Máster en Ingeniería Informática

Computación de Altas Prestaciones 2024-2025
Grupo 1

Práctica 2

"Paralelización de código con MPI + OpenMP"

Luis Daniel Casais Mezquida – 100429021
Lucas Gallego Bravo – 100429005
Francisco Montañés de Lucas – 100406009
Diego Picazo García – 100549459

Equipo E

Profesor

Jesús Carretero

Índice

Parte I

Introducción

La mejora de contraste es una operación fundamental en el procesamiento de imágenes, ampliamente utilizada en diversos campos científicos y tecnológicos.

Entre las técnicas más utilizadas, la ecualización de histograma destaca por su efectividad y simplicidad. A través del análisis y transformación del histograma, es posible identificar y corregir regiones con intensidades predominantes, obteniendo un resultado visual más equilibrado y de mayor calidad.

El presente trabajo tiene como objetivo optimizar una aplicación secuencial para la mejora de contraste de imágenes mediante la ecualización del histograma, aprovechando las capacidades de paralelización ofrecidas por MPI y OpenMP.

El documento presenta un análisis del programa secuencial, una implementación OpenMP, MPI y un formato híbrido que aproveche ambos *frameworks*.

En cada sección, se incluye una descripción metodológica basada en las cuatro fases de paralelización: descomposición, asignación, orquestación y reparto. Además, se presentan gráficos que muestran los tiempos de ejecución y la aceleración lograda en comparación con la versión secuencial.

Finalmente, las conclusiones recogen un análisis comparativo del rendimiento de las distintas estrategias de paralelización, destacando sus ventajas, limitaciones y casos de uso óptimos. Este trabajo tiene como meta no solo mejorar el tiempo de ejecución en el procesamiento del contraste de imágenes, sino también explorar y comprender cómo los paradigmas de paralelismo pueden ser utilizados para resolver problemas computacionalmente intensivos de manera eficiente.

Parte II

Desarrollo

1. Programa secuencial

El programa base, presentado en el enunciado de la práctica, ofrece una ecualización de histograma, así como sus respectivos pasos para calcularlo.

Está presente un reparto homogéneo de iteraciones y cálculos redundantes, con gran cantidad de accesos y vectores. La Figura 1, Figura 2 y Figura 3 recogen las métricas de ejecución obtenidas en las máquinas del laboratorio, Avignon. Será turno de los desarrolladores y de los *frameworks* OpenMP y MPI reducir los tiempos, además de mejorar la eficiencia y eficacia al programa.

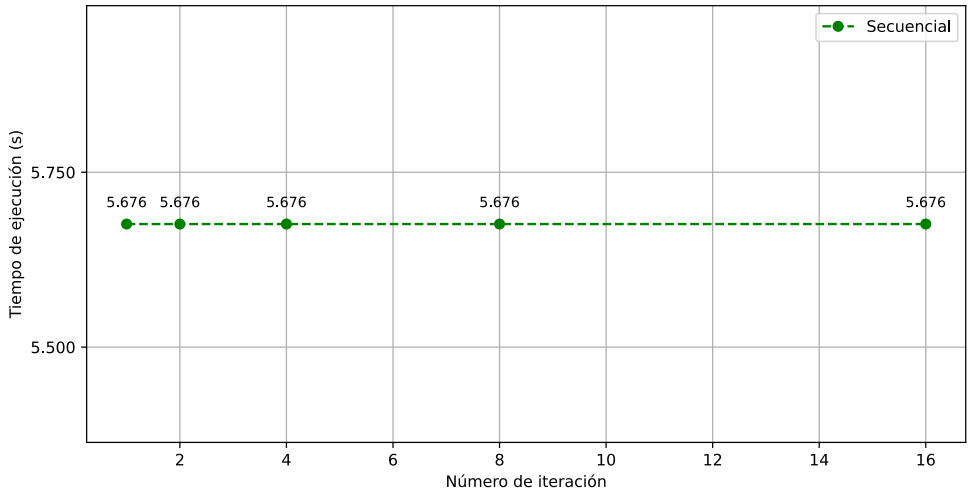

Fig. 1: *Tiempo de ejecución HSL del programa secuencial.*

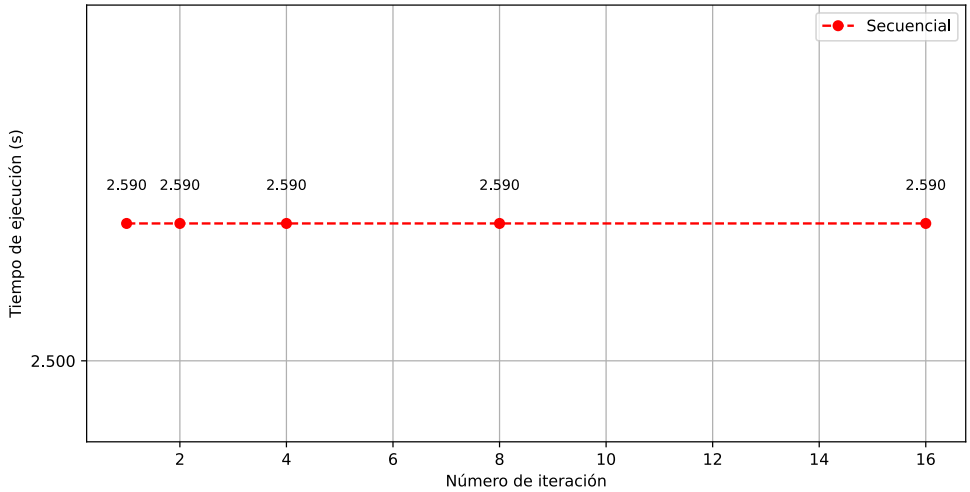

Fig. 2: *Tiempo de ejecución YUV del programa secuencial.*

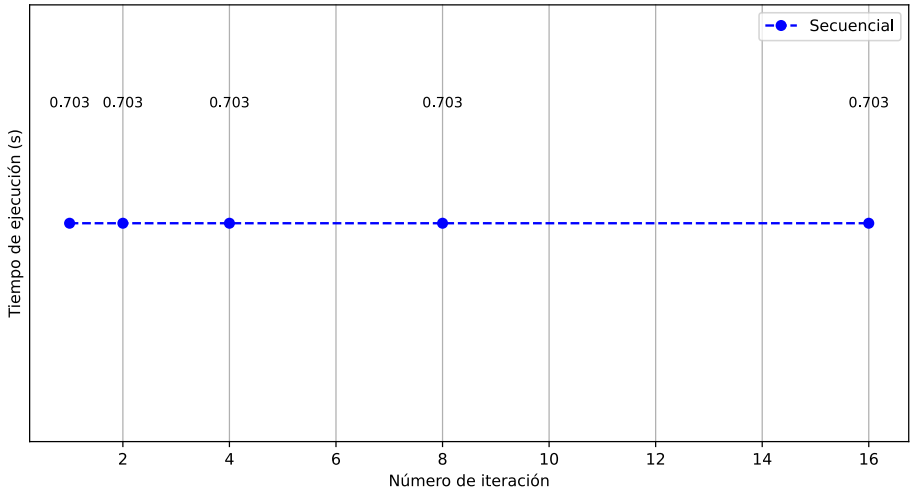

Fig. 3: *Tiempo de ejecución del procesado de grises del programa secuencial.*

2. Paralelización con OpenMP

A continuación, en esta sección, se expone un estudio de comportamiento del *framework* OpenMP, que incluye la metodología empleada en cada caso, medidas de tiempo y aceleración, además de una interpretación de estas a través de gráficas.

2.1. Metodología

La metodología de paralelización aplicada se desarrolla siguiendo las cuatro fases clave presentadas en el enunciado de la práctica: descomposición, asignación, orquestación y reparto. Estas fases permiten establecer una dinámica de trabajo para transformar un conjunto de operaciones secuenciales en un código paralelizado eficiente y escalable, aprovechando al máximo los recursos disponibles. A continuación, se incluye la implementación de estas fases en los puntos críticos, previamente identificados, utilizando **OpenMP**.

