UNIVERSIDAD NACIONAL DE INGENIERÍA
FACULTAD DE CIENCIAS

PRÁCTICA CALIFICADA 1

AUTORES:

Lezama Verastegui, Dagmar Nicole 20204096K

Perez Cueva, Chandler Steven 20200605H

CURSO:

CC4P1 Pogramación Concurrente

LABORATORIO 01

Lima - Perú

Abstract

Este informe presenta el análisis y la implementación de dos algoritmos en Java: Counting Sort y la Transformada Rápida de Fourier (FFT). Ambos algoritmos se adaptaron para ejecutarse de manera paralela, aprovechando el paralelismo de hilos. Counting Sort, conocido por su eficiencia en la ordenación de datos, se compara con la función parallelSort proporcionada por Java. Por otro lado, la implementación paralela de FFT se examina en términos de eficiencia en la Transformada de Fourier Discreta.
El estudio incluye una evaluación de la complejidad en memoria y el rendimiento de estos algoritmos, con énfasis en la influencia del número de hilos utilizados para diferentes tamaños de entrada. Se presentan resultados comparativos detallados, respaldados por análisis de hardware.

Keywords: Counting Sort, Transformada Rápida de Fourier, Algoritmos paralelos, Complejidad.

Contents

---

1 Introducción

El presente informe aborda el desarrollo y análisis de dos algoritmos implementados en el lenguaje de programación Java, aprovechando el poder de la concurrencia mediante el uso de hilos para optimizar su eficiencia. Los algoritmos en cuestión son el Counting Sort y la Transformada Rápida de Fourier (FFT), ambos adaptados para ejecutarse de forma paralela.

El primero de ellos es una implementación paralela del algoritmo de ordenación Counting Sort, cuya característica distintiva radica en su complejidad temporal de $\mathcal{O}(n)$. Este enfoque será comparado con la función `parallelSort` proporcionada por Java, que también aprovecha el paralelismo para ordenar conjuntos de datos. El segundo algoritmo es una implementación paralela de la Transformada Rápida de Fourier (FFT), una técnica ampliamente reconocida por su eficacia en el cálculo de la Transformada de Fourier Discreta.
En ambos casos, se realizará un minucioso análisis de la complejidad en memoria inherente a los algoritmos y se llevará a cabo una rigurosa comparación entre sus implementaciones secuenciales y paralelas.

Además, para proporcionar una visión más clara de la paralelización de los algoritmos, se presentarán gráficos descriptivos de los resultados obtenidos, los cuales se examinarán detenidamente, incluyendo la descripción de los componentes del hardware que influyeron en los resultados de las pruebas.
Con el fin de ofrecer una evaluación completa, se llevará a cabo una comparativa entre diferentes tamaños de

entrada (n) y se medirá con precisión el tiempo de ejecución de estos algoritmos. A partir de estos resultados, se extraerán conclusiones fundamentadas sobre el número óptimo de hilos a utilizar para un tamaño de entrada dado (n).
Este informe tiene como objetivo proporcionar una visión detallada del rendimiento y la eficiencia de estos dos algoritmos cuando se implementan utilizando un enfoque paralelo en Java.

2 Marco Teórico

2.1 Ordenamiento Counting Sort

Counting Sort, propuesto por Harold Seward en 1954, es un algoritmo de ordenamiento no comparativo que se distingue por su enfoque en contar la frecuencia de los elementos presentes en la lista a ordenar.
En lugar de recurrir a comparaciones entre los elementos, Counting Sort evita este proceso y se aprovecha de las inserciones y eliminaciones de tiempo constante $O(1)$ en la matriz. Este logro se materializa a través de un método que se centra en el mapeo de frecuencias para determinar la posición adecuada de cada elemento en la lista ordenada. Aunque demuestra una alta eficiencia en términos de tiempo cuando se trabaja con rangos de valores pequeños, su aplicabilidad puede verse limitada en el caso de valores extremadamente grandes debido a las demandas de asignación de memoria.

