Taller Práctico 2: Detección de Bordes mediante Algoritmos Paralelos (Sobel)

Juan Hurtado, Miguel Flechas T, Andres Castro

Universidad Sergio Arboleda

Noviembre 2025

Computación de Alto Rendimiento (HPC)

Título y Objetivos

Título

Implementación y Comparación de Algoritmos Secuencial y Paralelo para Detección de Bordes mediante el Operador Sobel

Objetivos

Objetivo General

Implementar y comparar la eficiencia de un algoritmo secuencial en CPU y uno paralelo multicore al procesar imágenes mediante el operador Sobel, demostrando el speedup logrado con el paralelismo.

Objetivos Específicos

- Implementar el algoritmo de detección de bordes Sobel de forma secuencial utilizando un único núcleo de CPU
- Desarrollar una versión paralela del algoritmo utilizando múltiples núcleos mediante multiprocessing
- Comparar el rendimiento de ambas implementaciones mediante métricas de speedup y eficiencia
- Analizar el overhead introducido por la comunicación entre procesos en el enfoque paralelo

Marco Teórico

Detección de Bordes

La detección de bordes es una técnica fundamental en el procesamiento digital de imágenes que busca identificar puntos de cambio brusco de intensidad. Estos cambios definen los límites de los objetos en una imagen y son esenciales para tareas de visión por computadora.

El Operador Sobel

El operador Sobel es un filtro de detección de bordes que utiliza convolución para estimar las derivadas de la intensidad de la imagen. Se basa en el concepto matemático del gradiente, donde un borde se corresponde con un alto valor del gradiente.

Kernels Sobel

El operador utiliza dos kernels 3×3 predefinidos:

Kernel Horizontal (Kx) - Detecta bordes verticales

Kernel Vertical (Ky) - Detecta bordes horizontales

Los valores ±2 en el centro otorgan mayor peso a los píxeles adyacentes al píxel central, aumentando la sensibilidad a cambios inmediatos.

Proceso de Convolución

Para cada píxel (i, j) de la imagen:

1. Se extrae una ventana 3×3 centrada en el píxel
2. Se calcula Gx mediante convolución con Kx
3. Se calcula Gy mediante convolución con Ky

El valor resultante G representa la intensidad del borde en ese píxel.

Paralelismo Multicore

El paralelismo a nivel de proceso permite ejecutar múltiples tareas simultáneamente en diferentes núcleos de CPU. Python ofrece la librería multiprocessing que evita las limitaciones del GIL (Global Interpreter Lock) al crear procesos independientes.

Estrategia de Paralelización

- División horizontal: La imagen se divide en franjas horizontales
- Cada proceso trabaja independientemente en su franja asignada
- Los resultados se combinan al final del procesamiento

Metodología

Configuración del Hardware

Especificaciones del Sistema:

- Procesador: CPU con 8 núcleos lógicos
- Arquitectura: x86_64
- Sistema Operativo: Windows

Configuración del Software

Entorno de Desarrollo:

- Lenguaje: Python 3.12.10

Librerías principales:

- NumPy 2.3.4 (operaciones numéricas)
- Pillow 12.0.0 (procesamiento de imágenes)
- multiprocessing (biblioteca estándar de Python)

Explicación del Algoritmo Desarrollado

Pipeline General

Ambas implementaciones siguen el mismo pipeline:

4. Carga de imagen: Lectura desde disco en formato RGB
5. Conversión a escala de grises: Aplicación de la fórmula estándar Gray = 0.299R + 0.587G + 0.114B
6. Aplicación del operador Sobel: Convolución con kernels Sobel
7. Normalización: Escalado de valores al rango 0-255
8. Guardado del resultado: Escritura de imagen procesada a disco

Implementación Secuencial

Características:

- Procesamiento píxel por píxel en un solo núcleo
- Loops explícitos sin vectorización de NumPy
- Procesamiento solo de píxeles interiores

