QUICKSORT SECUENCIAL VS QUICKSORT CONCURRENTE

Datos del autor: Ángel Valentín Altieri
Repositorio con el trabajo: https://github.com/angelvalentinn/TP-programacion-concurrente/tree/main
Video explicativo (<10min): https://www.youtube.com/watch?v=o5arbp82bTo
E.mail: altieriangel1@gmail.com

RESUMEN (ABSTRACT)

El objetivo de este trabajo es comparar el algoritmo de ordenamiento QuickSort en su versión secuencial y concurrente, demostrando las diferencias entre sí y analizando si hay mejoras en el tiempo de ejecución al aplicar concurrencia. Los resultados destacan la importancia de elegir un enfoque adecuado según el volumen de datos: para arreglos de menor tamaño (<1M), el secuencial es más rápido, mientras que el concurrente escala mejor con grandes volúmenes (hasta 5% más rápido en 100M elementos). Los algoritmos utilizados para la comparación están implementados en Java y utilizan el primer elemento como pivote. Ambos no son de elaboración personal y el link está citado en las referencias.

Keywords: Concurrencia, QuickSort, Secuencial, Algoritmo, Paralelismo

1. INTRODUCCIÓN

El algoritmo QuickSort es un algoritmo de ordenamiento basado en el paradigma "divide y vencerás". Este algoritmo consiste en elegir un pivote y ubicar los elementos menores al pivote a su izquierda y los elementos mayores a su derecha. De esta manera se crea dos sub arreglos alrededor del pivote, donde se aplicará recursivamente el mismo proceso hasta que queden sub arreglos de un elemento.

La elección del pivote resulta importante en este algoritmo, el mejor elemento sería el promedio de todos los elementos, pero encontrarlo es costoso. Entonces se deben tomar estas posibles opciones:

- Primer elemento
- Ultimo elemento
- Elemento aleatorio

Sin embargo, estas alternativas resultan ineficientes en arreglos ordenados parcialmente. Nicklaus Wirth propuso superar este problema eligiendo como pivote al valor medio entre tres elementos: el primero, el medio y el último.

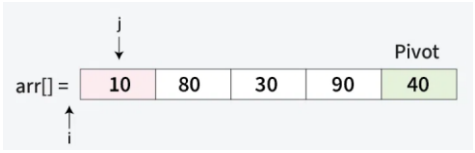
Ejemplo QuickSort

El link del código del algoritmo tanto secuencial como concurrente se encuentran en las referencias, dicho código no es de elaboración propia y fue adaptado.

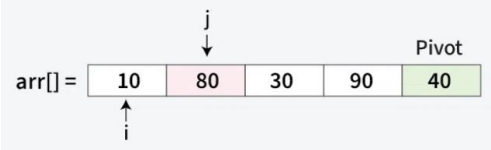
Se elige como pivote al último elemento del arreglo.

Dado arr[] = {10,80,30,90,40}

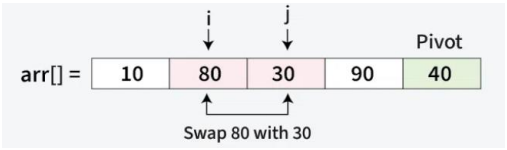
1. Se compara el número 10 con el pivote, es decir, 40. Al ser menor que el pivote se coloca a su izquierda y se incrementa en uno i, es decir, el último menor que el pivote, que ahora es 10.



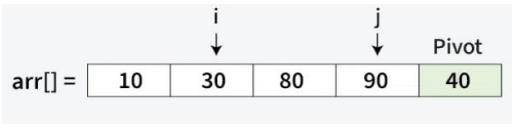
2. Se compara el número 80 con el pivote 40, al ser mayor que el pivote no se hace nada.



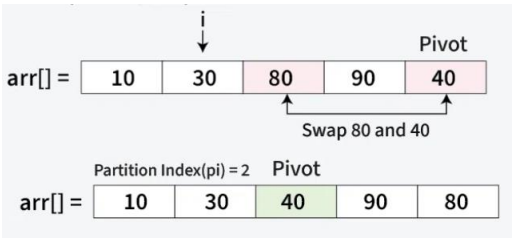
3. Se compara el número 30 con el pivote 40, al ser menor que el pivote se coloca a la izquierda como último menor.



