Escuela
Politécnica
Superior

Algoritmos paralelos para la reducción de ruido mixto gaussiano-impulsivo en imágenes médicas

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:

Ignacio Encinas Rubio

Tutor:

Josep Arnal García

Universitat d'Alacant
Universidad de Alicante

Algoritmos paralelos para la reducción de ruido mixto gaussiano-impulsivo en imágenes médicas

Filtrado de imágenes en tiempo real acelerado con C++, OpenMP y MPI

Autor

Ignacio Encinas Rubio

Tutor

Josep Arnal García

Departamento de Ciencia de la Computación e Inteligencia Artificial

Grado en Ingeniería Informática

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Mayo 2022

Preámbulo

Las imágenes digitales suelen verse contaminadas mediante ruido durante el proceso de adquisición, transmisión y almacenamiento. Por lo tanto, el procesamiento de dichas imágenes con el fin de eliminar el ruido preservando la mayor cantidad de información posible es un problema fundamental en el campo. Los algoritmos que obtienen buenos resultados suelen ser especialmente costosos, por lo que el objetivo de este trabajo es la implementación de un filtro de ruido mixto gaussiano-impulsivo capaz de funcionar prácticamente en tiempo real.

Se ha partido de métodos previamente propuestos e implementados que han sido modificados para trabajar de manera distribuida en un *cluster*. Se ha utilizado MPI para trabajar de manera distribuida y OpenMP para paralelizar a nivel de nodo.

El trabajo ha sido evaluado en el Cluster de Computación del Instituto Universitario de Investigación Informática.

Por último, agradecer al *Ministerio de Educación y Formación Profesional* la subvención de este trabajo a través del programa de *Becas de colaboración 2021-22*.

Agradecimientos

En primer lugar, quiero dedicarle estos agradecimientos a mi tutor, Josep Arnal, por su completa atención y disponibilidad a lo largo de tantos meses de trabajo. Ha sido un placer contar con su apoyo y supervisión para hasta el más mínimo detalle.

Me gustaría agradecer a la *Vim gvng* el apoyo brindado y todo lo que me han enseñado a lo largo de los años. Especialmente a Ernesto, por las configuraciones de vim robadas y por nuestras peleas sobre qué parámetros se pasan por registro y cuáles en el stack. He tenido la suerte de tener buenos compañeros de clase que han hecho algo más amenos estos dos últimos años de carrera que han sido de lo más desconcertante.

Por último, agradecer a todos esos profesores que preparan sus asignaturas con dedicación que hacen que graduarse valga la pena.

Índice general

Índice de figuras

Todas las figuras son de elaboración propia

Índice de tablas

Todas las tablas son de elaboración propia

Índice de Códigos

Todo el código listado es de elaboración propia

1. Introducción

La tomografía computarizada (CT en inglés) es una herramienta médica esencial utilizada extensivamente para el diagnóstico y procedimientos guiados por imagen. Las imágenes CT contienen ruido consecuencia de las técnicas de adquisición y los intentos de reducir la radiación al paciente. Por lo tanto, es necesario reducir el ruido presente para poder analizar los resultados obtenidos adecuadamente. El ruido gaussiano aditivo es el tipo de ruido más común en estas imágenes (Gravel y cols., 2004; Lu y cols., 2002; Lei y Sewchand, 1992; J. y cols., 2019). Puede presentarse combinado con otro tipo de ruido, dificultando el tratamiento de la imagen. En el presente trabajo estudiaremos el filtrado de la combinación de ruido gaussiano e impulsivo en imágenes CT.

El ruido gaussiano está caracterizado porque su función de densidad de probabilidad es igual a la de la distribución normal (Ecuación 1.1).

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{1.1}$$

donde x denota la intensidad, μ el valor promedio de x, σ la desviación estándar y σ^2 la varianza.

El ruido impulsivo, también conocido como «sal y pimienta», presenta la siguiente función de densidad de probabilidad:

$$f(x) = \begin{cases} P_a & x = a \\ P_b & x = b \\ 0 & x \neq a, x \neq b \end{cases} \tag{1.2}$$

Donde las intensidades a y b aparecerán como puntos brillantes y oscuros respectivamente (asumiendo $a > b$). En el caso de que alguna de las probabilidades P_a o P_b sean 0, el ruido se denomina unipolar.

Los procesos de reducción de ruido pueden trabajar en el dominio espacial o de frecuencia (Gonzalez y Woods, 2008). Los filtros basados en la frecuencia operarán sobre la transformada de Fourier de la imagen, mientras que los filtros basados en el dominio espacial lo harán sobre los propios píxeles de la imagen (Sánchez Cervantes, 2013). En el presente trabajo nos centraremos en los filtros basados en el dominio del espacio.

Muchos algoritmos han sido propuestos para reducir el ruido gaussiano (Tomasi y Manduchi, 1998; Schulte y cols., 2006; Dabov y cols., 2007b,a) o impulsivo (Smolka, 2010; Camarena y cols., 2008; Toprak y Güler, 2007). Sin embargo, no todos los métodos son apropiados cuando las imágenes están contaminadas por ambos tipos de ruido.

Un enfoque posible es el de reducir el ruido consecutivamente aplicando distintos filtros. El principal inconveniente es la posible pérdida de rendimiento, lo que podría volverlo inviable. Por lo tanto, disponer de filtros que trabajen con un ruido mixto es conveniente (Arnal y cols., 2020; Arnal y Súcar, s.f.).

2. Marco Teórico

El presente trabajo está basado en (Camarena y cols., 2013). En dicha publicación se propone un filtro basado en lógica difusa para reducir el ruido gaussiano-impulsivo en imágenes a color. Dicho método ha sido adaptado para tratar con imágenes monocromáticas obtenidas mediante tomografía computarizada. Además, se ha desarrollado un filtro iterativo y se han estudiado ciertas modificaciones que facilitan la paralelización del filtro para la obtención de imágenes en tiempo real.

2.1. Lógica difusa

A continuación explicaremos brevemente los fundamentos de la lógica difusa y posteriormente el funcionamiento del método estudiado. Para ello nos basaremos en (Morillas Gómez, 2008).

2.1.1. Conjuntos difusos

La lógica difusa trabaja con conjuntos difusos. Esto es una generalización de la teoría clásica de conjuntos que permite la membresía parcial. En lugar de determinar la pertenencia a un conjunto como un valor binario (cierto, falso) lo haremos con un valor comprendido entre 0 y 1, correspondiéndose los valores 0 y 1 con los valores clásicos de cierto y falso.

2.1.2. Variables lingüísticas

Las variables lingüísticas tienen la función de permitirnos describir cualitativamente los valores que toman nuestras variables. Esto es de gran relevancia porque facilita la comprensión de estos sistemas y el desarrollo de las reglas que utilizaremos para regir el sistema (Sección 2.2.4)

2.1.3. Reglas «if-then»

Las reglas fuzzy «if-then» son una generalización del *modus ponens*. Recordemos que *modus ponens* nos permite inferir a partir de la proposición lógica r que si p es cierto, q también lo es.

$$p \rightarrow q \tag{r}$$

Esta manera de proceder no nos permite trabajar con los valores de verdad parciales que utilizamos en la lógica difusa. La inferencia difusa, por lo tanto, nos debe permite inferir a partir del grado de certeza del antecedente p. Una vez hemos realizado la inferencia satisfactoriamente podremos realizar un proceso de defuzzificación para producir valores de verdad clásicos.

Por lo tanto, el funcionamiento del sistema puede resumirse en tres fases: en primer lugar caracterizaremos las entradas mediante grados de membresía (Secciones 2.2.2 y 2.2.3), a continuación construimos las reglas difusas (Sección 2.2.4) y por último realizaremos la defuzzificación ayudados por variables lingüísticas (Sección 2.2.5) y las reglas difusas (Sección 2.2.6)

2.2. Método difuso

Una vez explicadas las nociones básicas procederemos a explicar el funcionamiento del filtro.

Consideramos una ventana W de tamaño $n \times n$ centrada en cada pixel $x_i \in [0, 255]$ de la imagen ruidosa. El método propuesto calculará el nuevo valor $\hat{x_i}$ mediante una media ponderada considerando solo q píxeles $x^j \in W$.

$$\hat{x_i} = \frac{\sum_{j=1}^{q} \omega_j \cdot x^j}{\sum_{j=1}^{q} \omega_j}, \tag{2.1}$$

Donde los pesos $\omega_j \in [0, 1]$ se obtienen mediante defuzzificación tras haber aplicado el sistema de lógica difusa que describiremos posteriormente.

2.2.1. La métrica Rank-Ordered Absolute Differences (ROAD) para la detección de impulsos

Dado un pixel x_i, consideremos la ventana W de tamaño $n \times n$ centrada en x_i. A continuación, ordenaremos los píxeles $x^j \in W$ según su distancia al pixel central x_i, definida como $d(x_i, x_j) = |x_i - x^j|$. Una vez dispongamos de dichos píxeles $x^1, x^2, \ldots, x^{n^2}$ calcularemos la métrica ROAD_m definida en Garnett y cols. (2005) de la siguiente manera:

$$\text{ROAD}_m(x_i) = \sum_{j=1}^{m} d(x_i, x^j). \tag{2.2}$$

En nuestro caso concreto ROAD_m es un número entero en el intervalo $[0, 255 \cdot (m-1)]$. Esta métrica nos aporta una medida de lo parecido que es cada pixel x_i a sus m vecinos más similares. La idea subyacente es que los impulsos tendrán un valor de esta métrica alto, mientras que los píxeles que no han sido contaminados por este tipo de ruido tendrán un valor bajo. Por lo tanto, un valor bajo de $\text{ROAD}_m(x_i)$ significa que x_i tiene m píxeles x^j similares a él, por lo que se espera que no sea un impulso. En la Figura 2.1 se ilustra el proceso descrito previamente.

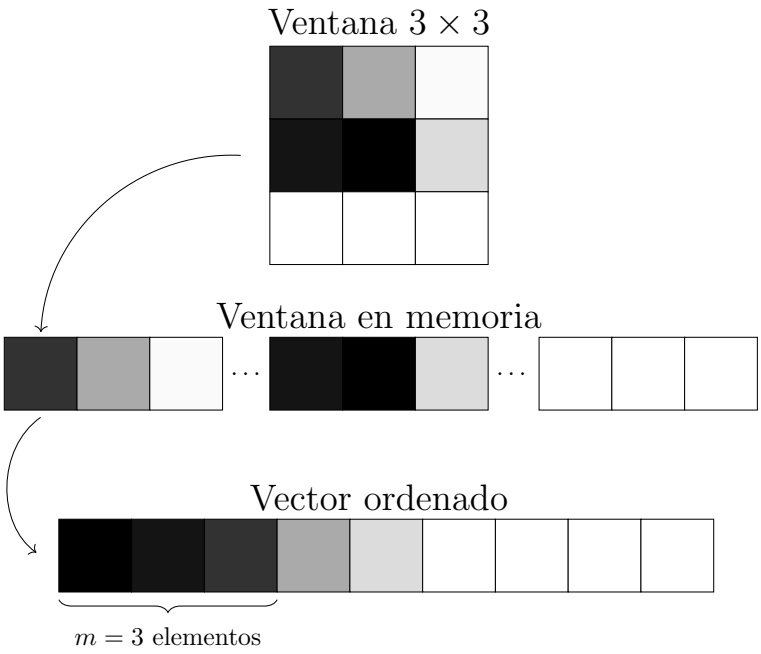


Figura 2.1: Ordenación de los píxeles para el cálculo del grado de impulsividad del pixel central

2.2.2. Determinación del grado de impulsividad

Consideramos el conjunto difuso determinado por el siguiente grado de membresía μ. Definimos el grado de impulsividad de un pixel x_i por la función.

$$\mu(x_i) = \begin{cases} 0, & \text{ROAD}_m(x_i) \leq p_1 \\ \frac{\text{ROAD}_m(x_i) - p_1}{p_2 - p1}, & p_1 < \text{ROAD}_m(x_i) < p_2 \\ 1, & \text{ROAD}_m(x_i) \geq p_2 \end{cases} \tag{2.3}$$

Utilizando este grado de membresía podemos definir el grado de que un pixel no sea un impulso mediante el operador de negación estándar $1 - \mu(x_i)$. La Figura 2.2 presenta los correspondientes conjuntos difusos μ y $1 - \mu$ definidos en $[0, 255 \cdot (m-1)]$.

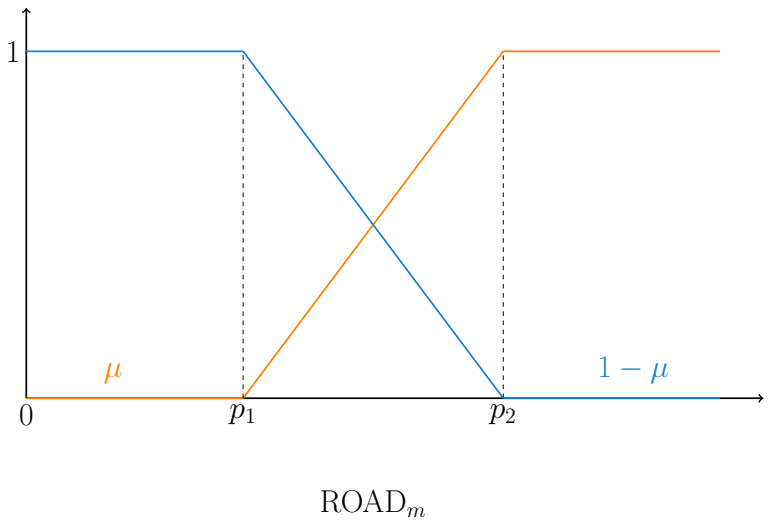


Figura 2.2: Grado de impulsividad para un pixel x_i.

2.2.3. Grado de semejanza

Utilizaremos la variable lingüística "Semejanza". Esta variable puede tomar valores "alta", "moderada" y "baja". Esto nos dará información de la similitud entre un píxel x y sus vecinos $x^j \in W$, siendo W la ventana de filtrado. Utilizaremos dicha información en nuestras reglas difusas.

Para el valor "alta" consideramos el conjunto difuso determinado por la siguiente función de membresía:

$$\mu_H(x_i, x^j) = \begin{cases} 1, & d(x_i, x^j) \leq p_3 \\ \frac{-d(x_i,x^j)}{3p_3} + \frac{4}{3} & p_3 < d(x_i, x^j) < 4p_3 \\ 0, & d(x_i, x^j) \geq 4p_3. \end{cases} \tag{2.4}$$

Y usando la negación estándar, definimos la función de membresía correspondiente al valor "baja" como $\mu_L(x_i, x^j) = 1 - \mu_H(x_i, x^j)$.