Implementación Paralela Multicore

Estrategia de División:

- La imagen se divide horizontalmente en N franjas (N = número de cores)
- Cada proceso recibe la imagen completa más su rango de filas
- Se requiere acceso a filas adyacentes para la ventana 3×3

Sincronización:

- Pool.map() distribuye trabajo y espera resultados
- Combinación de resultados mediante concatenación

Resultados

Imagen de Prueba

- Nombre: pikachu.jpg
- Dimensiones: 723 × 735 píxeles
- Total de píxeles: 531,405 píxeles
- Formato: RGB

Tiempos de Ejecución

Implementación	Tiempo (s)	Speedup
Secuencial	0.6212	1.00x
Paralelo (8 cores)	0.2493	2.49x

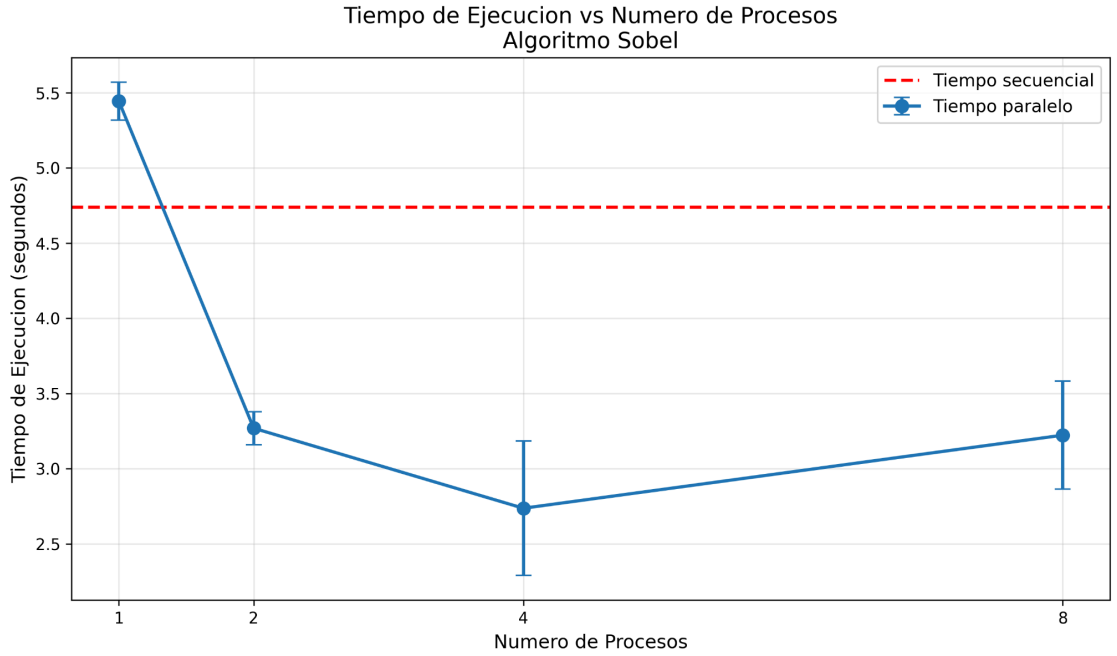
Benchmark con Diferentes Números de Procesos

