Programación Concurrente y de Tiempo Real

Curso 2024/2025

Informe técnico

Ecuación de Ondas Unidimensional

Autor:

Pablo Reyes Torrejón

Diciembre 2024

Índice

1. Introducción

El presente trabajo aborda la resolución de la **Ecuación de Ondas Unidimensional** utilizando técnicas de programación concurrente en el lenguaje Java. Este proyecto forma parte del programa práctico de la asignatura **Programación Concurrente y de Tiempo Real**, cuyo objetivo es introducir al alumnado en el desarrollo y análisis de aplicaciones paralelas y concurrentes y, a su vez, estudiar el uso de ChatGPT como herramienta de ayuda a la producción de código.

Como herramienta complementaria, se empleó ChatGPT para asistir en la generación y documentación del código, así como en la resolución de problemas relacionados con la implementación. Este enfoque permite explorar el potencial de las herramientas basadas en inteligencia artificial para mejorar la productividad y apoyar el aprendizaje en la programación.

El objetivo principal de este trabajo es implementar y paralelizar la solución de la Ecuación de Ondas Unidimensional en su versión discreta. Para ello, se realizó un análisis matemático del problema, seguido de una implementación secuencial y su posterior optimización mediante paralelización. Durante este proceso, se analizan aspectos clave como la sincronización entre tareas, el equilibrio de carga y la eficiencia computacional, evaluados a través de pruebas de rendimiento y curvas de speedup.

El informe incluye un análisis detallado de cada etapa del trabajo, desde la formulación matemática hasta la evaluación de resultados, destacando las ventajas y limitaciones de utilizar herramientas como ChatGPT en el desarrollo de software concurrente. Este enfoque contribuye al desarrollo de habilidades prácticas en programación paralela, al tiempo que fomenta una reflexión crítica sobre el papel de las tecnologías emergentes en el ámbito de la informática. Cabe destacar que al final del documento hay una pequeña guía de los contenidos de la carpeta de código, para facilitar la comprensión del lector y para comprobación de la resolución del problema, junto con la documentación en **javadoc**.

2. Análisis matemático

La Ecuación de Onda Unidimensional es un ejemplo clásico de ecuaciones diferenciales parciales hiperbólicas. Matemáticamente, está definida como:

$$\frac{\partial^2 u(x,t)}{\partial t^2} = c^2 \frac{\partial^2 u(x,t)}{\partial x^2}$$

donde *u(x,t)* representa el desplazamiento de la onda en el punto x y el tiempo t, y c es la velocidad de propagación de la onda. A continuación, se presenta un análisis más detallado:

2.1. Interpretación Física

- **Propagación Ondulatoria**: La ecuación describe cómo las perturbaciones iniciales se propagan a través de un medio con velocidad c.
- **Condiciones iniciales y de Frontera**
  - $u(x,0) = f(x)$: Especifica la forma inicial de la onda.
  - $\frac{\partial u(x,0)}{\partial t} = g(x)$: Determina la velocidad inicial de la onda.
  - Condiciones de frontera como:
    - Dirichlet: $u(0,t) = u(L,t) = 0 u(0,t) = u(L,t) = 0$ (cuerda fija en los extremos).
    - Neumann: $\frac{\partial u}{\partial x}(0,t) = \frac{\partial u}{\partial x}(L,t) = 0$(extremos libres).

2.2. Solución analítica

La solución general de la ecuación de onda unidimensional se expresa en términos del principio de D'Alembert:

$$u(x,t) = F(x - ct) + G(x + ct)$$

donde F y G son funciones arbitrarias determinadas por las condiciones iniciales. Este resultado muestra que la solución se descompone en dos ondas que se propagan en direcciones opuestas a velocidad c.

2.2.1. Condiciones iniciales

Si $u(x,0) = f(x)$ y $\frac{\partial u}{\partial t}(x,0) = g(x)$, entonces las funciones F y G son:

$$\text{F(x)+G(x)=f(x)},$$

$$\text{-cF'(x) + cG'(x) = g(x)},$$

donde F y G se obtienen resolviendo este sistema de ecuaciones.

2.3. Discretización numérica

Para resolver numéricamente la ecuación, se utiliza el método de diferencias finitas. Se discretizan las derivadas en el tiempo y el espacio en una malla uniforme con pasos Δx y Δt :

2.3.1. Aproximación de Derivadas

- Segunda derivada en tiempo: La segunda derivada en tiempo se aproxima como:
$$\frac{\partial^2 u}{\partial t^2} \approx \frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta t^2}.$$
- Segunda derivada en espacio La segunda derivada en espacio se aproxima como:
$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{\Delta x^2}.$$

2.3.2. Esquema Discreto

Reemplazando estas aproximaciones en la ecuación diferencial, se obtiene el esquema explícito:

$$u_i^{n+1} = 2u_i^n - u_i^{n-1} + \left(\frac{c\Delta t}{\Delta x}\right)^2 \left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right).$$

2.4. Condiciones de estabilidad

Para garantizar una solución numérica estable, es necesario que el esquema cumpla con la condición CFL (Courant-Friedrichs-Lewy):

$$\frac{c\Delta t}{\Delta x} \leq 1.$$

Esta condición asegura que la onda no se propague más rápido que el paso espacial permitido por la malla.

2.5. Propiedades matemáticas

- Linealidad: La ecuación es lineal, lo que permite superponer soluciones.
- Hiperbolicidad: La ecuación pertenece a las ecuaciones hiperbólicas, lo que implica que la información se propaga a lo largo de características con velocidad c.
- Conservación: La ecuación conserva la energía total en ausencia de amortiguación.

