Escuela
Politécnica
Superior

Algoritmos paralelos para la reducción de ruido speckle en imágenes médicas.

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:

Adrián Jiménez González

Tutor:

Josep Arnal García

Universitat d'Alacant
Universidad de Alicante

Algoritmos paralelos para la reducción de ruido speckle en imágenes médicas.

Paralelización del método SRAD haciendo uso de OpenMP y MPI

Autor

Adrián Jiménez González

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

El ruido *speckle* aparece en diversos tipos de imágenes médicas. Por ejemplo, en las imágenes de Ultra Sonidos (US) y en particular en las imágenes de Tomografía de Coherencia Óptica (OCT). Estas imágenes se degradan severamente debido a la presencia de este tipo de ruido, lo que causa dificultad para interpretar pequeños detalles y características morfológicas de baja intensidad requeridas para el diagnóstico clínico. Las imágenes de US se obtienen a partir de la lectura del eco de una onda transmitida por el medio en el que se encuentra. Esta onda interactúa con los tejidos del cuerpo devolviendo parte de ella al transductor y siendo captada para la interpretación de la imagen. Este tipo de ruido aparece de forma granular en la imagen disminuyendo la resolución de los cambios en la imagen.

Asimismo, el ruido *speckle* es inherente a las imágenes de US, y puede tener un efecto negativo en la interpretación de dichas imágenes y en las tareas de diagnóstico. El ruido *speckle* degrada significativamente la calidad de la imagen de US y complica las decisiones de diagnóstico para discriminar detalles finos en dichas imágenes de ultrasonido. En consecuencia, el filtrado del ruido speckle es un paso importante en la generación de este tipo de imágenes médicas y permite mejorar los procesos de detección de lesiones o la clasificación del tejido.

El tratamiento de imágenes de gran resolución y el filtrado de imágenes en tiempo real, nos conduce a requerimientos computacionales más altos. En esta investigación se

diseñarán e implementarán métodos de filtrado de ruido *speckle* mediante el filtro Speckle Reduction Anistropic Diffusion (SRAD) haciendo uso de técnicas de computación de altas prestaciones para tratar imágenes médicas reduciendo el tiempo de filtrado. SRAD se trata de un filtro guiado iterativo con buenos resultados para las imágenes OCT y US.

El proyecto se va a centrar en la implementación y obtención de resultados del método de filtrado SRAD mediante distintas técnicas de paralelización. Se ha elegido este método por sus buenos resultados eliminando el ruido *speckle* o moteado (en su traducción al español) ayudando a la distinción de los distintos elementos existentes en la imagen. El objetivo de la paralelización es mejorar los tiempos del procesado de la imagen para obtener ejecuciones en tiempo real.

Agradecimientos

Me gustaría agradecer a mis padres, mi hermano y mi novia por apoyarme durante toda mi etapa universitaria, sobretodo durante este último año con el desarrollo de este trabajo. Su motivación me ha ayudado a llevar de mejor manera la labor realizada. Han sabido aguantar mis altibajos y soportarme en los momentos de crispación. Gracias por vuestra comprensión y paciencia. También agradecer a los distintos profesores que me han ayudado a llegar hasta donde estoy ahora mismo. Mi camino por la universidad ha estado acompañado de mis compañeros de clase y mis amigos a quien quiero hacer mención, por haber hecho este trayecto más llevadero y ameno.

Este trabajo ha sido supervisado y tutelado por Josep Arnal García, a quien le transmito mi más sincero agradecimiento. Con él realicé unas prácticas a partir de una asignatura, con la que encontré lo que más me gusta y con lo que he decidido que voy a enfocar mi futura trayectoria profesional y especializarme más en la materia. Su ayuda y atencion en el trabajo ha sido fundamental. Desde dudas insignificantes hasta errores imperceptibles que me llevaban por el camino de la amargura. En las asignaturas que me ha impartido clase ha sabido transmitir sus conocimientos de la mejor manera posible, siempre estando atento a las necesidades de sus alumnos. Por ello, solo puedo decir gracias.

Gracias a todas las personas que han pasado por mi vida y han dejado su granito de

arena para que yo sea quien soy a día de hoy, y haya llegado hasta donde estoy.

En dedicatoria a mis abuelos,

que allá donde estéis, estéis orgullosos.

Sin entrenamiento, no existe el conocimiento.

Sin conocimiento, no existe la confianza.

Sin confianza, la victoria no existe.

Cayo Julio Cesar.

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

El ruido *speckle* o también conocido como ruido moteado, es un tipo de ruido implícito en las imágenes US. Estas imágenes se obtienen a partir del retorno de las ondas de sonido que son percibidas por un transductor. Estas ondas interactúan por el medio en el que se desplazan, por lo que al ser usadas en un cuerpo de un ser vivo, interactuará con los distintos tejidos y también con la sangre, creando ese ruido característico de las imágenes US.

El ruido speckle, de naturaleza multiplicativa, se puede modelar como [1]

$$i(s) = o(s)n(s) \tag{1.1}$$

donde i, o, y n denotan las intensidades de la imagen observada, la imagen original y el ruido speckle, respectivamente.

Este ruido deteriora la calidad de la imagen, llegando a perder características importantes de la misma. Por ello, es importante retirar este ruido de la imagen para poder mejorar la interpretación de la misma. Es cierto que hay ocasiones que la retirada del ruido puede ser contraproducente, ya que el filtrado de la imagen puede llegar a eliminar alguna zona importante y perder información que nos aporta [2].

Además, el granulado del ruido *speckle* reduce el contraste entre las distintas zonas, haciendo que las aristas de la imagen se difuminen y no sea posible hacer uso de

distintos algoritmos automatizados como extracción de características, segmentación, visualización 3D, entre otras. Por lo que el filtrado digital de las imágenes con ruido *speckle* debe mejorar la calidad y el diagnóstico de la imagen conservando las características importantes de la imagen. Entre las características mencionadas encontramos zonas de alto contraste entre órganos colindantes, elementos del cuerpo con tamaños similares a las partículas del ruido moteado y zonas de ligero contraste que permiten al especialista médico detectar deformidades o irregularidades como tumores, quistes o roturas de tejidos.

Con el fin de poder eliminar o reducir el ruido *speckle*, se han desarrollado varios métodos, algunos de los cuales vamos a recopilar a continuación. La primera aproximación que vamos a mencionar es la composición espacial o *compounding*. En ella, para reducir el ruido se combinan imágenes correlativas parcialmente o imágenes no correlativas de la misma región de interés obtenidas mediante transductores con distinta localización espacial. Behar et al [3], proponen en su artículo un metodo de composición que combina imágenes del mismo espacio de interés mejorando la calidad de la imagen resultante sin aumentar el tiempo de obtención de las imágenes.

Las técnicas de composición espacial son muy costosas, por lo que se han propuesto muchos filtros de post-procesado. Entre los que encontramos Adaptive Weighted Median Filter (AWMF) [4], el filtro de Lee [5], Frost [6], Kuan [7] y Gamma MAP [8].

El filtro AWMF es seguramente el filtro más sencillo e intuitivo de los mencionados anteriormente. Se trata de ajustar los coeficientes de peso y características de suavizado de cada píxel acorde con las estadísticas del alrededor de cada píxel de la imagen. De esta manera obtenemos una reducción del ruido *speckle* sin perder considerablemente aristas y otras características de la imagen en comparación con la imagen original.

El fitro de Lee fue diseñado para minimizar el ruido aditivo blanco y ruido multiplicativo blanco además de mejorar el contraste. Se basa en la Minimización de la Media Cuadrada del Error (MMSE). Junto al filtro de Kuan, que tiene la misma base, obtienen una imagen computando la combinación lineal del centro de cada píxel en una ventana con el intensidad media de toda la imagen. Los filtros obtienen un balance entre la mejora del ruido en zonas homogéneas y las zonas donde detecta aristas. El filtro de Frost también es usado para reducir el ruido *speckle* y además mantiene un buen balance, al igual que los filtros de Lee y Kuan. La diferencia es que el balance se obtiene mediante un filtro de forma exponencial, pudiendo variar de un filtro de media básico a un filtro de identificación, es decir, tiene una base adaptativa. Este filtro responde ante el coeficiente de variación de una Región de Interés o *Region Of Interest* (ROI). Dependiendo del resultado del coeficiente de variación el comportamiento del filtro se asemejará más a un filtro de media básico cuando el coeficiente de variación sea bajo, o intentará mantener los contrastes de la imagen y preservar las aristas presentes en caso de que el coeficiente de variación sea elevado.

El último de los filtros mencionados es Gamma MAP. Este fitro obtiene su nombre de Gamma máximo *a posteriori* (MAP). Este filtro funciona haciendo uso del coeficiente de variación. Se establecen dos umbrales en el que si el coeficiente obtenido es menor al umbral bajo se comportará únicamente realizando la media. Mientras que si supera el umbral superior impuesto se comporta estrictamente como un filtro de identificación. Cuando el coeficiente obtiene un valor entre los dos umbrales obtiene un balance entre realizar la media y detectar aristas, es decir, tiene un comportamiento similar a los filtros de Lee y de Frost.

A pesar de que los filtros mencionados son calificados como «preservadores de aristas» y «preservadores de características», existen importantes limitaciones en las aproximaciones del filtrado de las imágenes. Todos los filtros son sensibles al tamaño y forma

de la ROI seleccionada. Teniendo comportamientos distintos en función del tamaño de la ROI en comparación con el tamaño de la imagen de entrada. En caso de que la ventana donde calculamos el coeficiente de varianza sea muy grande en proporción con la imagen, obtendremos un suavizado excesivo y se obtendrá un efecto de *blur* en las aristas, es decir, se difuminarán perdiendo el contraste entre las distintas áreas de alto contraste. Por otro lado, una ventana pequeña disminuirá la capacidad de reducción de filtro y se dejará motas en la imagen final. Otra limitación que encontramos, es que la ventana con la que calculamos la variación no puede contener zonas de alto contraste o aristas. Esto se debe a que en caso de encontrarlos dentro de la ROI, el coeficiente de variación será alto. Este coeficiente de variación se traduce en que al aplicar el filtro, las zonas cercanas a las aristas mantendrán el ruido y no se eliminará. Por lo que podemos decir que el filtrado queda inhibido. La tercera limitación que encontramos es que los filtros no son direccionales. Esto quiere decir que en los bordes no se suaviza el ruido mediante el alisado en direcciones perpendiculares, sino que se fomenta el alisado en direcciones perpendiculares a la arista. Y por último, encontramos que los umbrales y parámetros que se le introducen a los filtros, aunque estén respaldados por datos estadísticos, no dejan de ser mejoras *ad hoc* que demuestran las limitaciones de las aproximaciones de algoritmos basados en ROI.

Los filtros de Lee y Frost son los más usados de cara a la restauración de imágenes con ruido *speckle* debido a su fácil implementación y control del filtro. Por ello, se han realizado muchas mejoras a los filtros clásicos. Una de las mejoras que se propusieron [9] para mejorar los filtros de Lee y Frost fue la de realizar una clasificación de los píxeles previa al filtrado, con el fin de aplicar distintos procesados a cada clase en específico. Otro filtro muy extendido y usado es SRAD [10], filtro que elimina eficientemente la presencia de ruido y que mantiene información esencial. Este filtro es el que, a lo largo de la presente memoria, vamos a explicar con detenimiento, vamos implementar y obtener resultados. El filtro SRAD utiliza Partial Differential Equation (PDE) de segundo

grado. En su proceso iterativo, utiliza este sistema de ecuaciones diferenciales como detector de bordes y aristas en la imagen, información esencial, en zonas propensas a bordes. Como acabamos de comentar, se trata de un proceso iterativo, por lo que es posible que algunas características de bordes se puedan perder, siendo este uno de los mayores inconvenientes del filtro SRAD. SRAD es adaptativo, por lo que no utiliza umbrales para alterar el comportamiento del filtro con las zonas homogéneas o las zonas de alto contraste. Este filtro fue propuesto para la aplicación en imágenes de radar e imágenes de ultrasonidos médicas en las que por ser dependientes de la entrada de la señal, el ruido multiplicativo correlativo o ruido *speckle* esta presente en las imágenes que se obtienen.

2 Método Speckle Reducing Anisotropic Diffusion

Para describir el método SRAD introducido en [10], vamos a considerar una imagen en escala de grises $I_0(x, y)$ definida en un dominio Ω. La imagen generada por el filtro SRAD $I(x, y, t)$, se obtiene como solución de la siguiente ecuación en derivadas parciales PDE (2.1). De esta forma, la ecuación (2.1) se suele conocer como SRAD PDE.

$$\begin{cases} \partial I(x,y;t)/\partial t = div[c(q)\nabla I(x,y;t)] \\ I(x,y;0) = I_0(x,y), (\partial I(x,y;t)/\partial \overrightarrow{n})|_{\partial\Omega} = 0 \end{cases} \tag{2.1}$$

donde $\partial\Omega$ son la frontera de Ω, $\overrightarrow{n}$ es el vector normal de $\partial\Omega$, y

$$c(q) = \frac{1}{1 + [q^2(x,y;t) - q_0^2(t)]/[q_0^2(t)(1 + q_0^2(t))]} \tag{2.2}$$

o

$$c(q) = exp\{-[q^2(x,y;t) - q_0^2(t)]/[q_0^2(t)(1 + q_0^2(t))]\} \tag{2.3}$$

En las dos ecuaciones mostradas anteriormente (2.2, 2.3), el valor de $q(x, y; t)$ es el coeficiente instantáneo de la variación determinada por la ecuación (2.4)

$$q(x,y;t) = \sqrt{\frac{(1/2)(|\nabla I|/I)^2 - (1/4^2)(\nabla^2 I/I)^2}{[1 + (1/4)(\nabla^2 I/I)]^2}} \tag{2.4}$$

y $q_0(t)$ es la función de escala del *speckle*.

En el método SRAD, el coeficiente instantáneo de variación (2.4) actúa como un detector de aristas en las imágenes con ruido *speckle*. Esta función obtiene valores altos en las aristas o regiones de alto contraste y valores bajos en zonas homogéneas. La función de escala del *speckle* $q_0(t)$ se utiliza para controlar eficazmente el suavizado aplicado a la imagen por SRAD. El valor se obtiene con

$$q_0(t) = \frac{\sqrt{var[z(t)]}}{\overline{z(t)}} \tag{2.5}$$

siendo $var[z(t)]$ la varianza de intensidad y $\overline{z(t)}$ la media sobre un área homogénea en t, también mencionada en la memoria como ROI.

Para poder resolver la ecuación diferencial (2.1) numéricamente se puede usar el método iterativo de Jacobi. Asumiendo un salto de tiempo suficientemente pequeño del tamaño de Δt y un salto espacial suficientemente pequeño del tamaño de h en los ejes x e y, discretizamos el tiempo y las coordenadas espaciales de la forma siguiente:

$$\begin{aligned} t &= n\Delta t, \quad n = 0, 1, 2, ... \\ x &= ih, \quad i = 0, 1, 2, ..., M-1 \\ y &= jh, \quad j = 0, 1, 2, ..., N-1 \end{aligned} \tag{2.6}$$

donde $Mh \times Nh$ es el tamaño de la imagen de entrada.

