QUICKSORT EN JAVA: ANÁLISIS DE RENDIMIENTO SECUENCIAL Y CONCURRENTE

Datos del autor: Ignacio Daniel Puig
Repositorio con el trabajo: https://github.com/ichi1i1/TrabajoQuickSortComparacion.git
Video explicativo : https://youtu.be/TSFrQkUHboA?si=51c4pdJUvz62TchX
E.mail: ignaciopuig2@gmail.com

RESUMEN

Este trabajo final presenta la implementación y comparación del algoritmo de ordenamiento Quick Sort en sus versiones secuencial y concurrente. El objetivo es analizar el rendimiento y la eficiencia de ambas mediante pruebas controladas.

Primero se detalla el funcionamiento del algoritmo secuencial. Luego, se explica la versión concurrente, señalando las partes paralelizadas y las herramientas utilizadas para lograr concurrencia, acompañadas de fragmentos de código ilustrativos.

En la sección de comparación se ejecutan casos de prueba con distintos volúmenes de datos (desde 10 hasta 1.000.000 elementos) y condiciones variadas (ordenados, aleatorios). Los resultados se presentan en tablas con tiempos de ejecución, junto con las especificaciones del hardware usado para asegurar la reproducibilidad.

Por último, se concluye con un análisis de los resultados, destacando las ventajas y desventajas observadas en el desempeño de la implementación concurrente frente a la secuencial.

Keywords: Concurrencia, algoritmos,Quicksort,análisis,Java,rendimiento.

1. INTRODUCCIÓN

Los algoritmos de *Divide y Vencerás* son una estrategia de diseño algorítmico que resuelve un problema dividiéndolo en subproblemas más pequeños, resolviendo cada uno de ellos recursivamente y combinando luego sus soluciones para obtener el resultado final.

Uno de los algoritmos más conocidos que utiliza este enfoque es **Quicksort**, un método eficiente de ordenamiento desarrollado por **Tony Hoare** en 1960.

El funcionamiento del algoritmo comienza eligiendo un pivote dentro de nuestro arreglo (en nuestro ejemplo es el último elemento) para compararlo con el resto de números.

1. Se inicializa un índice **aux** ,justo antes del inicio del segmento a ordenar.

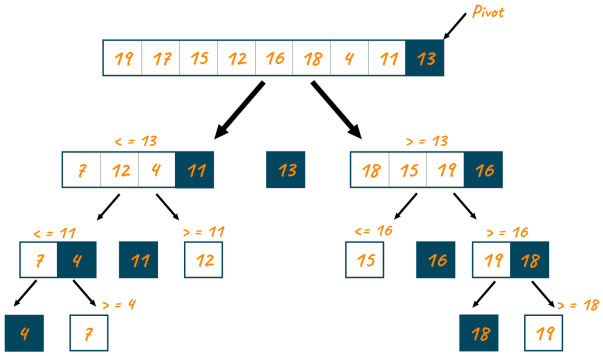
2. Se recorre el arreglo desde el inicio hasta antes del pivote, comparando cada elemento con el pivote.

3. Cada vez que se encuentra un elemento menor o igual al pivote, se incrementa **aux** y se intercambia ese elemento con el elemento en la posición **aux+1**. Esto agrupa todos los valores menores o iguales al pivote al inicio del arreglo.

4. Finalmente, se intercambia el pivote con el elemento que está justo después del último elemento menor o igual, posicionando el pivote en su lugar correcto dentro del arreglo.

Este proceso asegura que todos los elementos a la izquierda del pivote sean menores o iguales, y todos los elementos a la derecha sean mayores, lo que

permite aplicar la misma lógica recursivamente en las sublistas para ordenar todo el arreglo.



En su caso promedio, Quicksort tiene una complejidad de **O(n log n)**, debido a que realiza **$\log_2$(n)** niveles de recursión, ya que en cada nivel el tamaño del problema se reduce a la mitad. En cada uno de estos niveles, se hacen **n** comparaciones para particionar el arreglo. Multiplicando ambos factores, se obtiene el costo total de comparaciones, que es $n \cdot \log_2 n$.

codigo:
https://github.com/ichi1i1/TrabajoQuickSortComparacion/blob/main/QuickSortEspaniol/src/ProyectoParalelo/QSSerial.java

2. IMPLEMENTACIÓN CONCURRENTE

En la versión concurrente que implementamos, la concurrencia se introduce solo en la **primera división** del arreglo. Es decir:

El arreglo principal se divide en dos partes mediante la función de partición.

```
 int s = particion
(datos, 0, datos.length - 1);
```

Se crean dos hilos independientes: cada hilo ordena una de las dos partes del arreglo.

```
 Thread t1 = new Thread
(new QSParallel(datos, 0, s - 1));
 Thread t2 = new Thread
 (new QSParallel(datos, s + 1, datos.length - 1))
```

Cada hilo es una instancia de la clase QSParallel, que implementa la interfaz Runnable. Esta clase define el método run(), que ejecuta el método quickSort en su parte asignada del arreglo:

```
public void run() {
   quickSort
   (this.datos, this.inicio, this.fin);
}
```

Se ejecutan los hilos de manera concurrente y se espera a que ambos terminen:

```
t1.start();
t2.start();
try {
    t1.join();
    t2.join();
} catch (InterruptedException e) {
    System.out.println(e);
}
```

Sin embargo, dentro de cada hilo, el algoritmo sigue funcionando de forma secuencial; no se crean más hilos para ordenar las subdivisiones posteriores.

