Programación Concurrente y de Tiempo Real Trabajo Práctico: Ecuación de Ondas Unidimensional

Cádiz, 10/1/2025
Jose Luis Venega Sánchez

Índice general

1. Introducción al Informe **4**

2. Informe técnico **5**
2.1. Análisis matemático de la propuesta escogida 5
2.2. Conjunto de fragmentos lógicos en que se divide el problema 5
2.3. Aproximaciones de código ofrecidas por ChatGPT 6
2.3.1. Inicialización de la onda 6
2.3.2. Actualización de la onda 6
2.3.3. Visualización de los resultados de la onda 7
2.4. Solución secuencial íntegra 7
2.5. Paralelización realizada 9
2.5.1. División del dominio de datos 9
2.5.2. Tareas paralelizadas 10
2.5.3. Sincronización de los resultados 10
2.5.4. Código paralelizado 10
2.6. Pruebas realizadas y Speedup 13
2.6.1. Resultado de las pruebas realizadas 13
2.6.2. Cálculo del Speedup 16
2.6.3. Conclusiones 19

3. Bibliografía **21**

Índice de figuras

2.1. Onda Unidimensional. 5
2.2. Gráfica ejecución programa secuencial. 14
2.3. Gráfica ejecución programa con 2 hebras. 14
2.4. Gráfica ejecución programa con 4 hebras. 15
2.5. Gráfica ejecución programa con 8 hebras. 15
2.6. Gráfica comparativa general. 16
2.7. Gráfica del Speedup (2 hebras). 17
2.8. Gráfica del Speedup (4 hebras). 17
2.9. Gráfica del Speedup (8 hebras). 18
2.10. Gráfica del Speedup general. 18

1. Introducción al Informe

El presente documento tiene como objetivo el desarrollo de una solución para un problema de computacional basado en la simulación de ondas unidimensionales que se actualizan en función de su valor actual y los valores de sus vecinos.

Este problema es un ejemplo clásico de de cómo se pueden modelar sistemas físicos mediante ecuaciones de diferencias finitas, y es ideal para poder ser abordado tanto de manera secuencial como paralela, lo que nos permitirá explorar las diferencias en el rendimiento de las dos implementaciones.

El enfoque de este trabajo no solo se centra en la implementación del código, si no que también se centrará en la optimización del rendimiento, haciendo uso de herramientas como **GnuPlot** [1].

A lo largo de este informe, encontraremos:

1. Un análisis detallado del problema desde un punto de vista computacional.
2. La implementación secuencial del problema.
3. La implementación paralela del problema, teniendo en cuenta la sincronización entre tareas.
4. Pruebas que ilustrarán el rendimiento de ambas implementaciones (secuencial y paralela), y la comparación entre los resultados obtenidos.

Finalmente, se ofrecerán conclusiones sobre las ventajas e inconvenientes de cada enfoque, además se hará uso de la herramienta **ChatGPT** en el proceso de desarrollo de los códigos, con el fin de comprobar la efectividad de este.

Este trabajo se hace con el fin de comprender los desafíos de trabajar con *programación concurrente y de tiempo real*, proporcionando una base sólida para poder abordar problemas más complejos en el campo de la simulación.

2. Informe técnico

2.1. Análisis matemático de la propuesta escogida

La simulación de la onda discreta se basa en una ecuación de onda en el dominio discreto, que describe cómo se propaga una onda a lo largo de una línea. La ecuación general es:

$$A(i, t+1) = 2 \cdot A(i, t) - A(i, t-1) + c \cdot (A(i-1, t) - 2 \cdot A(i, t) + A(i+1, t))$$

La simulación se realiza utilizando un array de puntos A, donde cada elemento de A representa el valor de la onda en un punto espacial y un tiempo t. El valor de la onda se actualiza en cada paso de tiempo utilizando la fórmula discreta mencionada, considerando los valores en los puntos vecinos de A.

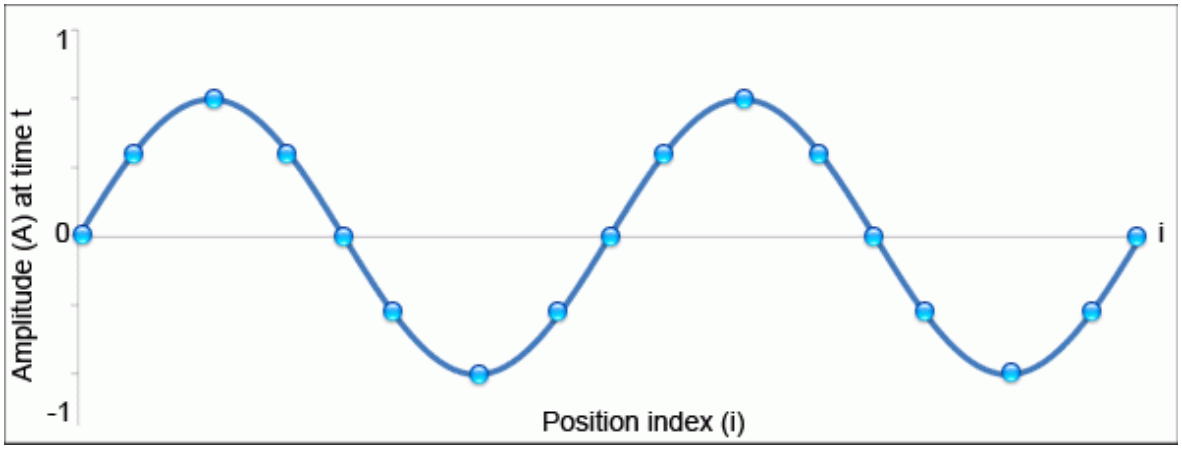

Figura 2.1: Onda Unidimensional.

2.2. Conjunto de fragmentos lógicos en que se divide el problema

El problema se puede dividir en tres unidades lógicas principales:

1. **Inicialización de la onda**: Se establece la condición inicial de la onda, utilizando una función matemática (el seno en este caso) para llenar el array `A` con valores iniciales.
    - Justificación: Al ser una operación independiente, la inicialización puede realizarse antes de iniciar la simulación.

