QuickSort Concurrente y Secuencial:
Un Análisis Comparativo Aplicado en Java

Datos del autor
Joaquin Vadillo
Video Explicativo
Repositorio con el trabajo
Joaquinvadillo1@gmail.com

RESUMEN (ABSTRACT)

En este trabajo analicé el comportamiento del algoritmo de ordenamiento QuickSort implementado en Java, comparando su ejecución en dos variantes: una versión secuencial y otra concurrente basada en el framework Fork/Join. El objetivo principal fue explorar cómo la programación concurrente puede influir en el rendimiento de algoritmos clásicos cuando se aplican sobre estructuras de datos de gran tamaño. Para ello, desarrollé ambas implementaciones y realicé pruebas controladas con arreglos de distintos tamaños. A lo largo del estudio, observé cómo se comporta cada enfoque en distintos escenarios y qué factores podrían afectar su desempeño. Este informe busca no solo mostrar una comparación técnica, sino también reflexionar sobre las ventajas, desafíos y posibles aplicaciones de incorporar paralelismo en procesos de ordenamiento. El código fuente se encuentra documentado línea por línea en castellano y alojado en un repositorio de acceso público.

Keywords:QuickSort, Concurrencia, Programación paralela, Java y Algoritmos de ordenamiento

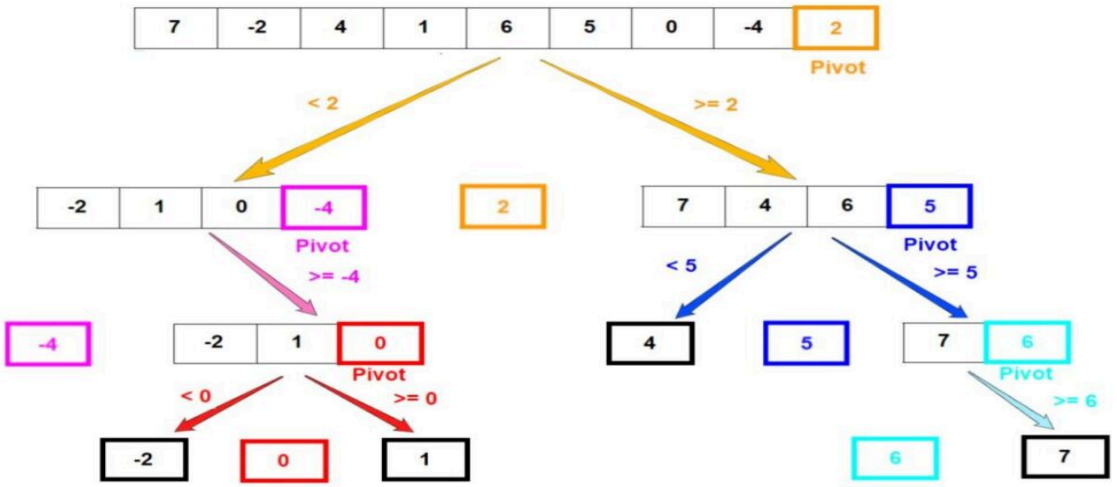
1. INTRODUCCIÓN

El algoritmo QuickSort es uno de los métodos de ordenamiento más utilizados en informática debido a su eficiencia y diseño recursivo basado en la técnica divide y vencerás. Su funcionamiento se basa en seleccionar un elemento llamado pivote, particionar el arreglo en dos subarreglos con valores menores y mayores al pivote, y luego ordenar recursivamente esas particiones hasta que todo el arreglo quede ordenado. Este algoritmo tiene un tiempo de ejecución rápido en la mayoría de los casos, lo que lo hace adecuado para ordenar grandes conjuntos de datos de manera eficiente. La implementación tradicional o secuencial de QuickSort ejecuta todas las operaciones de particionado y ordenamiento de manera lineal, una tras otra, sin aprovechar capacidades de procesamiento paralelo. Aunque esta versión es eficiente para muchos casos, puede verse limitada en rendimiento cuando se manejan volúmenes de datos muy grandes o cuando se ejecuta en sistemas con múltiples núcleos de procesamiento.

En este trabajo se presenta una comparación entre la implementación secuencial de QuickSort y una versión concurrente desarrollada con el framework Fork/Join de Java. La versión secuencial sirve como referencia para medir el impacto que tiene la programación concurrente en el desempeño del algoritmo, explorando cómo el paralelismo puede mejorar los tiempos de ordenamiento en arquitecturas modernas multinúcleo.

