Evaluación de Estrategias de Programación Paralela y Distribuida en Procesamiento de Imágenes

Juan Collazos

31 de agosto de 2025

1. Introducción

El presente trabajo tiene como objetivo evaluar diferentes enfoques de programación paralela y distribuida aplicados al procesamiento de imágenes digitales. Se implementaron cuatro versiones de un filtro de imágenes:

1. **Secuencial (baseline)**
2. **Pthreads (memoria compartida con hilos)**
3. **OpenMP (directivas de paralelismo automático en memoria compartida)**
4. **MPI (memoria distribuida con paso de mensajes)**

Cada implementación fue probada sobre un conjunto de imágenes en formato `PGM` (escala de grises) y `PPM` (color). Se analizaron los tiempos de ejecución medidos en *CPU time* y en *wall-clock time*, con el fin de comparar eficiencia y escalabilidad.

2. Metodología

- **Lenguaje**: C++17
- **Compiladores**: `g++`, `mpic++`
- **Plataforma de ejecución**: contenedores Docker configurados con soporte para MPI, Pthreads y OpenMP.
- **Filtros aplicados**: *blur*, *laplace* y *sharpening*.
- **Mediciones**:
  - **CPU time**: tiempo total de uso de procesador (sumado en todos los hilos/nodos).
  - **Wall-clock time**: tiempo real transcurrido en la ejecución.

3. Resultados

3.1. Ejecución Secuencial

En la Tabla 1 se presentan los resultados de la ejecución secuencial para diferentes imágenes y filtros. Es importante notar que, aunque los filtros aplicados poseen la misma complejidad computacional, el formato de la imagen influye significativamente en el tiempo de procesamiento. Las imágenes en formato `.ppm` utilizan representación RGB (tres valores por píxel), mientras que las imágenes en formato `.pgm` se representan en escala de grises (un único valor por píxel). Por esta razón, los tiempos para las imágenes en `.ppm` suelen ser mayores en comparación con sus equivalentes en `.pgm`, especialmente en resoluciones grandes.

Imagen	**Formato**	**Dimensiones**	**CPU (s)**	**Wall (s)**
lena	PPM (RGB)	128×128	0.018	0.023
lena	PGM (Gray)	512×512	0.156	0.159
fruit	PPM (RGB)	900×450	0.349	0.356
fruit	PGM (Gray)	900×450	0.241	0.270
puj	PPM (RGB)	1950×600	1.008	1.029
puj	PGM (Gray)	1950×600	0.720	0.726

Cuadro 1: Tiempos de ejecución secuencial para distintas imágenes y formatos

3.2. Pthreads

La Tabla 2 muestra los resultados obtenidos utilizando memoria compartida con *Pthreads*, fijando el número de hilos en 4. En este caso, el tiempo de CPU corresponde a la suma del uso de los cuatro hilos, mientras que el tiempo de *wall-clock* refleja el tiempo real de ejecución. Se observa que, aunque el consumo total de CPU es mayor que en la versión secuencial, el tiempo real se reduce significativamente gracias a la ejecución en paralelo. Al igual que en el caso secuencial, el formato de la imagen influye de manera importante en el desempeño: las imágenes en formato `.ppm` (RGB) requieren más operaciones por píxel que las `.pgm` (escala de grises).

Imagen	**Formato**	**Dimensiones**	**CPU (s)**	**Wall (s)**
damma	PPM (RGB)	1000×1278	1.246	0.826
damma	PGM (Gray)	1000×1278	1.303	0.836
sulfur	PPM (RGB)	823×1000	0.783	0.515
sulfur	PGM (Gray)	823×1000	0.576	0.312

Cuadro 2: Tiempos de ejecución con Pthreads (4 hilos) para distintas imágenes y formatos

3.3. OpenMP

En este caso, la paralelización con *OpenMP* se implementó únicamente sobre el ciclo que aplica los filtros a la imagen original. De esta forma, los tres filtros se ejecutan de

manera simultánea, generando tres imágenes de salida en cada ejecución. El resto del procesamiento conserva la estructura de la versión secuencial.

La Tabla 3 resume los resultados. Se observa que, aunque el tiempo de CPU (suma de todos los hilos) aumenta respecto a la versión secuencial, el tiempo real de ejecución (*wall-clock*) se reduce de forma considerable gracias a la ejecución paralela. Además, como en los casos anteriores, el formato `.ppm` (RGB) requiere más operaciones que el formato `.pgm` (escala de grises), reflejándose en tiempos mayores.

Imagen	**Formato**	**Dimensiones**	**CPU (s)**	**Wall (s)**
sulfur	PPM (RGB)	823×1000	1.815	0.636
sulfur	PGM (Gray)	823×1000	1.374	0.470
damma	PPM (RGB)	1000×1278	2.759	0.959
damma	PGM (Gray)	1000×1278	2.318	0.827

Cuadro 3: Tiempos de ejecución con OpenMP para distintas imágenes y formatos (3 filtros aplicados en paralelo)

3.4. MPI

La implementación con *MPI* se ejecutó sobre contenedores Docker configurados con una imagen Debian que incluía `mpi` y `ssh` para la comunicación entre procesos. Se levantaron tres contenedores conectados mediante una red virtual de Docker, todos en la misma máquina física. Este entorno permite validar la funcionalidad del sistema distribuido, aunque no representa un clúster real, ya que la comunicación se limita a la red virtual del host.