El grado de membresía para el valor "moderada" $\mu_H(x_i, x^j)$ se define como

$$\mu_M(x_i, x^j) = \begin{cases} \frac{d(x_i,x^j)-p_3}{p_3}, & p_3 < d(x_i, x^j) < 2p_3 \\ 1, & 2p_3 \leq d(x_i, x^j) \leq 3p_3 \\ \frac{4p_3-d(x_i,x^j)}{p_3} & 3p_3 < d(x_i, x^j) < 4p_3 \\ 0, & \text{en otro caso} \end{cases} \tag{2.5}$$

La Figura 2.3 muestra la variable "Semejanza" y los correspondientes conjuntos difusos, μ_H, μ_M y μ_L definidos en $[0, 255]$

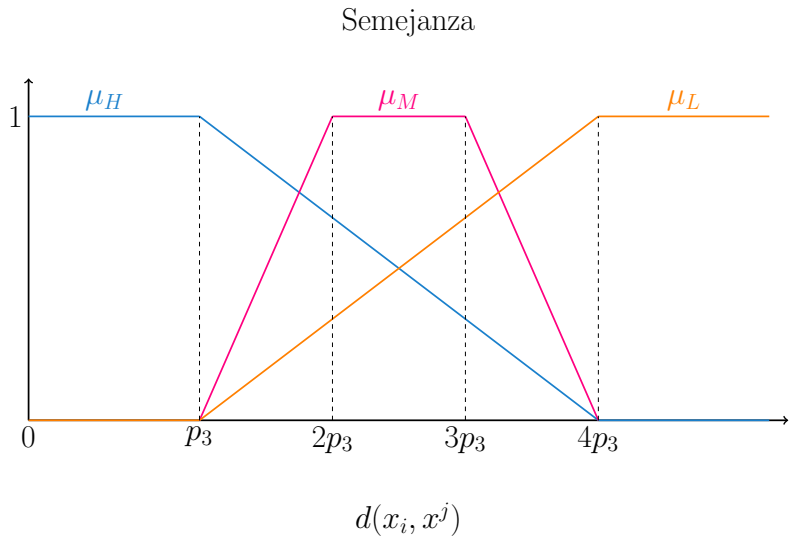


Figura 2.3: Grado de semejanza para un pixel x^j en función de $d(x_i, x^j)$

2.2.4. Reglas difusas

1. **SI** (x^j es impulsivo) **O** (x^j no es impulsivo **Y** x_i es impulsivo **Y** la semejanza entre x^j y x_i es alta) **O** (x^j no es impulsivo **Y** x_i no es impulsivo **Y** la semejanza entre x^j y x_i es moderada **O** (x^j no es impulsivo Y x_i no es impulsivo Y la semejanza entre x^j y x_i is baja) **ENTONCES** ω_j es un peso bajo.

2. **SI** (x^j no es impulsivo **Y** x_i es impulsivo **Y** la semejanza entre x^j y x_i es moderada) **ENTONCES** ω_j es un peso moderado.

3. **SI** (x^j no es impulsivo **Y** x_i es impulsivo **Y** la semejanza x^j y x_i es baja) **O** (x^j no es impulsivo **Y** x_i no es impulsivo **Y** la semejanza entre x^j y x_i es alta) **ENTONCES** ω_j es un peso alto.

2.2.5. Ponderación mediante coeficientes difusos

Para representar la información de salida del sistema difuso consideramos una variable lingüística para caracterizar los pesos ω_j en el proceso de ponderación. Esta variable puede tomar los valores “alto”, “moderado” y “bajo”. Los grados de membresía $\eta_H(\omega_j)$, $\eta_M(\omega_j)$ y

$\eta_L(\omega_j)$ correspondientes a los valores "alto", "moderado" y "bajo" son dadas por las siguientes expresiones

$$\eta_H(\omega_j) = \begin{cases} \frac{\omega_j - 1}{1 - p_4} + 1, & p_4 < \omega_j \leq 1 \\ 0, & \text{en otro caso} \end{cases} \tag{2.6}$$

$$\eta_M(\omega_j) = \begin{cases} \frac{2\omega_j - 1}{2p_4 - 1} + 1, & 1 - p_4 < \omega_j \leq \frac{1}{2} \\ \frac{1 - 2\omega_j}{2p_4 - 1} + 1, & \frac{1}{2} < \omega_j < p_4 \\ 0, & \text{en otro caso} \end{cases} \tag{2.7}$$

$$\eta_L(\omega_j) = \begin{cases} \frac{\omega_j}{p_4 - 1} + 1, & 0 \leq \omega_j \leq 1 - p_4 \\ 0, & \text{en otro caso} \end{cases} \tag{2.8}$$

2.2.6. Defuzzificación

El valor del peso ω_j correspondiente a cada pixel $x^j \in W$ se obtiene mediante defuzzificación. Para este paso, utilizamos el centro de gravedad (COG en inglés) (Passino y Yurkovich, 1998; Driankov y cols., 1996). Sean K_L, K_M y K_H los grados de membresía asociados a las reglas difusas 1, 2 y 3 correspondientemente. Tomando los tres triángulos determinados por η_H, η_M y η_L y las tres funciones constantes K_H, K_M y K_L en sus respectivas zonas de aplicación se forman tres trapecios (ver Figura 2.4). Si tomamos L_j como la línea poligonal determinada por la partes superior y lateral de dichos trapecios, el peso viene dado por

$$\omega_j = \frac{\int_0^1 x \cdot L_j(x) dx}{\int_0^1 L_j(x) dx}. \tag{2.9}$$

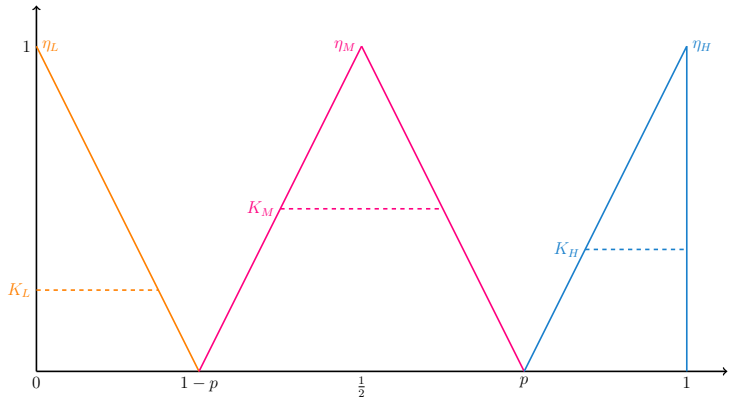


Figura 2.4: Funciones de membresía $\eta_H(\omega_j)$, $\eta_M(\omega_j)$, y $\eta_L(\omega_j)$ junto a K_L, K_M, K_H correspondientes a las reglas difusas $\{1, 2, 3\}$ respectivamente para determinados píxeles x_i y x^j

La ecuación (2.9) puede reescribirse como la ecuación (2.10), una expresión más simple en términos de las áreas A_i y las proyecciones en el eje de abcisas de los centros de gravedad C_{G_i} asociados a los polígonos descritos anteriormente (véase la Figura 2.5). Esta observación que puede resultar evidente es crucial en términos de la implementación (detallado en la sección 5.2.3).

$$\omega_j = \frac{\sum_i A_i \cdot C_{G_{i_x}}}{\sum_i A_i}. \tag{2.10}$$

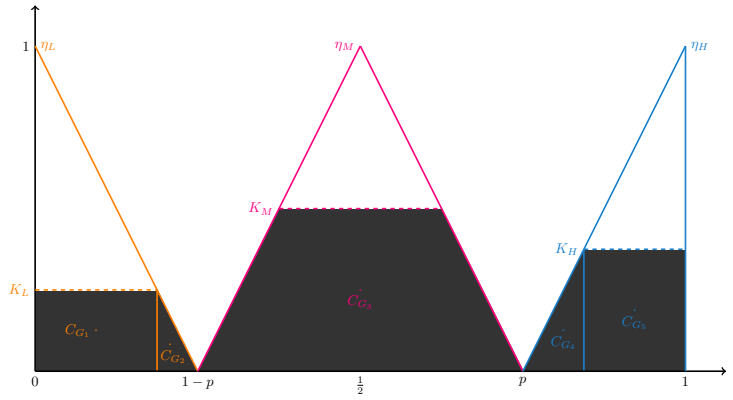


Figura 2.5: Áreas y centros de gravedad asociados a la ecuación 2.10

3. Objetivos

El objetivo de este trabajo es el de aplicar las técnicas de la computación de altas prestaciones al filtrado de imágenes obtenidas mediante tomografía computarizada (CT en inglés) frecuentemente utilizadas en medicina. En concreto, trataremos el ruido impulsivo-gaussiano que se manifiesta en la obtención de dichas imágenes. Se implementará un filtro previamente propuesto basado en lógica difusa.

Los métodos basados en la lógica difusa ofrecen una calidad de filtrado muy buena, pero presentan el inconveniente de tener un coste computacional muy alto. Por esto, se han estudiado modificaciones al algoritmo para acercarnos al filtrado en tiempo real.

A partir de la implementación secuencial se han obtenido paralelizaciones tanto en memoria compartida como en memoria distribuida. Adicionalmente, se han propuesto un esquema de comunicaciones propio para la versión distribuida que reduce el número de comunicaciones sin afectar a la calidad de filtrado. Esta reducción en el número de comunicaciones se traduce en una reducción significativa en el tiempo de ejecución.

4. Metodología

4.1. Implementación

La implementación ha sido realizada en C++. Es uno de los lenguajes más utilizados en la Computación de Altas Prestaciones junto a Fortran y C. La implementación paralela en un único nodo ha sido realizada con OpenMP (Dagum y Menon, 1998) y la versión distribuida mediante MPI (Forum, 1994) + OpenMP.

4.2. Organización

En la Figura 4.1 podemos ver una representación del trabajo realizado.

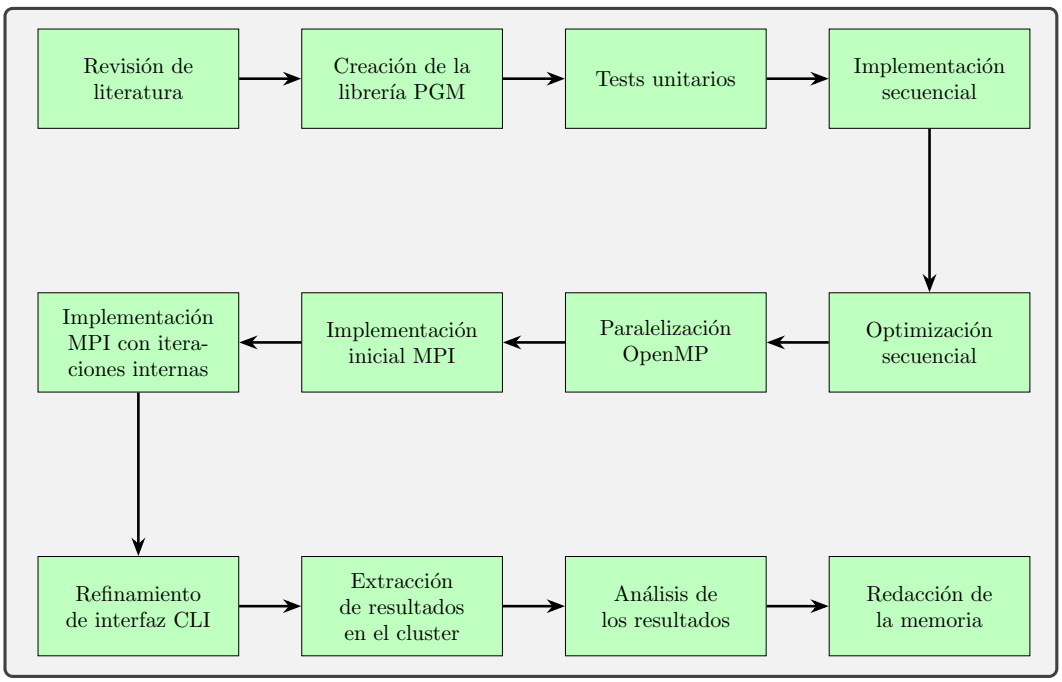


Figura 4.1: Organización del trabajo realizado

Inicialmente, se realizó un estudio general de la bibliografía, comparando distintos métodos basados en lógica difusa. Una vez se seleccionó el algoritmo sujeto a estudio se desarrolló una librería para el manejo de imágenes PGM y el uso de ventanas de filtrado sobre ellas. Esto nos permite controlar todos los detalles de nuestra implementación.

A continuación se implementó el filtro secuencialmente. Acto seguido se desarollaron las versiones multihilo y multinodo mediante OpenMP y MPI respectivamente.

Por último, obtuvimos los resultados en los distintos equipos de prueba.

4.3. Corrección

Para controlar la corrección de nuestra implementación hemos utilizado los sanitizers que nos ayudan a detectar comportamiento no definido, fugas de memoria y condiciones de carrera. Adicionalmente, se han desarrollado tests unitarios con la biblioteca Boost ut.

4.4. Setups experimentales

Para la ejecución y el análisis del rendimiento del código se han utilizado dos equipos distintos. El Equipo 1 se trata del ordenador personal del alumno mientras que el Equipo 2 se trata del Cluster del Instituto Universitario de Investigación Informática.

- Equipo 1:
  - (1x) AMD Ryzen 2700X (8C/16T), 16GB RAM DDR4 3200MHz
  - ArchLinux. Kernel 5.16.12
  - GCC 11.2.0
- Equipo 2. Cluster de 26 nodos equipados con:
  - (2x) CPU Intel Xeon X 5660 (6C/12T), 48GB RAM DDR3 1333Mhz
  - CentOS 7. Kernel 3.10.0
  - GCC 7.5.0
  - OpenMPI 4.0.2

4.5. Benchmarking

Para la obtención de tiempos de ejecución generales y el estudio de partes concretas del código en el Equipo 1 hemos utilizado la biblioteca Google Benchmark.

En el Equipo 2 hemos utilizado un script en bash que interactúa directamente con el gestor de recursos del cluster.

4.6. Análisis del rendimiento

Para el análisis del rendimiento y el estudio de métricas concretas se han utilizado las herramientas *perf* de Linux, así como distintos *profilers* como Tracy o AMD μProf. Dichas herramientas han motivado las distintas decisiones de implementación que se tomaron durante el desarrollo de la aplicación.

5. Desarrollo

5.1. Dependencias

Utilizaremos distintos *frameworks* y bibliotecas detalladas a continuación:

- OpenMP
- MPI
- Cxxopts
- Boost ut

5.1.1. Formato de imágenes

Se ha utilizado el formato de imágenes PGM (Portable Gray Map) por su simplicidad. Esto nos permite controlar completamente la implementación y sus detalles, pudiendo así optimizar todos los aspectos necesarios. Se ha desarrollado una pequeña biblioteca que nos permite tratar con dicho formato transparentemente pese a controlar su representación y que además nos facilita implementar cualquier algoritmo sobre este tipo de imágenes de manera cómoda. La biblioteca nos brinda dos clases.

Como hemos adelantado las imágenes PGM son muy simples. La única peculiaridad es que utilizamos el tipo *double* internamente para no incurrir en errores de redondeo cada iteración, ya que nuestro filtro produce valores en coma flotante para cada pixel filtrado. En cualquier caso se podría valorar el usar un tipo entero para reducir notablemente[1] el uso de memoria y aceptar el error que esto conlleva. Véase el Código 5.1.

