Laboratorio 1: Programación Paralela con OpenMP

Simulador de Propagación de Ondas en Redes (C++)

Departamento de Ingeniería Informática
Universidad de Santiago de Chile

6 de septiembre de 2025

1. Objetivos

Al finalizar este laboratorio, los estudiantes serán capaces de:

- Implementar aplicaciones paralelas usando OpenMP
- Utilizar diferentes cláusulas de scheduling en OpenMP
- Aplicar cláusulas de sincronización (atomic, critical, reduce)
- Calcular métricas de performance (speedup, eficiencia)
- Analizar la Ley de Amdahl en aplicaciones paralelas
- Generar gráficos de performance y análisis
- Desarrollar código orientado a objetos en C++
- Implementar clases y métodos con OpenMP
- Comprender fenómenos de propagación de ondas en redes

2. Descripción del Problema

Se requiere desarrollar un **Simulador de Propagación de Ondas en Redes** que simule cómo se propagan ondas a través de una red de nodos conectados y analice el comportamiento del sistema. La aplicación debe ser completamente paralela usando OpenMP.

2.1. Contexto del Problema

2.1.1. Simulación de Propagación en Redes

En este laboratorio simularemos cómo se propaga una señal a través de una red de nodos conectados. Este tipo de simulación es común en:

- **Redes de comunicación:** Propagación de información entre routers
- **Redes sociales:** Difusión de información entre usuarios

- **Simulaciones físicas:** Propagación de calor, sonido o vibraciones

Concepto Básico:
Cada nodo de la red tiene un **valor** (amplitud) que representa la intensidad de la señal en ese punto. La señal se propaga de la siguiente manera:

1. Cada nodo recibe influencia de sus nodos vecinos
2. El valor del nodo cambia según la diferencia con sus vecinos
3. Se aplica amortiguación para simular pérdida de energía
4. Se pueden agregar fuentes externas

Ecuación de Propagación:
Para cada nodo i, su valor $A_i(t)$ evoluciona según:

$$A_i(t + \Delta t) = A_i(t) + \Delta t \cdot \left[D \sum_{j \in vecinos(i)} (A_j(t) - A_i(t)) - \gamma A_i(t) + S_i(t) \right] \tag{1}$$

donde:

- $A_i(t)$ = valor del nodo i en el tiempo t
- D = coeficiente de difusión (qué tan rápido se propaga)
- γ = coeficiente de amortiguación (pérdida de energía)
- $S_i(t)$ = fuente externa en el nodo i
- Δt = paso de tiempo

2.1.2. Redes de Propagación

Tipos de Redes:

1. **Red regular 1D:** Línea de nodos conectados secuencialmente
2. **Red regular 2D:** Cuadrícula de nodos con conexiones a vecinos inmediatos
3. **Red aleatoria:** Conexiones aleatorias entre nodos
4. **Red de mundo pequeño:** Combinación de estructura local y conexiones alejadas

Propiedades de la Red:

- **Grado de nodo:** Número de conexiones de cada nodo
- **Distancia de propagación:** Tiempo para que una onda llegue de un nodo a otro
- **Coeficiente de clustering:** Qué tan conectados están los vecinos de un nodo

2.1.3. Algoritmo de Simulación Paso a Paso

Paso 1: Inicialización

1. Crear N nodos en la red
2. Conectar nodos según el tipo de red (1D, 2D, aleatoria)
3. Asignar valores iniciales $A_i(0)$ a cada nodo
4. Configurar parámetros D, γ, Δt

Paso 2: Loop Principal (para cada paso de tiempo)

1. **Calcular nuevas amplitudes:** Para cada nodo i:
   - *a)* Sumar las diferencias con todos los vecinos: $\sum_{j \in vecinos(i)} (A_j(t) - A_i(t))$
   - *b)* Multiplicar por el coeficiente de difusión: $D \cdot \sum_{j \in vecinos(i)} (A_j(t) - A_i(t))$
   - *c)* Restar la amortiguación: $-\gamma A_i(t)$
   - *d)* Agregar fuente externa: $+S_i(t)$
   - *e)* Multiplicar por el paso de tiempo: $\Delta t \cdot$ [resultado anterior]
   - *f)* Actualizar: $A_i(t + \Delta t) = A_i(t) +$ resultado
2. **Actualizar fuentes externas** (si las hay)
3. **Calcular métricas** del sistema