2.2 Transformada Rápida de Fourier (FFT)

2.2.1 Análisis de Fourier

El análisis que sigue, originado en el estudio de las series de Fourier, se enfoca en la representación de una función continua mediante una serie potencialmente infinita de funciones seno y coseno. En este contexto, podemos establecer lo siguiente:

$$f(x) = \frac{1}{2}a_0 + \sum_{n=1}^{\infty} a_n \cos(nx) + \sum_{n=1}^{\infty} b_n \sin(nx)$$

Donde n $= 1, 2, 3 \ldots$

$$a_0 = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)dx \quad a_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(nx)dx$$
$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin(nx)dx$$

Los números a_n y b_n se llaman coeficientes de Fourier de f. Por lo tanto, la suma infinita $f(x)$ se denomina serie de Fourier de f. Las series de Fourier se pueden generalizar a números complejos y, posteriormente, se generalizan aún más para derivar la Transformada de Fourier.

2.2.2 Transformada de Fourier

La Transformada de Fourier se define mediante la siguiente expresión:
Transformada de Fourier Directa:

$$F(k) = \int_{-\infty}^{\infty} f(x)e^{-2\pi ikx}dk$$

Transformada Inversa de Fourier:

$$f(x) = \int_{-\infty}^{\infty} F(k)e^{2\pi ikx}dk$$
$$\text{Nota:} \quad e^{xi} = \cos(x) + i\sin(x)$$

Esta ecuación define $F(k)$, la Transformada de Fourier de $f(x)$. $f(x)$ se refiere a una función en función del tiempo, mientras que $F(k)$ es una función en función de la frecuencia.

La Transformada de Fourier en realidad mapea un dominio de tiempo (serie) en el dominio de frecuencia (serie). Por lo tanto, a menudo se le llama el dominio de frecuencia. La Transformada Inversa de Fourier

mapea el dominio de frecuencias nuevamente al dominio de tiempo correspondiente. Estas dos funciones son inversas entre sí.

Las ideas del dominio de frecuencia son importantes en muchas áreas de aplicación, incluyendo audio, procesamiento de señales y procesamiento de imágenes. Por ejemplo, el análisis espectral se utiliza ampliamente para analizar el habla, comprimir imágenes y buscar periodicidades en una amplia variedad de datos en biología, física, entre otros campos. En particular, el algoritmo de compresión JPEG, que se utiliza ampliamente y es muy efectivo, utiliza una versión de la Transformada Coseno de Fourier para comprimir los datos de la imagen.

Sin embargo, la Transformada de Fourier no es adecuada para la computación en máquinas porque requiere considerar un número infinito de muestras. Existe un algoritmo llamado Transformada de Fourier Discreta, que se modifica a partir de la Transformada de Fourier y se puede utilizar para la computación en máquinas.

2.2.3 Transformada discreta de Fourier (DFT)

La Transformada de Fourier Discreta es un tipo específico de Transformada de Fourier. Requiere una función de entrada que sea discreta y cuyos valores no nulos tengan una duración finita, esta puede ser una secuencia finita de números reales o complejos, por lo que la DFT es ideal para procesar información almacenada en computadoras. Sin embargo, si los datos son abundates, estos deben muestrearse cuando utilicemos la DFT. Existe la posibilidad de que si el intervalo de muestreo es demasiado grande, pueda causar el efecto de aliasing, sin embargo, si es demasiado estrecho, el tamaño de los datos digitalizados podría aumentar.

Dada una secuencia de f_n para $k = 0, 1, 2, \ldots, \mathrm{N} - 1$, se transforma en la secuencia de X_k mediante la DFT de acuerdo con la fórmula:

$$X_k = \sum_{n=0}^{N-1} f_n e^{\frac{-2\pi i}{N} kn}$$

La Transformada Inversa de Fourier Discreta se expresa como:

$$f_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi i}{N} kn}$$

Donde $w_n = e^{\frac{2\pi i}{N}} = \cos(2\pi/N) + i\sin(2\pi/N)$ es una raíz primitiva N-ésima de la unidad.

