Comparación entre QuickSort Secuencial y Concurrente

Datos del Autor: Luca Nicolás Lazarte – DNI 46.623.965
Repositorio con el Trabajo: https://github.com/LucaLzt/QuickSort_Concurrency
Video Explicativo (<10min): https://youtu.be/4Op7qF6vKjU
Email: luca.lazarte05@gmail.com

RESUMEN (ABSTRACT)

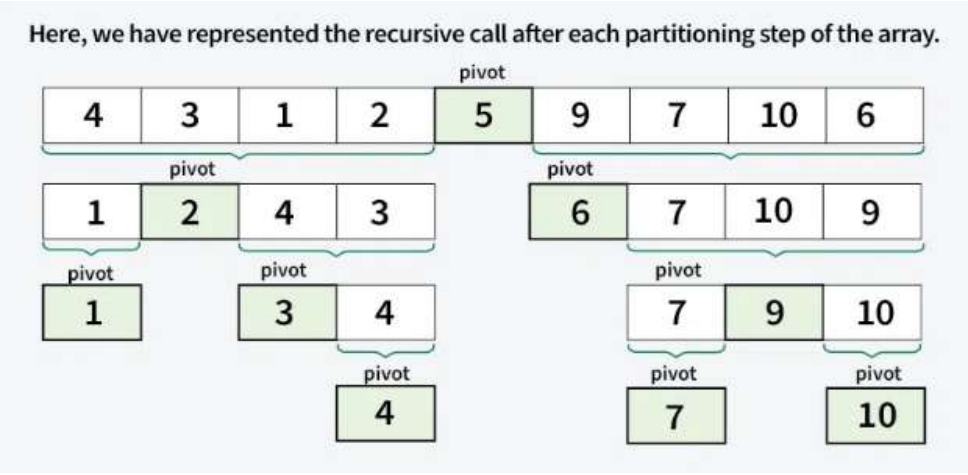
Este trabajo presenta una comparación entre la implementación secuencial y concurrente del algoritmo de ordenamiento QuickSort en el lenguaje de programación Java. Se detalla el funcionamiento de ambas versiones, analizando sus características, ventajas y limitaciones. La versión concurrente hace uso de hilos (Thread) para dividir el problema en subproblemas paralelos, aprovechando múltiples núcleos del procesador. Se realizaron pruebas con arreglos de diferentes tamaños para medir el tiempo de ejecución y se observó una mejora significativa en tiempos con la versión concurrente, particularmente con arreglos grandes. El código fue extraído y adaptado de recursos disponibles públicamente y se encuentra comentado línea por línea para su comprensión.

1. INTRODUCCIÓN

https://github.com/LucaLzt/QuickSort_Concurrency/blob/master/src/test/QuickSort_Sequential.java

QuickSort es un algoritmo de ordenamiento altamente eficiente, fundamentado en el paradigma de "divide y vencerás". En su implementación secuencial, el algoritmo selección un pivote y procede a particionar el arreglo en dos subarreglos, para luego aplicar recursivamente el mismo procedimiento en cada partición.

Figura 1 – Representación del funcionamiento de QuickSort (Divide y Vencerás)



Nota: La figura ilustra cómo funciona el algoritmo QuickSort utilizando recursión. En cada paso se selecciona un pivote, se divide el arreglo en subarreglos menores y mayores al pivote, y se aplica el mismo procedimiento a cada partición.

El funcionamiento de QuickSort puede resumirse en los siguientes pasos:

- **Selección del pivote**: Se elige un elemento del arreglo que actuará como referencia para la partición. Las estrategias de selección más comunes incluyen el primer o último elemento del arreglo, un elemento aleatorio o la mediana.
- **Partición del arreglo**: Se reorganizan los elementos de manera que aquellos menores al pivote queden ubicados a su izquierda y los mayores a su derecha. Al finalizar, el pivote se encuentra en su posición definitiva dentro del arreglo ordenado.
- **Aplicación recursiva**: El procedimiento anterior se aplica de forma recursiva sobre los subarreglos izquierdo y derecho hasta que cada partición contenga un solo elemento o esté vacío.

Uno de los aspectos más determinantes para el rendimiento del algoritmo es la estrategia de selección del pivote, siendo las más utilizadas las siguientes:

- **Último elemento**: De implementación sencilla, aunque puede inducir el peor caso de complejidad si el arreglo ya se encuentra ordenado.
- **Elemento aleatorio**: Minimiza la probabilidad del peor caso al evitar patrones predecibles en los datos.
- **Mediana**: Permite una división equilibrada del arreglo, optimizando el desempeño, aunque su cálculo puede resultar costoso computacionalmente.