[1]El tipo double ocupa 8 veces más espacio que el tipo uint8_t

Código 5.1: Miembros de la clase Pgm

```
class Pgm {
public:
    using img_t = double;
    using store_t = uint8_t;

protected:
    uint32_t cols;
    uint32_t rows;
    uint32_t max_gray;
    std::vector<img_t> data;
}
```

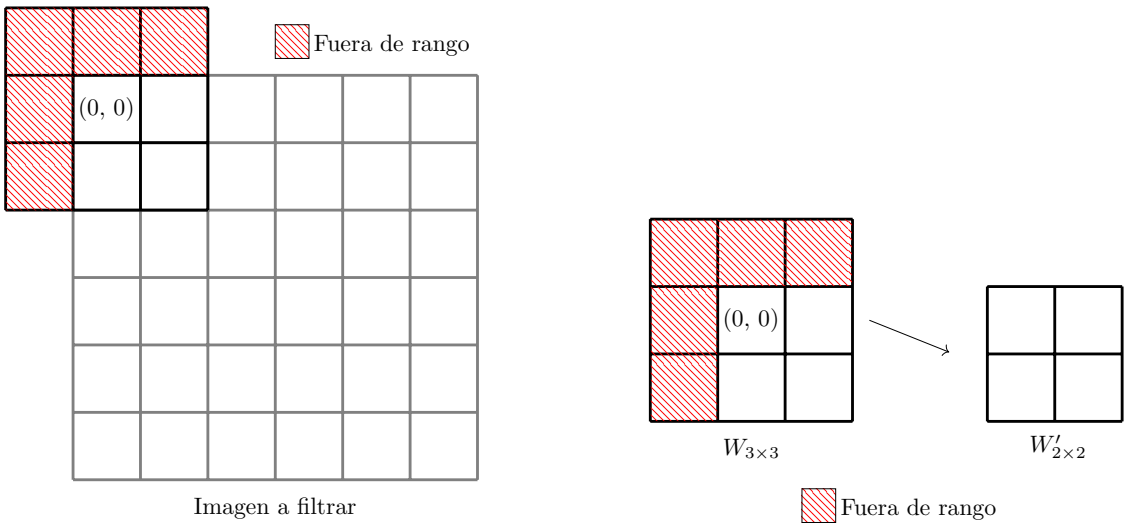
Por otro lado, tenemos la clase *PgmWindow* que nos permite crear ventanas de tamaño arbitrario centradas en cada pixel. Su implementación resulta algo más compleja de lo que a primera vista podría parecer porque hay que lidiar con los bordes de la imagen. Una técnica común en estos casos consiste en añadir ceros alrededor de la imagen pero en nuestro caso concreto esto afectaría al resultado del algoritmo, por lo que no resulta de utilidad. Lo que se decidió fue que en el momento de la construcción de la ventana se compruebe si esta se sale de la imagen y si fuese el caso, se ajuste su tamaño al máximo disponible. Podemos ver este proceso representado en la Figura 5.1. Véase el Código 5.2 para ver la definición de la clase.

Código 5.2: Miembros de la clase PgmWindow

```
class PgmWindow {
    /*
     * Número de columnas que tiene la imagen original (Pgm).
     * Necesario para implementar el operador [] en nuestra clase.
     * Para más detalle consultar el código fuente
     */
    uint32_t originalNumCols;

    /*
     * Filas y columnas finales. En el ejemplo de la Figura 5.1b
     * tendríamos que cols = 2 y rows = 2
     */
    uint32_t cols;
    uint32_t rows;

    /* Puntero a la esquina superior izquierda de la ventana final */
    img_t *data;
    /*
     * Puntero al centro de la ventana. No es posible recuperarlo siempre
     * en las ventanas de los bordes de la imagen.
     */
    img_t *center;
}
```



(a) Creación de una ventana en una de las esquinas de la imagen

(b) Ventana resultante tras las comprobaciones pertinentes

Figura 5.1: Problemática de la creación de ventanas en los bordes de la imagen

Pese a no ser el objetivo principal del trabajo se ha intentado que la biblioteca fuese cómoda de utilizar. No es excesivamente extensa y está documentada detalladamente con doxygen. En el Código 5.3 disponemos de un simple ejemplo donde se aplica un filtro de mediana.

Código 5.3: Ejemplo de uso de la biblioteca

```
1   /* Función aportada por el usuario */
2   img_t medianFilter(const PgmWindow& window);
3
4   /* Definiciones y funciones aportadas por la biblioteca */
5   using FilterFunc = img_t (*)(const PgmWindow& w);
6   Pgm applyFilter(FilterFunc filter, uint32_t windowSize) const;
7
8   /* Código de ejemplo */
9   Pgm image("resources/imgA.pgm");
10  Pgm filtered = image.applyFilter(medianFilter, /* Tam. ventana */ 3);
11  filtered.Store("imgA_filtered.pgm");
```

5.1.2. OpenMP

A la hora de paralelizar código secuencial una de las opciones a considerar es OpenMP. Según lo simple que sea el esquema paralelo de la aplicación en cuestión puede realizarse sorprendentemente rápido. Por otra parte es cierto que en los últimos estándares de C++ se ha avanzado en el soporte de programas multihilo integrado en la biblioteca estándar del lenguaje. Pese a los avances que comentamos nuestro esquema resulta especialmente fácil de expresar mediante las primitivas que ofrece OpenMP, por lo que esta fue la opción escogida.

5.1.3. MPI

Por otra parte a la hora de distribuir nuestro programa para poder ejecutarlo en un cluster la herramienta por excelencia es *Message Passing Interface* (MPI), cuyo estándar 4.0 ha sido publicado en 2021. El código ha sido desarrollado y probado en el Cluster del Instituto Universitario de Investigación Informática de la Universidad de Alicante. Allí disponemos de varias implementaciones de MPI como *MPICH, MVAPICH* y *OpenMPI*. Aunque no sea representativo la implementación de *OpenMPI* es la que mejor resultados ofrece en este caso concreto.

5.1.4. Usabilidad

Nuestra aplicación se le presenta al usuario mediante una *Interfaz de Línea de Comandos* o *CLI*. Para ello nos hemos ayudado de la biblioteca cxxopts que se encarga de procesar los parámetros introducidos por el usuario a través de la terminal. Hay muchas opciones como esta pero su simplicidad y la facilidad que presenta para introducirla en la construcción de nuestro programa al ser una biblioteca *Header-only* nos hizo decantarnos por ella. Podemos ver el resultado en el Código 5.4

Código 5.4: Interfaz de usuario

```
./mpi-gaussian-impulsive --help

Fuzzy filter for images with gaussian-impulsive noise.

Usage:
  mpirun [OPTION...] ./mpi-gaussian-impulsive [OPTION...]

  -w, --window_size arg        Window size. The resulting window will be (2*ws + 1)x(2*ws + 1)
                               For example, window_size = 1 will produce a 3x3 window
                               (default: 1)
  -q, --q_elements arg         Number of elements inside the NxN window that will actually be
                               used to filter each pixel. (default: 7)
  -r, --road_elements arg      Number of elements to be considered in the ROAD_m statistic.
                               (default: 3)
  -i, --iterations arg         Number of iterations (default: 5)
      --input_image arg        Path to the image to be filtered. Sigma from the gaussian noise
                               must be specified inside *_g{sigma}_*.pgm
      --output_image arg       Path where the filtered image will be stored (default:
                               filtered_image.pgm)
      --original_image arg     Path where the original image is stored
      --number_of_threads arg  Number of threads to be spawned by OPENMP (default: 0)
      --inner_iterations arg   Number of independent iterations to be carried out in each node
                               before they communicate (default: 1)
  -h, --help                   Print help
```

5.2. Implementación

5.2.1. Rutinas comunes a todas las implementaciones

Todas las implementaciones tienen como parte común las funciones que se encargan de expresar en software los componentes básicos del filtro descrito en la Sección 2. En el Código 5.5 listamos la representación en C++ de las rutinas más importantes referenciando a las secciones donde fueron introducidas. Omitimos detalles de implementación.

Código 5.5: Funciones comunes a todas las implementaciones

```
1   Pgm::img_t RoadStatistic(uint32_t numNeighbours);
2
3   /*
4    * Calculamos el grado de impulsividad descrito en 2.2.2
5    * Toma como parámetro una NWindow asociada a una PgmWindow
6    * descrita en 5.1.1 con ciertas peculiaridades cuya
7    * motivación se desarrolla en 5.3.1
8    */
9   double ImpulsivityDegree(const NWindow &nwindow);
10
11  /* Funciones de membresía para las variables lingüísticas
12   * descritas en 2.2.3
13   */
14  double HighSimilarityEstimation(img_t distance);
15  double LowSimilarityEstimation(img_t distance);
16  double ModerateSimilarityEstimation(img_t distance);
17
18  /* Implementación de la inferencia difusa descrita en 2.2.4 */
19  double S_norm(double x);
20
21  template <typename... Args>
22  double S_norm(double x, Args... args);
23
24  double ModerateFuzzyRule(double idegree_i, double idegree_j,
25      double MSE);
26  double HighFuzzyRule(double idegree_i, double idegree_j,
27      double HSE, double LSE);
28  double LowFuzzyRule(double idegree_i, double idegree_j,
29      double HSE, double MSE, double LSE);
30
31  /*
32   * Cálculo de los pesos. Sección 2.2.6
33   * Esta función devuelve el peso asociado al pixel central de adjcent x^j
34   * respecto al pixel central x_i.
35   */
36  double calculateWeight(const NWindow &central, const NWindow &adjacent);
37
38  /* Obtención del nuevo valor del pixel según la ecuación 2.1 */
39  img_t updatePixel(NWindow &central);
```

5.2.2. Estructura de variables

Para calcular el grado de impulsividad de cada pixel en esencia necesitamos un vector ordenado que contenga las distancias entre x_i y sus vecinos x^j (incluido él mismo). Por lo tanto, nos basta un vector de tamaño $(2w+1)^2$ (en los bordes y esquinas tenemos menos vecinos). Para evitar hacer reservas de memoria constantemente para cada pixel usaremos una variable estática (local a cada hilo). Para ello nos valdremos del almacenamiento local al hilo o *Thread Local Storage*. En concreto, emplearemos *thread_local*. Se incorporó al estándar de C++ en la versión C++11 a partir de esta propuesta.

Para no recalcular el grado de impulsividad constantemente utilizaremos también un vector que mapee cada pixel x_i con su grado de impulsividad $\mu(x_i)$. El objetivo de esta estructura es evitar repetir el cálculo del grado de impulsividad. En la Sección 5.3.1 se discute este tema en más profundidad aportando datos sobre la aceleración que esto supone.

Por otra parte necesitamos un vector también de tamaño $(2w+1)^2$ para almacenar las ventanas adyacentes. En la implementación concreta almacenamos también la distancia entre la ventana y el pixel central, además de su posición en la imagen global para poder acceder al almacén de grados de impulsividad previamente descrito. Véase el Código 5.6 ver la declaración de las variables globales.

Código 5.6: Variables globales

```
/* Vector de las ventanas adyacentes al pixel x_i */
thread_local std::vector<NWindow> neighbourWindows;

/*
 * Buffer reutilizado para calcular el ROAD de cada pixel x_i
 * Almacenamos las distancias entre x_i y sus vecinos x^j
 * ordenamos el vector y sumamos desde el indice 0 al indice m.
 * Véase la Sección 2.2.1 para más información.
 */
thread_local std::vector<img_t> distances;

/*
 * Almacén previamente mencionado para evitar recalcular el grado de
 * impulsividad para cada pixel más de una vez
 */
std::vector<double> impulsivityDegrees;
```

5.2.3. Cálculo de los pesos

Para el cálculo de los pesos tenemos que resolver el cálculo presentado en la ecuación 2.9. La primera aproximación considerada fue la más evidente: realizar este cálculo mediante integración numérica. Posteriormente, llegamos a la conclusión de que, como adelantamos en la Sección 2.2.6, existe una manera drásticamente más eficiente que la previamente considerada. El ejercicio de llegar a la ecuación (2.10) desde la ecuación (2.9) y las expresiones correspondientes a cada A_i y C_{G_i} se detallan en el Apéndice A. Véase el Código 5.7.

Código 5.7: Cálculo del peso del pixel central de la ventana *adjacent*

```
double calculateWeight(const NWindow &central, const NWindow &adjacent) {

    img_t x_i = central.window.Center();
    img_t x_j = adjacent.window.Center();

    double impulsivityDegree_i = ImpulsivityDegree(central);
    double impulsivityDegree_j = ImpulsivityDegree(adjacent);

    img_t pixelsDistance = Pgm::PixelDistance(x_i, x_j);
    double HSE = HighSimilarityEstimation(pixelsDistance);
    double MSE = ModerateSimilarityEstimation(pixelsDistance);
    double LSE = LowSimilarityEstimation(pixelsDistance);

    double LowFR = LowFuzzyRule(impulsivityDegree_i, impulsivityDegree_j,
            HSE, MSE, LSE);
    double ModerateFR = ModerateFuzzyRule(impulsivityDegree_i,
            impulsivityDegree_j, MSE);
    double HighFR = HighFuzzyRule(impulsivityDegree_i, impulsivityDegree_j,
            HSE, LSE);

    double Area1 = LowFR * (1 - LowFR) * (1 - P4);
    double Area2 = LowFR * LowFR * (1 - P4) / 2;
    double AreaM = ModerateFR * (2 - ModerateFR) * (2 * P4 - 1) / 2;
    double Area6 = (1 - P4) * HighFR * HighFR / 2;
    double Area7 = HighFR * (1 - HighFR) * (1 - P4);

    double CG1 = (1 - LowFR) * (1 - P4) / 2;
    double CG2 = (3 - 2 * LowFR) * (1 - P4) / 3;
    double CGM = 0.5;
    double CG6 = (3 * P4 + 2 * HighFR - 2 * P4 * HighFR) / 3;
    double CG7 = (1 - HighFR) * (1 - P4) / 2;

    double numerator = Area1 * CG1 + Area2 * CG2 + AreaM * CGM +
            Area6 * CG6 + Area7 * CG7;
    double denominator = Area1 + Area2 + AreaM + Area6 + Area7;

    return numerator / denominator;
}
```

A primera vista este cambio puede interpretarse como menor, pero nada más lejos de la realidad. Tenemos dos ventajas con este método, y ambas notables. Ahorramos gran cantidad de operaciones en coma flotante, lo que reduce considerablemente el tiempo de ejecución. En el caso concreto de la integración numérica con paso = 0.01 obtenemos aceleraciones cercanas a 45x como podemos ver en la Figura 5.2. Adicionalmente, la reducción de operaciones en coma flotante y el cálculo analítico previo disminuyen el error asociado al cálculo del peso.

