



## ***Laboratorio 1 - Simulador de Propagación de Ondas en Redes mediante programación paralela***

### ***Integrantes***

- Ignacio Lara
- Gonzalo Moncada

### ***Profesor***

- Miguel Cárcamo



## **Tabla de contenidos**

<b>1. Introducción</b>	<b>3</b>
<b>2. Diseño de la solución</b>	<b>4</b>
<b>3. Resultados experimentales</b>	<b>7</b>
<b>4. Análisis de Performance</b>	<b>8</b>
<b>5. Conclusiones</b>	<b>9</b>
<b>6. Anexos</b>	<b>10</b>
Figura 1: Fórmula de Propagación de onda.	10
Figura 2: Extracto de código de la función “propagateWaves”	10
Figura 3: Extracto de código de la función “IntegrateEulerCore”	11
Figura 4: Extracto de código de la función “calculateEnergy”.	11
Figura 5: Extracto de código de la función “processNodes”.	11
Figura 6: Fórmulas de Speedup, de eficiencia y de la ley de Amdahl respectivamente.	12
Figura 7: Gráfico generado mediante simulaciones con la obtención de energía total vs tiempo.	12
Figura 8: Gráfico generado mediante simulaciones con el tiempo de ejecución vs número de chunks.	13
Figura 9: Gráfico generado mediante simulaciones con el valor teórico obtenido de la Ley de Amdahl y el Speedup obtenido.	14
Figura 10: Gráficos generados mediante simulaciones con el speedup y el número de hilos utilizados, además de eficiencia y el número de hilos.	14
Figura 11: Gráficos generados mediante simulaciones con el speedup y el tipo de schedule, además de eficiencia y el tipo de schedule.	14



## **1. Introducción**

El reporte presentado representa el desarrollo e implementación de una simulación de ondas en redes regulares. Como objetivo principal es modelar y analizar la dinámica de difusión en mallas de distintas dimensiones, esto mediante el uso de distintas técnicas presentes en las librerías de OpenMP, la cual permite realizar una optimización paralela del código realizado.

La propagación de señales en redes es un fenómeno fundamental en física, ingeniería y ciencias computacionales, con aplicaciones que van desde la transferencia de calor hasta la comunicación en sistemas distribuidos. En este contexto, se replicó un entorno virtual donde la velocidad de propagación, el comportamiento de los nodos y el rendimiento bajo diferentes configuraciones fueron observados y evaluados en detalle.



## 2. Diseño de la solución

Para realizar el siguiente trabajo se realizó diferentes clases para poder representar el fenómeno, las principales son tanto como la clase “Node” como la clase “Network” que representan tanto la red como cada uno de los nodos de manera individual, en el caso de los nodos estos se componen de su id, su amplitud actual como también la previa, además de sus vecinos, esta clase contiene funciones básicas para la actualización de su amplitud, además de la asignación de sus nodos vecinos, como último de esta clase se declaran todos sus getters para la obtención de sus valores en otras clases.

La clase “Network” representa la estructura de la red y almacena internamente un arreglo de nodos, lo cual permite modelar redes en dos dimensiones—cada nodo tiene una amplitud propia y conexiones con sus vecinos—y extenderse fácilmente a tres dimensiones utilizando un arreglo de redes. Además, esta clase guarda la cantidad total de nodos que conforman la red y los parámetros físicos relevantes: el coeficiente de difusión (relacionado con la rapidez de propagación de la onda), el coeficiente de amortiguación (que determina la pérdida de energía en cada paso de la simulación) y el timestep (que fija cuánto avanza el tiempo virtual en cada iteración), como último se almacena lo que es la fuente externa o ruido, el cual es calculado mediante la frecuencia angular con el paso del tiempo. Entre sus funciones principales destacan los métodos para iniciar tanto topologías aleatorias como regulares, estas últimas configurables en una o dos dimensiones. Finalmente, incluye métodos de acceso (“getters”) para consultar sus atributos y así facilitar la gestión y uso externo de la red.

Como función fundamental dentro de la clase “Network” está el método “propagateWaves”, encargado de simular la evolución física de la onda en la red. Su lógica consiste en calcular, para cada nodo, el efecto que los vecinos ejercen sobre su amplitud y aplicar las ecuaciones de propagación que consideran la difusión y la amortiguación. Utilizando estos cálculos, el método actualiza de forma eficiente las amplitudes de todos los nodos, reflejando cómo se propaga un impulso inicial, y puede apoyarse en técnicas de paralelización para optimizar el rendimiento en simulaciones de gran tamaño.