En la fase de **descomposición**, el objetivo principal es identificar las tareas que pueden ejecutarse de forma concurrente. Esto se ha logrado aislando las operaciones repetitivas que trabajan de forma independiente sobre píxeles o datos individuales. Por ejemplo, en las funciones de lectura y escritura de imágenes (`read_ppm()` y `write_ppm()`, caso 1 y 2 de ahora en adelante), cada iteración del bucle procesa un píxel independiente, separando o combinando los canales RGB en sus respectivas estructuras de datos. Se han observado operaciones independientes y similares a casos anteriores entre píxeles en las transformaciones de color RGB a HSL y viceversa (`rgb2hsl()` y `hsl2rgb()`, caso 3 y 4 de ahora en adelante), así como en las conversiones entre los espacios de color RGB y YCbCr (`rgb2yuv()` y `yuv2rgb()`, caso 5 y 6 de ahora en adelante). En estos casos, cada repetición aplica cálculos matemáticos que no dependen de los resultados de otras iteraciones, lo que permite una descomposición natural a nivel de píxel. Adicionalmente, se incluyen procesos más complejos como el cálculo de histogramas (`histogram()`, de ahora en adelante caso 7), donde la descomposición implica dividir los píxeles entre los hilos y acumular los resultados en histogramas locales. Finalmente, en operaciones de reducción como la búsqueda del mínimo no nulo en un histograma (`histogram_equalization()`, caso 8 de ahora en adelante) y las transformaciones basadas en una tabla de búsqueda (`histogram_equalization()`, caso 9 de ahora en adelante), la descomposición se ha realizado a nivel de elementos individuales, como valores del histograma o píxeles.

Una vez identificadas las tareas, se procede a la fase de **asignación**, distribuyendo las cargas de trabajo entre los recursos disponibles utilizando hilos. En la mayoría de los casos, como en las operaciones de lectura, escritura y conversiones de color, se ha empleado la directiva `#pragma omp parallel for` para repartir las iteraciones de los bucles entre los hilos de forma automática, aunque para las métricas posteriores se controla su reparto con fines educativos. Esta aproximación resulta eficiente debido a la homogeneidad de las operaciones en cada repetición. Para tareas más específicas, como el cálculo de histogra-

mas, caso 7, se asignan bloques de píxeles a cada hilo, generando histogramas parciales que posteriormente se combinaron. En la búsqueda del mínimo del histograma, caso 8, se permite que los hilos trabajen en diferentes rangos del histograma, con le objetivo de asegurar una cobertura completa del espacio de datos.

La **orquestación** es la próxima etapa, crucial para garantizar que los hilos trabajen de manera coordinada y así evitar conflictos. En tareas completamente independientes, como las transformaciones de píxeles (casos 1-6 y 9), no es necesario emplear sincronización adicional. Sin embargo, en procesos como el cálculo de histogramas (casos 7, 8 y 9), donde múltiples hilos actualizan un resultado global, se han introducido secciones críticas, `#pragma omp critical`, para combinar los histogramas parciales en un único histograma global. Esta sincronización también se utiliza en la búsqueda del mínimo en el histograma, caso 8, donde una sección crítica permite asegurar que la variable compartida `min` se actualice de manera coherente, sin dar lugar a problemas típicos de memoria compartida.

Por último, en la fase de **reparto**, es vital definir cómo distribuir las tareas de manera eficiente. Para la mayoría de las operaciones homogéneas, en los casos 1-6 y 9, se opta por un reparto estático `schedule(static)`, dividiendo equitativamente las iteraciones entre los hilos para minimizar una posible sobrecarga en el planificador del procesador. En el cálculo de histogramas, el reparto es implícito, dado que cada hilo procesa un subconjunto de píxeles. En aquellos procesos donde la carga de trabajo puede variar dinámicamente, como la búsqueda del mínimo no nulo en el histograma, la implementación incluye un punto de sincronización adicional, aunque no requiere un reparto explícito debido a la naturaleza del problema.

En conjunto, esta metodología permite identificar, asignar, orquestar y repartir el trabajo de forma paralela en puntos clave del programa, logrando una ejecución más eficiente sin comprometer la precisión de los resultados. Las decisiones tomadas en cada fase aseguran un balance adecuado entre simplicidad, rendimiento y escalabilidad.

2.2. Métricas

En esta sección, se incluyen las métricas obtenidas durante la ejecución del programa con paralelismo **OpenMP**. Quedan recogidas representaciones gráficas, agrupados dos a dos (HSL y YUV para color y grises), un tipo de representación expone una comparativa con la ejecución secuencial en términos de tiempo, otra en términos de aceleración y una tercera recoge una equiparación del *speed-up* empírico versus la **ley de Gustafson**. Se dispone de un análisis posterior de las mismas, con el objetivo de definir conclusiones que demuestren qué tan buen rendimiento puede llegar a obtener OpenMP con la optimización implementada. [1]

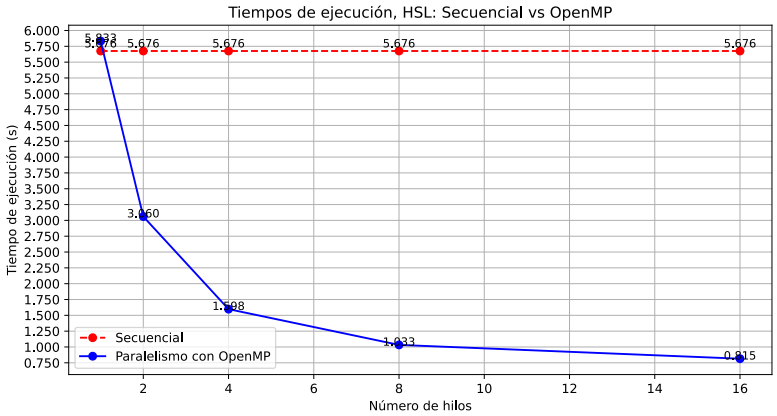

Fig. 4: *Comparativa de tiempos de ejecución HSL.*

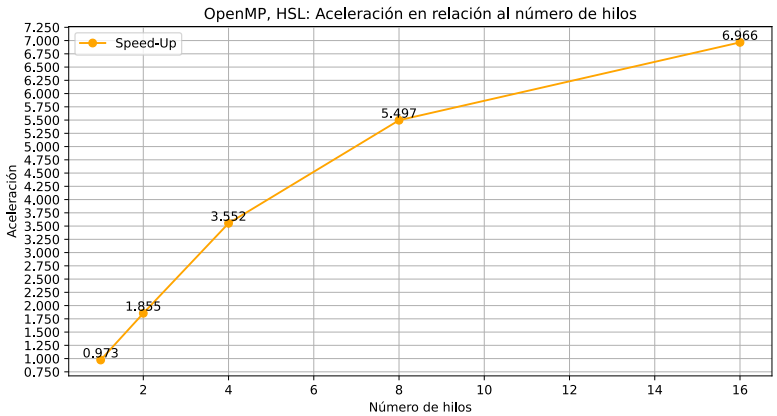

Fig. 5: *Aceleración empírica del programa OpenMP procesando HSL respecto al programa secuencial.*[1]

[1] La fórmula utilizada para medir el *speed-up* experimental es Speed-up $= \frac{T_{sn}}{T_{pn}}$, donde T_{s_n} es el tiempo secuencial medido con n procesos y T_{p_n} es el tiempo medido del programa paralelo con OpenMP utilizando n hilos. Esto genera un margen de error asumible que podría ser solventado con la Ley de Amdahl.