2.2.4 Transformada rápida de Fourier

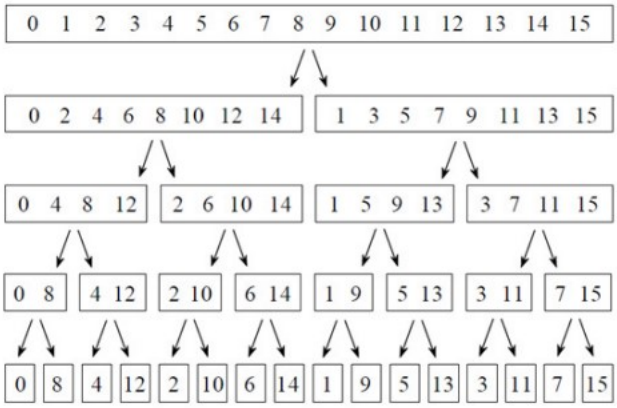
La complejidad temporal de la Transformada de Fourier Discreta (DFT) para n muestras es de $O\left(n^2\right)$ si se implementa de manera directa. Por lo tanto, en la práctica, la utilización de la DFT no resulta la opción más eficiente. Sin embargo, existe un algoritmo mejorado conocido como la Transformada Rápida de Fourier (FFT), que produce el mismo resultado que la DFT, pero de manera mucho más eficiente. La FFT se basa en una estrategia de "divide y vencerás" y puede calcular n muestras en $\mathrm{O(n \log n)}$ tiempo. La principal diferencia entre la DFT y la FFT radica en su velocidad de cálculo, siendo la FFT considerablemente más rápida, lo que la convierte en una versión optimizada de la DFT.

La idea subyacente en la FFT consiste en dividir repetidamente una secuencia DFT de N muestras en dos subsecuencias, separando los índices pares e impares en cada iteración. En el caso de que N sea una potencia de 2, este proceso continúa hasta que cada subsecuencia contiene únicamente un elemento. Como resultado de esta división, el orden de los índices se reorganiza en un patrón inverso con respecto al índice original. (Ver Figura 1)

De modo que, podemos reescribir la función DFT de la siguiente manera:

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} f_n w_N^{-kn} = \sum_{n=0}^{N/2-1} f_{2n} w_N^{-2kn} + \sum_{n=0}^{N/2-1} f_{2n+1} w_N^{-(2k+1)n} \\ &= \sum_{n=0}^{N/2-1} f_n^{even} w_{N/2}^{-kn} + w_N^k \sum_{n=0}^{N/2-1} f_n^{odd} w_{N/2}^{-kn} \end{aligned}$$

Entonces, por ejemplo, con 16 puntos, tendremos $\log_2$ N pasos, que son 4 pasos, y cada uno requiere N operaciones.
Finalmente, la complejidad temporal es de $O(N \log N)$.



Decimal	*Binary*
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Decimal	*Binary*
0	0000
8	1000
4	0100
12	1100
2	0010
10	1010
6	0100
14	1110
1	0001
9	1001
5	0101
13	1101
3	0011
11	1011
7	0111
15	1111

.

Figure 1: División de un arreglo según la FFT

3 Metodología

3.0.1 Counting Sort

A continuación, se define la estructura que sigue el código implementado:

1. Generar un arreglo de números aleatorios: El primer paso es generar un arreglo de números enteros aleatorios. El tamaño del arreglo se puede especificar como un parámetro del algoritmo.
2. Dividir el arreglo en H subarreglos iguales: El segundo paso es dividir el arreglo en H subarreglos iguales. El número de subarreglo H se puede especificar como un parámetro del algoritmo.
3. Crear un hilo para cada subarreglo: El tercer paso es crear un hilo para cada subarreglo. Cada hilo se encargará de ordenar su correspondiente subarreglo.
4. Cada hilo ordena su subarreglo de forma secuencial: El cuarto paso es que cada hilo ordena su subarreglo de forma secuencial. Para ello, utiliza el algoritmo de conteo secuencial.

5. Una vez que todos los subarreglos han sido ordenados, se fusionan para obtener el arreglo ordenado completo: El quinto y último paso es fusionar los subarreglos ordenados para obtener el arreglo ordenado completo.

El método `main()` recibe dos parámetros:

- `N`: El tamaño del arreglo de números aleatorios.
- `H`: El número de subarreglos en los que se dividirá el arreglo.

La salida del método `main()` es la siguiente:

- **Datos**: El tamaño del arreglo `N` y el número de subarreglos `H`.
- **Secuencial**: El tiempo de ejecución del algoritmo de conteo secuencial.
- **Paralelo**: El tiempo de ejecución del algoritmo de conteo paralelo.

3.0.2 Transformada rápida de Fourier

A continuación, se define la estructura que sigue el código implementado:
El código está escrito en Java y utiliza la biblioteca 'org.apache.commons.math3.complex.Complex' para facilitar la manipulación de números complejos.
Se define una clase principal llamada ParallelFFT:

1. Se define una clase interna FFTThread que representa un hilo para calcular la FFT de una porción de la señal.
2. La función fft realiza la FFT de una señal dada. Si la señal tiene solo un elemento, se devuelve tal cual.
3. Se divide la señal en partes pares e impares y se crean hilos para procesar cada parte.
4. Se inician y se esperan los hilos para que calculen sus partes de la FFT.
5. Los resultados de los hilos se combinan para obtener la FFT completa en la etapa de combinación.