Ejemplo Numérico:
Supongamos un nodo con valor $A_i(t) = 1{,}0$, dos vecinos con valores $A_j(t) = 0{,}8$ y $A_k(t) = 1{,}2$, $D = 0{,}1$, $\gamma = 0{,}01$, $\Delta t = 0{,}01$:

$$\text{Diferencia con vecinos} = (0{,}8 - 1{,}0) + (1{,}2 - 1{,}0) = -0{,}2 + 0{,}2 = 0{,}0 \tag{2}$$
$$\text{Término de difusión} = 0{,}1 \times 0{,}0 = 0{,}0 \tag{3}$$
$$\text{Término de amortiguación} = -0{,}01 \times 1{,}0 = -0{,}01 \tag{4}$$
$$\text{Fuente externa} = 0{,}0 \text{ (asumiendo sin fuente)} \tag{5}$$
$$\text{Cambio total} = 0{,}01 \times (0{,}0 - 0{,}01 + 0{,}0) = -0{,}0001 \tag{6}$$
$$\text{Nuevo valor} = 1{,}0 + (-0{,}0001) = 0{,}9999 \tag{7}$$

2.2. Fundamentos Teóricos

2.2.1. De la Ecuación Diferencial a la Discreta

La ecuación diferencial que describe la propagación es:

$$\frac{dA_i}{dt} = D \sum_{j \in vecinos(i)} (A_j - A_i) - \gamma A_i + S_i \tag{8}$$

Para resolverla numéricamente, usamos el método de Euler explícito, que aproxima la derivada como:

$$\frac{dA_i}{dt} \approx \frac{A_i(t + \Delta t) - A_i(t)}{\Delta t} \tag{9}$$

Sustituyendo en la ecuación diferencial:

$$\frac{A_i(t+\Delta t) - A_i(t)}{\Delta t} = D \sum_{j \in vecinos(i)} (A_j(t) - A_i(t)) - \gamma A_i(t) + S_i(t) \tag{10}$$

Despejando $A_i(t + \Delta t)$:

$$A_i(t+\Delta t) = A_i(t) + \Delta t \left[D \sum_{j \in vecinos(i)} (A_j(t) - A_i(t)) - \gamma A_i(t) + S_i(t) \right] \tag{11}$$

Esta es la ecuación que implementaremos en el código.

2.2.2. Métricas del Sistema

Energía Total:

$$E_{total}(t) = \sum_{i=1}^{N} A_i^2(t) \tag{12}$$

Velocidad de Propagación:

$$v_{prop} = \frac{d_{max}}{t_{llegada}} \tag{13}$$

Amplitud Promedio:

$$\langle A \rangle = \frac{1}{N} \sum_{i=1}^{N} A_i \tag{14}$$

3. Requisitos Técnicos

3.1. Introducción a OpenMP

OpenMP (Open Multi-Processing) es una API para programación paralela que permite paralelizar código C/C++ usando directivas del compilador. Las directivas OpenMP comienzan con `#pragma omp`.

3.2. Cláusulas OpenMP Obligatorias

La implementación debe incluir **todas** las siguientes cláusulas:

1. **Scheduling:** Controla cómo se distribuyen las iteraciones entre los threads
   - `schedule(static, chunk)` - Distribución estática con tamaño de chunk fijo
   - `schedule(dynamic, chunk)` - Distribución dinámica con tamaño de chunk fijo
   - `schedule(guided, chunk)` - Distribución guiada con tamaño mínimo de chunk
2. **Sincronización:** Controla el acceso a variables compartidas
   - `atomic` - Operaciones atómicas (más eficiente)
   - `critical` - Secciones críticas (más general)
   - `reduce` - Reducciones paralelas (suma, producto, etc.)
3. **Otras cláusulas:**

