El algoritmo de ordenación por mezcla

¿En qué medida las técnicas de concurrencia optimizan el algoritmo de ordenación *Merge Sort*?

Monografia de Informática NS

Cómputo de palabras: xxxx

Código del alumno: lqv708

Índice

1. Introducción

La finalidad de esta investigación evaluar holísticamente cuatro implementaciones del algoritmo de ordenación Merge Sort en términos de complejidad temporal y espacial, en el peor caso; tiempos de ejecución y memoria usados; y, la naturaleza específica de cada implementación. Concretamente, se comparan el Merge Sort Iterativo Serial (MSIS), el Merge Sort Recursivo Serial (MSRS), el Merge Sort Recursivo Paralelo (MSRP), y el Merge Sort Iterativo Paralelo (MSIP).

1.1. Algoritmos de ordenación

En el día a día usamos estructuras de datos ordenadas, como la lista de contactos del teléfono, en los almacenes para la gestión de inventario, en los resultados de una búsqueda en internet, etc. El proceso de colocar los datos en un cierto orden se llama ordenación.[1] La ordenación es una operación común en los sistemas informáticos y ha sido ampliamente estudiada.[2] Aun así, no existe un algoritmo de ordenación perfecto. Actualmente, se siguen desarrollando, y además estos relacionan una gran variedad de conceptos de informática. Por ende, este estudio resulta relevante. En esta investigación, se aplican los paradigmas de la concurrencia, el paralelismo, la ejecución serial, la iteración y la recursión al algoritmo *Merge Sort*.

1.2. Merge Sort

El ordenamiento por mezcla (*Merge Sort*) es un algoritmo de ordenación inspirado en la técnica divide y vencerás (*divide-and-conquer*). Es capaz de ordenar un conjunto de datos a través de los siguientes pasos:

1. dividir la colección en dos mitades,
2. dividir las sub-colecciones en más mitades hasta que contengan cero o un elemento,
3. ordenar cada sub-colección,
4. unir (*merge*) todas las sub-colecciones de forma ordenada, y, finalmente
5. la colección queda ordenada.[3]

[1] (Knuth, 1997) [2] (McMillan, 2007) [3] (Skiena, 2008)

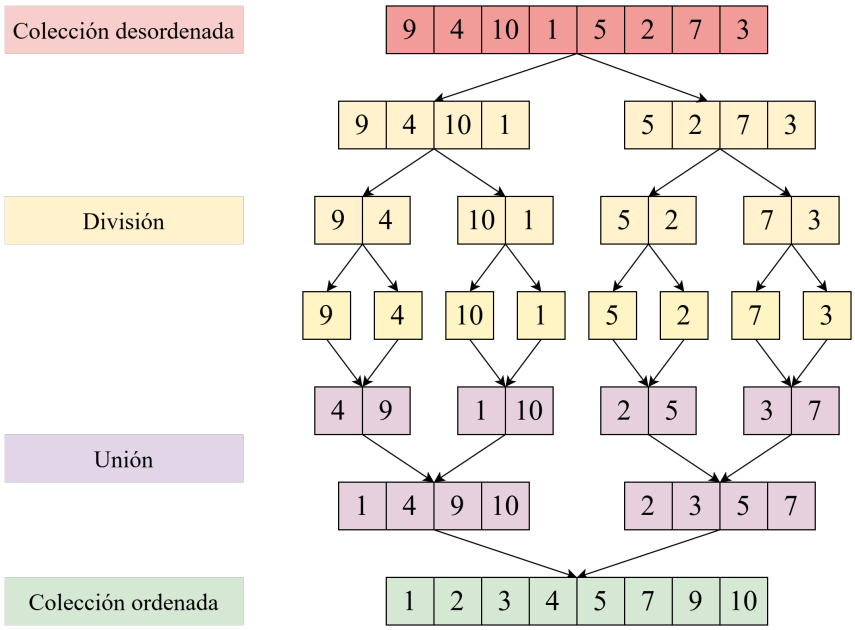

Figura 1: Diagrama de árbol de *Merge Sort*

1.3. Concurrencia

La concurrencia se refiere a la habilidad de un sistema para ejecutar múltiples tareas al mismo tiempo. [4].

1.4. Ejecución serial y paralela

Un **programa informático** es un conjunto de instrucciones que un sistema informático ejecuta. A su vez, un programa se divide en partes más pequeñas e independientes que llamamos tareas. Cuando las tareas se ejecutan una tras otra durante períodos de tiempo no superpuestos, hablamos de **ejecución serial**. A veces, la ejecución de una tarea depende del resultado de la tarea anterior. En dicho caso, la tarea bloquea a la siguiente. Esto es la **computación secuencial**. En contraposición, la **ejecución paralela** consiste en ejecutar varias tareas simultáneamente. Sin embargo, el paralelismo real solo es posible si el sistema consta de más de una unidad de procesamiento y si las tareas del algoritmo son independientes. [5]

La idea es recorrer las dos partes izquierda y derecha del árbol del *Merge Sort* (Figura 1) paralelamente para aumentar la eficiencia del algoritmo.

