ANÁLISIS DEL ALGORITMO QUICKSORT EN SU IMPLEMENTACIÓN SECUENCIAL Y PARALELA

Autor: Ojeda Lucas Gerónimo
Repositorio con el trabajo: https://github.com/lucasojeda98/quicksort-analysis
Video explicativo: https://drive.google.com/drive/folders/1zrzbzsj3ZLudMshCcb-fPWXic8mPrjDu?usp=drive_link
E-mail: lucasg.ojeda98@gmail.com

RESUMEN (ABSTRACT)

En el presente trabajo de investigación se aborda el algoritmo de ordenamiento quicksort, explicando sus fundamentos y analizando su complejidad temporal para destacar las condiciones necesarias para alcanzar una ejecución eficiente. Se presenta la implementación del algoritmo tanto en su versión secuencial como paralela en el lenguaje de programación Java, y utilizando técnicas de programación concurrente se busca mejorar su rendimiento dentro de un sistema multinúcleo. Finalmente, se realiza la comparativa del algoritmo en ambas versiones para evaluar su desempeño teniendo en cuenta diferentes tamaños de datos. Los resultados muestran una mejora significativa en la velocidad de ejecución en su versión paralela, pero no en la totalidad de los casos.

Keywords: Algoritmo, Complejidad, Concurrencia, Hilos, Eficiencia, Recursividad.

1. INTRODUCCIÓN

El algoritmo de ordenamiento quicksort se basa en la premisa de "divide y vencerás". Su funcionamiento consiste en tomar un array desordenado y seleccionar un elemento a modo de pivote, cuya posición puede ser elegida según distintos criterios, por ejemplo, siendo el primer elemento, el último o aleatorio. Una vez elegido, se reorganiza el array de manera que todos los valores menores al pivote queden a su izquierda y todos los mayores a su derecha. El mismo proceso se repite recursivamente para cada subarray formado alrededor del pivote hasta que cada elemento esté en su posición correspondiente.

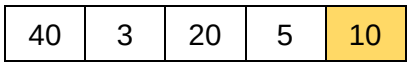
Este método de ordenamiento toma como parámetros al array que se desea ordenar y dos índices, que en el presente trabajo se denominarán *lowIndex* y *highIndex*, y cuya función es establecer los límites del array en cada llamado a la función (para el llamado inicial *lowIndex = 0 y highIndex = tamañoArray – 1*).

1.1 - Funcionamiento

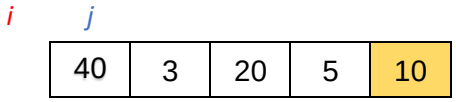
El ordenamiento consiste en recorrer el array para comparar cada elemento con el pivote y realizar las permutaciones necesarias para colocarlos del lado correspondiente al mismo. Para ello se utilizan dos variables, que en este caso se llamarán i y j. La variable i guarda la posición del último elemento menor al pivote encontrado, y j señala la posición actual durante el recorrido del array.

A continuación, se ejemplifica el algoritmo mediante una representación gráfica tomando como pivote al último elemento:

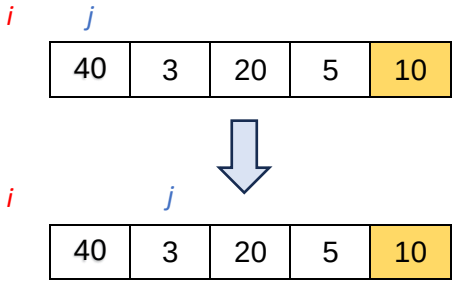
1) Se parte de un array desordenado y se elige al último elemento como pivote.



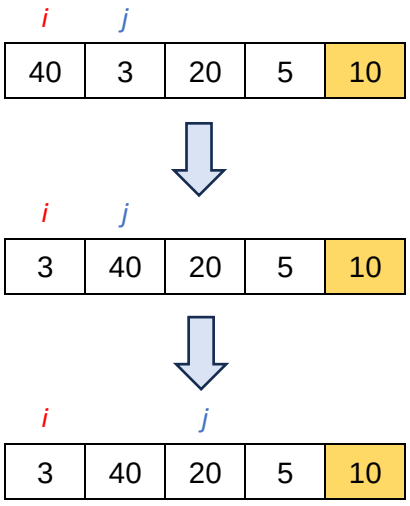
2) Se inician las variables i y j en su estado inicial ($i = lowIndex - 1$ y $j = array[0]$).