Por otro lado, el proceso de partición constituye el núcleo del algoritmo. Entre las técnicas de particionado más relevantes se encuentran:

- **Partición ingenua**: Utiliza estructuras auxiliares para almacenar los elementos menores y mayores al pivote, combinándolos posteriormente. Presenta una mayor demanda de memoria.
- **Partición de Lomuto**: Emplea un único índice para seguir la posición de los elementos menores al pivote, realizando los intercambios necesarios en una sola pasada. Es de implementación simple, aunque menos eficiente en algunos escenarios.
- **Partición de Hoare**: Utiliza dos índices que se desplazan desde los extremos hacia al centro, intercambiando elementos según corresponda. Esta técnica suele ser más eficiente, reduciendo la cantidad de intercambios necesarios.

Las complejidades asociadas a QuickSort son:

- **Caso promedio**: $O(n \log n)$
- **Pero caso**: $O(n^2)$, que se presenta cuando el pivote genera divisiones muy desbalanceadas.
- **Mejor caso**: $O(n \log n)$, cuando el pivote divide el arreglo de forma equitativa.
- **Complejidad espacial**: $O(\log n)$, atribuida al uso de la pila de llamadas recursivas.

Finalmente, es importante destacar que QuickSort no es un algoritmo estable. Esto implica que, en el caso de elementos con valores equivalentes, no se garantiza la preservación de su orden relativo original dentro del arreglo resultante. Esta característica puede ser determinante en aplicaciones donde la estabilidad del ordenamiento sea un requisito.

2. IMPLEMENTACIÓN CONCURRENTE

https://github.com/LucaLzt/QuickSort_Concurrency/blob/master/src/test/QuickSort_Concurrent.java

La versión concurrente del algoritmo QuickSort permite aprovechar el paralelismo ofrecido por los sistemas multinúcleo, mejorando significativamente el rendimiento frente a grandes volúmenes de datos. Esta implementación, se basa en la creación de múltiples hilos de ejecución (Threads), delegando el procesamiento de cada subarreglo a un hilo distinto durante la partición recursiva del arreglo.

Figura 2 – Concurrencia en el algoritmo

```
// Control para evitar crear demasiados hilos (overthreading)
if(depth < MAX_DEPTH) {
        // Crea hilos para ordenar las mitades izquierda y derecha concurrentemente
        Thread left = new QuickSort_Concurrent(arr, low, pivot - 1, depth + 1);
        Thread right = new QuickSort_Concurrent(arr, pivot + 1, high, depth + 1);

        // Inicia los hilos
        left.start();
        right.start();

        // Espera que ambos hilos terminen
        try {
                left.join();
                right.join();
        } catch (InterruptedException e) {
                e.printStackTrace(); // --- Manejo básico de error si se interrumpen los hilos --- //
        }
} else {
        // --- Si se alcanzó el límite máximo depth, ordena secuencialmente sin crear más hilos ---
        quickSort(arr, low, pivot - 1, depth);
        quickSort(arr, pivot + 1, high, depth);
}
```

En Java, esta lógica se implementa utilizando la clase Thread, permitiendo que cada partición (izquierda y derecha) del arreglo sea ordenada en paralelo. Sin embargo, una creación descontrolada de hilos puede ser contraproducente, generando una sobrecarga del sistema (Overthreading), consumo excesivo de memoria y degradación del rendimiento general.

Para mitigar este problema, se introduce un mecanismo de control basado en dos variables clave:

- **MAX_DEPTH**: Define la profundidad máxima de recursión concurrente. Establece un límite hasta el cual se permite la creación de nuevos hilos.
- **depth**: Representa el nivel actual de recursión. Este contador se incrementa cada vez que el arreglo se divide en nuevas subparticiones.

Con este enfoque, el algoritmo sigue un principio claro:

Si depth < MAX_DEPTH, se crean nuevos hilos para continuar la ejecución concurrente. De lo contrario, el procesamiento de las siguientes particiones se realiza de forma secuencial.

Este control resulta fundamental para mantener un equilibrio entre la paralelización y el consumo eficiente de recursos.

Cada nivel de recursión concurrente genera hasta dos nuevos hilos, uno para cada subarreglo resultante. Por tanto, el número máximo de hilos generados está acotado por la siguiente expresión: Máximo de hilos teóricos = 2^{MAX_DEPTH}. De esta forma:

- Para MAX_DEPTH = 3, se crearán máximo $2^3 = 8$ hilos.
- Para MAX_DEPTH = 8, se crearán máximo $2^8 = 256$ hilos.