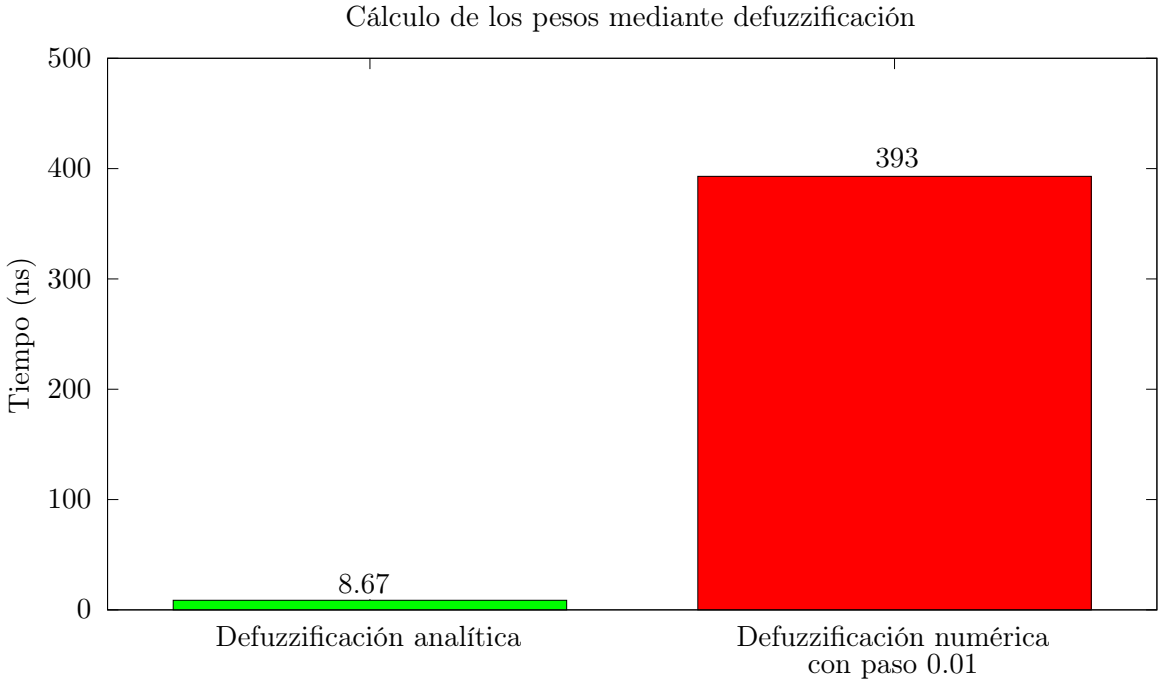


Figura 5.2: Comparación de la implementación final con la implementación ingenua inicialmente considerada. Mediciones realizadas en el Equipo 1

5.3. Implementación secuencial

Como puede deducirse de las Secciones 2 y 5.2.2 el filtro a implementar presenta ciertas peculiaridades pese a no ser excesivamente complejo. Si queremos desarrollar software con buen rendimiento debemos hacer un buen trabajo inicial al determinar su estructura y analizarlo en su forma secuencial previa a la paralelización para poder sacar conclusiones posteriormente y evaluar nuestro trabajo objetivamente. El Algoritmo 1 detalla nuestro punto de partida:

Algoritmo 1: Filtro difuso secuencial

Datos: Imagen ruidosa I, parámetros $n, q, m, p_1, p_2, p_3, p_4$

Resultado: Imagen filtrada I'

Imagen $I_0 = I$

para *Iteración* $it = 1, \ldots$ **hacer**

  Imagen $I_{it} = I_{it-1}$

  para x_i *pixel* $\in I_{it}$ **hacer**

    Tomar la ventana W $n \times n$ centrada en x_i

    Cálculo del grado de impulsividad

      Calcular $\mu(x_i)$ usando Ec. (2.3)

    Cálculo del grado de semejanza

      Ordenar los píxeles $x^j \in W$ según $d(x_i, xj)$

      Seleccionar los q píxeles mas cercanos $x^1, \ldots, x^q$

      para $j = 1, \ldots, q$ **hacer**

        Calcular $\mu_H(x_i, x^j), \mu_L(x_i, x^j), \mu_H(x_i, x^j)$, usando Ecs. (2.4), (2.5)

      fin

    Cálculo de los pesos mediante defuzzificación

      para $j = 1, \ldots, q$ **hacer**

        Calcular las reglas difusas para $\{x_i, x^j\}$

        Calcular el peso w_j correspondiente a x^j mediante COG

      fin

    Cálculo del nuevo valor para x_i

      $$\hat{x}_i = \frac{\sum_{j=1}^{q} \omega_j \cdot x^j}{\sum_{j=1}^{q} \omega_j}$$

  fin

fin

5.3.1. Implementación optimizada

En esta primera fase del desarrollo se obtuvo una solución especialmente lenta. Tras un análisis se determinó que el problema era fácilmente mitigable mediante la *memoización*[2]. El problema principal es que calculamos el grado de impulsividad para cada pixel x_i y x_j constantemente cuando únicamente necesitamos hacerlo una vez pues este no cambia en una misma iteración del filtro. Por lo tanto, la solución directa consiste en guardar ese grado de impulsividad para cada pixel x_i la primera vez que se calcula, y la próxima vez que se necesite simplemente leerlo. Discutiremos los efectos que tienen estos cambios en el coste temporal y espacial del filtro en la sección de resultados. Podemos ver los cambios aplicados en el Código 5.8.

Código 5.8: Introducción de la memoización

```
1 double ImpulsivityDegree(const NWindow &nwindow) {
2+    if (impulsivityDegrees[nwindow.index] != -1.0)
3+        return impulsivityDegrees[nwindow.index];
4     nwindow.window.distanceToNeighboursSorted(distances);
5     double road_m = RoadStatistic(nwindow.window.Size());
6
7     if (road_m <= P1) {
8+        impulsivityDegrees[nwindow.index] = 0;
9-        return 0;
10    } else if (road_m >= P2) {
11+        impulsivityDegrees[nwindow.index] = 1;
12-        return 1;
13    } else {
14+        impulsivityDegrees[nwindow.index] = (road_m - P1) / (P2 - P1);
15-        return (road_m - P1) / (P2 - P1);
16    }
17+   return impulsivityDegrees[nwindow.index];
18 }
```

[2] https://es.wikipedia.org/wiki/Memoización

5.4. Paralelización con OpenMP

La paralelización consiste en marcar como paralelo el núcleo del filtro mediante las directivas de OpenMP. El único detalle es que debemos controlar las variables globales descritas previamente. Marcandolas como thread_local e inicializandolas en nuestra primera entrada al bloque paralelo evitamos la problemática. El almacén global es compartido, y aunque puede ser que se calculen varias grados de impulsividad más de una vez por condiciones de carrera se ha mostrado como la opción más rapida. La sincronización entre hilos es costosa y la planificación estática (véase la Figura 5.3) del bucle reduce drásticamente esta posibilidad. En el Código 5.9 podemos ver la paralelización del núcleo del filtro.

Código 5.9: Filtro paralelizado con OpenMP

```
void fuzzyFilter(Pgm &imageIn, Pgm &imageOut, double &psnr) {
    std::fill(impulsivityDegrees.begin(), impulsivityDegrees.end(), -1.0);
#pragma omp parallel
{
    uint32_t range = 2 * windowSize + 1;
    if(distances.size() == 0)
        distances.resize(range * range);
    if(neighbourWindows.size() == 0)
        neighbourWindows.resize(range * range);

    #pragma omp for
    for(uint32_t x = 0; x < imageIn.Rows(); x++)
        for(uint32_t y = 0; y < imageIn.Cols(); y++) {
            updateNeighbourWindows(imageIn, x, y);
            NWindow currentWindow(0,
                    PgmWindow(imageIn, x, y, windowSize),
                    x * imageIn.Cols() + y);
            imageOut[x][y] = updatePixel(currentWindow);
        }
}
}
```

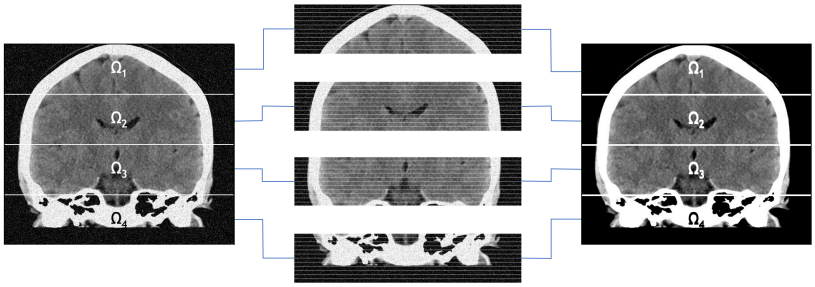


Figura 5.3: Descomposición del dominio en un entorno paralelo con 4 nodos. Bucle paralelizado con distribución estática OpenMP

5.4.1. Consideraciones adicionales

Cuando escribimos programas que utilizan memoria compartida tenemos que considerar el subsistema de memoria, cuyo comportamiento viene determinado por el modelo de memoria (Sorin y cols., 2011). Normalmente como programadores asumimos el modelo de consistencia secuencial, en el que el procesador no puede reordenar las operaciones sobre la memoria. Esta asunción viene reforzada por el comportamiento de los procesadores x86, que utilizan el llamado modelo *TSO* que resulta relativamente similar. Si queremos que nuestro código sea portable entre las distintas arquitecturas debemos seguir ciertas normas. Afortunadamente, disponemos de distintas primitivas que nos ayudan a sincronizar los distintos accesos a memoria sin abrumarnos con las distintas sutilezas, aunque a su vez pueden degradar el rendimiento innecesariamente.

Aquí es donde entran los llamados *Lockless algorithms*, o algoritmos sin cerrojos (Paolo Bonzini, 2021). Si nos aventuramos por este camino debemos tener en cuenta que podemos encontrarnos con distintos comportamientos no esperados. En (Aglave y cols., 2019) podemos ver algunos de ellos.

En nuestro caso concreto, la parte sensible a estos problemas es el almacén compartido que introdujimos en la Sección 5.3.1. Como el cálculo dependiente de la comprobación del almacén no produce efectos colaterales y gestionamos la sincronización con el valor de la variable a determinar podemos evitar el uso de primitivas de sincronización, cuyo efecto en el tiempo de ejecución sería notable.

5.5. Implementación multinodo: MPI + OpenMP

Una vez que disponemos de una implementación multihilo que utiliza la memoria compartida paralelizando la ejecución de nuestro filtro el siguiente paso es adaptarla para que sea capaz de ejecutarse en un entorno de memoria distribuida. Esto nos permitirá utilizar un cluster de computadores y reducir nuestro tiempo de ejecución.

Para distribuir la carga de trabajo consideraremos la partición de la imagen en los subdominios Ω_i $i = 1, \ldots, P$. Cada nodo será encargado de procesar uno de estos subdominios. Para poder realizar el filtrado correctamente necesitamos considerar los subdominios extendidos Ω_i^ω $i = 1, \ldots, P$ que contienen las w filas superiores e inferiores necesarias para construir las ventanas de filtrado en los píxeles de los bordes. Recordemos que el parámetro w determina el tamaño de la ventana de filtrado $W_{n\times n}$ de la siguiente manera: $n = 2\omega + 1$. En la Figura 5.4 podemos ver un ejemplo de este esquema.

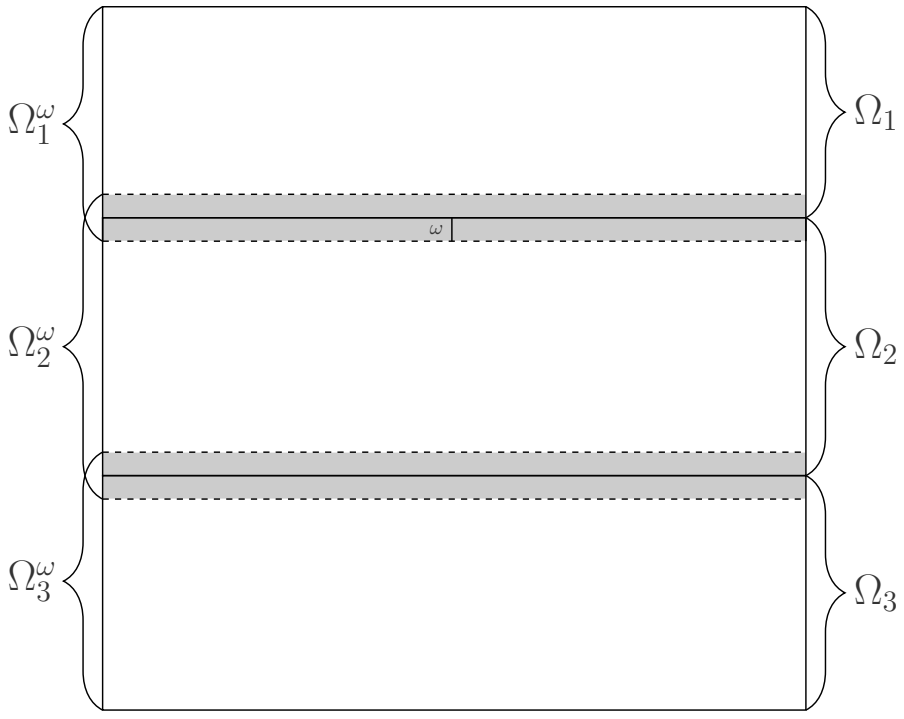


Figura 5.4: Descomposición del trabajo para 3 nodos

5.5.1. Esquema de comunicaciones

Las zonas de solapamiento de cada subdominio Ω_i^ω $i = 1, \ldots, P$ quedarán desactualizadas después de cada iteración. Por lo tanto, los nodos que filtren fragmentos adyacentes entre sí, deberán comunicarse para mantener su subdominio actualizado. Un nodo no podrá continuar filtrando hasta que dicha comunicación haya finalizado, por lo que esto genera un posible cuello de botella. Además, hemos de ser cuidadosos y realizar la comunicación de tal manera que evitemos los interbloqueos.

Con el fin de reducir el impacto de las comunicaciones en el rendimiento del filtro se ha estudiado la posibilidad de realizar iteraciones de manera local sin actualizar las zonas solapadas. La intuición nos dice que el impacto en la calidad del filtro debe ser menor puesto que la zona solapada afecta a muy pocos píxeles de cada subdominio Ω_i, mientras que las posibles ganancias de rendimiento pueden ser bastante significativas. Es por esto que consideramos otro parámetro z que determina el número de iteraciones internas que cada nodo hará antes de comunicarse con el resto. A continuación presentamos el Algoritmo 2 que presenta las modificaciones correspondientes.

