenamiento MergeSort. El objetivo fue analizar el impacto del paralelismo sobre el rendimiento, midiendo los tiempos de ejecución para distintos tamaños de entrada. La variante secuencial se programó íntegramente desde cero, mientras que la versión concurrente se desarrolló tomando como referencia el artículo técnico publicado en HackerNoon (2020), que propone la partición paralela de subarreglos mediante la clase ForkJoinPool. Ambas versiones del algoritmo se implementaron en Java, se ejecutaron bajo condiciones controladas y se evaluaron con métricas cuantitativas de tiempo. Los resultados muestran que, a partir de ciertos volúmenes de datos, la implementación concurrente reduce de forma significativa el tiempo de procesamiento. Este estudio ayuda a delimitar cuándo y cómo conviene aplicar concurrencia en algoritmos clásicos.

Keywords: concurrencia, ForkJoinPool, MergeSort, rendimiento, tiempo de ejecución, Java.

1. INTRODUCCIÓN

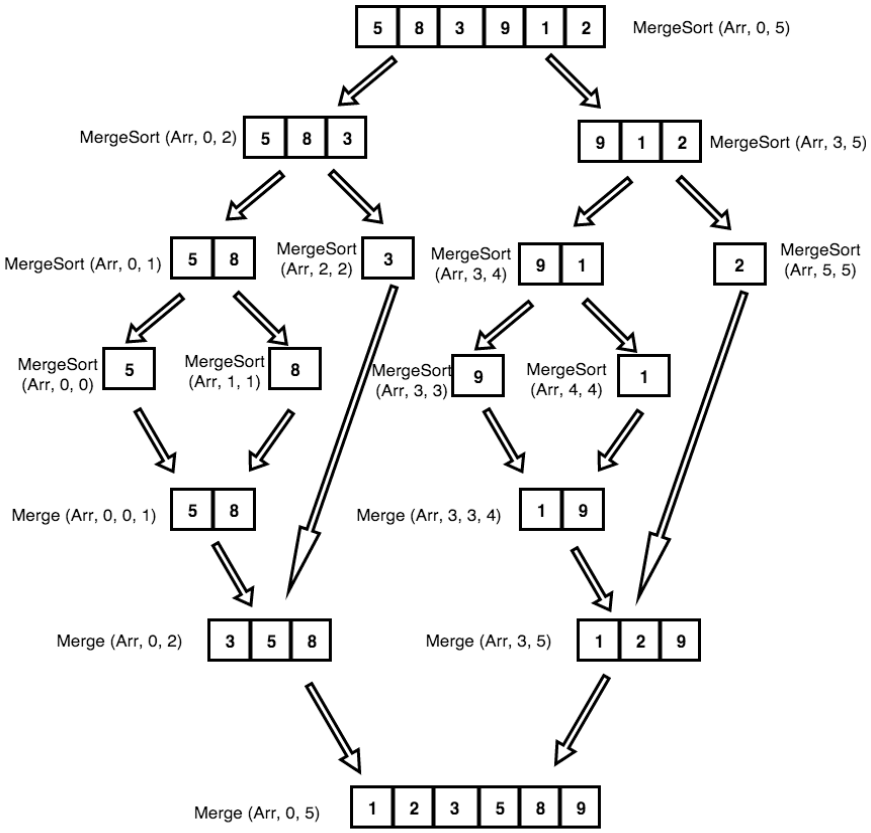
MergeSort es un algoritmo de ordenamiento comparativo que ejemplifica la estrategia *divide y vencerás*. El proceso comienza al fraccionar el arreglo original en dos mitades de manera recursiva; cada subarreglo se trata como un problema independiente y se resuelve hasta quedar reducido a un solo elemento, momento en el que se considera ordenado. Al deshacerse la recursión, las mitades resultantes se combinan en orden ascendente, reconstruyendo gradualmente la lista completa. De este modo, las tres fases división, resolución de subproblemas triviales y fusión se encadenan para producir la secuencia ordenada final. Por diseño, el algoritmo ejecuta log2n niveles de división y realiza, en cada nivel, un recorrido lineal sobre los datos; de ahí su complejidad temporal estable de O(n log n) tanto en el mejor como en el peor caso, sin depender de la disposición inicial de los datos.

La versión secuencial, programada completamente en Java, se organiza alrededor del método central mergeSort.
Si el arreglo contiene cero o un elemento, la función finaliza porque ya está ordenado. De lo contrario, calcula el punto medio y, con Arrays.copyOfRange, divide el arreglo en dos partes. Cada subarreglo se ordena mediante una llamada recursiva a sí mismo y, una vez que ambas llamadas retornan, se invoca el método MergeUtils.merge el cual fusiona los resultados, recorre las dos mitades de forma paralela, selecciona el menor de los elementos comparados y lo coloca en la posición correspondiente del arreglo original, completando la unión final.