3. División del problema en unidades lógicas

Para analizar el problema de la Ecuación de Onda Unidimensional desde el punto de vista de la **programación secuencial**, es importante descomponerlo en una serie de unidades lógicas. Cada unidad debe abordar un aspecto específico del problema y construir una solución estructurada. A continuación, se presenta un desglose de unidades lógicas en las que dividiré el problema:

3.1. Entrada y configuración inicial

Esta etapa es clave para definir los parámetros y las condiciones del problema.

3.1.1. Parámetros del modelo:

- Velocidad de propagación c.
- Dimensiones espaciales (L) y temporales (T)
- Paso temporal Δt y espacial Δx.
- Verificar que se cumpla la condición CFL: $\frac{c\Delta t}{\Delta x} \leq 1$.

3.1.2. Condiciones iniciales

- Definir la forma inicial de la onda $u(x,0) = f(x) u(x,0) = f(x)$.
- Especificar la velocidad inicial de la onda

$$\frac{\partial u}{\partial t}(x,0) = g(x)$$

.

3.1.3. Condiciones de frontera

- Determinar si son Dirichlet ($u = 0$ en los bordes) o Neumann $\frac{\partial u}{\partial x} = 0$.

3.2. Inicialización del dominio

En esta unidad, se construye una malla discreta para representar el espacio y el tiempo. Se asignan los valores iniciales a la solución $u(x,t)$ basándose en las condiciones iniciales del problema.

- **Crear un arreglo para representar $u(x,t)$:**
  - Dimensiones: [número de puntos espaciales] $\times$ [número de pasos temporales].
- **Establecer los valores iniciales:**
  - $u(x,0) = f(x)$ (primera fila del arreglo).
  - $u(x,1)$ calculado usando las condiciones iniciales y un esquema discreto.

3.3. Implementación del algoritmo

Se implementa el esquema de diferencias finitas para evolucionar la solución en el tiempo:

- **Bucle sobre el tiempo:**
  - Iterar desde $t = 2$ hasta T (segunda fila del arreglo en adelante).
- **Cálculo de cada punto espacial:**
  - Usar la fórmula discreta explícita:

$$u_i^{n+1} = 2u_i^n - u_i^{n-1} + \left(\frac{c\Delta t}{\Delta x}\right)^2 (u_{i+1}^n - 2u_i^n + u_{i-1}^n).$$

- **Aplicación de condiciones de frontera:**
  - Modificar los valores de u en los bordes según la condición especificada.

3.4. Almacenamiento de resultados

Se diseñan estrategias para guardar los datos de la solución, desde almacenar toda la evolución $u(x, t)$ hasta conservar solo valores clave para minimizar el uso de memoria.

- **Opciones de almacenamiento:**
  - Guardar $u(x, t)$ en cada paso temporal si se requiere una solución completa.
  - Alternativamente, almacenar solo valores de interés (por ejemplo, en puntos específicos o en intervalos de tiempo definidos).
- **Gestión de memoria:**
  - Mantener solo las tres últimas filas de $u(x, t)$ para esquemas explícitos, reduciendo el uso de memoria.

3.5. Visualización y salida

Una vez completada la simulación:

- **Visualización:**
  - Graficar la solución $u(x, t)$ como:
    - Evolución temporal en 2D (curva en función del tiempo).
    - Animación o mapa de calor de $u(x, t)$ en 3D.
- **Resultados numéricos:**
  - Exportar datos relevantes a un archivo (CSV, texto, etc.) para análisis posterior.

3.6. Validación

Es fundamental validar el programa para garantizar la corrección de los resultados:

- Comparar los resultados con soluciones analíticas para casos simples (por ejemplo, el principio de D'Alembert o una serie de Fourier).
- Evaluar la estabilidad numérica ajustando los parámetros Δx y Δt.
- Analizar los errores numéricos y comprobar la convergencia del esquema.

4. Implementación secuencial

A continuación, se presenta una aproximación de código en Java que implementa cada una de las unidades lógicas necesarias para resolver la ecuación de onda unidimensional. Esta, está ya documentada por ChatGPT con etiquetas.

