QUICKSORT SECUENCIAL Y CONCURRENTE

Datos del autor: FERREIRO, Ramiro Xavier

Repositorio con el trabajo: RamiroFerreiro/Trabajo-Practico-Programacion-Concurrente: Algoritmo QuickSort (github.com)

Video explicativo: Trabajo práctico de Programación Concurrente (youtube.com)

E.mail: ramiro.x.ferreiro@gmail.com

RESUMEN

La intención de este trabajo es la de mostrar: las diferencias principales que existen entre el algoritmo QuickSort secuencial y el algoritmo QuickSort concurrente; las benevolencias de la concurrencia y los resultados de su aplicación en algoritmos recursivos. El algoritmo utilizado para comparar la secuencialidad y la concurrencia es el ya mencionado QuickSort con pivote aleatorio, ambos no son de autoría propia y dejaré el link de ambas paginas en las referencias de este documento. La elección del algoritmo estuvo determinada por la velocidad que desempeña en su formato secuencial en comparación con otros y con la intención de mejorarla trabajando con hilos.

Keywords: QuickSort, Concurrente, Secuencial, Algoritmo.

1. INTRODUCCIÓN

El algoritmo QuickSort es un algoritmo de ordenación que se basa en la selección de un pivote y la partición del arreglo de elementos a ordenar.

El proceso consiste en realizar una partición y ubicar al pivote en la posición correcta del arreglo. Para esto previamente se deberán colocar a la izquierda del pivote todos los elementos más pequeños y a la derecha todos los elementos mayores al mismo.

La partición se realiza de manera recursiva hacia los lados una vez ubicado el primer pivote. Finaliza el algoritmo cuando los subarreglos ya no se pueden ordenar porque estos son de tamaño unitario.

Elección del pivote.

Existen diferentes formas de selección el pivote para realizar las comparaciones que determinar su posición.

- Ultimo elemento como pivote:

Se elije el último elemento del arreglo como pivote. Explicado con el ejemplo que se encuentra a continuación.

- Primer elemento como pivote:

Se elije el primer elemento como pivote. Algoritmo muy similar al ejemplo que se encuentra a continuación. Se diferencia con el anterior en que el arreglo se recorre de final a principio decrementando su índice.

- Elemento aleatorio como pivote:

Se elije el último elemento como pivote. Previamente este fue intercambiado por un valor aleatorio del arreglo.

- Mediana de tres como pivote:

Se elijen tres valores del arreglo, por ejemplo, el primero, el ultimo y el centro. Se compara y se elige el valor medio como pivote.

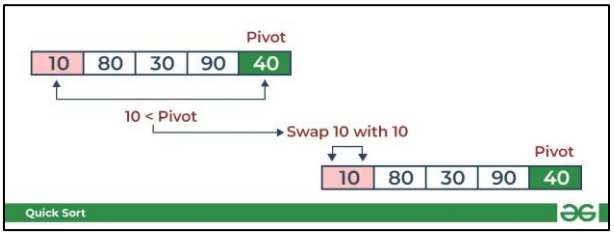
Ejemplo QuickSort.

Se elige en todo momento al último elemento del arreglo como pivote.

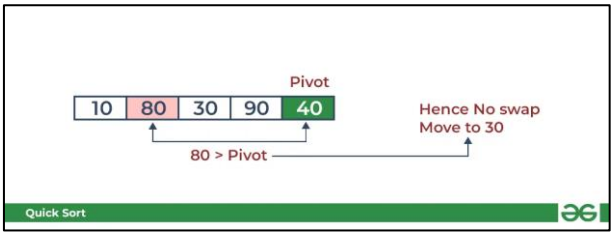
Considere: arr[] = {10, 80, 30, 90, 40}.