El código fuente, junto con las instrucciones para compilarlo y reproducir las pruebas, está disponible en el siguiente repositorio: https://github.com/lucadelavia/MergeSort-Concurrente-vs-Secuencial(De La Via, 2025).

Figura 1

Diagrama del funcionamiento recursivo de MergeSort



Nota. El diagrama ilustra cómo el arreglo original se divide recursivamente en mitades hasta alcanzar subarreglos con un solo elemento, que luego se fusionan en orden ascendente al regresar de la recursión. Tomado de *FreeCodeCamp (2023)*, https://www.freecodecamp.org/news/sorting-algorithms-explained-with-examples-in-python-java-and-c/

2. IMPLEMENTACIÓN CONCURRENTE

La versión concurrente del algoritmo MergeSort fue desarrollada utilizando la clase ForkJoinPool del paquete java.util.concurrent, que permite ejecutar tareas recursivas en múltiples hilos de forma automática. Esta estructura se basa en el patrón fork/join, que consiste en dividir un problema en subproblemas más pequeños que pueden resolverse en paralelo (fork) y luego combinar sus resultados cuando ambos han finalizado (join).

A diferencia de la implementación secuencial, esta versión incorpora un umbral definido por la constante UMBRAL = 100_000. Si el tamaño del subarreglo es menor o igual a ese valor, no se lanza ninguna tarea concurrente: se utiliza directamente la implementación secuencial, lo que evita la sobrecarga de crear hilos innecesarios.

Este criterio actúa como un filtro para decidir si el costo del paralelismo vale la pena en función del tamaño de entrada.

Cada fragmento del arreglo se encapsula en una instancia de la clase TareaMergeSort, que extiende

RecursiveTask<int[]>. Esta clase representa una unidad de trabajo que puede ser ejecutada por un hilo del pool. Su método compute() define la lógica de ordenamiento. En primer lugar, verifica si el arreglo contiene uno o ningún elemento; en ese caso, lo retorna directamente. Si el arreglo supera el umbral, se divide en dos mitades usando Arrays.copyOfRange. Luego, se crean dos tareas recursivas: una para la mitad izquierda y otra para la derecha.

La mitad izquierda se ejecuta de forma asincrónica mediante el método fork(), lo que permite que se resuelva en un hilo del pool. Mientras tanto, la mitad derecha se procesa en el hilo actual usando compute(). Una vez finalizada, se sincroniza con join() para obtener el resultado de la primera tarea. Finalmente, ambas mitades ordenadas se combinan utilizando MergeUtils.merge, que realiza la fusión de forma secuencial.

De esta manera, solo la fase de ordenamiento recursivo se ejecuta en paralelo. La verificación del caso base y la fusión de los resultados se mantienen secuenciales. Esta decisión se basa en una cuestión práctica: paralelizar la fusión no generaría beneficios significativos y complicaría innecesariamente la implementación. El diseño resultante conserva la complejidad temporal O(n log n) y el uso lineal de memoria, al mismo tiempo que mejora el rendimiento cuando el volumen de datos justifica el uso de múltiples hilos.

El código fuente, junto con las instrucciones para compilarlo y reproducir las pruebas, está disponible en el siguiente repositorio: https://github.com/USUARIO/PROYECTO-MergeSort-Concurrente (De La Via, 2025).

Figura 2

Método compute() de la clase TareaMergeSort en la implementación concurrente

```
@Override
protected int[] compute() {
    if (array.length <= UMBRAL) { // ordena secuencialmente si el arreglo es chico
        int[] copia = Arrays.copyOf(array, array.length); // evita modificar el original
        MergeSortSecuencial.mergeSort(copia); // usa la version secuencial
        return copia;
    }

    int medio = array.length / 2; // calcula el punto medio
    int[] izquierda = Arrays.copyOfRange(array, from: 0, medio);
    int[] derecha = Arrays.copyOfRange(array, medio, array.length);

    TareaMergeSort tareaIzquierda = new TareaMergeSort(izquierda);
    TareaMergeSort tareaDerecha = new TareaMergeSort(derecha);

    tareaIzquierda.fork(); // ejecuta izquierda en paralelo
    int[] ordenadoDerecha = tareaDerecha.compute(); // derecha en el hilo actual
    int[] ordenadoIzquierda = tareaIzquierda.join(); // espera resultado izquierdo

    return MergeUtils.merge(ordenadoIzquierda, ordenadoDerecha); // fusiona y retorna
}
```

Nota. Este fragmento pertenece a la implementación concurrente desarrollada para este trabajo. Muestra la lógica central de la recursión paralela mediante Fork/Join. Si el subarreglo es menor al umbral, se ordena con la versión secuencial; en caso contrario, se divide y se resuelve en paralelo usando fork(), compute() y join().

