uc3m | Universidad Carlos III de Madrid

Máster en Ingeniería Informática

2020 - 2021

Computación de Altas Prestaciones

“Paralelización de código con OpenMP + MPI”

100363974 - Carlos Vigil González
100363815 - David Gil López
100316890 - Daniel Alejandro Rodríguez López

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

La presente memoria explica los pasos seguidos, experimentacón y resultados obtenidos durante la paralelización del código de equalización de histograma entregado inicialmente. Pero antes de entrar en materia, conviene explicar en qué consiste una equalización de histograma.

De forma simplista, una equalización de histograma consiste en estirar el rango de valores en una imagen para aplanar su histograma de valores.

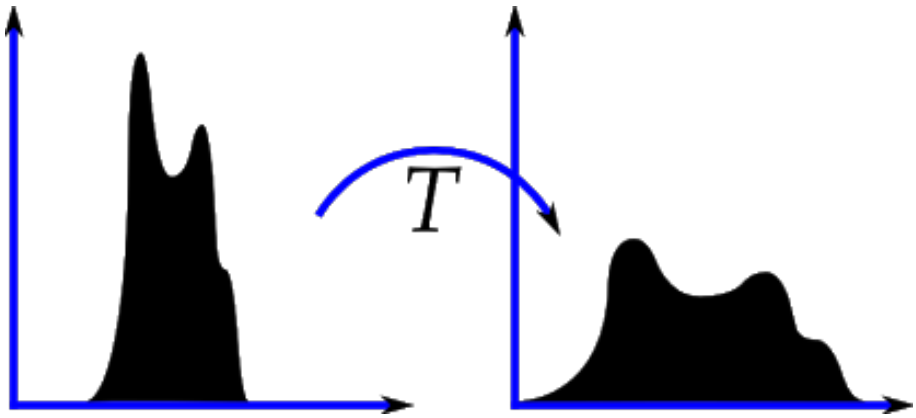

Fig. 1.1. Ejemplo de Equalización de Histograma

De forma un poco más técnica, una equalización de histograma distribuye los valores de color en una imagen de forma que todos se encuentran presentes de forma equitativa dentro de la imagen, como se puede apreciar en la figura 1.1 Ejemplo de Equalización de Histograma

Para calcular estos histogramas se debe recorrer la imagen múltiples veces, tanto para calcular los valores iniciales como para establecer los valores finales, por lo tanto hemos orientado nuestros esfuerzos con OpenMP [1] en optimizar estas iteraciones sobre la imagen, mientras que con MPI [2] nos hemos centrado en el reparto de fragmentos de la imagen entre distintos procesos para reducir el número de iteraciones.

1.0.1. Speed up teóricos

Antes de empezar a realizar ningún cambio sobre el código, con el objetivo de conocer qué mejoras podemos esperar a la hora de paralelizarlo, se ha calculado un *speed up* teórico para la Ley de Amdahl y para la Ley de Gustafson.

Ley de Amdahl

La Ley de Amdahl nos permite dado un programa con carga fija, obtener el *speed up* máximo que podríamos alcanzar

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

Siendo S el *speed up*, P la fracción paralela de código y N el número de núcleos entre los que la dividimos.

De esta forma, en el código original tenemos 288 líneas de código, de las cuales hemos identificado 94 como parelizables, dándo lugar a una $P = 0{,}32638$.

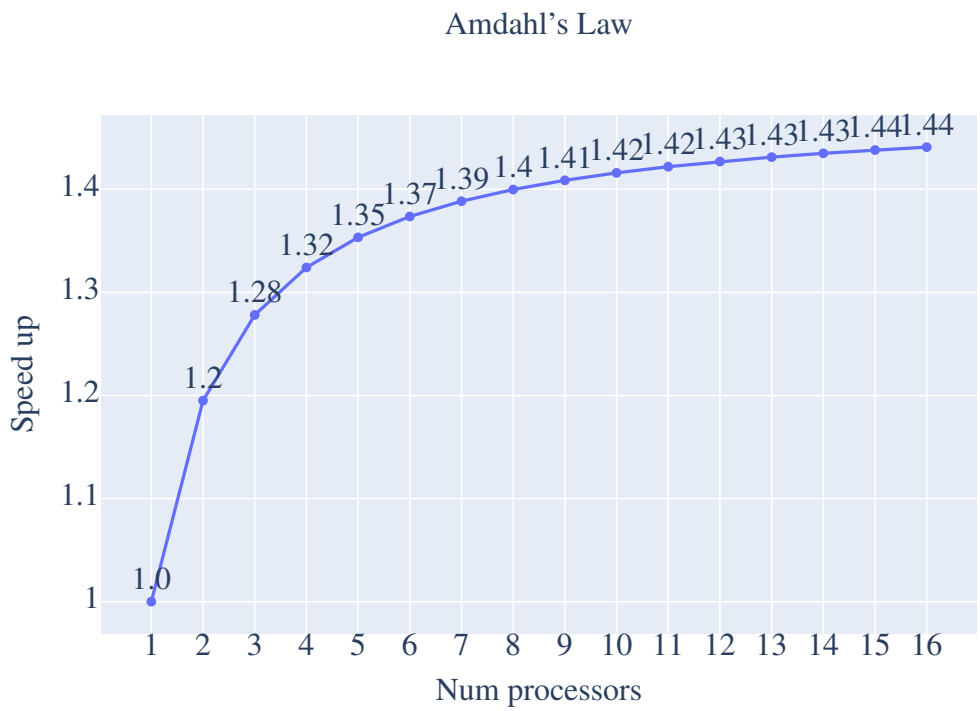

Fig. 1.2. Amdahl teórico

Como se puede observar, con Amdahl el *speed up* se empieza a detener a partir de los 8 procesadores, ya que de un *speed up* de 1.4 pasamos a 1.44 con 16 procesadores, por lo que no tiene mucho sentido aumentar el número de procesadores ya que la ganancia sería mínima.

Ley de Gustafson

A diferencia de en la Ley de Amdahl, en Gustafson se considera que al mismo tiempo que se aumenta el número de procesadores disponibles, también aumenta el tamaño del

problema.

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

Siendo P el número de procesadores y α la parte de código no paralelizable.

