MERGESORT SECUENCIAL Y CONCURRENTE

Datos del autor: Santiago Cano
Repositorio con el trabajo: TP Programación Concurrente Santiago Cano - MergeSort (GitHub)
Video explicativo (<10min): Trabajo Práctico Programación Concurrente (YouTube)
Email: canosantiago404@gmail.com

RESUMEN

Este informe estudia el algoritmo MergeSort, uno de los ordenamientos de datos más fundamentales y eficientes. El objetivo es entender su funcionamiento tanto en su versión tradicional, la secuencial, como en su implementación concurrente. Elegí este algoritmo por su diseño basado en el "Divide y vencerás" y lo considero un buen candidato para la paralelización, permitiendo ver como la concurrencia puede optimizar el rendimiento. Se busca analizar las diferencias entre ambas aproximaciones, destacando las ventajas que trae la concurrencia y como se pueden traducir en mejoras en el tiempo de ejecución. Ninguna de las dos aproximaciones del algoritmo son mías, pero la implementación concurrente fue modificada un poco, dejo el link en las referencias. Finalmente se presentan resultados de rendimiento comparativos, discutiendo si la aplicación de la concurrencia justifica su implementación, especialmente en el manejo de grandes volúmenes de datos.

Keywords: MergeSort, Concurrente, Secuencial, Algoritmo, Paralelo, Java, Divide y vencerás, Fork, Join.

1. INTRODUCCIÓN

¿Qué es el MergeSort?

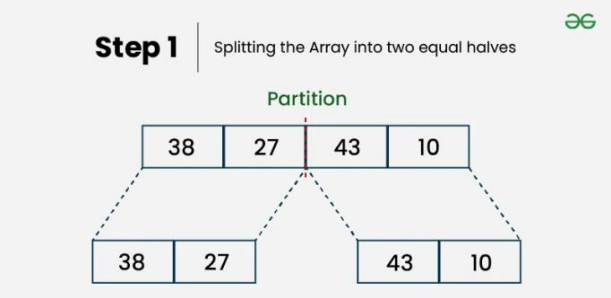
MergeSort es un algoritmo de ordenamiento que se destaca por su eficiencia y su aplicación generalizada. Su diseño se basa en una estrategia conocida como "Divide y vencerás". Esto implica abordar un problema complejo y partirlo en subproblemas más pequeños y manejables. En el contexto del MergeSort, esta estrategia se puede pensar en tres etapas:

- **Dividir**: La lista o arreglo se divide recursivamente en sublistas más pequeñas hasta que cada sublista contiene un solo elemento. Una sublista de un solo elemento se considera ordenada.
- **Vencer:** Se fusionan repetidamente las sublistas ordenadas para producir nuevas sublistas ordenadas de mayor tamaño, hasta que todas las sublistas se hayan fusionado en una única lista que constituye el arreglo ordenado completo.