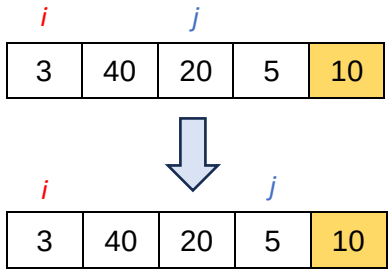
3) Se comienza a recorrer el array. Si el valor en la posición j es mayor al pivote (40>10), lo único que se hace es pasar a la siguiente posición ($j = 3$).



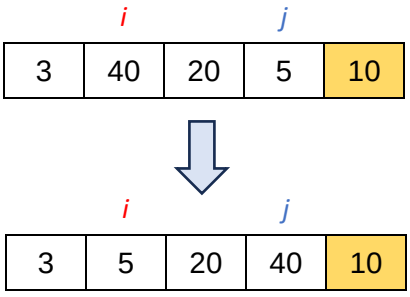
4) Si el valor en la posición j es menor al pivote (3 < 10), se incrementa i en uno, se permuta su valor con j y se pasa al siguiente elemento.



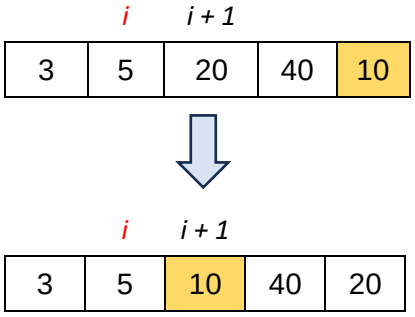
5) En el siguiente caso 20>10, por lo que se pasa a la próxima posición.



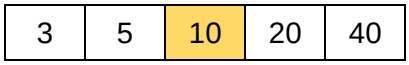
6) Como 5 < 10, se incrementa i en uno y se permuta su valor con j. En este punto ya no quedan más elementos por comparar.



7)Cuando ya se compararon todos los elementos con el pivote, solamente queda ubicar a este último en su lugar correspondiente, por lo que se lo intercambia con el valor de la posición *i + 1*.



8) Una vez que se tienen los elementos menores al pivote a su izquierda y los elementos mayores a su derecha, se realiza el mismo procedimiento para ambos subarrays. Como el subarray izquierdo [3, 5] ya está ordenado, se sigue con el subarray derecho [40, 20]. Se elige el 20 como pivote y se realizan los pasos explicados anteriormente para que finalmente el array original quede ordenado.



1.2 - Complejidad

La complejidad del algoritmo quicksort puede variar según el método de elección del pivote y la semejanza en el tamaño de los subarreglos generados. Sin embargo, se puede destacar que el caso optimista y el caso promedio resultan muy similares, siendo ambos $O(n \log n)$, por lo que se lo considera un algoritmo eficiente a la hora de trabajar con arrays de gran tamaño. Por otro lado, el peor caso (menos probable) resulta en una complejidad de $O(n^2)$.

Caso optimista: Se supone un caso favorable cuando cada iteración se produce una división equitativa alrededor del pivote (Rajput et al, 2012), y se tiene el siguiente árbol recursivo:

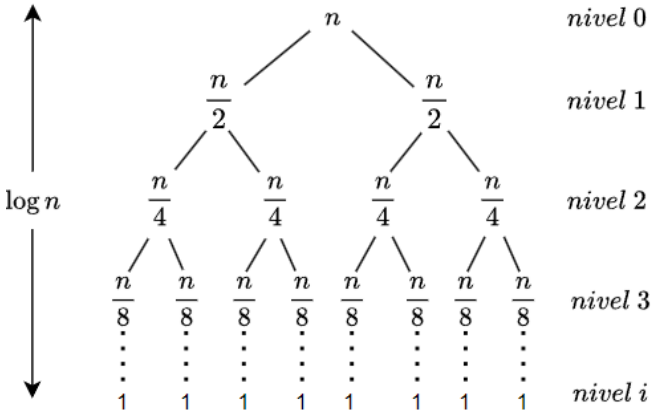


Figura 1- Mejor caso de QuickSort. Adaptado de "Performance Comparison of Sequential Quick Sort and Parallel Quick Sort Algorithm" (p.15), Ishwari Singh Rajput et al, 2012, International journal of computer applications, 57(9)