De esta forma, la paralelización se limita a la primera llamada recursiva, dividiendo el trabajo en dos tareas simultáneas. Esta implementación simple puede acelerar el ordenamiento para arreglos grandes.

Aunque la complejidad algorítmica sigue siendo **O(n log n)** en el caso promedio, esta paralelización inicial puede reducir significativamente el tiempo de ejecución real

Codigo:
https://github.com/ichi1i1/TrabajoQuickSortComparacion/blob/main/QuickSortEspaniol/src/ProyectoParalelo/QSParallel.java

3. COMPARATIVA Y DESEMPEÑO

Para evaluar el rendimiento del algoritmo en sus versiones secuencial y paralela, se realizarán pruebas con distintos tamaños de entrada. Con el fin de obtener resultados más representativos y evitar depender de una única ejecución, que podría

ser atípicamente buena o mala, se harán tres ejecuciones por cada caso y se calculará el promedio de los tiempos obtenidos.

Estas pruebas se ejecutarán próximamente en una computadora con sistema operativo Windows 11, equipada con un disco sólido SSD Kingston A400 de 480 GB, un procesador AMD Ryzen 5 5000 a 3.6 GHz y 16 GB de memoria RAM DDR4-3200 Kingston. El desarrollo y la ejecución del algoritmo se realizará utilizando el lenguaje Java en la versión "21.0.7" LTS dentro del entorno de desarrollo Eclipse.

La medición del tiempo de ejecución se realizará utilizando el método System.nanoTime() de Java. En los casos donde el valor obtenido sea considerablemente alto, se lo convertirá a milisegundos para facilitar la interpretación de los resultados.

Tabla 1 Algoritmo secuencial

N elementos	10	100	1000	10000	100000	1000000
Ejecución 1	5800.0 ns	43900.0 ns	441400.0 ns	1618200 ns	13.4001 ms	70.945 ms
Ejecución 2	6100.0 ns	46000.0 ns	379100.0 ns	1953300 ns	13.1825 ms	72.5588 ms
Ejecución 3	5100.0 ns	55000.0 ns	464600.0 ns	1721100 ns	13.7182 ms	71.231 ms
Promedio	5666.67 ns	48300.0 ns	428366.67ns	1764200 ns	13.4336 ms	71.57826ms

Tabla 2 Algoritmo Paralelo

N elementos	10	100	1000	10000	100000	1000000
Ejecución 1	414900.0 ns	462700.0 ns	779100.0 ns	1981200ns	11.2636 ms	50.2704 ms
Ejecución 2	434700.0 ns	470900.0 ns	838700.0 ns	1820400 ns	10.0924 ms	67.4887 ms
Ejecución 3	399300.0 ns	463600.0 ns	680300.0 ns	1897400.0ns	10.0568 ms	50.7381 ms
Promedio	416300.0 ns	465733.3 ns	766033.3 ns	1899666.6ns	10.4709 ms	56.1657 ms

4. CONCLUSIÓN

Los resultados obtenidos muestran que el algoritmo secuencial es claramente más eficiente para tamaños pequeños de entrada (10, 100 y 1,000 elementos), debido a que la sobrecarga del manejo de hilos en la versión paralela no se compensa con tan pocos datos. Esta sobrecarga incluye la creación, sincronización y coordinación de hilos, que consume tiempo adicional.

A partir de los 10,000 elementos, el algoritmo paralelo comienza a ser competitivo y, desde los 100,000 elementos en adelante, supera en rendimiento al secuencial. En el caso donde comienza a tomar una ventaja significativa, es el de un millón de elementos, donde la versión paralela tardó 56 ms, mientras que la secuencial necesitó 71 ms.

En conclusión, el paralelismo resulta ventajoso para tareas con grandes volúmenes de datos, donde la reducción del tiempo justifica la complejidad adicional, mientras que para conjuntos pequeños es preferible usar el algoritmo secuencial para evitar la sobrecarga innecesaria.

Referencias

Jain, S. (2025, April 17). *Quick Sort*. GeeksforGeeks. https://www.geeksforgeeks.org/quick-sort-algorithm/

Muñoz, A. (2013, 10 de mayo). *Algorítmia - Tema 4: QuickSort - Andŕes Muñoz* [Video]. UCAM Universidad Católica de Murcia. http://www.youtube.com/watch?v=lNL_C9TmjpY

FathiahHusna. (2018). *Quick-Sort-Parallel-* [Repositorio de GitHub]. GitHub. https://github.com/FathiahHusna/Quick-Sort-Parallel-

Visualgo. (n.d.). *Sorting*. https://visualgo.net/en/sorting