[4] (Magee y Kramer, 2006)

[5] (Bobrov, 2023)

Ejecución serial

Tiempo	1	2	3	4	5	6
Tarea 1						
Tarea 2						
Tarea 3						

Ejecución parelela

Tiempo	1	2	3	4	5	6
Tarea 1						
Tarea 2						
Tarea 3						

Figura 2: Comparación de los diagramas de Gantt (//Nota: Probablemente cambiar por dos árboles mostrando la secuencia de instrucciones)

1.5. Iteración y recursividad

El paradigma de la **programación estructurada** considera que todo programa informático está formado por las estructuras de control de Secuencia, Selección y Repetición.[6] La recursión es una estructura de repetición.[7] Si un programa incorpora una estructura recursiva, quiere decir que hay una función que se llama a sí misma. [8] Toda función recursiva contiene caso recursivo, estructura condicional que llama a la propia función, y un caso base, que retorna un valor constante o finaliza la función.

[

Nota: añadir gráfico para mostrar la recursión??]

1.6. Complejidad

Esta investigación evalúa la aplicación de todos estos conceptos al algoritmo de ordenación por mezcla. La eficiencia de un algoritmo de ordenación usualmente se cuantifica en términos de complejidad temporal y espacial. La complejidad temporal es una medida de la variación del tiempo de ejecución de un algoritmo a medida que el tamaño de la entrada crece. El tiempo de ejecución se ve afectado por variables como el soporte físico y lógico (CPU, RAM, lenguaje de programación, SO...)[9]Es por ello, que se estudia el comportamiento asintótico de la complejidad, es decir, cuando el tamaño de la entrada n tiende al infinito. Lo hace mediante las funciones del: peor caso posible $O(n)$, el caso promedio $\theta(n)$, y el mejor caso posible $\Omega(n)$. [8] De ahí la notación *Big-O*, que permite comparar algoritmos a través de distintas plataformas y predecir el tiempo de ejecución de un algoritmo.

[6] ([ELI] y [NOVA], s.f.) [7] (College, 2000) [8] (Bhargava, 2016) [9] (Heineman, Pollice, y Selkow, 2008)

La complejidad temporal se mide usualmente mediante la función del peor caso posible por varios motivos: es más sencillo de analizar al no ser necesario conocer la distribución de datos de entrada, garantiza el buen funcionamiento del programa en el que se incorpore en caso de situaciones desfavorables... Caracterizar un algoritmo mediante su complejidad $O(n)$ es una abstracción, ya que se fundamenta en el modelo de Máquina de Acceso Aleatorio, que considera que las *operaciones básicas*, como las aritméticas, lógicas o los accesos a la memoria, duran una sola unidad de tiempo. Asimismo, se considera que cada línea de código es una *instrucción básica* que ejecuta un número constante de operaciones básicas. Los bucles (`for`, `for`, `while`... y llamadas a funciones se evalúan sumando sus instrucciones básicas.

La notación *Big-O* proporciona un límite superior para el cual una función nunca lo sobrepasará. Esta notación estudia el orden de crecimiento de una función. Formalmente, se define como: dadas dos funciones $f(n)$, $g(n)$, entonces $f(n) = O(g(n))$ siempre que existan las constantes c y n_0 tal que $f(n) \leq c \cdot g(n)$, para todo $n \geq n_0$. Esto significa que para una función $f(n)$ solo existirá un Big-O $O(g(n))$ si todos los valores de su entrada n son inferiores al producto entre una constante c y una función $g(n)$, que es el límite superior. De forma que, $f(n)$ nunca crecerá más que $g(n)$.

Por ejemplo, se da un algoritmo que toma un tiempo de ejecución $f(n) = 4n + 7$ y queremos saber si se comporta de forma lineal. Esto es $g(n) = n$ Entonces para que $f(n) = O(g(n))$ hay que encontrar un valor de c para el que se cumpla que $4n + 7 \leq c \cdot n$. Con $c = 5$ la inecuación se cumple, por tanto: $f(n) = O(g(n)) = O(n)$; y esto solo se cumple para $n \geq n_0$, en este caso $n \geq 7$ porque: $4n + 7 \leq 5n \rightarrow 7 \leq 5n - 4n \rightarrow 7 \leq n$. De forma que el *Big-O* de $4n + 7$ es $O(n)$ para cualquier entrada mayor que 7.