1. Se compara el numero 10 con el pivote número 40. Como es menor que el pivote se

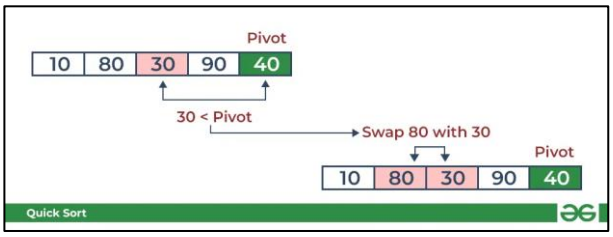
coloca a la izquierda. Se toma al 10 como el ultimo menor.



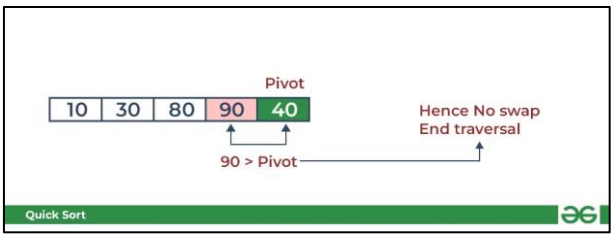
2. Se compara el número 80 con el pivote número 40. Como es mayor que el pivote no se realizan intercambios.



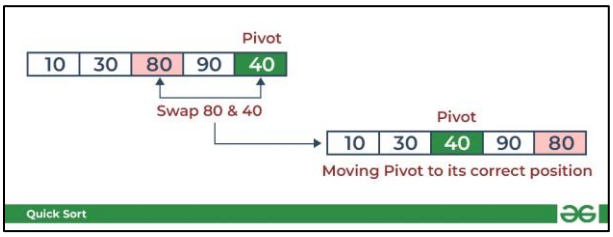
3. Se compara el número 30 con el pivote número 40. Como es menor se coloca a la izquierda como último menor.



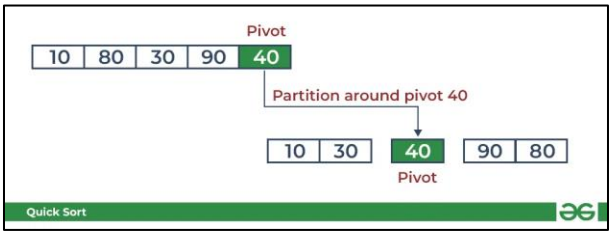
4. Se compara el número 90 con el pivote número 40. Como es mayor que el pivote no se realizan intercambios.

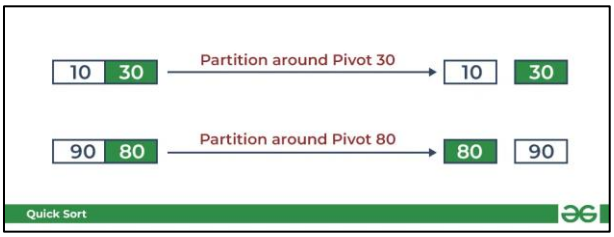


5. Una vez recorrido todo el arreglo. Se coloca el pivote en la posición siguiente al último menor. En este caso a la derecha del 80. Aquí finaliza la primera ubicación del pivote.



6. El algoritmo sigue adelante de manera recursiva particionando el arreglo principal en subarreglos hasta que no se pueda seguir particionando.





Complejidad de QuickSort.

Big O define el tiempo de ejecución necesario para ejecutar un algoritmo identificando como cambiará el rendimiento de su algoritmo a medida que crece el tamaño de la entrada.

La notación Big O mide la eficiencia y el rendimiento del algoritmo usando la complejidad del tiempo y el espacio.

La complejidad temporal de un algoritmo especifica cuanto tiempo llevará ejecutar un algoritmo en función de su tamaño de entrada. De manera similar, la complejidad espacial de un algoritmo especifica la cantidad total de espacio o memoria necesaria para ejecutar un algoritmo en función del tamaño de la entrada.

- Mejor caso: O(n . log n).

  Ocurre cuando el pivote elegido divide al arreglo en mitades aproximadamente iguales.