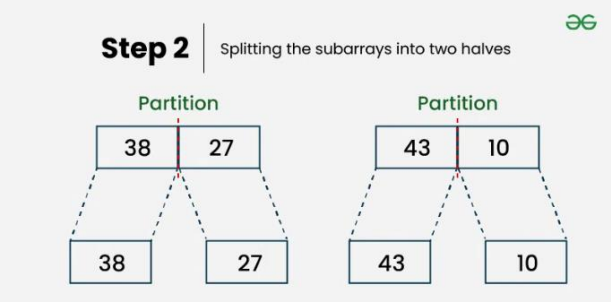
Ejemplo de MergeSort

Considerando un arr[] = {38, 27, 43, 10}

1. Se divide el array en dos partes iguales.

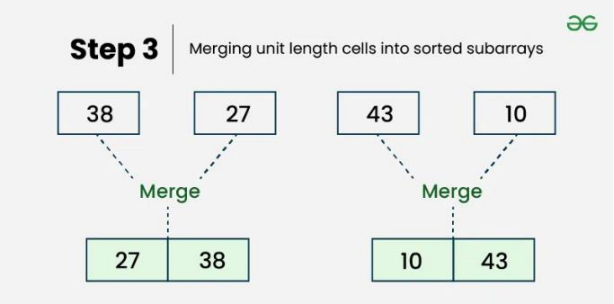


2. Se vuelven a dividir los dos subarrays en partes iguales.

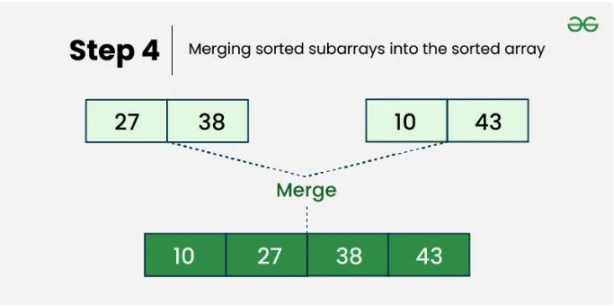


3. Una vez más, se vuelven a dividir los subarrays, esta vez quedando un solo elemento en cada uno, ya están ordenados.

4. Se fusionan los subarrays de un solo elemento en subarrays ordenados.



5. Se siguen fusionando los subarrays ordenados hasta llegar a un array completo totalmente ordenado.



Complejidad de MergeSort

Para hablar de la complejidad de los algoritmos primero hay que saber que es el Big O.

Big O define el tiempo de ejecución necesario para ejecutar un algoritmo identificando como cambiara el rendimiento del mismo a medida que crece el tamaño de entrada.

Mide la eficiencia y el rendimiento del algoritmo usando la complejidad temporal y espacial.

- La complejidad temporal de un algoritmo especifica cuanto tiempo llevara ejecutarlo en función del tamaño de la entrada.
- La complejidad espacial especifica la cantidad total de espacio o memoria necesaria para ejecutar el algoritmo también en función del tamaño de entrada.

En cuanto al MergeSort estos son sus casos:

- **Mejor caso: O(n log n):** Ocurren cuando el arreglo ya está ordenado o casi ordenado y no mejora más allá porque MergeSort siempre divide, no analiza si ya está ordenado o no.
- **Peor caso: O(n log n):** Ocurre cuando el arreglo está completamente inverso o tiene muchos elementos iguales. Pero sigue siendo n log n, no empeora más porque MergeSort siempre divide el arreglo exactamente por la mitad.
- **Caso promedio: O(n log n):** Ocurre con cualquier tipo de arreglo aleatorio. Sigue siendo n log n porque sin importar el contenido, MergeSort siempre hace el mismo proceso.

En conclusión, MergeSort es un algoritmo eficiente y predecible, ya que mantiene una complejidad temporal de O(n log n) en todos los casos: mejor, peor y promedio.

A diferencia de otros algoritmos como QuickSort, MergeSort no depende de la distribución de los elementos, lo que lo hace ideal para casos donde se requiere un comportamiento estable. Dado a esto, el lenguaje Java utiliza una versión optimizada de MergeSort llamada TimSort para el ordenamiento de arrays de objectos en su librería utils.

¿Entonces porque no se usa MergeSort siempre? Por su complejidad espacial O(n), que puede llegar a ser una desventaja en entornos con recursos limitados.

2. IMPLEMENTACIÓN CONCURRENTE

¿Por qué hacerlo concurrente? Ventajas y desventajas

La concurrencia representa la capacidad de un sistema para ejecutar múltiples partes de un programa de forma simultánea, aprovechando al máximo los procesadores con múltiples núcleos. Se puede pensar como tener varios cerebros trabajando en paralelo sobre diferentes partes de un mismo problema.

MergeSort es un algoritmo ideal para la paralelización por su estrategia de “Divide y vencerás”. El paso de dividir genera subarrays que son independientes entre sí. Esto significa que cada subarray puede ser ordenado de forma simultánea,

esto significaría una mejora de rendimiento y escalabilidad para sistemas con múltiples núcleos.