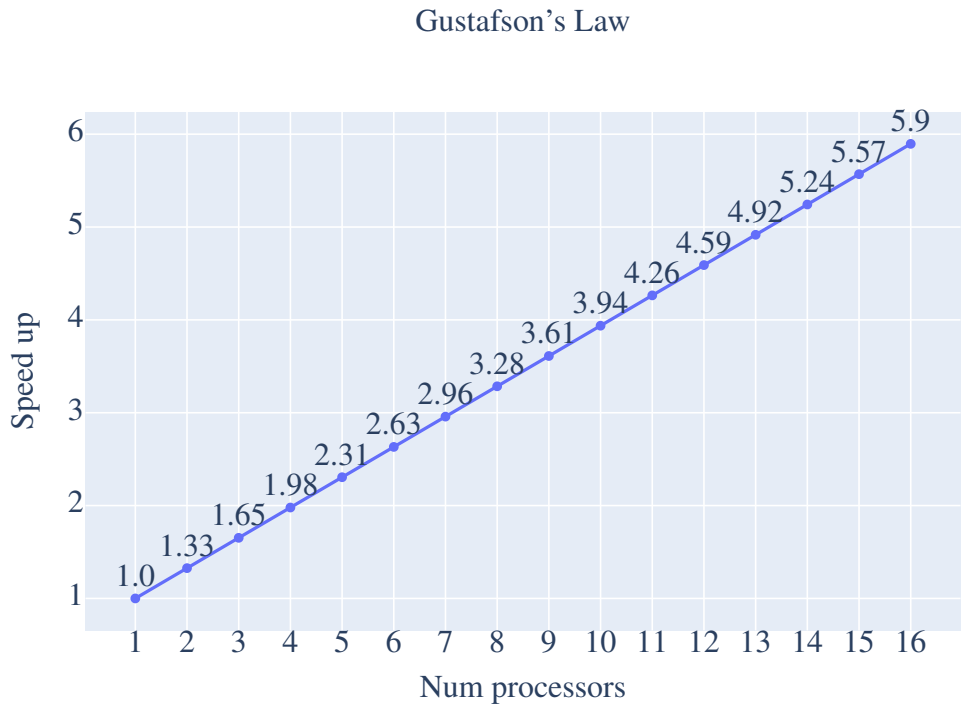

Fig. 1.3. Gustafson teórico

En cambio, si utilizásemos Gustafson, al ser dependiente del tamaño de los datos de entrada, el *speed up* no se detiene. Por lo tanto, dado que este problema depende completamente del tamaño de la imagen, es más conveniente utilizar este *speed up* para realizar estimaciones y comparar mejoras. Suponiendo que para las pruebas del código paralelo se pudiese ejecutar en todas estas posibles configuraciones, se debería de ver un aumento constante del speed up (si el tamaño de la imagen aumentase también).

2. ANÁLISIS Y EXPERIMENTACIÓN

En esta sección se describe el análisis de código realizado para determinar qué partes se van a paralelizar mediante OpenMP y cuales con MPI. También se comentarán las razones por las cuales se ha optado por elegir cada método en cada caso.

2.1. Análisis

Esta sección se descompondrá en los cuatro pasos que se pueden seguir a la hora de paralelizar: Descomposición, Asignación, Orquestación y Reparto.

2.1.1. Descomposición

En esta fase se analiza el código para identificar qué partes son paralelizables y cómo se podrían paralelizar.

Como se ha explicado previamente en la Introducción, el esfuerzo de paralelización con OpenMP se va a orientar a bucles e iteraciones mientras que con MPI se va a reducir la carga de trabajo que deberá ejecutar cada hilo dividiendo la imagen de entrada en fragmentos más pequeños.

Para identificar qué secciones de código son las más costosas se ha ejecutado callgrind y visualizado sus resultados con Kcachegrind [3].

Incl.	Auto	Llamado	Función	Posición
28.90	28.90	1	hsl2rgb(HSL_IMG)	contrast: contrast-enhancement.c...
24.28	24.28	1	rgb2hsl(PPM_IMG)	contrast: contrast-enhancement.c...
12.06	12.06	1	yuv2rgb(YUV_IMG)	contrast: contrast-enhancement.c...
9.03	9.03	3	histogram_equalization(...	contrast: histogram-equalization.c...
8.34	8.34	1	rgb2yuv(PPM_IMG)	contrast: contrast-enhancement.c...
7.36	7.36	2	write_ppm(PPM_IMG, c...	contrast: contrast.cpp
5.02	5.02	1	read_ppm(char const*)	contrast: contrast.cpp
5.02	5.02	3	histogram(int*, unsigned...	contrast: histogram-equalization.c...
0.01	0.00	2 087	_dl_lookup_symbol_x	ld-2.28.so: dl-lookup.c
0.00	0.00	2 087	do_lookup_x	ld-2.28.so: dl-lookup.c, ldsodefs.h
0.01	0.00	10	_dl_relocate_object	ld-2.28.so: dl-reloc.c, dl-machine....
0.00	0.00	3 757	strcmp	ld-2.28.so: strcmp.S
0.00	0.00	2 071	check_match	ld-2.28.so: dl-lookup.c
0.00	0.00	1	_dl_addr	libc-2.28.so: dl-addr.c
0.00	0.00	1	__GI___tunables_init	ld-2.28.so: dl-tunables.c, dl-tunab...
0.00	0.00	714	_dl_name_match_p	ld-2.28.so: dl-misc.c

Fig. 2.1. Funciones más costosas

En la figura 2.1 Funciones más costosas se muestran ordenados de mayor a menor coste las llamadas que se realizan al ejecutar la versión secuencial del código y, como se puede apreciar, alrededor del 50 % del tiempo se gasta en funciones relacionadas con la equa-

lización de una imagen en color (`hsl2rgb`, `rgb2hsl`, `yuv2rgb`, `rgb2yuv`). También alrededor del 10 % del tiempo se gasta en la equalización como tal (`histogram_equalization`).

2.1.2. Asignación

A través de las funciones identificadas en el paso anterior mediante cachegrind y Kcachegrind se puede analizar porqué son las más costosas de toda la aplicación.

Esto se debe a que son funciones que iteran sobre todos los píxeles de la imagen, por lo tanto, siguiendo el planteamiento inicial de fragmentar la imagen en trozos más pequeños y paralelizar estos bucles mediante OpenMP el tiempo final debería reducirse considerablemente.

A excepción de `histogram_equalization`, que está formado de dos bucles, y uno tiene dependencias con sus iteraciones previas, el resto de bucles carecen de este tipo de dependencias.

El código no parece excesivamente dependiente del tamaño del grano, dado que casi todo es independiente de lo que pueda ocurrir en los otros fragmentos en los se decide dividir la imagen original, por lo tanto un grano más fino que grueso puede ayudar alcanzar un mayor *speed up*

2.1.3. Orquestación

En esta sección se plantea la forma de comunicar los datos entre los distintos procesos e hilos que se van a generar.

Sin embargo, en el entorno que se dispone para realizar la práctica no es un aspecto tan configurable como podría ser, dado que todas las ejecuciones se realizan sobre una misma máquina, por lo tanto no existe tiempo de red, y otras variables como localidad de datos no hay que tenerse en cuenta, ya que todos los datos son locales.