- Peor caso: O(n^2)

  Ocurre cuando el arreglo está ordenado o parcialmente ordenado y el pivote elegido en cada paso da como resultado particiones muy desequilibradas. Es altamente improbable que el algoritmo tenga una complejidad n^2 si se toman medidas al respecto. La elección de un pivote aleatorio o media de tres y mezclarlo es una buena opción para mitigar el peor caso.

- Caso promedio: O(n . log n).

  El rendimiento promedio de casos de Quicksort suele ser muy bueno rondando la complejidad n . log n.

2. IMPLEMENTACIÓN CONCURRENTE

El algoritmo de ordenamiento QuickSort basado el "divide y vencerás" puede ejecutar su recursividad de manera concurrente. Esto mejora su complejidad algorítmica de O(n . log n) hasta O(log n) lo cual reduce considerablemente el tiempo empleado para ordenar.

Las llamadas recursivas de los subarreglos, al trabajarlas con hilos pueden ejecutarse a la vez. La velocidad de ejecución depende de la cantidad de elementos a ordenar y la cantidad de núcleos lógicos que tenga el procesador, cuantos más tenga, más rápido será.

Concurrencia y recursividad.

Existe un problema clave a la hora de aplicar concurrencia a un método recursivo. Es imposible determinar cuántos hilos se van a utilizar porque la recursividad se aplica en cantidades aleatorias. Dependiendo de cómo se distribuyan las mitades, la condición de corte en el QuickSort actuará antes o después.

Para solucionar este problema se utiliza la clase ForkJoin que nos permite generar máxima cantidad de hilos que soporta el procesador y esos hilos se van a asignar a las tareas que surgan de la recursividad.

ForkJoin.

La implementación de Fork/Join toma forma a través de las clases ForkJoinTask y ForkJoinPool. La clase ForkJoinPool invoca a una tarea de tipo ForkJoinTask pero, para poder invocar múltiples subtareas en paralelo de forma recursiva, invocará a una tarea de tipo RecursiveAction, que extiende de ForkJoinTask.

La clase RecursiveAction contiene el método compute(), que será el encargado de ejecutar nuestra tarea paralelizable. Para paralelizar una tarea secuencial, por tanto, crearemos una clase que extienda de RecursiveAction, y sobrecargaremos el método compute() con dicha tarea. Además, existe una diferencia entre el código de la tarea secuencial y el código que contenga el método compute(); mientras la tarea secuencial consiste en un método que se llame a sí mismo de forma recursiva, el método compute() creará de forma recursiva nuevas instancias de su misma clase, correspondiendo cada nueva instancia a una subtarea distinta.

Aplicación de la concurrencia.

Vista desde el main.

```
//array que sera ordenado
int[] A = generarArrayAleatorio(75000000, 1, 90000);

//Se crea la tarea que va a ordenar al vector, todavía no la hace
QuicksortForkJoinRandomized quicksortTaskRandom = new QuicksortForkJoinRandomized(A);

//Creo la pileta de hilos que genera muchos hilos, tantos como pueda
ForkJoinPool pool = new ForkJoinPool();

//Desde la pileta de hilos invocamos a la tarea
pool.invoke(quicksortTaskRandom);
```

1. Se genera el arreglo de aleatorios
2. Se crea la tarea que va a ordenar el vector
3. Se genera la pileta de hilos
4. Se invoca la tarea

Vista desde la clase.

```
@Override
protected void compute() {
    ///condicional de corte de recursividad. Identico al secuencial.
    if(inicio < fin){

        //Se obtiene la posicion del pivote para utilizar como indice
        int q = partition(array, inicio, fin);

        //Se crean nuevas tareas para ordenar las particiones y las invoco de
        //manera que se ejeucuten concurrentemente.
        invokeAll(new QuicksortForkJoinRandomized(array, inicio, q - 1),//izq
                  new QuicksortForkJoinRandomized(array, q + 1, fin)); //der
    }
}
```

Se ejecuta el método compute() donde se sitúa el algoritmo de ordenamiento. Al igual que método secuencial la recursividad tiene condición de corte y el método partición nos devuelve la posición del pivote.