Pero hay un problema llamado "overhead", la creación, gestión y sincronización de hilos es una operación costosa. Implica un consumo de tiempo de CPU y memoria. Para conjuntos de datos pequeños este "overhead" puede ser tan significativo que la versión concurrente termine siendo más lenta que la secuencial. También es imposible determinar cuántos hilos se van a utilizar porque la recursividad se aplica en cantidades aleatorias.

Para solucionar este problema se utiliza la clase ForkJoin que nos permite generar máxima cantidad de hilos que soporta el procesador y esos hilos se van a asignar a las tareas que surgan de la recursividad.

ForkJoin

El funcionamiento del modelo ForkJoin se puede describir de esta manera:

- **Fork:** En puntos específicos del programa, la ejecución se "ramifica" para crear nuevas tareas paralelas. En el MergeSort, esto puede ocurrir cuando el arreglo se divide en dos mitades y se desea ordenar ambas en paralelo. Cada operación de Fork lanza una subtarea que puede ejecutarse en un hilo diferente.
- **Join:** Después estas ramas se unen en un punto de sincronización y la ejecución secuencial se reanuda. La operación Join asegura que todas las subtareas hayan completado su trabajo antes de que el hilo principal pueda continuar con la siguiente fase.

MergeSort concurrente

En el artículo [Parallel Merge Sort in Java], se presenta una implementación del algoritmo MergeSort usando concurrencia mediante la librería ForkJoin del paquete java.util.concurrent. Lo modifiqué un poco y añadí comentarios para explicarlo mejor. El repositorio está en GitHub.

En esta implementación, se extiende la clase RecursiveAction y se sobrescribe el método compute(), donde se realiza la lógica de división del arreglo y la creación de subtareas. Cada subtarea se ejecuta en paralelo utilizando el método invokeAll() y luego se combinan los resultados mediante fusión.

Vistazo del Main()

1. Se genera un array aleatorio

```
// Genera un arreglo aleatorio de 10 millones de elementos
int[] aux = generarArrayAleatorio(10_000_000, 1, 9000);

// Crea copias del arreglo para cada método de ordenamiento
int[] A = Arrays.copyOf(aux, aux.length); // Para MergeSort secuencial
int[] B = Arrays.copyOf(aux, aux.length); // Para MergeSort concurrente

System.out.println("El arreglo tiene " + aux.length + " elementos");
```

2. Se mide el tiempo del MergeSort secuencial

```
// Medición del tiempo de ejecución para MergeSort secuencial
long inicioSecuencial = System.currentTimeMillis();
MergeSortSecuencial.sort(A);
long finSecuencial = System.currentTimeMillis();
System.out.println("Tiempo MergeSort Secuencial: " + (finSecuencial - inicioSecuencial) + " ms");
```

3. Se mide el tiempo del MergeSort concurrente

```
// Medición del tiempo de ejecución para MergeSort concurrente
try (
ForkJoinPool pool = new ForkJoinPool()) {
    int[] temp = new int[B.length]; // Arreglo auxiliar para la fusión
    MergeSortConcurrente task = new MergeSortConcurrente(B, temp, 0, B.length - 1);
    long inicioConcurrente = System.currentTimeMillis();
    pool.invoke(task); // Ejecuta la tarea en el pool de hilos
    long finConcurrente = System.currentTimeMillis();
    System.out.println("Tiempo MergeSort Concurrente: " + (finConcurrente - inicioConcurrente) + " ms");
}
```

Para iniciar la ejecución, se crea un ForkJoinPool, por defecto con tantos hilos como núcleos disponibles. Es crucial definir un umbral, un tamaño mínimo por debajo del cual la división se detiene y se aplica un algoritmo de ordenamiento secuencial.