En cuanto a la comunicación y sincronización entre los distintos procesos e hilos que se puedan crear, solamente existen cuatro momentos en los que tener cuidado, dado que dependen de toda la imagen, y no de fragmentos de la misma:

- Lectura de la imagen: Dado que solo 1 proceso debe leer, y comunicárselo al resto.
- Escritura de la imagen: Para evitar conflictos en disco, 1 único proceso debería recuperar toda la imagen y escribirla en disco.
- Cálculo del histograma: El histograma debe ser de toda la imagen, si no la equalización estará mal.
- Equalizar la imagen: Un paso previo antes de equalizar la imagen consiste en calcular una tabla de valores que depende del valor mínimo presente en la imagen.

2.1.4. Reparto

Dada la topología que se tiene (1 única máquina, con 8 procesadores), se va a intentar no sobrecargar la máquina, es decir. no lanzar 4 procesos de MPI con 8 hilos de OpenMP, dado que serían más tareas que procesadores disponibles. Sin embargo, hasta que no se realice una experimentacion probando todas las configuraciones disponibles no se puede afirmar que configuración sería la adecuada.

2.2. OpenMP + MPI

La paralelización de código con OpenMP se ha centrado en bucles *for* que operan sobre la imagen, dado que con OpenMP es muy sencillo seguir un modelo SIMD (*Single Instruction / Multiple Data*) y, además representan un mayor número de iteraciones que cualquier otro tipo de bucles dado que una imagen pequeña de 200x200 píxeles, se traduciría en 40000 iteraciones.

Empezando por el fichero principal, *contrast.cpp* se han paralelizados los bucles *for* que se ejecutan a la hora de realizar las lecturas y escrituras de las imágenes de tipo *PPM*, ya que se itera sobre los tres canales RGB por separado.

En el fichero *contrast-enhancement.cpp* se han paralelizado los bucles que se encuentran dentro de las funciones que cambian los formatos de color de las imágenes: de *rgb* a *hsl*, de *hsl* a *rgb*, de *rgb* a *yuv* y de *yuv* a *rgb*.

En el tercer fichero, *histogram-equalization.cpp*, también se ha realizado la paralelización de ciertos bucles. Los dos primeros, que se encuentran en la función *histogram*, se ha decidido optar por paralelizarlos como *SIMD* en vez de como simples bucles for, ya que al tener pocas iteraciones, el particionado no ofrecería ninguna ventaja, por otro lado el utilizar las operaciones vectoriales nos aportará la velocidad suficiente para contrarestar el *overhead* de paralelizarlo.

Por otro lado, en este mismo fichero también se ha paralelizado el bucle *for* que escribe la imagen final tras ecualizar el histograma, en este caso usando las directivas propias de los bucles *for* y de las instrucciones de tipo *SIMD*. El bucle que ecualiza el histograma no ha sido paralelizado debido a las dependencias que tienen entre una iteración y la anterior.

En cuanto MPI, la paralelización de código se ha centrado en repartir la carga de trabajo inicial entre los distintos procesos existentes. De forma similar a OpenMP, la parelización de MPI, aunque puede seguir otros patrones como MISD (*Multiple Instruction / Single Data*) o MIMD (*Multiple Instruction / Multiple Data*), se ha seguido un planteamiento SIMD.

Para decidir en qué secciones del código se puede paralelizar siguiendo un modelo SIMD, hay que identificar las operaciones que dependen de la imagen completa para evitar realizar esas operaciones con fragmentos de la imagen total, ya que se obtendrían

resultados distintos de la versión secuencial y por tanto, incorrectos.

Estas secciones de código que dependen de la imagen original, como se ha comentado previamente, se encuentran en la lectura inicial y escritura final de la misma e inicialización de variables en *contrast.cpp*, en la definición del histograma inicial y en la normalización del histograma final en *histogram-equalization.cpp*. Sin embargo, a pesar de depender de la imagen inicial, la inicialización del histograma es fácilmente paralelizable sin un impacto negativo en el rendimiento al tratarse de un array de números cuyo resultado final es una suma (reducción). Para el resto de operaciones que dependen de la imagen total, se ha delegado a que solamente el proceso con el rank 0 sea el que se encarge de realizar esas operaciones para posteriormente comunicárselas al resto de procesos.

2.2.1. Experimentación

La experimentación se ha realizado usando la imagen proporcionada de ejemplo (11472 x 6429 píxeles), usando una CPU Intel(R) Core(TM) i7-4702MQ CPU @ 2.20GHz con 4 núcleos físicos y otros 4 virtuales, variando el números de procesos de MPI y el número de hilos de OpenMP con valores 1, 2, 4 y 8 y repitiendo cada ejecución 10 veces.

En los resultados se pueden observar el tiempo medio obtenido con el código original y los tiempos medios obtenidos para cada una de las posibles combinaciones de MPI+OpenMP.

Partiendo de un valor promedio de ejecuciones secuenciales (sin contar entrada y salida) que se sitúa en valores cercanos a 0.11 s en grises y 4s en color, dividiendo a su vez esto en 0.74s cuando se procesa en formato YUV y 3.2s cuando se procesa en HSL. Y con valores de 1.1s en escala de grises y 10.5s en color al contar la entrada y salida.

Los valores promedio de la paralelización del código son los siguientes:

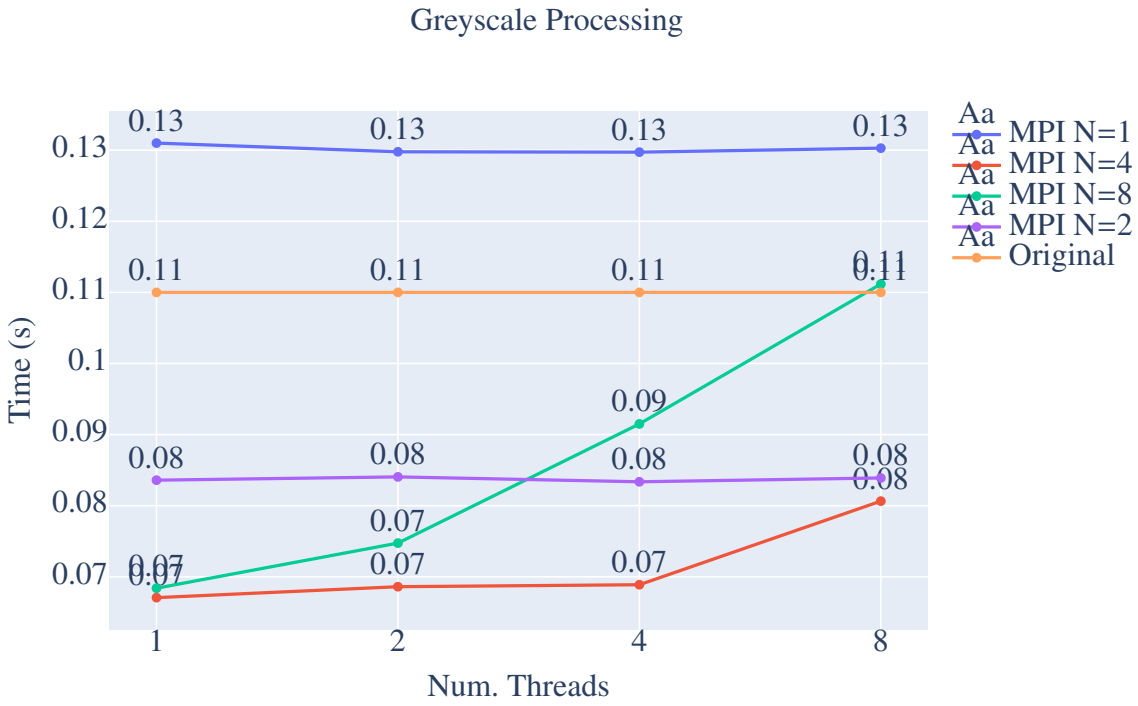

Fig. 2.2. Grises. Tiempos (sin I/O) Secuencial vs MPI+OpenMP

Como se puede apreciar en la figura, cuando se utiliza un número bajo de procesos el tiempo es peor, independientemente del número de hilos, ya que existe la sobrecarga de MPI, pero ningún tipo de paralelismo al haber un único proceso. Sin embargo, cuando empieza a existir paralelísmo y reparto de datos, aunque sea mínimo, como en el caso de N=2, se empieza a notar una mejora en los tiempos, siendo este máximo cuando se ocupan todos los procesadores de la máquina.

No obstante, cuando se empieza a sobrecargar la máquina, y al ser relativamente "baratoçomputar la equalización de histograma sobre la imagen en escala de grises, el tiempo comienza a empeorar, ya que el número de procesos e hilos es mayor que el número de núcleos disponibles.

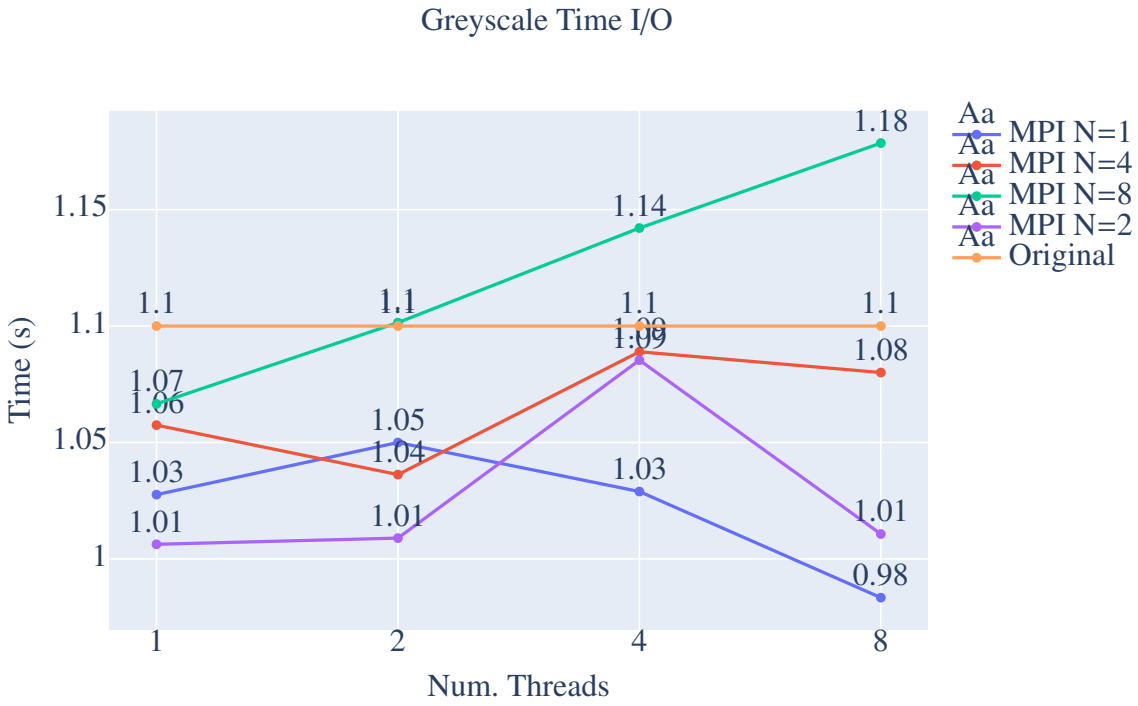

Fig. 2.3. Grises. Tiempos (con I/O) Secuencial vs MPI+OpenMP

Teniendo en cuenta la entrada y salida hay que tener en cuenta también el paso por mensajes de MPI, dado que para escribir en disco es necesario que todos los procesos comuniquen su trozo de la imagen al proceso con rank 0, por lo tanto, aquellas pruebas con un N>1 añadirán a su tiempo total de ejecución el tiempo de comunicación, por lo tanto, como se puede apreciar, a mayor N, mayor tiempo de ejecución, llegando a ser peor que la versión secuencial cuando N=8.

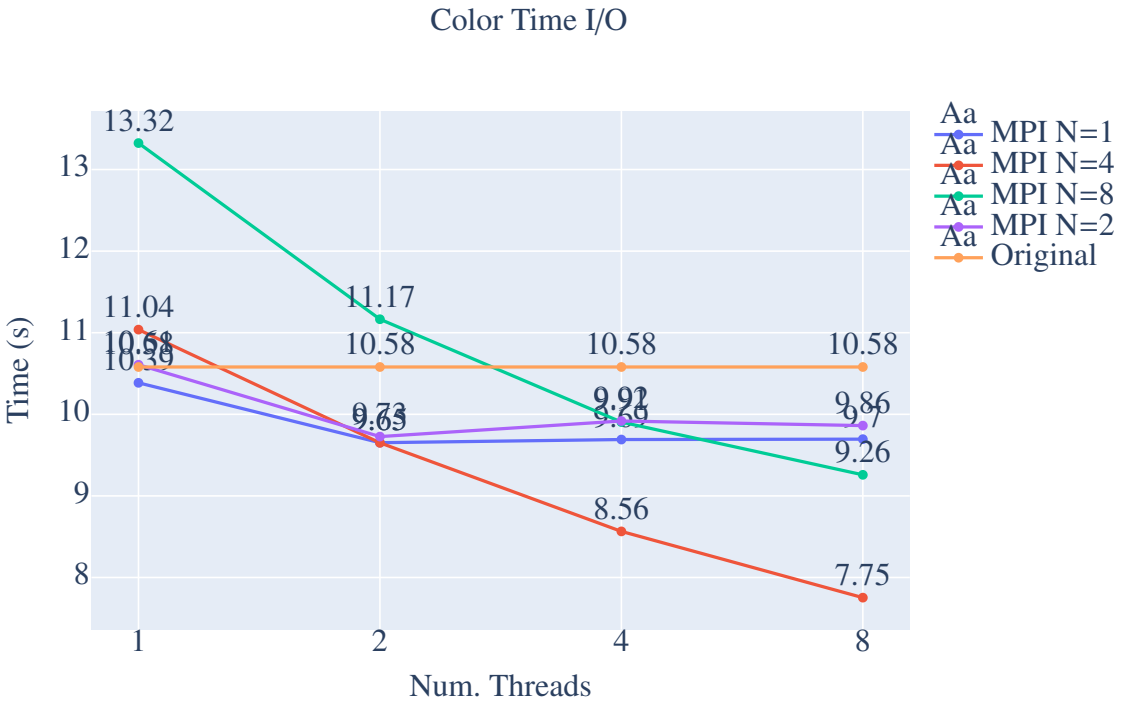

Fig. 2.4. Color. Tiempos paralelos (con I/O) Secuencial vs MPI+OpenMP.