Sea $I^n_{i,j} = I(ih, jh, n\Delta t)$. Para calcular el lado derecho de la SRAD PDE (2.1) se usa una aproximación de tres pasos [11]. En el primer paso, calculamos las aproximaciones de las derivadas y la aproximación del laplaciano como:

$$\nabla_R I^n_{i,j} = \left[\frac{I^n_{i+1,j} - I^n_{i,j}}{h}, \frac{I^n_{i,j+1} - I^n_{i,j}}{h} \right] \tag{2.7}$$

$$\nabla_L I^n_{i,j} = \left[\frac{I^n_{i,j} - I^n_{i-1,j}}{h}, \frac{I^n_{i,j} - I^n_{i,j-1}}{h} \right] \tag{2.8}$$

$$\nabla^2 I^n_{i,j} = \frac{I^n_{i+1,j} + I^n_{i-1,j} + I^n_{i,j+1} + I^n_{i,j-1} - 4I^n_{i,j}}{h^2} \tag{2.9}$$

con las condiciones del contorno simétricas:

$$I^n_{-1,j} = I^n_{0,j}, \quad I^n_{M,j} = I^n_{M-1,j}, \quad j = 0, 1, 2, ..., N-1 \tag{2.10}$$

$$I^n_{i,-1} = I^n_{i,0}, \quad I^n_{i,N} = I^n_{i,N-1}, \quad i = 0, 1, 2, ..., M-1. \tag{2.11}$$

El segundo paso consiste en calcular el coeficiente de difusión $c(q)$ haciendo uso de las ecuaciones mencionadas (2.2, 2.3) de la forma siguiente

$$c^n_{i,j} = c\left[q\left(\frac{1}{I^n_{i,j}} \sqrt{|\nabla_R I^n_{i,j}|^2 + |\nabla_L I^n_{i,j}|^2}, \frac{1}{I^n_{i,j}} \nabla^2 I^n_{i,j} \right) \right]. \tag{2.12}$$

En el tercer paso, se calcula la divergencia de $c(\cdot)\nabla I$, necesaria para el SRAD PDE (2.1), como

$$d^n_{i,j} = \frac{1}{h^2}[c^n_{i+1,j}(I^n_{i+1,j} - I^n_{i,j}) + c^n_{i,j}(I^n_{i-1,j} - I^n_{i,j}) + c^n_{i,j+1}(I^n_{i,j+1} - I^n_{i,j}) + c^n_{i,j}(I^n_{i,j-1} - I^n_{i,j})] \tag{2.13}$$

con las condiciones de contorno simétricas

$$d^n_{-1,j} = d^n_{0,j}, \quad d^n_{M,j} = d^n_{M-1,j}, \quad j = 0, 1, 2, ..., N-1 \tag{2.14}$$

$$d^n_{i,-1} = d^n_{i,0}, \quad d^n_{i,N} = d^n_{i,N-1}, \quad i = 0, 1, 2, ..., M-1. \tag{2.15}$$

Por último, aproximando la diferencial temporal con diferenciación directa, la apro-

ximación numérica de la ecuación diferencial (2.1) viene dada por

$$I_{i,j}^{n+1} = I_{i,j}^{n} + \frac{\Delta t}{4} d_{i,j}^{n}. \tag{2.16}$$

Esta ecuación (2.16) se le llama *función de actualización SRAD*.

3 Organización

El proyecto ha sido realizado de manera secuencial siendo dividido en distintos procesos hasta su finalización.

En primer lugar, se realizó un estudio exhaustivo de la referencia bibliográfica [10] referente al método a implementar (SRAD). Los autores del método también desarrollaron un código Matlab que sirvió para el estudio del algoritmo y el entendimiento del mismo.

A continuación, comenzó el desarrollo del código secuencial en C desde 0. En esta implementación también se desarrollaron funciones necesarias para la lectura y escritura de imágenes PGM, reserva de memoria para matrices y estructuras de datos para almacenar la imagen PGM correctamente. Una vez desarrollado el algoritmo con esta aproximación, se comprobaron los resultados comparándolos con los resultados obtenidos por el código del autor, asegurándonos del correcto funcionamiento del código.

Después, se realizó la implementación del código paralelo sobre el secuencial de C en MPI. MPI es una librería que explota el uso de varios procesadores mediante una interfaz de paso de mensajes, eficaz para clusters con conexiones entre los distintos computadores. Al igual que en la implementación anterior, se comprobaron los resultados y se contrastaron con los resultados esperados, corrigiendo errores que se obtuviesen hasta obtener el funcionamiento esperado del código y de su salida.

El siguiente paso fue el desarrollo de una implementación paralela con la librería OpenMP. Estas librerías aplican paralelización al código, tanto en C, como C++ y Fortran, en máquinas de memoria compartida. El comportamiento del código fue comprobado con las anteriores versiones disponibles para comparar los resultados de la salida.

La última versión de código desarrollada fue la implementación de código paralelo híbrido, es decir, haciendo uso de las librerías MPI+OpenMP. Esta versión de código está enfocada a clusters con nodos multicore donde los nodos están conectados en red, necesitando el paso de mensajes proporcionado por MPI, y cada nodo tiene varios cores que se puedan aprovechar de la eficacia de la memoria compartida de OpenMP.

Durante el desarrollo de cada una de las implementaciones mencionadas anteriormente, se realizaron varias versiones con distintos enfoques para los problemas que se abordaban en cada una de las situaciones que se planteaban. Se terminaron eligiendo las más eficaces haciendo uso de la métrica *speedup* sobre el mismo número de cores.

Con todas las implementaciones desarrolladas y en correcto funcionamiento, se realizaron pruebas en distintos computadores con características diferentes. Los resultados obtenidos de las ejecuciones de distintas imágenes en Cluster3 (UA), Euler (UA) y Trunks (UMH) son estudiados estadísticamente con más profundidad en la Sección 5. Estos resultados han sido obtenidos mediante una estructura fija, igual para todas las imágenes y con las condiciones necesarias para que los datos sean relevantes.

Por último, se ha realizado la redacción de la presente memoria del Trabajo de Fin de Grado. El diagrama de la Figura 3.1 sirve como síntesis, en la que se puede observar la organización secuencial del proyecto expuesta anteriormente.

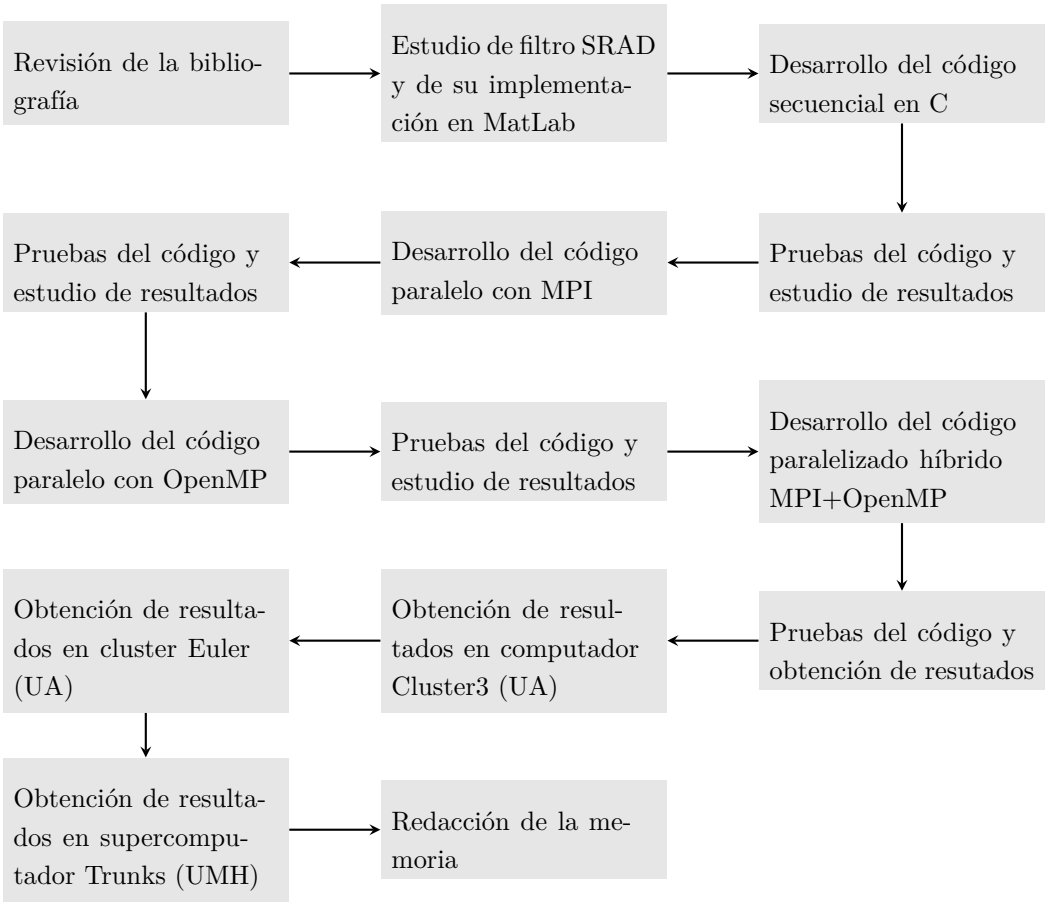


Figura 3.1: Diagrama secuencial de actividades del proyecto

4 Desarrollo

4.1 Paralelización

Para las implementaciones paralelas hemos considerado una descomposición del dominio de la imagen para una topología de anillo bidireccional de manera que cada proceso se encargará del cálculo de una porción de filas consecutivas de la imagen, tal y como indica la Figura 4.1, en la que se muestra cómo distribuimos la imagen en un ejemplo con tres nodos de computo. Esta topología es fácilmente implementable haciendo uso de MPI y OpenMP.

Para explicar como se implementó la paralelización dividimos el dominio Ω de la imagen en varios subdominios P, siendo P el número de procesadores entre los que vamos a dividir la imagen. Obteniendo así una serie de subdominios Ω_k, $k = 1, 2, ..., P$. Cada uno de estos dominios será computado por una Unidades de Procesamiento (UP), trabajando en paralelo.

Se ha elegido una descomposición del dominio por filas debido a que las matrices bidimensionales en el lenguaje de programación C se almacenan por filas. Para que la carga de trabajo entre los procesadores sea equitativa, a uno de los procesadores se le asignan $\frac{N^oFilas}{P} + mod(N^oFilas, P)$ filas y al resto de procesadores se le asignan $\frac{N^oFilas}{P}$, siendo $mod(N^oFilas, P)$ el resto de la división entera $\frac{N^oFilas}{P}$.

En la Figura 4.1 se puede observar cómo quedan los subdominios en una división de Ω en 3 UP. Cada uno de los subdominios será procesado por un procesador, trabajando todos simultáneamente.

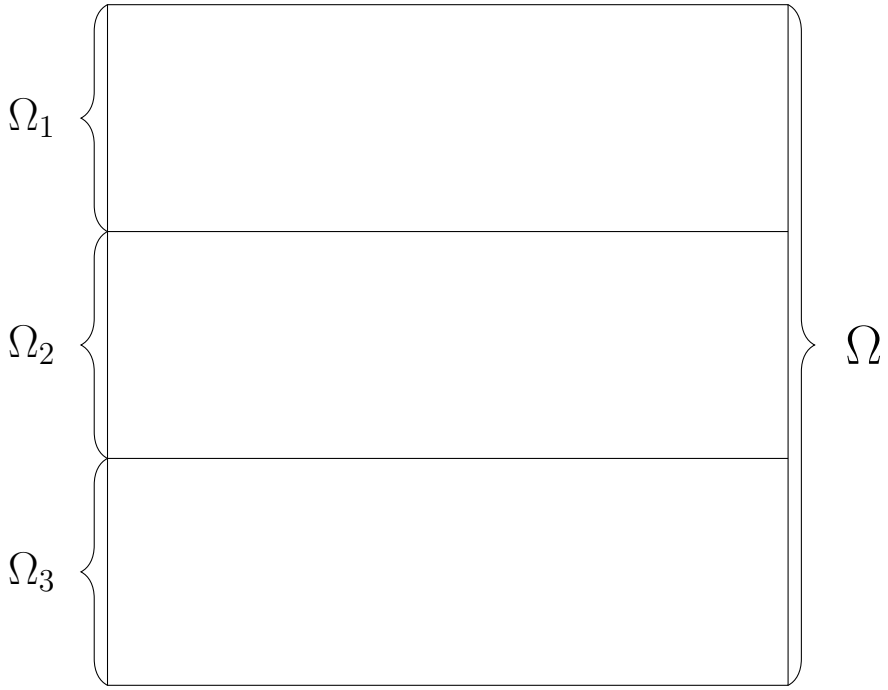


Figura 4.1: División del dominio Ω

4.2 Implementaciones

En esta sección vamos a describir las distintas implementaciones del método SRAD junto a rasgos específicos que hemos tenido que abordar para el desarrollo de cada una de las versiones.

En el desarrollo de este proyecto, se han realizado cuatro implementaciones distintas del código: la versión secuencial en C, la versión de C con MPI, la versión de C con OpenMP y la versión híbrida MPI+OpenMP.

4.2.1 Versión secuencial en C

Este código fue el primero que se desarrolló en el proyecto. Este código sirve como base para el resto de implementaciones que se han desarrollado. En esta versión, se obtiene el algoritmo funcional en C, con las funciones necesarias para el correcto funcionamiento del algoritmo.

En la Figura 4.2 podemos observar un diagrama de flujo del algoritmo implementado. En las siguientes líneas describimos más detalladamente esta implementación

El código se ha desarrollado para el trabajo con imágenes en formato PGM. Este tipo de imagen en escala de grises es relativamente sencillo de leer y escribir y nos ha permitido trabajar fácilmente con imágenes US y OCT.

En esta primera versión, se realizó la investigación y desarrollo de las funciones de lectura y escritura de ficheros de imagen PGM, junto a las estructuras de datos necesarias para el correcto manejo de los datos dentro del código. También se realizaron funciones de reserva de memoria para las matrices, de forma que todas las posiciones de la matriz sean contiguas, siendo esto necesario de cara a la implementación con MPI. Esta particularidad está explicada en el apartado de la versión MPI.

Después de desarrollar todas las funciones y estructuras necesarias, comenzamos con la implementación del algoritmo. Lo primero que encontramos es la lectura de la imagen y almacenamiento en una estructura para procesarla. A continuación, efectuamos la reserva de memoria para todas las matrices necesarias para el algoritmo con la función de reserva de memoria contigua. Hay que mencionar que la matriz de la imagen original se amplía en 2 columnas y dos filas de ceros. Este acolchado de la imagen es necesario para poder calcular la discretización del SRAD PDE (2.1) en los píxeles de los bordes de la imagen.