A continuación se muestra una imagen que representa el funcionamiento del Algoritmo Quicksort secuencial

Figura 1
Funcionamiento de Quicksort Secuencial. Link de github con el código



Código Java. Ordenación QuickSort [Imagen]. https://codigojava.online/ordenacion-qucksort/

2.IMPLEMENTACIÓN CONCURRENTE

La implementación concurrente del algoritmo QuickSort mantiene el mismo principio básico que la versión secuencial: seleccionar un pivote, particionar el arreglo en subarreglos con elementos menores y mayores al pivote, y ordenar recursivamente esas particiones. Sin embargo, esta versión aprovecha el paralelismo que ofrece Java a través del framework Fork/Join, diseñado para dividir tareas en subtareas que se ejecutan de manera simultánea en múltiples núcleos de procesamiento.

En concreto, después de elegir el pivote y realizar la partición, la implementación concurrente crea dos tareas independientes para ordenar cada subarreglo. Estas tareas se representan mediante una clase que extiende RecursiveAction, una clase abstracta que permite definir tareas sin valor de retorno.

La recursión concurrente se implementa en el método compute(), donde se invocan ambas subtareas utilizando el método invokeAll().

La gestión y coordinación de estas tareas la realiza un ForkJoinPool, que distribuye automáticamente la carga de trabajo entre los hilos disponibles en el sistema, aprovechando la arquitectura multinúcleo de los procesadores modernos. Esto puede traducirse en una mejora significativa en los tiempos de ejecución, especialmente cuando se trabaja con arreglos de gran tamaño. No obstante, para arreglos pequeños el overhead asociado a la creación y sincronización de tareas puede superar los beneficios de la concurrencia, por lo que se suele implementar un umbral mínimo para aplicar paralelismo.

A continuación, se mostrará un fragmento representativo del código concurrente que ilustra la creación y ejecución de tareas en paralelo.

Código 1
Fragmento del Código Concurrente. Link de github con el código

```java
// Clase que implementa QuickSort concurrente usando Fork/Join
public class QuickSortConcurrente extends RecursiveAction {
    // ID de versión de clase serializable (buena práctica)
    private static final long serialVersionUID = 1L;

    // Umbral a partir del cual se hace recursión paralela, debajo de eso se usa el secuencial
    private static final int UMBRAL_DIRECTO = 10_000;

    // Arreglo que se va a ordenar
    private final int[] array;

    // Índice inicial del segmento a ordenar
    private final int inicio;

    // Índice final del segmento a ordenar
    private final int fin;

    // Constructor que inicializa los campos necesarios para la tarea
    public QuickSortConcurrente(int[] array, int inicio, int fin) {
        this.array = array;
        this.inicio = inicio;
        this.fin = fin;
    }

    // Método que define el trabajo que realiza esta tarea (Fork/Join)
    @Override
    protected void compute() {
        // Si el segmento es válido (al menos dos elementos)
        if (inicio < fin) {
            // Si el segmento es chico, usamos el algoritmo secuencial
            if ((fin - inicio) < UMBRAL_DIRECTO) {
                QuickSortSecuencial.quickSort(array, inicio, fin);
            } else {
                // En caso contrario, particionamos el arreglo y dividimos la tarea en dos
                int pivote = dividir(array, inicio, fin);

                // Creamos subtareas para la parte izquierda y derecha
                QuickSortConcurrente izquierda = new QuickSortConcurrente(array, inicio, pivote - 1);
                QuickSortConcurrente derecha = new QuickSortConcurrente(array, pivote + 1, fin);

                // Ejecutamos ambas subtareas en paralelo
                invokeAll(izquierda, derecha);
            }
        }
```

3. COMPARATIVA Y DESEMPEÑO

Para evaluar el impacto de la programación concurrente en el rendimiento del algoritmo QuickSort, se realizaron pruebas comparativas entre la implementación secuencial y la concurrente con Fork/Join. Se utilizaron arreglos de tamaños variados, desde 1,000 hasta 5,000,000 elementos, y un sistema con 6 núcleos disponibles para la ejecución paralela.