Algoritmo 2: Filtro difuso paralelizado MPI-OpenMP

Datos: Imagen ruidosa I, parámetros $n, q, m, p_1, p_2, p_3, p_4, z$

Resultado: Imagen filtrada I'

Realizar descomposición en dominios $\{I_{\Omega_k}\}_{k=1}^{P}$ y asignarlos a los nodos

Imagen $I_0 = I$

para *Iteración* $it = 1, \ldots$ **hacer**

  Imagen $I_{it} = I_{it-1}$

  para *k = 1, ..., P, en paralelo* **hacer**

    Procesador k:

    para *Iteración interna* l *= 1, ..., z* **hacer**

      para x_i *pixel* $\in I_{it_{\Omega_k}}$ **hacer**

        Tomar la ventana W $n \times n$ centrada en x_i

        Cálculo del grado de impulsividad

          Calcular $\mu(x_i)$ usando Ec. (2.3)

        Cálculo del grado de semejanza

          Ordenar los píxeles $x^j \in W$ según $d(x_i, xj)$

          Seleccionar los q píxeles mas cercanos $x^1, \ldots, x^q$

          para $j = 1, \ldots, q$ **hacer**

            Calcular $\mu_H(x_i, x^j), \mu_L(x_i, x^j), \mu_H(x_i, x^j)$, usando Ecs. (2.4), (2.5)

          fin

        Cálculo de los pesos mediante defuzzificación

          para $j = 1, \ldots, q$ **hacer**

            Calcular las reglas difusas para $\{x_i, x^j\}$

            Calcular el peso w_j correspondiente a x^j mediante COG

          fin

        Cálculo del nuevo valor para x_i

          $$\hat{x}_i = \frac{\sum_{j=1}^{q} \omega_j \cdot x^j}{\sum_{j=1}^{q} \omega_j}$$

      fin

    fin

    Comunicación entre nodos

  fin

fin

Una vez se ha completado el filtrado el nodo maestro será el encargado de reconstruir la imagen a partir de los subdominios finales Ω_i.

6. Resultados

En la presente sección analizaremos los resultados obtenidos. Partiremos determinando nuestro conjunto de pruebas, a continuación discutiremos el rendimiento logrado en los distintos *benchmarks* y por último la calidad del filtro mediante métricas utilizadas en los estudios de referencia y la inspección visual.

6.1. Imágenes consideradas

Para realizar el estudio del algoritmo implementado hemos utilizado tres imágenes distintas que han sido contaminadas con cuatro configuraciones de ruido: $\{\sigma = 5,\ \rho = 0.05\}$, $\{\sigma = 10,\ \rho = 0.1\}$, $\{\sigma = 20,\ \rho = 0.2\}$ y $\{\sigma = 30,\ \rho = 0.3\}$. Siendo σ la desviación típica del ruido gaussiano y ρ la probabilidad de que un pixel presente ruido impulsivo.

Las imágenes usadas para el estudio proceden de la base de datos Radiopaedia. El caso concreto es cortesía del profesor Frank Gaillard con rID 35508. Utilizaremos las vistas sagital, axial y coronal presentes en la Figura 6.1

El criterio de parada utilizado por el método viene determinado por la evolución del PSNR. Puesto que utilizamos imágenes contaminadas por nosotros, podemos calcular el PSNR en cada iteración. Cuando detectemos que el PSNR deja de crecer pararemos el filtrado.

(a) Coronal **(b)** Axial **(c)** Sagital

Figura 6.1: Imágenes utilizadas para el estudio

6.2. Rendimiento de la implementación

Para estudiar el rendimiento utilizaremos las métricas fundamentales: el *speedup* y la eficiencia. Estas se definen como:

$$S_p = \frac{T_{sec}}{T_p} \tag{6.1}$$

$$E = \frac{S_p}{N} \tag{6.2}$$

$T_{\{sec,\ p\}}$ se refieren al tiempo secuencial y paralelo respectivamente. N representa el número de elementos de cómputo que intervienen en la paralelización.

Sin pérdida de generalidad, expondremos los resultados de la ejecución de nuestro filtro con unos parámetros[1] determinados. Dado que la carga computacional del filtro únicamente depende del número de píxeles basta con escoger un número de iteraciones suficientemente alto para obtener mediciones representativas. Los parámetros w, q y r toman los valores que resultan en mayor calidad de la imagen resultante. Por lo tanto, a continuación estudiaremos la evolución del rendimiento respecto al número de hilos, al número de nodos y posteriormente la aceleración que produce el uso del esquema de iteraciones internas descrito en el Apartado 5.5.1.

[1] Véase la sección 5.1.4 para un resumen de los parámetros disponibles

6.2.1. Secuencial

En la Sección 5.3.1 discutimos que utilizando técnicas de memoización reduciríamos drásticamente la cantidad de veces que calcularíamos el grado de impulsividad de cada pixel. En la Figura 6.2 podemos ver la aceleración que esto supone para varios valores del parámetro q. Dicha aceleración aumenta junto con el aumento de q y toma valores entre 2.17 y 3.5 para $q = 3$ y $q = 8$ respectivamente.

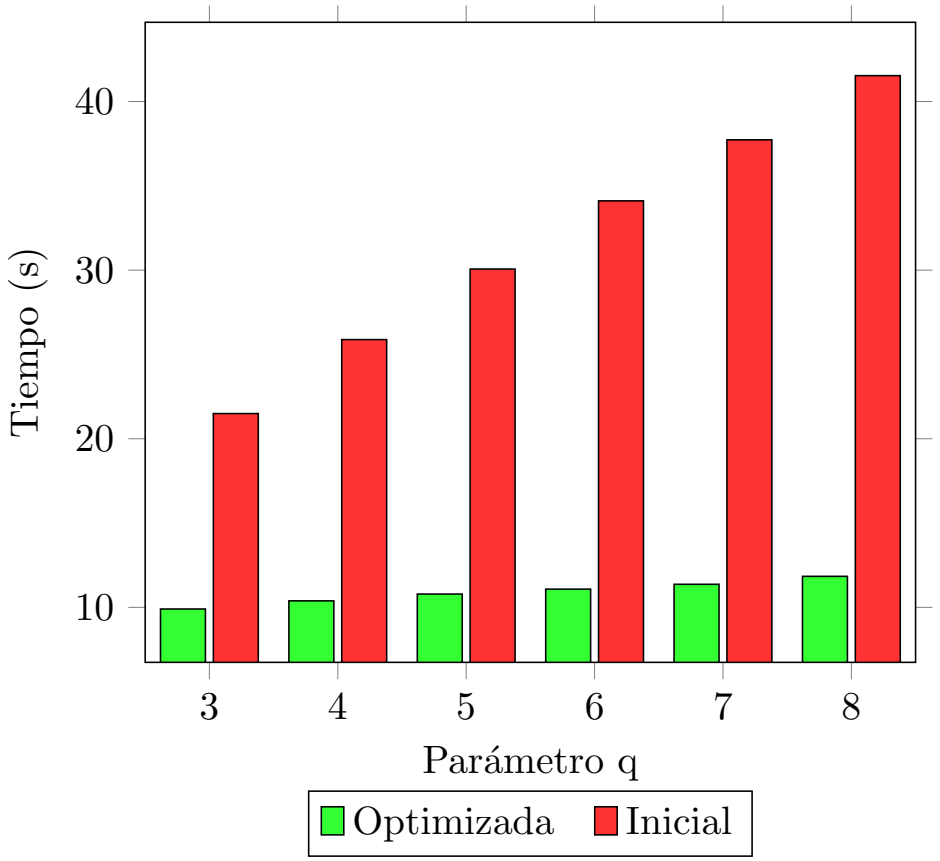


Figura 6.2: Filtrado secuencial de 30 iteraciones a la vista coronal $\{\sigma = 20, \rho = 0.2\}$. Equipo 1

6.2.2. OpenMP

A continuación estudiaremos la eficiencia de la paralelización en ambos equipos de pruebas. En las Figuras 6.3 y 6.4 disponemos de los tiempos obtenidos y la eficiencia.

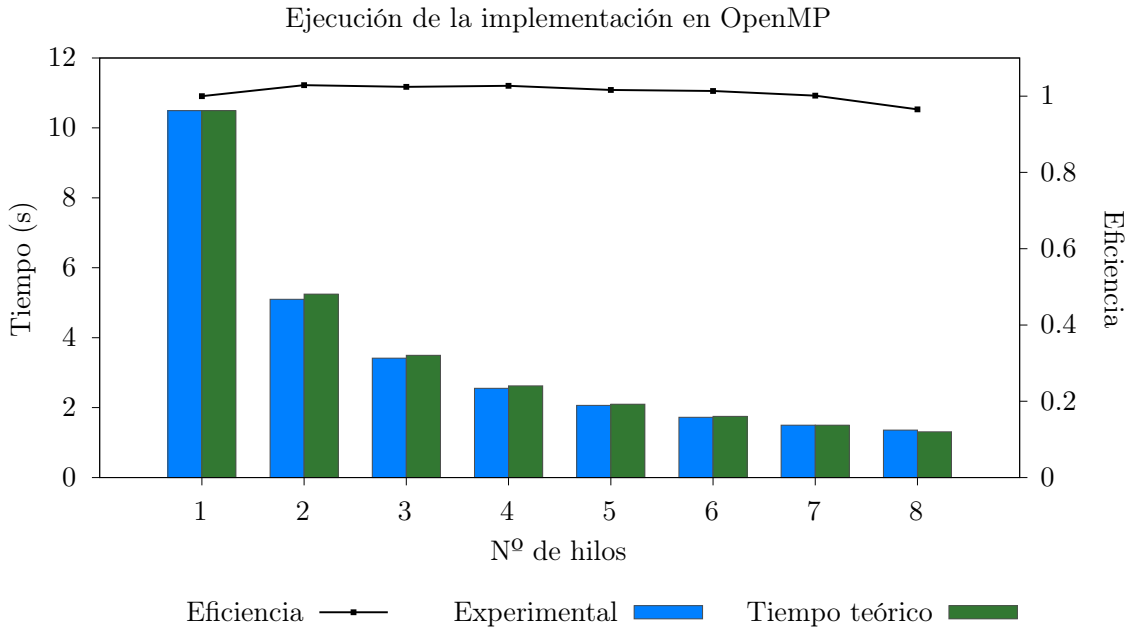


Figura 6.3: Filtrado paralelo de 30 iteraciones a la vista coronal $\{\sigma = 20, \rho = 0.2\}$. Equipo 1

Los resultados en el Equipo 1 muestran una eficiencia cercana e incluso superior al máximo teórico. Por supuesto, hay que tener en cuenta que en la práctica es posible conseguir el llamado *superlinear speedup* (Ristov y cols., 2016). Los resultados se obtuvieron con google benchmark, previamente configurando la frecuencia de la CPU a su máximo tal y como indica la documentación.

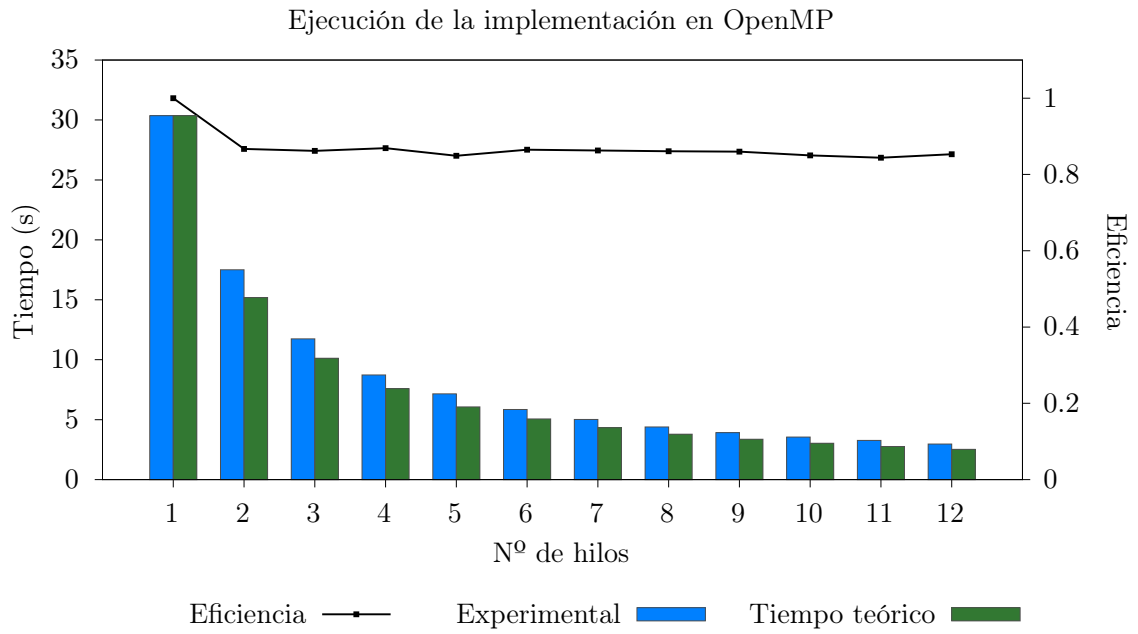


Figura 6.4: Filtrado de 60 iteraciones a la vista axial $\{\sigma = 30, \rho = 0.3\}$. Equipo 2

Por otra parte el Equipo 2 no presenta resultados tan buenos. En términos generales la eficiencia es buena puesto que es consistentemente superior al 80% pero la diferencia es notable respecto al Equipo 1. También es cierto que estos dispositivos son *dual-socket* y además tienen desactivado el *hyperthreading*.

6.2.3. MPI

Una vez disponemos de una implementación paralela capaz de utilizar un procesador en su totalidad debemos dar el siguiente paso, que es desarrollar el filtro para ejecutarlo en memoria distribuida. Con ello conseguiremos reducir drásticamente el tiempo de ejecución.

A continuación estudiaremos el rendimiento de la implementación que utiliza MPI. Tomaremos medidas para ejecuciones que utilicen MPI junto con la paralelización local con OpenMP, para la implementación pura de MPI y por último analizaremos los efectos que tiene el esquema de iteraciones locales en el tiempo de ejecución.

6.2.3.1. Obtención de resultados

Como adelantamos en la Sección 4.4, utilizaremos el cluster del Instituto Universitario de Investigación Informática. En éste, la gestión de los trabajos la realiza Grid Engine. Tendremos que preparar un script que contenga los recursos necesarios para nuestra ejecución y utilizar el comando qsub. En el Código 6.1 tenemos la configuración de nuestro programa utilizada para esta sección.

Código 6.1: Parámetros de nuestro filtro

```
1 mpirun --bind-to none -np $NP mpi-gaussian-impulsive -i 60 --inner_iterations $inner -n $threads...
```

Los tiempos medidos varían sensiblemente puesto que no podemos controlar variables como el tráfico en la red, la temperatura de los procesadores y su frecuencia. Con la intención de paliar el ruido cada ejecución se ha repetido 100 veces, tomándose como valor final la media de todos los resultados.

En todas las pruebas filtraremos la vista axial contaminada con $\sigma = 30, \rho = 0.3$ y 60 iteraciones.

6.2.3.2. Rendimiento de la implementación MPI

En la Figura 6.5 vemos que la eficiencia de la implementación MPI se mantiene superior al 80%. Teniendo en cuenta lo baja[2] que es la carga computacional una vez utilizamos un gran número de nodos y que el esquema de comunicaciones es síncrono y en principio hay margen de mejora, son datos bastante positivos. Cada nodo únicamente ejecuta un proceso MPI y éstos se comunican cada iteración.