Entrada: Se proporciona un ejemplo de entrada en el método main, donde se define la señal de entrada data como una matriz de números complejos.
Salida: La salida del programa muestra los resultados de la FFT, que representan la descomposición de la señal de entrada en sus componentes de frecuencia.

4 Resultados y Discusiones

Computadora 1

- Procesador: AMD Ryzen 9 4900HS with Radeon Graphics, 3000 MHz, 8 procesadores principales, 16 procesadores lógicos.
- Unidad de Estado Sólido: 1024 GB
- RAM: 16 GB a 3200 MHz.

Computadora 2

- Procesador: Procesador AMD Ryzen 7 7735HS with Radeon Graphics, 3201 Mhz, 8 procesadores principales, 16 procesadores lógicos
- Unidad de Estado Sólido: 512 GB
- RAM: 16 GB a 4800 MHz.

4.0.1 Counting Sort

El código toma un arreglo de tamaño N, lo subdivide en varios subarrays y cada subarray es tratado por un hilo distinto.
Entonces tenemos:
Entrada: $O(n)$
Subprocesos: $O(2n + k)$
Salida: $O(n)$
Total: $O(n) + O(2n + k) + O(n) = O(2n)$
Por lo tanto, la complejidad en memoria del código de Counting Sort paralelo es O(2n).

