# Paralelización de la transformada wavelet basada en líneas para imágenes de alta resolución

Vicente Galiano[†], Héctor Migallón[†], Manuel P. Malumbres[†] y José Oliver[‡†]

*Resumen*—**La transformada wavelet se ha convertido en una herramienta imprescindible en la codificación de imágenes. Sin embargo, se trata de una herramienta computacionalemnte costosa que además necesita disponer de la imagen en memoria para realizar la transformación. Se han propuesto distintas variantes a la hora de implementar la transformada 2D-DWT con el objeto de reducir su coste computacional como es el caso de la versiones lifting, por bloques y por líneas. En este trabajo proponemos una versión distribuida de la transformada DWT basada en líneas para el procesamiento de imagenes GIS de muy alta resolución. Tras analizar los problemas de dependencias y realizar una serie de experimentos se observa que la versión distribuida de la DWT empleada mejora su rendimiento conforme aumenta el tamaño de la imagen.**

*Palabras clave*—**Transformada wavelet, procesamiento paralelo, uso eficiente de memoria, codificación de imagen.**

## I. Introducción

Los codificadores de imagen basados en la transformada wavelet han alcanzado un gran interés en los últimos años. Esto se debe a dos razones fundamentales: la primera es que el conjunto de coeficientes obtenidos es muy compacto, lo cual facilita la obtención de buenas eficiencias en la compresión de imágenes; y la segunda es la multirresolución de los coeficientes obtenidos. De hecho, debido a la eficiencia en la tasa de compresión, la DWT es la transformada utilizada en el nuevo estándar JPEG 2000 [4]. Sin embargo, algunas de las desventajas de los actuales codificadores wavelet son la complejidad y el alto uso de la memoria de los mismos. Sobre todo si comparamos estos codificadores con los codificadores basados en la transformada discreta del coseno (DCT), como el JPEG original [3].

Debido a que la transformada wavelet necesita gran cantidad de memoria para su cómputo, los codificadores wavelet hacen un uso intensivo de la memoria. Además, algunos codificadores, como los que se pueden encontrar en [4] y [9], usan listas o estructuras complejas de datos adicionales para almacenar el estado o el proceso de la codificación. Esta situación se agrava cuando se trabaja con imágenes de muy alta resolución, como sucede en aplicaciones GIS (Geographical Information Systems) en donde se pueden alcanzar los cientos de millones de pixels por imagen. En este tipo de aplicaciones la resolución de las imagenes viene determinada por la utilización de cámaras de muy alta resolución (hasta 1 pixel/m) instaladas en aviones o satélites.

En este trabajo tratamos de facilitar el procesamiento de dichas imágenes con un consumo moderado de recursos de computo en un tiempo reducido. Aunque la versión paralela que proporcionamos sólo hace referencia a la transformada wavelet 2D-DWT, éste sería el primer paso hacia la obtención de un compresor eficiente de imágenes de muy alta resolución.

En la sección II veremos las características del algoritmo de referencia, en la sección III presentaremos la versión paralela de este algoritmo, y por último, en la sección IV mostraremos los resultados experimentales obtenidos, en términos de eficiencia del propio algoritmo y en términos de ruido respecto a la versión secuencial.

## II. Algoritmo recursivo basado en líneas para la transformada wavelet

El algoritmo recursivo basado en líneas para la transformada wavelet con el que vamos a trabajar, ofrece una solución para hacer un uso eficiente de la memoria en la obtención de la DWT. Además, este algoritmo realiza un cálculo eficiente de la DWT. En la transformada clásica DWT se realiza una descomposición de Mallat [6]. En esta descomposición, la imagen es tratada, inicialmente, fila por fila y posteriormente columna por columna para cada nivel de descomposición. Por lo tanto, es necesario almacenar toda la imagen en memoria. En el algoritmo propuesto se hace uso de una estrategia basada en líneas para obtener la transformada wavelet, dado que no es necesario disponer de toda la imagen en memoria se hace un uso más eficiente de ésta, además, se busca tener disponibles los coeficientes wavelet tan pronto como sean calculados.

