Impacto de la concurrencia en el rendimiento del algoritmo de ordenación *Merge Sort*

¿En qué medida las técnicas de concurrencia de Java optimizan el *Merge Sort* recursivo e iterativo?

MONOGRAFIA DE INFORMÁTICA

Cómputo de palabras: 3990
Código del alumno: lqv708

Índice

1. Introducción

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... El proceso de colocar datos en un cierto orden se llama ordenación y es una operación común en los sistemas informáticos, ya que facilita su búsqueda posterior.[1] Aunque, la ordenación ha sido ampliamente estudiada, no existe un algoritmo de ordenación perfecto.[2] Actualmente, se siguen desarrollando, y además relacionan una variedad de conceptos informáticos, como la concurrencia, el paralelismo, la ejecución serial, la iteración o la recursión. Algunos algoritmos ordenación comunes son la ordenación de burbuja, por inserción, o el ordenamiento rápido (*quicksort*).[3]

La finalidad de esta investigación es evaluar cuatro implementaciones del algoritmo de ordenación *Merge Sort* en términos de **complejidad temporal** y **tiempos de ejecución** reales, además, de la naturaleza específica de cada implementación; con el objetivo de responder a la pregunta «¿En qué medida las técnicas de concurrencia de Java optimizan el *Merge Sort* recursivo e iterativo?». Concretamente, se comparan el Merge Sort Iterativo Serial (MSIS) y el Merge Sort Recursivo Serial (MSRS) con sus contrapartes paralelizadas: el Merge Sort Recursivo Paralelo (MSRP) y el Merge Sort Iterativo Paralelo (MSIP).

La evaluación se compone de un *análisis a priori*, basado en la estructura de cada algoritmo sin ejecutarlo y calculando un rendimiento esperado; y de un *análisis a posteriori*, enfocado en medir el rendimiento de un algoritmo para una muestra concreta.[4]

1.1. *Merge Sort*

El ordenamiento por mezcla es un algoritmo basado en la técnica divide y vencerás. Permite ordenar un conjunto de datos a través de, primero, dividir la colección en dos mitades; dividir las sub-colecciones en más mitades hasta que contengan cero o un elemento; ordenar cada una; y, finalmente, unir (*merge*) todas las sub-colecciones de forma ordenada, quedan-

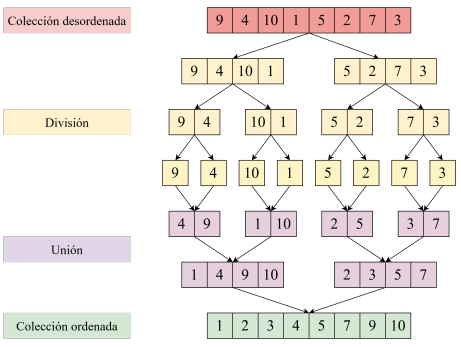

Figura 1: Funcionamiento del *Merge Sort*

[1] (Knuth, 1997) [2] (McMillan, 2007) [3] (Pandey, 2008) [4] (Mohan y Kapoor, 2025)

do ordenada la colección.[5]

1.2. 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**. 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.[6] Este estudio pretende aplicar el paralelismo a la ordenación por mezcla para aumentar su eficiencia.

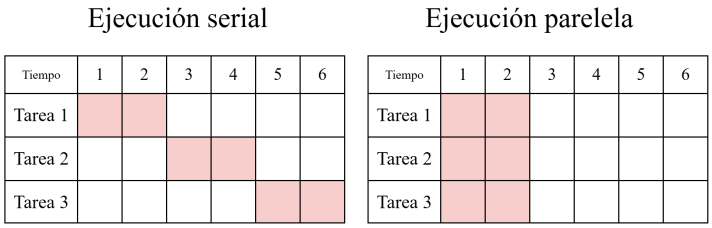

Figura 2: Comparación de los diagramas de Gantt

1.3. 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.[7] La recursión es una estructura de repetición.[8] Si un programa incorpora una estructura recursiva, quiere decir que hay una función que se llama a sí misma. [9] 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.

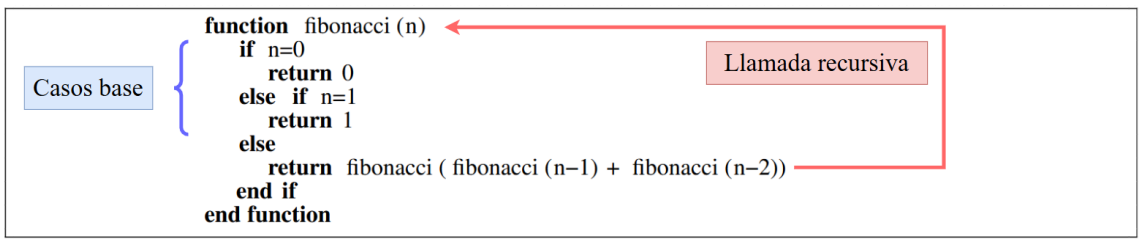

Figura 3: Ejemplo de recursión: serie de fibonacci

[5] (Skiena, 2008) [6] (Bobrov, 2023) [7] ([ELI] y [NOVA], s.f.) [8] (College, 2000) [9] (Bhargava, 2016)

1.4. Complejidad

La eficiencia de un algoritmo de ordenación usualmente se cuantifica en términos de complejidad temporal y espacial.[10] La complejidad temporal es una medida de la variación del tiempo de ejecución de un algoritmo a medida que el tamaño de la entrada n crece. En el caso de un algoritmo de ordenación es la longitud del arreglo.[11]

Puesto que el tiempo de ejecución se ve afectado por variables como el soporte físico y lógico se estudia el comportamiento asintótico: cuando n tiende al infinito. Este método permite comparar algoritmos entre plataformas distintas.[12]

Caracterizar un algoritmo por medio de su complejidad es una abstracción basada en la suposición de que las operaciones básicas duran una unidad de tiempo y que cada línea de código es una instrucción básica. Los bucles y llamadas a funciones se evalúan sumando sus instrucciones básicas.[13]

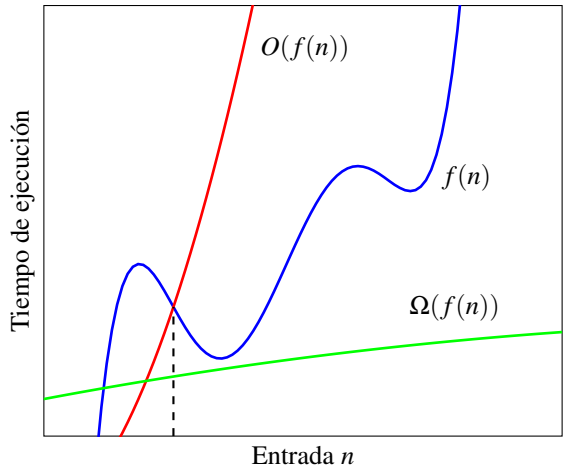

Figura 4: Función de ejemplo $f(n)$ acotada superiormente por $O(n)$ y inferiormente por $\Omega(n)$

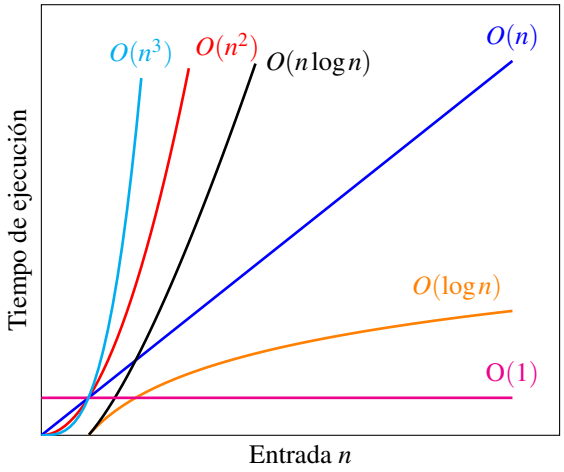

Figura 5: Complejidades temporales comunes

Existen tres formas de medir la complejidad temporal expresadas en la función del peor caso $O(n)$ que mide el tiempo máximo de ejecución que un algoritmo puede necesitar para una entrada n; la función del mejor caso $\Omega(n)$ que mide el tiempo mínimo de ejecución; y, la del caso promedio $\Theta(n)$ que mide el tiempo de ejecución típico.[14]

[10] (Molluzzo y Buckley, 1997) [11] (Ching, Phoon, y Shuku, 2023) [12] (Heineman, Pollice, y Selkow, 2008) [13] (DataCamp, 2024) [14] (Levitin, 2012)

1.4.1. Notación de Landau

Normalmente se califica a un algoritmo de ordenación a través de la función del peor caso por su simplicidad y porque garantiza el buen funcionamiento en sistemas críticos y previene problemas de escalabilidad.[15] Esta será la que se valore para las diferentes implementaciones del *Merge Sort*.