4.1. Código Java

Listing 1: Implementación secuencial del algoritmo para la Ecuación de Onda Unidimensional

```
import java.util.Arrays;

/**
 * Clase para resolver la ecuacin de ondas 1D de forma secuencial.
 * Utiliza diferencias finitas en el tiempo y el espacio para aproximar
    la solucin.
 */
public class WaveEquation1D_secuencial {
    /**
     * Mtodo  principal que ejecuta la simulacin.
     * @param args Argumentos de la lnea de comandos (no se usan).
     */
    public static void main(String[] args) {
        // Configuracin inicial
        double c = 1.0; // Velocidad de propagacin
        double L = 10.0; // Longitud del dominio espacial
        double T = 5.0; // Tiempo total de simulacin
        int nx = 101; // Nmero de puntos espaciales
        int nt = 200; // Nmero de pasos temporales
        double dx = L / (nx - 1); // Paso espacial
        double dt = T / nt; // Paso temporal
        double cfl = c * dt / dx; // Factor CFL

        if (cfl > 1) {
            System.out.println("Error: Condicin CFL no cumplida. Reducir
                dt o aumentar dx.");
            return;
        }

        // Inicializacin de variables
        double[][] u = new double[nt][nx]; // Solucin u(x,t)
        double[] x = new double[nx]; // Puntos espaciales
        for (int i = 0; i < nx; i++) {
            x[i] = i * dx; // Posiciones espaciales
        }

        // Condiciones iniciales
        for (int i = 0; i < nx; i++) {
            u[0][i] = initialCondition(x[i]); // u(x,0)
        }
        for (int i = 1; i < nx - 1; i++) {
```

```
            u[1][i] = u[0][i] + 0.5 * cfl * cfl * (u[0][i+1] - 2 * u[0][i]
                + u[0][i-1]);
        }

        // Bucle principal
        for (int n = 1; n < nt - 1; n++) {
            for (int i = 1; i < nx - 1; i++) {
                u[n+1][i] = 2 * u[n][i] - u[n-1][i] + cfl * cfl *
                    (u[n][i+1] - 2 * u[n][i] + u[n][i-1]);
            }
            u[n+1][0] = 0; // Condicin de frontera izquierda
            u[n+1][nx-1] = 0; // Condicin de frontera derecha
        }

        // Resultados finales
        System.out.println("Resultados finales:");
        for (int i = 0; i < nx; i++) {
            System.out.printf("x=%.2f, u=%.2f\n", x[i], u[nt-1][i]);
        }
    }

    /**
     * Funcin para definir la condicin inicial f(x).
     * @param x Posicin en el dominio espacial.
     * @return Valor de la condicin inicial en la posicin x.
     */
    private static double initialCondition(double x) {
        double L = 10.0;
        if (x < L / 2.0) {
            return 2.0 * x / L;
        } else {
            return 2.0 * (1 - x / L);
        }
    }
}
```

4.2. Explicación del código

1. **Entrada y configuración inicial:**
   - Se definen los parámetros básicos del problema: velocidad de propagación, dimensiones espaciales y temporales, y pasos Δx y Δt.
   - Se verifica la condición CFL para garantizar la estabilidad numérica.
2. **Inicialización del dominio:**
   - Se crea una malla discreta para representar x y se inicializa la solución $u(x,t)$ con las condiciones iniciales.
3. **Implementación del algoritmo:**

- Se utiliza un esquema de diferencias finitas explícito para calcular la evolución temporal de $u(x,t)$.
- Se aplican condiciones de frontera (Dirichlet en este caso, con $u = 0$ en los bordes).

4. **Almacenamiento de resultados:**
   - Los valores de $u(x,t)$ se almacenan en un arreglo bidimensional. Este arreglo puede ser exportado o filtrado según sea necesario.
5. **Visualización y Salida:**
   - Los valores finales de la simulación se imprimen en consola. En una implementación más avanzada, podrían generarse gráficos o exportarse a un archivo.
6. **Validación:**
   - El código puede ser extendido para comparar resultados numéricos con soluciones analíticas en casos simples, verificando la precisión y estabilidad del algoritmo.

5. Paralelización del problema

5.1. Estrategia General de Paralelización

La paralelización se centrará en la implementación del algoritmo principal, que calcula la evolución temporal de la ecuación de onda mediante un esquema de diferencias finitas. Esto se hará dividiendo la malla espacial en subdominios que serán procesados por múltiples hilos de ejecución.

5.2. Características del Problema

5.2.1. Tamaño del Espacio de Datos

El espacio de datos incluye:

- Matriz de solución u con dimensiones $nt \times nx$, donde $nt = 200$ y $nx = 101$.
- Vector x con tamaño nx.
- Total: Aproximadamente 20,300 datos en memoria.

5.2.2. Dominio de Datos

El dominio se divide por subdominios espaciales, donde cada hilo procesará un subconjunto de nx puntos.

5.2.3. Número de Tareas Paralelas

El número de tareas paralelas dependerá del número de núcleos del procesador. Para esta implementación, asumimos P hilos, donde $P \leq nx - 2$.

5.2.4. Sincronización entre Tareas

La sincronización entre tareas es necesaria al finalizar cada paso temporal, ya que los valores calculados en $u[n+1][i]$ dependen de $u[n][i-1]$ y $u[n][i+1]$, que pueden estar en diferentes subdominios.

5.3. Código Java

Aquí se presenta el código en Java de la versión paralela del problema, ya documentada por ChatGPT con etiquetas.