2. **Actualización de la onda en cada paso de tiempo**: En cada paso de tiempo, se actualizan los valores de los puntos de la onda en función de los puntos vecinos utilizando la fórmula discreta. Este proceso se repite durante el número de saltos (timestamps) especificados.

    - Justificación: Esta es la operación principal de la simulación, que se repite iterativamente durante todos los pasos de tiempo.

3. **Visualización de los resultados**: Después de completar todos los pasos de tiempo, se imprime el resultado final de la onda.

    - Justificación: La visualización es una operación final que no interfiere con las simulaciones anteriores.

2.3. Aproximaciones de código ofrecidas por ChatGPT

2.3.1. Inicialización de la onda

Este trozo de código inicializa el array `A` con los valores calculados de aplicar la función *seno* (`Math.sin(punto)`) de cada índice del array, es decir de cada punto.

```
public static void InicializaValores(double[] A, double[] A_antiguo){
//Inicializamos los valores
  for(int punto = 0; punto < A.length; punto++){
    A[punto] = Math.sin(punto); //Inicializamos con el valor del seno de
    ↪  ese punto
    A_antiguo[punto] = A[punto];
  }
}
```

2.3.2. Actualización de la onda

Aplicamos la *ecuación de onda unidimensional discreta* dada para poder actualizar los valores del array `A` en cada paso o *timestamp* y almacena los resultados en un array temporal `A_temporal`, para no tener pérdidas de datos.

```
public static void ActualizaOnda(double[] A, double[] A_antiguo,
↪  double[]A_temporal, double constante){
  //Simulación
  for(int tiempo = 0; tiempo < numero_saltos; tiempo++){
  //Actualizamos los puntos aplicando la fórmula de la ecucacion de onda
  ↪  discreta
    for(int punto = 1; punto < numero_puntos-1; punto++){
      A_temporal[punto] = 2.0 * A[punto] - A_antiguo[punto] + constante *
      ↪  (A[punto-1] - (2.0 * A[punto]) + A[punto+1]);
    }
```

```
    //Copiamos los valores del A_temp en el array 'A' para obtener el
    ↪  resultado e iniciar el calculo en el siguiente punto
    System.arraycopy(A,0,A_antiguo,0,numero_puntos);
    System.arraycopy(A_temporal, 0, A, 0, numero_puntos);
  }
}
```

2.3.3. Visualización de los resultados de la onda

Finalmente, este código nos muestra los valores que toma cada punto luego de haber finalizado la ejecución de la simulación.

```
public static void ImprimeResultadosSimulacion(double[] A){
  //Devolvemos el valor del resultado
  System.out.println("El resultado de la sumulación para una onda con "
  ↪  +numero_puntos+ " puntos y " +numero_saltos+ " timestamps es:");
  for(int i = 0; i < numero_puntos; i++){
    System.out.println("A["+i+"] = " + A[i]);
  }
  System.out.println("Simulación completada");
}
```

Como vemos es una aproximación numérica y algunos valores puenden sobrepasar los valores 1 y -1, cosa que no puede suceder en la ecuación del seno.

Esto es especialmente común en simulaciones físicas, donde ciertos parámetros (como la constante de la ecuación de onda) pueden hacer que los valores de los puntos se desborden.

2.4. Solución secuencial íntegra

La solución secuencial consiste en tres fases principales:

1. **Inicialización de los valores de la onda:**
   En esta fase, los valores del array `A`, que representa la onda, son inicializados. Cada valor del array `A` se asigna a un valor calculado mediante la función seno (`Math.sin(punto)`), donde `punto` es el índice del array. Esto significa que la onda comienza con los valores de seno de los índices en el dominio, formando una onda suave. El array `A_antiguo` también se inicializa con estos mismos valores, ya que servirá como referencia para el cálculo en pasos de tiempo posteriores.

2. **Simulación de la evolución de la onda:**
   La onda se actualiza en función de una ecuación de onda discreta unidimensional durante un número determinado de *saltos de tiempo* (también conocidos como *timestamps*). En cada paso, la onda en cada punto es actualizada mediante una fórmula matemática que depende del valor de ese punto y de sus vecinos inmediatos (izquierda y derecha).

- Se usa un array temporal (`A_temporal`) para almacenar los valores calculados en un paso de tiempo, para evitar sobrescribir datos que aún no han sido procesados.
- Después de cada actualización, el contenido de `A_temporal` se copia a `A`, y este se convierte en el nuevo estado de la onda.
- Este proceso se repite durante los `numero_saltos` especificados.

3. **Visualización de los resultados:**
   Una vez que la simulación ha finalizado, la onda resultante (almacenada en el array `A`) se imprime en la consola. Cada valor de la onda es mostrado en el formato `A[i] = valor`, donde `i` es el índice del array, permitiendo observar cómo evoluciona la onda a lo largo de los diferentes puntos de la simulación.