En la Figura 4.3 podemos observar como queda la matriz tras el acolchado. Las filas

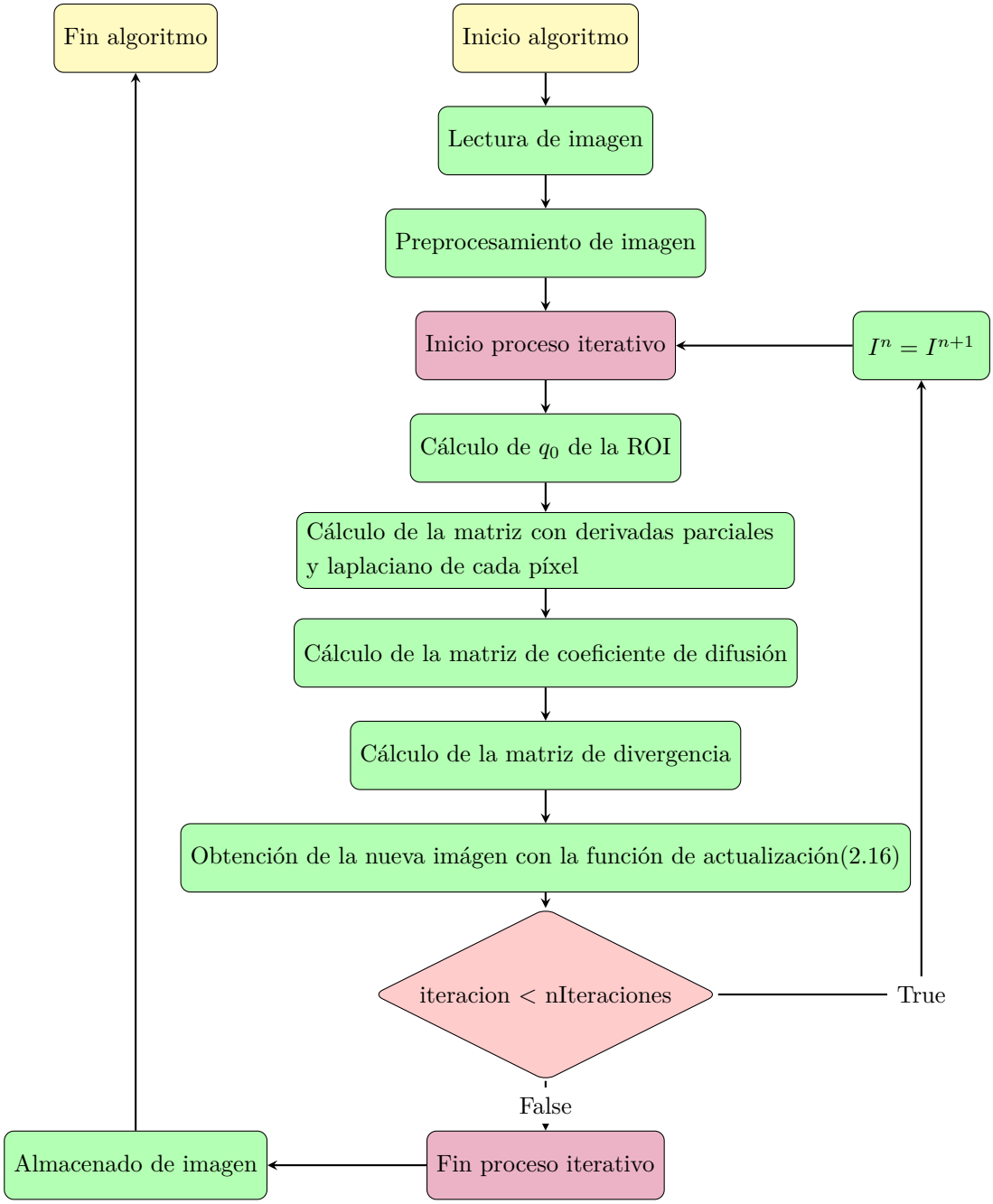


Figura 4.2: Diagrama de flujo del código SRAD

0	0	0	0	0	0	0	0
0	X	X	X	X	X	X	0
0	X	X	X	X	X	X	0
0	X	X	X	X	X	X	0
0	X	X	X	X	X	X	0
0	X	X	X	X	X	X	0
0	X	X	X	X	X	X	0
0	0	0	0	0	0	0	0

Figura 4.3: Acolchado de imagen

y columnas que se añaden como acolchado, representado con un mallado rojo, se introducen con un valor 0. Mientras que el resto de la matriz de la imagen se mantiene con su valor original.

Hay que mencionar que en la implementación se hace uso de dos matrices bidimensionales, necesarias para almacenar las imágenes, I^n y I^{n+1}, que se obtienen en dos iteraciones consecutivas del proceso iterativo, como se observa en la función de actualización (2.16).

En este proceso iterativo, en primer lugar se calcula el valor q_0 de la ROI acorde a la ecuación (2.5), siendo t la ventana de la región elegida. Esta ventana se elige por parámetros. Se le indica una posición de inicio superior izquierda del rectángulo y luego el valor del tamaño de los lados del rectángulo.

Con este valor calculado, procedemos a realizar los cálculos pertinentes para cada píxel, obteniendo así las derivadas parciales con las fórmulas (2.7) y (2.8) de cada píxel

al igual que el laplaciano con la fórmula (2.9). Siguiendo los pasos de la discretización del PDE (2.1) tenemos que calcular el coeficiente de difusión (2.12) de cada uno de los píxeles de la imagen. A continuación, calculamos la divergencia (2.13) con los valores obtenidos en la matriz de coeficientes de difusión.

Por último, tenemos que calcular I^{n+1} con la ecuación (2.16). El proceso iterativo se repite el número de veces seleccionado para obtener el resultado. Una vez terminado el proceso iterativo, hay que deshacer las modificaciones que hemos realizado a la imagen, eliminando el acolchado de los bordes y por último, guardando la matriz obtenida en un archivo PGM para poder visualizar el resultado.

4.2.2 Versión MPI

MPI o interfaz de paso de mensajes [12] es una biblioteca que introduce diversas funciones de paso de mensajes que permiten el manejo de máquinas paralelas de memoria distribuida. Está diseñada para los lenguajes C, C++ y Fortran.

Esta versión se desarrolla a partir del código implementado para C. A partir de este código se ha diseñado una paralelización haciendo uso de la librería MPI que nos permite hacer uso de una máquina paralela de memoria distribuida.

Para la implementación paralela MPI utilizamos el paradimga master-slave. El procesador principal (el master) genera varios subproblemas, que son ejecutados por el resto de procesos (slave). En la implementación MPI el procesador master distribuye la imagen entre los procesos. Para hacer esta distribución, como se ha explicado en la Sección 4.1, hemos considerado una descomposición del dominio de la imagen para una topología de anillo bidireccional de manera que cada proceso se encargará del cálculo de una porción de filas consecutivas de la imagen, tal y como indica la Figura 4.4, en la

que se muestra cómo distribuimos la imagen en un ejemplo con tres nodos de computo.

Para distribuir la carga de trabajo de forma equitativa, dividimos la imagen en P subdominios, siendo P el número de elementos de computo utilizados. Asignamos a cada elemento de computo un subdominio de la imagen de tamaño n/P filas, siendo n el número de filas de la imagen. Si la división n/P no es exacta al procesador master se le asignan $n/P + mod(n, P)$ filas.

Además de la porción de la imagen que ha de filtrar cada uno de los procesos implicados, para efectuar el filtrado cada nodo requiere de unas filas adicionales de píxeles. Teniendo en cuenta la dependencia de variables en la discretización mostrada en la Sección 2, como se muestra en la Figura 4.4, cada proceso necesitará un número de filas adicionales que depende de la posición del subdominio en la topología de anillo.

Como ilustra la Figura 4.4, para procesar el subdominio Ω_1, el procesador correspondiente requiere de dos filas adicionales del subdominio inferior Ω_2. Para el último de los subdominios, Ω_3 en la Figura 4.4, el proceso correspondiente requiere de una fila adicional solapada con el subdominio anterior Ω_2. Por último, en caso de tratarse de procesos intermedios, Ω_2 en el ejemplo de la Figura 4.4, el proceso correspondiente requiere de una fila del subdominio anterior Ω_1, y de dos filas del subdominio posterior Ω_3. Este solapamiento «asimétrico» se debe a que, para calcular la imagen de la siguiente iteración (2.16) necesitamos $d^n_{i,j}$ que viene dada por la ecuación (2.13). Si nos fijamos en la fórmula, hace uso de $c^n_{i+1,j}$ para cuyo cálculo en la ecuación (2.12) se necesita $\nabla_R I^n_{i+1,j}$ (ecuación (2.7)) y $\nabla^2 I^n_{i,j}$ (ecuación (2.9)). De las ecuaciones (2.7) y (2.9), se observa que para calcular $d^n_{i,j}$ necesitamos $I_{i+2,j}$, lo que se traduce en 2 filas de solapamiento con el subdominio inferior.

Estas filas solapadas son comunicadas en cada iteración del proceso iterativo.

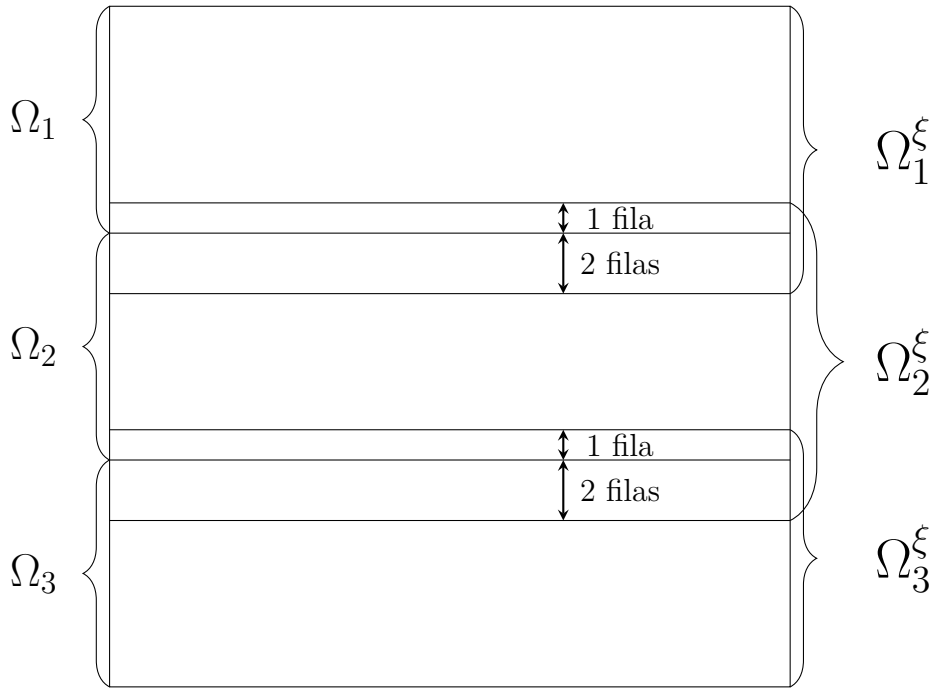


Figura 4.4: Subdominios solapados $\Omega_k^\xi,\ k = 1, 2, 3$

En la Figura 4.4 observamos como quedaría el solapamiento, siendo $\Omega_1^\xi,\ \Omega_2^\xi,\ \Omega_3^\xi$ los subdominios ampliados en los que trabajará cada proceso.

El proceso de filtrado efectuado en cada nodo es similar al proceso secuencial mencionado en la sección 4.2.1. Hay que resaltar que el cálculo de q_o (2.5) en cada iteración lo lleva a cabo únicamente el padre, por lo que tendremos que asegurar que la ROI se encontrará en la porción de la imagen perteneciente al padre para poder tener el valor actualizado en cada iteración.

En cada iteración, los procesos vecinos en la topología de anillo, se comunican las filas solapadas necesarias. Con estas filas cada uno de los procesos pueden calcular las derivadas parciales (2.8) y (2.7), el laplaciano (2.9) y los coeficientes de variación y la divergencia (2.13). Con estos valores calculados, se realiza la actualización de las imágenes con la función de actualización (2.16).

Una vez terminado el proceso iterativo, cada uno de los procesos enviará su porción de la imagen al nodo padre, que será quien se encargue de crear la imagen PGM.

Un punto importante a resaltar es la función de reserva de memoria contigua mencionada al principio del apartado. Esta función es muy importante en esta implementación, ya que por la forma en la que está implementada la librería MPI, es necesario tener todas las posiciones de memoria de la matriz 2D a comunicar de manera contigua.

4.2.3 Versión OpenMP

OpenMP [13] es una interfaz de programación de apliaciones (API) diseñada para ser capaz de hacer uso de múltiples unidades de procesamiento dotadas de memoria compartida. Permite añadir concurrencia y paralelizar códigos escritos en C, C++ o

Fortran. OpenMP se compone de directivas de compilador, rutinas y variables de entorno que modifican el comportamiento del programa en tiempo de ejecución.

Esta implementación, al igual que la implementada en MPI, se ha desarrollado a partir del código secuencial en C. Para esta versión, hemos tenido que estudiar los bucles que son paralelizables dentro del código. En este caso, la librería al estar diseñada para estructuras de memoria compartida no es necesario el paso de mensajes que hacíamos en la versión de MPI.

Teniendo en cuenta cómo se implementa la librería OpenMP, hemos utilizado la directiva *#pragma for parallel* para paralelizar los bucles *for* del código. De cara a obtener la mejor optimización posible, en los bucles anidados se ha paralelizado el bucle externo. Con esta librería tenemos la ventaja de no tener que implementar una superposición de filas de píxeles al igual que hacíamos en la versión MPI, obteniendo un código más sencillo.

A pesar de las ventajas que nos encontramos en esta implementación gracias a la memoria compartida y no necesitar el paso de mensajes para compartir y actualizar las variables, encontramos la desventaja implícita de las condiciones de carrera. Es decir, hay que controlar los accesos a la memoria, o en su defecto, conseguir que las variables a las que pueden acceder distintos hilos estén protegidas.

Para ello, la librería OpenMP tiene la cláusula *private(variable)* que permite generar copias privadas de variables en cada uno de los hilos, evitando accesos inadecuados a las variables en las regiones paralelas.

4.2.4 Versión Hibrida MPI+OpenMP

MPI y OpenMP se pueden combinar [14] para explotar características de ambas bibliotecas. Esta combinación es útil por ejemplo en clusters de computadores donde hay varios nodos multicore conectados en red. Se hace uso de MPI para comunicación de datos entre los distintos nodos conectados en red y se utiliza OpenMP para la paralelización en memoria compartida en cada nodo.

Esta versión se ha desarrollado a partir de la versión implementada en MPI. Dicha implementación ha sido modificada paralelizando con OpenMP los bucles paralelizables en el proceso de filtrado que efectúa cada nodo. De esta forma se utiliza MPI para la comunicación entre los nodos del cluster y se hace uso de OpenMP para la paralelización en cada nodo.