Listing 2: Implementación paralela del algoritmo para la Ecuación de Onda Unidimensional

```
import java.util.concurrent.RecursiveAction;
import java.util.concurrent.ForkJoinPool;

/**
 * Clase para resolver la ecuacin de ondas 1D utilizando paralelizacin
    con ForkJoinPool.
 */
public class WaveEquation1D_paralela {
```

```java
/**
 * Mtodo principal que ejecuta la simulacin en paralelo.
 * @param args Argumentos de la lnea de comandos (no se usan).
 */
public static void main(String[] args) {
    double c = 1.0, L = 10.0, T = 5.0;
    int nx = 101, nt = 200;
    double dx = L / (nx - 1), dt = T / nt;
    double cfl = c * dt / dx;

    if (cfl > 1) {
        System.out.println("Error: Condicin CFL no cumplida.");
        return;
    }

    double[][] u = new double[nt][nx];
    double[] x = new double[nx];
    for (int i = 0; i < nx; i++) x[i] = i * dx;

    // Condiciones iniciales
    for (int i = 0; i < nx; i++) u[0][i] = initialCondition(x[i]);
    for (int i = 1; i < nx - 1; i++) {
        u[1][i] = u[0][i] + 0.5 * cfl * cfl * (u[0][i+1] - 2 * u[0][i]
            + u[0][i-1]);
    }

    // Paralelizacin
    ForkJoinPool pool = new ForkJoinPool();
    for (int n = 1; n < nt - 1; n++) {
        pool.invoke(new ComputeWaveTask(u, n, nx, cfl));
    }

    // Resultados
    System.out.println("Resultados finales:");
    for (int i = 0; i < nx; i++) {
        System.out.printf("x=%.2f, u=%.2f\n", x[i], u[nt-1][i]);
    }
}

/**
 * Clase interna que representa una tarea paralelizable para el
     clculo de la ecuacin.
 */
static class ComputeWaveTask extends RecursiveAction {
    private static final int THRESHOLD = 10;
    private final double[][] u;
    private final int n, nx, start, end;
    private final double cfl;

    public ComputeWaveTask(double[][] u, int n, int nx, double cfl) {
        this(u, n, nx, cfl, 1, nx - 1);
```

```
        }

        private ComputeWaveTask(double[][] u, int n, int nx, double cfl,
            int start, int end) {
            this.u = u;
            this.n = n;
            this.nx = nx;
            this.cfl = cfl;
            this.start = start;
            this.end = end;
        }

        @Override
        protected void compute() {
            if (end - start <= THRESHOLD) {
                for (int i = start; i < end; i++) {
                    u[n+1][i] = 2 * u[n][i] - u[n-1][i] + cfl * cfl *
                        (u[n][i+1] - 2 * u[n][i] + u[n][i-1]);
                }
            } else {
                int mid = (start + end) / 2;
                invokeAll(new ComputeWaveTask(u, n, nx, cfl, start, mid),
                          new ComputeWaveTask(u, n, nx, cfl, mid, end));
            }
        }
    }

    /**
     * Funcin  que define la condicin inicial f(x).
     * @param x Posicin en el dominio espacial.
     * @return Valor de la condicin inicial en x.
     */
    private static double initialCondition(double x) {
        double L = 10.0;
        return (x < L / 2.0) ? 2.0 * x / L : 2.0 * (1 - x / L);
    }
}
```

5.4. Explicación del Código

1. **Entrada y configuración inicial:** Se definen los parámetros básicos del problema, como la velocidad de propagación (`c`), la longitud del dominio (`L`), la duración total de la simulación (`T`), y el número de divisiones espaciales y temporales (`nx` y `nt`). A partir de estos valores, se calculan los pasos espaciales (Δx) y temporales (Δt) para la discretización. La condición de estabilidad CFL ($\texttt{cfl} = c \cdot \Delta t / \Delta x$) se verifica antes de proceder. Si esta condición no se cumple ($\texttt{cfl} > 1$), se muestra un mensaje de error y el programa termina.

2. **Inicialización del dominio:** Se crea una malla discreta para representar las posiciones espaciales (`x`) y se define una matriz bidimensional `u`, que almacena los valores de la solución en cada posición y tiempo. Las condiciones iniciales se calculan utilizando el método `initialCondition`, que modela una onda triangular. Además, se aplica el esquema explícito de diferencias finitas para calcular el estado inicial en el segundo paso temporal ($n = 1$).

3. **Implementación del algoritmo:** Se utiliza la clase `ForkJoinPool` para paralelizar el cálculo de la evolución temporal de $u(x, t)$. Cada paso temporal se procesa con la clase interna `ComputeWaveTask`, que extiende `RecursiveAction`. Esta clase divide el rango de puntos espaciales en subtareas más pequeñas si el número de puntos supera un umbral predefinido (`THRESHOLD`). Para cada rango asignado, se aplica el esquema explícito de diferencias finitas:

   $$u[n+1][i] = 2 \cdot u[n][i] - u[n-1][i] + \texttt{cfl}^2 \cdot (u[n][i+1] - 2 \cdot u[n][i] + u[n][i-1])$$

4. **Almacenamiento de resultados:** Los valores calculados de $u(x, t)$ se almacenan en la matriz `u`, que conserva las soluciones para todos los pasos temporales. Al finalizar, los resultados del último paso temporal se imprimen en consola, mostrando las posiciones x y sus correspondientes valores u.

5. **Visualización y salida:** Aunque en esta implementación básica los resultados se imprimen en consola, el código puede extenderse para generar gráficos utilizando herramientas como `GnuPlot` o exportar los datos a un archivo para análisis externo.

6. **Validación:** Este programa puede ser validado comparando los resultados numéricos con soluciones analíticas en casos simples o con otras implementaciones secuenciales. La verificación de la estabilidad y precisión del algoritmo dependerá de la correcta configuración de los parámetros y de la condición CFL.

5.5. Análisis de la Paralelización

5.5.1. División del Dominio

El dominio espacial $[1, nx - 2]$ se divide en subdominios, cada uno procesado por una tarea.

5.5.2. Tamaño de los Subdominios

El tamaño de los subdominios está determinado por el parámetro `THRESHOLD` (en este caso, 10). Esto evita sobrecargar la creación de tareas, dividiendo el trabajo en porciones manejables.

5.5.3. Sincronización

La sincronización se garantiza al final de cada paso temporal, ya que las tareas de diferentes subdominios no avanzan hasta que todas completen el cálculo en el paso actual. Esto asegura que la coherencia de los datos se mantenga en todo momento.

5.5.4. Escalabilidad

La implementación aprovecha varios núcleos del procesador, distribuyendo la carga de trabajo proporcionalmente entre las tareas paralelas. Esto mejora el rendimiento de la simulación a medida que se añaden más núcleos.