Este enfoque permite simular cómo la onda evoluciona a lo largo de una serie de pasos de tiempo.

```
public class SimulacionSecuecial{
  static int numero_puntos = 1000000; //número de puntos de la onda
  static int numero_saltos = 5000; // veces que se ejecuta

  public static void main(String[] args){
    double[] A = new double[numero_puntos];
    double[] A_antiguo = new double[numero_puntos]; //Almacenará los
    ↪  valores antiguos
    double constante = 0.1;
    double[] A_temporal = new double[numero_puntos];
    InicializaValores(A, A_antiguo);
    ActualizaOnda(A, A_antiguo, A_temporal, constante);
    ImprimeResultadosSimulacion(A);
  }

  public static void InicializaValores(double[] A, double[] A_antiguo){
    //Inicializamos los valores
    for(int punto = 0; punto < A.length; punto++){
      A[punto] = Math.sin(punto); //Inicializamos con el valor del seno de
      ↪  ese punto
      A_antiguo[punto] = A[punto];
    }
  }

  public static void ActualizaOnda(double[] A, double[] A_antiguo,
  ↪  double[]   A_temporal, double constante){
    //Simulación
    for(int tiempo = 0; tiempo < numero_saltos; tiempo++){
      //Actualizamos los puntos aplicando la fórmula de la ecucacion de
      ↪  onda  discreta
      for(int punto = 1; punto < numero_puntos-1; punto++){
```

```
        A_temporal[punto] = 2.0 * A[punto] - A_antiguo[punto] + constante
        ↪  * (A  [punto-1] - (2.0 * A[punto]) + A[punto+1]);
      }
      //Copiamos los valores del A_temp en el array 'A' para obtener el
      ↪  resultado e   iniciar el calculo en el siguiente punto
      System.arraycopy(A,0,A_antiguo,0,numero_puntos);
      System.arraycopy(A_temporal, 0, A, 0, numero_puntos);
    }
  }

  public static void ImprimeResultadosSimulacion(double[] A){
    //Devolvemos el valor del resultado
    System.out.println("El resultado de la sumulación para una onda con "
    ↪  +numero_puntos+ " puntos y " +numero_saltos+ " timestamps es:");
    for(int i = 0; i < numero_puntos; i++){
      System.out.println("A["+i+"] = " + A[i]);
    }
    System.out.println("Simulación completada");
  }
}
```

2.5. Paralelización realizada

En la versión paralela de la simulación, el objetivo es dividir el trabajo entre varios hilos para acelerar el proceso de cálculo de la evolución de la onda. Para lograr esto, se ha utilizado la biblioteca **java.util.concurrent** [3] y su clase **ExecutorService** [4], que permite gestionar hilos de manera eficiente.

Además, hemos hecho uso de las clases **List** [5] y **ArrayList** [2] para almacenar las tareas que serán enviadas al grupo de hilos (thread pool) gestionado por ExecutorService. Esto facilita la programación paralela al permitir dividir dinámicamente las tareas según las necesidades del sistema.

La implementación sigue un enfoque modular en el que cada hilo se encarga de procesar un subconjunto del dominio de datos, asegurando una carga de trabajo equilibrada entre ellos. Este enfoque no solo mejora la eficiencia, sino que también permite una escalabilidad sencilla si se aumenta el número de hilos o el tamaño del dominio de datos.

Se han seguido las siguientes fases para realizar la paralelización:

2.5.1. División del dominio de datos

El trabajo se divide entre los hilos en función de los puntos de la onda. El número total de puntos (`numero_puntos`) se reparte equitativamente entre el número de hilos

(numero_hilos), de modo que cada hilo procese una porción del array A. Los puntos de la onda se dividen de la siguiente manera:

- Cada hilo procesa una porción de la onda correspondiente a un intervalo de puntos. Por ejemplo, si se tienen 1000 puntos y 2 hilos, cada hilo manejará 500 puntos.

2.5.2. Tareas paralelizadas

Usamos un **pool de hilos** fijo para gestionar las tareas. Cada hilo calcula los valores de los puntos en su segmento de forma independiente, utilizando fórmulas basadas en la física del problema (por ejemplo, el cálculo de una ecuación de onda discreta).

2.5.3. Sincronización de los resultados

Después de que todos los hilos han actualizado sus fragmentos de la onda, se copian los resultados desde el array temporal A_temporal al array A y se actualiza A_antiguo con el contenido actual de A. Esto asegura que las modificaciones realizadas por los hilos se reflejen de manera correcta y coherente.

2.5.4. Código paralelizado

El código presentado en la subsección 2.5.4 ejemplifica el uso de estas técnicas en la simulación de una onda discreta, implementada en Java con paralelización basada en ExecutorService. Este enfoque no solo mejora el rendimiento, sino que también simplifica el manejo de hilos y tareas, permitiendo al programador concentrarse en la lógica del problema en lugar de en la gestión de la concurrencia

```java
import java.util.concurrent.*;
import java.util.List;
import java.util.ArrayList;

public class SimulacionHilos {
  static int numero_puntos = 1000; //puntos de la onda
  static int numero_saltos = 5000; //veces que se ejecuta
  static double constante = 0.1;
  static int numero_hilos = 2; // Número de hebras

  public static void main(String[] args) throws InterruptedException,
  ↪  ExecutionException {
    // Inicializamos un pool de hilos con ExecutorService
    ExecutorService executor = Executors.newFixedThreadPool(numero_hilos);

    int puntos_proceso = numero_puntos / numero_hilos;
    double[] A = new double[numero_puntos];
```

```
  double[] A_antiguo = new double[numero_puntos];
  double[] A_temporal = new double[numero_puntos];

  // Inicializa los valores
  InicializaValores(A, A_antiguo, puntos_proceso);

  // Enviar los trabajos a las hebras
  for (int t = 0; t < numero_saltos; t++) {
    System.out.println("Timestep " + t + " comenzando simulación...");

    // Crear tareas para cada hilo
    List<Callable<Void>> tareas = new ArrayList<>();
    for (int i = 0; i < numero_hilos; i++) {
      final int hilo = i;
      tareas.add(() -> {
        int inicio = hilo * puntos_proceso;
        int fin = (hilo == numero_hilos - 1) ? numero_puntos : (hilo +
        ↪  1) * puntos_proceso;
        for (int punto = inicio + 1; punto < fin - 1; punto++) {
          A_temporal[punto] = 2.0 * A[punto] - A_antiguo[punto] +
          ↪  constante * (A[punto - 1] - (2.0 * A[punto]) + A[punto +
          ↪  1]);
        }
        return null;
      });
    }

    // Ejecutamos las tareas
    executor.invokeAll(tareas);

    // Sincronización: Copiar los valores actualizados
    System.arraycopy(A, 0, A_antiguo, 0, numero_puntos);
    System.arraycopy(A_temporal, 0, A, 0, numero_puntos);
  }

  // Imprimir resultados
  ImprimeResultadosSimulacion(A);

  // Terminamos el pool de hilos
  executor.shutdown();
}

public static void InicializaValores(double[] A, double[] A_antiguo, int
↪  puntos_proceso) {
  // Inicializamos los valores
  for (int punto = 0; punto < numero_puntos; punto++) {
```

```
        A[punto] = Math.sin(punto); // Inicializamos con el valor del seno
        ↪  de ese punto
        A_antiguo[punto] = A[punto];
      }
    }

    public static void ImprimeResultadosSimulacion(double[] A) {
      // Devolvemos el valor del resultado
      System.out.println("El resultado de la simulación para una onda con "
      ↪  + numero_puntos + " puntos, " + numero_saltos + " timestamps y una
      ↪  constante " + constante + ":");
      for (int i = 0; i < A.length; i++) {
        System.out.println("A[" + i + "] = " + A[i]);
      }
      System.out.println("Simulación completada");
    }
}
```