3. COMPARATIVA Y DESEMPEÑO

Con el objetivo de evaluar el rendimiento de ambas versiones del algoritmo MergeSort, se llevaron a cabo pruebas controladas en distintos escenarios. Para reducir el impacto de variaciones puntuales en el entorno de ejecución, cada configuración fue ejecutada diez veces consecutivas y se calculó el promedio del tiempo de ejecución en milisegundos (ms).

Las pruebas se realizaron sobre arreglos generados aleatoriamente de distintos tamaños: 1 000, 10 000, 100 000, 1 000 000, 10 000 000 y 100.000.000 elementos.

En todos los casos, se aplicó tanto la versión secuencial como la concurrente. Se utilizó un mismo conjunto de datos por tamaño para asegurar la comparabilidad entre ambas implementaciones.

Las especificaciones de la computadora utilizada para la ejecución fueron las siguientes: procesador AMD Ryzen 5 5600 con 6 núcleos y 12 hilos, 16 GB de memoria RAM DDR4, sistema operativo Windows 11 de 64 bits y JDK 21.

Tabla 1 Comparativas

Promedio de tiempo de ejecución (en milisegundos) sobre 10 repeticiones.

Tamaño del arreglo	Secuencial (ms)	Concurrente (ms)
1 000	≈ 2	≈ 3
10 000	≈ 4	≈ 4
100 000	≈ 17	≈ 18
1 000 000	≈ 116	≈ 42
10 000 000	≈ 1039	≈ 319
100 000 000	≈ 10 132	≈ 4 147

Nota. Cada valor representa el promedio aproximado de diez ejecuciones sobre arreglos aleatorios. El símbolo ≈ indica redondeo.

4. CONCLUSIÓN

Los resultados obtenidos en las pruebas confirmaron que la versión concurrente de MergeSort solo ofrece mejoras de rendimiento significativas cuando se procesan volúmenes grandes de datos. Para entradas de hasta 100 000 elementos, la variante secuencial registró tiempos de ejecución iguales o incluso mejores, debido al costo adicional asociado a la creación y sincronización de tareas paralelas, sobrecosto que no se justifica cuando el problema es pequeño.

A partir de un millón de elementos la tendencia se invirtió: la versión concurrente superó de manera sostenida a la secuencial y esa ventaja se amplió con conjuntos de 10 y 100 millones. En el escenario más exigente (100 millones), la implementación concurrente resultó más de dos veces más rápida.

El éxito del paralelismo se vio reforzado por la incorporación de un umbral que limitó la generación de hilos en tamaños reducidos, manteniendo un equilibrio adecuado entre velocidad y eficiencia. Además, la fusión final se mantuvo secuencial, lo que simplificó el diseño y

evitó sincronizaciones innecesarias en una fase que no constituye cuello de botella.

En síntesis, MergeSort concurrente supera a su contraparte secuencial solo a partir de un tamaño umbral de entrada, lo que confirma que la concurrencia no es la opción óptima por defecto, sino una herramienta eficaz cuando se aplica en el contexto adecuado. Este trabajo permitió poner en práctica conceptos de programación paralela y diseño algorítmico, y proporcionó evidencia concreta sobre cuándo y cómo conviene emplear técnicas concurrentes en algoritmos clásicos.

REFERENCIAS

Codecademy. (n.d.). *Time complexity of Merge Sort*. https://www.codecademy.com/article/time-complexity-of-merge-sort

Makeev, A. (2020, enero 3). *Parallel Merge Sort with ForkJoin Framework*. HackerNoon. https://hackernoon.com/parallel-merge-sort-with-forkjoin-framework

Makeev, A. (s.f.). *mergesort-with-forkjoinpool*. GitHub. https://github.com/alexandermakeev/mergesort-with-forkjoinpool?ref=hackernoon.com

Mathrani, S. (2023, agosto 29). *Sorting algorithms explained with examples in Python, Java, and C*. freeCodeCamp. https://www.freecodecamp.org/news/sorting-algorithms-explained-with-examples-in-python-java-and-c/