Codificador de vídeo basado en Wavelet 3D usando OpenMP y Pthreads

Ricardo Fernández, José M. García, Gregorio Bernabé y Manuel E. Acacio

Resumen— **En este artículo se estudian dos alternativas para la implementación de un algoritmo paralelo de codificación de vídeo basado en la Wavelet 3D usando OpenMP y Pthreads desde el punto de vista de la velocidad de ejecución y la facilidad de implementación y mantenibilidad del código obtenido. Se parte de un codificador 3D-FWT secuencial y se paraleliza usando OpenMP y Pthreads, llegando a la conclusión de que es posible obtener con OpenMP una velocidad de ejecución casi óptima sin tener que sacrificar la mantenibilidad del código usando Pthreads.**

Palabras clave— **OpenMP, Pthreads, Transformada Wavelet, Codificación de Vídeo.**

I. Introducción

LAS tendencias actuales en arquitectura de computadores buscan explotar cada vez más el paralelismo a todos los niveles. Cada vez resulta más normal que las estaciones de trabajo sean multiprocesadores de memoria compartida o dispongan de un procesador capaz de ejecutar varios hilos a la vez mediante *Simultaneous Multithreading* (SMT [1] [2]). El aprovechamiento eficaz de estas arquitecturas requiere el uso de varios hilos o procesos, lo que obliga a replantearse la implementación de muchos algoritmos.

Existen varios enfoques a la hora de obtener algoritmos paralelos a partir de otros secuenciales. La paralelización automática ([3] [4]) resulta atractiva y prometedora para algunas aplicaciones, pero es incapaz de obtener buenos resultados en muchas otras. Cuando se requiere un conocimiento de alto nivel de la aplicación es necesario recurrir a la intervención manual para paralelizar el código directamente.

Las aplicaciones multimedia tienen una importancia cada vez mayor en diversas áreas. Un ejemplo es el vídeo médico, cuyos requisitos de calidad son muy altos y existen regulaciones que obligan a almacenarlo durante un tiempo determinado, haciendo necesaria una compresión eficaz. Los requisitos de calidad hacen que suelan usarse técnicas de compresión sin pérdida (JPEG-LS [5]), pero esto no resulta práctico debido a las bajas tasas de compresión posibles con ellas.

Por otro lado, la transformada Wavelet tridimensional permite codificar vídeo con gran calidad y grandes tasas de compresión, haciendo atractivo el uso de un codificador basado en ella [6] [7] y obteniendo mejores resultados que otros algoritmos de propósito más general (MPEG [8], MPEG-4 [9] [10]). Uno de los principales problemas al trabajar con la secuencia de vídeo como una señal tridimensional es que se tiene que manejar una gran cantidad de datos y los accesos a memoria ralentizan la ejecución del algoritmo.

En el caso de la paralelización de un codificador de vídeo los métodos automáticos a nuestro alcance no obtienen ningún resultado. Es necesario la intervención manual, especialmente para aprovechar la tecnología HyperThreading [11].

La paralelización manual añade una complejidad considerable al desarrollo del software. Sería deseable que ese aumento de complejidad fuera el mínimo posible. Existen varias alternativas a la hora de implementar un algoritmo paralelo, cada una con distinto grado de dificultad de uso, mantenibilidad y flexibilidad. En este artículo se pretende evaluar la aplicación de dos de esas alternativas, OpenMP [12] y Pthreads [13], a la implementación de un codificador de vídeo paralelo usando la transformada Wavelet 3D desde el punto de vista de la velocidad del programa resultante, dificultad de implementación, mantenibilidad y reusabilidad del código obtenido.

En este artículo se presentan cuatro implementaciones del codificador paralelo descrito en [11] y se compara su tiempo de ejecución, la dificultad de implementación y mantenibilidad del programa resultante. En la sección II se presentan los fundamentos de la transformada Wavelet 3D y de las tecnologías a utilizar para la implementación (OpenMP y Pthreads). La sección III discute el trabajo previo realizado en el codificador secuencial y las estrategias de paralelización estudiadas. La sección IV expone las implementaciones realizadas de la paralelización según las cuatro estrategias de implementación. Finalmente, la sección V estudia el tiempo de ejecución de cada implementación y los compara con la implementación secuencial, y la sección VI presenta las conclusiones del trabajo.

II. Fundamentos y trabajo relacionado

A. OpenMP

OpenMP [12] es una especificación que permite la implementación de programas portables en arquitecturas de multiprocesadores de memoria compartida usando C, C++ o FORTRAN.