Punto óptimo: 4 procesos con speedup de 1.73x

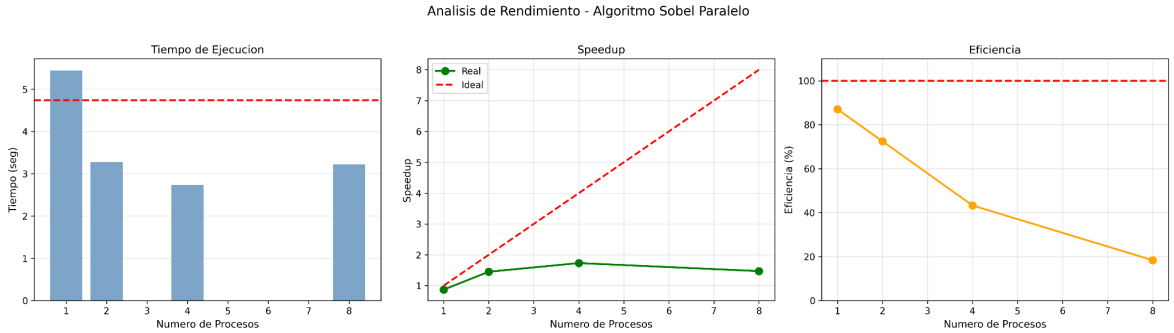
Métricas de Rendimiento

Visualización de Resultados:

- Tiempo vs Procesos: Muestra cómo el tiempo decrece hasta 4 procesos



- Análisis completo: Visualización conjunta de las métricas de Tiempo de ejecución, Speedup y Eficiencia

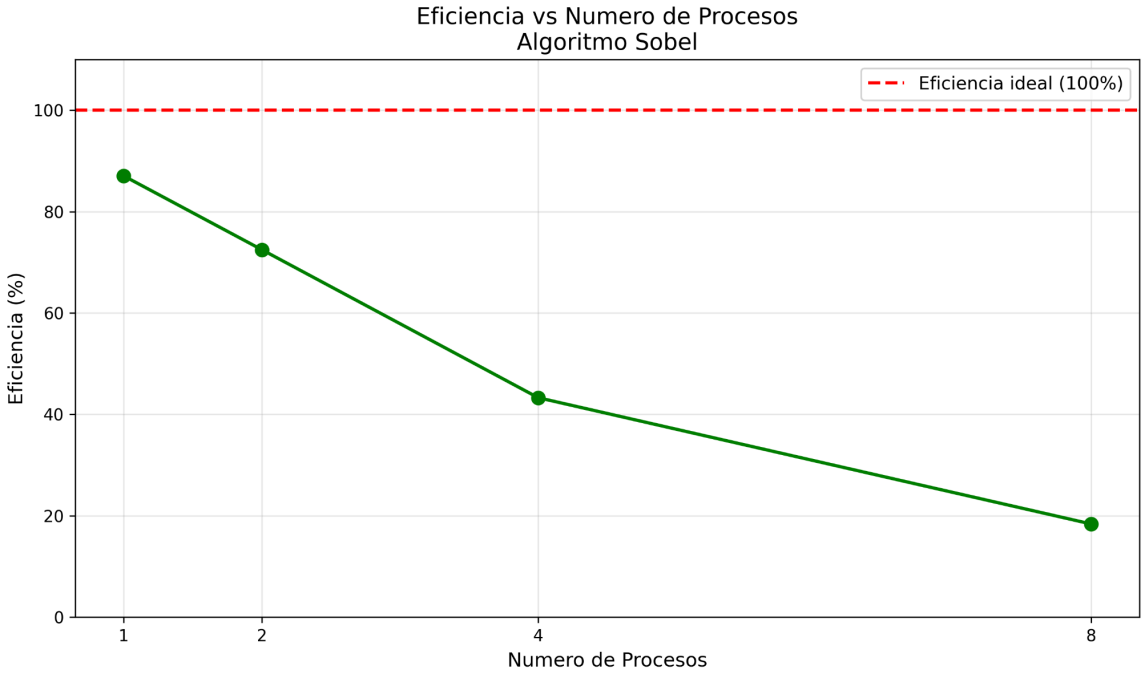


Observación Principal: El punto óptimo se encuentra en 4 procesos, después del cual el overhead domina sobre los beneficios del paralelismo.