Sean dos funciones $f(n)$ y $g(n)$, 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 $g(n)$.[16][17]

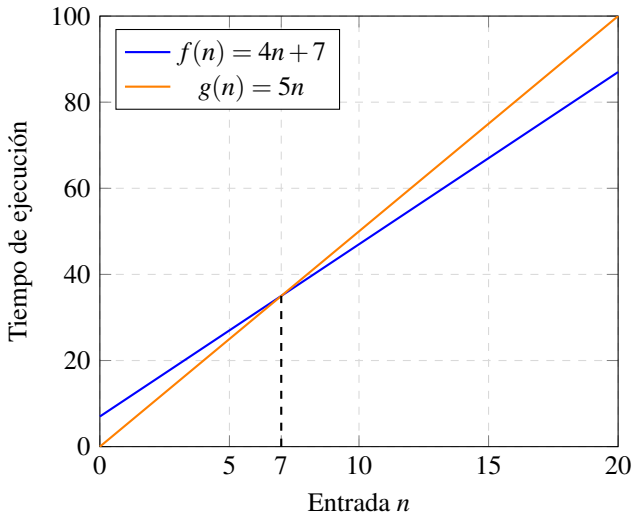

Figura 6: Función de ejemplo $f(n)$ acotada por *Big-O* $O(n)$ para $c = 5$ y $n_0 = 7$

[15] (Correa, 2024) [16] (Sipser, 1997) [17] (Landau, 1909)

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.[18]

```
1  public static void sort(int[] arr, int[] aux, int left, int right) {
2     if (left >= right) return;
3
4     int mid = left + (right - left) / 2;
5
6     sort(arr, aux, left, mid);
7     sort(arr, aux,mid+1, right);
8
9     merge(arr, aux, left, mid, right);
10 }
```

Figura 7: Función `sort()` del Merge Sort Recursivo Serial

El algoritmo a estudiar consta de una función `sort()` que toma una colección `arr[]`, un índice inicial `left` y un índice final `right`. (Figura 7) Después calcula el pivote `mid` desde donde dividir la colección original `arr[]` y se ejecuta una llamada recursiva a `sort()` para cada mitad `arr[left]... arr[mid]` y `arr[mid+1]... arr[right]`.

```
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
4     int i = left, j = mid + 1, k = left;
5
6     while (i <= mid && j <= right) arr[k++] = (aux[i] <= aux[j])? aux[i++] : aux[j++];
7
8     while (i <= mid) arr[k++] = aux[i++];
9  }
```

Figura 8: Función `merge()` del Merge Sort Recursivo Serial

Finalmente, se unen las mitades mediante la función `merge()` de la Figura 8 que toma los mismos argumentos que `sort()`, además del parámetro `mid`. Para controlar el recorrido de las tres colecciones se emplean tres índices: `i` para la midad izquierda `arr[left]... arr[mid]`, `j` para la mitad derecha `arr[mid+1]... arr[right]` y `k` para el arreglo original `arr[left]... arr[right]`. Un primer bucle copia en un arreglo auxiliar `aux[]` todos los elementos de `arr[]`. Después, un segundo bucle recorre simultáneamente las dos

[18] (Sedgewick, 2003)

mitades y la colección auxiliar; mientras coloca en `arr[k]` el elemento más pequeño entre `aux[i]` y `aux[j]`. Existen dos posibilidades: primera, que el bucle se detenga porque `i > mid`, que implica que todos los elementos de la primera mitad `aux[left]... arr[mid]` han sido copiados en `arr[]`; segunda, que el bucle se detenga porque `j > right`, que implica que todos los elementos de la segunda mitad `aux[mid+1]... arr[right]` han sido copiados en `arr[]`. Sin embargo, si `i` no alcanza `mid` quedan elementos sin copiar en `arr[]`: entonces un tercer bucle copia los elementos restantes de `aux[i]... aux[mid]` en `arr[]`.

Todas las implementaciones (MSRS, MSIS, MSRP, MSIP) emplean el mismo método `merge()` para unir las sucesivas mitades.

2.1.1. Complejidad temporal

La complejidad temporal del algoritmo será la suma de las complejidades de cada línea. La comprobación del caso base (línea 2, Figura 7) y el cálculo del pivote `mid` toman tiempo constante $O(1)$ al ser operaciones básicas. Por último, `merge()` toma $O(n)$ porque en el peor caso se realizarán `right + 1 - left` $= n$ comparaciones en el primer bucle, que es $O(n)$. Por tanto, el tiempo de ejecución respecto a la entrada por ahora es $f(n) = 2O(1) + O(n)$, 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.

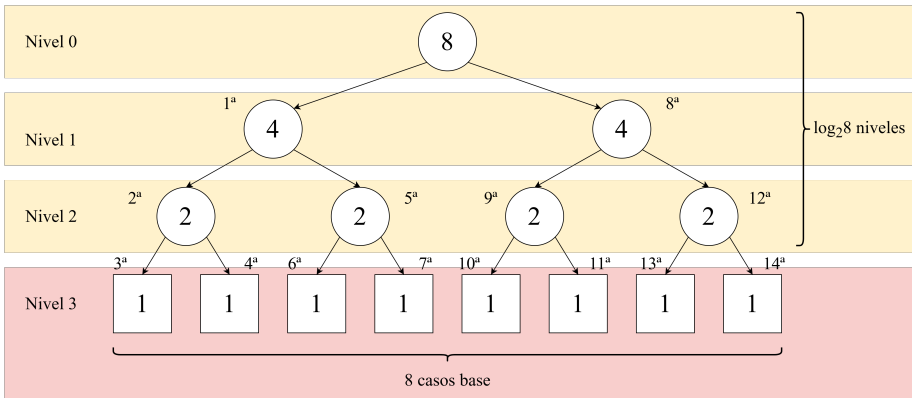

Figura 9: Árbol binario para $n = 8$

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 9 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 $log_2 8 = 3$ 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 balanceado como

el de Figura 11, en caso contrario está desbalanceado y hay llamadas extras. El trabajo realizado en cada nivel i es $2^i \cdot \frac{n}{2^i} = n$ como se observa en la Figura 10 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`; quedando:

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

A continuación se aplica la definición de la notación *Big-0* para $f(n) = n\log_2 n$ 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. [19]