Partiendo de un array de tamaño n en el nivel $i = 0$, se divide sucesivamente cada nivel en 2^i partes hasta llegar al caso base 1 (donde todos los elementos están en su posición correspondiente). Por lo tanto, se determina la altura del árbol de la siguiente manera:

$$\frac{n}{2^i} = 1$$

$$n = 2^i$$

$$\log n = \log 2^i$$

$$\log n = i \text{ niveles}$$

Entonces, dado un array de tamaño n, este se deberá descomponer en $\log n$ niveles hasta que todos los elementos estén ordenados. Adicionalmente, recorrer el array en cada nivel posee un costo de $2^i(\frac{n}{2^i} - 1)$, es decir, el producto entre el número de particiones en un nivel y el tamaño menos uno (ya que el pivote no se compara con si mismo). Descartando constantes, esto da una complejidad de $O(n)$. Finalmente, como resultado de todo lo mencionado, se llega a una complejidad total de $O(n \log n)$.

Caso promedio: Teniendo en cuenta un caso menos favorable donde la división del array alrededor del pivote no sea pareja, sino que el subarray derecho represente el 10% del tamaño original y el izquierdo el 90%, se puede establecer el siguiente árbol de recursividad:

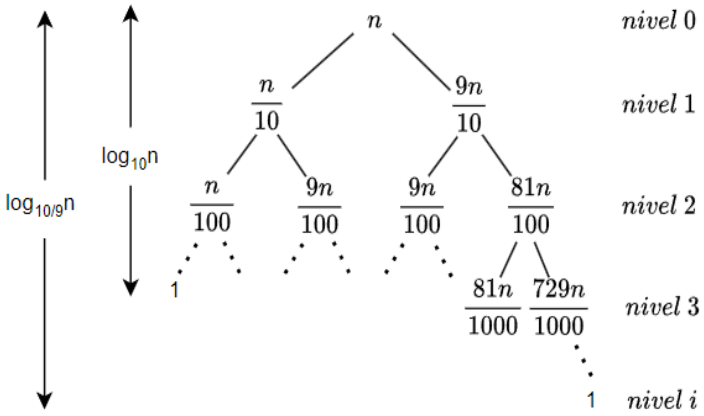


Figura 2 – Caso promedio de QuickSort. Adaptado de "Performance Comparison of Sequential Quick Sort and Parallel Quick Sort Algorithm" (p.16), Ishwari Singh Rajput et al, 2012, International journal of computer applications, 57(9)

En este caso cada rama izquierda se divide por 10 y cada rama derecha por $\frac{10}{9}$. Como la parte izquierda llegará al caso base 1 antes de la parte derecha, la altura del árbol está definida por:

$$\frac{n}{\frac{10^i}{9}} = 1$$

$$n = \frac{10^i}{9}$$

$$\log_{\frac{10}{9}} n = \log_{\frac{10}{9}}(\frac{10}{9})^i$$

$$\log_{\frac{10}{9}} n = i \text{ niveles}$$

Por otra parte, aunque las divisiones no sean iguales, cada nivel sigue teniendo una complejidad de $O(n)$ para determinar la posición del pivote. Por lo tanto, se puede decir que incluso en un caso menos favorable donde la elección del pivote no sea del todo oportuna, el algoritmo quicksort seguirá teniendo una complejidad de $O(n \log n)$.