De forma similar a la escala de grises, un número pequeño de procesos de MPI mejora ligeramente el rendimiento y un número elevado lo empeora debido al paso por mensajes.

No obstante, esto empieza a cambiar en cuanto aumentamos el número de hilos disponibles para cada proceso ya que , a diferencia de la escala de grises, en color se trabaja con tres canales (r,g,b), lo que implica el triple de datos a comunicar entre procesos, y es hasta que no se alcanza un cierto nivel de paralelismo (por ejemplo, 4 procesos y 8 hilos) que aproveche al máximo la CPU, no se obtiene una mejora significativa.

Sin embargo, contrario a la suposición inicial de que sobrecargar el procesador lanzando tantos procesos como núcleos e hilos de OpenMP podría ser contraproducente, se obtiene el mejor rendimiento posible para este caso. Se considera que esto se debe a que con 4 procesos el tamaño de la imagen tiene el tamaño de grano perfecto para con 8 hilos de OpenMP, no haya demasiados cambios de contexto en los núcleos y se encuentre trabajando la mayor parte del tiempo. No como puede ocurrir con 8 procesos.

Este mismo comportamiento se puede apreciar en el procesamiento de HSL y YUV.

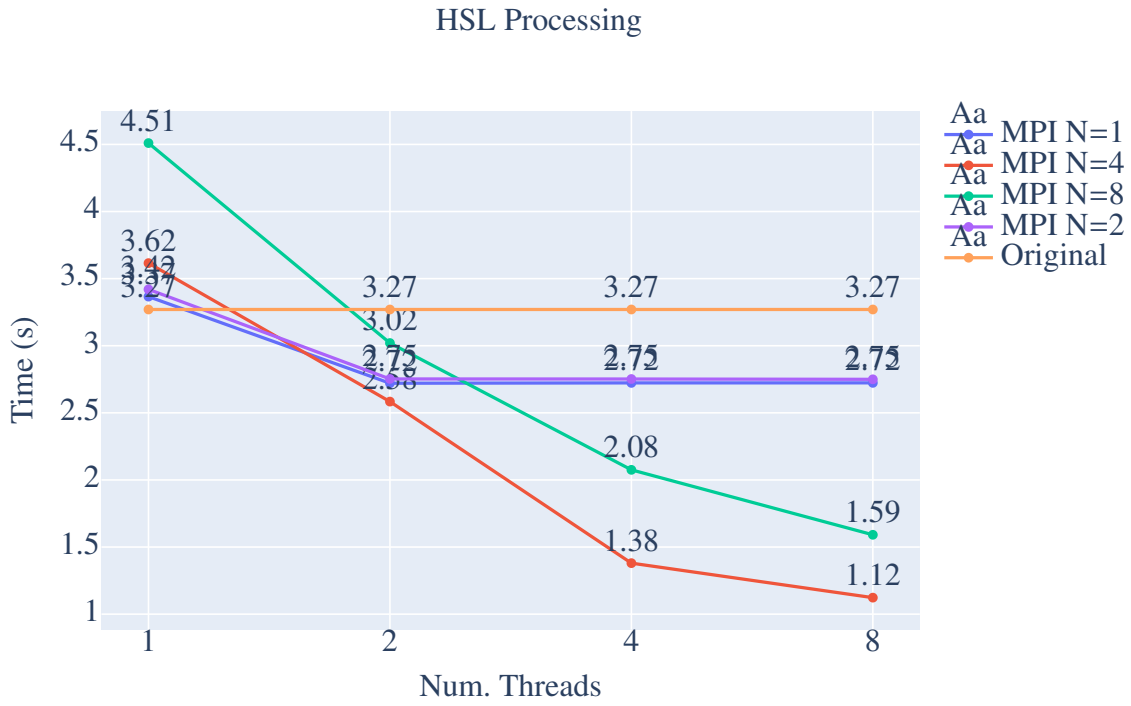

Fig. 2.5. HSL. Tiempos paralelos (sin I/O) Secuencial vs MPI+OpenMP.

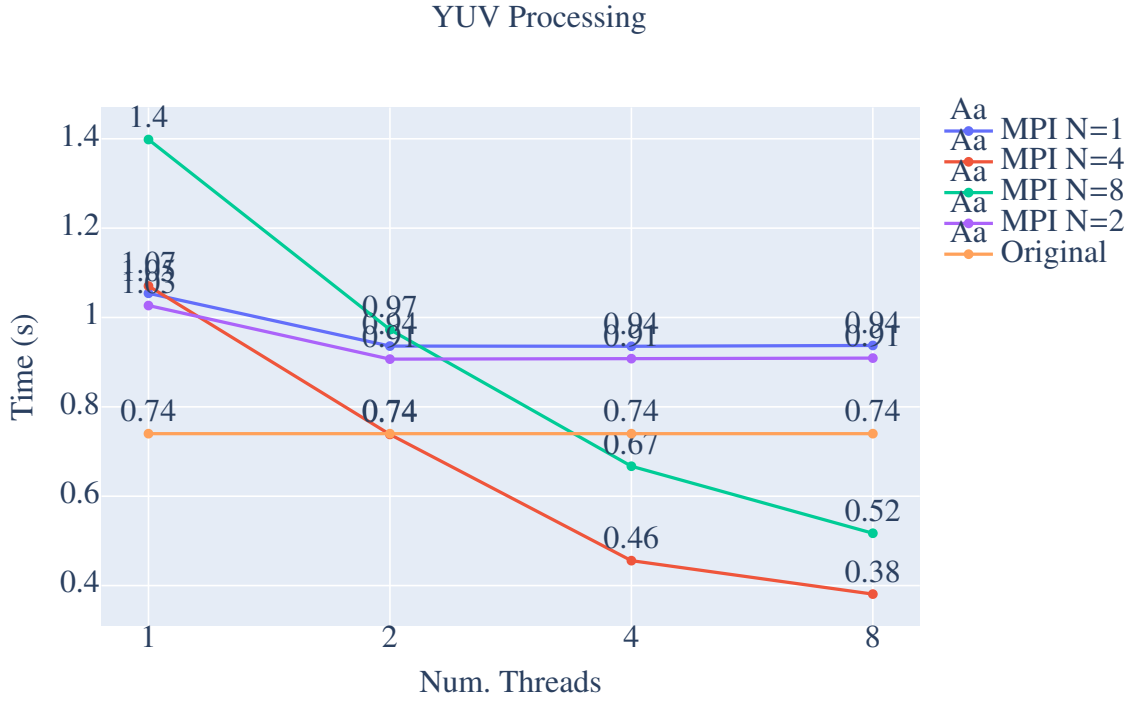

Fig. 2.6. YUV. Tiempos paralelos (sin I/O) Secuencial vs MPI+OpenMP.