Funcionamiento del código anterior

1. **Inicialización del pool de hilos:**
   Se crea un `ExecutorService` que administra un conjunto fijo de hilos mediante `Executors.newFixedThreadPool(numero_hilos)`.

2. **Distribución de trabajo entre hilos:**
   El trabajo se divide en fragmentos, y cada hilo procesa un fragmento de la onda. La variable `puntos_proceso` determina cuántos puntos de la onda procesa cada hilo.

3. **Cálculo paralelo de la ecuación de onda:**
   Cada hilo calcula la actualización de la onda para su fragmento usando la fórmula discreta de la ecuación de onda. Las tareas se envían al `ExecutorService` mediante `invokeAll(tareas)`.

4. **Sincronización de resultados:**
   Después de que todos los hilos han terminado su trabajo, los resultados se copian de vuelta a los arrays correspondientes (`A` y `A_antiguo`) para continuar con la simulación en la siguiente iteración.

5. **Finalización del pool de hilos:**
   Finalmente, el `ExecutorService` se cierra con `shutdown()`, lo que asegura que todos los hilos se terminen adecuadamente.

2.6. Pruebas realizadas y Speedup

« Pruebas realizadas con un Intel Core i5 7400, a una frecuencia de 3,30GHz, el cual tiene 4 núcleos y 4 hilos. »

Para la realización de la pruebas vamos a modificar los siguientes parámetros:

- **Número de puntos** (`numero_puntos`) → Realizaremos estas pruebas con diferentes números de puntos para ver como se comporta con valores pequeños y grandes (por ejemplo, 1.000, 10.000, 100.000 y 1.000.000 de puntos), tanto para la versión secuencial como paralela.
- **Número de hilos**(`numero_hilos`) → En el caso de la versión paralela, también modificaremos el número de hebras (por ejemplo, 2, 4, 8), hasta que la curva de tiempo no varíe.

Como el número de saltos equivale a las veces que se ejecuta el bucle, mantendremos un número de saltos común para todas las pruebas que será de 5000, para poder apreciar la mejora en el uso de hilos. Además, pondremos que la constante será siempre 0.1, ya que no afecta al tiempo de ejecución.

Para cada configuración, registraremos el tiempo de ejecución en milisegundos haciendo uso de la clase **System.nanoTime();** [6] para medir el tiempo de la ejecución de todo el código:

```
long tiempo_inicio = System.nanoTime();
//Código secuencial o paralelo
long tiempo_fin= System.nanoTime();
```

2.6.1. Resultado de las pruebas realizadas

Vamos a ver las gráficas donde podemos ver y comparar el tiempo de ejecución de cada programa a las diferentes pruebas realizadas.

Resultado programa secuencial

Primero, vamos a ver la gráfica que produce el programa secuencial.

Como vemos en la *Figura 2.2*, el aumento del tiempo es lineal a medida que vamos incrementando el número de puntos de la onda. Esto hace que para una onda de 100.000 y 1.000.000 el tiempo de ejecución aumente drásticamente.

A partir de los cálculos de los tiempos para 1000 y 10.000 puntos en el modelo secuencial podemos buscar una aproximación del tiempo de ejecución para 100.000 y 1.000.000 de puntos, siendo la de 100.000 puntos de 1.5 horas aproximadamente y la de 1.000.000 puntos se acercaría a las 14 horas.

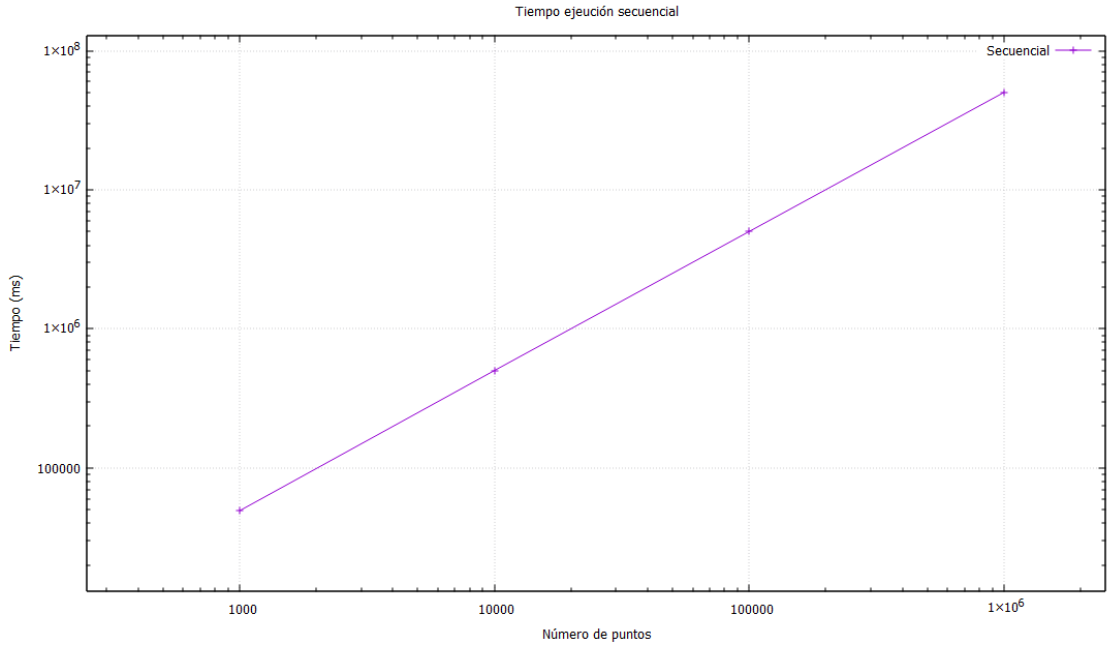

Figura 2.2: Gráfica ejecución programa secuencial.