En la [figura 1](#) se presenta la fórmula de propagación, la cual actualiza la amplitud de cada nodo sumando la difusión entre vecinos, la pérdida por amortiguación y una fuente externa periódica, todo multiplicado por el paso de tiempo, esta fórmula contiene todos los atributos previamente almacenados en Network de tal manera que se realizó el siguiente código para calcularla:

El código mostrado en la [figura 2](#) calcula la nueva amplitud para cada nodo sumando la influencia de sus vecinos (difusión) y restando el efecto de amortiguación, usando la ecuación de propagación de ondas y actualizando todos los nodos en paralelo al final del ciclo. La lógica emplea integración explícita tipo Euler y almacena temporalmente las nuevas amplitudes en un vector para evitar



condiciones de carrera, como último se agrega lo que es el coeficiente de ruido al centro geométrico calculado. Además, la clase cuenta con métodos sobrecargados (function overloading) que permiten aplicar distintas técnicas de paralelismo con OpenMP, como schedule static, dynamic o guided, para optimizar el rendimiento según la configuración de la simulación, de esta manera podemos comparar estos distintos rendimientos dado distintos tipos de schedule. Como último se tiene la función de “propagateWavesCollapse”, la cual permite paralelizar la propagación en una red 2D, utilizando la directiva de OpenMP “collapse”.

Además de esto se generó otra clase llamada “WavePropagator”, esta incluye varios métodos para simular la propagación de ondas y calcular propiedades del sistema usando técnicas de integración numérica y paralelismo con OpenMP.

Como se muestra en la [figura 3](#) los métodos integrateEuler implementan la integración temporal mediante el método de Euler explícito, donde las amplitudes de los nodos se actualizan en cada paso según un decaimiento simple (se utiliza el valor 0.99 para representar este amortiguamiento) y la influencia de los vecinos, usando diferentes tipos de sincronización paralela (atomic, critical, nowait) para evitar condiciones de carrera y optimizar la ejecución concurrente, además de esto se realizó una función que utiliza la directiva de “barrier” para sincronizar los hilos.

Los métodos calculateEnergy suman la energía total del sistema (suma de cuadrados de amplitudes), de la misma forma que se muestra en la [figura 4](#), además de esto se tienen versiones usando técnicas paralelas como reducción o atomic para mantener la eficiencia en la acumulación, como último se tiene una función de overloading que permite el uso de la directiva de private.

La sección processNodes aplica procesamiento paralelo de nodos con tareas (task) o paralelismo clásico (parallel for) para actualizar amplitudes con distintas estrategias, mostrado en la [figura 5](#). Otros métodos exploran cláusulas específicas de OpenMP como barrier para sincronizar fases de simulación, single para inicialización exclusiva de un hilo, y firstprivate/lastprivate para controlar variables privadas en paralelo. Así, la clase estructura la simulación física y su paralelización óptima manejando actualización, sincronización y métricas de manera flexible y eficiente.

Además de esta clase se realizó un “MetricsCalculator”, el cual está diseñado para calcular métricas físicas clave en la simulación, utilizando paralelismo con OpenMP para mejorar la eficiencia en el procesamiento de grandes redes de nodos. Implementa métodos para calcular de forma eficiente la energía total del sistema —sumando los cuadrados de las amplitudes de todos los nodos— y la amplitud promedio. Esta estructura permite obtener información relevante sobre el estado global de la onda y facilita el análisis de la evolución temporal y la conservación de energía en el sistema.



Como último, para el análisis de la simulación se generó la clase “Benchmarks”, la cual permite evaluar el rendimiento paralelo del modelo bajo diferentes configuraciones, abarcando comparaciones de schedule, sincronización, manejo de datos y escalabilidad en OpenMP.

Para cada prueba se mide el tiempo de ejecución serial y paralelo, calculando métricas clave como el speedup, la eficiencia, y el valor teórico máximo de aceleración mediante la Ley de Amdahl, estas están basadas en las fórmulas presentadas en la [figura 6](#). Estos ensayos se aplican tanto a la forma de repartir el trabajo entre hilos (por ejemplo, usando schedule static, dynamic y guided), como a la sincronización entre ellos (atomic, critical, barrier), el manejo de variables privadas o compartidas, y el procesamiento avanzado con tareas o inicialización exclusiva.