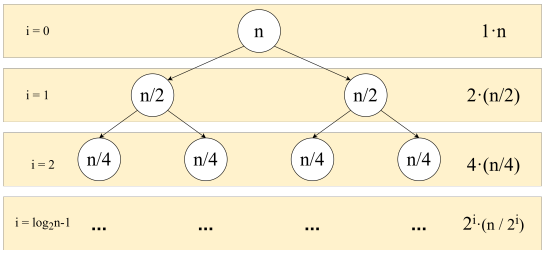

Figura 10: Tamaño de la entrada a lo largo de las llamadas a `sort()`

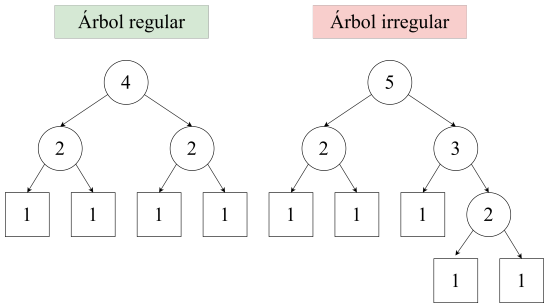

Figura 11: Árbol balanceado versus árbol desbalanceado del MSRS

2.2. Merge Sort Iterativo Serial (MSIS)

El MSIS es la versión análoga al clásico MSRS, en esta implementación 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

[19] (Sedgewick, 2003)

de las subcolecciones: `size` $= 2, 4, 8...$ El segundo bucle recorre `arr[]` en pasos de `2*size`, uniendo el subarreglo `arr[left]..arr[left+size-1]` a su adyacente `arr[mid]...arr[right]` mediante `merge()`. El índice MID determina el final del primer arreglo y `right` el final del adyacente. En la Figura 13 se presenta el proceso de ordenación de una colección de ejemplo: cada nivel es una iteración del segundo bucle donde se une una parte (casillas azules) a su adyacente (casillas amarillas).

```
public static void sort(int[] arr, int[] aux) {
   int n = arr.length;

   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);
         merge(arr, aux, left, mid, right);
      }
   }
}
```

Figura 12: Función `sort()` del Merge Sort Iterativo Serial

2.2.1. Complejidad temporal

El MSIS recorre el árbol desde la base de la recursión hasta la parte superior, ya que se realizan uniones entre partes de longitud 1-1, 2-2- 4-4... Como en el MSRS, el árbol será balanceado solo si el tamaño de entrada es 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.

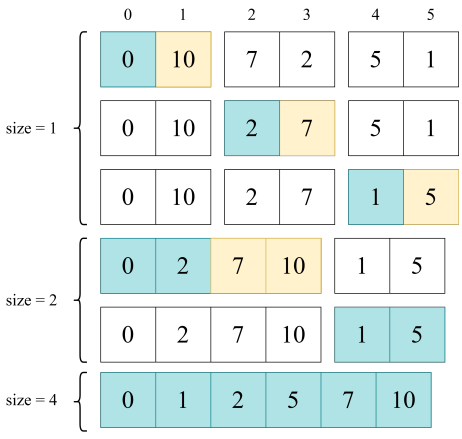

Figura 13: Ejecución del Merge Sort Iterativo Serial

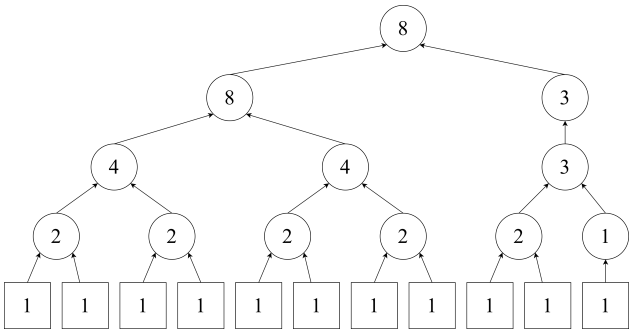

Figura 14: Árbol binario del Merge Sort Iterativo Serial

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 hasta la primaria. El sistema operativo crea los procesos y guarda en la memoria información asociada. En cambio, un **hilo de ejecución** es una secuencia de instrucciones que el planificador del sistema operativo puede manejar independientemente. [20] Hasta el momento se han mostrado programas que al ejecutarse toman la forma de un proceso con un solo hilo de ejecución (MSIS y MSRS). 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 este problema es la clase `ForkJoinPool`.[21] Una piscina de hilos es un espacio en el que se mantienen un conjunto fijo de hilos de ejecución reusables que esperan a que se les pase un conjunto de instrucciones a ejecutar. Una piscina evita la necesidad de crear y destruir hilos constantemente, hecho que conlleva un costo computacional elevado.[22]

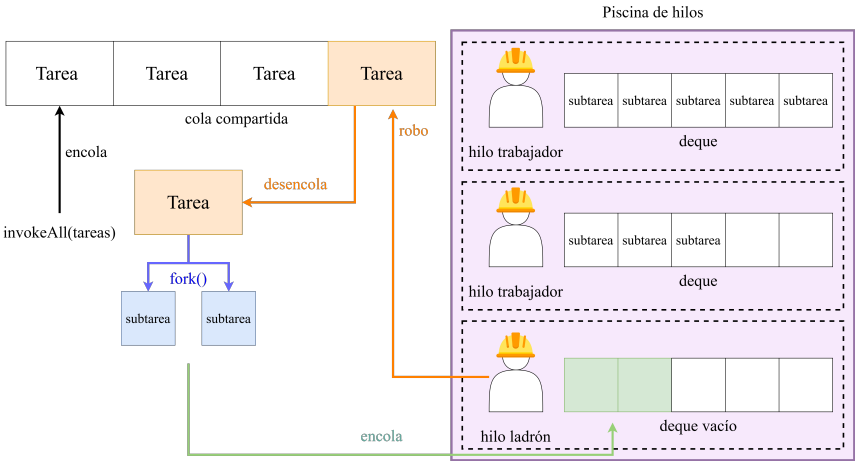

Figura 15: Funcionamiento del `ForkJoinPool`

La `ForkJoinPool` permite crear piscinas basadas en un algoritmo de robo de trabajo (*work-stealing*).[23] 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.[24] La `ForkJoinPool` es útil si el algoritmo genera muchas subtareas, ya que el uso de decolas propias reduce significativamente la cantidad

[20] (Bobrov, 2023) [21] (*ForkJoinPool (java SE 10 & JDK 10)*, s.f.) [22] (Engle, 2022) [23] (Ramgir y Samoylov, 2017) [24] (Kumar, 2024)

de accesos a la cola compartida.[25] Adicionalmente, las tareas de la cola compartida serán siempre de mayor tamaño que las subtareas que generen tareas ubicadas en las decolas. Por ende, `ForkJoinPool` es apropiada para la paralelización del *Merge Sort*, en tanto que se generan subtareas que después se volverán a unir.

```
ForkJoinPool pool = new ForkJoinPool(parallelismLevel);
MSrecursivoParalelo task = new MSrecursivoParalelo(arr, aux, left, right);
pool.invoke(task);
```

Figura 16: Inicialización de la `ForkJoinPool`

Para crear una `ForkJoinPool` se llama a su constructor y se pasa el número de hilos trabajadores que tendrá la piscina (`parallelismLevel`). Después, se debe pasar una tarea a la piscina que esta procesará cuando se llame al método `.invoke()`. Existen dos tipos tareas: aquellas que no retornan ningún valor (`RecursiveAction`) y aquellas que sí retornan (`RecursiveTask`).[26] En este caso, se pasa una tarea del tipo `RecursiveAction`. (Figura 16)

Concretamente, la lógica del MSRP se inscribe en una clase `MSrecursivoParalelo` que hereda de la clase abstracta `RecursiveAction` (Figura 17).

```
public class MSrecursivoParaleloSinUmbral extends RecursiveAction {
   private final int[] arr, aux;
   private final int right, left;

   public MSrecursivoParaleloSinUmbral(int[] arr, int[] aux, int left, int right) {
      this.arr = arr;
      this.aux = aux;

      this.left = left;
      this.right = right;
   }

   @Override
   protected void compute() { }

   private static void merge(int[] arr, int[] aux, int left, int mid, int right) { }
}
```

Figura 17: Esquema de la clase `MergeSortRecursivoParalelo`

En dicha clase el algoritmo queda encapsulado en el método `compute()`, como se observa en la Figura 18. Primero se comprueba el caso base. 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 común `merge()`.