La programación usando OpenMP se basa en el uso de directivas que indican al compilador qué secciones debe paralelizar y cómo. Estas directivas permiten crear secciones paralelas, indicar bucles paralelos y definir secciones críticas. Cuando se define una sección o bucle paralelo el programador debe especificar qué variables son privadas a cada hilo, compartidas o usadas para reducciones. El programador

Departamento de Ingenieria y Tecnologia de Computadores-Facultad de Informatica. Universidad de Murcia. Campus de Espinardo - 30080 Murcia (España). Correo electrónico: {r.fernandez,jmgarcia,gbernabe,meacacio}@ditec.um.es

no tiene que especificar si no quiere cuántos hilos se usarán o cómo se planificará la ejecución de un bucle paralelo, ya que OpenMP incluye diversas políticas aplicables en tiempo de ejecución.

OpenMP permite una paralelización incremental de programas, por lo que resulta especialmente adecuado para añadir paralelismo a programas escritos de forma secuencial. En la mayoría de los casos es posible tener el mismo código para versiones secuenciales y paralelas del programa, tan sólo es necesario ignorar las directivas OpenMP para compilar el programa secuencial. Esto es especialmente útil en las fases de desarrollo del programa para permitir una depuración más fácil.

B. Pthreads

Pthreads (POSIX threads, [13]) es un API muy portable y extendido para la programación de multiprocesadores con memoria compartida. Este API es de más bajo nivel que OpenMP por lo que permite un mayor control de cómo se realiza la concurrencia a costa de una mayor dificultad de uso.

El modelo de programación de Pthreads se basa en la creación de hilos mediante llamadas a funciones de la librería. El programador debe crear explícitamente todos los hilos y encargarse de todas las labores de sincronización necesarias.

Pthreads incluye un rico conjunto de primitivas de sincronización, que incluye cerrojos (mutex), semáforos y variables de condición.

C. SMT e HyperThreading

Con SMT [1] [2], un sólo procesador es capaz de ejecutar simultáneamente dos hilos, compartiendo la mayoría de los recursos hardware tales como las cachés, unidades de ejecución, predictores de salto, etc. y duplicando aquellos recursos necesarios para almacenar el estado de los procesos. De esta forma, un sólo procesador físico parece al sistema operativo y las aplicaciones como dos procesadores virtuales independientes.

La compartición de recursos permite un mejor aprovechamiento de los recursos del procesador, permitiendo que un hilo se ejecute mientras otro está detenido por un fallo de caché o de predicción de saltos.

Intel® ha introducido una implementación de SMT denominada HyperThreading [14] en sus nuevos procesadores (Xeon™, Pentium™IV) obteniendo mejoras en el tiempo de ejecución de cerca del 30% en algunos casos [15].

D. Transformada Wavelet Rápida

La transformada Wavelet descompone la señal extrayendo la información presente a varias resoluciones [16]. La transformada Wavelet rápida se implementa usando un par de filtros QMF (Quadrature Mirror Filter). Para calcular la transformada de una señal discreta se calcula la convolución de la señal con cada filtro y se submuestrea por 2. Uno de los filtros, H, es un filtro pasa baja que extrae la información de grano grueso de la señal, mientras que el otro, G, es un filtro pasa alta que se encarga de extraer los detalles de la señal. De esta forma obtenemos dos señales, *low* y *high*, con la mitad de muestras de la original cada una.

La transformada se puede aplicar varias veces a la parte de grano grueso, lo que permite conseguir mejores tasas de compresión sacrificando la calidad de la señal reconstruida.

Para generalizar la transformada Wavelet a dos o más dimensiones se aplica sucesivamente la transformada en cada dimensión. Por ejemplo, en el caso de dos dimensiones se aplica primero la transformada Wavelet unidimensional a cada una de las filas y después se aplica a cada una de las columnas de la imagen resultante.

III. Trabajo Previo

Estamos interesados en aplicar el algoritmo rápido de la transformada Wavelet 3D (3D-FWT) a la compresión de vídeo médico de tamaño 512×512 en escala de grises (8 bits por pixel) y en secuencias de 64 frames. A partir de un trabajo anterior [7], hemos obtenido que los mejores parámetros son usar como Wavelet madre la Daubechies de 4 coeficientes y realizar dos pasadas en cada una de las dimensiones.

En el codificador que hemos desarrollado, después de la transformada Wavelet se aplica un paso de umbralización en el que se descartan los valores cuyo valor absoluto es menor que una constante determinada. Tras la umbralización se realiza el paso más costoso en cuanto a tiempo de CPU del codificador: la cuantización, en la que los coeficientes de la transformada se convierten en enteros sin signo. Por último, se realiza una codificación entrópica utilizando codificación run-length 3D y se comprime el resultado usando un compresor aritmético.

En trabajos anteriores [7] [17] [18] se han desarrollado varias técnicas que mejoran el tiempo de ejecución del codificador secuencial. Estas técnicas incluyen una estrategia de *blocking rectangular con solapamiento y reuso de operaciones*, vectorización manual usando instrucciones SSE y vectorización por columnas para aprovechar la localidad en el cálculo de la dimensión Y. Por tanto, la versión secuencial de la que partimos para vectorizar está tan optimizada como nos ha sido posible sin recurrir a la paralelización.