El análisis de estos resultados revela cómo la elección del esquema de paralelismo y los parámetros de chunk size influyen directamente en la mejora de rendimiento, mostrando que estrategias adaptativas como guided consiguen mayores speedups y mejor eficiencia que los modos estáticos en la mayoría de los casos. Además, la evaluación de la conservación de energía y la fracción serial valida la robustez del modelo físico, además de evidenciar los límites prácticos de escalabilidad al aumentar el número de threads. Estas pruebas permiten identificar cuellos de botella y áreas de posible optimización para maximizar el aprovechamiento de los recursos computacionales en simulaciones científicas.



### 3. Resultados experimentales

Se realizaron distintas simulaciones de las cuales se lograron obtener mediante los benchmarks los siguientes gráficos generados, estos se presentan en la sección de [anexos](#).

En la [figura 7](#) podemos observar que efectivamente se logra realizar una amortiguación dado el coeficiente implementado, ya que a medida que pasa el tiempo se observa como la energía realizada por el impulso va disminuyendo, además de esto se observa cómo el sistema evoluciona de forma estable, ya que se puede ver un descenso suave y continuo, además logra simular correctamente un evento físico real, ya que según la teoría la energía empieza a decaer de forma exponencial, al igual que lo muestra nuestro gráfico.

En cuanto a eficacia del código dado el número de chunks, podemos ver en la [figura 8](#) que cada vez que la cantidad aumenta, podemos observar como la parallelización obtenida de OpenMP permite un tiempo de ejecución menor, además de esto podemos observar que el cambio más drástico se encuentra entre 1 a 4 chunks, desde ahí no aumenta mucho la rapidez, por lo que una excesiva fragmentación podría generar overhead y no mayores beneficios.

En la [figura 9](#) se observa cómo debido a limitaciones con la paralelización (esto por ejemplo como a la hora de impresión de resultado o la propia asignación de valores) además de la sobrestimación que tiene la ley utilizada, podemos ver cómo a pesar de no conseguir el ideal, obtuvimos una mejora proporcional al teórico.

De estos gráficos presentados en la [figura 10](#) podemos obtener que mientras el speedup aumenta dado el número de hilos, la eficiencia por thread disminuye, esto se debe a que no se puede obtener toda la potencia paralela, debido a que parte del tiempo se pierde esperando la sincronización o por partes en las cuales no se puede distribuir entre los hilos, aún así podemos ver como el speedup aumenta dado el número de hilos logrando una mejora de rendimiento.

Como último se logró generar las visualizaciones correspondientes, en las cuales se observa como tiene el impulso inicial y luego tiene un ruido dirigido solo al nodo central, de tal manera que se ve como se disipa la energía a lo largo del plano además de incluir el impulso externo sinusoidal.



#### **4. Análisis de Performance**

A partir de estos dos gráficos presentados en la [figura 11](#), se aprecia claramente que al incrementar el tamaño de los chunks, el schedule tipo guided tiende a sobresalir, logrando mejores resultados tanto en speedup como en eficiencia, seguido por el dynamic y finalmente el static. Esto demuestra que en este tipo de simulación y carga de trabajo paralela, la opción guided es la más efectiva ya que distribuye el trabajo de forma más equilibrada entre los hilos y adapta dinámicamente el tamaño de los fragmentos, mientras que dynamic también mejora el balance pero con algo más de variabilidad y el método static resulta menos eficiente, especialmente para chunks pequeños. Por lo tanto, para este caso, la estrategia guided es la opción preferente para maximizar rendimiento y aprovechamiento de los recursos disponibles.



## **5. Conclusiones**

La implementación del simulador de propagación de ondas logró modelar correctamente el fenómeno físico, observando un decaimiento exponencial de la energía y un comportamiento estable a lo largo del tiempo, tal como predice la teoría. El uso de técnicas de paralelismo con OpenMP permitió optimizar el rendimiento y reducir los tiempos de ejecución, siendo especialmente notable el impacto positivo de la estrategia de scheduling guided frente a las alternativas static y dynamic, tanto en speedup como en eficiencia. Sin embargo, los resultados también evidencian las limitaciones prácticas derivadas de fracciones seriales y overhead de sincronización, lo que impide alcanzar el rendimiento ideal previsto por la Ley de Amdahl en escenarios reales.