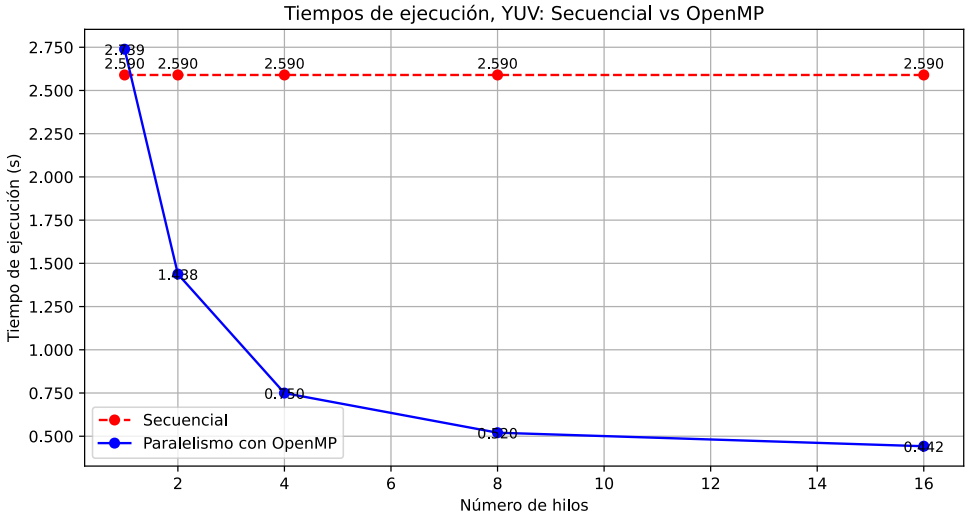

Fig. 6: *Comparativa de tiempos de ejecución YUV.*

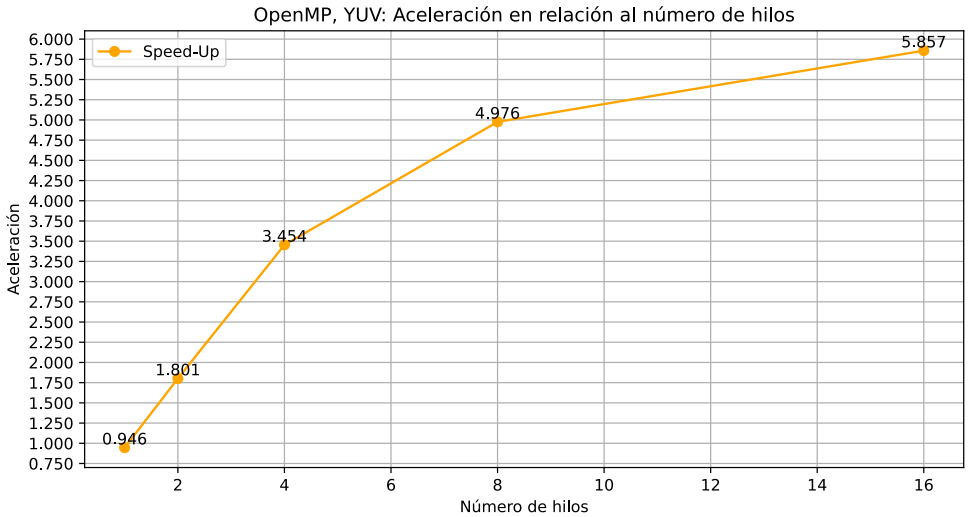

Fig. 7: *Aceleración empírica del programa OpenMP procesando YUV respecto al programa secuencial.*

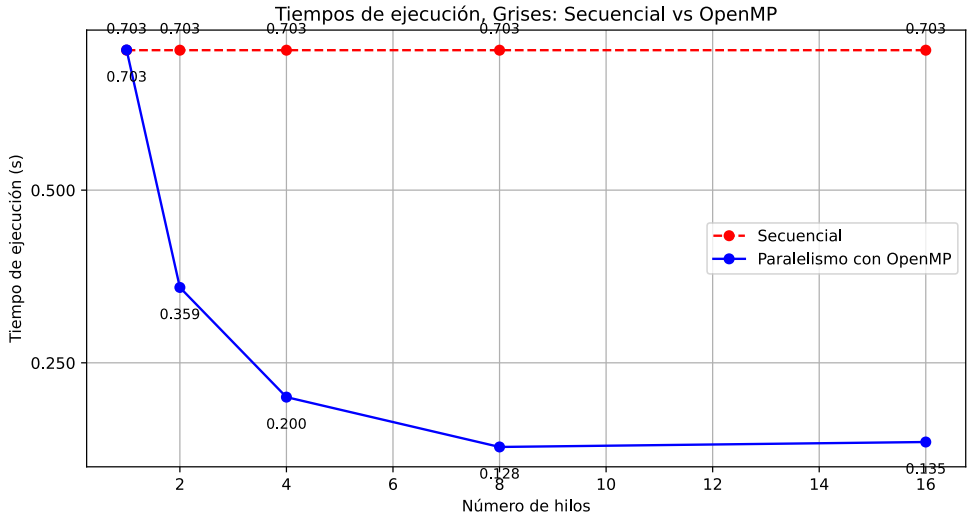

Fig. 8: *Comparativa de tiempos de ejecución de la escala de grises.*

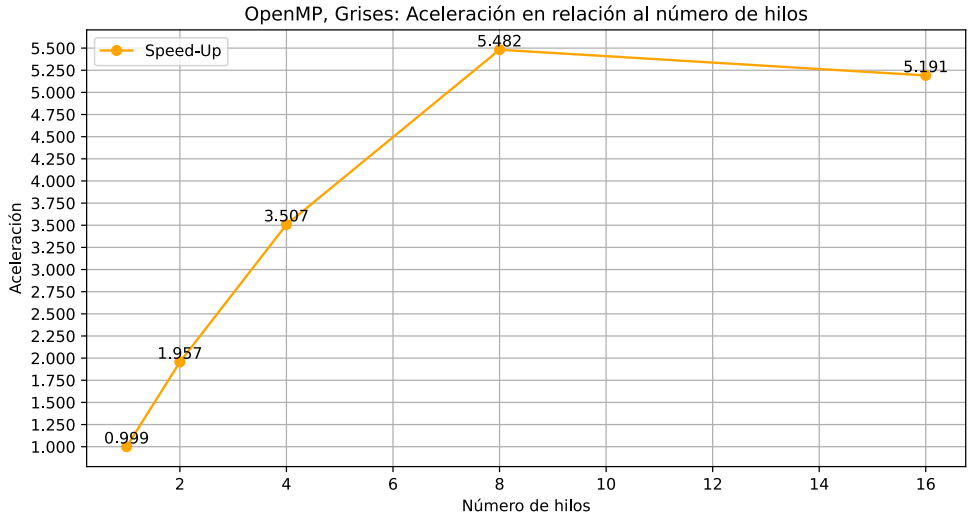

Fig. 9: *Aceleración empírica del programa OpenMP procesando la escala de grises respecto al programa secuencial.*

Como puede apreciarse, las métricas y visualizaciones anteriores muestran un patrón consistente para el procesamiento de color y escala de grises. La paralelización con OpenMP resulta en mejoras significativas en el rendimiento, especialmente en los primeros incrementos de hilos, pero el impacto disminuye gradualmente debido a factores inherentes al paralelismo como *overheads*, sincronización y porciones no paralelizables del código. La aceleración o *speed-up* conseguido demuestra un uso eficiente de recursos hasta límites prácticos impuestos por la arquitectura y la naturaleza del problema.

A continuación, se recoge una comparación de la aceleración empírica respecto a la **ley de Gustafson** para cada caso anterior, expuesta en la siguiente fórmula:

$$S(N) = N - \alpha \cdot (N - 1)$$

El factor *alpha* queda indicado como la fracción del código que es secuencial, y por ende, no paralelizable. En este caso, queda resumido en la siguiente fórmula su cálculo. Se obtiene despejando en la **ley de Gustafson** y utilizando los datos de la aceleración empírica:

$$\alpha = \frac{N - S(N)}{N - 1}$$

Posteriormente, realizando la media de las fracciones obtenidas por N procesos, se define un valor para cada caso:

Caso	α
YUV-OpenMP	0.3723
Grises-OpenMP	0.3220
HSL-OpenMP	0.3136

Fig. 10: *Fracciones obtenidas para cada caso.*

Finalmente, se incluyen las distintas visualizaciones mencionadas. En general, la Figura 11, la Figura 12 y la Figura 13 muestran respectivamente un patrón común: la aceleración teórica siempre supera a la empírica, lo que refleja las limitaciones del paralelismo en un entorno práctico. Los tres casos sugieren que el sistema experimenta una saturación o una reducción en el paralelismo, posiblemente debido a problemas de sincronización o una carga de trabajo mal distribuida.

Entrando más en detalle, la discrepancia más notoria se encuentra en el procesado YUV, que enfrenta mayores desafíos de escalabilidad y complejidad, mientras que HSL logra un mejor rendimiento en configuraciones con mayor número de procesos. Esto resalta la importancia de optimizar tanto el software como el hardware para minimizar estos factores y cerrar la brecha entre la teoría y la práctica.