Observando los resultados obtenidos, se puede deducir que para la configuración actual, los mejores resultados se van a obtener usando 4 procesos de MPI y 8 hilos de OpenMP. Con esta configuración, se obtiene el siguiente histograma de tiempos en comparación con su versión secuencial.

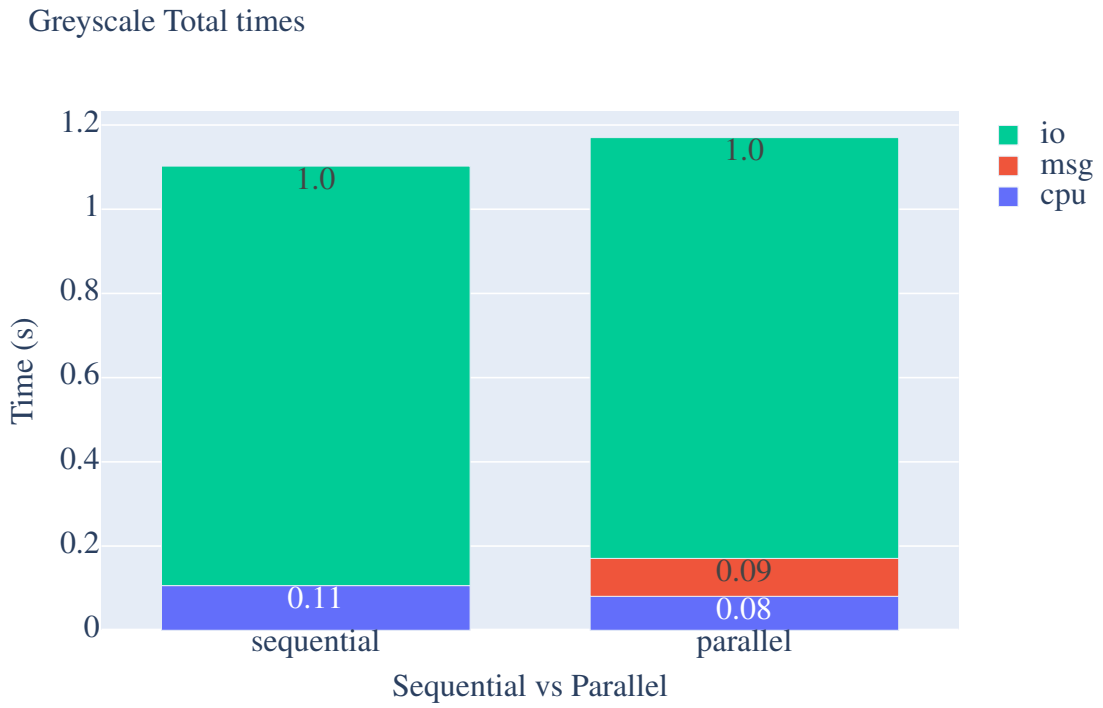

Fig. 2.7. Histograma acumulado - Escala de grises - Secuencial vs Paralelo.

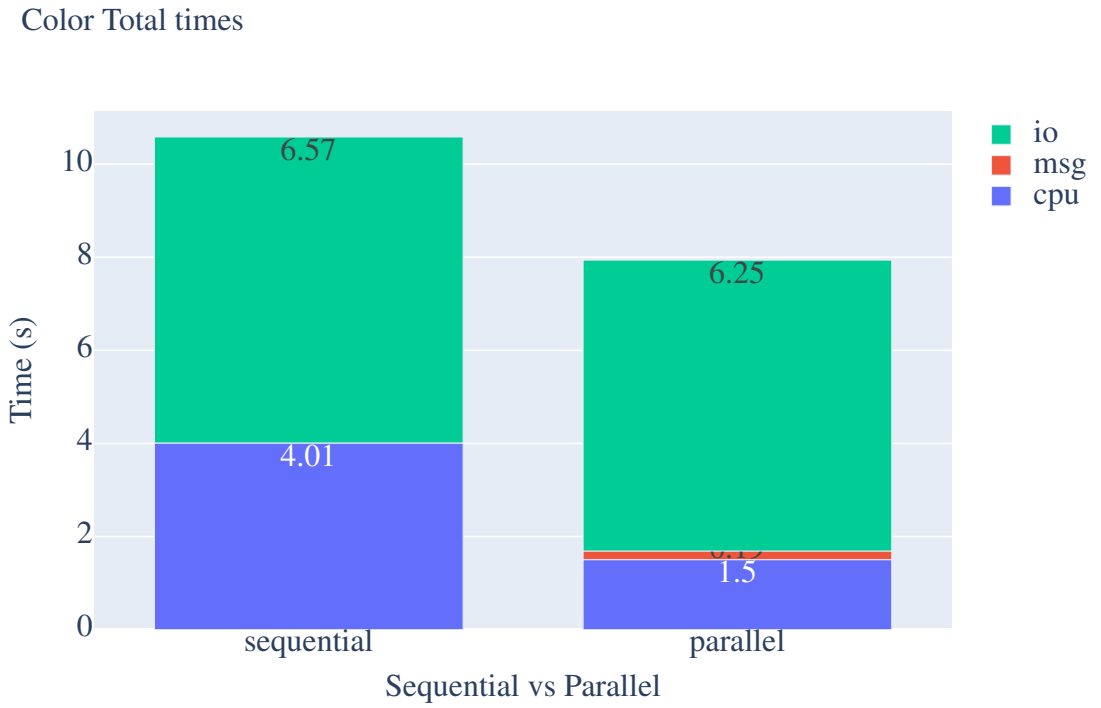

Fig. 2.8. Histograma acumulado - Color - Secuencial vs Paralelo.

Donde se aprecia claramente que los tiempos de entrada y salida se mantienen constante para la escala de grises y mejoran ligeramente en color debido a una pequeña parelización a la hora de reconstruir la imagen. Los tiempos de CPU descienden considerablemente y aparece un tiempo que antes no existía en la versión secuencial, el tiempo de los pasos de mensajes de MPI.

Este tiempo de paso de mensajes hace que la versión paralela para imágenes en escala de grises sea exáctamente igual a la versión original o incluso un poco peor, como se vió previamente, dado que el tiempo de CPU se reduce a la mitad, pero el tiempo de los mensajes es igual al de la CPU, por lo tanto se mantiene estable. En cambio, para las imágenes a color el tiempo de mensajes es bastante inferior al tiempo de cómputo (que se reduce casi 4 veces), por lo que el *overhead* que añade MPI es despreciable.