El análisis global del desempeño señala que, aunque la eficiencia por hilo disminuye al aumentar la cantidad de threads, el speedup obtenido respalda el beneficio de la paralelización bien implementada en simulaciones científicas. La comparación entre las distintas configuraciones de chunks y schedules permitió identificar configuraciones óptimas para maximizar la utilización de los recursos y minimizar el tiempo de cómputo. En suma, el trabajo desarrollado valida tanto el diseño orientado a objetos como el uso de métodos numéricos y paralelos mediante la librería de OpenMP, y genera una base sólida para futuras mejoras y simulaciones más complejas, aportando herramientas útiles para el estudio y optimización de sistemas distribuidos en ingeniería y ciencias computacionales.



## 6. Anexos

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

Figura 1: Fórmula de Propagación de onda.

```
void Network::propagateWaves() {
    std::vector<double> newAmplitudes(getSize(), 0.0);
    double D = getDiffusionCoeff();
    double gamma = getDampingCoeff();
    double dt = getTimeStep();

    double rowsize_d = std::sqrt(static_cast<double>(getSize()));
    int rowsize = static_cast<int>(std::round(rowsize_d));
    int center = static_cast<int>(std::round((rowsize / 2.0) * rowsize + (rowsize / 2.0)));

    for (int i = 0; i < getSize(); ++i) {
        double sum_neighbors = 0.0;
        double Ai = getNodes()[i].getAmplitude();

        for (int neighborId : getNodes()[i].getNeighbors()) {
            sum_neighbors += getNodes()[neighborId].getAmplitude() - Ai;
        }

        double diffusion = D * sum_neighbors;
        double damping = -gamma * Ai;
        double delta = diffusion + damping;

        newAmplitudes[i] = Ai + delta * dt;

        if (i == center) {
            newAmplitudes[i] += noiseCoeff;
        }
    }

    for (int i = 0; i < getSize(); ++i) {
        getNodes()[i].updateAmplitude(newAmplitudes[i]);
    }
}
```

Figura 2: Extracto de código de la función “propagateWaves”



```
void WavePropagator::integrateEulerCore(std::vector<double>& newAmplitudes, double& totalEnergy, int syncType) {
    if (syncType == 0) {
        #pragma omp parallel for
        for (int i = 0; i < network.getSize(); ++i) {
            double Ai = network.getNodes()[i].getAmplitude();
            newAmplitudes[i] = Ai * 0.99;
            #pragma omp atomic
            totalEnergy += Ai * Ai;
        }
    } else if (syncType == 1) {
        #pragma omp parallel for
        for (int i = 0; i < network.getSize(); ++i) {
            double Ai = network.getNodes()[i].getAmplitude();
            newAmplitudes[i] = Ai * 0.99;
            #pragma omp critical
            {
                totalEnergy += Ai * Ai;
            }
        }
    } else if (syncType == 2) {
        #pragma omp parallel
        {
            #pragma omp for nowait
            for (int i = 0; i < network.getSize(); ++i) {
                double Ai = network.getNodes()[i].getAmplitude();
                newAmplitudes[i] = Ai * 0.99;
            }
        }
    }
}
```

Figura 3: Extracto de código de la función “IntegrateEulerCore”

```
double WavePropagator::calculateEnergy() {
    double totalEnergy = 0.0;

    for (int i = 0; i < network.getSize(); ++i) {
        double amp = network.getNodes()[i].getAmplitude();
        totalEnergy += amp * amp;
    }

    return totalEnergy;
}
```

Figura 4: Extracto de código de la función “calculateEnergy”.

```
void WavePropagator::processNodes() {
    for (int i = 0; i < network.getSize(); ++i) {
        double amp = network.getNodes()[i].getAmplitude();
        network.getNodes()[i].updateAmplitude(amp * 0.99);
    }
}
```

Figura 5: Extracto de código de la función “processNodes”.



$$S_p = \frac{T_1}{T_p} \quad E_p = \frac{S_p}{p} \quad S_p = \frac{1}{f + \frac{1-f}{p}}$$

Figura 6: Fórmulas de Speedup, de eficiencia y de la ley de Amdahl respectivamente.