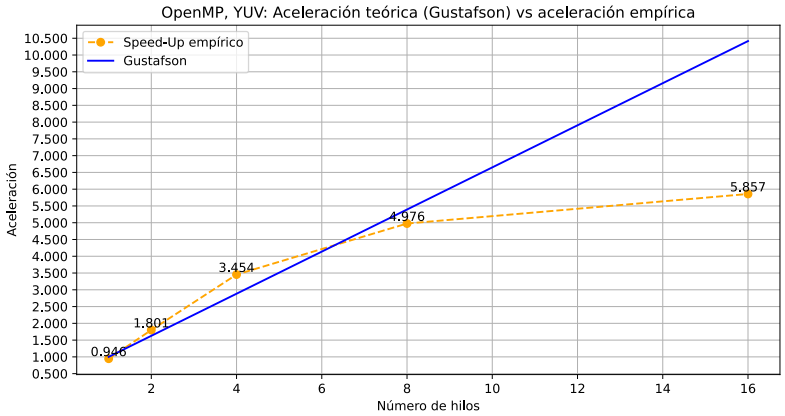

Fig. 11: *Aceleración empírica del programa OpenMP procesando YUV respecto al cálculo teórico de Gustafson.*

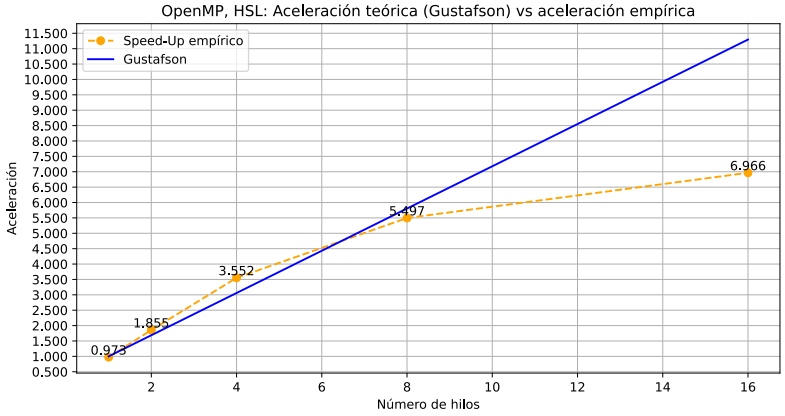

Fig. 12: *Aceleración empírica del programa OpenMP procesando HSL respecto al cálculo teórico de Gustafson.*

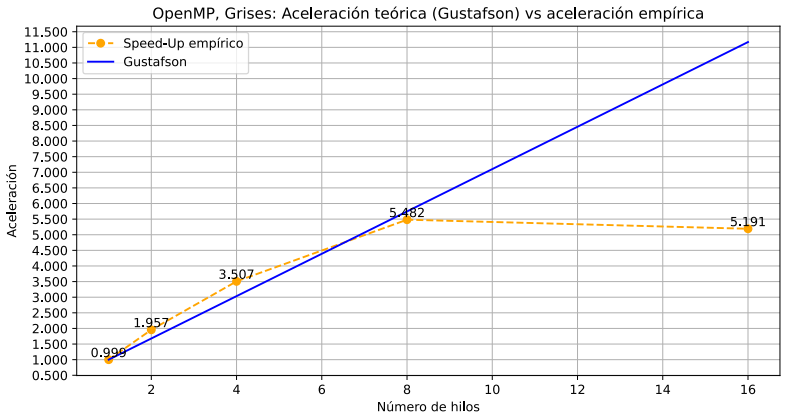

Fig. 13: *Aceleración empírica del programa OpenMP procesando escala de grises respecto al cálculo teórico de Gustafson.*

3. Paralelización con MPI

Message Passing Interface (MPI) es un estándar para la computación paralela, la cual también permite realizar computación distribuída. En MPI, se generan un número de procesos repartidos en uno o más nodos.

3.1. Metodología

Como ya hemos mencionado anteriormente, la metodología de paralelización se puede abstraer en cuatro fases principales.

Para la implementación con MPI, en la fase de **descomposición** se divide la imagen de entrada (de ancho w y altura h) entre el número total de procesos n[2]. Éste proceso se realiza por filas, dado que el lenguaje de programación, C++, es *row major*. Para cubrir el caso de que la imagen no se pueda dividir equitativamente en filas, el resto de filas $r = h \bmod w$ se repartirán entre los r primeros procesos.

Para definir la **asignación** los elementos de cada *chunk* usaremos el número de elementos del mismo, $C(p_i)$, y el desplazamiento, $D(p_i)$, entendido como el índice del primer elemento de la imagen que pertenece al mismo. Estos valores quedan definidos por las ecuaciones 1 y 2.

$$C(p_i) = \begin{cases} \lfloor h/w \rfloor + rw, & \text{si } i = 0 \\ \lfloor h/w \rfloor + rw, & \text{si } 0 < i \leq r \\ \lfloor h/w \rfloor, & \text{si } i > r \end{cases} \tag{1}$$

$$D(p_i) = \begin{cases} 0, & \text{if } i = 0 \\ D(p_{i-1}) + C(p_{i-1}) - w, & \text{si } i > 0 \end{cases} \tag{2}$$

Dado que para calcular la ecualización del histograma de un píxel (elemento) es necesario conocer los valores de los píxeles que le rodean, a cada trozo (o *chunk*) que se le asigne a cada proceso p_i se le añadirán los elementos de la fila inmediatamente superior e inferior. En el caso de que no exista esa fila (el caso del primer y el último *chunk*), esto no será necesario. Ésto genera un solapamiento que será obviado a la hora de generar la imagen final. El reparto inicial, por tanto, queda definido por la ecualización 3.

[2]El primer proceso será el proceso p_0 y el último será el proceso p_{n-1}. El proceso p_0 será considerado también como proceso *root*, o proceso orquestador.

$$C'(p_i) = \begin{cases} \lfloor h/w \rfloor + (1+r)\cdot w, & \text{si } i = 0 \\ \lfloor h/w \rfloor + (2+r)\cdot w, & \text{si } 0 < i \leq r \\ \lfloor h/w \rfloor + 2w, & \text{si } r < i < n-1 \\ \lfloor h/w \rfloor + w, & \text{si } i = n-1 \end{cases} \tag{3}$$

Para la **orquestación** y el **reparto** de los elementos se realizarán llamadas MPI. El proceso seguido para cada uno de los sub-procesos (escala de grises, color HSL y color YUV), es el siguiente:

1. El proceso p_0 lee el fichero de entrada, y envía, mediante una llamada `MPI_Bcast` el tamaño de la imagen al resto de procesos.
2. Todos los procesos calculan los distintos tamaños y desplazamientos iniciales ($C'(p)$ y $D(p)$).
3. Se realiza una llamada `MPI_Scatterv` por canal para que el proceso p_0 envíe los datos correspondientes al resto de procesos, generando una sub-imagen en cada proceso.[3]
4. Cada proceso calcula su histograma parcial. En éste histograma no se tienen en cuenta las filas extras de solapamiento ($C(p)$ y $D(p)$).
5. Se realiza una llamada `MPI_Allreduce` para generar el histograma de la imagen completa y repartirlo a todos los nodos.
6. Cada proceso realiza la ecualización de su sub-imagen. Es importante recalcar que para este cálculo se tiene en cuenta el histograma de la imagen completa.
7. Los procesos vuelven a computar los desplazamientos y tamaños, excluyendo en esta ocasión las filas extras de solapamiento ($C(p)$ y $D(p)$).
8. Se realiza una llamada `MPI_Gatherv` por canal para generar la imagen final en el proceso p_0.
9. El proceso p_0 escribe el fichero de salida.

3.2. Métricas

En esta sección, se incluyen las métricas obtenidas durante la ejecución del programa con paralelismo **MPI**. Quedan recogidas representaciones gráficas, agrupados dos a dos (HSL y YUV para color y grises), un tipo de representación expone una comparativa con la ejecución secuencial en términos de tiempo, otra en términos de aceleración y

[3] Para el caso de las imágenes a color, en este punto también se transforma la sub-imagen de RGB a HSL/YUV.

una tercera recoge una equiparación del *speed-up* empírico versus la **ley de Gustafson**. Se dispone de un análisis posterior de las mismas, con el objetivo de definir conclusiones que demuestren qué tan buen rendimiento puede llegar a obtener MPI con la optimización implementada.

Las fórmulas utilizadas para calcular el speedup y la ley Gustafson son las mismas definidas en la sección de 2.2Métricas de OpenMP.