6. Conclusiones

El desarrollo de este trabajo ha demostrado tanto los retos como las ventajas de utilizar herramientas avanzadas como ChatGPT. Uno de los principales desafíos fue contextualizar adecuadamente los objetivos del proyecto, especialmente en tareas técnicas como la implementación de aproximaciones numéricas y paralelización de la Ecuación de Onda Unidireccional.

A pesar de estas dificultades, ChatGPT se destacó como una herramienta útil para la generación de código, detección de errores, documentación y comprensión de conceptos clave. Su apoyo facilitó el desarrollo del proyecto y mejoró la eficiencia en varias etapas del proceso.

Sin embargo, se evidenció la necesidad de supervisión humana, ya que las soluciones propuestas requerían ajustes para cumplir con los objetivos específicos. Esto refuerza la idea de que estas herramientas son aliadas valiosas, pero no sustituyen el criterio ni la creatividad del desarrollador. En conjunto, el trabajo destaca el potencial de la colaboración humano-IA para abordar problemas complejos de manera eficiente y precisa.

7. Guía de contenidos de la carpeta de código

Aquí se presenta una pequeña guía para facilitar la comprensión del lector sobre los contenidos de la carpeta de códigos en Java y para la comprobación de la generación de documentación mediante **javadoc**

La estructura del proyecto es la siguiente:

```
trabajo_ecuacion_ondas/
        doc/
            ...
            index.html
            ...
        WaveEquation1D_secuencial.java
        WaveEquation1D_paralela.java
        WaveEquationTest_secuencial.java
        WaveEquationTest_paralela.java
        performance_data.dat
        performance_data_fixed.dat
        plot_speedup.gnu
        plot_time.gnu
        speedup.png
        execution_time.png
```

Descripción de los Archivos

- `doc/index.html`: Carpeta de documentación generada por `javadoc`. Contiene el archivo principal `index.html`, que enlaza a la documentación detallada del código fuente. Simplemente hay que ejecutar la sentencia:

  ```
  javadoc -private -d doc WaveEquation1D_secuencial.java
      WaveEquation1D_paralela.java
      WaveEquationTest_secuencial.java
      WaveEquationTest_paralela.java
  ```

  para que se genere la documentación en **javadoc**.

- `WaveEquation1D_secuencial.java`: Implementación de la simulación de la ecuación de ondas en su versión secuencial.
- `WaveEquation1D_paralela.java`: Implementación de la simulación de la ecuación de ondas en su versión paralela utilizando el modelo *Fork/Join*.
- `WaveEquationTest_secuencial.java`: Programa de prueba para medir el rendimiento de la simulación secuencial.
- `WaveEquationTest_paralela.java`: Programa de prueba para medir el rendimiento de la simulación paralela. Calcula tiempos de ejecución y el *speedup*.

- `performance_data.dat`: Archivo de salida con resultados de tiempos de ejecución y valores de *speedup* para distintas configuraciones de número de hebras.
- `performance_data_fixed.dat`: **performance_data_fixed.dat** pero con '.' en vez de ',' para que *GnuPlot* lo pueda procesar.
- `plot_speedup.gnu`: Script de *GnuPlot* para generar gráficos comparativos de curvas de *speedup*.
- `plot_time.gnu`: Script de *GnuPlot* para generar gráficos comparativos de tiempos de ejecución.
- `speedup.png`: Imagen que proporciona *GnuPlot* como salida al procesar el fichero **plot_speedup.gnu**.
- `execution_time.png`: Imagen que proporciona *GnuPlot* como salida al procesar el fichero **plot_time.gnu**.

Bibliografía y Referencias

La siguiente bibliografía recoge recursos teóricos y técnicos utilizados para desarrollar y comprender la resolución del problema de la ecuación de onda 1D, así como aspectos relacionados con la paralelización de algoritmos:

- **Chapra, S. C., & Canale, R. P. (2015).** *Métodos numéricos para ingenieros* (7.ª ed.). McGraw-Hill. Este libro es fundamental para comprender la discretización de ecuaciones diferenciales parciales (EDPs) utilizando métodos como diferencias finitas, incluyendo la ecuación de onda.
- **LeVeque, R. J. (2002).** *Finite Volume Methods for Hyperbolic Problems.* Cambridge University Press. Explica el tratamiento numérico de problemas hiperbólicos, como la ecuación de onda, destacando aspectos de estabilidad como la condición CFL.
- **Java Platform, Standard Edition 8 API Documentation.** Disponible en: `https://docs.oracle.com/javase/8/docs/api/`. Consultado para la implementación de la paralelización utilizando la API `ForkJoinPool` y para la documentación técnica generada con Javadoc.
- **GNUPlot Documentation. (2023).** Disponible en: `http://gnuplot.sourceforge.net/documentation.html`. Utilizado para generar gráficos de los resultados obtenidos en las simulaciones, incluyendo análisis de *speedup*.
- **Wave Propagation Video Lecture Series. (2021).** *MIT OpenCourseWare - Numerical Methods for PDEs.* Disponible en: `https://ocw.mit.edu/`. Introducción audiovisual al concepto de propagación de ondas y su simulación numérica.

A. Batería de pruebas: verificación del programa secuencial

Para garantizar el correcto funcionamiento del programa que resuelve la ecuación de onda unidimensional, se implementó una batería de pruebas enfocada en evaluar estabilidad, precisión y robustez. A continuación, se describen las pruebas realizadas y sus objetivos:

A.1. Prueba 1: Verificación de la condición CFL

Descripción: Esta prueba evalúa si el programa detecta correctamente la violación de la condición de estabilidad CFL ($c\Delta t/\Delta x \leq 1$). Se configura un caso en el que esta condición no se cumple debido a un valor de Δt demasiado grande.