Hay que tener en cuenta que las directivas OpenMP se implementan tanto en las partes del código destinado al nodo master como a los nodos slave, ya que todos los nodos intervienen en el proceso de filtrado.

4.3 Máquinas utilizadas

En esta sección vamos a describir las distintas máquinas que hemos utilizado en la experimentación y obtención de resultados con los códigos implementados.

Las máquinas utilizadas son:

- Máquina 1: Computador Cluster3.
- Máquina 2: Cluster Euler.
- Máquina 3: Supercomputador Trunks.

4.3.1 Máquina 1. Cluster3

Este computador pertenece al Grupo de Investigación Computación de Altas Prestaciones y Paralelismo de la Universidad de Alicante (UA). Esta máquina esta compuesta por un procesador Intel(R) Xeon(R) E5320 CPU @ 1.86 GHz de 8 cores físicos. El sistema tiene 42GB de memoria RAM. El sistema operativo de la maquina es un Linux Ubuntu 18.04.4 LTS.

Para este computador presentaremos los resultados obtenidos con las versiones C, OpenMP y MPI. La versión híbrida no aprovecharía su potencial al no haber varios computadores que necesiten una comunicación por cable.
Veremos, que a pesar de que la librería MPI estar diseñada para máquinas de memoria distribuida, las paralelizaciones con MPI pueden obtener excelentes resultados en máquinas de memoria compartida.

4.3.1.1 Compiladores

Los compiladores instalados en esta máquina y que han sido utilizados para la obtención de resultados son:

- C/OpenMP: gcc 7.4.0
- MPI: Openmpi 4.0.3

4.3.1.2 Conexión

La conexión con la máquina se ha realizado mediante conexión ssh. El acceso se ha realizado mediante la siguiente instrucción por terminal:

```
$ ssh -l nombre_usuario cluster3.cp.ua.es
```

Una vez realizada la conexión, todo se ha realizado mediante la terminal: scripts, ejecución, compilación, obtención de resultados, etc.

4.3.1.3 Scripts de lanzamiento

En esta máquina no se utiliza un sistema de colas para la gestión de ejecutables. Para ejecutar nuestro código, solo ha sido necesario compilarlo de manera adecuada a cada versión:

- C: $ gcc -o nombre_ejecutable SRADseq.c -lm
- OpenMP: $ gcc -fopenmp -o nombre_ejecutable SRADMP.c -lm
- MPI: $ mpicc -o nombre_ejecutable SRADMPI.c -lm

Es necesario introducir el flag `-lm` para que importe correctamente la librería con funciones matemáticas que se utiliza en el código.

Para poder compilar el código con OpenMP, es necesario introducir el flag `-fopenmp` para que las directivas de la librería OpenMP sean reconocidas como directivas y no como comentarios.

En la versión paralela con OpenMP no es necesario especificar el número de *cores* en los que se ejecutará, ya que esto se indica en el mismo código.

4.3.2 Máquina 2. Euler

Este cluster pertenece al Instituto Universitario de Investigación Informática de la UA. Esta plataforma consta de 26 nodos. Cada uno de estos nodos se compone de 2 CPU Intel(R) Xeon(R) X5660 @ 2.8GHz de 12 cores físicos por procesador. Cada nodo consta de 48GB de memoria RAM. El sistema operativo de la máquina es Linux Centos 7.9. Para la interconexión de los nodos que componen la plataforma se utiliza una Red Infiniband QDR.

La estructura de este cluster es la más versátil a la hora de poder comparar las distintas versiones que tenemos, ya que podemos explotar todo el potencial de cada una de las versiones que se han desarrollado para este proyecto. La versión MPI se puede usar tanto dentro del mismo nodo simulando el paso de mensajes con la RAM o haciendo uso de la Infiniband que dispone el cluster para utilizar un mayor número de nodos. Cabe destacar que podremos sacar el potencial de las características de la versión híbrida. Ya que podremos utilizar MPI para distribuir la información entre distintos nodos y hacer uso de OpenMP para paralelizar entre los cores de cada nodo.

4.3.2.1 Compiladores

Los compiladores que utiliza esta máquina y con los que hemos realizado las pruebas son los siguientes:

- C/OpenMP: gcc 4.8.5
- MPI: módulo mpich-gcc → mpicc 5.8.5

La compilación del código se hace de formas distintas en las diferentes implementaciones. Tanto el código secuencial en C como la versión paralela con OpenMP se pueden compilar de la misma forma que hemos explicado para la Máquina 1. Sin embargo, la compilación para la versión MPI es distinta.

Para compilar el código MPI tenemos que cargar el módulo correspondiente con el compilador necesario. Para ello hacemos uso del siguiente comando:

```
$ module load mpich-gcc
```

Y posteriormente, podremos compilar el código con la instrucción:

```
$ mpicc -o nombre_ejecutable SRADMPI.c -lm
```

4.3.2.2 Conexión

Para conectarnos a esta máquina hacemos uso de la terminal. En ella nos conectamos mediante SSH, al igual que en la Maquina 1. En este caso tendremos que utilizar otra URL de conexión:

```
$ ssh -l nombre_usuario euler.iuii.ua.es
```

Una vez conectados, se ha utilizado la consola de terminal para realizar todas las tareas con esta máquina.

4.3.2.3 Scripts de lanzamiento

Para poder ejecutar el código en esta máquina, es necesario crear un *script* de lanzamiento para ponerlo en cola. A continuación mostramos el código que hay que introducir en el *script* con los parámetros necesarios, siendo distinto entre las versiones de C, OpenMP y MPI. Vamos a comenzar por el *script* 4.1 para MPI, que a su vez, también sirve para la versión híbrida MPI+OpenMP.

Código 4.1: Script de lanzamiento MPI en Euler

```
#!/bin/bash
#$ -N SRADMPI
#$ -o salidasTexto/work_mpi_1_nodos10.$JOB_ID.out
#$ -e salidasTexto/work_mpi_1_nodos10.$JOB_ID.err
#$ -pe rr_openmpi 1
#$ -cwd
#$ -q normal.q
source /etc/profile.d/modules.sh;
module load mpich-gcc;
NSLOTS=12;
mpiexec -np $NSLOTS /home/ajimenez/SRADMPI;
```

En este script, en primer lugar se indica el nombre del ejecutable. A continuación, indicamos dónde queremos que nos sitúe los archivos con la salida y la salida de error generadas por el ejecutable. Después con `-pe` indicamos el número de nodos que queremos utilizar y con `-q` la cola de prioridad en la que entrará el ejecutable.

A continuación, tenemos que cargar el módulo mpich-gcc, necesario para poder ejecutar el filtro. La variable `NSLOTS` es el número de cores que vamos a utilizar en la ejecución. Por último, se introduce el comando que hará la ejecución.

Para la versión OpenMP también es necesario crear un *script* 4.2, aunque ligeramente distinto al anterior.

Código 4.2: Script de lanzamiento OpenMP en Euler

```
#!/bin/bash
#$ -N srad
#$ -o testomp.out
#$ -e testomp.err
#$ -l exclusive=true
#$ -cwd
#$ -q normal.q
source /etc/profile.d/modules.sh;
module load mpich-gcc;
/home/ajimenez/srad
```

En este caso, no es necesario indicar el número de nodos a utilizar, ya que OpenMP solo puede trabajar sobre memoria compartida, es decir, sobre un único nodo. Además, tampoco es necesario introducir el número de cores a utilizar, ya que este valor se encuentra en el código.

4.3.2.4 Funcionamiento de cola

La Máquina 2 utiliza como sistema de gestión de colas el gestor de recursos distribuidos Grid Engine [15].

Una vez tenemos el código MPI compilado y creado el *script*, para poder ejecutarlo tenemos que lanzarlo a la cola de ejecución de esta máquina. Esta máquina tiene habilitadas distintas colas de trabajo, con distinta prioridad (la cola «*high*» con mayor prioridad y la cola «normal»). El uso de una u otra cola no afecta a los resultados obtenidos, solo al tiempo que está el trabajo en cola esperando a ser ejecutado. La cola a la que se somete el trabajo se elige en el *script*.

De este modo, para poder lanzar a cola utilizamos el *script* que acabamos de crear con el siguiente comando:

```
$ qsub nombre_script.sh
```

Con esta instrucción, pondremos en la cola pertinente la ejecución de nuestro código. Podemos ver el estado del trabajo sometido a la cola mediante el siguiente comando:

```
$ qstat
```

4.3.3 Máquina 3. Trunks

Este supercomputador pertenece a la Universidad Miguel Hernández (UMH). La plataforma multicore está equipada con 2 CPU sockets. En cada CPU, el procesador instalado es un Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz con 24.75 MB L3 Cache. Cada procesador está compuesto por 18 cores físicos, obteniendo un número total de 36 cores físicos en el sistema. La memoria principal consta de un tamaño de 384 GB de memoria DDR3.

De la misma forma que con la Máquina 1, para este supercomputador de memoria compartida, presentaremos los resultados obtenidos con las implementaciones secuen-

cial, OpenMP y la versión MPI.

4.3.3.1 Compiladores

Los compiladores instalados en esta máquina y con los que se ha trabajado para la experimentación y posterior obtención de resultados son:

- C/OpenMP: GNU gcc 11.1.0
- OpenMPI: openmpi 4.0.5

Para compilar el código C y OpenMP se procede de la misma forma que en las Máquinas 1 y 2. La compilación de la versión MPI es similar al proceso utilizado en la Máquina 2. En primer lugar, cargamos el módulo correspondiente a la librería MPI:

```
$ module load openmpi-4.0.5-gcc-11.1.0-3irqfkv
```

Tras cargar dicho módulo, podremos compilar el código con la instrucción:

```
$ mpicc -o nombre_ejecutable SRAPI.c -lm
```

4.3.3.2 Conexión

Al igual que en las máquinas anteriores, vamos a utilizar SSH para conectarnos a la máquina mediante la terminal. Para poder conectarnos utilizaremos el siguiente comando:

```
$ ssh -l nombre_usuario trunks.umh.es
```

Una vez conectados con la máquina, se ha utilizado la terminal para realizar todas las tareas pertinentes.

4.3.3.3 Scripts de lanzamiento

Tras la compilación, necesitamos de un *script* 4.3 para poder ejecutar en la máquina. Con este ejecutable podremos mandar nuestro trabajo a la cola.

Primero vamos a mostrar el *script* 4.3 de lanzamiento de la versión MPI.

Código 4.3: Script de lanzamiento MPI en Trunks

```
#!/bin/bash

#SBATCH --job-name=SRADMPI
#SBATCH --output=salida_mpi.out
#SBATCH --error=error_mpi.err
#SBATCH -n 36
#SBATCH --exclusive

module load openmpi-4.0.5-gcc-11.1.0-3irqfkv

srun ./SRADMPI
```

En este *script*, lo primero que hacemos es indicar el nombre del ejecutable. Al igual que en el *script* de la Máquina 2, se tiene que indicar el nombre de dos archivos donde se guardarán las salidas de terminal y la salida de error obtenida de la ejecución. Con el flag `-n` indicamos el número de cores que vamos a utilizar, en este caso 36. A diferencia de Euler, en este caso no se pone en el ejecutable directamente. También indicamos que queremos que se ejecute de manera exclusiva con el flag `--exclusive`. De esta forma

nos aseguramos de que los procesadores ejecuten nuestro código en exclusiva y no se compartan con otra ejecución.

Por último, solo nos queda cargar el módulo con el compilador de MPI e introducir el comando que ejecutará el programa.

El siguiente *script* 4.4 muestra cómo lanzar trabajos OpenMP.

Código 4.4: Script de lanzamiento OpenMP en Trunks

```
#!/bin/bash

#SBATCH --job-name=sradomp
#SBATCH --output=salida_omp_36.out
#SBATCH --error=error_omp_36.err
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=36

srun ./sradomp_36
```

Como se puede observar, tenemos una estructura muy parecida, donde indicamos el nombre, ficheros de salida de terminal y salida de errores. Pero podemos encontrar diferencias como el flag `-ntask` y `--cpus-per-task` donde indicamos el número de CPUs a reservar para la ejecución del código. En este caso, no es necesario cargar ningún módulo.

4.3.3.4 Funcionamiento de cola

La Máquina 3 utiliza el sistema de colas «*slurm*» [16].

Para enviar a cola el *script* se utiliza el siguiente comando:

```
$ sbatch ./nombre_script.sh
```

Podemos ver el estado del trabajo sometido a la cola mediante:

```
    $ squeue
```

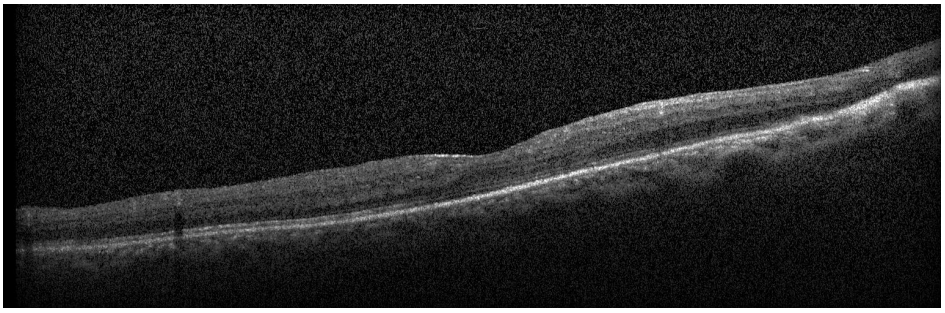
5 Resultados

En esta sección vamos a presentar los resultados obtenidos en los experimentos desarrollados. Se presentarán los resultados obtenidos por las implementaciones desarrolladas en las tres máquinas descritas en la Sección 4.

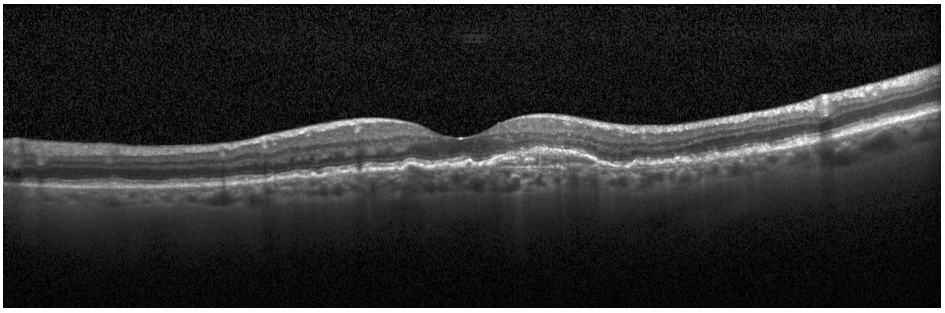
Cabe destacar que para obtener dichos resultados se han realizado 30 ejecuciones de cada experimento con las mismas características (iteraciones, hilos, cores, imágenes, etc). Se han descartado los 5 valores máximos y mínimos, y se ha realizado la media aritmética de las ejecuciones restantes.