Los tiempos previamente obtenidos durante las pruebas se pueden transforman en los siguientes *speed ups*:

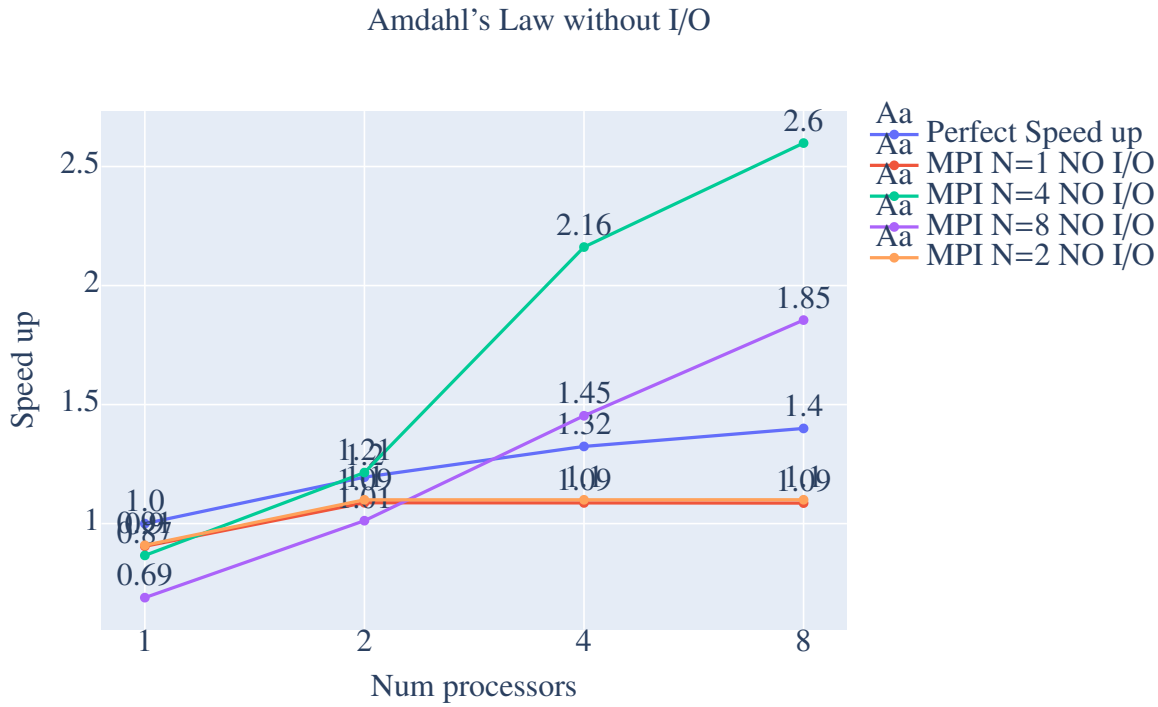

Fig. 2.9. Speedup MPI+OPenMP sin I/O frente a la Ley de Amdahl.

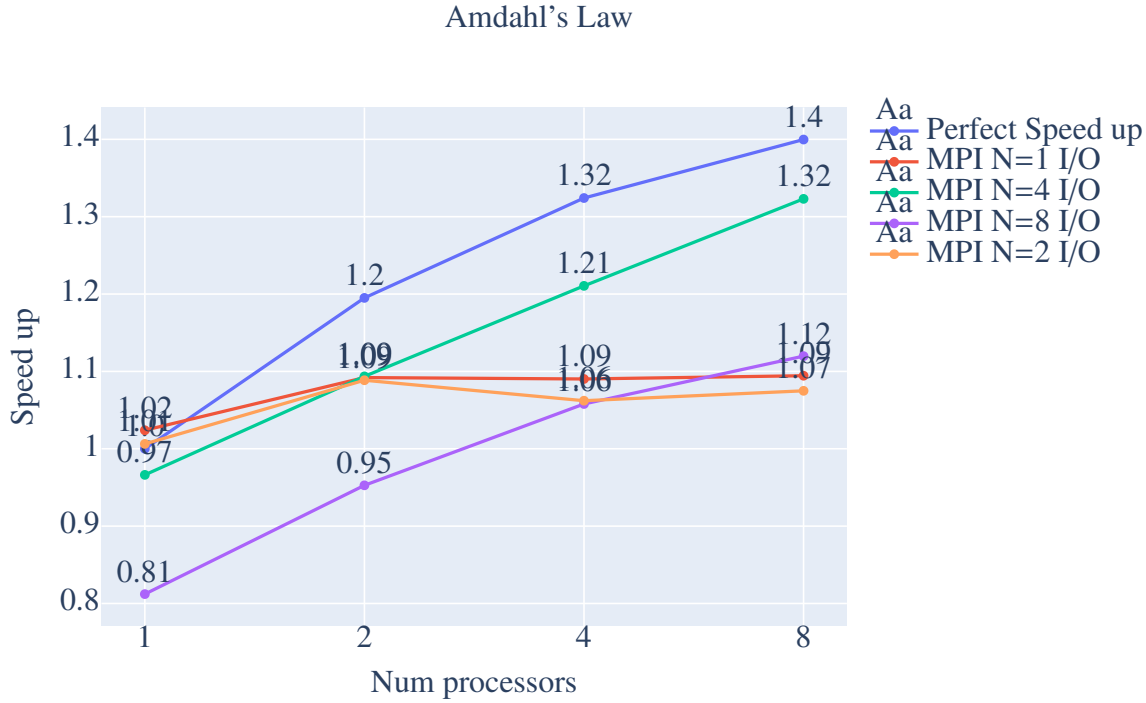

Fig. 2.10. Speedup MPI+OPenMP con I/O frente a la Ley de Amdahl.

Como se pudo apreciar en las figuras 2.7 Histograma acumulado - Escala de grises - Secuencial vs Paralelo. y 2.8 Histograma acumulado - Color - Secuencial vs Paralelo., lo que más retrasa la ejecución es el tiempo de entrada y salida, siendo los casos en los que se ignora para el tiempo total cuando se consigue obtener un *speed up* super lineal. No obstante, si se introducen estos procesos de lectura y escritura, el *speed up* empeora considerablemente y, por tanto, obteniendo un resultado más realista con lo que se puede esperar a la hora de paralelizar código.