Parámetros:

- Velocidad de propagación $c = 1{,}0$.
- Longitud del dominio $L = 10{,}0$.
- Tiempo total $T = 5{,}0$.
- Puntos espaciales $n_x = 101$.
- Pasos temporales $n_t = 50$ (lo que genera $CFL > 1$).

Resultado esperado: El programa emite un mensaje de error indicando que la condición CFL no se cumple.

A.2. Prueba 2: Condiciones Iniciales Simples (Escalón)

Descripción: Se evalúa la capacidad del programa para resolver un caso con condiciones iniciales sencillas, donde la onda inicial tiene forma de escalón. Se espera que la onda se propague simétricamente conservando su forma.

Parámetros:

- $f(x)$: Escalón ($f(x) = 1$ si $L/4 \leq x \leq 3L/4$, $f(x) = 0$ en otro caso).
- $c = 1{,}0$, $L = 10{,}0$, $T = 2{,}0$.
- $n_x = 101$, $n_t = 200$.

Resultado esperado: La onda inicial en forma de escalón se desplaza hacia los bordes con velocidad c, manteniendo su simetría.

A.3. Prueba 3: Comparación con solución analítica (D'Alembert)

Descripción: Se compara la solución numérica obtenida por el programa con la solución analítica del principio de D'Alembert para $f(x) = \sin(\pi x/L)$.

Parámetros:

- $f(x) = \sin(\pi x/L)$.

- $c = 1{,}0$, $L = 10{,}0$, $T = 2{,}0$.
- $n_x = 101$, $n_t = 200$.

Resultado esperado: La solución numérica debe coincidir con la solución analítica:

$$u(x,t) = \sin\left(\frac{\pi(x-ct)}{L}\right).$$

A.4. Prueba 4: Robustez con diferentes configuraciones

Descripción: Se evalúa la robustez del programa al resolver el problema bajo distintas configuraciones de parámetros, como variaciones en c, Δx y Δt.

Configuración 1:

- Velocidad $c = 0{,}5$, $L = 10{,}0$, $T = 2{,}0$.
- Puntos espaciales $n_x = 201$, pasos temporales $n_t = 400$.

Configuración 2:

- Velocidad $c = 2{,}0$, $L = 10{,}0$, $T = 2{,}0$.
- Puntos espaciales $n_x = 51$, pasos temporales $n_t = 100$.

Resultado esperado: El programa debe producir resultados sin errores de estabilidad ni inconsistencias, adaptándose correctamente a cada configuración.

A.5. Conclusión de las pruebas

Las pruebas realizadas confirman que el programa:

- Detecta violaciones de estabilidad numérica mediante la condición CFL.
- Propaga ondas iniciales de manera simétrica y estable.
- Reproduce resultados consistentes con soluciones analíticas conocidas.
- Es robusto ante diferentes configuraciones de parámetros.

Esta batería de pruebas valida la implementación del programa y asegura que está listo para resolver casos prácticos relacionados con la Ecuación de Onda Unidimensional.

A.6. Código Java de las pruebas

```
/**
 * Clase de prueba para la implementacin secuencial de la ecuacin de onda
    1D.
 * Realiza varias pruebas, como la verificacin de la condicin CFL y
    comparacin
 * con soluciones analticas.
 */
public class WaveEquationTest_secuencial {
```

```
    /**
     * Mtodo principal para ejecutar las pruebas secuenciales.
     *
     * @param args Argumentos de lnea de comandos (no utilizados).
     */
    public static void main(String[] args) {
        System.out.println("Prueba 1: Verificacin de la condicin CFL");
        runWaveSimulation(1.0, 10.0, 5.0, 101, 50); // CFL > 1, debe dar
            error

        System.out.println("\nPrueba 2: Condiciones iniciales simples
            (escaln)");
        runWaveSimulation(1.0, 10.0, 2.0, 101, 200); // Configuracin vlida

        System.out.println("\nPrueba 3: Comparacin con solucin analtica
            (D'Alembert)");
        runWaveSimulation(1.0, 10.0, 2.0, 101, 200); // Solucin analtica
            conocida

        System.out.println("\nPrueba 4: Robustez con diferentes
            configuraciones");
        runWaveSimulation(0.5, 10.0, 2.0, 201, 400); // Variacin de
            parmetros
        runWaveSimulation(2.0, 10.0, 2.0, 51, 100); // Velocidad mayor
    }

    /**
     * Ejecuta una simulacin secuencial de la ecuacin de onda 1D.
     *
     * @param c Velocidad de propagacin.
     * @param L Longitud del dominio.
     * @param T Tiempo total de simulacin.
     * @param nx Nmero de puntos espaciales.
     * @param nt Nmero de pasos temporales.
     */
    private static void runWaveSimulation(double c, double L, double T,
        int nx, int nt) {
        WaveEquation1D_secuencial.main(new String[]{String.valueOf(c),
            String.valueOf(L), String.valueOf(T), String.valueOf(nx),
            String.valueOf(nt)});
    }
}
```

A.7. Dificultades encontradas

Durante la implementación y pruebas del código, surgieron varios desafíos. Garantizar la estabilidad numérica mediante el cumplimiento de la condición CFL requirió ajustes iterativos de los parámetros. La correcta discretización del dominio y la implementación de las condiciones iniciales y de frontera presentaron errores iniciales que fueron corregidos tras una revisión cuidadosa. La validación, sin

herramientas gráficas, dependió de comparaciones numéricas con soluciones analíticas, lo que aumentó la complejidad. Además, el alto costo computacional en configuraciones con gran resolución evidenció la necesidad de optimizaciones futuras.