[25] (Blumofe y Leiserson, 1999)

[26] (Jenkov, 2024)

```java
@Override
protected void compute() {
   if (left >= right) return;

   int mid = left + (right - left) / 2;

   final MSrecursivoParalelo Left = new MSrecursivoParalelo(arr, aux, left, mid);
   final MSrecursivoParalelo Right = new MSrecursivoParalelo(arr, aux, mid + 1, right);

   invokeAll(Left, Right);

   merge(arr, aux, left, mid, right);
}
```

Figura 18: Método `compute()` del MSRP

Normalmente llega un momento en que generar más subtareas torna ineficiente dado que el costo de gestión de hilos es mayor que el costo que conllevaría ejecutar las tareas de forma serial. En este caso, colecciones con baja longitud ralentizarían al MSRP. Por ende es común establecer un umbral (*threshold*) a partir del cual se ejecutan las subtareas serialmente.

```java
protected void compute() {
   int length = (right + 1 - left);
   if (length <= THRESHOLD) {
      MSrecursivoSerial.sort(arr, aux, left, right);
   } else {
      // Ejecucion paralela
   }
}
```

Figura 19: MSRP modificado con umbral

Se ha conducido un test piloto para determinar el umbral adecuado para esta implementación. Se han tomado 50 muestras del tiempo de ejecución para 20 longitudes de arreglo para el MSRS y para el MSRP, pero sin un umbral definido (Figura 18). En la Tabla 1 se comparan los promedios de las muestras obtenidas. Los tamaños de las colecciones son potencias de dos, en tanto que se intenta conseguir un árbol binario balanceado que aprovecha mejor los recursos. Obsérvese que hasta el tamaño 2^{15} el algoritmo más rápido es el MSRS. Entonces se modifica el método `compute()` para que, en caso de que la longitud del arreglo de entrada sea inferior al umbral de 2^{15}, se ejecute la versión serial. (Figura 19)

Tamaño	Tiempo MSRS (ms)	Tiempo MSRP sin umbral (ms)
2	0,002	0,271
4	0,004	0,325
8	0,003	0,375
16	0,003	0,402
32	0,005	0,519
64	0,004	0,751
128	0,005	0,806
256	0,013	0,924
512	0,027	0,987
1024	0,057	1,109
2048	0,120	1,227
4096	0,263	1,424
8192	0,529	1,468
16384	1,099	1,689
32768	2,224	2,186
65536	4,603	2,529
131072	9,332	3,770
262144	19,450	6,248
524288	39,380	11,012
1048576	81,661	25,592

Tabla 1: Datos de prueba para MSRS y MSRP sin umbral

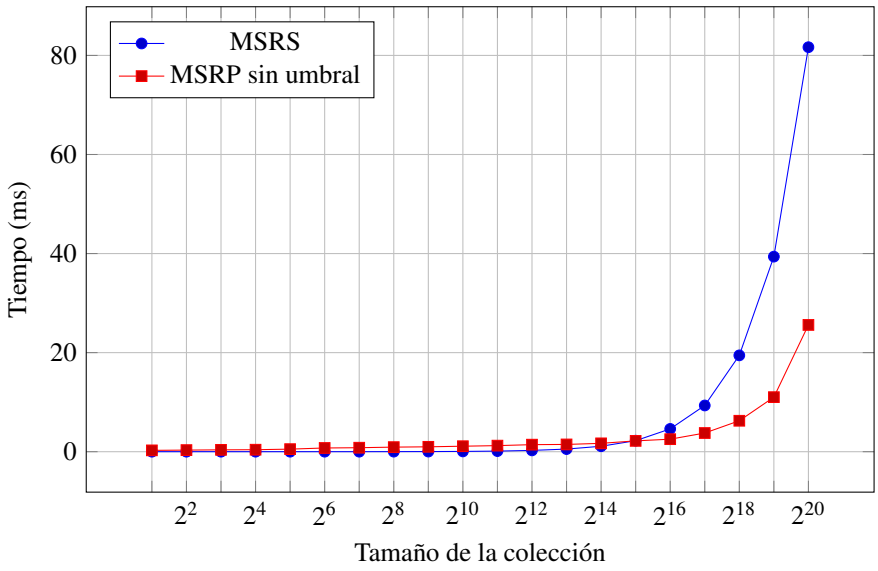

Figura 20: Datos de prueba para MSRS y MSRP sin umbral

2.4. Merge Sort Iterativo 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.[27]

```
public static void sort(int[] array, int aux[], ExecutorService executor) {
   int length = arr.length;
   List<Future<?>> futures = new ArrayList<>();

   for (int size = 1; size < length; size *= 2) {

      for (int left = 0; left < length - size; left += 2 * size) {
         int mid = left + size - 1;
         int right = Math.min(left + 2 * size - 1, length - 1);
         int finalLeft = left; //El resto son efectivamente finales
         futures.add(executor.submit(() -> merge(arr, aux, finalLeft, mid, right)));
      }
      for (Future<?> future : futures) {
         try {
            future.get();
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
      futures.clear();

   }
   executor.shutdown();
}
```

Figura 21: Método `sort()` del MSIP

En este caso, la implementación iterativa paralela se hace en una función `sort()` (Figura 21). Para crear una instancia de `ExecutorService` se usan los métodos que proporciona la clase `Executors` de Java. Particularmente, `.newFixedThreadPool(parallelismLevel)` crea una piscina con un número de hilos fijo determinado.

```
ExecutorService executorService = Executors.newFixedThreadPool(parallelismLevel);
MSiterativoParalelo.sort(arr, aux, executor);
```

Figura 22: Inicialización del `ExecutorService`

[27] (Oracle, 2025)

A continuación, para cada iteración se encola internamente en `executor` una tarea `merge()` mediante una expresión lambda que recibe `executor.submit()` como argumento. Entonces un número determinado de hilos trabajadores toman las tareas de la cola interna y las ejecutan. Si todos los hilos están ocupados, la tarea esperará en la cola hasta que un hilo esté disponible. Una vez que un hilo esté disponible, tomará la tarea de la cola y la ejecutará. Cada llamada a `merge()` implica que `.submit()` retorne un objeto de la clase `Future` que representa el resultado de una operación asíncrona. En este caso, se añaden los `Future` a una lista. Finalmente se recorre la lista y para cada `Future` se llama a `future.get()` que obliga al hilo principal a esperar a que acaben de procesarse todas las tareas encoladas. De lo contrario, podríamos pasar al siguiente `size` (del bucle inicial) sin asegurarse de que todas las partes están ordenadas. Una vez completado todos los bucles se llama a `executor.shutdown()`, que espera a que, una vez terminen de ejecutarse todas las tareas encoladas anteriormente, cierra la piscina `executor` y libera los hilos.

Al igual que en el MSRP se ha tratado de establecer un umbral, para en este caso particular cambiar en ese punto a la versión iterativa serial (MSIS). Los resultados del test preliminar (Figura 23), desarrollados bajo mismo procedimiento que el anterior, arrojan que no existe tamaño de colección alguno para el cual esta implementación (MSIP) sea más rápida que el MSIS.