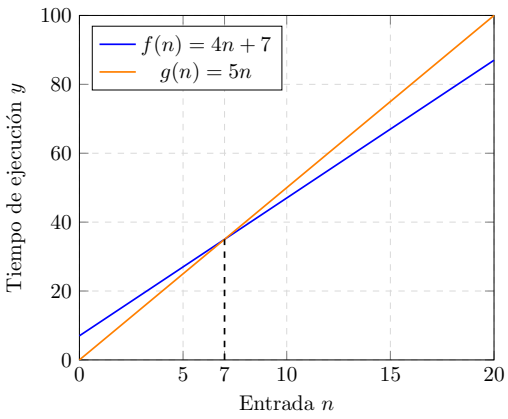

Figura 3: Ejemplo de *Big-O*

2. Implementaciones

2.1. Merge Sort Recursivo Serial (MSRS)

El algoritmo de ordenación por mezcla clásico se basa en el paradigma "divide y vencerás". Es decir, primero se divide un problema en otros subproblemas, se solucionan los subproblemas, y, se combinan para llegar a la solución final.[10]

```
1        public static void sort(int[] A, int length) {
2            if (length <=1) return;
3
4            int mid = length / 2;
5            int[] L = new int[mid];
6            int[] R = new int[length - mid];
7
8            for (int i = 0; i < mid; i++) L[i] = A[i];
9            for (int i = mid; i < length; i++) R[i - mid] = A[i];
10
11           sort(L, mid);
12           sort(R, length - mid);
13
14           merge(A, L, R, mid, length - mid);
15       }
```

Figura 4: Función `sort()` del Merge Sort Recursivo Serial

Como se observa en la Figura 4 el algoritmo a estudiar consta de una función `sort()` que toma una colección `A` y su longitud `length`. Después calcula el pivote `mid` desde donde dividir la colección original `A`, instancia dos colecciones locales, `L` y `R`, y copia los elementos de A a cada mitad. Estas mitades se ordenan independientemente a través de las llamadas recursivas de las líneas 11 y 12 a `sort()`.

```
1        public static void merge(int[] A, int[] L, int[] R, int left, int right) {
2            int i = 0, j = 0, k = 0;
3
4            while (i < left && j < right) A[k++] = (L[i] <= R[j])? L[i++] : R[j++];
5
6            while (i < left)  A[k++] = L[i++];
7            while (j < right) A[k++] = R[j++];
8        }
```

Figura 5: Función `merge()` del Merge Sort Recursivo Serial

Finalmente, se unen las mitades mediante la función `merge()` de la Figura 5 que toma la colección original `A`, y las sub-colecciones `L` y `R`, además de sus respectivas longitudes `left`, `right`. Para

[10] (Sedgewick, 2003)

controlar el recorrido de las tres colecciones se emplean tres índices: `i` para `L`, `j` para `R` y `k` para `A`. Un primer bucle coloca en `A[k]` el elemento más pequeño entre `L[i]` y `L[j]` sucesivamente. Los dos bucles posteriores colocan en `A[k]` los elementos no colocados en el bucle anterior. Por último, en caso de que una mitad se haya recorrido por completo, la restante se copia en `A`.

2.1.1. Complejidad

La complejidad temporal del algoritmo será la suma de las complejidades de cada línea. Según el modelo MAN, la comprobación del caso base (línea 2, Figura 4) y el cálculo del pivote `mid` toman tiempo constante $O(1)$ al ser operaciones básicas. La inicialización de `L` y `R` toma tiempo $O(n)$ porque en Java la creación de un arreglo implica la asignación de espacio en la memoria proporcional al tamaño de este, en este caso `lenght =` n. La copia de las sub-colecciones (bucles de las líneas 8-9) toman $O(n)$ al su tiempo de ejecución depender de n. Por último, `merge` (Figura 5) toma $O(n)$ porque en el peor caso se realizaran `left` + `right` = `lenght` comparaciones `L[i]<=R[j]`, que es lo mismo que n. Por tanto, el tiempo de ejecución respecto la entrada por ahora es $f(n) = 2O(1) + 2O(n) + 2O(n) + O(n)$, que es igual a $O(n)$ ya que según la notación *Big-O* se ignoran los términos de menor orden y los factores constantes, puesto que estos no afectan significativamente al crecimiento cuando n tiende a un número grande.