- `single` - Ejecuta código en un solo thread
- `nowait` - Elimina la barrera implícita
- `collapse` - Colapsa loops anidados en uno solo

4. **Manejo de datos:** Controla cómo se comparten variables entre threads
   - `private` - Variable privada para cada thread
   - `shared` - Variable compartida entre todos los threads
   - `firstprivate` - Variable privada inicializada con valor del thread maestro
   - `lastprivate` - Variable privada que conserva el valor del último thread
5. **Sincronización avanzada:**
   - `barrier` - Punto de sincronización explícito
   - `task` - Creación de tareas paralelas independientes

3.3. Análisis de Performance

Se debe implementar un sistema completo de benchmarks que incluya:

1. **Medición de tiempos:**
   - Usar `omp_get_wtime()` para medir tiempo de ejecución
   - Ejecutar cada experimento múltiples veces (mínimo 10 repeticiones)
   - Calcular promedio y desviación estándar de los tiempos
   - Reportar resultados como: $T = \bar{T} \pm \sigma_T$
2. **Cálculo de speedup:**
$$S_p = \frac{T_1}{T_p} \tag{15}$$
   donde:
   - T_1 = tiempo de ejecución serial (1 thread)
   - T_p = tiempo de ejecución paralelo (p threads)
3. **Cálculo de eficiencia:**
$$E_p = \frac{S_p}{p} \tag{16}$$
   donde:
   - E_p = eficiencia del paralelismo ($0 \leq E_p \leq 1$)
4. **Análisis de Ley de Amdahl:**
$$S_p = \frac{1}{f + \frac{1-f}{p}} \tag{17}$$
   donde:
   - f = fracción serial del código ($0 \leq f \leq 1$)
   - p = número de threads
   - Predice el speedup máximo teórico
   - $f = \frac{T_{serial}}{T_{total}}$
5. **Gráficos de performance** usando Python/Matplotlib o Gnuplot

3.3.1. Propagación de Errores

Cuando calculamos métricas derivadas (como speedup y eficiencia) a partir de mediciones con error, necesitamos propagar esos errores correctamente.

Concepto básico:

Si tenemos una función $z = f(x, y)$ donde x y y tienen errores σ_x y σ_y, el error en z se calcula como:

$$\sigma_z = \sqrt{\left(\frac{\partial f}{\partial x}\sigma_x\right)^2 + \left(\frac{\partial f}{\partial y}\sigma_y\right)^2} \tag{18}$$

Error en Speedup:

Para $S_p = \frac{T_1}{T_p}$, aplicando la fórmula de propagación:

$$\sigma_{S_p} = S_p\sqrt{\left(\frac{\sigma_{T_1}}{T_1}\right)^2 + \left(\frac{\sigma_{T_p}}{T_p}\right)^2} \tag{19}$$

Error en Eficiencia:

Para $E_p = \frac{S_p}{p}$, como p es constante (sin error):

$$\sigma_{E_p} = \frac{\sigma_{S_p}}{p} \tag{20}$$

Ejemplo:

Para $T_1 = 9{,}97 \pm 0{,}15$ s y $T_4 = 3{,}01 \pm 0{,}12$ s:

$$\sigma_{S_4} = 3{,}31\sqrt{\left(\frac{0{,}15}{9{,}97}\right)^2 + \left(\frac{0{,}12}{3{,}01}\right)^2} = 0{,}14 \tag{21}$$

$$\sigma_{E_4} = \frac{0{,}14}{4} = 0{,}035 \tag{22}$$

Resultado: $S_4 = 3{,}31 \pm 0{,}14$, $E_4 = 0{,}83 \pm 0{,}04$

4. Estructura del Proyecto

4.1. Archivos Requeridos

El proyecto debe tener la siguiente estructura:

```
wave_propagation/
|-- main.cpp                # Archivo principal
|-- Node.h/.cpp             # Clase Node para nodos de la red
|-- Network.h/.cpp          # Clase Network para manejo de la red
|-- WavePropagator.h/.cpp # Clase WavePropagator para propagación
|-- Integrator.h/.cpp       # Clase Integrator para integración temporal
|-- MetricsCalculator.h/.cpp # Clase MetricsCalculator para métricas
|-- Benchmark.h/.cpp        # Clase Benchmark para análisis de performance
|-- Visualizer.h/.cpp       # Clase Visualizer para gráficos
|-- Makefile                # Compilación
`-- README.md               # Instrucciones
```