Tamaño	Tiempo MSIS (ms)	Tiempo MSIP (ms)
2	0,004	0,240
4	0,005	0,432
8	0,004	0,649
16	0,005	0,645
32	0,006	0,504
64	0,004	0,504
128	0,006	0,605
256	0,011	0,685
512	0,024	0,663
1024	0,051	0,624
2048	0,111	0,931
4096	0,239	1,513
8192	0,509	2,686
16384	1,062	5,074
32768	2,197	9,538
65536	4,448	18,727
131072	8,980	45,305
262144	18,553	78,727
524288	37,924	173,462
1048576	78,274	320,937

Tabla 2: Datos de prueba para MSIS y MSIP sin umbral

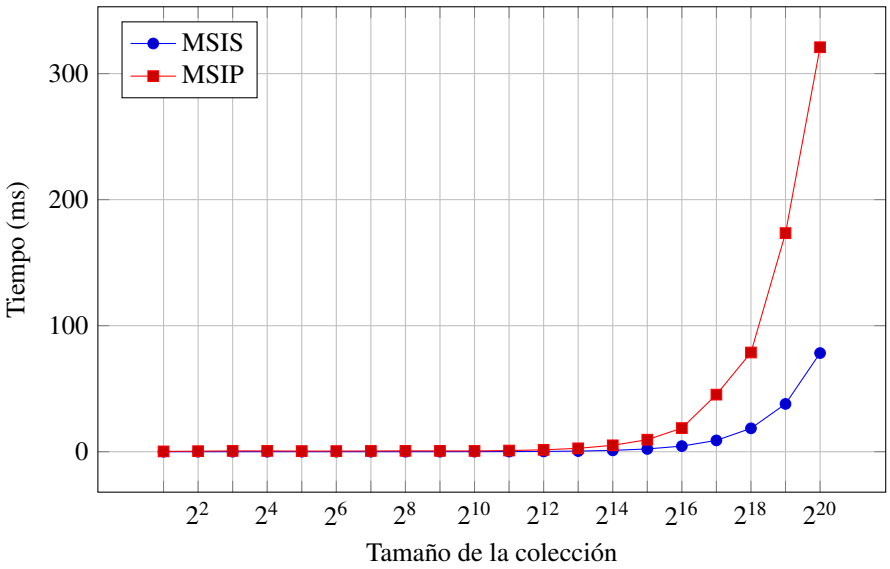

Figura 23: Datos de prueba para MSIS y MSIP sin umbral

3. Experimentación

Para la realización de una comparación empírica de los cuatro algoritmos (MSRS, MSIS, MSRP, MSIP) se toman 50 muestras del tiempo de ejecución para n tamaños de colección de entrada para cada algoritmo. 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) e IDE de ejecución IntelliJ Idea Community Edition 2023.3.3.

3.1. Procedimiento

1. Se generan las longitudes crecientes de las colecciones de entrada siguiendo la secuencia $n = 10$, 45, 100, 450, 1000... 10^8. El máximo tamaño es 10^8 ya que a partir de este tamaño la memoria del computador empleado es insuficiente.

2. Para cada tamaño n se genera una colección mediante la clase `SplittableRandom` de Java, que en este caso genera colecciones de tipo `int` con valores pseudoaleatorios extraídos de una distribución uniforme.[28] Particularmente, el generador produce números del 0 al 999 y siempre emplea la misma semilla (6180339887), por tanto, en todas las muestras la secuencia de elementos a ordenar es la misma. Esto se hace para que la variación del tiempo de ejecución a lo largo del tamaño de entrada creciente se deba a la naturaleza misma del algoritmo y no influya el estado inicial del arreglo ya que son «idénticos».

3. Para cada toma de muestra se instancia un nuevo arreglo y se copia en este el arreglo generado anteriormente; de lo contrario en la siguiente muestra se estaría ordenando un arreglo ya ordenado. Para la colección auxiliar se sigue el mismo procedimiento.

4. 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 de exactitud.[29] Cada ejecución se realizan entre una variable `long startTime` y `long endTime`. El tiempo de ejecución es la diferencia entre estas.

5. Después se elimina el peor y mejor tiempo de entre las 50 ejecuciones, quedando así 48.

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

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

6. En el caso de los algoritmos paralelos se establece un mismo número de hilos para cada piscina para que la comparación sea justa. Concretamente, `parallelismLevel=10` ya que el computador empleado consta de 12 hilos y se reservan 2 en caso de que se ejecute algún proceso en segundo plano que los necesite.

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

4. Discusión de resultados

En la Tabla 3, se presentan los tiempos de ejecución medios en milisegundos (ms) para los cuatro algoritmos de ordenación: MSRS, MSIS, MSRP y MSIP, en función del tamaño de la colección de datos. Se han agrupado los tamaños de colección en: pequeños (10 – 1000), medianos (4500 – 450.000) y grandes (10^6 – 10^8). Nótese que en el caso del MSIP para 10^8 no se podido mensurar el tiempo de ejecución en tanto que al ejecutar `sort()` se ha producido una excepción `OutOfMemoryError` indicando que Java no constaba de suficiente espacio en el *heap* para colocar un objeto.[30]

Tamaño	Tiempo MSRS (ms)	Tiempo MSIS (ms)	Tiempo MSRP (ms)	Tiempo MSIP (ms)
10	0,005	0,009	0,217	0,842
45	0,008	0,013	0,227	0,654
100	0,006	0,009	0,230	0,629
450	0,024	0,023	0,265	0,704
1.000	0,054	0,049	0,330	0,893
4.500	0,275	0,247	0,530	1,762
10.000	0,636	0,592	0,936	3,204
45.000	3,067	2,774	2,137	13,043
100.000	6,855	6,301	2,765	33,804
450.000	32,441	30,049	7,527	134,608
1.000.000	74,761	68,275	14,576	322,416
4.500.000	354,405	332,042	62,903	1.498,327
10.000.000	804,113	749,785	142,320	3.200,823
45.000.000	3.825,093	3.570,820	678,702	15.648,251
100.000.000	8.534,543	7.889,013	1.535,203	–

Tabla 3: Media de los tiempos de ejecución en ms

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

4.1. Rendimiento general

En la Figura 24 se observa que a partir del tamaño 10^6 el comportamiento de los algoritmos difiere significativamente: mientras que los algoritmo seriales (MSRS y MSIS) crecen en tiempo de ejecución de forma similar, el tiempo del MSRP crece más lentamente y el MSIP se ralentiza rápidamente.

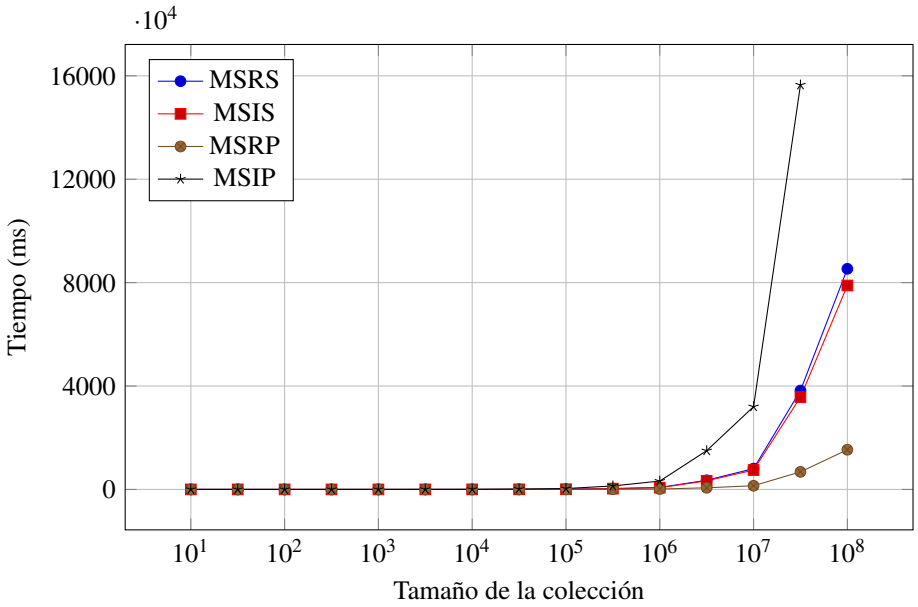

Figura 24: Tiempos medios de ejecución en ms para el MSRS, MSIS, MSRP y MSIP

En la Figura 25 se observa que para tamaños pequeños el MSIS es sustancialmente más lento que el MSRS: un 80 % más lento para un arreglo de longitud 10 por ejemplo. A partir del tamaño 1000 el MSIS torna más rápido que el MSRS para los tamaños que siguen, en torno un 8 % más rápido que el MSRS.