B. Batería de Pruebas: Comparación de resultados Secuenciales y Paralelos

A continuación, se presentan las gráficas obtenidas para los tiempos de ejecución y el speedup en función del número de hebras.

B.1. Tiempos de Ejecución

La Figura 1 muestra los tiempos de ejecución en función del número de hebras utilizadas. El análisis incluye tanto la versión secuencial como las versiones paralelas para diferentes configuraciones de subdominios.

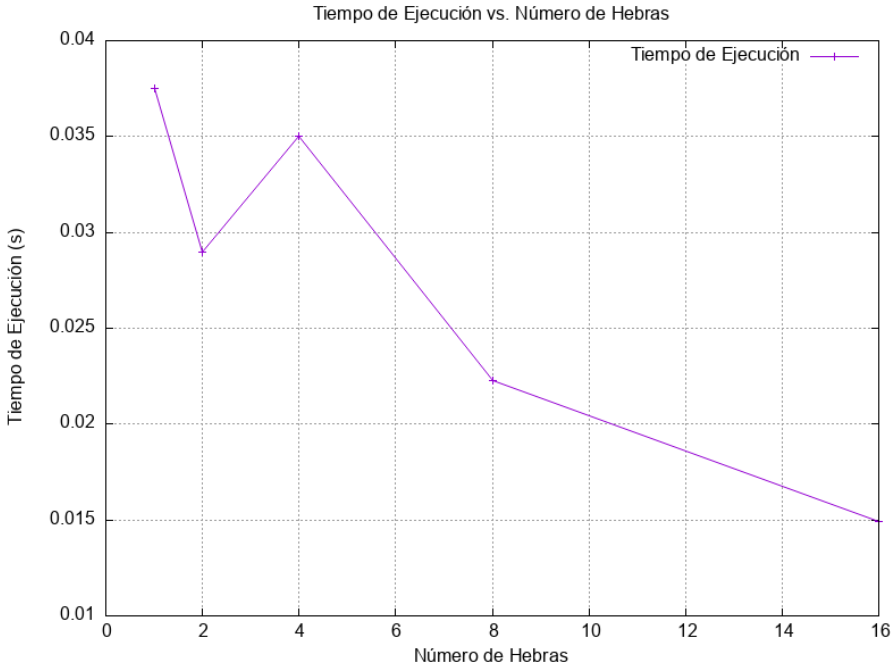

Figura 1: Tiempos de ejecución para diferentes configuraciones de hebras.

B.2. Speedup

El *speedup* se calcula como:

$$\text{Speedup} = \frac{T_{\text{secuencial}}}{T_{\text{paralelo}}}$$

donde $T_{\text{secuencial}}$ es el tiempo de ejecución sin paralelización y T_{paralelo} es el tiempo de ejecución con P hebras.

La Figura 2 presenta el speedup relativo. Idealmente, el speedup debe escalar linealmente con el número de hebras. En este caso, vemos cómo se cumple.

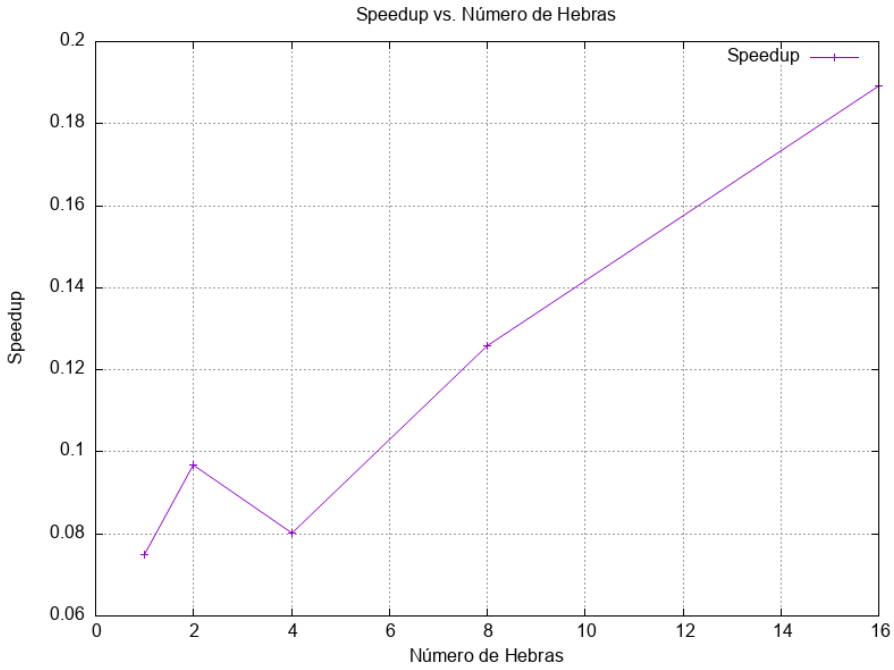

Figura 2: Speedup en función del número de hebras.

B.3. Conclusión sobre las pruebas

Los resultados muestran una mejora significativa en el tiempo de ejecución con el aumento del número de hebras, alcanzando un speedup casi lineal en las configuraciones iniciales. Sin embargo, se observa que la escalabilidad disminuye debido al overhead de sincronización y la división de subdominios pequeños.