Resultado programa paralelo con 2 hebras

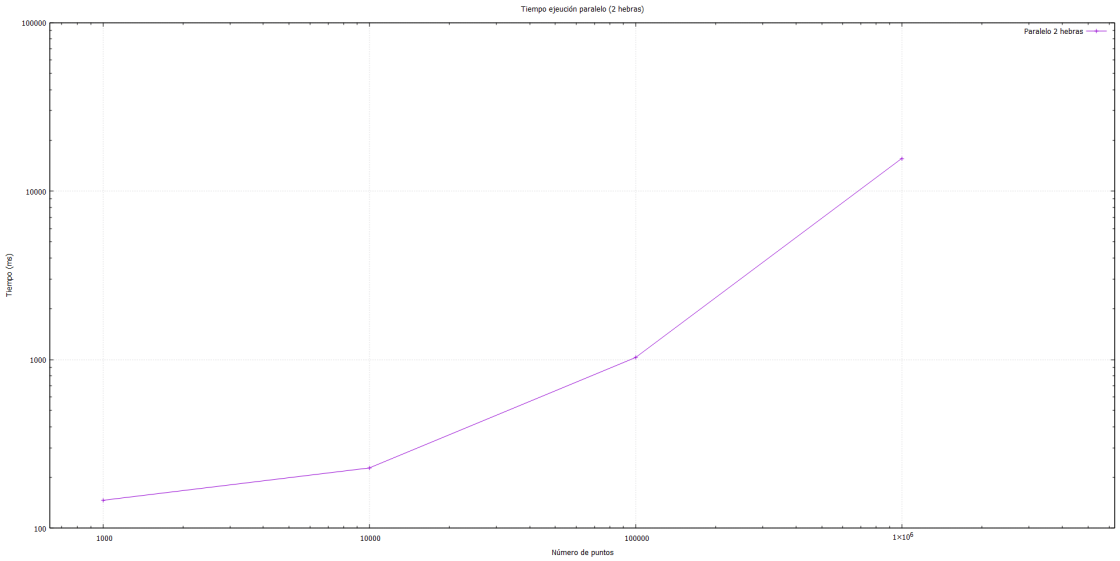

Figura 2.3: Gráfica ejecución programa con 2 hebras.

Resultado programa paralelo con 4 hebras

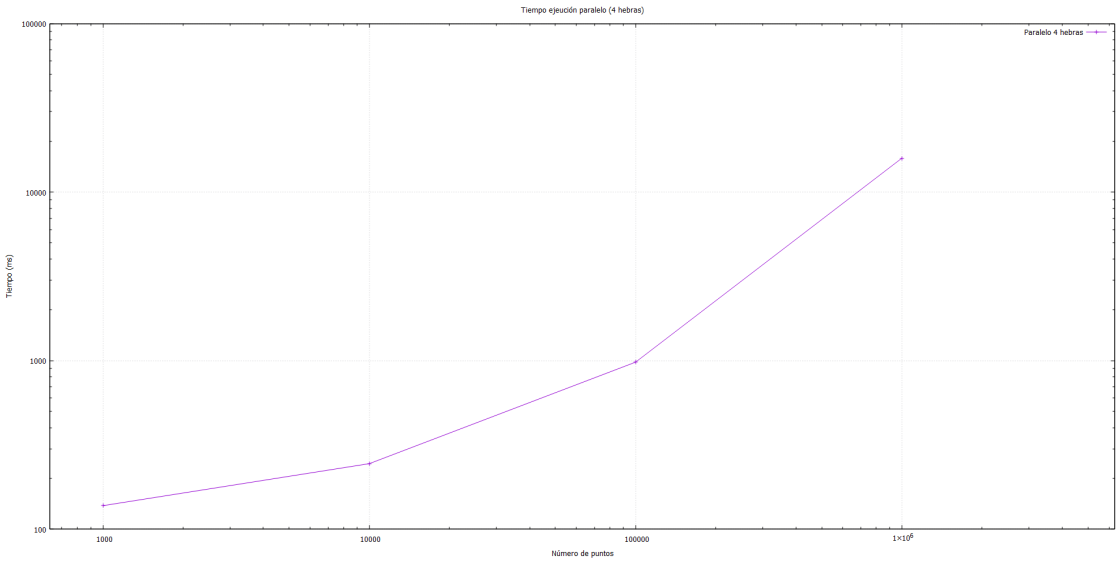

Figura 2.4: Gráfica ejecución programa con 4 hebras.