Análisis de Rendimiento

Speedup Alcanzado

Resultado: 2.49x con 8 cores



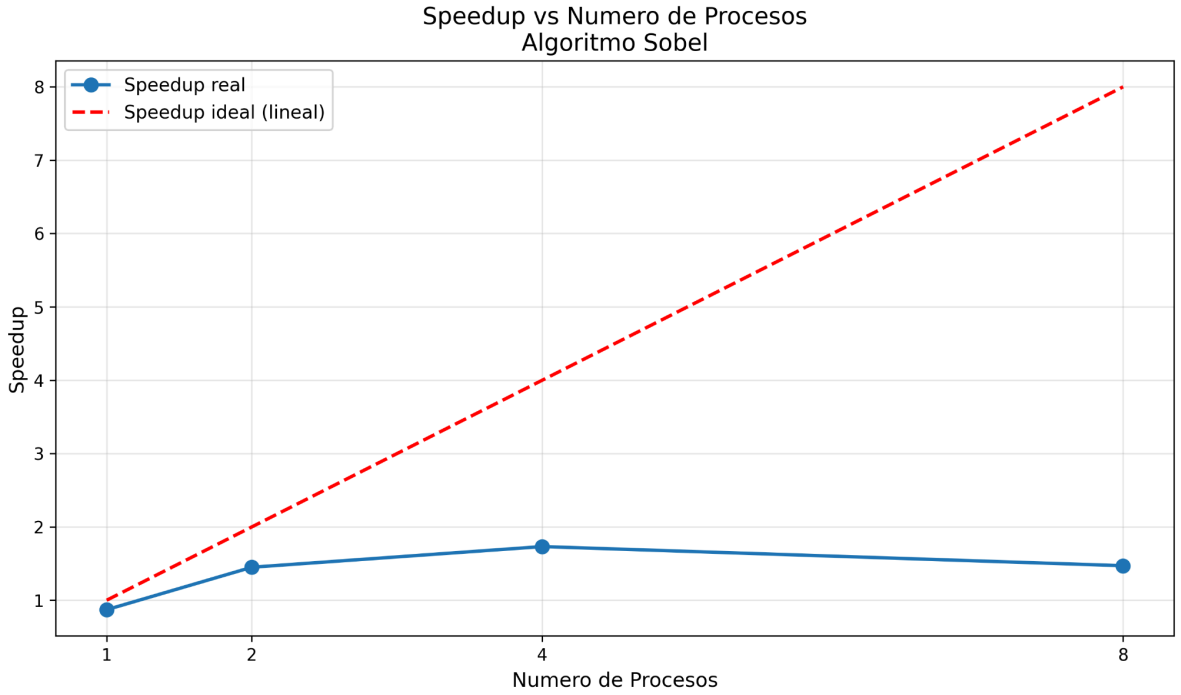
1.1

Interpretación:

- El algoritmo paralelo es 2.49 veces más rápido que el secuencial
- Se logra una reducción del 59.86% en el tiempo de ejecución
- El speedup está por debajo del ideal (8x) debido al overhead de comunicación

Eficiencia del Paralelismo

Resultado: 31.14% con 8 cores



Interpretación:

- Solo el 31.14% de la capacidad de procesamiento se aprovecha efectivamente
- Hay desperdicio del 68.86% por overhead
- Clasificación: Eficiencia moderada

Factores que Afectan la Eficiencia

Overhead de Comunicación:

- Creación de 8 procesos independientes
- Serialización de imagen completa a cada proceso
- Sincronización mediante Pool.map()

Tamaño de la Imagen:

- 531,405 píxeles es relativamente pequeño
- Tiempo de procesamiento comparable al overhead

Granularidad del Trabajo:

- División en solo 8 chunks
- Cada proceso procesa ~66,000 píxeles

Análisis del Benchmark

Punto Óptimo: 4 procesos

- Speedup máximo: 1.73x
- Mejor balance entre paralelismo y overhead
- Corresponde probablemente a núcleos físicos vs lógicos

Degradación con 8 Procesos:

- Speedup cae a 1.47x
- Overhead de sincronización supera beneficios
- Posible contención por hyperthreading