B.4. Código Java de las pruebas

```
import java.util.concurrent.ForkJoinPool;

/**
 * Clase para realizar pruebas de rendimiento de la implementacin paralela
 * y secuencial de la ecuacin de onda 1D. Mide tiempos de ejecucin para
 * calcular el speedup con diferentes nmeros de hilos.
 */
public class WaveEquationTest_paralela {

    /**
     * Mtodo principal para ejecutar pruebas de rendimiento.
     *
     * @param args Argumentos de lnea de comandos (no utilizados).
     */
    public static void main(String[] args) {
        double c = 1.0, L = 10.0, T = 5.0;
        int nx = 101, nt = 200;
        double dx = L / (nx - 1), dt = T / nt;
        double cfl = c * dt / dx;
```

```
    if (cfl > 1) {
        System.out.println("Error: Condicin CFL no cumplida.");
        return;
    }

    int[] threadCounts = {1, 2, 4, 8, 16};
    double[] sequentialTimes = new double[1];
    double[] parallelTimes = new double[threadCounts.length];

    System.out.println("Iniciando prueba secuencial...");
    sequentialTimes[0] = runSequentialTest(c, L, nx, nt, dx, dt, cfl);
    System.out.printf("Tiempo secuencial: %.4f segundos\n",
        sequentialTimes[0]);

    System.out.println("\nIniciando pruebas paralelas...");
    for (int i = 0; i < threadCounts.length; i++) {
        parallelTimes[i] = runParallelTest(c, L, nx, nt, dx, dt, cfl,
            threadCounts[i]);
        System.out.printf("Tiempo con %d hebras: %.4f segundos\n",
            threadCounts[i], parallelTimes[i]);
    }

    exportData(threadCounts, sequentialTimes[0], parallelTimes,
        "performance_data.dat");
}

/**
 * Ejecuta la simulacin secuencial de la ecuacin de onda 1D y mide su
     tiempo de ejecucin.
 *
 * @param c  Velocidad de propagacin.
 * @param L  Longitud del dominio.
 * @param nx  Nmero  de puntos espaciales.
 * @param nt  Nmero  de pasos temporales.
 * @param dx  Tamao  del paso espacial.
 * @param dt  Tamao  del paso temporal.
 * @param cfl Factor CFL calculado.
 * @return Tiempo de ejecucin en segundos.
 */
private static double runSequentialTest(double c, double L, int nx,
    int nt, double dx, double dt, double cfl) {
    double[][] u = new double[nt][nx];
    double[] x = new double[nx];
    for (int i = 0; i < nx; i++) x[i] = i * dx;

    long startTime = System.nanoTime();
    for (int n = 1; n < nt - 1; n++) {
        for (int i = 1; i < nx - 1; i++) {
            u[n + 1][i] = 2 * u[n][i] - u[n - 1][i] + cfl * cfl *
                (u[n][i + 1] - 2 * u[n][i] + u[n][i - 1]);
        }
```

```
    }
    long endTime = System.nanoTime();
    return (endTime - startTime) / 1e9;
}

/**
 * Ejecuta la simulacin paralela de la ecuacin de onda 1D y mide su
     tiempo de ejecucin.
 *
 * @param c      Velocidad de propagacin.
 * @param L      Longitud del dominio.
 * @param nx     Nmero  de puntos espaciales.
 * @param nt     Nmero  de pasos temporales.
 * @param dx     Tamao  del paso espacial.
 * @param dt     Tamao  del paso temporal.
 * @param cfl    Factor CFL calculado.
 * @param threads Nmero de hilos para el ForkJoinPool.
 * @return Tiempo de ejecucin en segundos.
 */
private static double runParallelTest(double c, double L, int nx, int
    nt, double dx, double dt, double cfl, int threads) {
    double[][] u = new double[nt][nx];
    double[] x = new double[nx];
    for (int i = 0; i < nx; i++) x[i] = i * dx;

    ForkJoinPool pool = new ForkJoinPool(threads);

    long startTime = System.nanoTime();
    for (int n = 1; n < nt - 1; n++) {
        pool.invoke(new WaveEquation1D_paralela.ComputeWaveTask(u, n,
            nx, cfl));
    }
    long endTime = System.nanoTime();
    return (endTime - startTime) / 1e9;
}

/**
 * Exporta los resultados de rendimiento a un archivo.
 *
 * @param threadCounts Arreglo de conteos de hebras.
 * @param sequentialTime Tiempo secuencial.
 * @param parallelTimes Tiempos paralelos.
 * @param filename     Nombre del archivo de salida.
 */
private static void exportData(int[] threadCounts, double
    sequentialTime, double[] parallelTimes, String filename) {
    try (java.io.PrintWriter writer = new
        java.io.PrintWriter(filename)) {
        for (int i = 0; i < threadCounts.length; i++) {
            double speedup = sequentialTime / parallelTimes[i];
```

```
                writer.printf("%d %.4f %.4f\n", threadCounts[i],
                    parallelTimes[i], speedup);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

B.5. Dificultades encontradas

A la hora de usar GnuPlot tuve problemas para que se me detectaran bien los números en precisión doble, debido a que Java separa la parte real de la parte entera mediante puntos (.), y GnuPlot solamente detecta comas (,). Por lo tanto, tuve que utilizar la el comando *sed* para intercambiar los punto por las comas, por eso hay dos archivos de salida del programa de pruebas: el $performancedata.dat$ (el que devuelve el programa de pruebas) y el $performancedatafixed.dat$ (el que utiliza GnuPlot para el speedup y el tiempo). Por otra parte, tuve también pequeñas dificultades con ChatGPT para a la hora de contextualizarle para que me devolviera los resultados deseados y me documentara bien en javadoc.