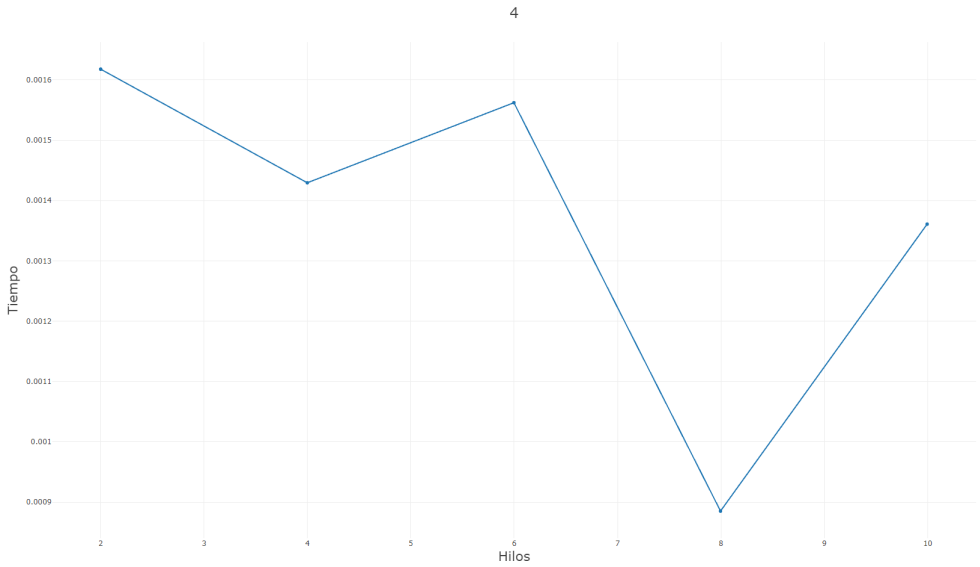


Figure 2: Gráfico h vs Tiempo con n=10

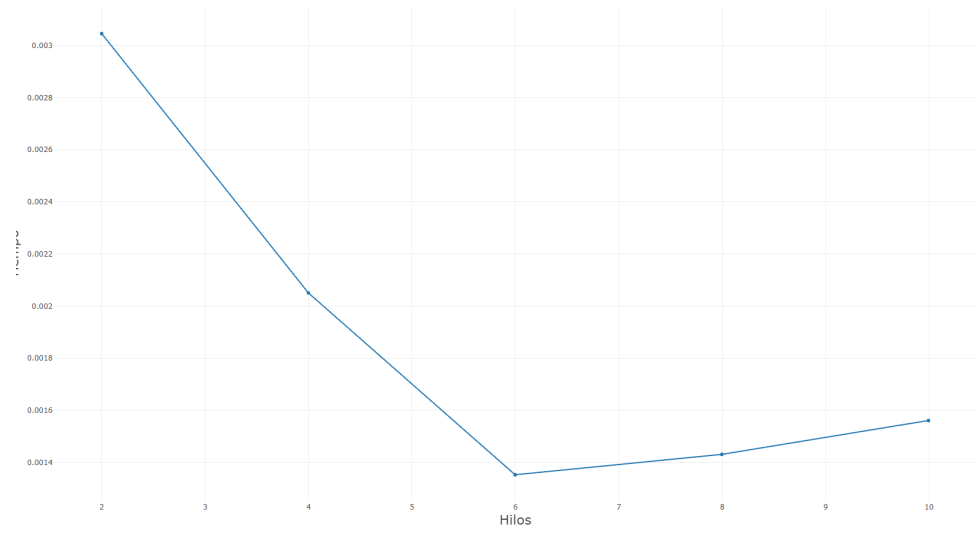


Figure 3: Gráfico h vs Tiempo con n=100

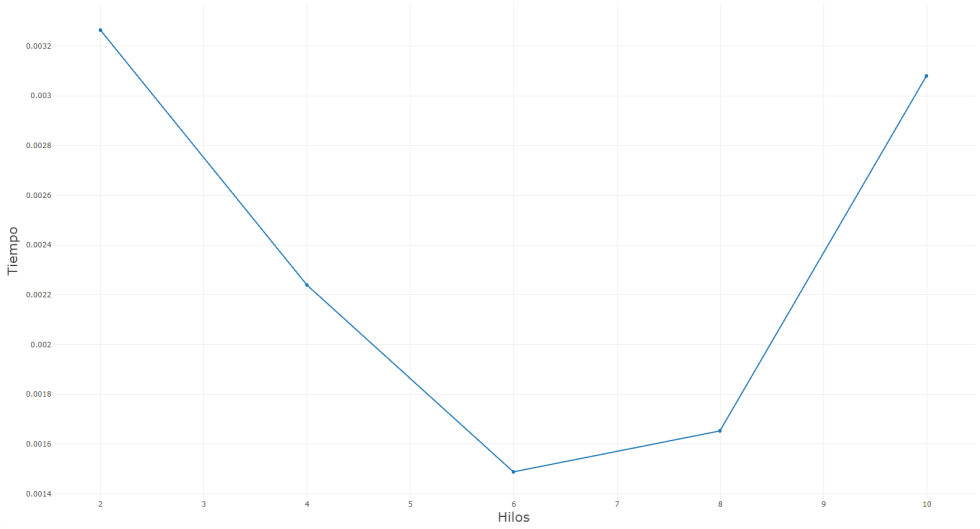


Figure 4: Gráfico h vs Tiempo con n=1000

a

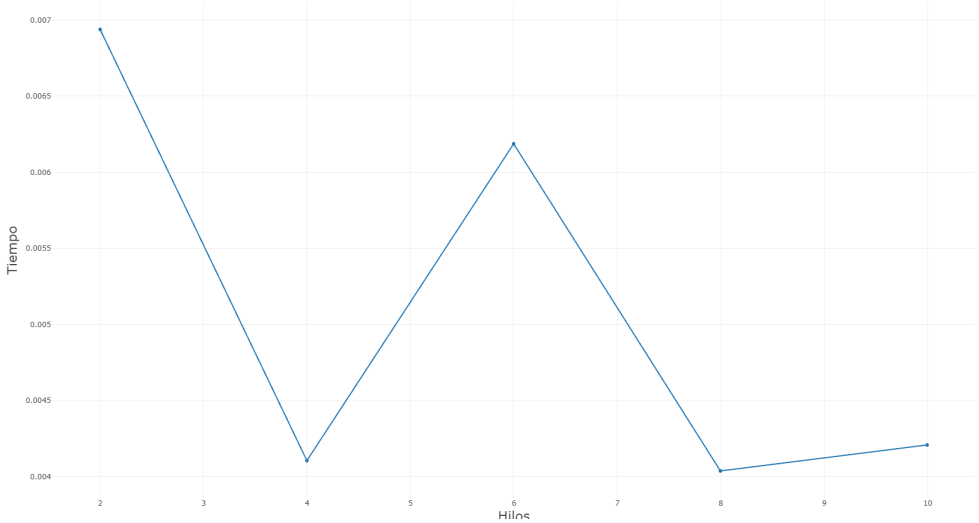


Figure 5: Gráfico h vs Tiempo con n=10000

4.0.2 Transformada rápida de Fourier

El código crea dos copias de la entrada, una para cada subproceso. Además, crea una copia de la salida, que es del mismo tamaño que la entrada.
El algoritmo empleado, también crea algunos objetos más pequeños, como objetos Complex y objetos Thread. Sin embargo, el tamaño de estos objetos es insignificante en comparación con el tamaño de las copias de la entrada y la salida.
Tomando en cuenta las consideraciones descritas superiormente, tenemos:
Entrada: $O(n)$
Subprocesos: $O(2)$
Salida: $O(n)$
Objetos pequeños: $O(1)$
Total: $O(n) + O(2) + O(n) + O(1) = O(n)$
Por lo tanto, la complejidad en memoria del código de FFT paralelo es O(n).

Se muestra el gráfico con distintos tamaños de entrada vs tiempo:

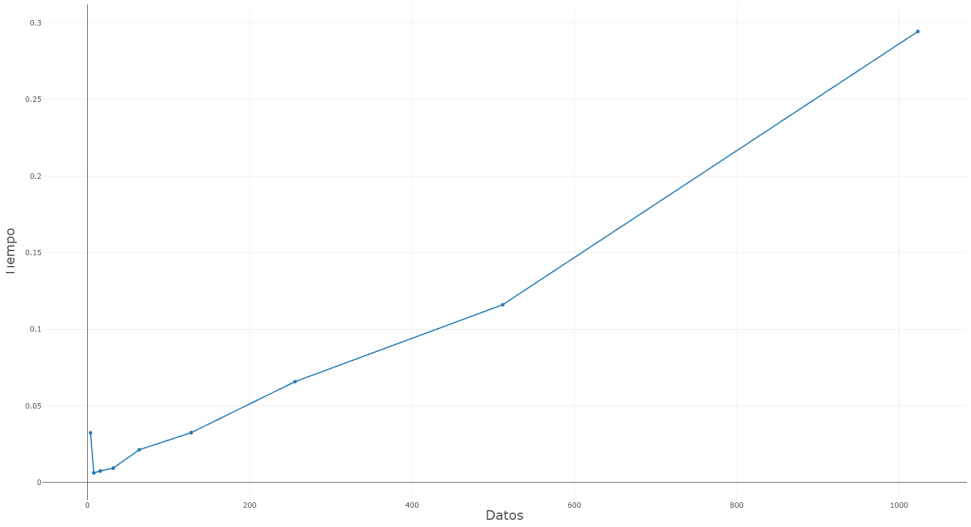


Figure 6: FFT Paralelo con distintos n

5 Conclusiones

En este informe, se han analizado dos algoritmos implementados en Java, el Counting Sort y la Transformada Rápida de Fourier (FFT), con enfoque en la optimización a través del paralelismo.