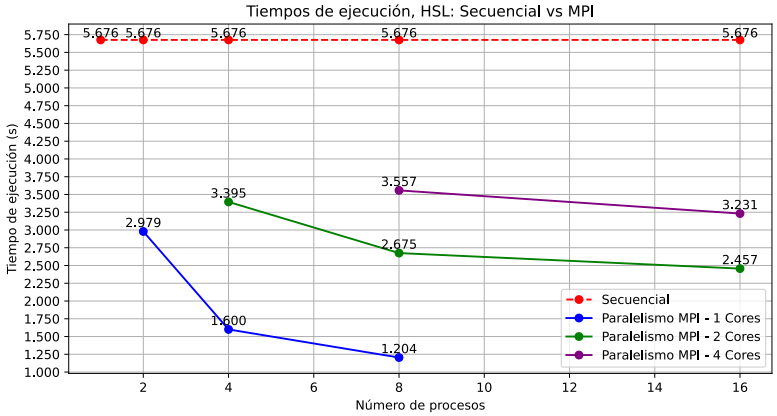

Fig. 14: *Comparativa de tiempos de ejecución HSL.*

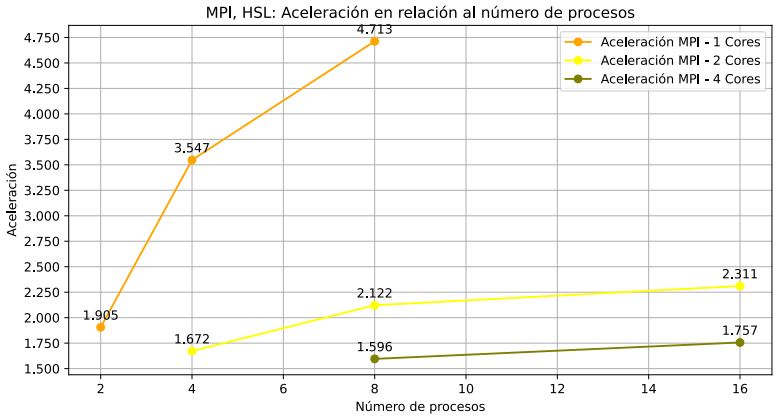

Fig. 15: *Aceleración empírica del programa MPI procesando HSL respecto al programa secuencial.*[1]

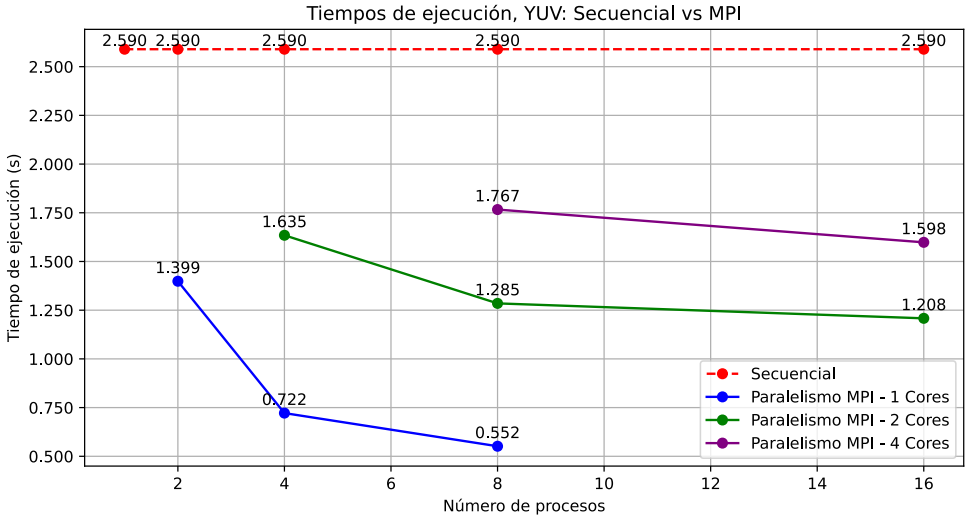

Fig. 16: *Comparativa de tiempos de ejecución YUV.*

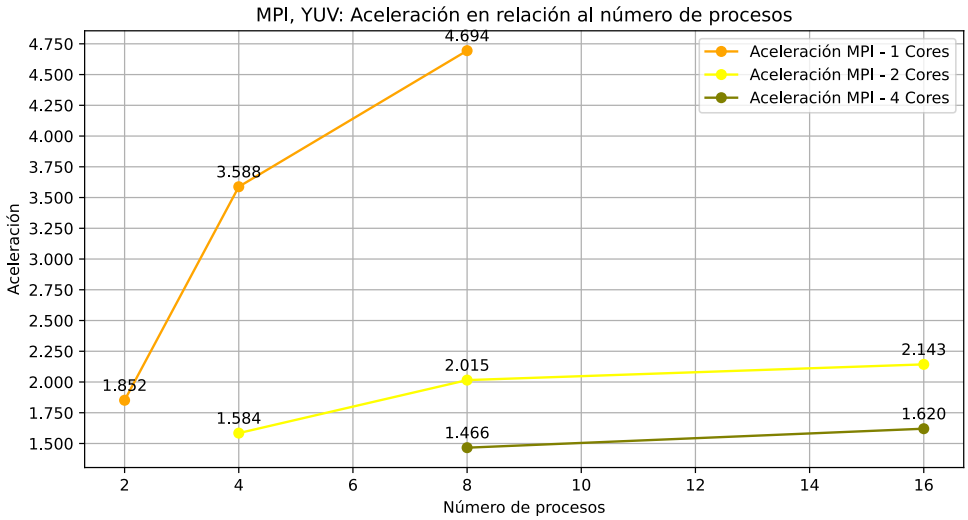

Fig. 17: *Aceleración empírica del programa MPI procesando YUV respecto al programa secuencial.*

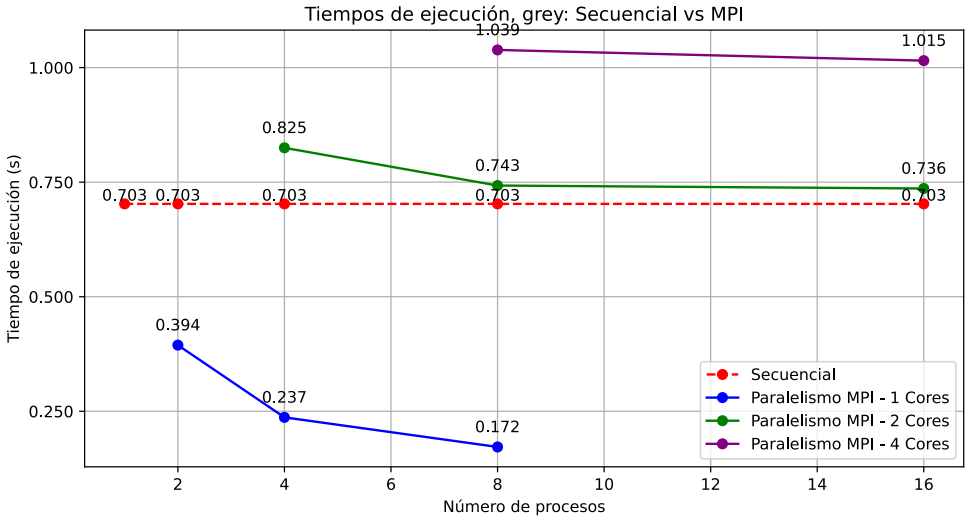

Fig. 18: *Comparativa de tiempos de ejecución de la escala de grises.*

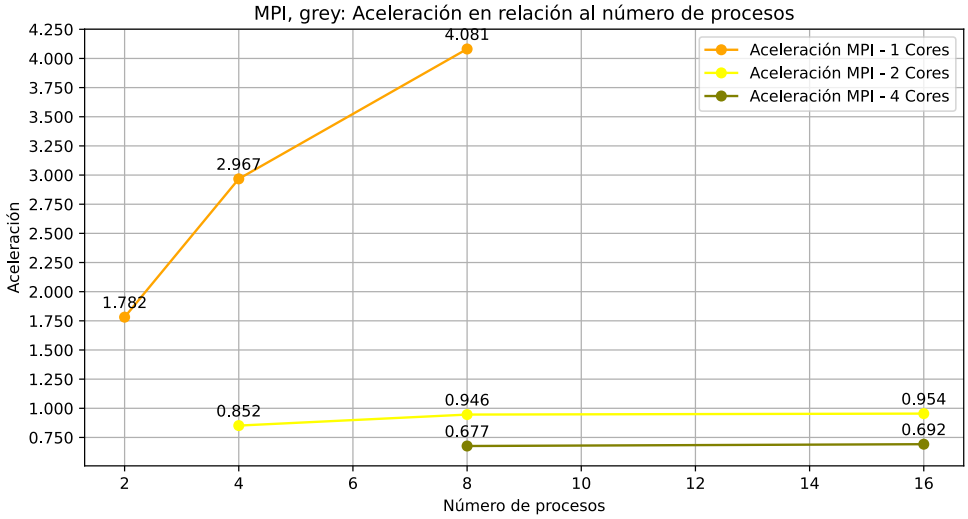

Fig. 19: *Aceleración empírica del programa MPI procesando la escala de grises respecto al programa secuencial.*