Los resultados obtenidos evidencian que para arreglos pequeños (1,000 a 10,000 elementos), la versión secuencial presenta un mejor desempeño. Esto se debe principalmente al costo adicional que implica la creación y administración de hilos en la versión concurrente, el cual supera los beneficios del paralelismo cuando el tamaño del arreglo es reducido. A medida que aumenta el tamaño del arreglo, la versión concurrente comienza a mostrar ventajas significativas. Para 1,000,000 de elementos, el tiempo concurrente (0.0206 s) es aproximadamente 4 veces menor que el secuencial (0.0859 s). Esta diferencia podría ampliarse con arreglos más grandes, donde

el algoritmo concurrente reduce considerablemente el tiempo de ejecución, llegando a ser aproximadamente 4 veces más rápido para 5,000,000 de elementos. A continuación, se presenta una tabla comparativa que permite visualizar claramente las diferencias de rendimiento entre ambas implementaciones (la secuencial y la concurrente) según el tamaño del arreglo.

Tabla 1

Tabla comparativa que muestra los tiempos de ejecución de las versiones secuencial y concurrente del algoritmo QuickSort para distintos tamaños de arreglos (Los valores expresados corresponden a una única ejecución del programa en un entorno con 6 hilos disponibles, los resultados pueden variar ligeramente entre ejecuciones.)

Tamaño del arreglo	Tiempo Secuencial (s)	Tiempo Concurrente (s)	Reducción de tiempo (%) respecto a la versión secuencial
1.000	0.000971	0.002404	-147.57%
10.000	0.000682	0.000956	-40.23%
100.000	0.007087	0.007832	-10.48%
1.000.000	0.085864	0.020571	76.00%
3.000.000	0.272595	0.080718	70.37%
5.000.000	0.465473	0.112907	75.76%

Elaboración propi

4. CONCLUSIÓN

En este trabajo se compararon dos implementaciones del algoritmo QuickSort: una versión secuencial y otra concurrente desarrollada con el framework Fork/Join de Java. A través de pruebas con diferentes tamaños de arreglos, se observó que la implementación concurrente no presenta mejoras significativas en arreglos pequeños, debido al costo adicional de creación y gestión de tareas paralelas. Sin embargo, para arreglos de gran tamaño, la versión concurrente demostró una mejora notable en los tiempos de ejecución, logrando una reducción superior al 70 % en ciertos casos. Estos resultados respaldan la idea de que la programación concurrente es particularmente efectiva en entornos donde se cuenta con múltiples núcleos de procesamiento. Según Thapar y Jain (2018), “el tiempo de ejecución de QuickSort-Fork Join es aproximadamente un 46 % más rápido que la versión secuencial” cuando se utiliza en procesadores multinúcleo. Por su parte, Oracle (s.f.) sostiene que “dividir tareas en fragmentos que puedan ejecutarse en paralelo maximiza el uso del poder computacional disponible”. Además de los beneficios en cuanto al desempeño, este tipo de comparativas permite valorar el rol de la

programación concurrente en el diseño de algoritmos modernos, no solo desde el punto de vista técnico, sino también en su aplicación práctica en la industria, donde la eficiencia puede impactar directamente en los costos de procesamiento. Desde el ámbito educativo, analizar versiones secuencial y concurrente de un mismo algoritmo ayuda a comprender conceptos clave como la recursividad, la paralelización de tareas y el uso eficiente de los recursos del sistema.En conclusión, se destaca que la implementación concurrente del algoritmo QuickSort es especialmente beneficiosa en contextos de procesamiento intensivo y con hardware adecuado, mientras que para volúmenes de datos pequeños, la versión secuencial sigue siendo más eficiente por su simplicidad y menor sobrecarga computacional. Estas observaciones refuerzan la importancia de seleccionar la estrategia de implementación adecuada según el contexto y objetivos.

REFERENCIAS

-Oracle. (s.f.). Fork and join: Java can excel at painless parallel programming too! Recuperado: https://www.oracle.com/technical-resources/articles/java/fork-join.html

-Thapar, S., & Jain, S. (2018). Performance evaluation of three quicksorting algorithms on single- and multi-core processors. Recuperado: https://www.researchgate.net/publication/332106031_Performance_Evaluation_of_Three_Quick_sorting_Algorithms_on_a_Single_and_Multi-core_Processors

-Ordenación QuickSort. (s.f.). CódigoJava. Recuperado : https://codigojava.online/ordenacion-quicksort/

-GeeksforGeeks. (s.f.). *QuickSort Algorithm*. GeeksforGeeks. Recuperado: https://www.geeksforgeeks.org/quick-sort-algorithm/