4.2. Programación Orientada a Objetos

- **Clases separadas** en archivos .h/.cpp
- **Métodos específicos** para cada cláusula OpenMP
- **Encapsulación** de datos y métodos
- **Documentación** en comentarios de clase
- **Herencia** opcional para diferentes tipos de redes

5. Especificaciones de Implementación

5.1. Diseño de Clases

Deben implementar las siguientes clases:

- **Node:** Representa un nodo de la red con ID, amplitud, vecinos
- **Network:** Maneja la red completa con nodos y conexiones
- **WavePropagator:** Calcula la propagación de ondas usando OpenMP
- **Integrator:** Integración temporal con diferentes métodos
- **MetricsCalculator:** Cálculo de métricas del sistema
- **Benchmark:** Análisis de performance y benchmarks
- **Visualizer:** Generación de gráficos y visualización

5.1.1. Ejemplo de Diseño de Clases

Clase Node:

```
class Node {
private:
    int id;
    double amplitude;
    double previous_amplitude;
    std::vector<int> neighbors;

public:
    Node(int node_id);
    void addNeighbor(int neighbor_id);
    void updateAmplitude(double new_amplitude);
    double getAmplitude() const;
    const std::vector<int>& getNeighbors() const;
    int getDegree() const;
};
```

Clase Network:

```cpp
class Network {
private:
    std::vector<Node> nodes;
    int network_size;
    double diffusion_coeff;
    double damping_coeff;

public:
    Network(int size, double diff_coeff, double damp_coeff);
    void initializeRandomNetwork();
    void initializeRegularNetwork(int dimensions);

    // Function overloading para diferentes schedules
    void propagateWaves();                    // Schedule por defecto
    void propagateWaves(int schedule_type);   // static=0, dynamic=1, guided=2
    void propagateWaves(int schedule_type, int chunk_size);
    void propagateWavesCollapse();            // Con collapse

    const std::vector<Node>& getNodes() const;
    int getSize() const;
};
```

Clase WavePropagator:

```cpp
class WavePropagator {
private:
    Network* network;
    double time_step;

public:
    WavePropagator(Network* net, double dt);

    // Function overloading para integración con diferentes cláusulas
    void integrateEuler();                    // Integración básica
    void integrateEuler(int sync_type);       // atomic=0, critical=1, nowait=2
    void integrateEuler(int sync_type, bool use_barrier);

    // Function overloading para cálculo de energía
    void calculateEnergy();                   // Cálculo básico
    void calculateEnergy(int method);         // reduce=0, atomic=1
    void calculateEnergy(int method, bool use_private);

    // Function overloading para procesamiento de nodos
    void processNodes();                      // Procesamiento básico
    void processNodes(int task_type);         // task=0, parallel_for=1
    void processNodes(int task_type, bool use_single);

```

```
    // Métodos específicos para cláusulas únicas
    void simulatePhasesBarrier();            // Con barrier
    void parallelInitializationSingle();     // Con single
};
```

5.2. Métodos Obligatorios con Function Overloading

1. **Clase Network - Cálculo de Propagación:**
   - `propagateWaves()` - método básico
   - `propagateWaves(int schedule_type)` - static=0, dynamic=1, guided=2
   - `propagateWaves(int schedule_type, int chunk_size)` - con chunk size
   - `propagateWavesCollapse()` - con collapse (método específico)
2. **Clase WavePropagator - Integración Temporal:**
   - `integrateEuler()` - método básico
   - `integrateEuler(int sync_type)` - atomic=0, critical=1, nowait=2
   - `integrateEuler(int sync_type, bool use_barrier)` - con barrier opcional
3. **Clase WavePropagator - Cálculo de Energía:**
   - `calculateEnergy()` - método básico
   - `calculateEnergy(int method)` - reduce=0, atomic=1
   - `calculateEnergy(int method, bool use_private)` - con private/shared
4. **Clase WavePropagator - Procesamiento de Nodos:**
   - `processNodes()` - método básico
   - `processNodes(int task_type)` - task=0, parallel_for=1
   - `processNodes(int task_type, bool use_single)` - con single opcional
5. **Métodos Específicos para Cláusulas Únicas:**
   - `simulatePhasesBarrier()` - con barrier para sincronizar fases
   - `parallelInitializationSingle()` - con single para inicialización
   - `calculateMetricsFirstprivate()` - con firstprivate
   - `calculateFinalStateLastprivate()` - con lastprivate