La Tabla 4 muestra los resultados obtenidos al ejecutar con 3 procesos (`-np 3`). Como se observa, los tiempos de CPU individuales son menores que en la versión secuencial, pero el *wall-clock* se ve afectado por el overhead de comunicación y sincronización entre procesos. Este comportamiento es esperable en entornos de red virtualizados y sin distribución física real de los nodos.

Imagen	**Rank**	**Dimensiones**	**CPU (s)**	**Wall (s)**
puj.pgm	0	1950×600	0.884	1.464
	1		0.780	1.175
	2		0.917	1.268
sulfur.pgm	0	823×1000	0.627	1.010
	1		0.517	0.776
	2		0.559	0.790

Cuadro 4: Tiempos de ejecución con MPI (3 procesos sobre contenedores Docker en la misma máquina física)

4. Análisis Comparativo

La Tabla 5 resume las ventajas y desventajas principales de cada estrategia de programación implementada. Se observa que, mientras la versión secuencial sirve como referencia

base, las versiones paralelas permiten reducir de manera significativa el *wall-clock*, aunque a costa de un mayor consumo de CPU y, en el caso de MPI, con un notable overhead de comunicación. El formato de la imagen (`PGM` vs `PPM`) también tiene un impacto en todos los enfoques, siendo más costoso procesar imágenes a color (RGB).

Estrategia	**Ventajas**	**Desventajas**
Secuencial	▪ Implementación más simple. ▪ Sirve como línea base para comparación.	▪ Escalabilidad nula. ▪ Tiempos elevados en imágenes grandes, especialmente en formato `PPM`.
Pthreads	▪ Permite control detallado sobre los hilos. ▪ Reducción importante en *wall-clock* con 4 hilos. ▪ Aprovecha bien la memoria compartida.	▪ Mayor complejidad de programación. ▪ El tiempo de CPU crece al sumar todos los hilos. ▪ Posible overhead por sincronización.
OpenMP	▪ Simplifica el paralelismo mediante directivas. ▪ Genera tres imágenes de salida en paralelo a partir de la misma entrada. ▪ Buena reducción en *wall-clock*.	▪ Menos control fino que Pthreads. ▪ Mayor consumo de CPU al ejecutar múltiples filtros en paralelo. ▪ Escalabilidad limitada a un solo nodo.
MPI	▪ Escalable a múltiples nodos. ▪ Adecuado para imágenes muy grandes o clústeres reales. ▪ Modelo de paso de mensajes explícito.	▪ Overhead de comunicación significativo. ▪ En este experimento, los contenedores Docker se ejecutaron en la misma máquina física, limitando los beneficios reales. ▪ Peor rendimiento en imágenes pequeñas o medianas.

Cuadro 5: Comparación de estrategias de programación paralela y distribuida

5. Conclusiones

1. La versión **secuencial** resulta adecuada como línea base, ya que permite contrastar el impacto del paralelismo. Sin embargo, su escalabilidad es nula y los tiempos crecen rápidamente en imágenes grandes, especialmente en formato `PPM` (RGB).

2. La implementación con **Pthreads** demuestra un uso eficiente de la memoria compartida, logrando reducciones significativas en el *wall-clock*. No obstante, este beneficio viene acompañado de un mayor consumo total de CPU y de una complejidad de programación considerable.

3. **OpenMP** logra un balance favorable entre facilidad de uso y desempeño. Su enfoque basado en directivas simplifica la paralelización y permite aplicar múltiples filtros en paralelo con tiempos reales reducidos. Es la opción más práctica para sistemas de memoria compartida con imágenes de tamaño medio o grande.

4. La implementación con **MPI**, ejecutada en contenedores Docker sobre una misma máquina física, confirmó la funcionalidad del modelo distribuido, pero evidenció un overhead de comunicación importante. En este escenario, el rendimiento fue inferior al de Pthreads y OpenMP. Sin embargo, su verdadero potencial radica en entornos de múltiples nodos físicos, donde puede escalar a clústeres reales y procesar imágenes de gran tamaño.

5. En términos prácticos:

   - Para imágenes pequeñas o medianas en un solo nodo, **OpenMP** ofrece el mejor compromiso entre simplicidad y rendimiento.
   - Para imágenes grandes en entornos de memoria compartida, **Pthreads** permite un control más fino, a costa de mayor complejidad.
   - Para procesamiento distribuido en múltiples máquinas, **MPI** es la alternativa más adecuada, siempre que se disponga de un clúster real que amortigüe el costo de comunicación.

En conclusión, la elección de la estrategia depende del **tamaño de la imagen** y del **entorno de ejecución disponible**: *OpenMP resulta la opción más eficiente en memoria compartida, mientras que MPI constituye la herramienta fundamental para escalar a infraestructuras distribuidas.*