Para comparar los resultados, vamos a considerar ejecuciones en la misma máquina, es decir, compararemos las versiones del código, tiempo y *Speed up* dentro de la misma plataforma.

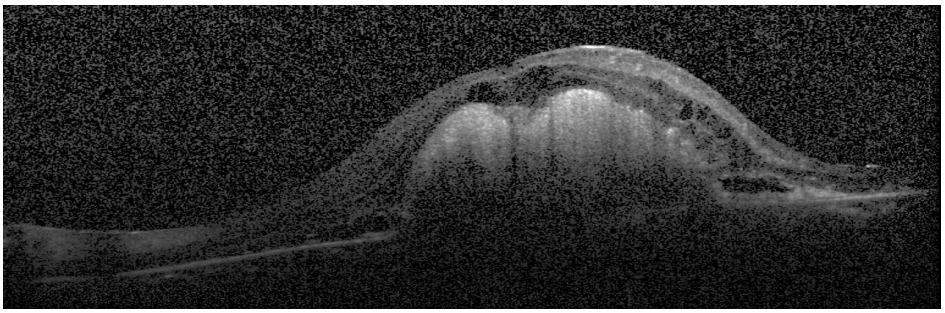
Las imágenes utilizadas en este proyecto han sido obtenidas de las bases de datos «*Large Dataset of Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images*» [17] y UltraSoundCase [18]. En las Figuras 5.1 y 5.2 podemos observar algunas de las que han sido utilizadas en este proyecto y sus respectivas dimensiones. Con el objetivo de mostrar el comportamiento de las implementaciones en imágenes de menor tamaño, en la memoria se presentan los resultados obtenidos en la imagen de ultrasonidos US 1 (Figura 5.2c). Se ha experimentado en otras imágenes de tamaño similar obteniendo conclusiones parecidas.



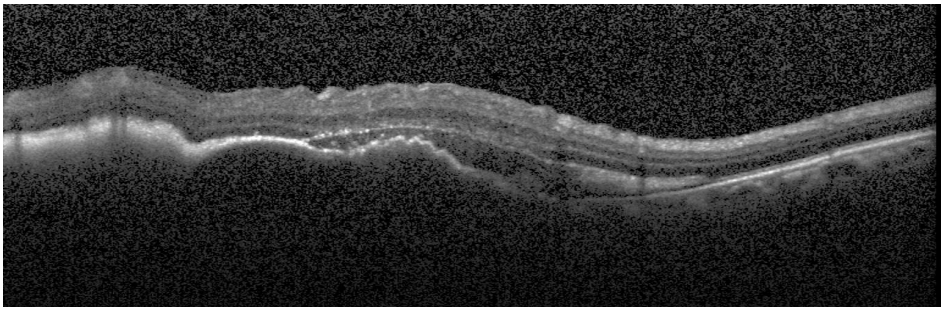
(a) OCT 1 (496 x 1536)



(b) OCT 2 (496 x 1536)

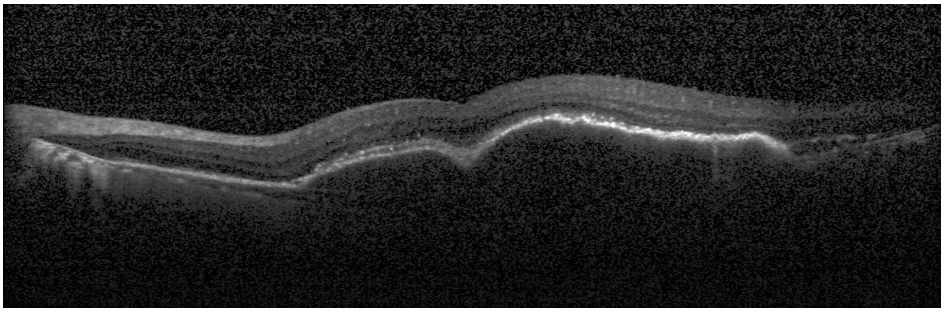


(c) OCT 3 (496 x 1536)

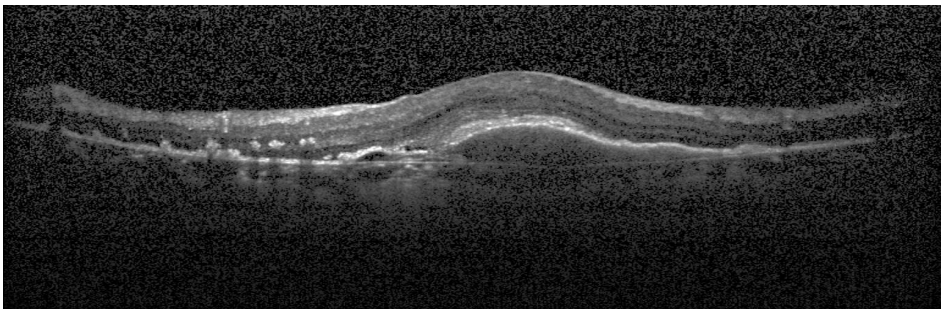


(d) OCT 4 (496 x 1536)

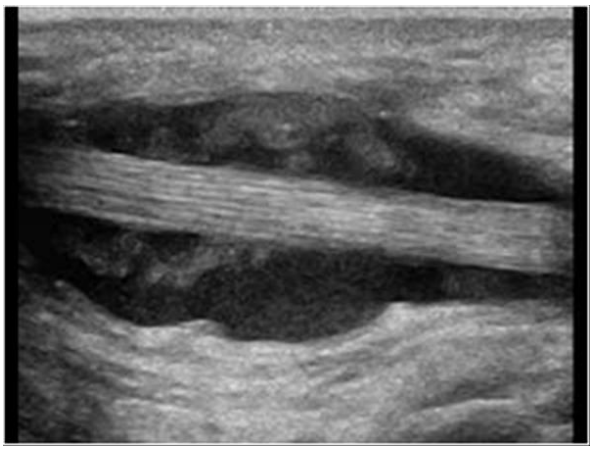
Figura 5.1: Imágenes utilizadas



(a) OCT 5 (496 x 1536)



(b) OCT 6 (496 x 1536)



(c) US 1 (450 x 600)

Figura 5.2: Imágenes utilizadas

Para poder comparar la eficiencia computacional vamos a hacer uso del *speedup* y la eficiencia. El *speedup* se define como:

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

Donde T_{sec} es el tiempo de la ejecución secuencial y T_p el tiempo de la ejecución del código paralelo al hacer uso de p unidades de cálculo. La eficiencia se define como:

$$E = \frac{S_p}{p}$$

Donde S_p es el *speedup* definido anteriormente, y p es el número de unidades de cálculo utilizadas.

Los parámetros del filtro SRAD han sido fijados según las indicaciones de los autores. Por otra parte, el proceso de filtrado SRAD es un proceso iterativo. En los experimentos se han utilizado 400 iteraciones. Este número de iteraciones ha sido seleccionado tras realizar un estudio de las imágenes de salida del algoritmo con distintas imágenes OCT. Para ello, se ha hecho uso del contrast-to-noise ratio (CNR) [19]. El CNR mide el contraste entre la ROI y una superficie de fondo seleccionada. Se calcula mediante:

$$CNR_m = 10log_{10}\frac{|\mu_m - \mu_b|}{(0.5(\sigma_m^2 + \sigma_b^2))^{\frac{1}{2}}} \tag{5.1}$$

donde σ_b y μ_b son la desviación estándar y la media de la sección de fondo, mientras que μ_m yσ_m son la media y la desviación estándar de la sección de interés m. La desviación estándar del ruido se estima a partir del coeficiente wavelet de subbanda diagonal de la imagen ruidosa mediante el estimador robusto por mediana:

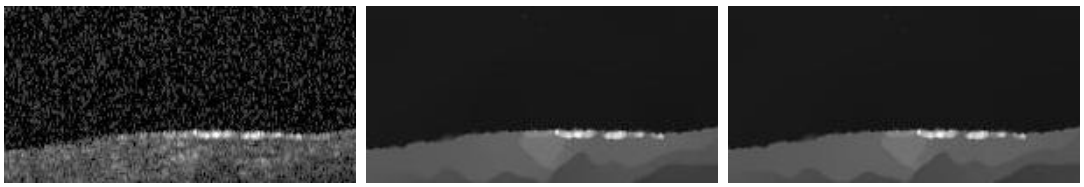
$$\sigma_n = \frac{median(|c_j|)}{0.6745}, \forall j \in HH \tag{5.2}$$

En la Tabla 5.1 se muestra el CNR de las imágenes filtradas con 100, 200, 300, 400,

500, 600, y 700 iteraciones de las imágenes OCT 1, OCT 2 y OCT 4. Como se puede observar, el valor de CNR obtenido en todas las imágenes aumenta de manera significativa en cada aumento de 100 iteraciones, pero a partir de 400 esta aumento se reduce considerablemente en todas las imágenes. Por ello, se ha elegido 400 como número de iteraciones para todas las imágenes procesadas, ya que no compensa el mínimo aumento de CNR con el coste temporal que supone el aumento de iteraciones. Esta decisión viene reforzada por el hecho de que las imágenes obtenidas por el filtro a partir de 400 iteraciones, son al ojo humano, casi idénticas con pequeñas mejoras, como se puede observar en las Figuras 5.3, 5.4 y 5.5.

Iteraciones	OCT 1	OCT 2	OCT 4
100	21.13	24.26	20.39
200	21.91	24.96	22.45
300	23.02	25.33	23.97
400	23.72	25.85	25.12
500	24.1	25.4	25.5
600	24.62	25.65	26.12
700	24.7	26.1	26.54

Tabla 5.1: CNR en distintas imágenes

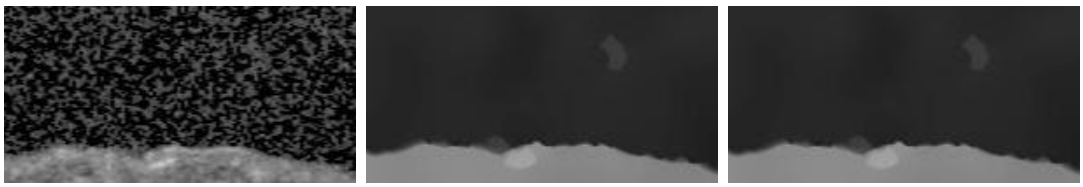


(a) Sin filtrar **(b)** 400 iteraciones **(c)** 700 iteraciones

Figura 5.3: Comparación ROI en OCT 1

(a) OCT 2 sin filtrar **(b)** 400 iteraciones **(c)** 700 iteraciones

Figura 5.4: Comparación ROI en OCT 2



(a) Sin filtrar **(b)** 400 iteraciones **(c)** 700 iteraciones

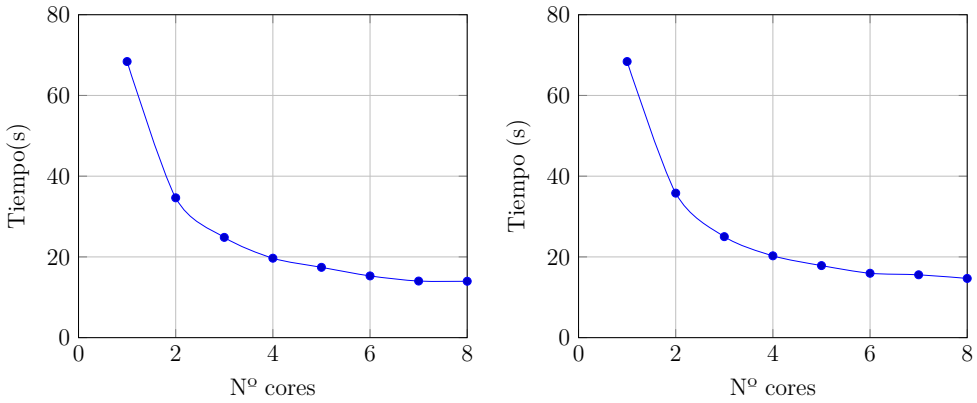
Figura 5.5: Comparación ROI en OCT 3

5.1 Máquina 1

Las imágenes de las bases de datos que hemos utilizado [18] y [17] que se muestran en la Figura 5.1 han sido procesadas en la Máquina 1 de acuerdo a lo mencionado al inicio del apartado. En esta sección presentamos los datos obtenidos para la imagen OCT 1. Para el resto de imágenes OCT se obtienen conclusiones similares. Además, se mostrarán los resultados obtenidos con la imagen US 1 (Figura 5.2c), para mostrar el comportamiento de las implementaciones en una imagen de tamaño inferior.

En la Figura 5.6 mostramos el tiempo en segundos obtenido para las versiones OpenMP y MPI para la imagen OCT 1 al incrementar el número de cores.

En la Figura 5.7 podemos observar los tiempos obtenidos a partir de la imagen US 1, de un tamaño más reducido. En este caso, observamos que el comportamiento es el mismo pero con tiempos de ejecución más bajos que los mostrados en la Figura 5.6. Debido al tamaño del resto de imágenes OCT, que son el mismo que OCT 1, encontramos resultados casi idénticos. Esto es debido a que la complejidad temporal

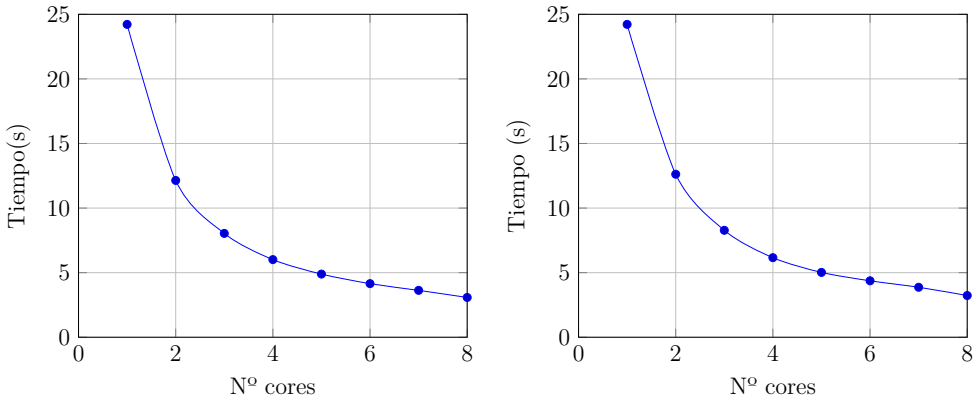


(a) Tiempos obtenidos en la versión OpenMP

(b) Tiempos obtenidos en la versión con MPI

Figura 5.6: Tiempos obtenidos con la imagen OCT 1 en la Máquina 1

del algoritmo depende del tamaño de la imagen y del número de iteraciones efectuadas en el proceso de filtrado.