Este concepto fue utilizado inicialmente en [11] al buscar una reducción de las necesidades de memoria en el desarrollo de la 1D DWT. Posteriormente, en [2], se propone uno de los primeros trabajos realizados en procesamiento de imagen con el objetivo de reducir el consumo de memoria. En este trabajo se propone un algoritmo que reduce el consumo de memoria reordenando los *bitstreams*, de forma que agrupa los coeficientes de diferentes subbandas correspondientes a una misma zona de la imagen. De esta forma, el decodificador puede realizar la transformada inversa DWT de un subconjunto de coeficientes y generar algunas líneas de la imagen. Una vez decodificadas estas líneas, la memoria utilizada puede ser liberada y utilizarse para leer los coeficientes de un nuevo grupo de líneas. Basado en este algoritmo por líneas, el algoritmo presentado en [1] reduce el uso de la memoria tanto en la transformada directa como en la transformada inversa, mientras que en [2] solo se reducía en la decodificación. Además, en [1] también se mejoran otros aspectos relacionados con el orden de los datos que permite un uso eficiente de la memoria tanto en la codifi-

[†] Departamento de Física y Arquitectura de Computadores, Universidad Miguel Hernández, 03202 Elche, Alicante, vgaliano,hmigallon,mels @umh.es

[‡] Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia, 46022 Valencia, joliver@disca.upv.es

cación como en la decodificación. Sin embargo, en esta versión 2D, los autores no proponen un algoritmo que implemente su propuesta, siendo esta implementación compleja al no estar concretado completamente. El algoritmo recursivo basado en líneas para la transformada wavelet puede desarrollarse completamente en cualquier lenguaje de programación y ha sido usado, por ejemplo, para el codificador de imagen de [8].

Vamos a ver con más detalle la estrategia basada en líneas para la obtención de la transformada wavelet. En primer lugar hay que remarcar que en memoria sólo están aquellas líneas que se están utilizando, permaneciendo en memoria mientras se calculan los coeficientes wavelet asociados. Una vez finalizada la codificación de estos coeficientes se libera la memoria, el resto de líneas de la imagen irá entrando en la etapa de codificación, y por tanto en la memoria, en el momento que son estrictamente necesarias.

En el primer nivel de descomposición, el algoritmo recibe secuencialmente las líneas de la imagen. Para cada línea de entrada se realiza una transformada wavelet de primer nivel 1D, dividiéndola en dos bandas de frecuencia L (Low) y H (High). Esta es la versión más pequeña de esta línea. Esta transformada se almacena en un buffer asociado al primer nivel de descomposición. El tamaño de este buffer debe ser el necesario para almacenar $2N+1$ líneas, siendo N el mayor orden de los filtros utilizados. Se trabajará únicamente con filtros impares, ya que estos presentan una mayor eficiencia al comprimir, y la compresión será una de las utilidades importantes, no obstante los análisis presentados se pueden extender al uso de filtros de orden par.

Cuando el buffer tiene las líneas suficientes para realizar la transformada wavelet a una columna, el proceso de convolución vertical se realiza dos veces, una vez con un filtro paso–bajo y otra vez con un filtro paso–alto. Tras esta operación se obtiene la primera línea de coeficientes de las subbandas wavelet $HL_1$, $LH_1$ y $HH_1$, y la primera línea de la subbanda $LL_1$. En este momento se puede codificar y liberar la primera línea de las subbandas wavelet. Sin embargo, la primera línea de la subbanda $LL_1$ no necesita ser codificada ya que va a ser la línea de entrada para el siguiente nivel de descomposición.

Por otra parte, una vez que las líneas del buffer del primer nivel han sido codificadas, dos líneas son descartadas y dos líneas de imagen entran en el buffer. Una vez el buffer ha sido actualizado se repite todo el proceso para la obtención de nuevas líneas de coeficientes.