Como puede apreciarse, las métricas y visualizaciones anteriores muestran un patrón consistente para el procesamiento de color y escala de grises. La paralelización con MPI resulta en mejoras significativas en el rendimiento, especialmente en los primeros incrementos de hilos, pero el impacto disminuye gradualmente debido a factores inherentes al paralelismo como *overheads*, sincronización y porciones no paralelizables del código. Este mismo overhead causa que el resultado obtenido usando más de un nodo en la escala de grises resulte en una **pérdida** de eficiencia (speedup inferior a 1) para más de un (1) nodo. La aceleración o *speed-up* conseguido, ya sea mediante el uso de más procesos en escala de grises o mediante el uso de más nodos y procesos, demuestra la posibilidad de mejorar eficiencia a la hora de resolver el problema, hasta límites prácticos impuestos por la arquitectura y la naturaleza del problema.

A continuación, se recoge una comparación de la aceleración empírica respecto a la **ley de Gustafson** para cada caso anterior, calculando la alfa para cada conjunto y a continuación mostrando el speedup acorde a ambas medidas. En general, la Figura 20, la Figura 21 y la Figura 22 muestran un patrón común: la aceleración teórica supera a la empírica, lo que refleja las limitaciones del paralelismo en un entorno práctico. Los tres casos sugieren que el sistema experimenta una saturación o una reducción en el paralelismo alrededor de los 4-8 procesos, potencialmente debido a problemas de sincronización o una carga de trabajo mal distribuida. Los problemas de sincronización son uno de los principales problemas a la hora de paralelizar, puesto que el tiempo usado en comunicarse y sincronizar limita el tiempo máximo paralelizable.

Cómo se había comentado previamente, el principal caso llamativo es la escala de grises, ya que puesto a que pierde eficiencia respecto al modelo secuencial, la ley de gustafson prevee una pérdida de eficiencia mayor que el estancamiento real. Esto resalta la importancia de evaluar la eficiencia tras paralelizar, para compribar que la implementación realizada sea de verdad una mejora. Resalta también como no todas las tareas mejoran mediante paralelización.

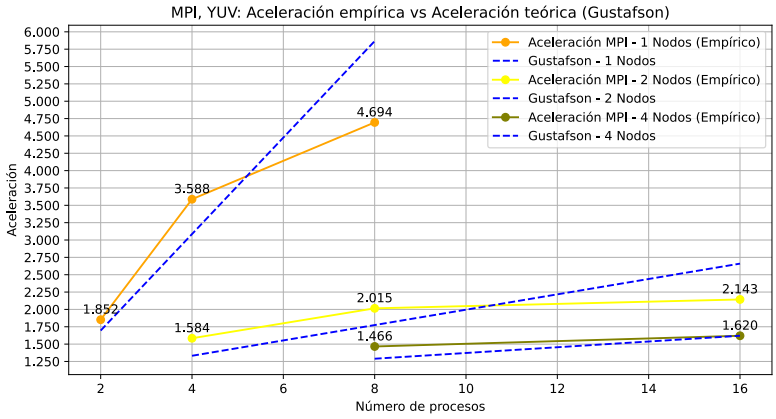

Fig. 20: *Aceleración empírica del programa MPI procesando YUV respecto al cálculo teórico de Gustafson.*

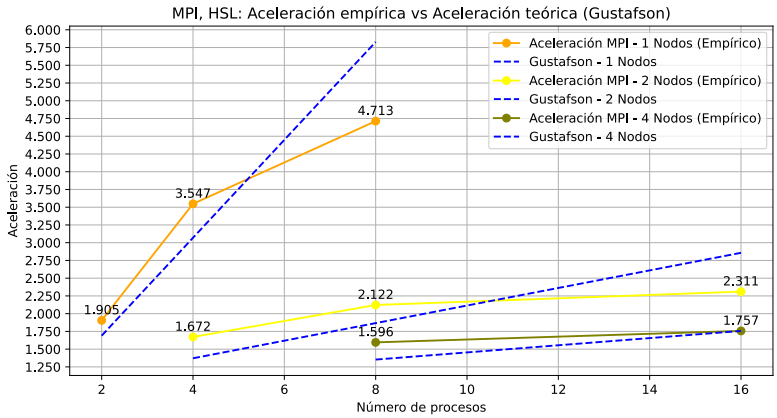

Fig. 21: *Aceleración empírica del programa MPI procesando HSL respecto al cálculo teórico de Gustafson.*

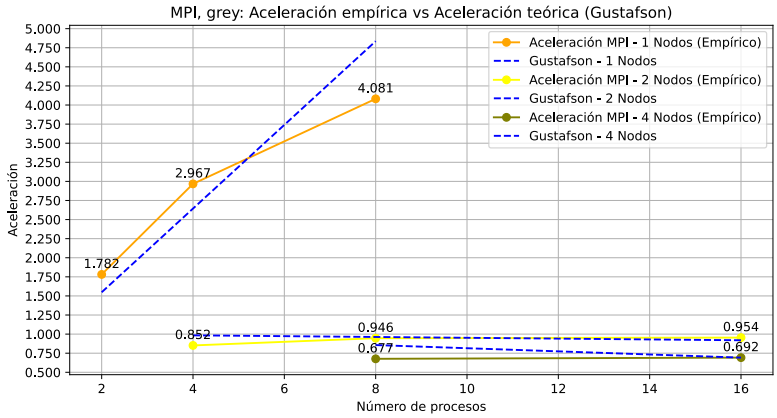

Fig. 22: *Aceleración empírica del programa MPI procesando escala de grises respecto al cálculo teórico de Gustafson.*

4. Paralelización híbrida

La paralelización híbrida se ha realizado combinando procesos MPI, los cuales permiten estar distribuídos en diferentes nodos, con los *threads* de OpenMP.

4.1. Metodología

La metodología de este enfoque híbrido se basa en combinar las metodologías de los anteriores enfoques.

Dado que la paralelización con MPI se centra en generar sub-imágenes para que cada proceso compute una parte, y la paralelización con OpenMP se centra en paralelizar ese proceso de cómputo, ambos enfoques pueden convivir y complementarse sin problema.

4.2. Métricas

En esta sección, se incluyen las métricas obtenidas durante la ejecución del programa con paralelismo **OpenMP+MPI**. Quedan recogidas representaciones gráficas, tanto para HSL y YUV (color) como greyscale (tonos de gris, o blanco y negro). Un tipo de representación expone una comparativa de la ejecución paralela con la ejecución secuencial en términos de tiempo y otra en términos de aceleración. Cada conjunto de datos formando una línea representa un grupo de pruebas con el mismo número de nodos y procesos asignados, diferenciados por el número de hilos asignados. Se dispone de un análisis posterior de las mismas, con el objetivo de extraer conclusiones que demuestren qué tan buen rendimiento puede llegar a obtener OpenMP+MPI con la optimización implementada.