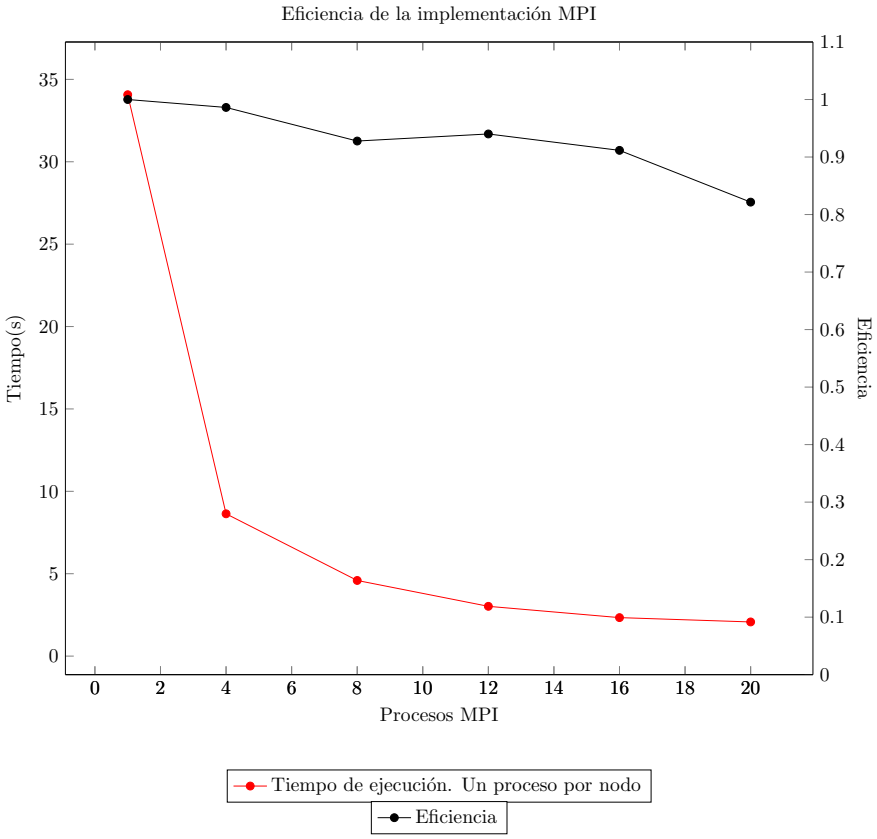


Figura 6.5: Rendimiento y eficiencia del filtro MPI. Equipo 2

[2]Recordemos que la imagen se distribuye equitativamente entre nodos.

6.2.3.3. Rendimiento de la implementación MPI + OpenMP

Como el objetivo final es el de extraer el máximo rendimiento posible, la configuración realmente importante es la que combina MPI con los 12 hilos disponibles en cada nodo. La eficiencia cae notablemente puesto que al utilizar al máximo todos los nodos, el tiempo de comunicaciones es mayor respecto al tiempo de cómputo si lo comparamos con el apartado anterior. Además, la paralelización presenta una eficiencia mejorable en los procesadores del Equipo 2 como mostramos en la Figura 6.4. En la Figura 6.6 disponemos de los resultados. La eficiencia ronda entre el 70-80% en la mayoría de casos.

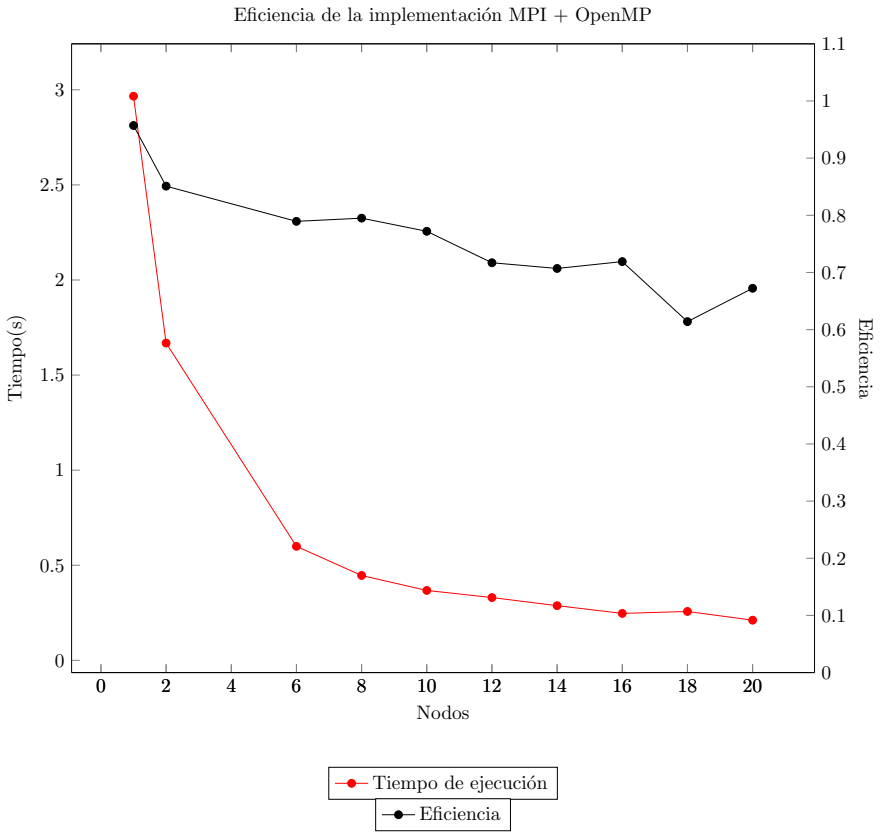


Figura 6.6: Rendimiento y eficiencia del filtro final. MPI y OpenMP. Equipo 2

Como podemos ver, conseguimos tiempos de ejecución de hasta 2 décimas de segundo con una configuración de alta carga computacional y elevada supresión de ruido. Para imágenes con bajo nivel de ruido que necesiten pocas iteraciones para llegar a su pico de PSNR obtenemos tiempos de ejecución inferiores a la décima de segundo.

6.2.3.4. Impacto del esquema de iteraciones internas

El parámetro *inner_iterations* determina el número de iteraciones que realizaremos localmente sin comunicarnos con el resto de nodos. Según el criterio seguido en la implementación, el número de iteraciones (*-n*) marcará el total de las iteraciones. Por lo tanto, dados $\{n = 60$, $inner = 10$ $\}$ realizaremos $\frac{60}{10} = 6$ comunicaciones.

Con esta consideración, en la Figura 6.7 presentamos el *speedup* en función del número de iteraciones internas respecto a la versión que comunica en cada iteración. Conseguimos unas aceleraciones cercanas al 12% respecto a la versión MPI + OpenMP que comunica tras cada iteración, lo que supone una mejora significativa. En el Apartado 6.3.2 comentaremos cómo se ve afectada la calidad del filtrado.

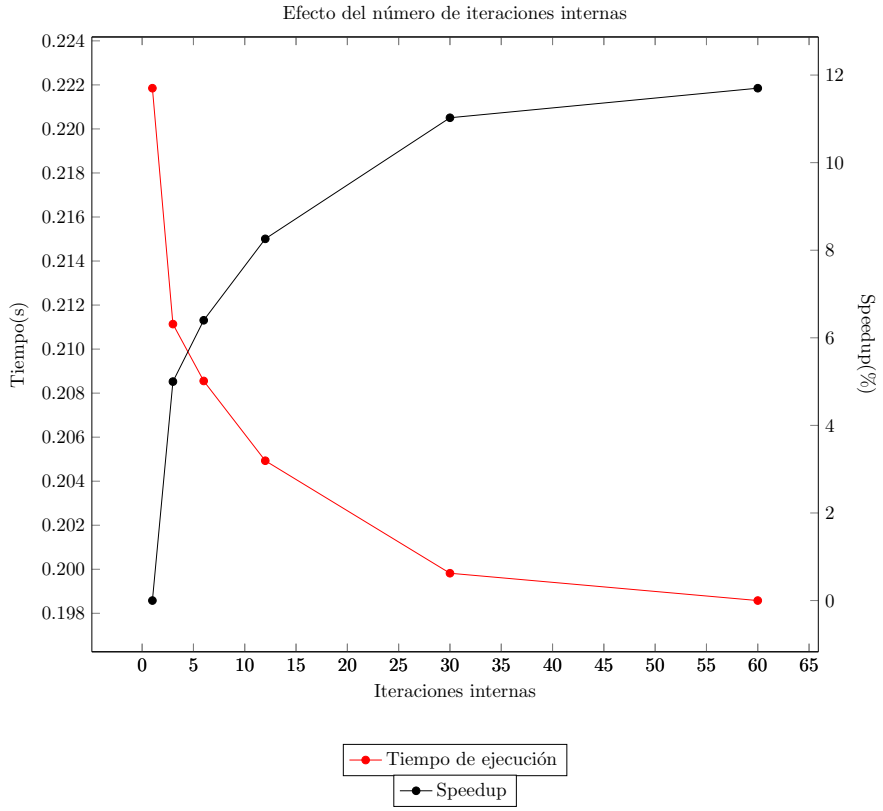


Figura 6.7: Speedup producido por la reducción de comunicaciones MPI mediante las iteraciones internas. Equipo 2

6.3. Calidad del filtrado

A continuación estudiaremos la calidad del filtrado mediante métricas numéricas y también con la inspección visual. Las distintas implementaciones producen el mismo resultado, exceptuando la versión distribuida de iteraciones internas por las razones explicadas previamente. Se ha utilizado la versión distribuida sin iteraciones internas para la extracción de los resultados, puesto que consigue la misma calidad en menor tiempo.

6.3.1. Métricas numéricas

El error medio absoluto o *MAE* (véase (Sánchez Cervantes, 2013)) se utiliza para determinar la cantidad de detalle que el filtro es capaz de preservar, y se define como:

$$\text{MAE} = \frac{1}{WH} \sum_{i=0}^{W-1} \sum_{j=0}^{H-1} |I_o(i,j) - I_f(i,j)| \tag{6.3}$$

El *Mean Square Error* o MSE (véase (Sánchez Cervantes, 2013)) se define como:

$$\text{MSE} = \frac{1}{WH} \sum_{i=0}^{W-1} \sum_{j=0}^{H-1} (I_o(i,j) - I_f(i,j))^2 \tag{6.4}$$

Siendo W la anchura de la imagen, H la altura, I_o la imagen original y I_f la imagen filtrada.

El *Peak Signal to Noise Ratio* o PSNR (véase (Sánchez Cervantes, 2013))es una métrica comúnmente utilizada para medir la capacidad de supresión de ruido. Su definición es la siguiente:

$$\text{PSNR} = 20 \log_{10} \frac{\text{MAX}_f}{\sqrt{\text{MSE}}} \tag{6.5}$$

En las Tablas 6.1 y 6.2 podemos ver los valores de MAE y PSNR para las imágenes consideradas junto a sus valores iniciales. Se consiguen altos valores de PSNR y bajos valores de MAE, lo que indica que nuestro filtro es efectivo a la hora de eliminar el ruido y a la vez conservar los detalles de la imagen.

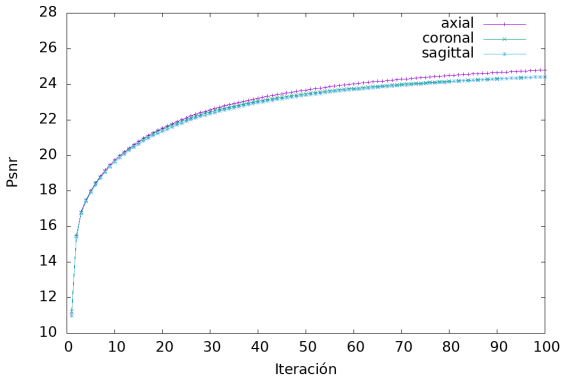
	PSNR					
	Vista axial		Vista sagital		Vista coronal	
Ruido	Ruidosa	Filtrada	Ruidosa	Filtrada	Ruidosa	Filtrada
$\sigma = 5,\ p = 0.05$	19.03	34.57	18.85	34.81	19.21	34.32
$\sigma = 10,\ p = 0.1$	16.06	32.22	15.86	32.53	16.13	32.03
$\sigma = 20,\ p = 0.2$	12.98	29.53	12.82	28.80	13.07	28.46
$\sigma = 30,\ p = 0.3$	11.17	25.80	11.02	25.24	11.25	25.06

Tabla 6.1: Valores máximos de PSNR en las imágenes contaminadas con los distintos ruidos gaussiano e impulsivos

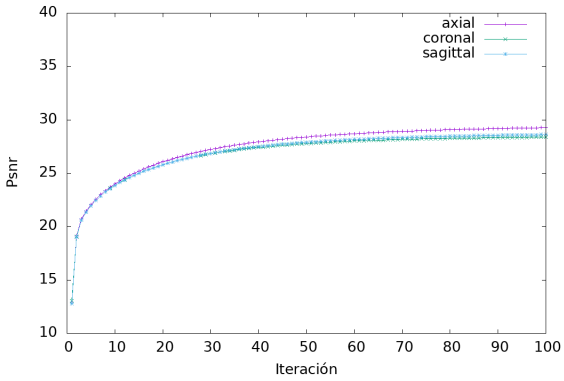
	MAE					
	Vista axial		Vista sagital		Vista coronal	
Ruido	Ruidosa	Filtrada	Ruidosa	Filtrada	Ruidosa	Filtrada
$\sigma = 5,\ p = 0.05$	13.19	4.32	8.54	2.60	9.20	2.98
$\sigma = 10,\ p = 0.1$	21.29	5.97	14.28	3.61	15.37	4.07
$\sigma = 20,\ p = 0.2$	35.65	8.26	24.87	5.91	26.64	6.47
$\sigma = 30,\ p = 0.3$	50.03	13.02	34.84	9.08	37.29	9.81

Tabla 6.2: Valores mínimos de MAE en las imágenes contaminadas con los distintos ruidos gaussiano e impulsivos

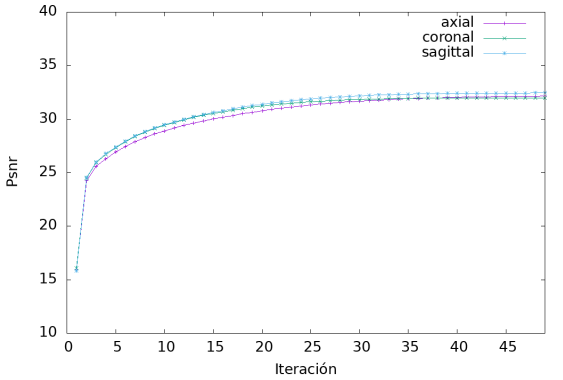
En la Figura 6.8 podemos ver la evolución que presenta el PSNR a lo largo del filtrado para todas las imágenes consideradas. En todos los escenarios la evolución inicial del PSNR es alta. Esto indica que podemos conseguir un filtrado de calidad sin demasiado tiempo de cómputo. También es cierto que existe una evolución notable cuando el número de iteraciones aumenta, especialmente en las imágenes más contaminadas.