En el segundo nivel el buffer se llena con las líneas $LL_1$ provenientes del primer nivel. Se procede de forma análoga a como se procedía en el primer nivel, de tal forma que cuando se han calculado las líneas de coeficientes de las subbandas wavelet del segundo nivel, las líneas $LL_2$ pasan al tercer nivel. En la figura 1 podemos observar gráficamente la estructura comentada. El proceso descrito se puede repetir hasta el nivel de descomposición deseado *nlevel*. Una vez se ha llegado a ese nivel se calculan las líneas de la última subbanda ($LL_{nlevel}$), siendo ésta la última parte de la descomposición wavelet.

Este algoritmo es muy sencillo desde un punto de vista lógico, pero la correcta sincronización de los buffers

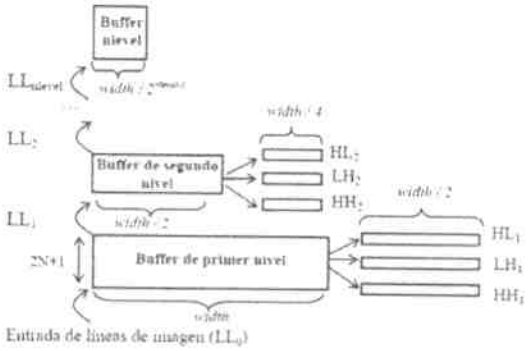


Fig. 1. Transformada wavelet basada en líneas.

complica su implementación. Hasta que un buffer no está completamente lleno con líneas que proceden del buffer del nivel anterior no se puede calcular líneas de coeficientes, por tanto cada buffer comienza a trabajar en distintos momentos, sus retrasos, por tanto, son diferentes. Además, los buffers se actualizan en intervalos diferentes de tiempo, en función de su nivel. Manejar diferentes buffers con diferentes retrasos y diferentes ritmos de trabajo es una tarea complicada. A continuación describimos el algoritmo recursivo que establece la comunicación entre buffers.

En [7] se presenta un algoritmo recursivo de propósito general que resolvía los problemas planteados en [1]. Dado que este algoritmo es eficiente y necesita poca memoria, será el utilizado en nuestro caso. Para resolver el problema de la sincronización, el algoritmo basado en líneas define una función recursiva que calcula las líneas de la subbanda de baja frecuencia LL de un determinado nivel, utilizando las líneas de la subbanda superior. La primera vez que se llama a dicha función ésta devuelve la primera línea del nivel especificado, la segunda vez devuelve la segunda línea, y así sucesivamente. En el momento que no hay más líneas para el nivel especificado la función devuelve un identificador de final de línea. Mientras se calcula la enésima línea de una subbbanda $LL_{level}$, las correspondientes enésimas líneas de las subbandas HL, LH y HH del mismo nivel también se calculan.

Para calcular la transformada completa se solicitan, inicialmente, líneas de la subbanda LL del último nivel, es decir de *nlevel*. Como ya se ha mencionado y se puede ver en la figura 1, el buffer de nivel *nlevel* ha de llenarse con líneas procedentes del nivel *nlevel-1* antes de poder generar líneas. Para conseguir esas líneas necesarias la función se llama recursivamente hasta alcanzar el nivel cero. En este momento, la función lee una nueva línea del sistema de entrada/salida. Hay que remarcar que, pese a que se realiza la transformada wavelet directa, la recursión va del nivel *nlevel* al nivel 0.

Veamos con más detalle como trabaja la función recursiva mencionada. Antes de calcular una línea wavelet han de prepararse todos los buffers con las líneas necesarias. Al comenzar la función ha de llamarse para cada nivel. En el caso de que los buffers están vacíos, la mitad superior de cada buffer (de N a 2N) se llena recursivamente con líneas del buffer del nivel anterior. Cada línea

recibida se codifica mediante la transformada wavelet 1D antes de almacenarla. Una vez la mitad superior del buffer está completa, la mitad inferior se rellena por extensión simétrica, es decir la línea N + 1 se copia en la línea N − 1 y así sucesivamente hasta que la línea 2N se copia en la línea 0. Si los buffers no están vacíos la actualización es muy sencilla, desplazamos en una unidad el buffer, por tanto la última línea (almacenada en la posición 0) se descarta y se introduce una nueva línea en la última posición 2N mediante una llamada recursiva. Esta operación se realiza dos veces. Si no hay más líneas en el buffer anterior, la llamada recursiva devuelve un parámetro de fin de línea. Hay que remarcar que en este momento el cálculo en el nivel dado ha finalizado, no obstante se necesita ir llenando el buffer para el cálculo de los coeficientes del resto de niveles, este llenado se hace también por extensión simétrica. Estos conceptos son muy importantes en el desarrollo del algoritmo paralelo, como se verá en la sección IV.