Resultado programa paralelo con 8 hebras

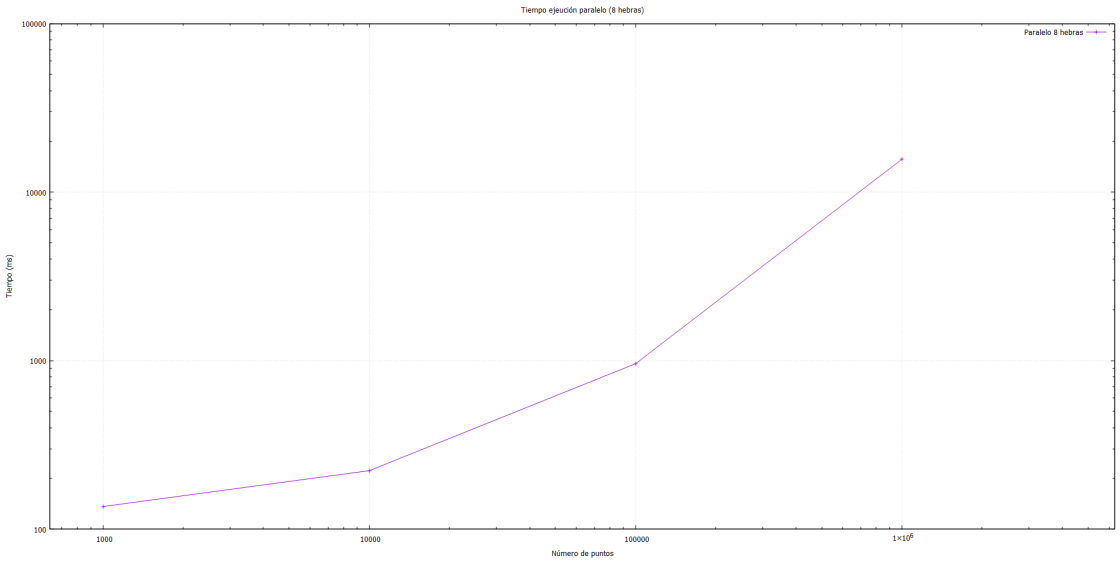

Figura 2.5: Gráfica ejecución programa con 8 hebras.

Comparativa general de los tiempo de ejecución

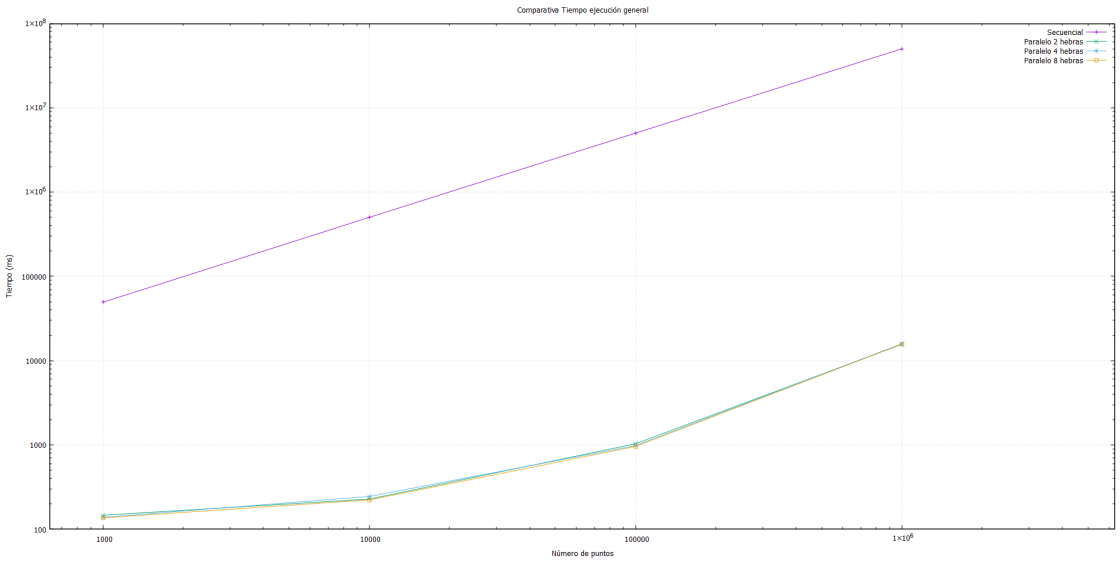

Figura 2.6: Gráfica comparativa general.

Como vemos, las gráficas de las pruebas realizadas con el código paralelizado es extremadamente inferior al reflejado en la gráfica de las pruebas con el código secuencial.

Esto se debe a lo que comentamos anteriormente de la optimización a la hora de dividir el número de puntos de la onda equitativamente entre todos los hilos del pool utilizado.

Aunque se podría esperar que un mayor número de hebras reduzca significativamente los tiempos de ejecución en un programa paralelizado, los resultados obtenidos muestran que los tiempos de las pruebas realizadas con diferente número de hebras son muy similares entre sí. Este comportamiento puede explicarse por una combinación de factores que limitan la eficacia de la paralelización en este caso específico.

2.6.2. Cálculo del Speedup

Ahora, vamos a ver el *Speedup* o mejora que se produce al hacer uso de las hebras.

Para ello, aplicaremos la siguiente fórmula que nos permite saber cuanto ha mejorado el tiempo de ejecución (Speedup) respecto a los tiempos del programa secuencial:

$$Speedup = \frac{TiempoSecuencial(ms)}{TiempoParalelo(ms)}$$

Curva Speedup 2 hebras

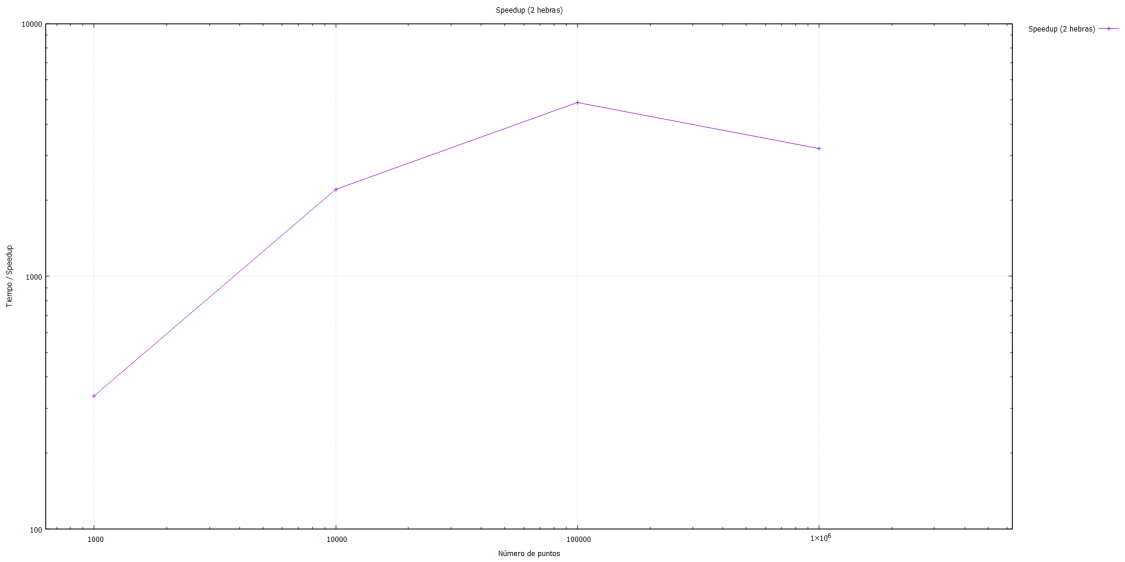

Figura 2.7: Gráfica del Speedup (2 hebras).

Curva Speedup 4 hebras

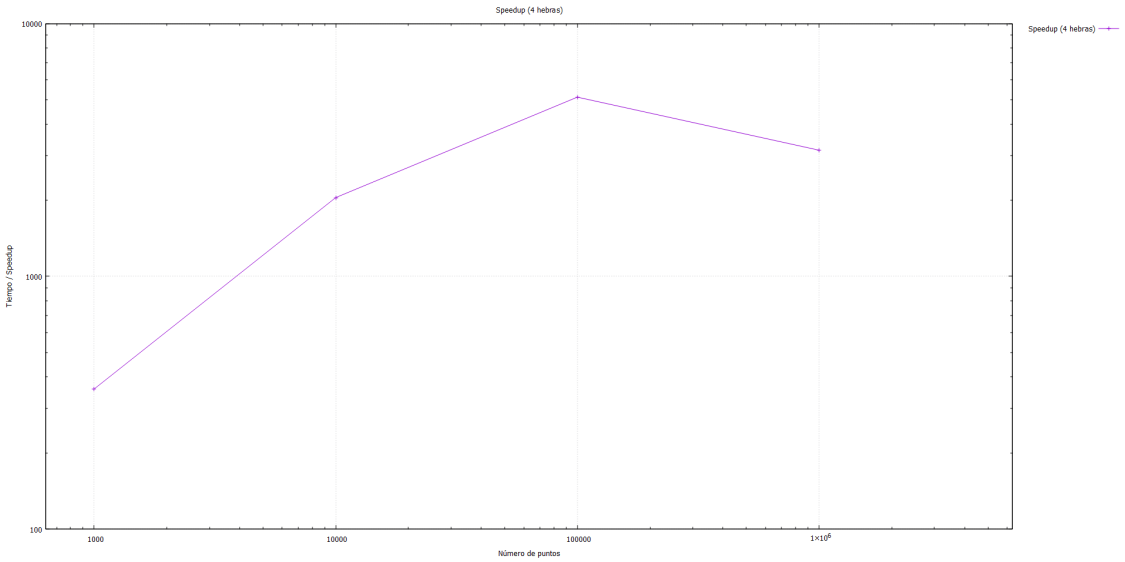

Figura 2.8: Gráfica del Speedup (4 hebras).

Curva Speedup 8 hebras

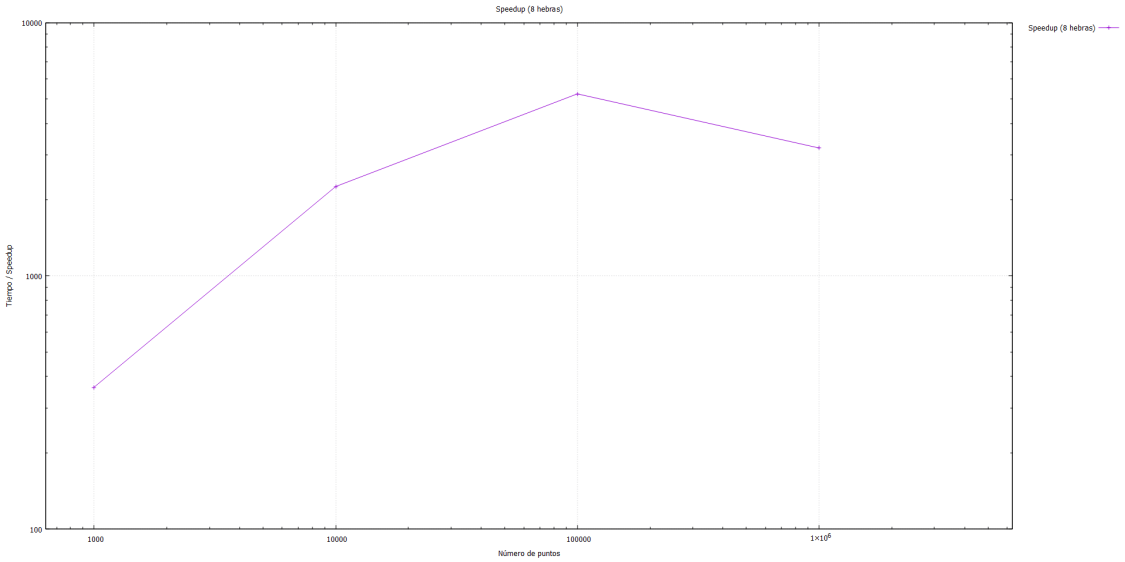

Figura 2.9: Gráfica del Speedup (8 hebras).