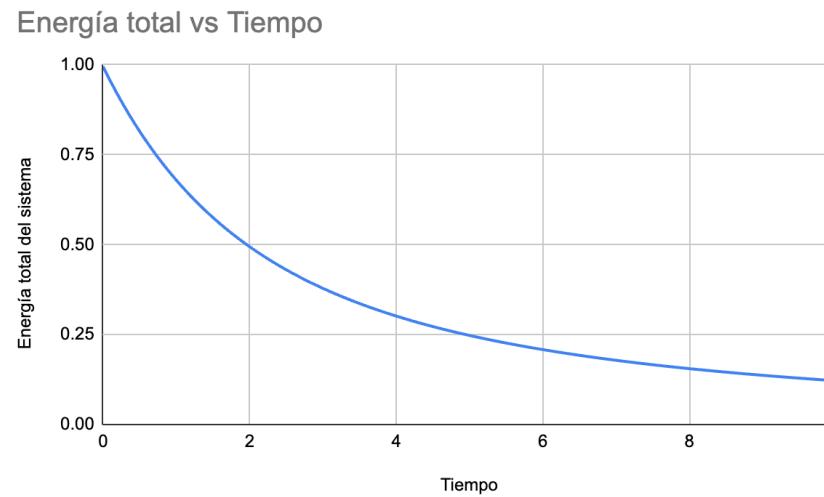


Figura 7: Gráfico generado mediante simulaciones con la obtención de energía total vs tiempo.

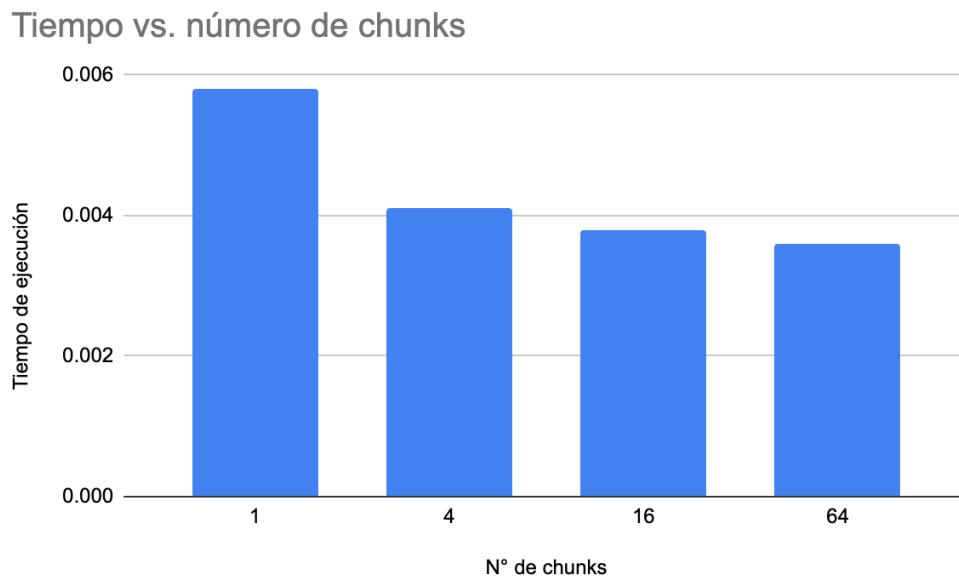




Figura 8: Gráfico generado mediante simulaciones con el tiempo de ejecución vs número de chunks.

Comparación de valor teórico vs speedup obtenido

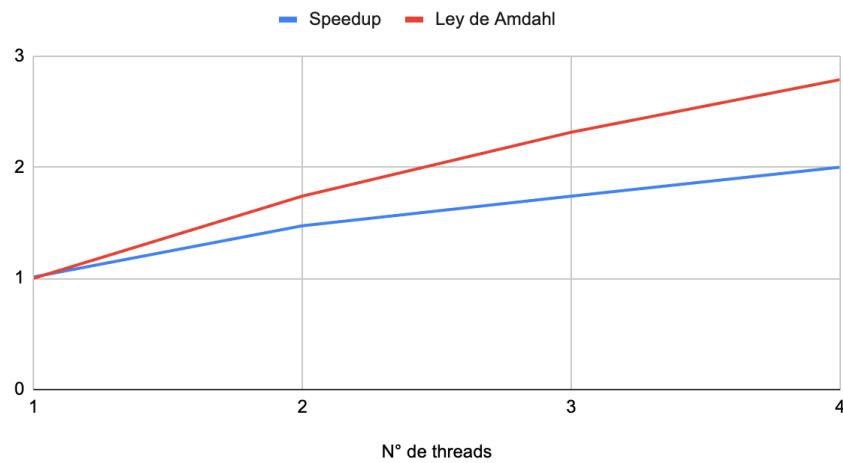




Figura 9: Gráfico generado mediante simulaciones con el valor teórico obtenido de la Ley de Amdahl y el Speedup obtenido.