La implementación del algoritmo Counting Sort para el conteo secuencial se acercó significativamente al rendimiento del algoritmo proporcionado por Java (parallelSort) al ordenar conjuntos de datos.

Además, pesar de no representar el enfoque más óptimo, se logró paralelizar con éxito la implementación del segundo algoritmo, la Transformada Rápida de Fourier (FFT). Esto se tradujo en un procesamiento más eficiente de la Transformada de Fourier Discreta en comparación con su implementación secuencial.
Para finalizar, resaltamos lo siguiente:

1. El algoritmo implementado para el Counting Sort demostró una eficiencia considerable en términos de tiempo de ejecución, especialmente en comparación con valores pequeños. Sin embargo, su aplicabilidad puede verse limitada en conjuntos de datos extremadamente grandes debido a las demandas de asignación de memoria.

2. La Transformada Rápida de Fourier (FFT) es una técnica altamente eficiente para calcular la Transformada de Fourier Discreta, y su implementación paralela mejoró significativamente su rendimiento en comparación con la versión secuencial.

3. La paralelización de algoritmos puede llevar a mejoras sustanciales en el rendimiento en sistemas con hardware adecuado. Sin embargo, es importante evaluar el número óptimo de hilos a utilizar para un tamaño de entrada dado, ya que un exceso de hilos puede resultar en un rendimiento subóptimo debido a la sobrecarga de administración de hilos.