Una vez que el buffer se ha rellenado o actualizado, se aplica a cada columna del buffer la transformada 1D-DWT. El resultado de esta convolución es una línea de coeficientes en cada subbanda wavelet del nivel dado y una línea LL.

La función recursiva desarrollada dispone de dos condiciones de parada. La primera condición se da cuando todas las líneas de un determinado nivel se han leído, esta condición puede darse por una igualdad entre el número de líneas leídas y el número de líneas que han de leerse en un nivel dado, pero también puede darse por la propagación de un nivel a otro de la condición de fin de línea. La segunda condición se da cuando se alcanza el nivel 0 y no son necesarias más llamadas recursivas hasta que no sea necesario leer una nueva línea de la imagen.

*Función 1:* GetLLlineBwd
Parámetro de entrada: *level*
Si *level* = 0 devuelve una línea de la imagen.
Si no hay mas líneas en el nivel *level* devuelve EOL (End Of Line).
Recursivo:
1. Preparación inicial del buffer:
   Si el buffer está vacío la mitad superior se llena con líneas LL del buffer anterior.
   Llamada recursiva para calcular líneas decrementando *level*.
   Aplicar 1D DWT a cada línea e introducirla en el buffer.
   Si la mitad superior está llena rellenar con extensión simétrica.
2. Obtención de coeficientes:
   Aplicar 1D-DWT a cada columna del buffer.
   Obtener una línea de cada subbanda (LH, HL y HH) y otra de baja frecuencia LL.

Utilizando la función (1), la transformada DWT de un nivel dado se puede calcular fácilmente si llamamos a dicha función pasándole el parámetro del nivel deseado para la transformada wavelet. De esta forma la transformada wavelet se realizaría mediante la siguiente función:

*Función 2:* LowMemoryUsageWT
Parámetro de entrada: *nlevel*
Vaciar todos los buffers para todos los niveles.
Mientras haya LL líneas que calcular en el último nivel (*nlevel*):
Obtener una línea mediante la función (1).

Hay que remarcar que la función (2) realiza un cálculo de los coeficientes de la transformada wavelet con un sistema basado en árboles. Esta estructura es muy interesante para codificadores de imagen basados en árboles.

## III. Algoritmo paralelo recursivo basado en líneas para la transformada wavelet

Hay que remarcar que la versión paralela de un algoritmo clásico para realizar la transformada wavelet es un trabajo complicado de abordar. Hay que tener en cuenta que los algoritmos clásicos para la transformada wavelet necesitan disponer de la imagen completa en memoria. Esto es un indicador claro que los algoritmos clásicos disponen de un pobre paralelismo inherente, lo que, lógicamente, complica el tratamiento paralelo del mismo. Las dependencias que nos encontramos en este tipo de algoritmos, tienen la particularidad especial de ser dependencias de datos verdaderas algorítmicamente extensas, es decir, los resultados de los cálculos que se van realizando son utilizados frecuentemente en el transcurso del algoritmo. Teniendo en cuenta esta circunstancia, se puede deducir fácilmente que la paralelización que se podría realizar tendría unos requisitos especiales en el hardware utilizado, es decir, estaríamos hablando de una versión paralela para un sistema multiprocesador homogéneo de memoria compartida. Aun utilizando un multiprocesador de memoria compartida, el trabajo de sincronización entre procesadores sería una tarea compleja que podría provocar una baja eficiencia del algoritmo. Por otra parte, esta sincronización con procesadores heterogéneos podría provocar una disminución mayor de la eficiencia.