(a) Tiempos obtenidos en la versión OpenMP

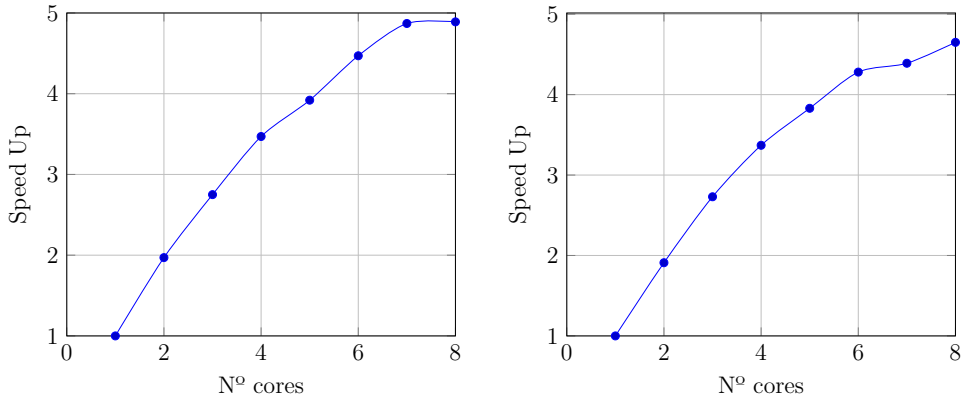
(b) Tiempos obtenidos en la versión con MPI

Figura 5.7: Tiempos obtenidos con la imagen US 1 en la Máquina 1

A pesar de tener tiempos de ejecución distintos, ambas imágenes presentan una evolución de la curva temporal muy parecida. Observamos que el comportamiento de la implementación paralela en ambas imágenes es el esperado, al incrementar el número de elementos de computo el tiempo de ejecución disminuye.

En cuanto a las dos implementaciones (OpenMP y MPI), podemos apreciar que en esta máquina no se aprecia gran diferencia y que las dos tienen un tiempo de ejecución parecido.

En la Figura 5.8 se observan los speedups obtenidos para las ejecuciones correspondientes a la Figura 5.6. En estas figuras podemos apreciar que la versión OpenMP obtiene mejores resultados, aunque la diferencia no es muy significativa.



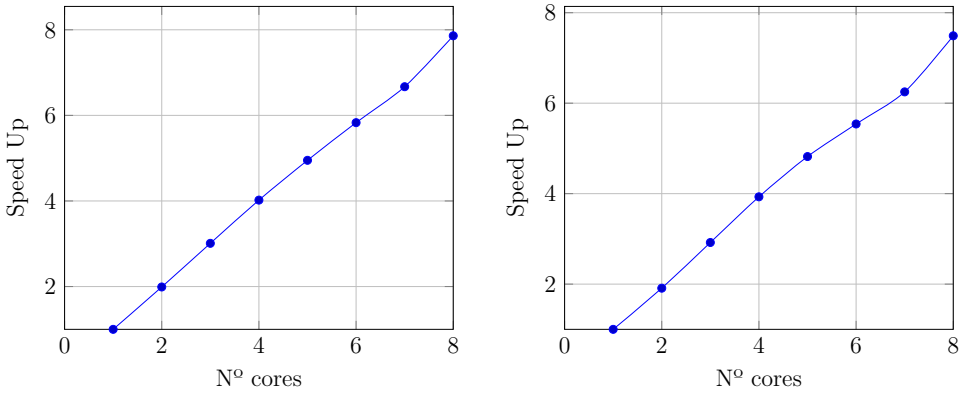
(a) Speedup obtenido con la versión OpenMP

(b) Speedup obtenido con la versión MPI

Figura 5.8: Speedups obtenidos con la imagen OCT 1 en la Máquina 1

La Figura 5.9 nos presenta el speedup para la imagen US 1. La figura es muy similar a la anterior, aunque encontramos valores más altos, por lo que podemos deducir que se ha obtenido un mejor rendimiento con esta imagen de menor tamaño. De igual forma que en la Figura 5.8, ambas Figuras (5.9a y 5.9b) muestran una evolución muy similar.

La Figura 5.10 nos muestra la eficiencia para la imagen OCT 1 y la Figura 5.11 para la imagen US 1. En ambos casos podemos observar como la versión de OpenMP hace mejor uso de los recursos y obtenemos valores de eficiencia ligeramente mayores que la versión MPI. Esto es esperable pues OpenMP ha sido diseñado para máquinas de

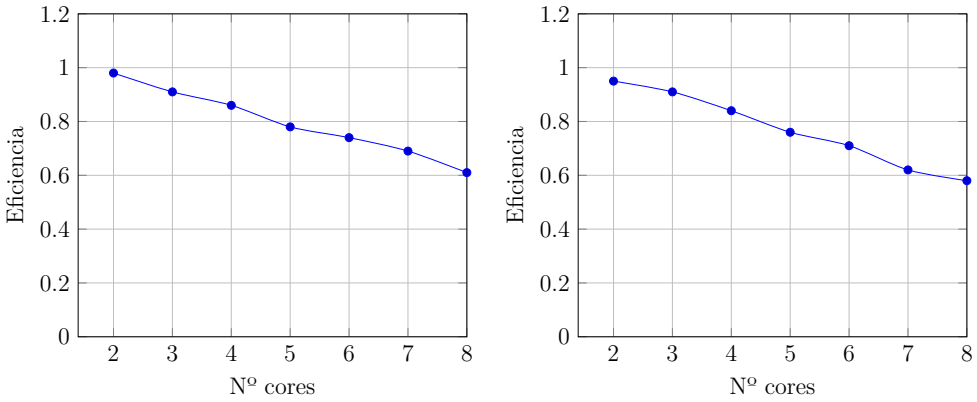


(a) Speedup obtenido con la versión OpenMP

(b) Speedup obtenido con la versión MPI

Figura 5.9: Speedups obtenidos con la imagen US 1 en la Máquina 1

memoria compartida.

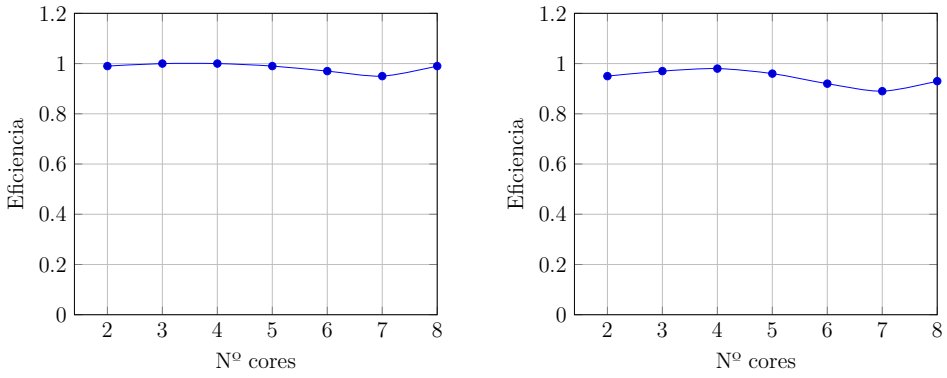


(a) Eficiencia de la versión OpenMP

(b) Eficiencia de la versión MPI

Figura 5.10: Eficiencia obtenida con la imagen OCT 1 en la Máquina 1

En la Figura 5.11 se presenta la eficiencia para la imagen US 1. En este caso, la eficiencia obtenida es mucho mayor que en la imagen OCT 1. Como se puede ver, la eficiencia obtenida en la versión OpenMP va del 95% al 100%, y en la versión MPI del 88% al 98%.



(a) Eficiencia de la versión OpenMP **(b)** Eficiencia de la versión MPI

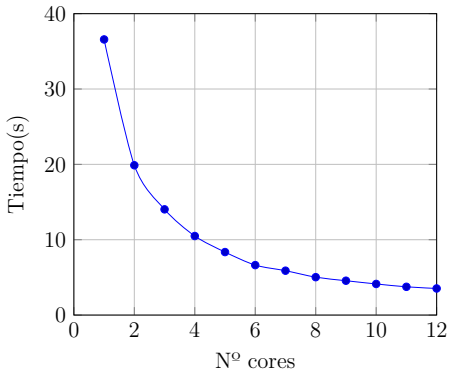
Figura 5.11: Eficiencia obtenida con la imagen US 1 en la Máquina 1

5.2 Máquina 2

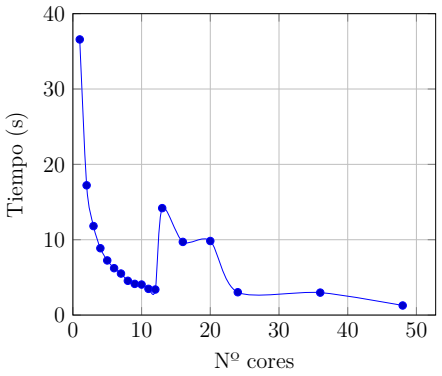
Debido a la arquitectura de esta máquina, hacemos uso de las 3 versiones desarrolladas, OpenMP, MPI y MPI+OpenMP. Como se ha comentado en la Sección 4.3.2, este cluster está compuesto por 26 nodos de 12 cores físicos cada uno y están interconectados mediante Red Infiniband. Hay que tener en cuenta que la versión OpenMP solamente se puede lanzar en un único nodo, por lo que el número máximo de cores que se podrá utilizar son 12. En las versiones MPI y MPI+OpenMP podemos utilizar un número mayor de unidades de cálculo, haciendo uso de varios nodos.

En la Figura 5.12 podemos observar el coste temporal de la ejecución de las distintas versiones con la imagen OCT 1. Este computador es mucho más potente que la Maquina 1 y podemos ver la disminución global del tiempo de ejecución. Al aumentar los cores en la versión OpenMP podemos observar que continua reduciéndose el tiempo de ejecución con unos tiempos cercanos al tiempo real. En la Figura 5.12b podemos ver que al hacer uso de 2 nodos y 13 cores aumenta el tiempo de ejecución. Esto se debe al coste de las comunicaciones al hacer uso de más de un nodo. Al aumentar el número de cores utilizados se compensa el coste temporal que implican estas comunicaciones y se consiguen mejores resultados, llegando a 1,27s con 48 cores.

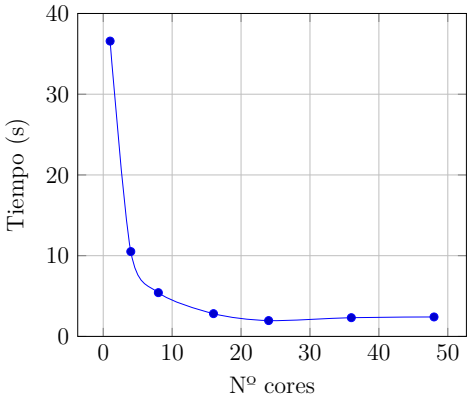
En la versión MPI+OpenMP (Figura 5.12c), se han realizado pruebas con 4, 8, 16, 24, 36 y 48 cores. Desde los 4 cores, se ha hecho uso de 2 nodos, por lo que el coste temporal de las comunicaciones está implícito en todos los valores de prueba. El aumento de cores ha sido el mismo en los dos nodos de manera simétrica. Cuando se han utilizado el número máximo de cores de los dos nodos (24 cores) se han realizado pruebas añadiendo otro nodo con todas las unidades de cómputo de las que dispone, obteniendo los valores de 36 y 48 cores. El uso de las comunicaciones entre nodos desde los 4 cores hace que no suframos un aumento del tiempo de ejecución al añadir un nuevo nodo pasados los 12 cores, como sucede en la versión MPI (Figura 5.12b).



(a) Tiempos obtenidos en la versión OpenMP



(b) Tiempos obtenidos en la versión con MPI

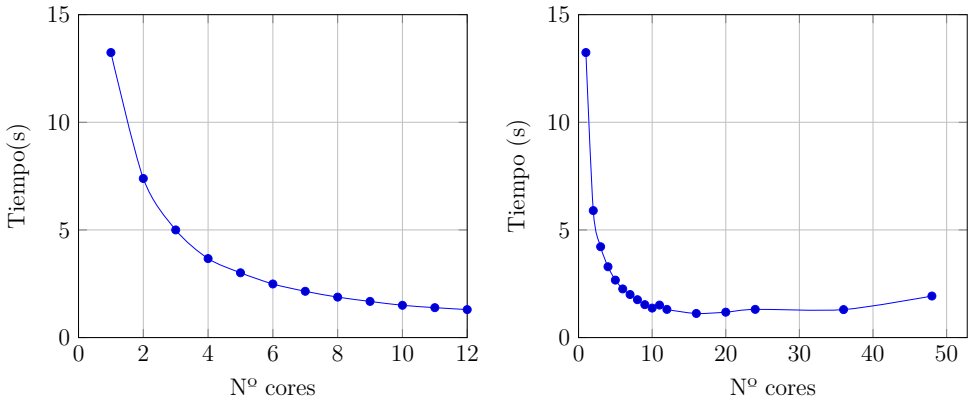


(c) Tiempos obtenidos en la versión híbrida MPI+OpenMP

Figura 5.12: Tiempos obtenidos con la imagen OCT 1 en la Máquina 2

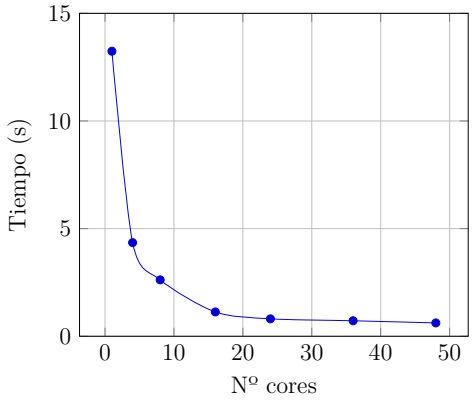
En la Figura 5.13 podemos observar el comportamiento del algoritmo con la imagen US 1. Podemos observar que, al ser una imagen de tamaño inferior, se obtienen unos tiempos de ejecución más bajos en todas las versiones. En la Figura 5.13b no se ha realizado la prueba con 13 cores, que es el valor más lento de la Figura 5.12b. Además, podemos observar que, a partir de un número de elementos de cómputo el tiempo de ejecución en MPI deja de mejorar. Esto es debido a que para la imagen de ese tamaño se ha llegado al punto óptimo de paralelización.

Por otra parte, podemos observar como la versión híbrida MPI+OepnMP, es la que obtiene mejores tiempos de ejecución. En esta implementación, una vez distribuida la imagen entre los nodos haciendo uso de MPI, el uso de OpenMP en la memoria compartida de cada nodo permite mejorar los tiempos obtenidos.