6 Anexo Código

```
1  import java.util.Arrays;
2  public class CountingParalelo{
3      static int N;
4      static int H;
5      static int[] arr;
6      static int[] arrParallel;
7      static int[] arrParallelJava;
8
9      public static void GenerateArray(){
10         //System.out.println("Generando arreglo...");
11         arr = new int[N];
12         arrParallel = new int[N];
13         arrParallelJava = new int[N];
```

```
14          for(int i = 0; i < N; i++){
15              int value = (int)(Math.random()*100);
16              arr[i] = value;
17              arrParallel[i] = value;
18              arrParallelJava[i] = value;
19          }
20          System.out.println("Arreglo generado");
21      }
22
23      public static void main(String[] args){
24          for(int i = 1; i < 8; i++){
25              for(int j = 1; j < 4; j++) {
26                  N = (int) Math.pow(10, i);
27                  H = (int) Math.pow(2, j);
28                  Main();
29              }
30          }
31      }
32
33      static void Main(){
34          GenerateArray();
35          for (int i = 0; i < N; i++) {
36              //System.out.print(arr[i] + " ");
37          }
38          //System.out.println();
39          System.out.println("Datos: " + N + " Hilos: " + H);
40          System.out.println("Secuencial");
41          //mide el tiempo de ejecucion
42          long startTime = System.nanoTime();
43          Sequential();
44          long endTime = System.nanoTime();
45          long duration = (endTime - startTime);
46          System.out.println("Tiempo: " + (double)duration/1_000_000_000 + " segundos");
47          for (int i = 0; i < N; i++) {
48              //System.out.print(arr[i] + " ");
49          }
50          //System.out.println();
51          CountingManager manager = new CountingManager(N, H);
52          System.out.println("Paralelo");
53          startTime = System.nanoTime();
54          manager.start();
55          try{
56              manager.join();
57              endTime = System.nanoTime();
58              duration = (endTime - startTime);
59              System.out.println("Tiempo: " + (double)duration/1_000_000_000 + " segundos");
60          }catch(InterruptedException e){
61              System.out.println("Error");
62          }
63          for (int i = 0; i < N; i++) {
64              //System.out.print(arrParallel[i] + " ");
65          }
66          //System.out.println();
67          System.out.println("Paralelo Java");
68          startTime = System.nanoTime();
69          Arrays.parallelSort(arrParallelJava);
70          endTime = System.nanoTime();
71          duration = (endTime - startTime);
72          System.out.println("Tiempo: " + (double)duration/1_000_000_000 + " segundos");
73      }
74
75      public static void Sequential() {
76          int max = Arrays.stream(arr).max().getAsInt();
77          int min = Arrays.stream(arr).min().getAsInt();
78          int k = max - min + 1;
79          int[] count = new int[k];
80          int[] out = new int[N];
81          for (int i = 0; i < N; i++) {
82              count[arr[i] - min]++;
83          }
84          for (int i = 1; i < k; i++) {
85              count[i] += count[i - 1];
```

```
        }
        for (int i = N - 1; i >= 0; i--) {
            out[count[arr[i] - min] - 1] = arr[i];
            count[arr[i] - min]--;
        }
        for (int i = 0; i < N; i++) {
            arr[i] = out[i];
        }
    }
}

class Counting extends Thread{
    int id, start, end;
    public Counting(int id, int start, int end){
        this.id = id;
        this.start = start;
        this.end = end;
    }
    public void run() {
        int N = this.end - this.start;
        int[] arr = new int[N];
        for (int i = 0; i < N; i++) {
            arr[i] = CountingParalelo.arrParallel[i + this.start];
        }
        int min = Arrays.stream(arr).min().getAsInt();
        int max = Arrays.stream(arr).max().getAsInt();
        int k = max - min + 1;
        int[] count = new int[k];
        int[] out = new int[N];

        for (int i = 0; i < N; i++) {
            count[arr[i] - min]++;
        }
        for (int i = 1; i < k; i++) {
            count[i] += count[i - 1];
        }
        for (int i = N - 1; i >= 0; i--) {
            out[count[arr[i] - min] - 1] = arr[i];
            count[arr[i] - min]--;
        }

        for (int i = 0; i < N; i++) {
            CountingParalelo.arrParallel[i + this.start] = out[i];
        }
    }
}

class CountingManager extends Thread{
    int N, H;
    public CountingManager(int N, int H){
        this.N = N;
        this.H = H;
    }
    public void run(){
        //System.out.println("Manager empezado");
        Counting[] hilos = new Counting[H];
        for(int i = 0; i < H; i++){
            Counting t = new Counting(i, i*N/H, (i+1)*N/H);
            hilos[i] = t;
        }
        for(int i = 0; i < H; i++){
            hilos[i].start();
        }
        boolean flag = true;
        while(flag){
            flag = false;
            for(int i = 0; i < H; i++){
                if(hilos[i].isAlive()){
                    //System.out.println("Hilo " + i + " sigue vivo");
                    flag = true;
                }
            }
```

```
158         }
159         Counting t = new Counting(0, 0, CountingParalelo.N);
160         t.start();
161         try{
162             t.join();
163         }catch(InterruptedException e){
164             System.out.println("Error");
165         }
166     }
167 }
```