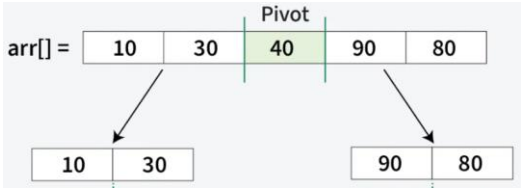
4. Se compara el número 90 con el pivote 40, al ser mayor que el pivote no se hace nada.



5. Una vez recorrido todo el arreglo se cambia el pivote por el elemento siguiente del último menor que el pivote.



6. El algoritmo aplica el mismo proceso de manera recursiva con los sub arreglos derivados.



Complejidad algorítmica

Mejor caso O(n.log.n)

La división inicial nos dará dos sub arreglos de igual tamaño, por lo tanto, la primera iteración del arreglo completo de tamaño n necesita O(n). La ordenación de los dos sub arreglos restantes con n/2 elementos toma la forma 2 * O(n / 2) para cada uno, por tanto, la complejidad es O(n.log.n).

Peor caso $O(n^2)$

El arreglo inicial está ordenado parcialmente y nos dará dos sub arreglos de distintos tamaños. Este caso se puede mitigar seleccionando como pivote un valor medio entre tres o un valor aleatorio.

Caso promedio O(n.log.n)

En promedio QuickSort es muy eficiente con una complejidad de O(n.log.n), lo que lo hace ideal para grandes conjunto de datos.

2. IMPLEMENTACIÓN CONCURRENTE

El algoritmo QuickSort al estar basado en el principio de dividir puede implementarse de forma concurrente. Esta implementación mejora considerablemente su complejidad de O(n.log.n) hasta O(log.n).

De esta forma, las llamadas recursivas pueden ejecutarse concurrentemente y si el hardware y el sistema operativo lo permiten, paralelamente. La velocidad de ejecución dependerá del tamaño del arreglo y de la cantidad de núcleos lógicos del procesador.

En el algoritmo QuickSort, la cantidad de llamadas recursivas es impredecible, ya que depende de la elección del pivote y de los datos de entrada. Si usáramos la clase Thread, en el peor de los casos podrían generarse hasta O(n) hilos, colapsando el sistema.

Para solucionar este problema se utiliza la clase ForkJoin que es una herramienta poderosa para el procesamiento de tareas recursivas que permite limitar la creación de hilos al número de núcleos del procesador y redistribuye las tareas recursivas entre los hilos existentes automáticamente.

El link del código del algoritmo tanto secuencial como concurrente se encuentran en las referencias, dicho código no es de elaboración propia y fue adaptado.

Algoritmo concurrente

```
//Metodo que ejecuta ForkJoinPool
@Override
protected void compute() {

  if(left < right){ //Condicion de corte de recursividad

    //Elige el pivote y acomoda los mas chicos a su izquierda y mas grandes a su derecha
    int pivot = partition(data, left, right);

    //Metodo que ejecuta ambas tareas en concurrente y en mejor de los casos paralelo
    invokeAll(new Quicksort_concurrente(data,left, pivot), //Se repite el algoritmo con sub array izq
              new Quicksort_concurrente(data, pivot + 1, right)); //Se repite el algoritmo con sub array izq

  }

}
```