(a) Tiempos obtenidos en la versión OpenMP

(b) Tiempos obtenidos en la versión con MPI

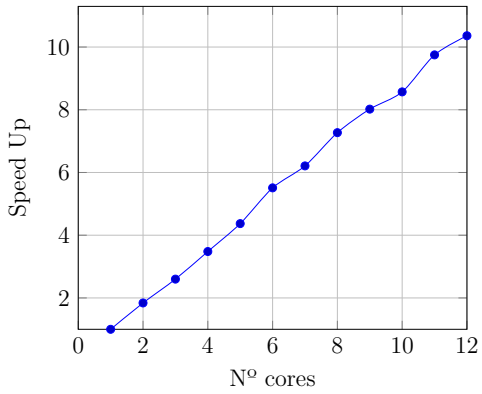


(c) Tiempos obtenidos en la versión híbrida MPI+OpenMP

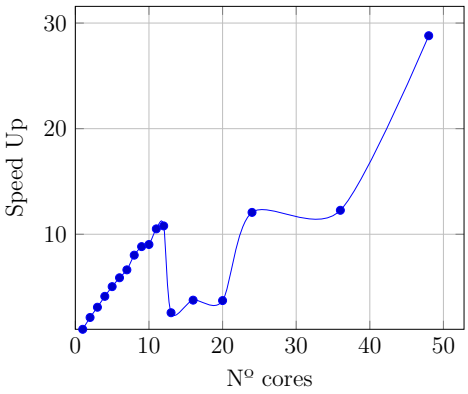
Figura 5.13: Tiempos obtenidos con la imagen US 1 en la Máquina 2

En la Figura 5.14 presentamos el speedup obtenido para la imagen OCT 1. Se obtienen valores de speedup significativos en todas las implementaciones, indicando el buen funcionamiento del código paralelo. El rendimiento mostrado por los algoritmos aumenta de manera casi lineal en la versión OpenMP (Figura 5.14a). En la versión MPI

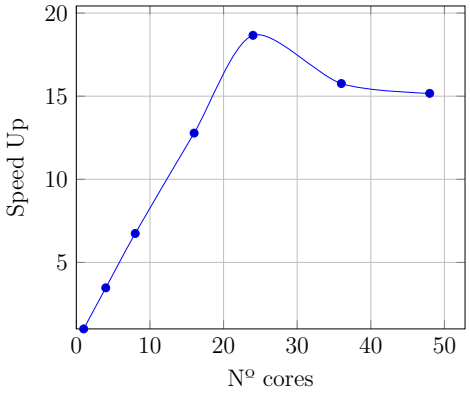
encontramos una disminución en el speedup al utilizar 2 nodos sin hacer uso de todos los cores que poseen. Esta perdida va desapareciendo al hacer uso de toda la potencia computacional de la que dispone cada nodo, llegando a obtener speedups cercanos a 30 con 48 cores.



(a) Speedup obtenido con la versión OpenMP



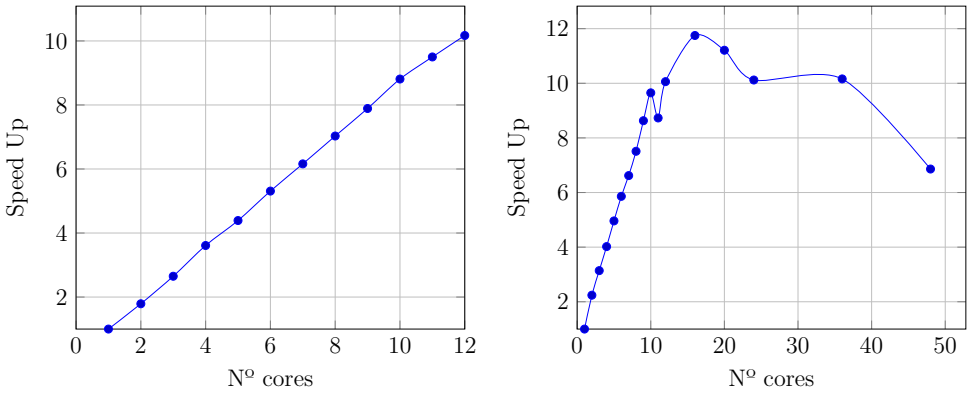
(b) Speedup obtenido con la versión MPI



(c) Speedup obtenido con la versión híbrida MPI+OpenMP

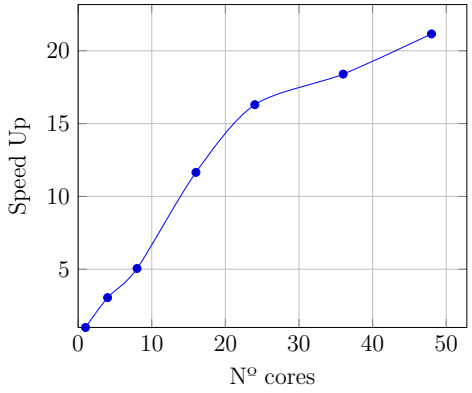
Figura 5.14: Speedups obtenidos con la imagen OCT 1 en la Máquina 2

Analizando los datos de la Figura 5.15 podemos comprobar el rendimiento del algoritmo con una imagen más pequeña. El comportamiento del código paralelo en la versión OpenMP es muy parecido al mostrado con la imagen US 1. Sin embargo, el speedup mostrado por las versiones MPI e híbrida son bastante distintas.



(a) Speedup obtenido con la versión OpenMP

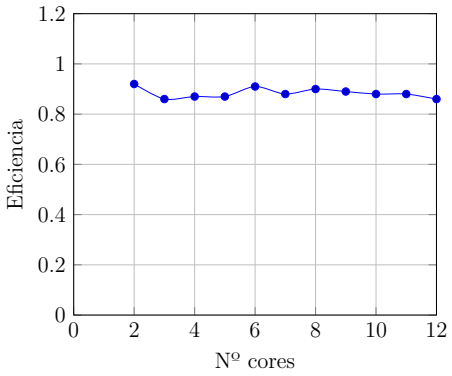
(b) Speedup obtenido con la versión MPI



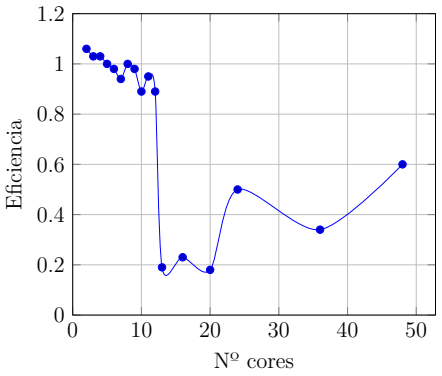
(c) Speedup obtenido con la versión híbrida MPI+OpenMP

Figura 5.15: Speedups obtenidos con la imagen US 1 en la Máquina 2

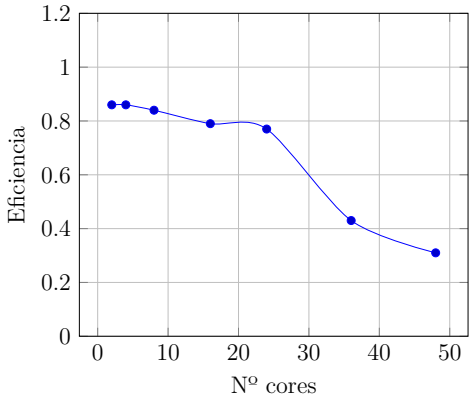
En primer lugar, si observamos la Figura 5.15b, el speedup consigue su valor máximo con 16 cores y va disminuyendo paulatinamente hasta los 48 cores. Este comportamiento se debe a que la mejora de tiempo conseguida por la paralelización de la imagen es inferior al coste temporal del paso de los mensajes, mermando la velocidad de la ejecución y reduciendo el speedup conseguido. Sin embargo, la versión híbrida no muestra este comportamiento. Hace uso de la memoria compartida con OpenMP dentro de cada nodo, consiguiendo resultados muy positivos.



(a) Eficiencia de la versión OpenMP



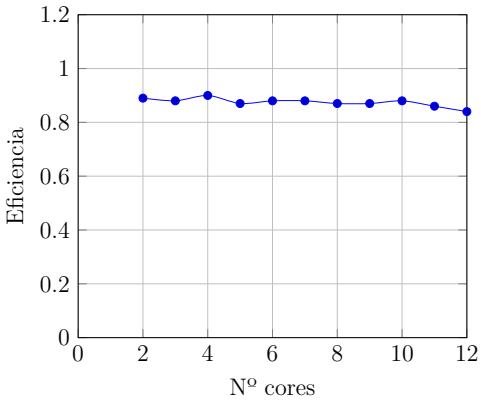
(b) Eficiencia de la versión MPI



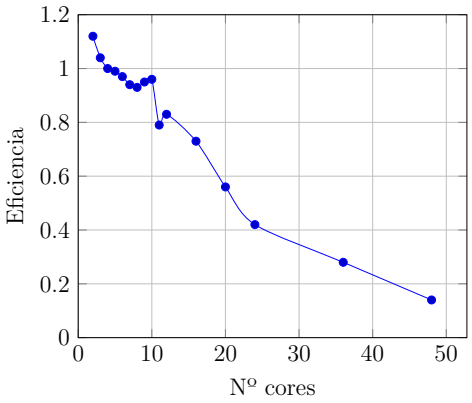
(c) Eficiencia de la versión híbrida MPI+OpenMP

Figura 5.16: Eficiencia obtenida con la imagen OCT 1 en la Máquina 2

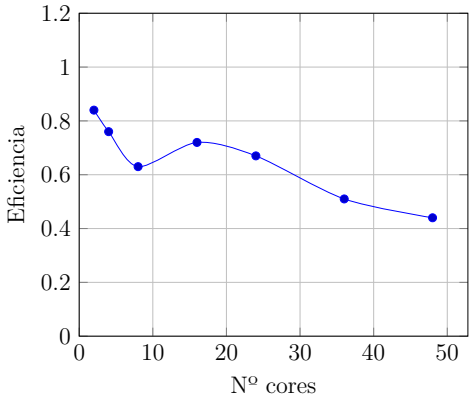
En la Figura 5.16 podemos comprobar la eficiencia obtenida. Los valores obtenidos son muy buenos en la versión OpenMP, que oscilan entre un 86% y un 92%. En la versión MPI en ejecuciones en un solo nodo obtenemos superspeedup con valores superiores al 100% en varios casos [20]. En esta implementación encontramos una caída de la eficiencia en el intervalo 13-20 cores por el uso de la conexión Red. A posteriori, al hacer uso de todos los cores del nodo, obtenemos valores mejores de eficiencia. Si nos fijamos en los valores de la Figura 5.16c, seguimos obteniendo valores muy buenos de eficiencia, sin caídas importantes al aumentar el número de cores. Solo disminuye a valores cercanos al 40% en los 36 y 48 cores.



(a) Eficiencia de la versión OpenMP



(b) Eficiencia de la versión MPI



(c) Eficiencia de la versión híbrida MPI+OpenMP

Figura 5.17: Eficiencia obtenida con la imagen US 1 en la Máquina 2

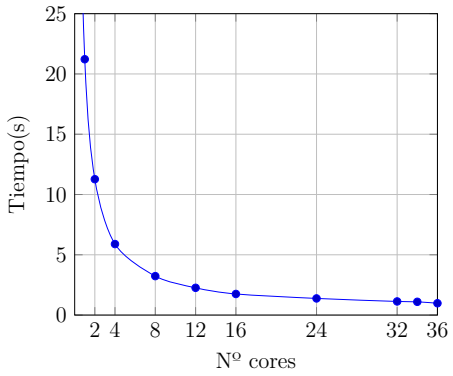
En la Figura 5.17 podemos observar el rendimiento con la imagen US 1. De igual forma, el comportamiento es muy parecido al mostrado en la Figura 5.16. Lo que nos indica que el rendimiento del algoritmo paralelo es correcto. En la Figura 5.17a podemos ver que los valores oscilan entre un 90% y un 84% de eficiencia. La Figura 5.17b tiene un comportamiento muy parecido al mostrado con la imagen OCT 1. Volvemos a encontrar valores superiores al 100% al hacer uso de pocos cores en el mismo nodo. Aunque la eficiencia llega a disminuir a valores muy bajos por debajo del 20%, debido a la caída del speedup al incrementar el número de cores, como se ha comentado anteriormente. Por su parte, el código híbrido, mantiene valores de eficiencia superiores superiores al 40% al hacer uso de un número mayor de elementos de computo.

5.3 Máquina 3

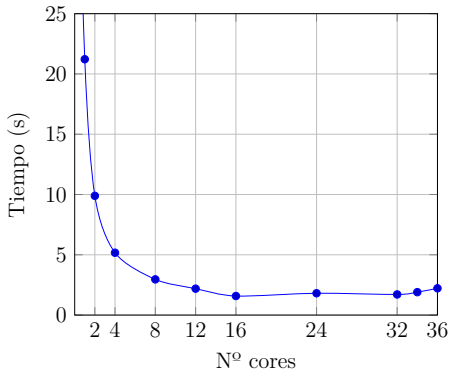
Esta máquina, como se indica en la Sección 4.3.3, consta de 36 cores provenientes de 2 CPUs en una misma plataforma de memoria compartida. Esta arquitectura hace que la obtención de resultados de la versión híbrida no aporte valor a la experimentación. Por ello, únicamente se han hecho pruebas con la versión OpenMP y la versión MPI.

En la Figura 5.18 podemos ver la evolución del tiempo de ejecución al aumentar el número de cores. En ambas versiones obtenemos los resultados esperados y acordes a los vistos anteriormente. En la versión OpenMP (Figura 5.18a) vemos como el tiempo disminuye hasta valores inferiores al segundo de ejecución con 36 cores. Por otra parte, los tiempos obtenidos por la implementación MPI no son tan reducidos, pero la diferencia no es significativa. También obtenemos valores cercanos a un segundo de ejecución, pero al aumentar el número de cores, a partir de 16, no obtenemos mejoras en tiempos de ejecución. Este mejor comportamiento de la implementación con OpenMP era el esperado en una máquina de memoria compartida.

En la Figura 5.19 vemos los resultados de la imagen US 1. En este caso, los resultados obtenidos son ciertamente buenos. Los tiempos de ejecución se encuentran por debajo del medio segundo. En ambas versiones, obtenemos valores de 0,4s en alguna ejecución. La Figura 5.18a muestra un comportamiento más constante y reduce el tiempo en cada iteración. Por otra parte, la Figura 5.18b presenta pequeñas oscilaciones debido a que la librería MPI, ha sido diseñada para el paso de mensajes en arquitecturas con memoria distribuida. A pesar de ello se puede observar que el rendimiento de MPI en máquinas de memoria compartida es bueno, obteniendo resultados cercanos a la implementación en OpenMP.