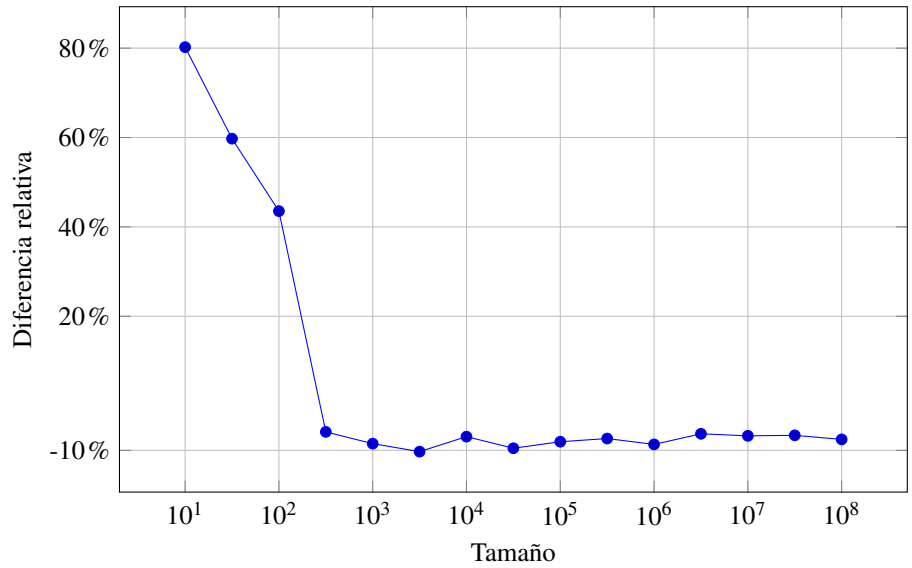

Figura 25: Porcentaje de diferencia entre los tiempos del MSIS respecto el MSRS

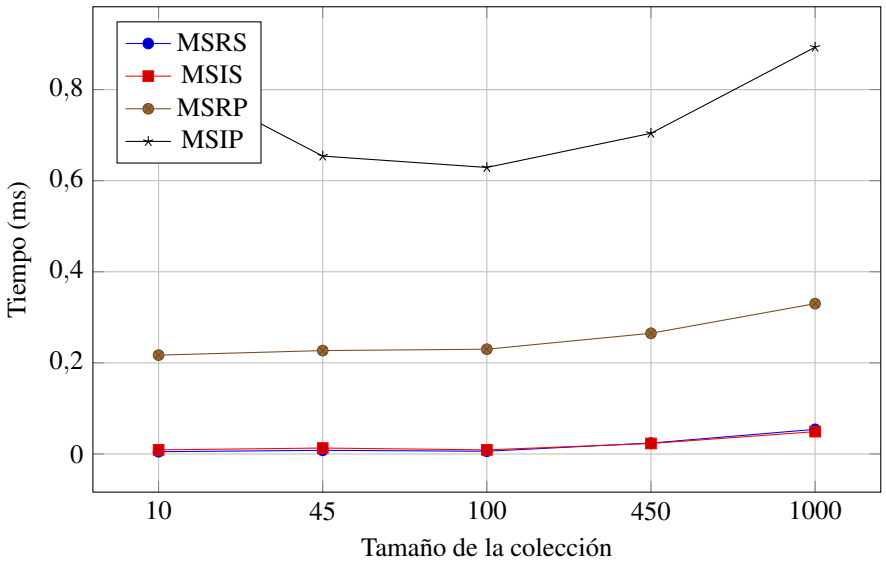

Figura 26: Tiempos medios de ejecución en ms para tamaños pequeños

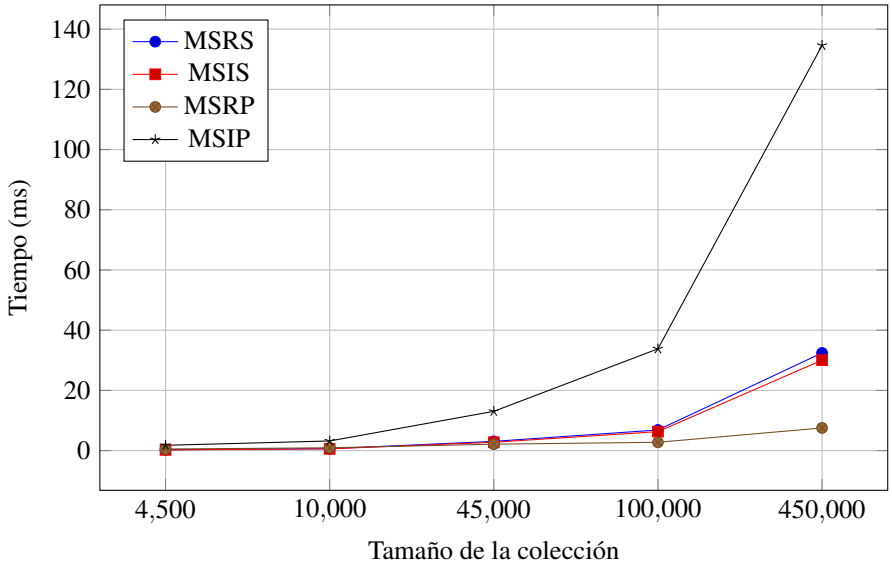

Figura 27: Tiempos medios de ejecución en ms para tamaños medianos

4.2. MSRS (Recursivo Serial)

- Ventajas: es la implementación más sencilla y común.
- Rendimiento: es eficiente para tamaños pequeños (Figura 26) y medianos (Figura 27) pero su desempeño se degrada sustancialmente para tamaños grandes, ya que se produce una sobrecarga del *call stack* al realizarse un número de llamadas a la función `sort()` que aumenta logarítmicamente.

4.3. MSIS (Iterativo Serial)

- Ventajas: evita la sobrecarga del *call stack* al no haber llamadas recursivas.

- Desventajas: implementación un poco más compleja y difícil de *debuggear*
- Rendimiento: consta de un rendimiento ligeramente mejor que el MSRS para tamaños medianos y grandes (Figura 25) pero también disminuye para tamaños grandes (Figura 24). En el caso de los tamaños pequeños el MSRS podría ser más eficiente que el MSIS porque inicialmente el pila de llamadas es pequeña y supone un costo de gestión inferior que el costo de manejo de bucles anidados del MSIS. En cambio para tamaños grandes, la profundidad de la recursión aumenta tanto que torna ineficiente en comparación al control de bucles anidados.

4.4. MSRP (Recursivo Paralelo)

- Ventajas: la concurrencia permite mayor rendimiento para colecciones grandes.
- Desventajas: la gestión de hilos introduce una sobrecarga adicional.
- Rendimiento: consta de un rendimiento excelente para colecciones medianas y grandes (Figura 27 y Figura 24), en tanto que el costo de gestión de hilos no compensa para colecciones pequeñas, donde es más eficiente usar solo un núcleo lógico del procesador que tener que controlar diez.

4.5. MSIP (Iterativo Paralelo)

- Desventajas: la gestión de hilos y la anidación de introduce una sobrecarga marcadamente mayor.
- Rendimiento: es ineficiente para cualquier tamaño, sea pequeño, mediano o grande.

Aunque las cuatro implementaciones muestran un comportamiento asimptótico linea-logarítmico $O(n \log n)$ (Véase la Figura 28), la introducción de mecanismos de concurrencia, concretamente la `ForkJoinPool` y `ExecutorService`, han aumentado notablemente la eficiencia en términos de tiempo de ejecución del MSRP.