Peor caso: El peor caso ocurre cuando array se divide de la forma más asimétrica posible, en donde el pivote siempre se ubica al final (o principio) del array (Rajput et al, 2012). En este escenario, el árbol recursivo queda de la siguiente forma:

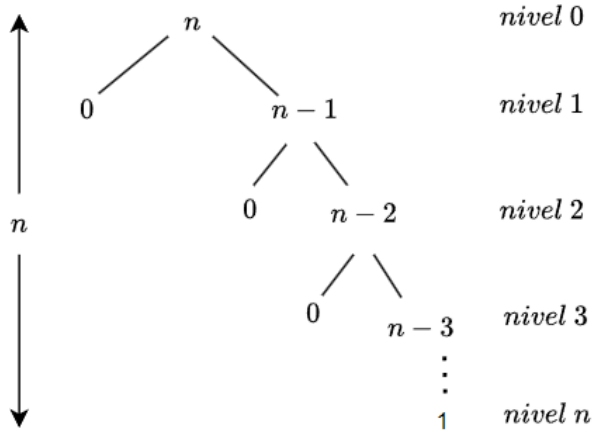


Figura 3 – Peor caso de QuickSort. Adaptado de "Performance Comparison of Sequential Quick Sort and Parallel Quick Sort Algorithm" (p.15), Ishwari Singh Rajput et al, 2012, International journal of computer application, 57(9)

Como todos los valores quedan a un lado del pivote, no se está realizando ninguna descomposición del problema. Para determinar la posición del pivote, en el nivel 0 se realizan $n-1$ comparaciones, en el nivel 1 se realizan $n-2$ y así sucesivamente hasta llegar al caso base 1 (Iliopoulos, 2013). De este modo se puede establecer la siguiente sucesión desde uno hasta $n-1$ elementos:

$$a_i = \{1, 2, 3, \dots, n-1\}$$

Encontrando la sumatoria se obtiene el número de comparaciones totales:

$$\sum_{i=1}^{n-1} a_i = \frac{n(a_1 + a_n)}{2}$$

$$\sum_{i=1}^{n-1} a_1 = \frac{(n-1)(1+(n-1))}{2}$$

$$\sum_{i=1}^{n-1} a_1 = \frac{(n-1)n}{2}$$

$$\sum_{i=1}^{n-1} a_1 = \frac{(n-1)n}{2}$$

Finalmente, eliminando las constantes se llega una complejidad de $O(n^2)$.

1.3 - Implementación en Java:

Para la implementación en Java se utilizó el código fuente obtenido en el sitio https://javachallengers.com/quicksort-algorithm-with-java/. El mismo resulta bastante intuitivo y de fácil comprensión en base a lo explicado en el apartado 1.1.

En este caso se utilizan tres métodos: el método principal `quicksort()` para realizar el ordenamiento recursivamente, un método `partition()`, para determinar la posición del pivote y ubicar los demás elementos a su izquierda y derecha, y adicionalmente el método `swap()` para realizar la permutación de elementos. También cabe destacar que en este algoritmo de ordenamiento todas las operaciones se realizan sobre el array original, por lo que no es necesario utilizar memoria adicional para creación de arrays auxiliares.

2. IMPLEMENTACIÓN CONCURRENTE

La implementación concurrente del algoritmo quicksort en Java se basa en que las divisiones del array se lleven a cabo mediante tareas que se ejecutan en paralelo, distribuyendo la carga del trabajo en los diferentes hilos disponibles.

El código utilizado es de elaboración propia y se puede encontrar en https://github.com/lucasojeda98/quicksort-analysis/blob/main/src/data/ParallelQuickSort.java

Para ello se utilizan las clases ForkJoinPool y RecursiveAction. La clase ForkJoinPool se utiliza para crear una pileta con los hilos disponibles en nuestro procesador o especificando cierta cantidad. Posteriormente cada hilo se encargará de ejecutar la tarea que le es asignada. Para definir una tarea, se crea la clase SortTask, que hereda de RecursiveAction. Esto permite utilizar el método `compute()`, en el cual se implementa una lógica similar al algoritmo secuencial, pero, en lugar de llamar recursivamente al método `quicksort()` para realizar el ordenamiento, se crean dos subtareas para el array izquierdo y derecho al pivote.

La ejecución se basa en crear una tarea (clase) principal cuyo constructor lleva como parámetros el array original y los índices *lowIndex* y *highIndex*, luego se crea una pileta de hilos y mediante el método `invoke()` se ejecuta la tarea. Dentro del método `compute()` de la tarea principal se crean y ejecutan dos nuevas tareas para los subarrays izquierdo y derecho, y así de forma sucesiva hasta que el array se encuentre ordenado.