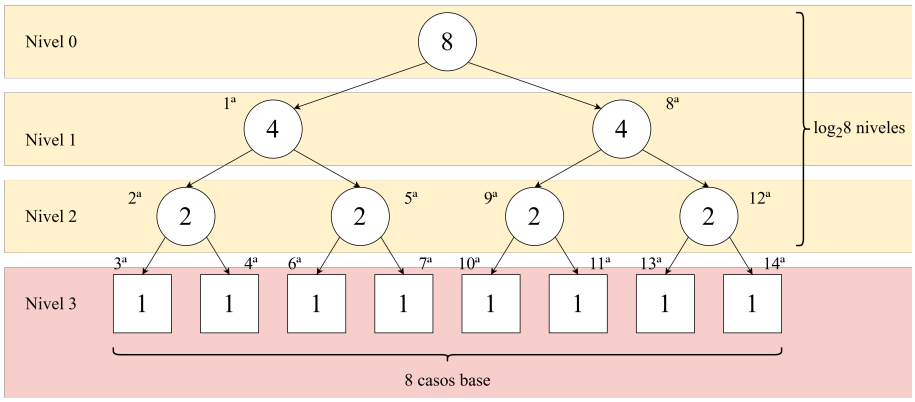

Figura 6: Árbol binario para $n = 8$

Lo anterior corresponde a una llamada a `sort()`, pero esta función es recursiva, por tanto, hay que determinar cuantas veces se llama a `sort()` en función del tamaño de la entrada n. Estos se facilita si rastreamos las llamadas a `sort()`, por ejemplo mediante un árbol binario como el de la Figura 6 donde cada nodo representa la longitud de la entrada `lenght`. Se observa que para 8 llamadas hay tres niveles en el árbol, la relación entre 8 y 3 es que $log_2 3 = 3$ ya que $8 = 2^3$. Por extensión, para un n tamaño de entrada hay $\log_2 n$ niveles. Esto se cumple siempre que n

sea múltiplo de 2, lo que da lugar a un árbol simétrico como el de Figura 8, en caso contrario es asimétrico y hay llamadas extras. El trabajo realizado en cada nivel i es $2^i \cdot n \cdot \frac{n}{2^i} = n$ como se observa en la Figura 7 donde n es el tamaño de entrada original. Finalmente, el trabajo $f(n)$ de una llamada a `sort()` es la suma del trabajo en cada nivel, menos el del último nivel porque los casos bases requieren $O(1)$ al realizarse un `return`. Esto es:

$$f(n) = \sum_{i=0}^{\log_2 n - 1} n = n \sum_{i=0}^{\log_2 n - 1} 1 = n \cdot \log_2 n$$

A continuación se aplica la definición de la notación *Big-0*, donde $f(n) = n \log_2 n$ y $g(n) = n \log n$. Existe un $f(n) = O(n \log n)$ siempre que:

$$n \log_2 n \leq c \cdot n \log n$$

Aislamos la constante c:

$$n \log_2 n \leq c \cdot n \log n \longrightarrow c \geq \frac{\log_2 n}{\log_{10} n} \longrightarrow c \geq \frac{\frac{\log_{10} n}{\log_{10} 2}}{\log_{10} n} \longrightarrow c \geq \frac{1}{\log 2}$$

Esto significa que la complejidad del MSRS se aleja de $O(n \log_n)$ en un factor aproximado de $3,32\,\%$ siempre que la entrada sea mayor que 1 y sea múltiplo de 2. Aun así, la literatura considera que el *Merge Sort* tiene complejidad $O(n \log n)$ por razones prácticas. [11]

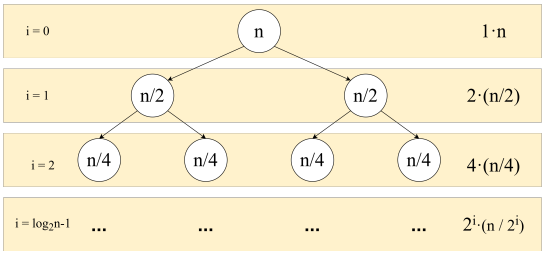

Figura 7: Tamaño de la entrada a lo largo de las llamadas a `sort()`

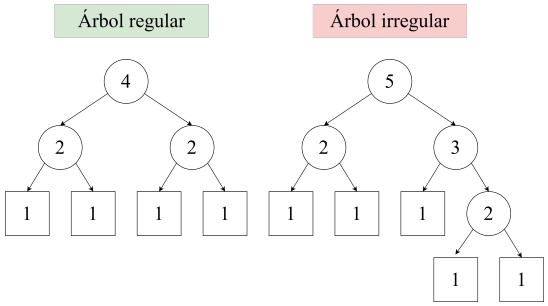

Figura 8: Árbol simétrico verus árbol asimétrico del MSRS

[11] (Sedgewick, 2003)