Vistazo de la clase concurrente. compute()

```
// Método que define la lógica de la tarea a ejecutar en paralelo
@Override
protected void compute() {
    if (right - left < THRESHOLD) {
        // Si el segmento es pequeño, utiliza ordenamiento secuencial
        mergeSortSequential(array, temp, left, right);
        return;
    }

    int mid = left + (right - left) / 2; // Calcula el punto medio

    // Crea tareas para ordenar ambas mitades en paralelo
    MergeSortConcurrente leftTask = new MergeSortConcurrente(array, temp, left, mid);
    MergeSortConcurrente rightTask = new MergeSortConcurrente(array, temp, mid + 1, right);

    // Ejecuta ambas tareas en paralelo
    invokeAll(leftTask, rightTask);

    // Fusiona las mitades ordenadas
    merge(array, temp, left, mid, right);
}
```

Se verifica si el tamaño del subarray es menor que el umbral THRESHOLD. Si es así, se realiza el ordenamiento de forma secuencial. De lo contrario, se divide el arreglo en dos mitades y se crean dos tareas (leftTask y rightTask) que se ejecutan en paralelo mediante el invokeAll(). Una vez finalizadas, se fusionan en el merge().

3. COMPARATIVA Y DESEMPEÑO

Para demostrar el desempeño de los algoritmos se generaron dos arrays de números aleatorios y se les aplico la misma cantidad de números cada vez mayor para ver su rendimiento.

Se utilizo como medida de comparación el tiempo en milisegundos, usando la función de Java System.currentTimeMillis().

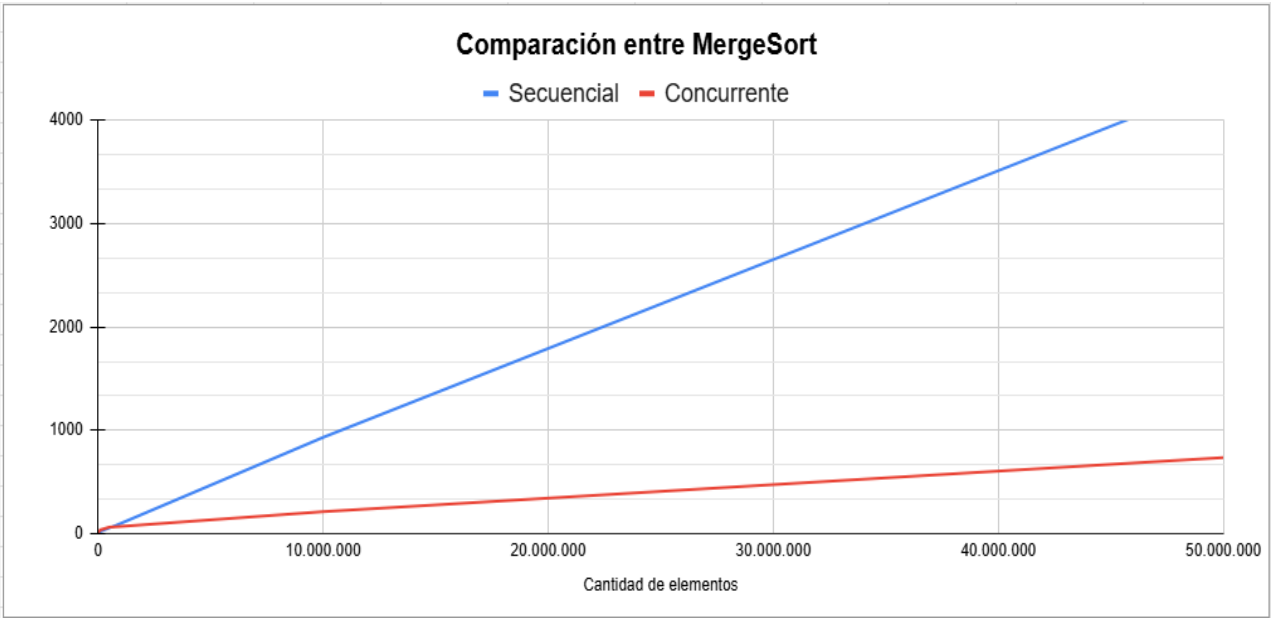
Se utilizo Google Sheets para hacer la tabla y graficar la comparación.

El verde indica el algoritmo más rápido, el rojo el más lento y el amarillo un posible empate. En el eje X se representan la cantidad de elementos y en el eje Y el tiempo en milisegundos.