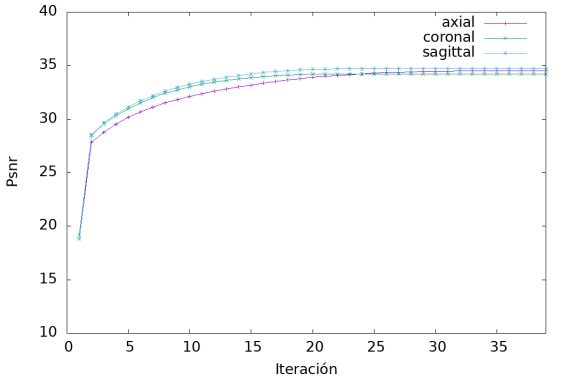
(a) Contaminadas $\sigma = 30, \rho = 0.3$



(b) Contaminadas $\sigma = 20, \rho = 0.2$



(c) Contaminadas $\sigma = 10, \rho = 0.1$



(d) Contaminadas $\sigma = 5, \rho = 0.05$

Figura 6.8: Evolución del PSNR en las tres imágenes consideradas en las distintas combinaciones de ruido estudiadas

6.3.2. Efecto de las iteraciones locales en la calidad del filtrado

Una vez determinado que la aceleración que implica el uso de las iteraciones internas es significativa debemos analizar el impacto que tiene en el resultado en cuanto a calidad de filtrado. En la Tabla 6.3 podemos comprobar que el efecto en el PSNR es mínimo, siendo su efecto inferior al 1%.

		Vista sagital	
Nº de comunicaciones	Iteraciones internas	$\sigma = 20,\ p = 0.2$	$\sigma = 30,\ p = 0.3$
60	1	28.731142	24.051815
30	2	28.729480	24.048405
12	5	28.719965	24.041835
6	10	28.706908	24.032427
4	15	28.694410	24.023486
2	30	28.667692	24.004453
1	60	28.636477	23.971676

Tabla 6.3: Evolución del PSNR respecto a las iteraciones internas

Aunque las métricas numéricas no se vean afectadas debemos asegurarnos de que no aparezcan artefactos en la imagen. En la Figura 6.9 podemos ver que este es el caso.

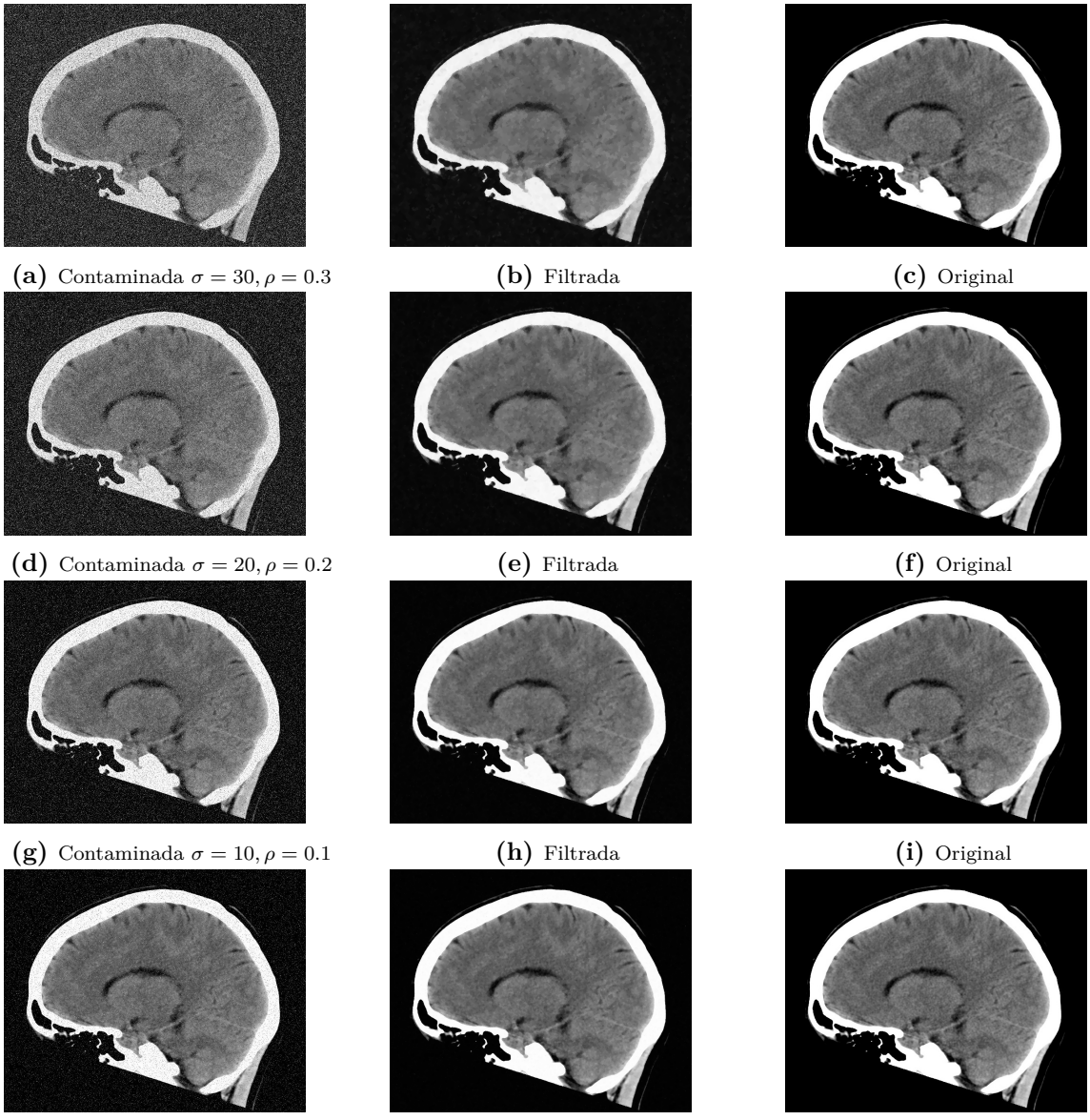
(a) Resultado con el máximo de iteraciones internas (máximo rendimiento)

(b) Resultado sin comunicaciones internas (máxima calidad)

Figura 6.9: Comparativa visual del filtro con y sin iteraciones internas. Vista sagital $\{\sigma = 20, \rho = 0.2\}$, 60 iteraciones

6.3.3. Resultados visuales

En las Figuras 6.10, 6.11 y 6.12 podemos ver los resultados visuales correspondientes a las Tablas 6.1 y 6.2. Para todas las combinaciones de ruido mostramos la imagen contaminada, la imagen filtrada y la imagen original. Tal y como indicaban las métricas, podemos comprobar que la calidad de filtrado es notable.



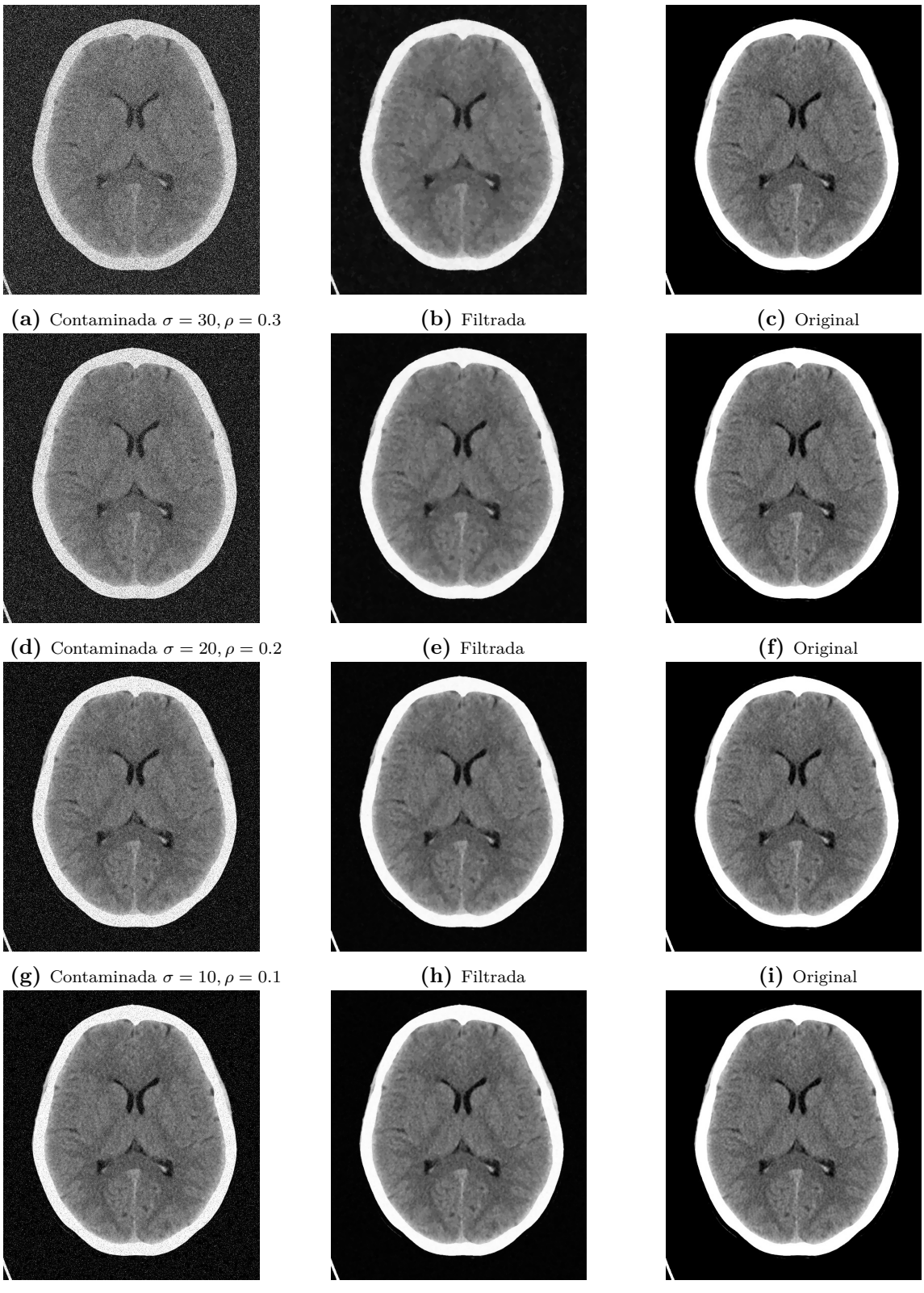
(a) Contaminada $\sigma = 30, \rho = 0.3$ **(b)** Filtrada **(c)** Original

(d) Contaminada $\sigma = 20, \rho = 0.2$ **(e)** Filtrada **(f)** Original

(g) Contaminada $\sigma = 10, \rho = 0.1$ **(h)** Filtrada **(i)** Original

(j) Contaminada $\sigma = 5, \rho = 0.05$ **(k)** Filtrada **(l)** Original

Figura 6.10: Vista Sagital. Imágenes contaminadas, filtradas y originales



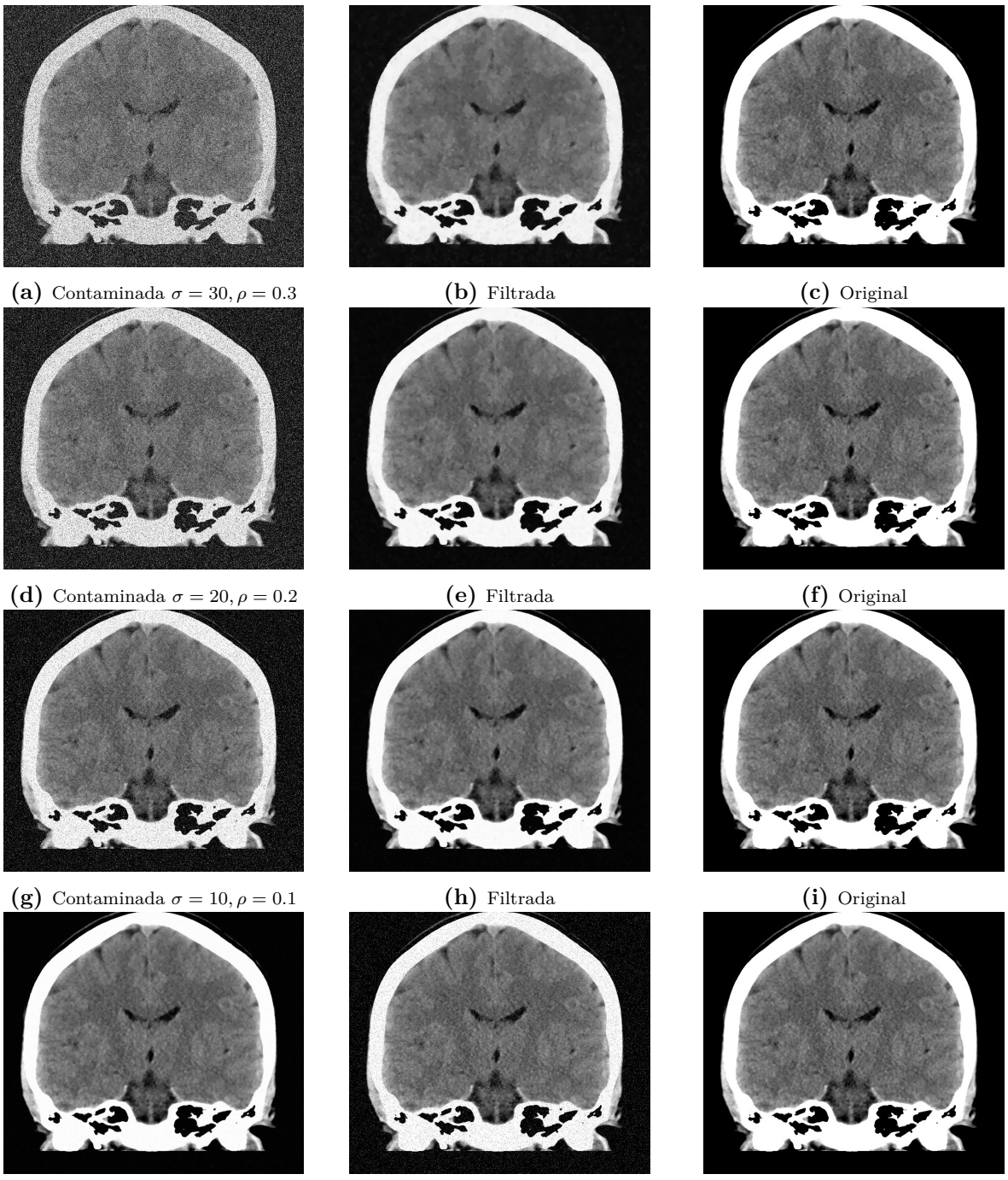
(a) Contaminada $\sigma = 30, \rho = 0.3$ **(b)** Filtrada **(c)** Original

(d) Contaminada $\sigma = 20, \rho = 0.2$ **(e)** Filtrada **(f)** Original

(g) Contaminada $\sigma = 10, \rho = 0.1$ **(h)** Filtrada **(i)** Original

(j) Contaminada $\sigma = 5, \rho = 0.05$ **(k)** Filtrada **(l)** Original

Figura 6.11: Vista Axial. Imágenes contaminadas, filtradas y originales



(a) Contaminada $\sigma = 30, \rho = 0.3$ **(b)** Filtrada **(c)** Original

(d) Contaminada $\sigma = 20, \rho = 0.2$ **(e)** Filtrada **(f)** Original

(g) Contaminada $\sigma = 10, \rho = 0.1$ **(h)** Filtrada **(i)** Original

(j) Contaminada $\sigma = 5, \rho = 0.05$ **(k)** Filtrada **(l)** Original

Figura 6.12: Vista Coronal. Imágenes contaminadas, filtradas y originales

7. Conclusiones

En el presente trabajo hemos implementado un filtro de ruido impulsivo-gausiano pensado para imágenes CT. A partir de una solución secuencial con un tiempo de cómputo muy elevado hemos obtenido implementaciones paralelas capaces de procesar una imagen en menos de un segundo. Para ello hemos utilizado herramientas propias del cómputo de altas prestaciones como OpenMP y MPI.