2.2. Merge Sort Iterativo Serial (MSIS)

El MSIS es la versión análoga al clásico MSRS, en esta implementación (Figura 9) se divide la colección en partes más pequeñas, después las partes adyacentes se unen, y se aumenta el tamaño de las partes. Estos tres pasos se repiten hasta que la parte tenga el tamaño de la colección original. El primer bucle determina el tamaño `lenght` de las partes: `size` $= 2, 4, 8...$ El segundo bucle determina el índice `left` desde el cual comienza la parte `arr[left]...arr[left+size-1]` que será unida a su adyacente `arr[mid]...arr[right]` mediante `merge()`. Este índice toma valores en intervalos de `2*size`. En la Figura 10 se presenta el proceso de ordenación de una colección: las casillas coloreadas representan las sucesivas iteraciones de `left` del segundo bucle; las casillas azules corresponden a la respectiva parte izquierda de una iteración; y las verdes la parte derecha.

```
1   public static void sort(int[] arr) {
2       int n = arr.length;
3       int[] aux = new int[n];
4
5       for (int size = 1; size < n; size *= 2) {
6           for (int left = 0; left < n - size; left += 2 * size) {
7               int mid = left + size - 1;
8               int right = Math.min(left + 2 * size - 1, n - 1);
9               merge(arr, aux, left, mid, right);
10          }
11      }
12  }
```

Figura 9: Función `sort()` del Merge Sort Iterativo Serial

Dado que las partes son de longitud variable, se ha modificado la función `merge()` anterior para no depender de dos colecciones auxiliares, sino de una sola. En esta implementación `arr[left]...arr[mid]` es una parte que se une a su adyacente `arr[mid+1]...arr[right]`. Las dos partes y la colección original se recorren mediante los índices `i`, `j`, `k` respectivamente.

2.2.1. Complejidad temporal

Si se observa el árbol binario de la Figura 12, el MSIS recorre el árbol desde la base de la recursión hasta la parte superior, ya que se realizan uniones entre partes de 1-1, 2-2- 4-4... Como en el MSRS, el árbol será simétrico solo si el tamaño de entrada en múltiplo de 2. Siguiendo el mismo método y tomando las mismas suposiciones que en el cálculo de la complejidad temporal del MSRS, la complejidad del MSRS es $O(n \log n)$ porque hay $\log_2 n - 1$ llamadas a `sort()` en las cuales se realiza un trabajo de $O(n)$ en el peor caso.

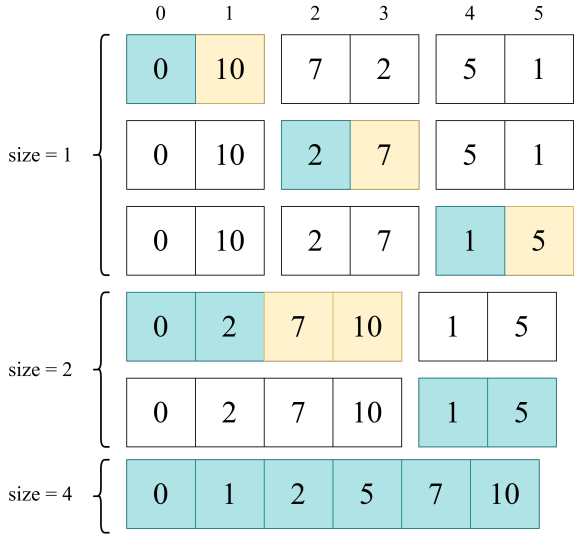

Figura 10: Ejecución del Merge Sort Iterativo Serial

```
1       private static void merge(int[] arr, int[] aux, int left, int mid, int right) {
2           for (int i = left; i <= right; i++) aux[i] = arr[i];
3           int i = left, j = mid + 1, k = left;
4
5           while (i <= mid && j <= right) arr[k++] = (aux[i] <= aux[j])?
6           aux[i++] : aux[j++];
7           while (i <= mid) arr[k++] = aux[i++];
8       }
```