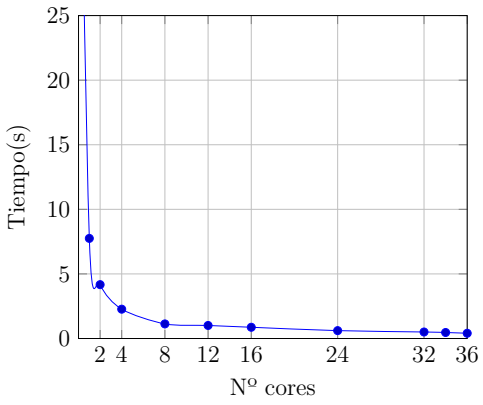


(a) Tiempos obtenidos en la versión OpenMP

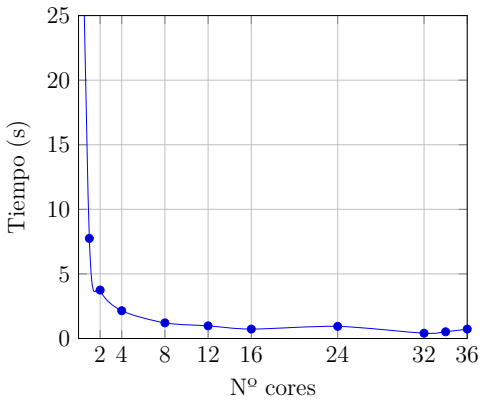


(b) Tiempos obtenidos en la versión con MPI

Figura 5.18: Tiempos obtenidos con la imagen OCT 1 en la Máquina 3



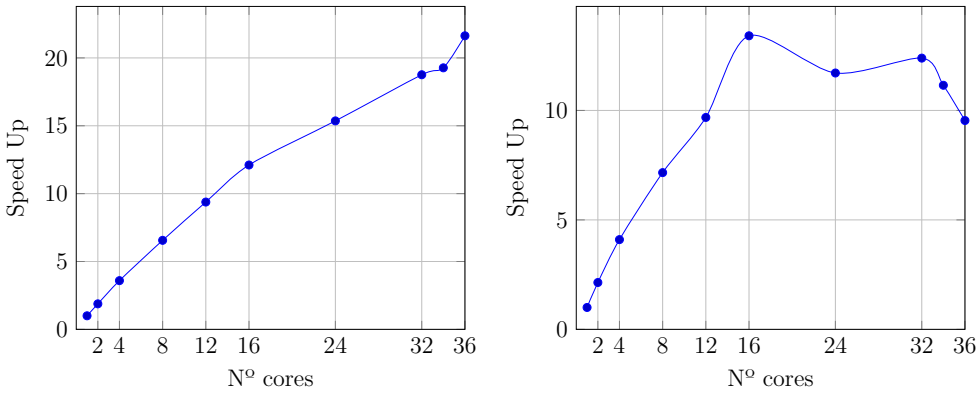
(a) Tiempos obtenidos en la versión OpenMP



(b) Tiempos obtenidos en la versión con MPI

Figura 5.19: Tiempos obtenidos con la imagen US 1 en la Máquina 3

En la Figura 5.20 podemos observar los valores de speedup obtenidos para la imagen OCT 1. Si nos fijamos en la Figura 5.20a encontramos que el speedup mejora al aumentar el número de cores. En la versión MPI, la Figura 5.20b muestra valores inferiores a los calculados en la versión OpenMP. Además, se observa una reducción del speedup a partir de los 16 cores, habiendo llegado en este punto al número de cores óptimos para esta imagen.

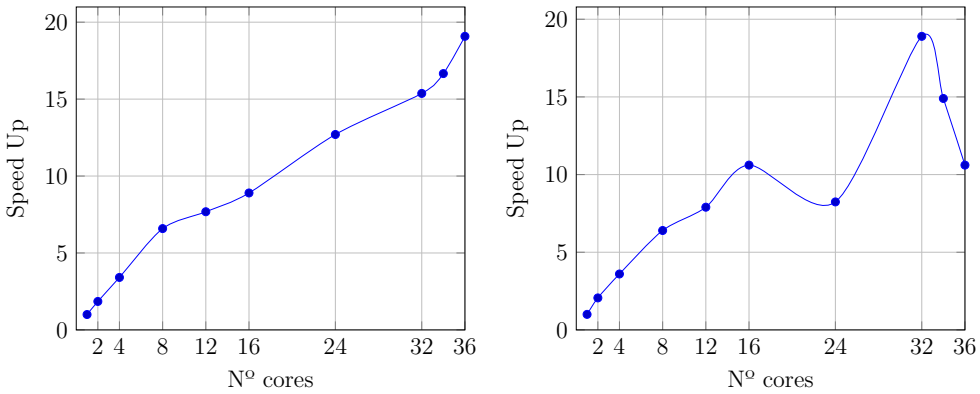


(a) Speedup obtenido con la versión OpenMP

(b) Speedup obtenido con la versión MPI

Figura 5.20: Speedups obtenidos con la imagen OCT 1 en la Máquina 3

En la Figura 5.21 podemos observar el speedup obtenido para la imagen US 1. En este caso, el speedup mostrado en la Figura 5.21 sigue el mismo patrón observado en la Figura 5.20. En la Figura 5.21a se observa una evolución casi lineal del speedup, mientras que en la Figura 5.21b encontramos mayor variación. No obstante en esta imagen se pueden obtener valores de speedup bastante cercanos entre ambas versiones, como se puede observar con 36 cores en la Figura 5.21b y con 32 cores en la Figura 5.21b.

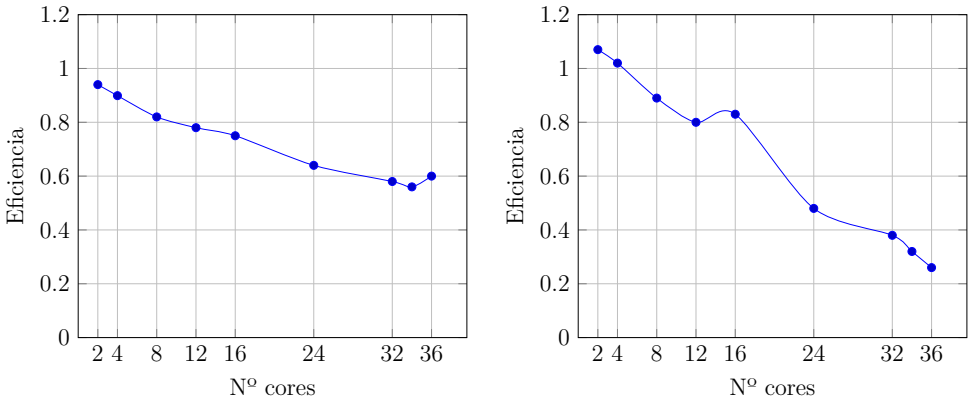


(a) Speedup obtenido con la versión OpenMP

(b) Speedup obtenido con la versión MPI

Figura 5.21: Speedups obtenidos con la imagen US 1 en la Máquina 3

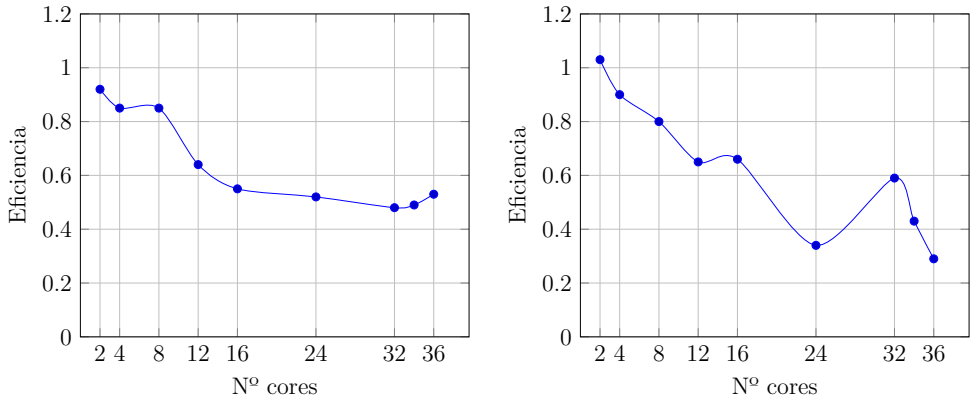
En la Figura 5.22 mostramos la eficiencia obtenida para la imagen OCT 1. En la versión OpenMP (Figura 5.22a), el valor de la eficiencia se haya entre el 94% con 2 cores y 56% con 33 cores. Por su parte, en la versión MPI, obtenemos una eficiencia que va desde 107% con 2 cores y 26% con 36 cores (Figura 5.22b).



(a) Eficiencia de la versión OpenMP **(b)** Eficiencia de la versión MPI

Figura 5.22: Eficiencia obtenida con la imagen OCT 1 en la Máquina 3

Por último, la Figura 5.23 muestra la eficiencia del código paralelo en la imagen US 1. Los valores obtenidos para esta imagen muestran una mayor eficiencia que la mostrada en la Figura 5.22 para la imagen OCT 1. Con esta imagen más pequeña US 1, obtenemos valores de eficiencia mínimos mayores que los obtenidos con la imagen OCT 1. En la Figura 5.23a obtenemos eficiencias entre 92% y 48%. Para la versión MPI, podemos observar en la Figura 5.23b valores de eficiencia entre 103% y 29%.



(a) Eficiencia de la versión OpenMP **(b)** Eficiencia de la versión MPI

Figura 5.23: Eficiencia obtenida con la imagen US 1 en la Máquina 3

6 Conclusiones

El método SRAD para el filtrado de ruido speckle en imágenes digitales presenta una buena calidad de filtrado, pero requiere de un alto coste computacional. Por este motivo, en este proyecto se ha desarrollado e implementado un algoritmo paralelo basado en el algoritmo SRAD para el filtrado de ruido speckle en imágenes médicas.

Se ha implementado una versión secuencial y a partir de ella se han obtenido implementaciones paralelas para máquinas de memoria compartida y máquinas de memoria distribuida, haciendo uso de las librerías de computación de altas prestaciones OpenMP y MPI.

Se han desarrollado experimentos en imágenes médicas OCT y de ultrasonido. Estos experimentos se han desarrollado en tres máquinas paralelas obteniendo un speedup significativo.

Los resultados obtenidos muestran que la reducción en el tiempo de computo de las implementaciones paralelas desarrolladas, facilitan el uso del método SRAD en el filtrado de ruido speckle en imágenes médicas.

7 Líneas de trabajo futuras

Se han estudiado distintas propuestas de ampliaciones futuras con las que mejorar el algoritmo paralelo propuesto. Las mejoras están orientadas a la disminución de tiempos de ejecución y la mejora del resultado final.

La primera propuesta se basa en la implementación del código paralelo mediante técnicas de paralelización mediante unidades de procesamiento gráfico (GPU). Entre las que hemos destacado la implementación del código con CUDA [21], librería y modelo de programación que nos permite hacer uso de la computación de la GPU. CUDA está desarrollado por NVIDIA. El modelo permite realizar las partes secuenciales del algoritmo en la CPU, que está optimizada para el funcionamiento secuencial de las tareas, mientras que las partes paralelizadas se ejecutan en la GPU, la que dispone de multitud de cores que permiten el paralelismo de uso intensivo.

Otra propuesta, es la implementación de un modelo de iteraciones internas locales. En cada nodo del cluster se efectuarían varias iteraciones internas, consiguiendo una reducción de comunicaciones entre los nodos. De esta forma, aceleraríamos las ejecuciones de las versiones MPI e híbrida (MPI+OpenMP).

Como última línea futura de trabajo, se plantea el desarrollo de un filtro paralelo que combine el filtro SRAD con otro filtro, como por ejemplo un filtro guiado, con la idea de mejorar la calidad de las imágenes filtradas.

Bibliografía

[1] J. W. Goodman and R. L. Haupt. Statistical optics. 2015.

[2] G. E. Trahey, S. M. Hubbard, and O. T. von Ramm. Angle independent ultrasonic blood flow detection by frame-to-frame correlation of b-mode images. *Ultrasonics*, 26(5):271–276, 1988. ISSN 0041-624X. doi: https://doi.org/10.1016/0041-624X(88)90016-9. URL `https://www.sciencedirect.com/science/article/pii/0041624X88900169`.

[3] V. Behar, D. Adam, and Z. Friedman. A new method of spatial compounding imaging. *Ultrasonics*, 41(5):377–384, 2003. ISSN 0041-624X. doi: https://doi.org/10.1016/S0041-624X(03)00105-7. URL `https://www.sciencedirect.com/science/article/pii/S0041624X03001057`.

[4] T. Loupas and W. N. Mcdicken. An adaptive weighted median filter for speckle suppression in medical ultrasonic images. *IEEE Transactions on Circuits and Systems*, 36(1):129–135, 1989. URL `www.scopus.com`.

[5] J. Lee. Digital image enhancement and noise filtering by use of local statistics. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-2(2): 165–168, 1980. URL `www.scopus.com`.

[6] V. S. Frost, J. A. Stiles, K. S. Shanmugan, and J. C. Holtzman. A model for radar images and its application to adaptive digital filtering of multiplicative noise. *IEEE*

Transactions on Pattern Analysis and Machine Intelligence, PAMI-4(2):157–166, 1982. URL `www.scopus.com`.

[7] D. Kuan, A. Sawchuk, T. Strand, and P. Chavel. Adaptive restoration of images with speckle. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35 (3):373–383, 1987. doi: 10.1109/TASSP.1987.1165131.

[8] A. Lopes, R. Touzi, and E. Nezry. Adaptive speckle filters and scene heterogeneity. *IEEE Transactions on Geoscience and Remote Sensing*, 28(6):992–1000, 1990. doi: 10.1109/36.62623.

[9] A. Lopes, R. Touzi, and E. Nezry. Adaptive speckle filters and scene heterogeneity. *IEEE Transactions on Geoscience and Remote Sensing*, 28(6):992–1000, 1990. URL `www.scopus.com`.

[10] Y. Yu and S. T. Acton. Speckle reducing anisotropic diffusion. *IEEE Transactions on Image Processing*, 11(11):1260–1270, 2002. doi: 10.1109/TIP.2002.804276.

[11] J. S. Jin, Y. Wang, and J. Hiller. An adaptive nonlinear diffusion algorithm for filtering medical images. *IEEE Transactions on Information Technology in Biomedicine*, 4(4):298–305, 2000. doi: 10.1109/4233.897062.

[12] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2014. ISBN 0262527391.

[13] OpenMP Website, May 2022. URL `https://www.openmp.org/`.

[14] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP McGraw-Hill Inc. 2004*.

[15] Sun shared visualization 1.1 software client administration guide, Jun 2008. URL `https://docs.oracle.com/cd/E19279-01/820-3257-12/n1ge.html`.

[16] Slurm workload manager. URL `https://slurm.schedmd.com/`.

[17] D. Kermany, K. Zhang, and M. Goldbaum. Large dataset of labeled optical coherence tomography (OCT) and chest X-ray images, Jun 2018. URL `https://data.mendeley.com/datasets/rscbjbr9sj/3`.

[18] T. Geertsma. Ultrasound image database collected from Gelderse Vallei Hospital in Ede, the Netherlands. URL `https://www.ultrasoundcases.info/`.

[19] H. Yu, J. Gao, and A. Li. Probability-based non-local means filter for speckle noise suppression in optical coherence tomography images. *Opt. Lett.*, 41(5):994–997, Mar 2016. doi: 10.1364/OL.41.000994. URL `http://opg.optica.org/ol/abstract.cfm?URI=ol-41-5-994`.

[20] S. Ristov, R. Prodan, M. Gusev, and K. Skala. Superlinear speedup in hpc systems: Why and when? In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 889–898, 2016.

[21] CUDA Zone - Library of resources, May 2022. URL `https://developer.nvidia.com/cuda-zone`.