A diferencia del ordenamiento secuencial, no se envían los subarreglos llamando al mismo método. Lo que se hace es crear nuevas tareas con los subarreglos determinados por el pivote y se invocan las tareas creadas para se ejecuten de manera concurrente. En este caso son dos tareas pero podrían ser más. La función invokeAll() es quien internamente asigna los hilos a las tareas pasadas por parámetros.

A continuación, una ilustración para realizar la comparativa entre la ejecución del algoritmo secuencial y el concurrente.

QuickSort secuencial.

```
public static void sort(int arreglo[], int bajo, int alto) {
    //Condicion de corte de la recursividad. Mientras los arregl

    if (bajo < alto) {
        //pi es la posicion del pivote, la cual ya se encuentra
        int pi = partition(arreglo, bajo, alto);

        //se vuelve a ordenar de manera recursiva las partes div
        sort(arreglo, bajo, pi-1);
        sort(arreglo, pi+1, alto);
    }
}
```

Aclaraciones.

Tanto el código secuencial como el concurrente no es de autoría propia y fueron modificados levemente para la exposición. En la sección de referencias se adjuntan los links de ambos códigos en formato original.

3. COMPARATIVA Y DESEMPEÑO

Para demostrar el desempeño de los algoritmos generé dos arreglos de números aleatorios idénticos, a los cuales les aplico una cantidad de números cada vez más grande para ver su comportamiento cuando estos tienden a infinito.

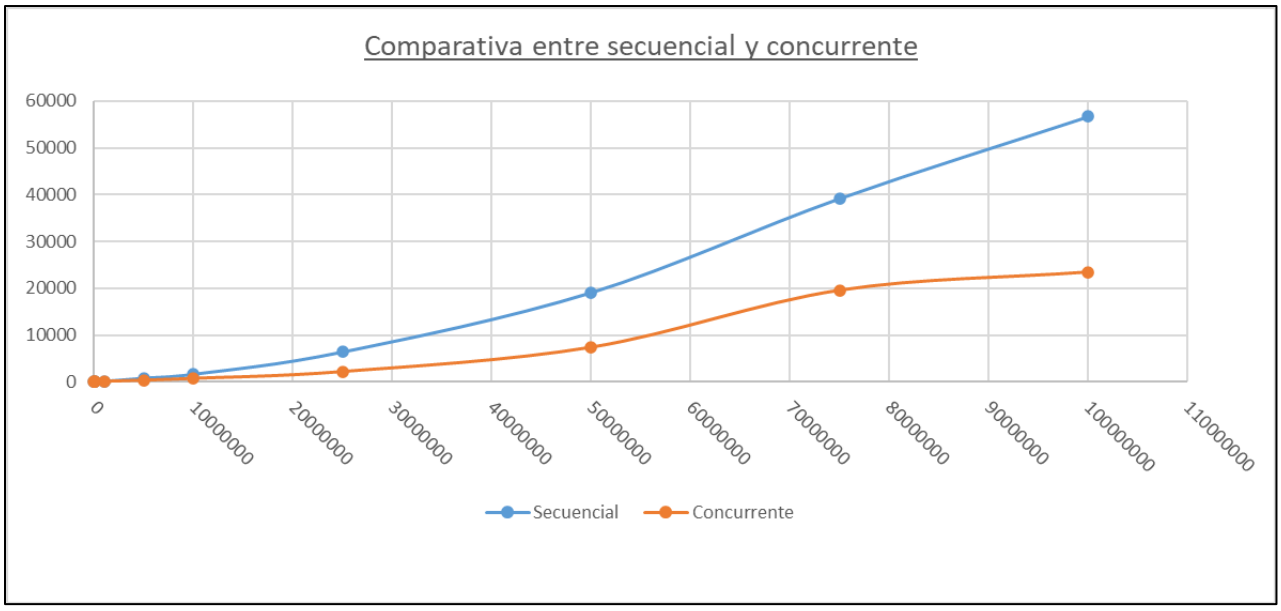
Utilicé como medida de comparación el tiempo de ejecución dado en milisegundos por la función "System.currentTimeMillis()".