Figura 11: Función `merge()` del Merge Sort Iterativo Serial

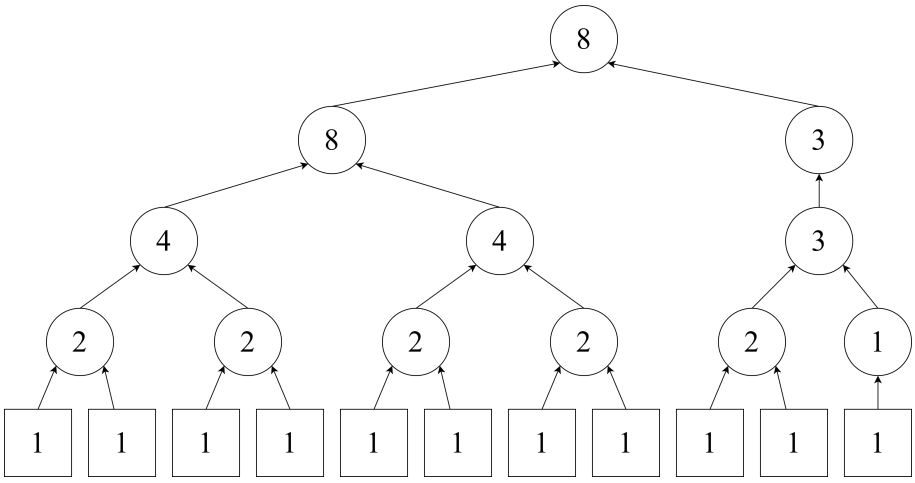

Figura 12: Árbol binario del Merge Sort Iterativo Serial

2.3. Merge Sort Recursivo Paralelo (MSRP)

Un proceso es la ejecución de las instrucciones de un programa, después de que estas instrucciones se hayan movido desde la memoria secundaria (SSD por ejemplo) hasta la primaria (RAM). El sistema operativo en quien se encarga de crear los procesos y guarda en la memoria la información asociada al proceso: un identificador de proceso único (*PID*); un espacio de direcciones de memoria, ya que un proceso lee y modifica datos guardados en la memoria; el estado del proceso; y, una lista de los archivos que el programa usa. En cambio, un hilo de ejecución es una secuencia de instrucciones que el planificador del sistema operativo puede manejar independientemente. [12] Hasta se han mostrado programas que al ejecutarse toman la forma de un proceso con un solo hilo de ejecución. La idea es mejorar el rendimiento del *Merge Sort* haciendo uso de más de un hilo de ejecución.

La herramienta más apropiada que proporciona Java para esta tarea es la clase `ForkJoinPool`. Una piscina de hilos (*thread pool*), es un espacio en el que se mantienen un conjunto fijo de hilos de ejecución reutilizables que esperan a que se les pase un conjunto de instrucciones a ejecutar. Una piscina evita la necesidad de crear y destruir hilos constantemente, cosa que es costosa.

La `ForkJoinPool` permite crear piscinas basadas en un algoritmo de robo de trabajo (*work-stealing algorithm*). [13] Esto significa que las tareas se acumulan en una cola compartida entre todos los hilos, pero, además, cada hilo consta de su propia cola doblemente terminada (decola). Los hilos extraen de la cola las tareas y las ejecutan, si la tarea produce subtareas estas se guardan en su decola. Puede ocurrir que la decola de un hilo se vacíe, en ese caso, el hilo desencola una tarea de la cola compartida. La `ForkJoinPool` es útil si el algoritmo genera muchas subtareas, ya que el uso de decolas propias reduce significativamente la cantidad de accesos a la cola compartida. Además, las tareas de la cola compartida serán siempre de mayor tamaño que las subtareas que generen tareas ubicadas en las decolas. Por ende, esta piscina es apropiada para el *Merge Sort*.

Para crear una `ForkJoinPool` simplemente se llama a su constructor y se pasa el número de hilos que tendrá la piscina: `ForkJoinPool forkjoinpool = new ForkJoinPool(numHilos)`. Existen dos maneras de pasar tareas a una `ForkJoinPool`: tareas que no retornan ningún valor (`RecursiveAction`) y tareas que sí retornan (`RecursiveTask`). Ambas subclases heredan de `ForkJoinTask`. [14]

[12] (Bobrov, 2023) [13] (Ramgir y Samoylov, 2017) [14] (*ForkJoinPool (java SE 10 & JDK 10)*, s.f.)