5.3. Benchmarks Requeridos

1. **Comparación de Schedules:**
   - static vs dynamic vs guided
   - Diferentes chunk sizes
   - Análisis de load balancing
2. **Comparación de Sincronización:**

- atomic vs critical vs reduce
- Análisis de contención

3. **Manejo de Datos:**
   - private vs shared - impacto en performance
   - firstprivate vs lastprivate - overhead de copia
   - Análisis de memory access patterns
4. **Sincronización Avanzada:**
   - barrier vs nowait - overhead de sincronización
   - task vs parallel for - escalabilidad de tareas
   - single vs master - inicialización paralela
5. **Escalabilidad:**
   - Speedup vs número de threads
   - Eficiencia vs número de threads
   - Análisis de Amdahl
6. **Medición de Fracción Serial:**
   - Implementar medición directa de tiempos seriales vs paralelos
   - Comparar con predicción teórica de Amdahl
   - Análisis de overhead de paralelización

6. Parámetros de Simulación

6.1. Configuración por Defecto

- **Número de nodos:** 10000
- **Pasos de simulación:** 1000
- **Paso de tiempo:** 0.01
- **Tamaño de red:** 100x100 (para redes 2D)
- **Coeficiente de difusión:** 0.1

6.2. Parámetros Físicos

- **Coeficiente de difusión:** $D = 0{,}1$
- **Coeficiente de amortiguación:** $\gamma = 0{,}01$
- **Amplitud inicial:** $A_0 = 1{,}0$
- **Fuente externa:** $S = 0{,}1 \sin(\omega t)$

7. Análisis y Visualización

7.1. Gráficos Requeridos

1. **Speedup vs Threads:** Comparar diferentes implementaciones
2. **Eficiencia vs Threads:** Análisis de escalabilidad
3. **Tiempo vs Chunk Size:** Para diferentes schedules
4. **Ley de Amdahl:** Comparación teórica vs experimental
5. **Propagación de Ondas:** Evolución temporal de amplitudes
6. **Energía del Sistema:** Conservación de energía

7.2. Archivos de Salida

- `benchmark_results.dat` - Resultados de benchmarks
- `scaling_analysis.dat` - Análisis de escalabilidad
- `performance_plots.png` - Gráficos de performance
- `wave_evolution.dat` - Evolución de las ondas
- `energy_conservation.dat` - Conservación de energía

8. Instrucciones de Compilación

8.1. Makefile

El Makefile debe incluir:

```
CXX = g++
CXXFLAGS = -Wall -Wextra -O3 -fopenmp -std=c++17
LDFLAGS = -fopenmp

TARGET = wave_propagation
SOURCES = main.cpp Node.cpp Network.cpp WavePropagator.cpp Integrator.cpp
↪ MetricsCalculator.cpp Benchmark.cpp Visualizer.cpp
HEADERS = Node.h Network.h WavePropagator.h Integrator.h MetricsCalculator.h
↪ Benchmark.h Visualizer.h

$(TARGET): $(SOURCES) $(HEADERS)
    $(CXX) $(CXXFLAGS) -o $(TARGET) $(SOURCES) $(LDFLAGS)

clean:
    rm -f $(TARGET) *.o *.dat *.png

benchmark: $(TARGET)
    ./$(TARGET) -benchmark
```

```
17
18 analysis: $(TARGET)
19     ./$(TARGET) -analysis
20
21 .PHONY: clean benchmark analysis
```