El algoritmo que hemos descrito en la sección II dispone de dos características importantes a la hora de considerar la realización de una versión paralela. Una de estas características es que es un algoritmo basado en líneas y que realiza un uso eficiente de la memoria, lo cual nos debe hacer pensar que la dependencia de datos ha de ser menor. La segunda característica es que es un algoritmo con muy buenas prestaciones en términos de rapidez.

La primera aproximación a la versión paralela de este algoritmo consiste en dividir la imagen en $p$ bloques de $n_i$ líneas, siendo $p$ igual al número de procesadores y $\sum_{i=1}^{p} n_i = height$ siendo $height$ el número de líneas de la imagen. Cada uno de estos bloques se asigna a un procesador diferente, de tal manera que cada procesador calcula los coeficientes asociados a su bloque de imagen. Este cálculo se realiza tal y como se ha explicado en la sección II. Es lógico pensar, que si trabajamos con un algoritmo basado en líneas, la segmentación de la imagen debe hacerse en bloques de un número determinado de líneas y no hacer una división en la que partes diferentes de una misma línea se asignen a procesadores diferentes, como podría ser una segmentación en cuadrantes.

Analizando el algoritmo paralelo segmentado

niendo en cuenta como se realiza el llenado inicial de los buffers (punto 1 de la función (1)), se puede deducir que el algoritmo paralelo tal y como lo hemos presentando no obtendrá valores correctos para los coeficientes correspondientes a las primeras líneas asignadas a cada uno de los procesadores, excepto el procesador al que se le asigna el bloque inicial de la imagen. Esto es debido a que el algoritmo, cuando empieza a trabajar, hace uso de la extensión simétrica para el llenado inicial de la mitad inferior del buffer. El algoritmo original secuencial sólo realizaría esta extensión simétrica en las primeras líneas de la imagen, mientras que en la versión paralela la extensión simétrica se realizará en todas las primeras líneas de cada bloque.

Por otra parte, recordemos que en el algoritmo original secuencial cuando se finaliza el cálculo de los coeficientes de un nivel, se necesita seguir llenando el buffer para el cálculo de los coeficientes del resto de niveles, este llenado se hace también por extensión simétrica. Por tanto, podemos encontrar una analogía entre el error que se comete en el cálculo de los coeficientes correspondientes a las primeras líneas de cada procesador, excepto el primero, y el error que se comete en los coeficientes correspondientes a las últimas líneas de cada procesador excepto el último.

En la sección IV veremos los detalles de la implementación y el comportamiento del algoritmo desarrollado.

## IV. Resultados Numéricos

Para realizar los experimentos numéricos hemos utilizado un multiprocesador de memoria distribuida, en particular, un cluster de 6 pentiums IV a 2 44 Ghz, conectados mediante un switch Gigabit Ethernet. El manejo del entorno paralelo se realiza mediante la librería MPI [5]. El lenguaje de programación utilizado ha sido C++, y hemos utilizado los compiladores *gcc* y *mpicxx*.

Como hemos comentado, distribuimos la imagen a los distintos procesadores en bloques de un número determinado de líneas, a la hora de hacer esta segmentación es necesario tener en cuenta la implementación detallada en la sección II, en particular la función (1) implica la necesidad de trabajar con bloques de tamaños tal que $mod(n_i\ 2^{nlevel}) = 0$. En nuestros experimentos trabajaremos siempre con $nlevel = 6$, por tanto, el número de líneas de cada bloque ha de ser múltiplo de 64.

El tamaño de bloque asociado a cada procesador debe elegirse para balancear correctamente la carga de trabajo, es decir si los nodos del multiprocesador son homogéneos, hemos de asignar tamaños de bloques iguales o muy parecidos a cada procesador. La segunda decisión es el modo de distribuir los datos a cada procesador, que también es un factor importante en el algoritmo paralelo. En este caso hemos optado por no distribuir las líneas de imagen a cada procesador, sino que lo que haremos será enviarle a cada procesador los datos necesarios para acceder al fichero fuente donde se almacena la imagen, por tanto el trabajo de envío de líneas a cada procesador, se hará en sentido inverso, es decir, cada procesador intentará leer del fichero fuente como si éste fuera local, pero realmente será el sistema de compartición de archivos NFS (Network File System) quien se encargue de enviar dicha línea a través de la red. En el caso de la escritura de resultados, cambiaremos ligeramente el modo de trabajo, haremos uso de NFS para que los resultados se envíen al sistema de almacenamiento del procesador padre, no obstante cada procesador escribirá en ficheros diferentes. Esto ha sido una decisión de diseño inicial inducida por las aplicaciones futuras y para analizar el comportamiento del algoritmo paralelo en circunstancias adversas. Otras opciones serían: hacer uso de NFS para enviar los resultados al procesador padre y que todos los nodos trabajen sobre los mismos ficheros de salida; y otra podría ser no hacer uso de NFS y que cada nodo guarde los resultados en su sistema de almacenamiento local.