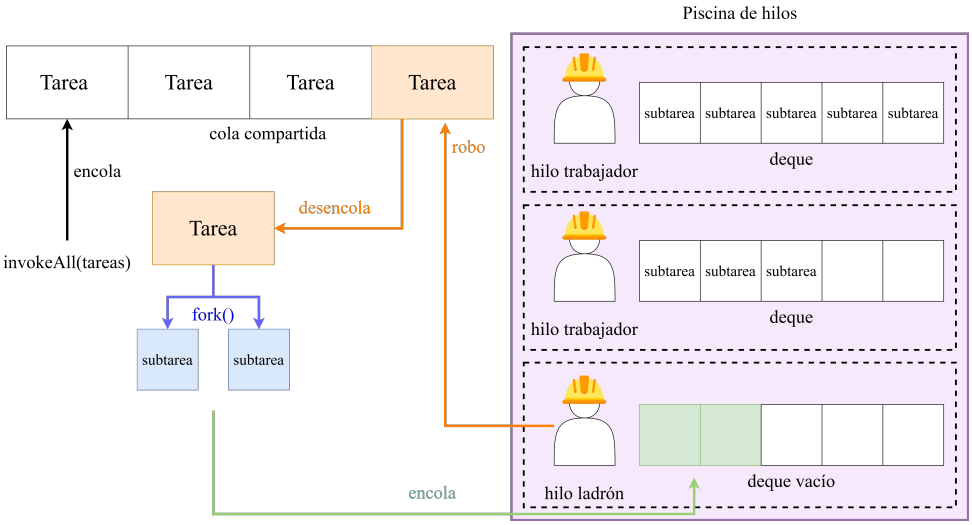

Figura 13: Funcionamiento del `ForkJoinPool`

Concretamente, la lógica del MSRP se inscribe en una clase `MergeSortRecursivoParalelo` que hereda de clase abstracta `RecursiveAction` como se observa en la Figura 14

```
1  public class MergeSortRecursivoParalelo extends RecursiveAction {
2      private final int[] A;
3      private final int length;
4      private final int threshold = 8192;
5
6      public MergeSortRecursivoParalelo(int[] A, int length){ //Constructor }
7      @Override
8      protected void compute() { //Logica del Merge Sort }
9      public static void merge(int[] A, int[] L, int[] R, int left, int right) { }
10 }
```

Figura 14: Esquema de la clase `MergeSortRecursivoParalelo`

En dicha clase el algoritmo de *Merge Sort* se queda encapsulado en el método abstracto `compute()`, como se observa en la Figura 15. La primera condición ejecuta el algoritmo iterativo en caso de que la longitud de entrada de la colección sea inferior a un límite. Eso se debe a que llega un momento en que ya no es eficiente generar más subtareas porque el costo de gestión de las subtareas es más elevado que el costo de resolver los subproblemas de forma iterativa. Después se crean dos subtareas, por ahora inactivas, para cada lado de la colección. A continuación, se pasan las subtareas a `invokeAll()`, que las encola en la cola compartida de piscina y espera a que `Left` y `Right` finalicen. Finalmente, se unen las dos partes con el mismo algoritmo `merge()` de la implementación recursiva iterativa.

```java
@Override
protected void compute() {
    if (length <= threshold) {
        MergeSortRecursivoSerial.sort(A, length);
    } else {
        final int mid = length / 2;
        int[] L = new int[mid];
        int[] R = new int[length - mid];

        for (int i = 0; i < mid; i++) L[i] = A[i];
        for (int i = mid; i < length; i++) R[i - mid] = A[i];

        final MergeSortRecursivoParalelo Left = new MergeSortRecursivoParalelo(L, mid);
        final MergeSortRecursivoParalelo Right = new MergeSortRecursivoParalelo(R,
            length - mid);

        invokeAll(Left, Right);

        merge(A, L, R, mid, length - mid);
    }
}
```

Figura 15: Método `compute()` del MSRP

2.4. Merge Sort Paralelo (MSIP)

Para la paralelización del algoritmo iterativo se ha hecho uso de la interfaz `ExecutorService` que proporciona Java y que proporciona un marco para gestionar y controlar la ejecución de tareas asincrónicas. A diferencia de las piscinas `ForkJoin`, el `ExecutorService` utiliza un algoritmo de trabajo compartido (*work-sharing algorithm*). Esto implica que solo hay una cola compartida entre todos los hilos: una vez termina un hilo de ejecutar una subtarea, extrae otra de la cola. Este flujo de ejecución es apropiado para tareas independientes entre ellas.

En este caso, la implementación iterativa paralela se hace en una función `sort()` como la de la Figura 16. Para crear una instancia de `ExecutorService` se usan los métodos que proporciona la clase `Executors` de Java. Particularmente, `.newFixedThreadPool(numHilos)` crea una piscina con un número de hilos fijo determinado. En este programa se crean un número de hilos correspondiente al número de núcleos disponibles en el ordenador para la JVM. A continuación, para cada iteración se encola en `executor` una tarea `merge()` mediante una expresión lambda que recibe `executor.submit()`. Cada llamada a `merge()` es una subtarea más. Estas subtareas se guardan en un objeto de la clase `Future` que permite sincronizar la ejecución de tareas, ya que después se llama a `future.get()` que obliga al hilo principal a esperar a que acaben todas las tareas encoladas. De lo contrario, podríamos pasar al siguiente `size` sin asegurarse de que todas las partes están ordenadas. Una vez completado los bucles se llama a `executor.shutdown()`, que espera a que, una vez terminen de

```java
public static void sort(int[] array) {
    int n = array.length;
    int[] aux = new int[n];
    ExecutorService executor =
        Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

    for (int size = 1; size < n; size *= 2) {
        for (int left = 0; left < n - size; left += 2 * size) {
            int mid = left + size - 1;
            int right = Math.min(left + 2 * size - 1, n - 1);
            int finalLeft = left; //El resto son efectivamente finales
            Future<?> future = executor.submit(() -> merge(array, aux, finalLeft, mid,
                right));
            try {
                future.get(); // Esperar a que la tarea termine
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
    executor.shutdown();
}
```

Figura 16: Método `sort()` del MSIP

ejecutarse todas las tareas encoladas anteriormente, cierra la piscina `executor` y libera los hilos.