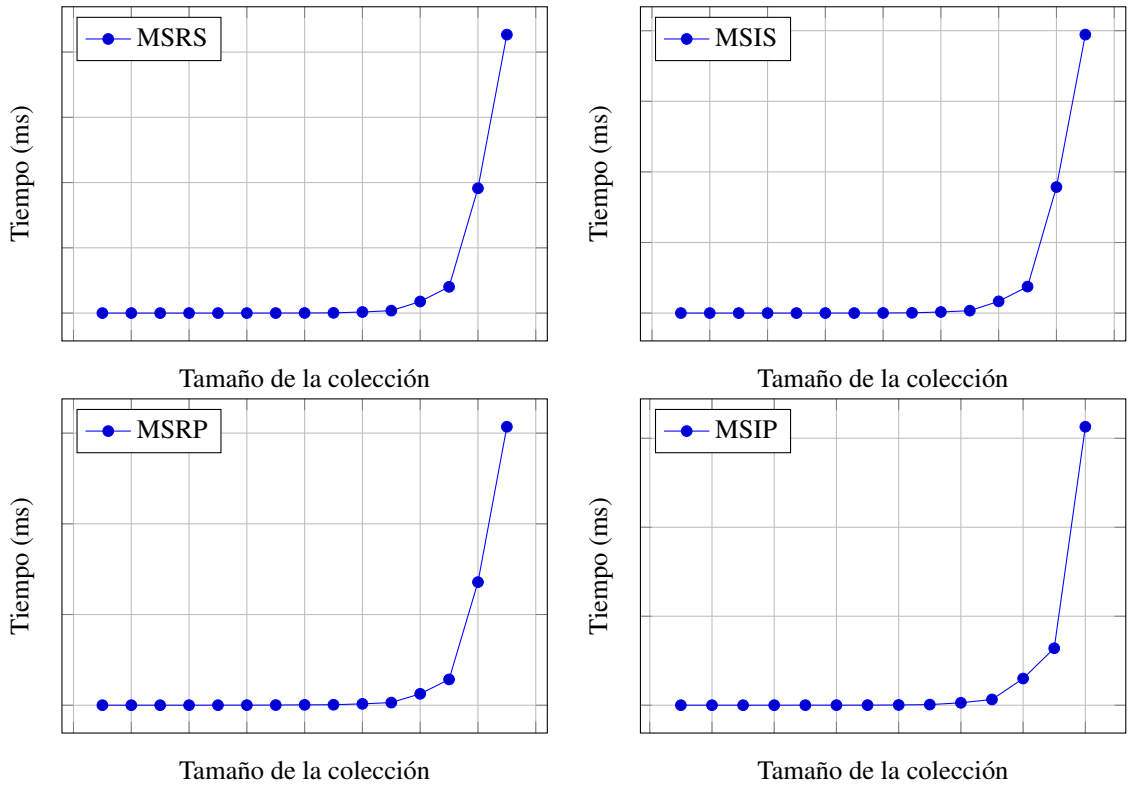

Figura 28: Comportamiento asintótico de los cuatro algoritmos

5. Conclusión

La presente monografía ha permitido evaluar el impacto de las técnicas de concurrencia en el algoritmo *Merge Sort*, comparando implementaciones seriales y sus contrapartes paralelas. Por un lado, el análisis teórico inicial ha previsto que el comportamiento asintótico, linealogarítmico en este caso, es independiente de la inserción de técnicas de concurrencia. Por otro lado, la experimentación empírica ha mostrado que, si bien la ejecución paralela mediante la `forkJoinPool` (MSRP) mejora significativamente el rendimiento en conjuntos grandes, el algoritmo iterativo paralelo (MSIP) no supera a su análogo serial debido a la sobrecarga de gestión concurrente.

Los hallazgos permiten concluir que la concurrencia optimiza el *Merge Sort* en la medida de que se encuentre un balance óptimo entre el costo de gestión de hilos y la ganancia por paralelización. Además de la importancia de considerar enfoques distintos para cada tamaño de entrada requerido.

Referencias

Bhargava, A. Y. (2016). *Grokking algorithms*. Manning Publications. https://www.manning.com/books/grokking-algorithms

Blumofe, R. D., y Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *J ACM*, *46*(5), 720–748. http://supertech.csail.mit.edu/papers/ft_gateway.pdf doi: 10.1145/324133.324234

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

Ching, J., Phoon, K.-K., y Shuku, T. (2023). *Uncertainty, modeling, and decision making in geotechnics*. CRC Press.

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

Correa, J. (2024). *Big o notation o notación big o: Todo lo que necesitas saber 2024.* https://developero.io/blog/big-o

DataCamp. (2024). *Notación big o y guía de complejidad temporal: Intuición y matemáticas.* https://www.datacamp.com/es/tutorial/big-o-notation-time-complexity

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

Engle, S. (2022). *Thread pools and work queues.* https://usf-cs272-spring2022.github.io/files/Thread%20Pools%20and%20Work%20Queues.pdf (CS 272 Software Development, Department of Computer Science, University of San Francisco, Contact: sjengle@cs.usfca.edu)

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.

Jenkov, J. (2024). *Java forkjoinpool.* https://jenkov.com/tutorials/java-util-concurrent/java-fork-and-join-forkjoinpool.html

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

Kumar, R. (2024). *A deep dive into java's forkjoinpool mechanics.* https://medium.com/@reetesh043/a-deep-dive-into-javas-forkjoinpool-mechanics-556f82d160fb (Medium, Contact: reetesh043@medium.com)

Landau, E. (1909). *Handbuch der lehre von der verteilung der primzahlen* (Vol. 1). Leipzig: B. G. Teubner. https://archive.org/details/handbuchderlehre01landuoft

Levitin, A. (2012). *Introduction to the design & analysis of algorithms* (3rd ed.). Upper Saddle River, New Jersey: Pearson Education, Inc., publishing as Addison-Wesley.

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

Mohan, D., y Kapoor, P. (2025). *Priori vs. posteriori analysis: Deep dive.* https://dmj.one/edu/su/course/csu083/theory/priori-posteriori-analysis. (CSU083 (Design and Analysis of Algorithm), Shoolini University)

Molluzzo, J., y Buckley, F. (1997). *A first course in discrete mathematics*. Waveland Press. https://books.google.es/books?id=CdF0DQAAQBAJ

Oracle. (2025). *Executorservice (java platform se 8).* https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html (Accedido: 19-ene-2025)

Pandey, R. C. (2008). Study and comparison of various sorting algorithms. *Computer Science and Engineering*.

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.

Sipser, M. (1997). *Introduction to the theory of computation*. Boston, MA: PWS Publishing. (Definition 7.2)

Skiena, S. S. (2008). *The Algorithm Design Manual*. Springer London. 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). https://docs.oracle.com/javase/8/docs/api/java/util/SplittableRandom.html

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

System (Java Platform SE 8). (2024b, 9). https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks002.html

Todas las figuras han sido creadas por el candidato.