En todos los experimientos se ha utilizado el banco de filtros B7/9. Hemos trabajado con una de las imágenes típicas utilizadas para análisis, conocida como *lena*, de tamaño 512x512. La figura 2(a) muestra la imagen *lena*, en la cual hemos obtenido su transformada wavelet directa con 1 procesador (el resultado es el mismo que la utilización del código secuencial) y después hemos reconstruido la imagen con la transformada inversa original. En la figura 2(b) presentamos el resultado del mismo proceso, pero el cálculo de la transformada directa se ha realizado con 3 procesadores. Las líneas asignadas a cada procesador son $n_1 = 256$, $n_2 = 128$ y $n_3 = 128$. Hemos asignado mayor tamaño de bloque al procesador 1 dado que para este procesador los ficheros son locales y no hace uso del sistema NFS, por lo tanto perderá menos tiempo en los procesos de lectura y escritura de ficheros. Hay que tener en cuenta que ésta es una optimización de adaptación al multiprocesador utilizado en estos experimentos. Podemos observar, gráficamente, que el primer procesador no comete error hasta llegar a las líneas finales de su bloque, mientras que los procesadores 2 y 3 comenten errores en las fronteras inferior y superior de su imagen. De hecho la imagen reconstruida no es una buena representación de la imagen.

El efecto que hemos visto en las líneas iniciales de cada bloque se debe a que cada bloque inicia su actividad llenando el buffer mediante extensión simétrica. En el algoritmo original esta situación solo se da en el inicio del proceso, por eso en las líneas iniciales del primer procesador no hay problemas, ya que el algoritmo secuencial realiza las mismas acciones que realiza el primer procesador. Una primera solución es que cada procesador no empiece a computar en su primera línea, sino que comience un determinado número de líneas antes, las cuales pertenecen a otro procesador, y por tanto, el cómputo de estas líneas se duplica. Hay que tener en cuenta que observamos errores tanto en puntos superiores como inferiores de la frontera. Por tanto y dado que cuando se finaliza un bloque, y no solo al inicio del mismo, se realiza extensión simétrica para el cálculo de coeficientes, lo que haremos será solapar superiormente e inferiormente. Es decir, cada procesador empezará a calcular los coeficientes antes de lo que realmente debería empezar y acabará en una línea posterior que no le pertenece. En la figura 3 presentamos la imagen reconstruida con un solapamiento de 64 líneas tanto superior como inferiormente, y se observa que el error sometido

(a) Secuecial, $p = 1$.

(b) Paralelo, $p = 3$.

Fig. 2. Comparación de resultados secuencial–paralelo sin solapamiento.

es sensiblemente inferior, pero que sigue existiendo.

La solución que se nos presenta es aumentar el número de líneas de solapamiento, en la figura 4 se muestra la imagen reconstruida solapando 128 líneas utilizando 3 procesadores. En esta figura prácticamente son inapreciables las diferencias con la imagen reconstruida del algoritmo secuencial (figura2(a)). Para mejorar todavía más la imagen deberíamos aumentar el tamaño del solapamiento, siempre manteniendo la estructura de bloques comentada. Hay que remarcar que al solapar 256 líneas tanto superior como inferiormente, los coeficientes obtenidos son exactamente los mismos que los obtenidos por el algoritmo secuencial. Esto se ha probado para otro tipo de imágenes, y para imágenes más grandes manteniéndose el comportamiento comentado.