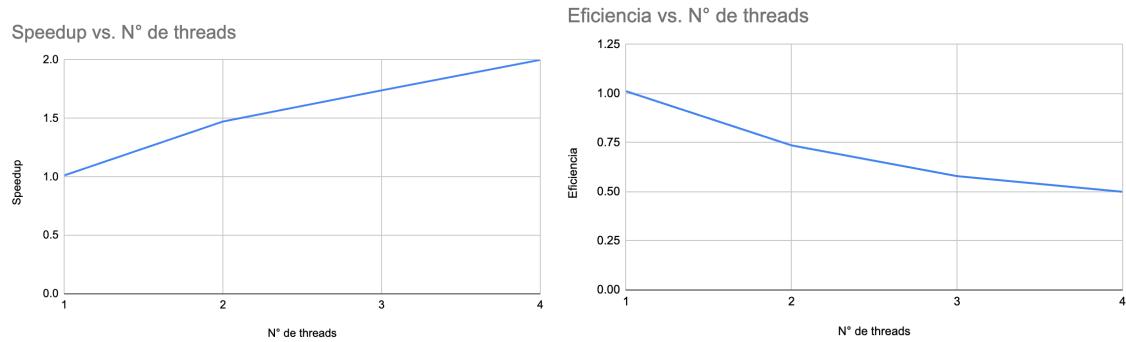


Figura 10: Gráficos generados mediante simulaciones con el speedup y el número de hilos utilizados, además de eficiencia y el número de hilos.

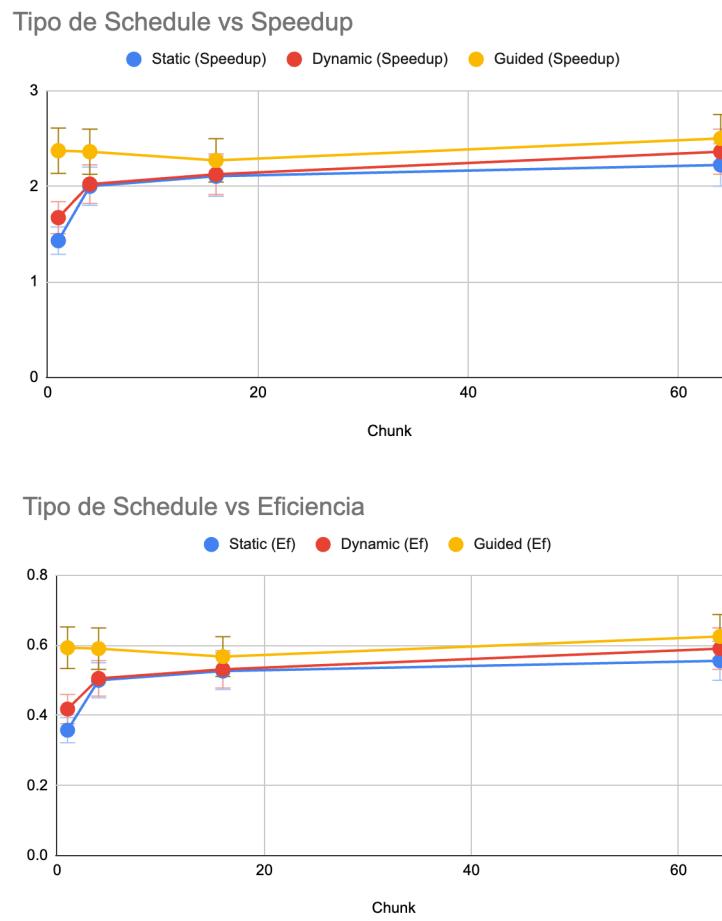


Figura 11: Gráficos generados mediante simulaciones con el speedup y el tipo de schedule, además de eficiencia y el tipo de schedule.