ALGORITMO QUICK SORT

Datos del autor: Roldan Riva Camila Belen
Repositorio con el trabajo: https://github.com/camiroldan017/TP-Prog.Concurrente.git
Video explicativo (<10min): Video explicativo
E.mail: camilaroldanriva@gmail.com

RESUMEN

En este trabajo se busca realizar una comparación entre un algoritmo secuencial y su versión concurrente. La comparación se enfoca en medir el rendimiento de ambas implementaciones, siendo la versión concurrente capaz de ejecutar partes del proceso en paralelo mediante hilos (threads). Para ello, se llevaron a cabo distintas pruebas de velocidad con el objetivo de analizar el desempeño en cuanto al tiempo de respuesta y determinar en qué contextos resulta más conveniente utilizar cada enfoque. Para llevar a cabo las pruebas, se seleccionó el algoritmo de ordenamiento QuickSort, utilizando diferentes implementaciones obtenidas de internet como base para la comparación.

Keywords: QuickSort, rendimiento, comparación, concurrente, secuencial.

1. INTRODUCCIÓN

QuickSort es uno de los algoritmos de ordenamiento más utilizados y eficientes en ciencias de la computación, ya que se basa en el principio de divide y vencerás, una estrategia que permite descomponer un problema complejo en subproblemas más pequeños.

Su ejecución comienza con la selección de un elemento dentro del arreglo, conocido como pivote, que sirve como referencia para distribuir el resto de los elementos en dos subconjuntos: aquellos menores y aquellos mayores que el pivote. A partir de esta división y tras la partición, el algoritmo repite el procedimiento de manera recursiva sobre los subproblemas generados a cada lado del pivote, asegurando que el proceso continúe hasta que cada grupo contenga solo un elemento o ninguno, momento en el cual se considera ordenado.

Debido a su rapidez y eficiencia, QuickSort es comúnmente utilizado en estructuras que requieren procesamiento intensivo de datos, ya que su tiempo de ejecución promedio tiene una complejidad de O(nlogn), lo que lo hace significativamente más rápido que métodos como el ordenamiento por burbuja o inserción.

La versión original (secuencial/recursivo) del código fuente que se utilizó es la siguiente: QuickSort - Secuencial/Recursivo

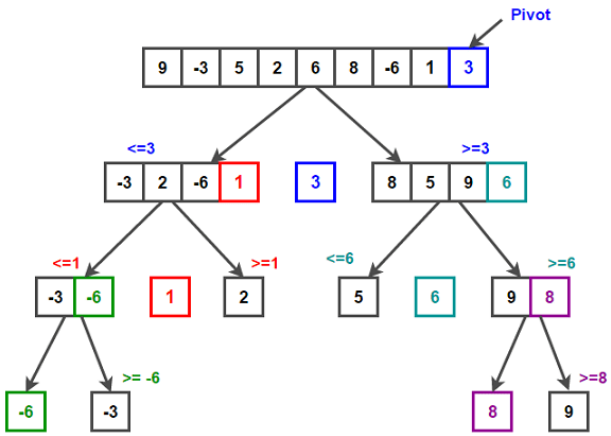

Figura 1 *Funcionamiento recursivo de QuickSort.*

2. IMPLEMENTACIÓN CONCURRENTE

En una implementación concurrente, no es necesario esperar a que un lado del arreglo termine de ordenarse antes de continuar, ya que podemos ejecutar ambos procesos en paralelo mediante hilos. Para lograr esto, utilizaremos la clase RecursiveTask del paquete java.util.concurrent, que nos permitirá dividir el trabajo en subtareas independientes y optimizar el procesamiento mediante ejecución concurrente, aprovechando la capacidad de múltiples núcleos para mejorar el rendimiento del algoritmo. Una vez realizada la partición del arreglo y ubicado el pivote en su posición correcta, el algoritmo divide el problema

en dos subtareas: la parte izquierda y la parte derecha del arreglo. Para introducir concurrencia, se utiliza el método fork() sobre la parte izquierda, lo que permite que se ejecute en un hilo separado. En cambio, la parte derecha se procesa secuencialmente en el hilo principal mediante el método compute(), por lo que no forma parte de la ejecución concurrente. Finalmente, se invoca el método join() para esperar a que finalice la ejecución de la subtarea paralela y así asegurar que ambas partes se completen correctamente.

Cabe destacar que el arreglo utilizado en las pruebas se genera de forma aleatoria mediante una clase auxiliar encargada de cargarlo con elementos aleatorios.