En la figura 5 podemos ver el MSE (*mean squared error*) y el PSNR (*peak signal-to-noise ratio*) asociado a

Fig. 3. Imagen reconstruida solapando superior e inferiormente 64 líneas. 3 procesadores.

Fig. 4. Imagen reconstruida solapando superior e inferiormente 128 líneas. 3 procesadores.

las diferentes implementaciones del algoritmo que hemos visto, y en las cuales se varía el nivel de solapamiento. En esta figura podemos ver que el PSNR aumenta al trabajar con mayor solapamiento. Si nos fijamos en el valor de PSNR para la versión sin solapamiento, es un valor bueno en el ámbito de los compresores de imágenes y similares. En nuestro caso debemos fijarnos más en el MSE que muestra mejor los resultados experimentales. Esto es debido a que el PSNR nos da una idea del valor medio del error cometido, y en nuestro caso no refleja la realidad porque el error cometido se concentra en determinadas zonas de la imagen. No obstante, como hemos mencionado anteriormente, llega un momento en el que se alcanza una completa igualdad entre los resultados obtenidos con el algoritmo secuencial y el paralelo. En este momento, ampliar el nivel de solapamiento no tiene ningún efecto positivo, provocando un empeoramiento en el rendiemiento del algoritmo paralelo. El solapamiento que asegura que el algoritmo paralelo obtiene resultados idénticos al algoritmo secuen-

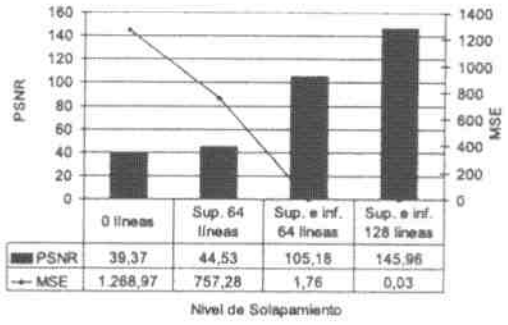


Fig. 5. Análisis PSNR y MSE. Tamaño de imagen 512x512 bytes. 3 procesadores.

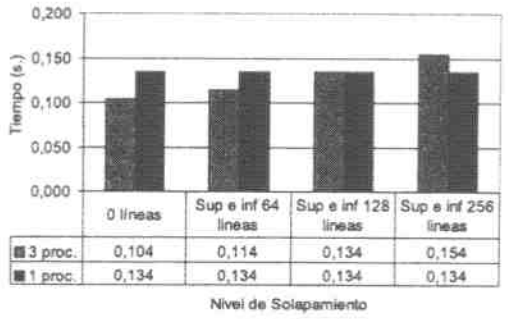


Fig. 6. Comparación con algoritmo secuecial. Tamaño de imagen 1024x1024 bytes. 3 procesadores

cial es $2^{nlevel}xN$ siendo N el número de orden del filtro utilizado. Queremos remarcar que el procesamiento paralelo lo aplicaremos para imágenes muy grandes, y en ese caso el solapamiento necesario provocará un incremento del coste computacional totalmente asumible. De hecho, en la figura 6 podemos observar que al aumentar el solapamiento el algoritmo paralelo necesita más tiempo para su ejecución, siendo el algoritmo paralelo peor que el secuencial cuando usamos 256 líneas de solapamiento. Hay que tener en cuenta que en la figura 6 se ha trabajado con una imagen pequeña de 1024x1024 bytes, con 3 procesadores, y la asignación ha sido tal que $n_1 = 384$ y $n_{2,3} = 320$. Si aumentamos el tamaño de la imagen el rendimiento va aumentando progresivamente, esto lo podemos observar en la figura 7, en la cual hemos trabajado con imágenes más grandes, de 2048x2048 pixels y 4096x4096 pixels, asignando $n_1 = 768$ y $n_{2,3} = 640$ líneas para la primera imagen y $n_1 = 1408$ y $n_{2,3} = 1344$ líneas para la segunda, obteniendo una mejora del 4% y del 33% respectivamente. Además, en [3] podemos ver que, por ejemplo para una imagen de 4 megapixels, el algoritmo basado en líneas obtiene unas mejoras de más del 60% respecto a la transformada clásica wavelet. En ambos casos hemos solapado 256 líneas. A medida que la imagen a tratar es mayor el comportamiento del algoritmo paralelo es mucho mejor.