8.2. Compilación

```
1  # Compilar
2  make
3
4  # Ejecutar benchmarks
5  make benchmark
6
7  # Ejecutar análisis
8  make analysis
9
10 # Limpiar
11 make clean
```

9. Criterios de Evaluación

9.1. Implementación (40 %)

- Correcta implementación de todas las cláusulas OpenMP en métodos de clase
- Diseño orientado a objetos con encapsulación apropiada
- Manejo de memoria y errores en C++
- Documentación del código con comentarios de clase

9.2. Benchmarks (30 %)

- Implementación completa del sistema de benchmarks
- Comparaciones entre diferentes implementaciones
- Análisis de escalabilidad
- Cálculos correctos de métricas
- Implementación correcta de propagación de errores en speedup y eficiencia

9.3. Análisis y Visualización (20 %)

- Gráficos de performance
- Análisis de Ley de Amdahl

- Interpretación de resultados físicos
- Conclusiones sobre performance
- Presentación correcta de resultados con barras de error

9.4. Reporte Técnico (10 %)

- Estructura y organización del reporte
- Claridad en la explicación del diseño
- Análisis crítico de los resultados
- Conclusiones bien fundamentadas
- Calidad de la presentación y formato

10. Entregables

1. **Código fuente** completo del proyecto (archivos .cpp y .h)
2. **Makefile** funcional para compilación
3. **README.md** con instrucciones detalladas de compilación y ejecución
4. **Resultados de benchmarks** en archivos .dat
5. **Gráficos de performance** en formato PNG
6. **Reporte técnico** (máximo 10 páginas) que incluya:
   - **Introducción:** Descripción del problema y objetivos
   - **Diseño de la solución:** Arquitectura de clases y decisiones de diseño
   - **Implementación:** Explicación de cómo se implementaron las cláusulas OpenMP usando function overloading
   - **Resultados experimentales:** Análisis de speedup, eficiencia y Ley de Amdahl con propagación de errores
   - **Análisis de performance:** Comparación entre diferentes schedules y cláusulas
   - **Conclusiones:** Interpretación de resultados y lecciones aprendidas
   - **Anexos:** Gráficos, tablas de resultados y código relevante

11. Fecha de Entrega

Fecha límite: Sábado 27 de septiembre de 2025 a las 23:59
Formato de entrega:

- **Plataforma:** Google Classroom del curso
- **Código:** Archivo comprimido (.zip o .tar.gz) con todo el proyecto
- **Reporte:** Documento PDF con el reporte técnico
- **Entrega separada:** El reporte puede entregarse por separado del código

12. Recursos Adicionales

- **OpenMP Documentation:** `https://www.openmp.org/`
- **GNU Compiler Collection:** `https://gcc.gnu.org/`
- **C++ Reference:** `https://en.cppreference.com/`
- **Performance Analysis Tools:** gprof, valgrind
- **Visualization:** Python/Matplotlib, Gnuplot
- **Wave Propagation:** `https://en.wikipedia.org/wiki/Wave_equation`

13. Tips de Implementación

- **Empezar simple:** Implementar primero la versión serial
- **Probar incrementalmente:** Verificar cada cláusula OpenMP por separado
- **Usar herramientas de debugging:** gdb, valgrind, gprof
- **Medir performance:** Ejecutar benchmarks múltiples veces
- **Documentar resultados:** Anotar observaciones durante el desarrollo

14. Notas Importantes

- El código debe compilar sin warnings
- Se debe incluir manejo de errores apropiado
- Los benchmarks deben ser reproducibles
- Se debe documentar cualquier asunción o limitación
- El análisis de performance debe ser riguroso y detallado
- **Importante:** Usar function overloading para reducir redundancia de código
- **Importante:** Cada cláusula OpenMP debe tener su propio método de prueba
- **Importante:** Los benchmarks deben ser estadísticamente significativos
- **Importante:** Usar encapsulación apropiada en las clases
- **Importante:** Considerar la física del problema en el diseño de las clases
- **Importante:** Implementar métodos básicos que llamen a versiones más específicas