La fórmula utilizada para calcular el speedup es la misma definida en la sección de 2.2Métricas de OpenMP

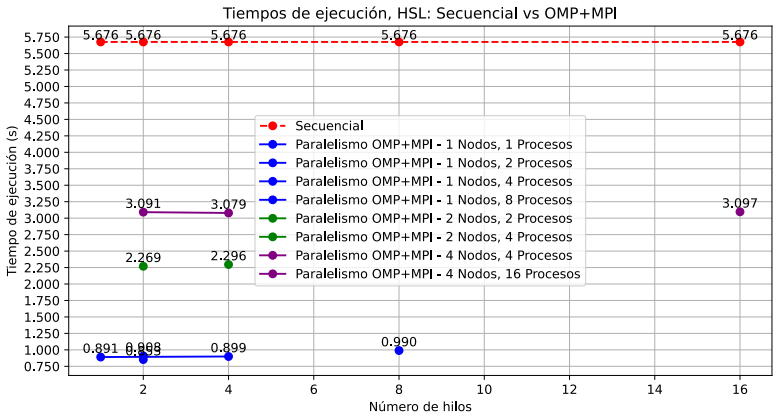

Fig. 23: *Comparativa de tiempos de ejecución HSL.*

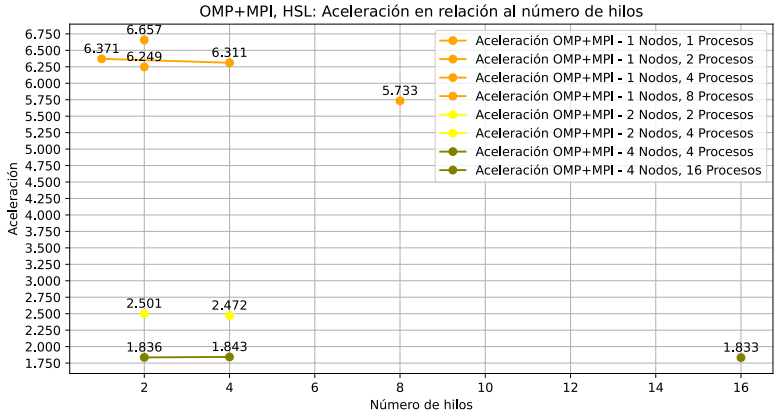

Fig. 24: *Aceleración empírica del programa OpenMP+MPI procesando HSL respecto al programa secuencial.*[1]

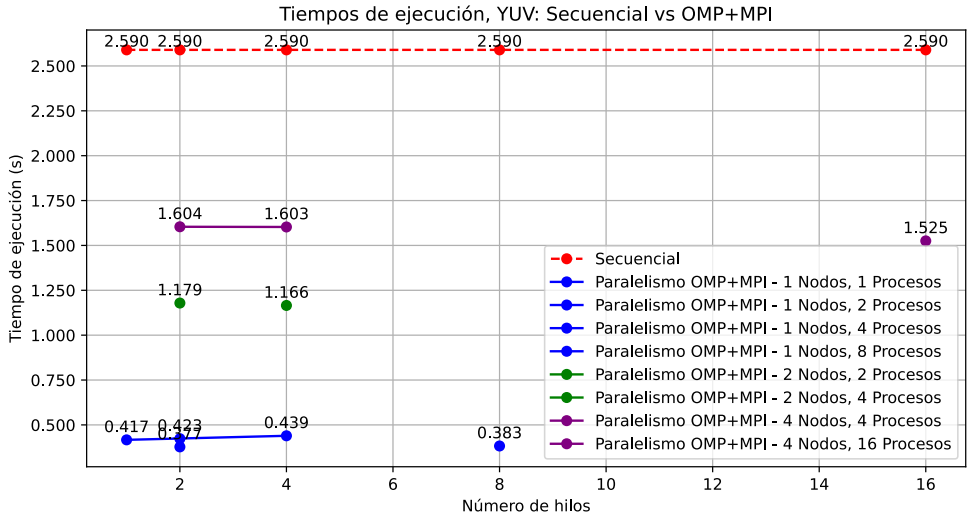

Fig. 25: *Comparativa de tiempos de ejecución YUV.*

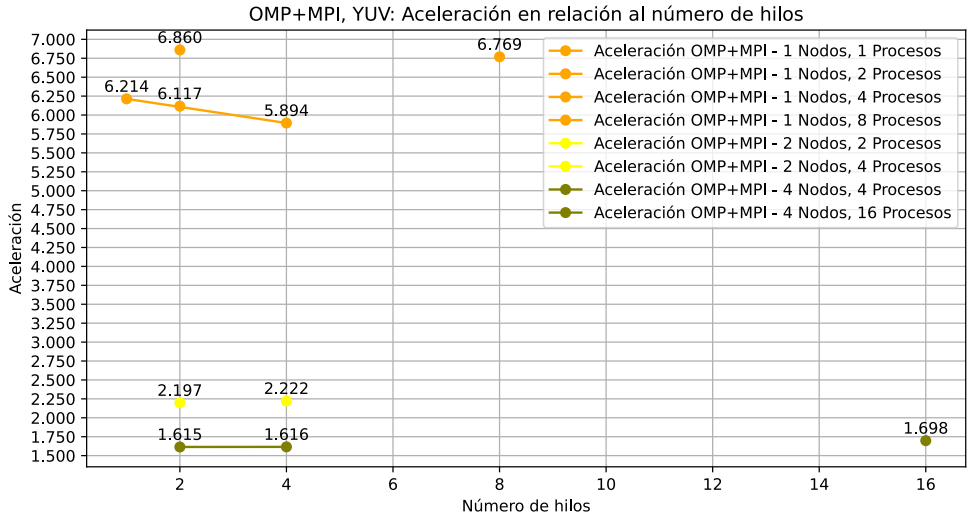

Fig. 26: *Aceleración empírica del programa OpenMP+MPI procesando YUV respecto al programa secuencial.*

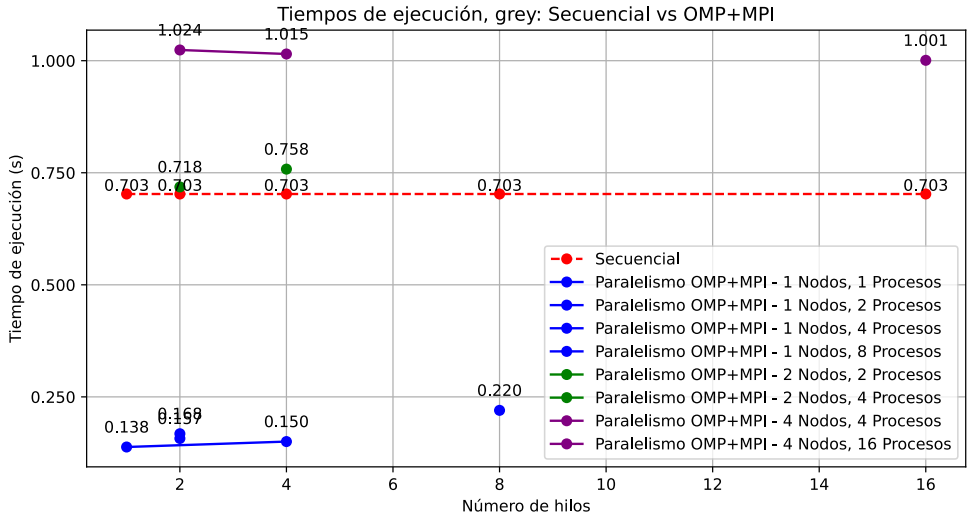

Fig. 27: *Comparativa de tiempos de ejecución de la escala de grises.*

Como puede apreciarse, las métricas y visualizaciones anteriores muestran un patrón consistente para el procesamiento de color y grises. De forma breve, los resultados con un sólo nodo dan mejores resultados que con más nodos, probablemente por porblemas de sincronización. Además, un incremento de procesos o hilos no necesariamente mejora la eficiencia, como se puede observar a continuación en las gráficas más coloridas. Con esto en cuenta, la mejora en rendimiento es muy considerable, superior a OpenMP o MPI por individual.

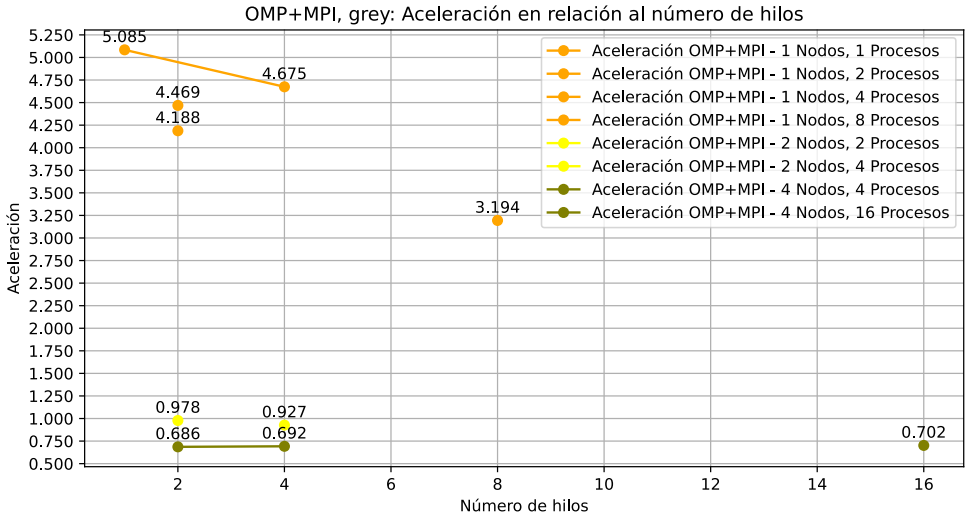

Fig. 28: *Aceleración empírica del programa OpenMP+MPI procesando escala de grises respecto al programa secuencial.*