## V. Conclusiones

Hemos presentado un algoritmo paralelo para la obtención de la transformada wavelet. El algoritmo desarrollado es un algoritmo rápido con un uso eficiente de la memoria y sin pérdidas en la reconstrucción de

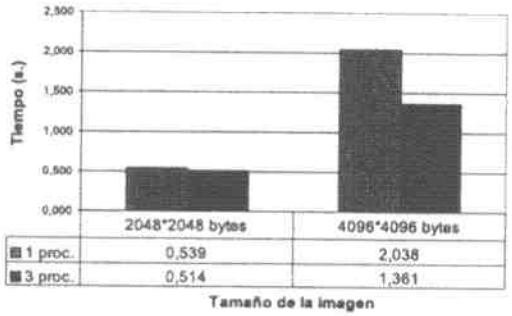


Fig. 7. Comparación con algoritmo secuencial. Solapamiento de 256 líneas. 3 procesadores

la imagen. Este es un algoritmo con muy alta eficiencia en multiprocesadores de memoria compartida, y un muy buen algoritmo para sistemas multiprocesador de memoria físicamente distribuida. Por tanto es una herramienta muy útil para el desarrollo de codificadores y compresores de imágenes paralelos, aplicables a sistemas con requisitos de tiempo real. Además, este algoritmo obtiene buenos comportamientos con imágenes relativamente pequeñas, aumentando progresivamente el rendimiento al aumentar el tamaño de la imagen tratada. En este trabajo se han realizado los primeros para el desarrollo de un sistema de codificación eficiente de imágenes de muy alta resolución con bajas demandas de recursos de computo. Los siguientes pasos, son añadir la codificación por líneas para completar el codificador usando imagenes GIS reales y comparando su rendimiento con otros codificadores tradicionales. Finalmente se propondrá un sistema de decodificación en tiempo real que permita reproducir la información de la imagen a distintas escalas basándose en las propiedades multiresolución que intrinsecamente tiene la transformada DWT.

## Referencias

[1] Christos Chrysafis, and Antonio Ortega, *Line-based, reduced memory, wavelet image compression*, IEEE Transactions on Image Processing, March 2000..

[2] Pamela Cosman and Kenneth Zeger, *Memory constrained wavelet-based image coding*, roc. UCSD Conf. Wireless Communications, March 1998.

[3] ISO/IEC 10918-1/ITU-T Recommendation T.81, *Digital Compression and Coding of Continuous-Tone Still Image, 1992.*

[4] ISO/IEC 15444-1, *JPEG 2000 image coding system, 2000.*

[5] William Gropp, Ewing Lusk, and Anthony Skjellum, *Using MPI: Portable Parallel Programming*, MIT Press, Cambridge, MA, 1994.

[6] Stephane G. Mallat, *A theory for multiresolution signal decomposition*, EEE Transactions on Pattern Analysis and Machine Intelligence, July 1989.

[7] José Oliver and Manuel P. Malumbres, *A fast wavelet transform for image coding with low memory consumption*, 24th Picture Coding Symposium, December 2004.

[8] José Oliver and Manuel P. Malumbres, *Fast tree-based wavelet image coding with efficient use of memory*, Visual Communications and Image Processing, July 2004.

[9] Amir Said and William A. Pearlman, *A new, fast, and efficient image codec based on set partitioning in hierarchical trees*, EEE Trans. on Circuits and Systems for Video Technology, June 1996.

[10] Jerome M. Shapiro, *Embedded Image Coding Using Zerotrees of Wavelet Coefficients*, IEEE Transactions on Signal Processing, vol. 41, pp. 3445-3462, Dec. 1993.

[11] Mohan Vishwanath, *The recursive pyramid algorithm for the discrete wavelet transform*, IEEE Transactions on Signal Processing, March 1994.