Las especificaciones del equipo de prueba son: Intel I5 11400 6 Núcleos 12 procesadores lógicos (12 hilos en paralelo), 16GB RAM, Windows 11, Java 21.

Comparación entre secuencial y concurrente

Cantidad de elementos	1000	10.000	100.000	500.000	1.000.000	10.000.000	50.000.000	100.000.000
MergeSort secuencial (ms)	1	2	22	54	98	931	4372	8949
MergeSort concurrente (ms)	1	6	36	62	70	214	736	1496



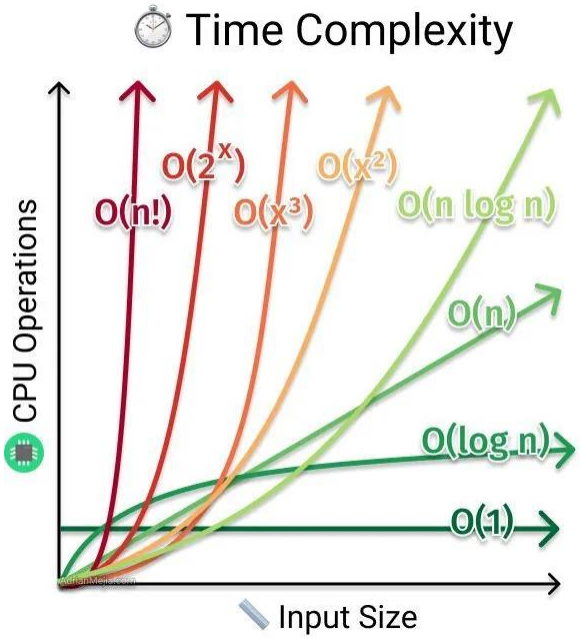
4. CONCLUSIÓN

Luego de realizar la comparación entre la implementación secuencial y concurrente de MergeSort y analizar sus resultados. Mis conclusiones son:

- MergeSort es uno de los algoritmos más estables y eficientes, caracterizado por su complejidad temporal consistente de O(n log n) en todos los casos.
- Su única desventaja es su complejidad espacial de O(n).
- Si la cantidad de elementos a ordenar es poca, la implementación secuencial es más rápida. Si es aproximadamente menor a un millón no habrá beneficios grandes al aplicar concurrencia debido a la generación de hilos.
- Si la cantidad de elementos es más grande que un millón o tiende a infinito, la

implementación concurrente es más rápida y eficiente.

- Para 500.000 casi igualan su tiempo de ejecución. La aplicación del algoritmo depende de la necesidad del software, queda a consideración cuando usar una implementación o la otra.
- El punto de equilibrio entre las dos implementaciones depende del hardware del sistema. Mientras más hilos para ejecutar, menos cantidad de elementos para que la concurrencia sea más eficiente.
- La concurrencia altera la complejidad Big O. El grafico de la comparación demuestra bastante bien esto, con la secuencialidad teniendo un O(n log n) y la concurrencia un O(log n).



REFERENCIAS

Anónimo (25 de abril 2025). Merge Sort - Data Structure and Algorithms Tutorials. Merge Sort - Data Structure and Algorithms Tutorials - GeeksforGeeks

Teiva Harsanyi (10 de octubre 2018). Parallel Merge Sort in Java. Parallel Merge Sort in Java - Medium

Rachit Vasudeva (3 de diciembre 2022). Parallel Merge Sort Algorithm. Parallel Merge Sort Algorithm - Medium

MobileDev (14 de octubre 2024) A Beginner's Guide to Time Complexity in Algorithms. A Beginner's Guide to Time Complexity in Algorithms: O (1), O(n), O(n²), and Beyond - Medium

Natalia Roales Gonzáles (21 de agosto 2015). El paralelismo en Java y el framework Fork/Join. El paralelismo en Java y el framework Fork/Join - Adictos al trabajo

Anónimo (Sin fecha) Advantages and Disadvantages of Merge Sort. Advantages and Disadvantages of Merge Sort - Youacademy