Esto permite que las descomposiciones del array puedan ser realizadas en paralelo por distintos hilos, destacando que, como se mencionó anteriormente, las operaciones se realizan sobre el array original y no se necesita implementar ningún método adicional para unir los resultados de las tareas. Por último, de debe señalar que la determinación de la posición del pivote es realizada en forma secuencial dentro de cada tarea, por lo que en este caso se ha reutilizado el método `partition()` de la clase SecuentialQuicksort.

3. COMPARATIVA Y DESEMPEÑO

3.1 – Metodología:

Para realizar la comparativa de desempeño entre el algoritmo secuencial y el paralelo se desarrollaron casos de prueba para arrays aleatorios con números entre uno y mil, y con tamaños de cien, mil, diez mil, cien mil, quinientos mil, un millón, cinco millones y diez millones de elementos. Para cada tamaño de array se realizó un total de 10 ejecuciones, tanto para el caso secuencial como paralelo, y se obtuvo el promedio de tiempo de ejecución para cada uno, expresado en microsegundos(µs). Esta prueba fue realizada con un procesador Intel Pentium g7400 de 4 hilos, y al momento de crear la pileta de hilos se optó por establecer esa cantidad ya que no se notó ninguna mejora sustancial con un mayor número de hilos.

3.2 – Resultados:

Los resultados de los casos de prueba se muestran en la tabla 1. Los datos obtenidos arrojan que para tamaños de array relativamente pequeños el algoritmo secuencial resultó más rápido, mientras que, a medida que aumenta el número de elementos, el algoritmo paralelo fue incrementando su eficiencia en relación al secuencial.

Tamaño array	Promedio tiempo secuencial	Promedio tiempo paralelo
100	63,35 µs	903,81 µs
1.000	186,11 µs	1.301,16 µs
10.000	853,37 µs	3.298,54 µs
100.000	7.538,41 µs	9.015,6 µs
500.000	71.155,67 µs	36.856,08 µs
1.000.000	210.978,31 µs	104.934,09 µs
5.000.000	5.093.377,2 µs	1.472.169,49 µs
10.000.000	21.628.006,65 µs	5.694.386,48 µs

Tabla 1 – Resultados de tiempos de ejecución secuencial y paralelo para arrays de distintos tamaños

Puede observarse que, en un comienzo, los tiempos de ejecución para quicksort secuencial son menores y que a partir de cien mil elementos los resultados comienzan a igualarse, con una diferencia de tiempo reducida a favor del secuencial. Sin embargo, con quinientos mil elementos se comienza a notar una significativa ventaja de quicksort paralelo, siendo este un 93% más rápido. A partir de este punto, la ejecución concurrente resulta cada vez más eficiente en todos los casos: un 101% más rápido para un tamaño de un millón, un 246% para cinco millones y un 280% para diez millones. En la figura 4 se muestra una representación gráfica de la comparativa a partir de quinientos mil elementos, en donde se puede dimensionar la magnitud de la diferencia entre ambos algoritmos.