Listing 1: Counting Sort

```
1  package org.example;
2  import org.apache.commons.math3.complex.Complex;
3
4  public class ParallelFFT {
5      private static class FFTThread extends Thread {
6          private Complex[] data;
7          private Complex[] result;
8
9          public FFTThread(Complex[] data) {
10             this.data = data;
11         }
12
13         public Complex[] getResult() {
14             return result;
15         }
16
17         @Override
18         public void run() {
19             try {
20                 result = fft(data);
21             } catch (InterruptedException e) {
22                 e.printStackTrace();
23             }
24         }
25     }
26
27     private static Complex[] fft(Complex[] data) throws InterruptedException {
28         if (data.length == 1) {
29             return data;
30         }
31
32         Complex[] even = new Complex[data.length / 2];
33         Complex[] odd = new Complex[data.length - even.length];
34         for (int i = 0; i < data.length; i++) {
35             if (i % 2 == 0) {
36                 even[i / 2] = data[i];
37             } else {
38                 odd[i / 2] = data[i];
39             }
40         }
41
42         FFTThread threadEven = new FFTThread(even);
43         FFTThread threadOdd = new FFTThread(odd);
44
45         threadEven.start();
46         threadOdd.start();
47
48         threadEven.join();
49         threadOdd.join();
50
51         Complex[] evenFFT = threadEven.getResult();
52         Complex[] oddFFT = threadOdd.getResult();
53
54         Complex[] output = new Complex[data.length];
55         for (int i = 0; i < data.length / 2; i++) {
56             output[i] = evenFFT[i].add(oddFFT[i].multiply(new Complex(Math.cos(-2 * Math.PI * i /
       data.length), Math.sin(-2 * Math.PI * i / data.length))));
57             output[i + data.length / 2] = evenFFT[i].subtract(oddFFT[i].multiply(new Complex(Math.
       cos(-2 * Math.PI * i / data.length), Math.sin(-2 * Math.PI * i / data.length))));
```

```
58          }
59
60          return output;
61      }
62
63      public static void main(String[] args) throws InterruptedException {
64         /* Complex[] data = new Complex[]{
65                  new Complex(-4.0, 0.0),
66                  new Complex(-4.0, 0.0),
67                  new Complex(-12.0, 0.0),
68                  new Complex(-16.0, 0.0)
69          };*/
70          Complex[] data = new Complex[]{
71                  new Complex(-1.0, 0.0),
72                  new Complex(2.0, 0.0),
73                  new Complex(0, 0.0),
74                  new Complex(0, 0.0),
75                  new Complex(4.0, 0.0),
76                  new Complex(0, 0.0),
77                  new Complex(-3.0, 0.0),
78                  new Complex(-1.0, 0.0)
79          };
80          Complex[] fft = fft(data);
81
82          for (Complex c : fft) {
83              System.out.println(c);
84          }
85      }
86  }
```

Listing 2: Fast Fourier Transform

---

7 Referencias

MIT Computer Science and Artificial Intelligence Laboratory. (2011). Recitation 11: Counting, Stable Matching. MIT OpenCourseWare. https://courses.csail.mit.edu/6.006/spring11/rec/rec11.pdf

Somasundaram Meiyappan, Implementation and performance evaluation of parallel FFT algorithms, School of Computing National University of Singapore

Bo Liu. “Parallel Fast Fourier Transform, 159.735 Studies in Parallel and Distributed System”

Michael Balducci., et al. “Comparative Analysis of FFT algorithms in sequential and parallel form”. Mississippi State University