Los experimentos muestran que las implementaciones paralelas y distribuidas obtienen un speedup significativo, lo que resulta en tiempos computacionales reducidos que las hacen apropiadas para el procesamiento de imágenes médicas.

Como línea de trabajo futuro se plantea el analizar un posible esquema de comunicaciones asíncrono. En la actualidad las comunicaciones bloquean, limitando la capacidad de filtrado innecesariamente, pues únicamente requerimos la comunicación entre nodos para filtrar ciertas partes de la imagen y no toda. Adicionalmente, la implementación del algoritmo para procesadores gráficos de próposito general es otra opción muy prometedora.

Bibliografía

Aglave, J., Deacon, W., y Boqun Feng, e. a. (2019). *Who's afraid of a big bad optimizing compiler?* Descargado de https://lwn.net/Articles/793253/

Arnal, J., Pérez, J. B., y Vidal, V. (2020). A parallel fuzzy method to reduce mixed gaussian-impulsive noise in CT medical images. En Y. Liu, L. Wang, L. Zhao, y Z. Yu (Eds.), *Advances in natural computation, fuzzy systems and knowledge discovery* (pp. 975–982). Springer International Publishing.

Arnal, J., y Súcar, L. (s.f.). Hybrid filter based on fuzzy techniques for mixed noise reduction in color images. *Applied Sciences*, 2080–2095.

Camarena, J.-G., Gregori, V., Morillas, S., y Sapena, A. (2008). Fast detection and removal of impulsive noise using peer groups and fuzzy metrics. *J. Vis. Commun. Image Represent.*, *19*, 20-29.

Camarena, J.-G., Gregori, V., Morillas, S., y Sapena, A. (2013). A simple fuzzy method to remove mixed gaussian-impulsive noise from color images. *IEEE Transactions on Fuzzy Systems*, *21*(5), 971-978. doi: 10.1109/TFUZZ.2012.2234754

Chi, J., Zhang, Y., Yu, X., Wang, Y., y Wu, C. (2019, 07). Computed tomography (CT) image quality enhancement via a uniform framework integrating noise estimation and super-resolution networks. *Sensors*, *19*, 3348. doi: 10.3390/s19153348

Dabov, K., Foi, A., Katkovnik, V., y Egiazarian, K. (2007a, septiembre). Color image denoising via sparse 3d collaborative filtering with grouping constraint in luminance-chrominance space. En *2007 IEEE international conference on image processing*. IEEE. doi: 10.1109/icip.2007.4378954

Dabov, K., Foi, A., Katkovnik, V., y Egiazarian, K. (2007b, agosto). Image denoising by sparse 3-d transform-domain collaborative filtering. *IEEE Transactions on Image Processing*, *16*(8), 2080–2095. doi: 10.1109/tip.2007.901238

Dagum, L., y Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, *5*(1), 46–55.

Driankov, D., Hellendoorn, H., Ljung, L., Palm, R., Reinfrank, M., Palm, R., … Ollero, A. (1996). *An introduction to fuzzy control*. Springer. Descargado de https://books.google.com.cu/books?id=T1P1XeyNvl4C

Forum, M. P. (1994). *MPI: A message-passing interface standard* (Inf. Téc.). USA.

Garnett, R., Huegerich, T., Chui, C., y He, W. (2005). A universal noise removal algorithm with an impulse detector. *IEEE Transactions on Image Processing*, *14*(11), 1747-1754. doi: 10.1109/TIP.2005.857261

Gonzalez, R., y Woods, R. (2008). *Digital image processing*. Pearson/Prentice Hall. Descargado de https://books.google.es/books?id=8uGOnjRGEzoC

Gravel, P., Beaudoin, G., y De Guise, J. (2004). A method for modeling noise in medical images. *IEEE Transactions on Medical Imaging*, *23*(10), 1221-1232. doi: 10.1109/TMI.2004.832656

Lei, T., y Sewchand, W. (1992). Statistical approach to x-ray CT imaging and its applications in image analysis. II. a new stochastic model-based image segmentation technique for x-ray CT image. *IEEE transactions on medical imaging*, *11 1*, 62-9.

Lu, H., Li, X., Hsiao, I.-T., y Liang, Z. (2002). Analytical noise treatment for low-dose CT projection data by penalized weighted least-square smoothing in the K-L domain. En L. E. Antonuk y M. J. Yaffe (Eds.), *Medical imaging 2002: Physics of medical imaging* (Vol. 4682, pp. 146 – 152). SPIE. Descargado de https://doi.org/10.1117/12.465552

Morillas Gómez, S. (2008). *Fuzzy metrics and fuzzy logic for colour image filtering* (Tesis Doctoral no publicada). Universitat Politècnica de València.

Paolo Bonzini. (2021). *An introduction to lockless algorithms.* Descargado de https://lwn.net/Articles/844224/

Passino, K., y Yurkovich, S. (1998). *Fuzzy control.* Addison-Wesley. Descargado de https://books.google.es/books?id=7eNSAAAAMAAJ

Ristov, S., Prodan, R., Gusev, M., y Skala, K. (2016). Superlinear speedup in hpc systems: Why and when? En *2016 federated conference on computer science and information systems (fedcsis)* (p. 889-898).

Sánchez Cervantes, M. G. (2013). *Algoritmos de detección y filtrado de imágenes para arquitecturas multicore y manycore* (Tesis Doctoral no publicada). Universitat Politècnica de València.

Schulte, S., Huysmans, B., Pižurica, A., Kerre, E. E., y Philips, W. (2006). A new fuzzy-based wavelet shrinkage image denoising technique. En *Advanced concepts for intelligent vision systems* (pp. 12–23). Springer Berlin Heidelberg. doi: 10.1007/11864349_2

Smolka, B. (2010, 04). Peer group switching filter for impulse noise reduction incolor images. *Pattern Recognition Letters*, *31*, 484-495. doi: 10.1016/j.patrec.2009.09.012

Sorin, D. J., Hill, M. D., y Wood, D. A. (2011). *A primer on memory consistency and cache coherence* (1st ed.). Morgan & Claypool Publishers.

Tomasi, C., y Manduchi, R. (1998). Bilateral filtering for gray and color images. En *Sixth international conference on computer vision (IEEE cat. no.98ch36271)* (p. 839-846). doi: 10.1109/ICCV.1998.710815

Toprak, A., y Güler, u. (2007, jul). Impulse noise reduction in medical images with the use of switch mode fuzzy adaptive median filter. *Digit. Signal Process.*, *17*(4), 711–723. doi: 10.1016/j.dsp.2006.11.008

A. Cálculo analítico del centro de gravedad para la obtención de los pesos de cada pixel

A continuación vamos a detallar el cálculo de la defuzzificación mencionado en la Sección 2 y utilizado en el Código 5.7.

Como mencionamos anteriormente, la ecuación 2.9 es el cálculo del centro de gravedad para la Figura 2.5, por lo que es equivalente a la ecuación 2.10. Para llevar a cabo el cálculo debemos tener en cuenta que el centro de gravedad de un triángulo rectángulo se encuentra a un tercio de la base partiendo desde el ángulo recto, como podemos ver en la Figura A.1.

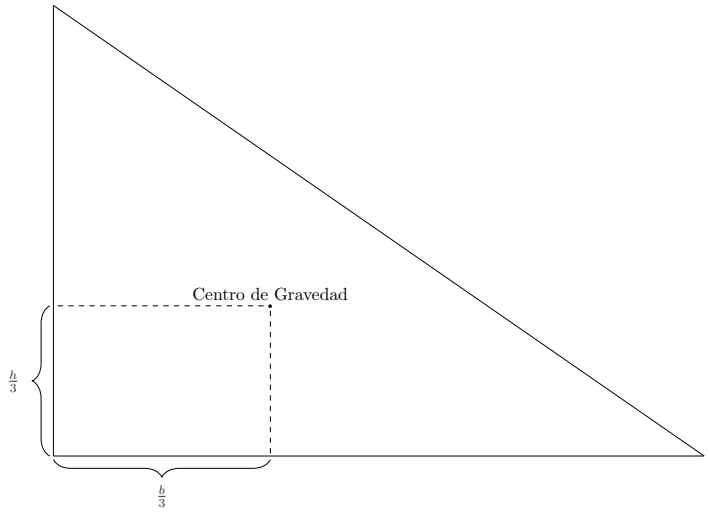


Figura A.1: Centro de gravedad de un triángulo rectángulo

Con esto en mente lo único que hay que hacer es considerar cada polígono descrito en la Figura 2.5 uno a uno e ir calculando sus centros de gravedad y sus áreas correspondientes.

A.1. Diagramas

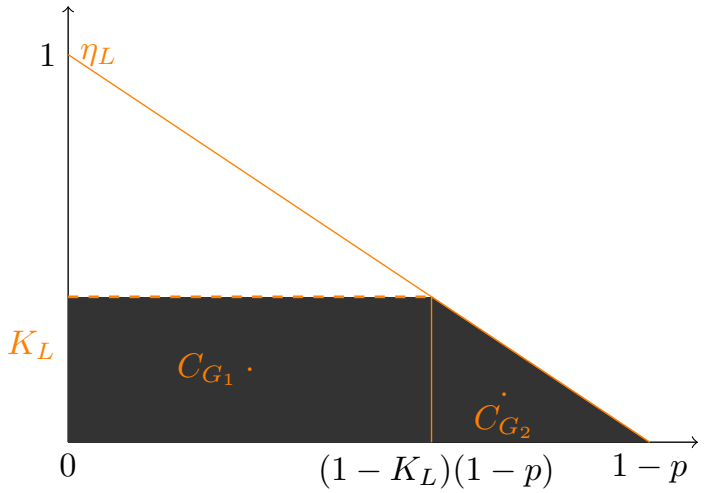


Figura A.2: Triángulo correspondiente a η_L

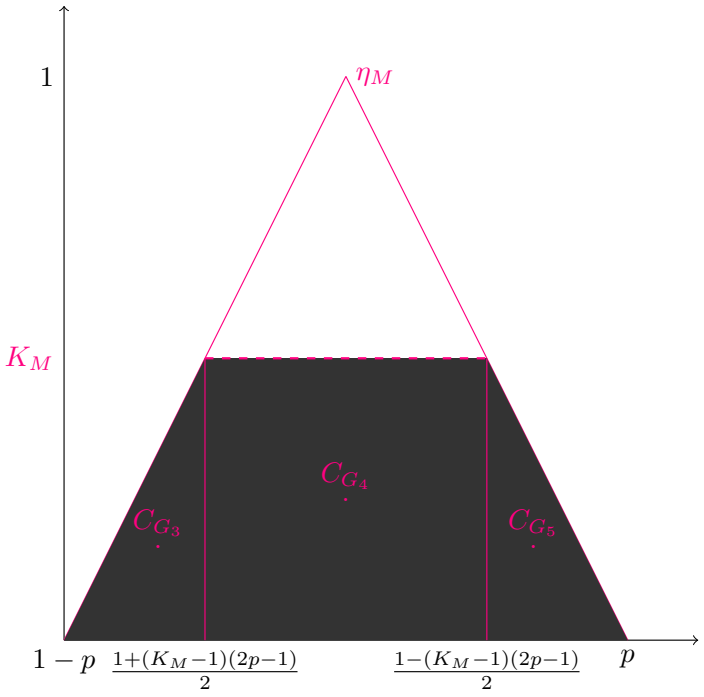


Figura A.3: Triángulo correspondiente a η_M

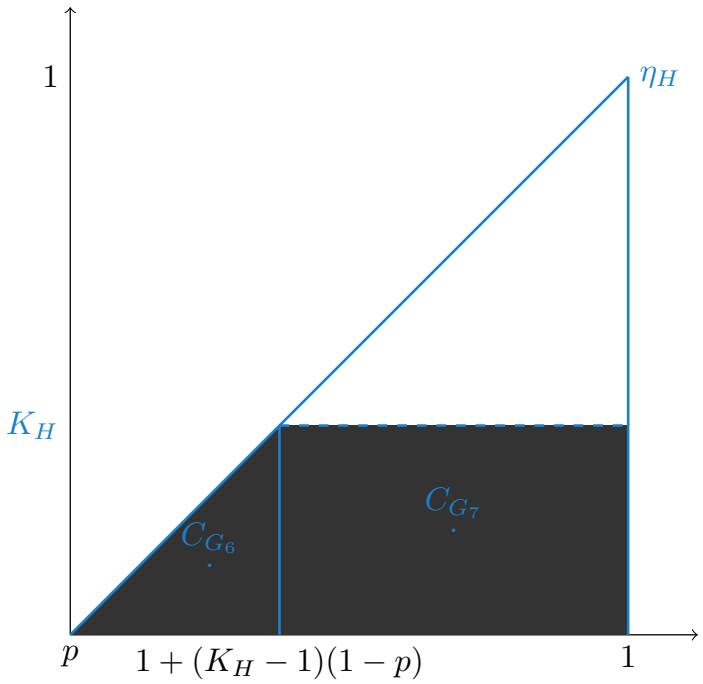


Figura A.4: Triángulo correspondiente a η_H

A.2. Centros de gravedad y áreas

$$A_1 = K_L(1-K_L)(1-p) \qquad C_{G_{1_x}} = \frac{(1-K_L)}{2}(1-p)$$

$$A_2 = \frac{K_L^2}{2}(1-p) \qquad C_{G_{2_x}} = \frac{3-2K_L}{3}(1-p)$$

$$A_3 + A_4 + A_5 = \frac{K_M}{2}(2-K_M)(2p-1) \qquad C_{G_{3,4,5_x}} = \frac{1}{2}$$

$$A_6 = (1-p)\frac{K_H^2}{2} \qquad C_{G_{6_x}} = \frac{1}{3}(3p+2K_H-2pK_H)$$

$$A_7 = K_H(1-K_H)(1-p) \qquad C_{G_{7_x}} = \frac{(1-K_H)}{2}(1-p)$$