El color verde indica cual algoritmo fue más rápido para la cantidad de elementos dada y el rojo cual fue más lento.

El gráfico en el eje X representa a la cantidad de elemento testeados y en el eje Y representa el tiempo tardado en milisegundos para ordenar.

El procesador tiene 8 núcleos lógicos. Por ende, la pileta de hilos va a generar 8 hilos.

Cantidad de elementos	1000	10000	100000	1000000	5000000	10000000	25000000	50000000	75000000	100000000
Algoritmo secuencial	1	7	35	172	851	1686	6430	19108	39200	56681
Algoritmo concurrente	6	23	72	190	443	827	2276	7450	19591	23457



4. CONCLUSIÓN

Después de haber realizado las comparativas podemos analizar sus resultados, concluyendo que:

- *Si cantidad de elementos a ordenar es poca, el algoritmo secuencial es más eficiente*. Es decir, si es aproximadamente menor a un millón, no va haber beneficios significativos al aplicar concurrencia y hasta puede ser menos eficiente como en la prueba que hicimos. Esto se debe que el esfuerzo de la generación de hilos conlleva demasiado tiempo y el algoritmo secuencial ordena más rápido.
- *Cuando la cantidad de elementos tiende a infinito el algoritmo concurrente es más eficiente.* Por ende, una vez superado el punto de equilibrio, en donde para la misma cantidad de elementos tardan lo mismo, el algoritmo secuencial siempre va a ser más eficiente.
- *Ambos algoritmos son eficientes.* Para mil elementos el secuencial es 5 veces más rápido. Para 100 millones el concurrente es casi 3 veces más rápido. Para 1 millón igualan en tiempo de ejecución. La aplicación de un algoritmo u otro depende la necesidad del software y queda consideración de grupo de desarrollo cual solución es mejor en cada caso.
- *El punto de equilibrio es variable.* Está determinado por el hardware de la computadora. Si hay más hilos para ejecutar, será necesario una menor cantidad de elementos para que la concurrencia sea más eficiente que la secuencialidad.
- *La concurrencia altera la complejidad Big O.* El grafico muestra casi a la perfección como el algoritmo secuencial tiene complejidad de O(n . log n) y el algoritmo concurrente una de O(log n) debido al trabajo con hilos.

REFERENCIAS

Anonimo (7 de junio 2024, ultima actualización). *QuickSort – Data Structure and Algorithm Tutorials*. QuickSort - Data Structure and Algorithm Tutorials - GeeksforGeeks

Anonimo (14 de Septiembre de 2023, ultima actualización). *QuickSort usando Pivoting Aleatorio*. Quicksort paralelo em Java utilizando o modelo Fork-Join (blogcyberini.com)

Felipe, Henrique (5 de Septiembre de 2018). *Quicksort paralelo em Java utilizando o modelo Fork-Join.*

Natalia Roales González (21 de agosto de 2015). *El paralelismo en Java y el framework Fork/Join*. El paralelismo en Java y el framework Fork/Join - Adictos al trabajo

Joel Olawanle (1 de Febrero de 2024). *Guía: Notación Big O - Gráfico de complejidad de tiempo.* Guía: Notación Big O - Gráfico de complejidad de tiempo (freecodecamp.org)

LINKS CODIGOS

Link código del algoritmo secuencial: Quicksort paralelo em Java utilizando o modelo Fork-Join (blogcyberini.com)

Link código del algoritmo concurrente: QuickSort using Random Pivoting - GeeksforGeeks