Anexos

Anexo A. Código de las implementaciones

MSRS (*Merge Sort* Recursivo Serial)

```java
package Implementaciones;

public class MSrecursivoSerial {
   public static void sort(int[] arr, int[] aux, int left, int right) {
      //Caso base
      if (left >= right) return;

      //Calcular la mitad
      int mid = left + (right - left) / 2; //Evitar desbordamiento de Integer.MAX_VALUE

      //Llamada recursiva
      sort(arr, aux, left, mid);
      sort(arr, aux,mid+1, right);

      //Unir las dos partes
      merge(arr, aux, left, mid, right);
   }

   private static void merge(int[] arr, int[] aux, int left, int mid, int right) {
      for (int i = left; i <= right; i++) aux[i] = arr[i];

      int i = left;
      int j = mid + 1;
      int k = left;

      while (i <= mid && j <= right) arr[k++] = (aux[i] <= aux[j])? aux[i++] : aux[j++];

      while (i <= mid) arr[k++] = aux[i++];
   }

}
```

MSIS (*Merge Sort* Iterativo Serial)

```java
package Implementaciones;

public class MSiterativoSerial {
   public static void sort(int[] arr, int[] aux) {
      int length = arr.length;

      // Subarreglos de tamano 1, 2, 4, 8, ... hasta n/2
      for (int size = 1; size < length; size *= 2) {
         for (int left = 0; left < length - size; left += 2 * size) {
            int mid = left + size - 1;
            int right = Math.min(left + 2 * size - 1, length - 1);
            merge(arr, aux, left, mid, right);
         }
      }
   }

   private static void merge(int[] arr, int[] aux, int left, int mid, int right) {
      for (int i = left; i <= right; i++) aux[i] = arr[i];

      int i = left;
      int j = mid + 1;
      int k = left;

      while (i <= mid && j <= right) arr[k++] = (aux[i] <= aux[j])? aux[i++] : aux[j++];

      while (i <= mid) arr[k++] = aux[i++];
   }

}
```

MSRP (*Merge Sort* Recursivo Paralelo)

```java
package Implementaciones;

import java.util.concurrent.RecursiveAction;

public class MSrecursivoParalelo extends RecursiveAction {
   //Atributos constantes
   private final int[] arr, aux;
   public static int THRESHOLD = 32768;
   //Atributos dinamicos
   private final int right, left;

   public MSrecursivoParalelo(int[] arr, int[] aux, int left, int right){
      ///Paso por referencia: constantes
      this.arr = arr;
      this.aux = aux;

      //Dinamicos
      this.left = left;
      this.right = right;
   }
   @Override
   protected void compute() {
      int length = (right + 1 - left);
      if (length <= THRESHOLD) {
         MSrecursivoSerial.sort(arr, aux, left, right);
      } else {
         int mid = left + (right - left) / 2;

         final MSrecursivoParalelo Left = new MSrecursivoParalelo(arr, aux, left, mid);
         final MSrecursivoParalelo Right = new MSrecursivoParalelo(arr, aux, mid+1, right);

         invokeAll(Left, Right);

         merge(arr, aux, left, mid, right);
      }
   }

   private static void merge(int[] arr, int[] aux, int left, int mid, int right) {
      for (int i = left; i <= right; i++) aux[i] = arr[i];

      int i = left;
      int j = mid + 1;
      int k = left;

      while (i <= mid && j <= right) arr[k++] = (aux[i] <= aux[j])? aux[i++] : aux[j++];

      while (i <= mid) arr[k++] = aux[i++];
   }

}
```

MSIP (*Merge Sort* Iterativo Paralelo)

```java
package Implementaciones;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;

public class MSiterativoParalelo {
   private static final int THRESHOLD = 8192;

   public static void sort(int[] arr, int[] aux, ExecutorService executor) {
      int length = arr.length;
      List<Future<?>> futures = new ArrayList<>();

      for (int size = 1; size < length; size *= 2) {

         for (int left = 0; left < length - size; left += 2 * size) {
            int mid = left + size - 1;
            int right = Math.min(left + 2 * size - 1, length - 1);
            int finalLeft = left; //El resto son efectivamente finales
            futures.add(executor.submit(() -> merge(arr, aux, finalLeft, mid, right)));
         }
         for (Future<?> future : futures) {
            try {
               future.get();
            } catch (Exception e) {
               e.printStackTrace();
            }
         }
         futures.clear();

      }
      executor.shutdown();
   }

   private static void merge(int[] arr, int[] aux, int left, int mid, int right) {
      for (int i = left; i <= right; i++) aux[i] = arr[i];

      int i = left;
      int j = mid + 1;
      int k = left;

      while (i <= mid && j <= right) arr[k++] = (aux[i] <= aux[j])? aux[i++] : aux[j++];

      while (i <= mid) arr[k++] = aux[i++];
   }

}
```

Anexo B. Código del *benchmark*

```
package Implementaciones;

import java.io.FileWriter;
import java.io.IOException;
import java.util.SplittableRandom;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.ForkJoinPool;

public class BenchmarkTiempo {

   private static final int[] arraySizes = generateSizes();
   private static final int numtrials = 50;
   private static long[] averages, minValues, maxValues;
   private static long[][] rawTrials;
   private static final String csvName = "nombre_del_benchmark_de_tiempo.csv";
   private static final int parallelismLevel = 10;

   public static void main(String[] args) throws IOException {
      averages = new long[arraySizes.length];
      rawTrials = new long[arraySizes.length][numtrials];
      minValues = new long[arraySizes.length];
      maxValues = new long[arraySizes.length];

      for (int i = 0; i< arraySizes.length; i++) {
         int size = arraySizes[i];
         int[] originalArray = generateArray(size);

         long[] times = new long[numtrials];
         long totalTime = 0;

         for (int trial = 0; trial < numtrials; trial++) {
            //Copia del array ya que es paso por referencia
            int[] arrayCopy = originalArray.clone();

            //Array auxiliar
            int[] aux = new int[size];

            //En el caso de las implementaciones paralelas:
            //MSRP
            //ForkJoinPool forkJoinPool = new ForkJoinPool(parallelismLevel);
            //MSrecursivoParalelo task = new MSrecursivoParaleloCustomThresh(arrayCopy,
                aux, 0, size-1);
            //MSIP
            //ExecutorService executorService =
                Executors.newFixedThreadPool(parallelismLevel);


            System.gc(); //Llamada al Garbage Collector
         ...
```

(Continua en la siguiente página)

```
            //Benchmark
            long startTime = System.nanoTime();
               /*Algoritmo a testear
                  * MSrecursivoSerial.sort(arrayCopy, aux, 0, size-1);
                  * MSiterativoSerial.sort(arrayCopy, aux);
                  * forkJoinPool.invoke(task);
                  * MSiterativoParalelo.sort(arrayCopy, aux, executorService);
               */
            long endTime = System.nanoTime();

            long time = endTime - startTime;
            times[trial] = time;
            totalTime += time;
            rawTrials[i][trial] = time;
         }

         //Encontrar maximo y minimo
         long max = Long.MIN_VALUE, min = Long.MAX_VALUE;
         for (long l : times) if (l > max) max = l;
         for (long p : times) if (p < min) min = p;

         //Calcular el promedio
         long average = (totalTime - min - max) / (numtrials - 2);
         averages[i] = average;
         minValues[i] = min;
         maxValues[i] = max;

         System.out.println("Size: "+size+" \t\t Time: "+average);
      }
      writeResults();
   }

   public static void writeResults() throws IOException {
      try (FileWriter writer = new FileWriter(csvName)) {
         for (int i = 0; i<arraySizes.length; i++) {
            //Escribir etiquetas
            writer.append(";").append(String.valueOf(arraySizes[i]));
            //Escribir datos
            writer.append(";").append(String.valueOf(averages[i]));
            writer.append(";").append(String.valueOf(minValues[i]));
            writer.append(";").append(String.valueOf(maxValues[i]));

            for (int j = 0; j<numtrials; j++){
               writer.append(";").append(String.valueOf(rawTrials[i][j]));
            }
            writer.append("\n");
         }
      }
   }

   public static int[] generateArray(int size) {
      int[] array = new int[size];
      long seed = 6180339887L;
      SplittableRandom random = new SplittableRandom(seed);
      //Todos los arrays constan de una misma secuencia
      for (int i = 0; i < array.length; i++) {
         array[i] = random.nextInt(1000); //De 0 a 999
      }
      return array;
   }

   private static int[] generateSizes() {
      return new int[]
         {10, 45, 100, 450, 1_000, 4_500, 10_000, 45_000, 100_000, 450_000, 1_000_000,
             4_500_000, 10_000_000, 45_000_000, 100_000_000};
   }
}
```