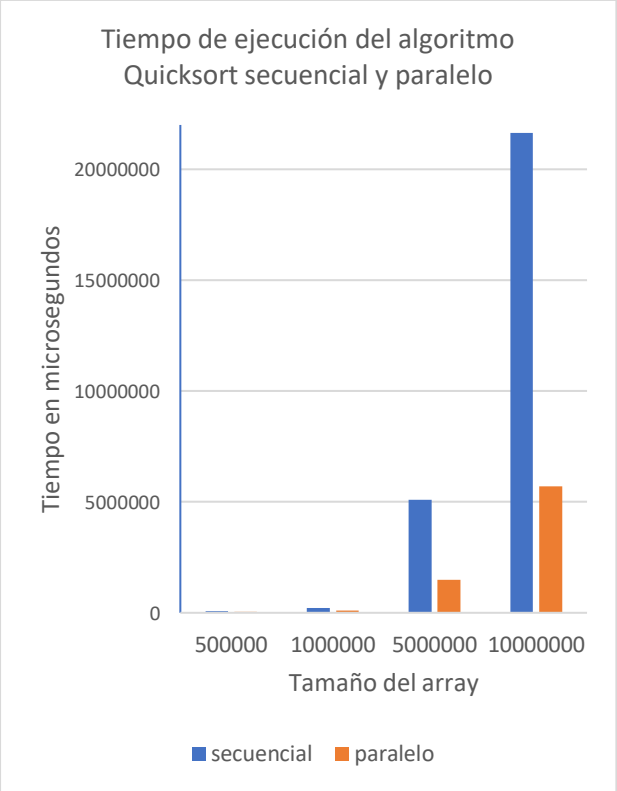


Figura 4 – Comparativa de tiempo de ejecución del algoritmo Quicksort secuencial y paralelo

Estos resultados obtenidos se pueden deber a que, para problemas pequeños, el tiempo que tarda el algoritmo secuencial en resolver el ordenamiento es menor al costo de producir los hilos y tareas del algoritmo paralelo, pero para problemas de gran tamaño la ejecución concurrente permite resolver el ordenamiento de manera más rápida.

3.3 – Comparativa con arrays ordenados

A continuación, se realizó una comparativa ejecutando los algoritmos secuencial y paralelo para un array ya ordenado previamente. Los resultados se muestran en la tabla 2, en donde se observa que, al igual que en el caso anterior, el algoritmo paralelo resulta más eficiente para arrays de mayor tamaño. Sin embargo, los tiempos de ejecución en general resultaron mayores tanto para el caso secuencial como paralelo.

Tamaño array	Promedio tiempo secuencial	Promedio tiempo paralelo
100	49,5 µs	944,18 µs
1.000	534,45 µs	1.802,98 µs
10.000	5.021,6 µs	8.296,05 µs
100.000	41.601,66 µs	42.010,56 µs
500.000	204.166,64 µs	153.475,86 µs
1.000.000	626.418,65 µs	356.336,22 µs
5.000.000	5.549.033,24 µs	2.572.170,78 µs
10.000.000	24.026.743,59 µs	7.205.551,02 µs

Tabla 2 - Resultados de tiempos de ejecución para arrays ordenados previamente

Además, si bien la diferencia a favor del algoritmo paralelo sigue siendo significativa, es un tanto menor: es del 33% para quinientos mil elementos, 76% para un millón, 116% para cinco millones y 233% para 10 millones. Los resultados obtenidos se pueden deber a que un array ordenado es la representación del peor caso en términos de complejidad (explicado en el apartado 1.2), siendo la misma de $O(n^2)$, ya que el pivote siempre será el último elemento del array y en realidad el problema no se estará descomponiendo en partes.

4-CONCLUSIÓN

Como conclusión, el algoritmo quicksort, el cual destaca como un método de ordenamiento eficiente a la hora de ordenar grandes volúmenes de datos, ha demostrado una mejora significativa de tiempos de ejecución en su implementación concurrente para arrays de gran tamaño. La investigación confirma que la concurrencia permite utilizar mejor los recursos de sistemas multinúcleo, mejorando considerablemente su rendimiento en comparación con la versión secuencial. Estos resultados subrayan la importancia de adaptar algoritmos secuenciales a sistemas modernos para maximizar su eficiencia y rendimiento.

REFERENCIAS

Rajput, I. S., Kumar, B., & Singh, T. (2012). Performance comparison of sequential quick sort and parallel quick sort algorithms. International Journal of Computer Applications, 57(9), 14-22.

Iliopoulos, V. (2015). The Quicksort algorithm and related topics. arXiv preprint arXiv:1503.02504.

Del Nero, Rafael (20 de marzo de 2023). Quicksort Algorithm with Java. javachallengers.com. https://javachallengers.com/quicksort-algorithm-with-java/

Jain, Ayush (24 de noviembre de 2021) How to solve time complexity Recurrence Relations using Recursion Tree method?. geeksforgeeks.org. https://www.geeksforgeeks.org/how-to-solve-time-complexity-recurrence-relations-using-recursion-tree-method/