3. Parte experimental

3.1. Procedimiento

1. Se realizan diez ejecuciones para un n tamaño de entrada para cada uno de los cuatro algoritmos: MSRS, MSIS, MSRP, MSIP.

2. Hay 20 n tamaños que van de 100 hasta 1.000.000.

3. El tiempo de ejecución se mide mediante la función `System.nanoTime` que retorna el tiempo actual más preciso disponible en el sistema. El valor devuelto son los nanosegundos desde un tiempo arbitrario y provee de precisión nanosegundaria, pero no necesariamente exactitud nanosegundaria Cada ejecución se realizan entre una variable `long startTime` y `long endTime`. El tiempo de ejecución es la diferencia entre estas.[15]

4. Después se elimina el peor y mejor tiempo de entre las diez ejecuciones, quedando así ocho.

5. Todas las colecciones son generadas mediante la clase `SplittableRandom` de Java, que en este caso genera colecciones de tipo `int` con valores pseudoaleatorios extraídos de una distribución uniforme. [16]

6. Antes de cada ejecución se llama al `Garbage Collector` asegurar que no haya interferencias de otros procesos.

7. En el caso de los algoritmos paralelos se establece un mismo número de hilos para cada piscina para que la comparación sea justa.

8. El código de las implementaciones, el código del *benchmark* y los resultados del *benchmark* quedan recogidos en los Apéndices A, B y C respectivamente.

3.2. Entorno

Los algoritmos son ejecutados en un computador con procesador Intel(R) Core(TM) i5-11400F @ 2,60GHz y 16GB de RAM DDR4 3200MHz. En el SO de Windows 10 Pro 22H2 (x64). El IDE de ejecución es IntelliJ Idea Community Edition 2023.3.3.

[15] (*System (Java Platform SE 8)*, 2024)

[16] (*SplitTableRandom (Java Platform SE 8)*, 2024)

4. Discusión de resultados

4.1. Comparación

4.2. Conclusión

5. Bibliografía

Referencias

Bhargava, A. Y. (2016). *Grokking algorithms.* Manning Publications.

Bobrov, K. (2023). *Grokking concurrency.* Manning. Descargado de https://www.manning.com/books/grokking-concurrency

College, W. (2000). *Forms of iteration.* Descargado de https://cs111.wellesley.edu/archive/cs111_spring00/public_html/lectures/iteration.html

[ELI], E. L. I., y [NOVA], N. V. C. C. (s.f.). *Reading: Structured Programming.* Descargado de https://courses.lumenlearning.com/sanjacinto-computerapps/chapter/reading-structured-programming/

ForkJoinPool (java SE 10 & JDK 10). (s.f.). https://docs.oracle.com/javase/10/docs/api/java/util/concurrent/ForkJoinPool.html. (Accessed: 2024-12-15)

Heineman, G. T., Pollice, G., y Selkow, S. (2008). *Algorithms in a nutshell.* Sebastopol, CA, Estados Unidos de América: O'Reilly Media.

Knuth, D. E. (1997). *The Art of Computer Programming: Seminumerical algorithms.* Addison-Wesley Professional.

Magee, J., y Kramer, J. (2006). *Concurrency.* John Wiley & Sons.

McMillan, M. (2007). Basic sorting algorithms. En *Data structures and algorithms using c#* (p. 42–43). Cambridge University Press.

Ramgir, M., y Samoylov, N. (2017). *Java 9 high performance.* Birmingham, Inglaterra: Packt Publishing.

Sedgewick, R. (2003). *Algorithms in java* (3.ª ed.). Boston, MA, Estados Unidos de América: Addison-Wesley Educational.

Skiena, S. S. (2008). *The Algorithm Design Manual*. Springer London. Descargado de https://doi.org/10.1007/978-1-84800-070-4 doi: 10.1007/978-1-84800-070-4

SplitTableRandom (Java Platform SE 8). (2024, 9). Descargado de https://docs.oracle.com/javase/8/docs/api/java/util/SplittableRandom.html

System (Java Platform SE 8). (2024, 9). Descargado de https://docs.oracle.com/javase/8/docs/api/java/lang/System.html

6. Anexo