La elección de un valor adecuado para MAX_DEPTH debe estar en función del tamaño del arreglo y de la capacidad de hardware disponible. Un ejemplo en base al hardware utilizado para las pruebas (Ryzen 5 4650G – 6 Núcleos – 12 Hilos):

- Para arreglos de hasta 1 millón de elementos, se recomienda un MAX_DEPTH de 3, garantizando una concurrencia moderada (8 hilos).
- Para arreglos superiores al millón, puede optarse por un MAX_DEPTH de 8 para maximizar el aprovechamiento de núcleos (hasta 256 hilos).

El control de profundidad presenta las siguientes ventajas:

- Evita la saturación del procesador.

- Limita el consumo excesivo de memoria y recursos del sistema.
- Mejora la escalabilidad del algoritmo en entornos con múltiples núcleos.
- Conserva la eficiencia de QuickSort sin comprometer la estabilidad del sistema operativo.

3. COMPARATIVA Y DESEMPEÑO

Se realizaron pruebas con arreglos de tamaños variados y se compararon los tiempos de ejecución. Las pruebas se realizaron en una PC con las siguientes características:

- **Procesador**: Ryzen 5 4650G, 6 núcleos / 12 hilos.
- **RAM**: 16GB.
- **Sistema Operativo**: Windows 11.
- **Java Versión**: 21.

Tabla 1 Comparativas (Comparación en milisegundos)

Tamaño del Arreglo	Secuencial	Concurrente (8 hilos)	Concurrente (256 hilos)
10	1ms	2ms	2ms
1.000	1ms	2ms	24ms
100.000	14ms	19ms	63ms
1.000.000	86ms	47ms	102ms
100.000.000	10048ms	5248ms	2287ms

Los resultados de las pruebas demuestran que la versión secuencial es más eficiente en arreglos pequeños y medianos (hasta aproximadamente 100.000 elementos). Esto se debe a la sobrecarga que implica la creación y sincronización de múltiples hilos en la versión concurrente, lo cual introduce un costo adicional que no se ve compensado por los beneficios de la paralelización.

Sin embargo, al aumentar significativamente el tamaño del arreglo, como en el caso de 100 millones de elementos, la versión concurrente logra mejorar el tiempo de ejecución si se utilizan más hilos y se gestiona correctamente la concurrencia. En ese escenario, el rendimiento mejora de forma notable, alcanzando tiempos de ejecución menores que los del algoritmo secuencial.

Estos resultados evidencian que la eficiencia de la concurrencia depende en gran medida del tamaño del problema y del número adecuado de hilos, así como de una estrategia de control para evitar la saturación del sistema.

4. CONCLUSIÓN GENERAL DEL TRABAJO

Este trabajo permitió comparar de manera concreta y práctica los enfoques secuencial y concurrente del algoritmo QuickSort, comprendiendo las ventajas y limitaciones de cada uno. La experiencia evidenció que la programación concurrente puede ser una herramienta poderosa para mejorar el rendimiento de algoritmos clásicos, siempre que se aplique de forma controlada y en contextos adecuados.

La implementación en Java y la realización de pruebas en distintos tamaños de arreglo ayudaron a profundizar en aspectos clave como la creación de hilos, la sincronización y el impacto del hardware en la ejecución. Además, permitió visualizar cómo un mal uso de la concurrencia puede incluso empeorar el rendimiento.

En conclusión, el trabajo aportó no solo una mejora en la comprensión técnica del algoritmo QuickSort, sino también habilidades prácticas en programación multihilo, análisis de rendimiento y evaluación crítica de cuándo y cómo aplicar la concurrencia en el desarrollo de software.

5. REFERENCIAS

Baeldung. (2022). *QuickSort in Java*. Recuperado de https://www.baeldung.com/java-quicksort

Geeks for Geeks. (2023). *Quick Sort*. Recuperado de https://www.geeksforgeeks.org/quick-sort-algorithm/

Medium. (2024). *Mastering QuickSort in Java: Partition, Pivot and Time Complexity Explained*. Recuperado de https://medium.com/@YodgorbekKomilo/mastering-quicksort-in-java-partition-pivot-and-time-complexity-explained-2306696d28e2

Oracle. (n.d.). *Class Thread (Java Platform SE 8)*. Recuperado de https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html

Normas APA. (s.f.). *Referencias en normas APA (7º edición)*. Recuperado de https://normas-apa.org/referencias/