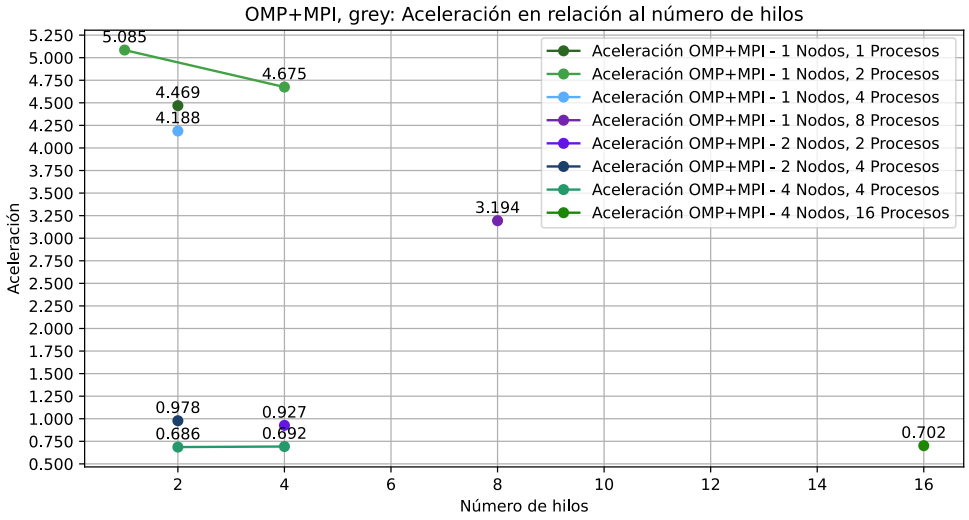

Fig. 29: *Aceleración empírica del programa OpenMP+MPI procesando escala de grises a color respecto al programa secuencial.*

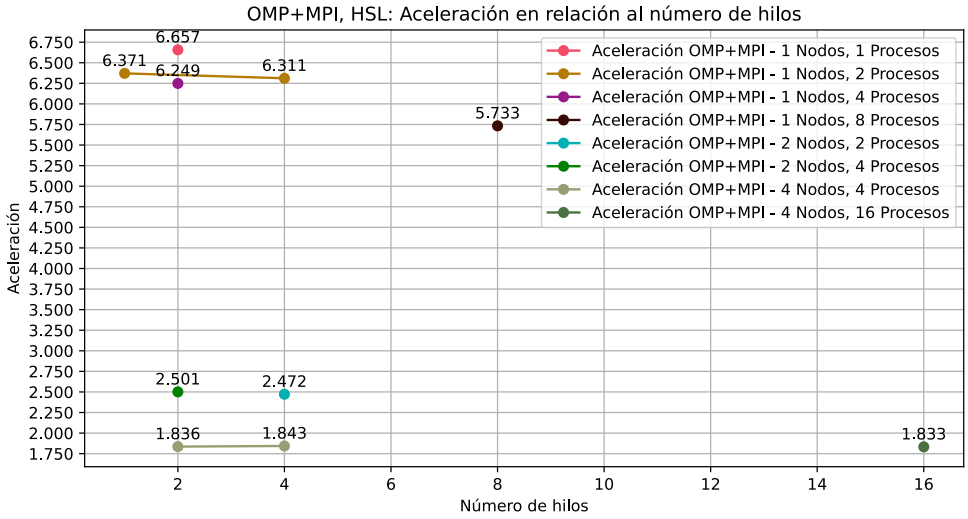

Fig. 30: *Aceleración empírica del programa OpenMP+MPI procesando HSL a color respecto al programa secuencial.*

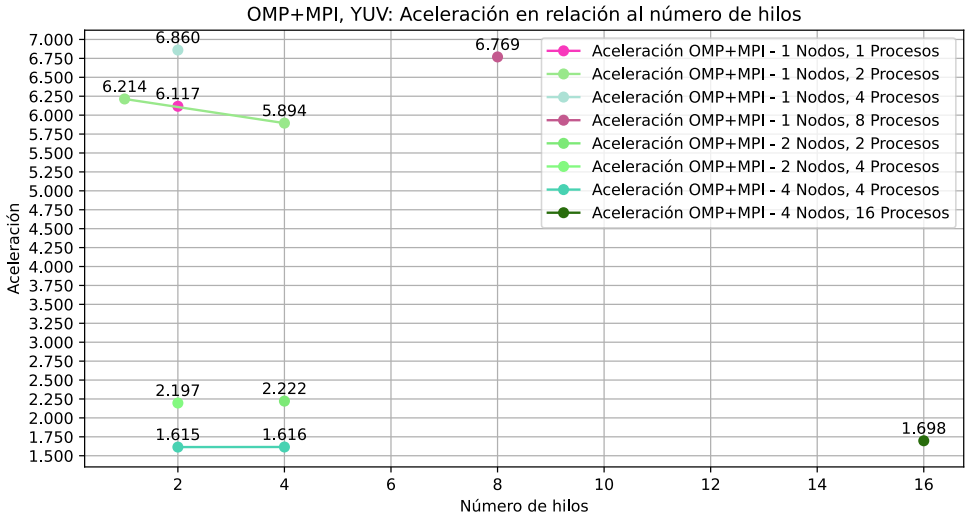

Fig. 31: *Aceleración empírica del programa OpenMP+MPI procesando YUV a color respecto al programa secuencial.*

Parte III

Conclusiones

A través de las métricas podemos sacar la conclusión de que el mejor *approach* para este caso y las máquinas donde se ejecutan, es el de OpenMP. Esto es probablemente debido a que la comunicación entre distintos nodos es muy lenta debido a la red del servidor de pruebas, y que en la paralelización de OpenMP se aprovecha mucho la vectorización.

Parte IV

Apéndices

A. Compilación y ejecución

Compilación

Para compilar este programa es necesario tener instalado en el sistema las siguientes dependencias:

- OpenMP[1]
- MPICH[2]
- CMake[3]
- GNU Make[4], o similar

Es necesario ejecutar los siguientes comandos desde la raíz del repositorio para realizar la compilación:

```
mkdir build
cd build/
cmake .. -DCMAKE_CXX_COMPILER=mpicxx.mpich
make
```

Ésto generará cuatro ejecutables, correspondientes a las cuatro versiones del *software*:

- `contrast` – versión original, secuencial
- `contrast-omp` – versión paralelizada con OpenMP
- `contrast-mpi` – versión paralelizada con MPI
- `contrast-mpi-omp` – versión paralelizada de forma híbrida, con OpenMP y MPI

[1] https://www.openmp.org/
[2] https://www.mpich.org/
[3] https://cmake.org/
[4] https://www.gnu.org/software/make/

Ejecución

Lo primero, es necesario convertir la imagen de entrada a los formatos PGM (`in.pgm`) y PPM (`in.ppm`). Ésto se puede realizar, por ejemplo, mediante el uso de la herramienta de línea de comandos Convert[5], y dejarla en el mismo directorio que el ejecutable, en nuestro caso, el directorio `build/`.

Por ejemplo:

```
convert highres.jpg build/in.pgm
convert highres.jpg build/in.ppm
```

Las distintas versiones pueden ser ejecutadas en un sistema de colas como Slurm[6], de la siguiente forma:

```
srun -N <nodos> -n <procesos> <ejecutable>
```

Especificando el número de nodos a usar y el número de procesos. Para la versión secuencial y la de OpenMP, estos números serán ambos 1, ya que ambas ejecutan un único proceso.

Para la versión OpenMP y la híbrida, además se podrá especificar el número de *threads* de ejecución mediante la variable de entorno `OMP_NUM_THREADS=<threads>`.

Al ejecutar el programa, se generarán tres ficheros de salida, `out.pgm`, `out_hsl.ppm` y `out_yuv.ppm`, correspondientes a los tres sub-procesos de ecualización.

[5] https://imagemagick.org/script/convert.php
[6] https://slurm.schedmd.com/documentation.html