La versión concurrente del código fuente que se utilizó es la siguiente: [QuickSort - Concurrente]

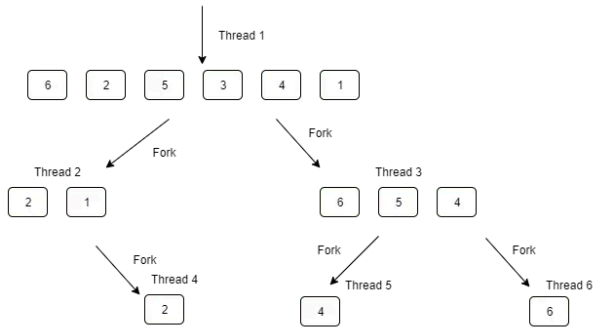

Figura 2 *Funcionamiento concurrente de QuickSort.*

3. COMPARATIVA Y DESEMPEÑO

Para comparar y evaluar el desempeño de ambos algoritmos y medir su tiempo de respuesta, realicé una serie de pruebas variando el tamaño del arreglo, utilizando valores de 10, 100, 1000, 5000, 10000, 100000, 500000 y 1000000 elementos. En cada prueba, los datos del arreglo fueron generados de manera aleatoria, lo que significa que los valores utilizados en cada ejecución son distintos, introduciendo cierta variabilidad en los resultados y evitando una comparación completamente exacta entre los algoritmos. Sin embargo, esta aproximación permite obtener una visión general sobre su rendimiento en distintos escenarios .

A partir de tamaños superiores a 100000 elementos, las diferencias en los tiempos de ejecución comienzan a ser notorias, donde la versión concurrente logra una mejora significativa en rendimiento y reduce considerablemente el tiempo de procesamiento. En contraste, cuando el tamaño del arreglo es pequeño, la diferencia entre ambos enfoques no es tan marcada, ya que el beneficio de la concurrencia se vuelve menos relevante en estructuras de menor tamaño.

Especificaciones:

Procesador que utilizo: AMD a8-9600 (4 núcleos)

Tabla 1. *Comparativa de algoritmos QuickSort.*

		Tiempo (microSegundos)	
Algoritmo	Tamaño del array	TEST 1	TEST 2
Secuencial	10	1099 microSegundos	1199 microSegundos
Secuencial	100	1843 microSegundos	1695 microSegundos
Secuencial	1000	1981 microSegundos	3976 microSegundos
Secuencial	10000	14835 microSegundos	4813 microSegundos
Secuencial	100000	39453 microSegundos	46642 microSegundos
Secuencial	500000	161294 microSegundos	120278 microSegundos
Secuencial	1000000	579692 microSegundos	338652 microSegundos
Concurrente	10	5106 microSegundos	9383 microSegundos
Concurrente	100	7799 microSegundos	4921 microSegundos
Concurrente	1000	7568 microSegundos	8847 microSegundos
Concurrente	10000	18361 microSegundos	15326 microSegundos
Concurrente	100000	38330 microSegundos	39928 microSegundos
Concurrente	500000	62011 microSegundos	87611 microSegundos
Concurrente	1000000	318911 microSegundos	312998 microSegundos

4. CONCLUSIÓN

Hemos logrado llegar a la conclusión de que el algoritmo QuickSort concurrente es significativamente más eficiente para arrays de gran tamaño, ya que reduce el tiempo de procesamiento en comparación con su versión secuencial. Sin embargo, cuando el número de elementos es pequeño, la diferencia entre ambos enfoques es mínima, debido a que la sobrecarga generada por la gestión de múltiples hilos no compensa el beneficio del paralelismo.

Estos resultados resaltan la importancia de elegir el enfoque adecuado dependiendo del tamaño del array y la cantidad de elementos a procesar. En problemas con grandes estructuras de datos, la concurrencia permite optimizar los tiempos de ejecución, aprovechando mejor la capacidad de procesamiento de múltiples núcleos. En contraste, para arreglos más pequeños, la ejecución secuencial sigue siendo una opción eficiente, dado que el impacto de la paralelización no es significativo.

En definitiva, ninguna implementación es inherentemente superior a la otra, sino que su eficiencia depende del contexto en el que se aplique. La elección entre QuickSort secuencial o concurrente debe basarse en el tamaño del array, la disponibilidad de recursos y la necesidad de optimización del rendimiento.

REFERENCIAS

GeeksforGeeks. (2025, 17 abril). Quick sort.

GeeksforGeeks. https://www.geeksforgeeks.org/quick-sort-algorithm/

GeeksforGeeks. (2023, 14 septiembre). Quick Sort using Multithreading.

GeeksforGeeks. https://www.geeksforgeeks.org/quick-sort-using-multi-threading/

Multi-threaded quick sort - expert mentoring, customized solutions. (s.f.)

https://www.venkys.io/articles/details/multi-threaded-quick-sort