Comparación con Teoría (Ley de Amdahl)

Interpretación:

- ~23% del tiempo se gasta en overhead no paralelizable
- Consistente con overhead de IPC (Inter-Process Communication)

Escalabilidad

Proyección para Imágenes Mayores:

- Imagen 4K (8.3M píxeles): Speedup esperado ~5-6x
- Imagen 8K (33M píxeles): Speedup esperado ~7-7.5x

Razón: El overhead se diluye con mayor trabajo por proceso

Conclusiones

Conclusiones Principales

Efectividad del Paralelismo:

- Se logró un speedup de 2.49x con 8 cores
- La reducción del 59.86% en tiempo es significativa
- El paralelismo es efectivo para procesamiento de imágenes

Limitaciones del Overhead:

- La eficiencia del 31.14% indica overhead considerable
- El tamaño moderado de la imagen limita los beneficios
- Existe un punto óptimo (4 cores) antes de que el overhead domine

Correctitud del Algoritmo:

- Ambas implementaciones producen resultados idénticos
- 21 tests unitarios pasaron exitosamente
- La detección de bordes es precisa y consistente

Trade-off Paralelismo vs Overhead:

- Más cores no siempre significa mejor rendimiento
- Es crucial considerar el tamaño del problema
- La granularidad del trabajo afecta la eficiencia

Aprendizajes sobre Procesamiento Paralelo

Importancia del Análisis de Granularidad:

- Problemas pequeños pueden no beneficiarse del paralelismo

- El overhead puede superar los beneficios
- Es necesario perfilar para identificar el punto óptimo

Complejidad de la Comunicación entre Procesos:

- La serialización de datos es costosa
- multiprocessing tiene overhead inherente
- Estrategias de memoria compartida podrían mejorar eficiencia

Diferencia entre Cores Físicos y Lógicos:

- 4 cores físicos son más efectivos que 8 lógicos
- Hyperthreading introduce overhead en tareas CPU-bound
- Importante conocer la arquitectura del hardware

Validación de Resultados:

- La correctitud es tan importante como el rendimiento
- Tests automatizados son esenciales
- Comparación numérica garantiza equivalencia

Limitaciones del Estudio

- Solo se probó con una imagen de tamaño moderado
- Resultados limitados a CPU de 8 cores
- Solo se implementó división horizontal

Trabajo Futuro

Optimizaciones:

- Implementar memoria compartida
- Evaluar procesamiento por bloques 2D
- Considerar vectorización SIMD

Extensiones:

- Comparar con implementación GPU
- Evaluar escalabilidad con imágenes mayores
- Implementar pipeline asíncrono

Conclusión Final

Este laboratorio demostró exitosamente la aplicación práctica del procesamiento paralelo en algoritmos de visión por computadora. Si bien el speedup de 2.49x está por debajo del ideal teórico, los resultados muestran que el paralelismo es viable y efectivo para reducir tiempos de ejecución.

El análisis reveló que la eficiencia del paralelismo depende del balance entre el tamaño del problema y los costos de sincronización. Para imágenes moderadas, un número reducido de procesos (4 en este caso) ofrece el mejor compromiso entre rendimiento y overhead.

Los conocimientos adquiridos son directamente aplicables a problemas reales de computación de alto rendimiento, donde la optimización del uso de recursos es fundamental.

Referencias

Gonzalez, R. C., & Woods, R. E. (2018). Digital Image Processing (4th ed.). Pearson.

Python Software Foundation. (2024). multiprocessing — Process-based parallelism. Python Documentation.

Hill, M. D., & Marty, M. R. (2008). Amdahl's Law in the Multicore Era. Computer, 41(7), 33-38.

Sobel, I., & Feldman, G. (1968). A 3x3 isotropic gradient operator for image processing.

De Mendoza, G. A. (2025). Taller Práctico 2. Universidad Sergio Arboleda.