Curva Speedup General

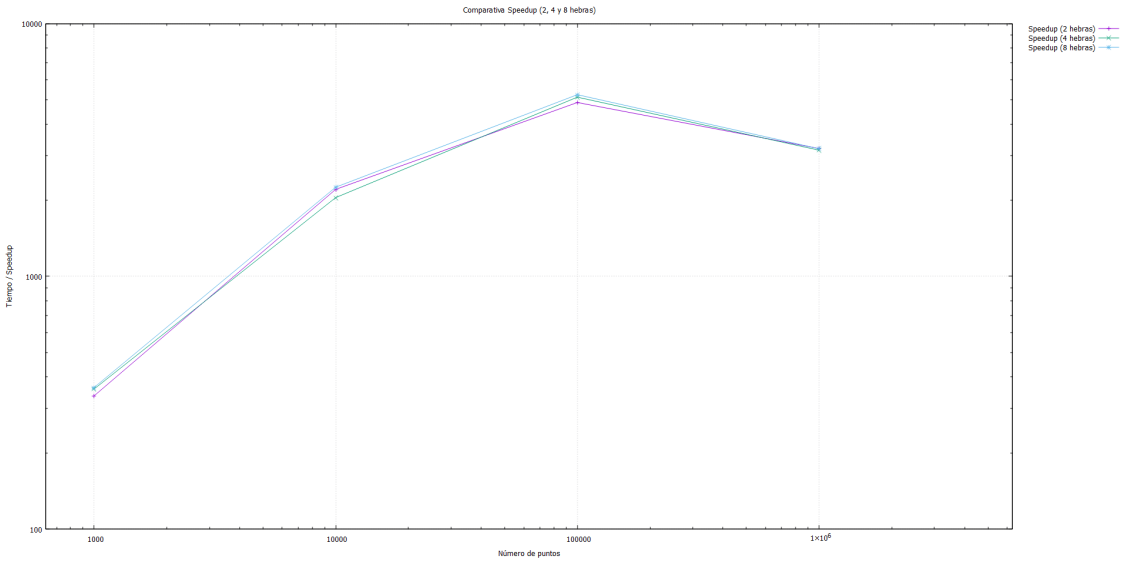

Figura 2.10: Gráfica del Speedup general.

Como vemos, a partir de 8 hebras y con un mayor número de puntos, los resultados empeoran considerablemente ya que se pueden producir **sobrecargas en la creación de los hilos**, **concatenación de recursos**, **dependencia entre hilos** y sobre todo la **Ley de Amdahl**.

2.6.3. Conclusiones

A lo largo del desarrollo de este trabajo, se ha utilizado ChatGPT como una herramienta clave para asistir en la producción de código, especialmente en la creación y paralelización de algoritmos en Java. El uso de esta herramienta ha presentado varias ventajas e inconvenientes, los cuales se detallan a continuación.

Ventajas

1. **Generación rápida de código:** ChatGPT ha facilitado la creación de fragmentos de código rápidamente, lo que ha permitido un avance significativo en las fases iniciales del proyecto. Esto ha sido especialmente útil en la implementación de la paralelización y en la adaptación del código para el uso de hebras.
2. **Asistencia en la comprensión de conceptos complejos:** Durante el proceso de paralelización y análisis de la eficiencia, ChatGPT ha servido para clarificar conceptos técnicos que en ocasiones resultan complejos, proporcionando explicaciones claras sobre temas de programación concurrente y optimización de rendimiento.
3. **Sugerencias para optimización:** La herramienta ha sido útil para sugerir mejoras en la estructura del código, incluyendo recomendaciones sobre el uso adecuado de hilos, sincronización y otros aspectos de la programación paralela. Esto ha ayudado a crear un código más eficiente y bien estructurado.

Inconvenientes

1. **Dependencia excesiva:** En algunas ocasiones, el uso de ChatGPT puede llevar a una dependencia excesiva, limitando el aprendizaje autónomo y la resolución de problemas de forma independiente. Es importante balancear el uso de la herramienta con el esfuerzo personal en la comprensión de los conceptos.
2. **Posibles errores en el código:** Aunque ChatGPT es eficiente en la generación de código, en ocasiones se han identificado pequeños errores o ineficiencias que requerían revisión y ajustes manuales. Esto puede generar una carga adicional en la etapa de depuración, aunque se puede considerar como parte del proceso de aprendizaje y mejora.
3. **Falta de contexto completo:** Aunque ChatGPT es capaz de generar soluciones adecuadas en muchos casos, la herramienta no siempre tiene el contexto completo del proyecto, lo que puede llevar a soluciones que no se ajusten completamente a las necesidades específicas del trabajo.
4. **Limitaciones en la comprensión profunda:** En algunas ocasiones, ChatGPT no ha logrado proporcionar explicaciones profundamente técnicas, especialmente cuando se trataba de problemas complejos de optimización matemática o de alto rendimiento en entornos concurrentes.

Reflexión Final

En conclusión, ChatGPT ha sido una herramienta muy útil durante el desarrollo del trabajo, especialmente en las fases de generación de código y documentación. No obstante, es importante tener en cuenta sus limitaciones y utilizarla como un complemento a la comprensión personal y al análisis detallado de los problemas. La herramienta ha facilitado el proceso de desarrollo, pero siempre debe ser utilizada con criterio y en combinación con el esfuerzo propio de aprendizaje y experimentación.

3. Bibliografía

[1] GNUPlot. Gnuplot homepage. http://www.gnuplot.info/, 2021. Último acceso: 2025-01-10.

[2] Oracle. Arraylist (java platform se 8). https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html, . Último acceso: 2025-01-10.

[3] Oracle. Concurrencia en java: java.util.concurrent (java platform se 8). https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html, . Último acceso: 2025-01-10.

[4] Oracle. Ejecutores en java: Executorservice (java platform se 8). https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html, . Último acceso: 2025-01-10.

[5] Oracle. Listas en java: List (java platform se 8). https://docs.oracle.com/javase/8/docs/api/java/util/List.html, . Último acceso: 2025-01-10.

[6] Oracle. Java system nanotime() method. https://www.tutorialspoint.com/java/lang/system_nanotime.htm, . Último acceso: 2025-01-10.