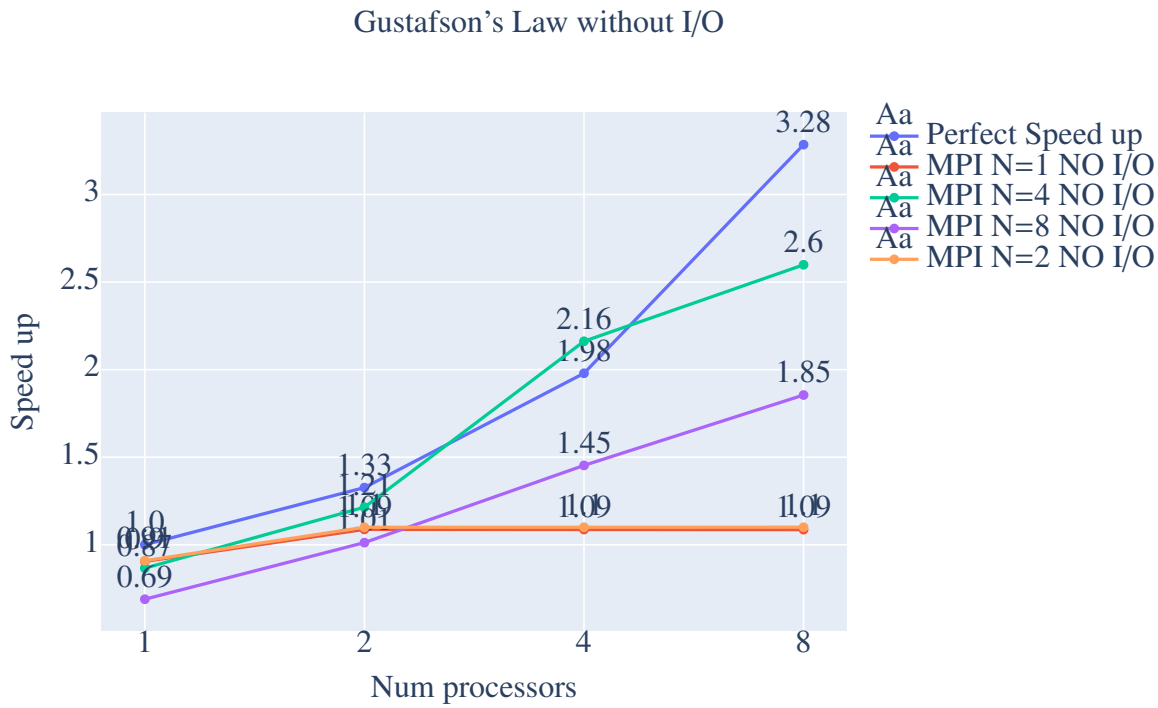

Fig. 2.11. Speedup MPI+OPenMP sin I/O frente a la Ley de Gustafson.

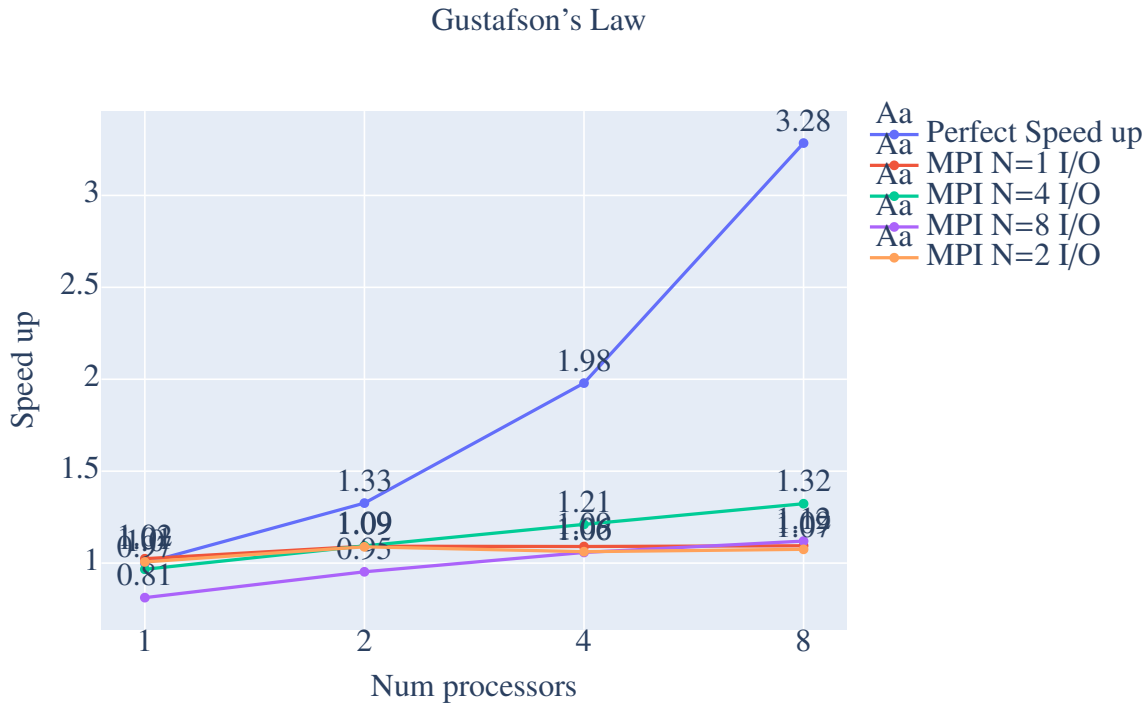

Fig. 2.12. Speedup MPI+OPenMP con I/O frente a la Ley de Gustafson.

Como se puede apreciar, ocurre el mismo comportamiento que en la Ley de Amdahl y exáctamente por los mismos motivos. Aunque a diferencia de Amdahl, se empieza a estancar con Nº hilos = 8 cuando no consideramos la entrada y salido, y esto se debe a que el tamaño de la imagen ha sido constante durante las pruebas.

3. CONCLUSIONES

La implementación de soluciones de computación distribuida mediante MPI es una forma relativamente sencilla de repartir datos a lo largo de varias máquinas sin mucha complicación, sin embargo, las modificaciones y aspectos a tener en cuenta a la hora de adaptar programas secuenciales hace que sea un proceso delicado.

Por otro lado, OpenMP es muchísimo más sencillo de utilizar y nos permite introducir un gran nivel de paralelismo sin ningún problema, ni apenas modificaciones en el código original. Sin embargo, a diferencia de MPI, es mucho más complicado paralelizar el reparto de datos cuando no se trabaja con tipos básicos, por ejemplo en el caso que nos atañe, dividir un array en trozos equitativos y repartirlos, operar sobre ellos y luego recomponerlo.

Cada estándar tiene sus pros y sus contras, y es importante conocer de antemano el entorno de ejecución de nuestro programa para poder aprovechar el máximo de cada uno de ellos.

BIBLIOGRAFÍA

[1] OpenMP. (2004). OpenMP, OpenMP, [En línea]. Disponible en: https://www.openmp.org/ (Acceso: 01-11-2020).

[2] SPI-INC. (2004). Open MPI: Open Source High Performance Computing, [En línea]. Disponible en: https://www.open-mpi.org/ (Acceso: 01-11-2020).

[3] J. Weidendorfer. (2013). KCachegrind, [En línea]. Disponible en: https://kcachegrind.github.io/html/Home.html (Acceso: 22-11-2020).