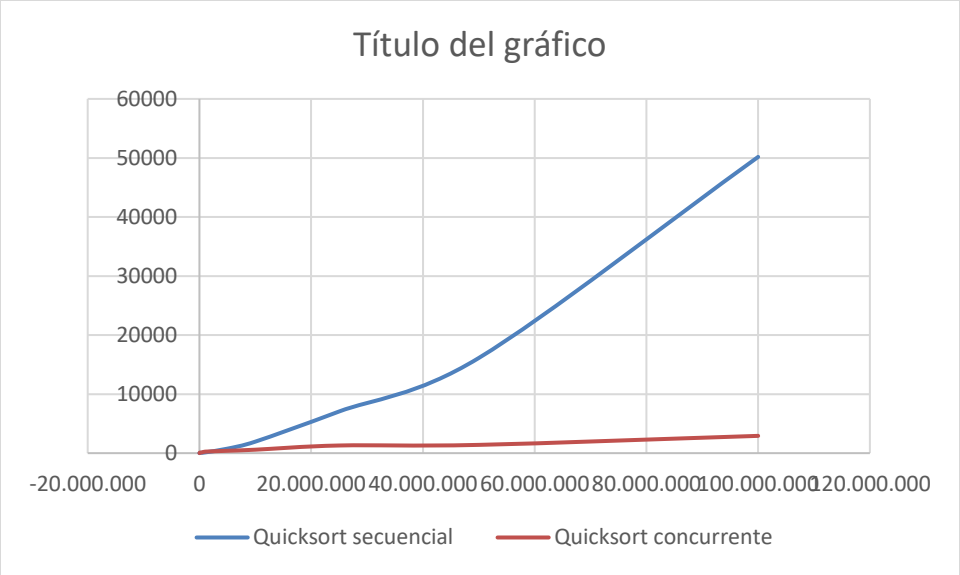
Compute es un método que se encarga de ejecutar las tareas concurrentes, dentro de este está el algoritmo QuickSort, que en una etapa inicial verifica la condición de corte de recursividad, luego se elige el pivote y el arreglo inicial es particionado en dos sub arreglos. En dichos sub arreglos se aplicará nuevamente el algoritmo, pero a diferencia del algoritmo secuencial, se aplicarán dentro del método invokeAll que hará que ambas tareas se ejecuten en concurrente.

3. COMPARATIVA Y DESEMPEÑO

El desempeño del QuickSort secuencial y concurrente se ilustra en la siguiente tabla comparativa. Dicha tabla permite visualizar el tiempo que tarda cada algoritmo en ordenar un mismo arreglo aleatorio con diferentes tamaños.

El objetivo de esta comparativa es ver cómo se comporta cada algoritmo cuando el tamaño del arreglo tiende a infinito. El tiempo de la tabla comparativa está dada en milisegundos. La PC donde realice la comparativa tiene un procesador de 8 núcleos lógicos.

Cantidad de elementos	**1.000**	**10.000**	**100.000**	**1.000.000**	**5.000.000**	**10.000.000**	**50.000.000**	**100.000.000**
Quicksort secuencial	1ms	7ms	34ms	122ms	757ms	1947ms	16811ms	50153ms
Quicksort concurrente	11ms	17ms	96ms	253ms	396ms	579ms	1475ms	2856ms



4. CONCLUSIÓN

En la tabla comparativa anterior notamos que el algoritmo secuencial resulta más eficiente que el algoritmo concurrente cuando se trata de arreglos con una cantidad de elementos menor a 1 millón. Esto se debe a que la gestión de hilos en la versión concurrente es costosa y supera el beneficio de la concurrencia y paralelismo.

Cuando el tamaño del arreglo ronda los 5 millones, se empiezan a notar los beneficios de la concurrencia y paralelismo. También podemos ver que a partir de los 5 millones conforme el tamaño del arreglo va tendiendo al infinito, la eficiencia es aún más notoria.

También es posible notar en el gráfico cómo mejora la complejidad algorítmica en la versión concurrente, pasando de ser O(n.log.n) a O(log.n)

En conclusión, podemos decir que los dos algoritmos son eficientes. El algoritmo QuickSort secuencial es más eficiente cuando se trata de un arreglo de tamaño menor a 1 millón y el algoritmo QuickSort concurrente es más eficiente cuando se trata de un tamaño mayor a 5 millones.

La elección entre ambos algoritmos debe considerar el tamaño de los datos y la cantidad de núcleos lógicos disponibles.

REFERENCIAS

KeepCoding. (s.f.). *¿Qué es QuickSort y cómo funciona?* KeepCoding. https://keepcoding.io/blog/que-es-quicksort-y-como-funciona-programacion/

Numerentur. (s.f.). *Quicksort*. https://numerentur.org/quicksort/

GeeksforGeeks. (s.f.). *Quick Sort algorithm*. https://www.geeksforgeeks.org/quick-sort-algorithm/

Código fuente adaptado:

Techie Delight. (s.f.). *Quicksort secuencial*. https://www.techiedelight.com/es/quicksort/

Rohana, E. (2019). *Parallel QuickSort in Java*. GitHub Gist. https://gist.github.com/EliaRohana/25b924048d990c5358313d18daf8f491