IV. Paralelización

En [11] se presentan dos formas de paralelizar la transformada Wavelet enfocadas especialmente a aprovechar la tecnología HyperThreading: usando descomposición por datos y usando descomposición funcional.

La descomposición por datos aplica simultáneamente la transformada a dos bloques de frames independientes de la secuencia. Esta estrategia permite un buen balanceo de la carga y tiene muy poca comunicación, pero el conjunto de trabajo aumenta al doble y ambos hilos realizan funciones

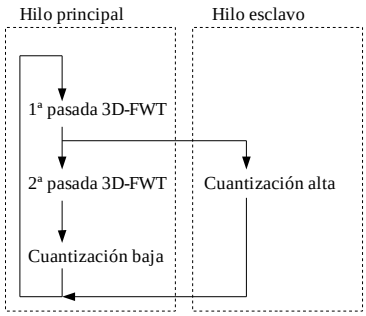

Fig. 1. Primer esquema de paralelización.

similares, lo que causa contención en la unidades funcionales del procesador y hace dicha técnica no apropiada para arquitecturas HyperThreading.

La descomposición funcional aprovecha fases independientes del codificador para realizarlas en paralelo. Se pueden realizar simultáneamente las fases de la transformada y la cuantización del codificador, pero la transformada Wavelet tarda menos de la mitad en ejecutarse que la cuantización, por lo que la carga de trabajo está desbalanceada. Además, aumentan los requerimientos de memoria al igual que en la descomposición por datos, debido a que es necesario copiar los resultados de la transformada para usarlos en la fase de cuantización sin que sean sobreescritos por las siguientes operaciones.

Es posible obtener una división funcional más adecuada fijándose en cómo se realiza la codificación. Cuando la primera pasada de la transformada se ha aplicado a un bloque de vídeo en la dimensión T, la mitad de los frames representan las frecuencias bajas y la otra mitad las altas. Después de aplicar la transformada en todos los ejes, sólo la octava parte de los pixels son necesarios para la segunda pasada de la transformada.

En el nuevo esquema de paralelización un hilo se encarga de la transformada Wavelet y la cuantización de la parte baja mientras que el otro se encarga de la cuantización de la parte alta. El segundo hilo puede empezar cuando se ha realizado la primera pasada. Por tanto, el solapamiento se produce entre la transformada Wavelet junto con la cuantización baja y la cuantización alta. Esta estrategia está mejor balanceada que la anterior y además no requiere de ninguna copia de datos.

La implementación efectiva de la estrategia descrita se puede realizar usando varias tecnologías. A continuación se describen las implementaciones realizadas y se comparan desde el punto de vista de la mantenibilidad y reusabilidad del código obtenido y su velocidad de ejecución.

En la figura 1 se muestra un esquema de las primeras dos implementaciones (*openmp1* y *pthreads1*) de la codificación de un bloque. En cada iteración para cada bloque de $32 \times 512 \times 16$ se realiza la cuantización de la parte alta en un hilo esclavo independiente, mientras que el hilo principal continua realizando la segunda pasada de la transformada Wavelet y la cuantización de la parte baja. La umbralización la realiza el mismo hilo que la transformada para conseguir un mejor balanceo de la carga.

```
for (...)  /* bucle exterior */
{
    /* 1ª pasada 3D-FWT */
    #pragma omp sections
    {
        #pragma omp section
        {
            /* 2ª pasada 3D-FWT */
            /* cuantización baja */
        }
        #pragma omp section
        {
            /* cuantización alta */
        }
    }
}
```

Fig. 2. Fragmento de código del esquema *openmp1*.

A. Usando OpenMP

La primera implementación paralela realizada parte de la implementación secuencial del algoritmo y la paraleliza según se ha descrito anteriormente (ver figura 1) usando OpenMP.

Los cambios que ha sido necesario realizar con respecto al código de la versión secuencial son mínimos gracias al alto nivel expresivo de OpenMP. En particular, no ha sido necesario utilizar ninguna primitiva de sincronización explícitamente.

Concretamente, se han utilizado simplemente una directiva `#pragma omp parallel sections` que define varios bloques que se ejecutan paralelamente por varios hilos. Cada una de estos bloques se especifica con la directiva `#pragma omp section`. Uno de los bloques (hilo principal) contendrá el código de la segunda pasada de la 3D-FWT y la cuantización baja y el otro (hilo esclavo) contendrá la cuantización alta. El resultado se muestra en la figura 2.

El código resultante es muy similar a la implementación secuencial inicial. De hecho, si ignoramos las directivas OpenMP, sería posible ejecutarlo secuencialmente obteniendo resultados correctos. Esto prueba la sencillez de la paralelización con OpenMP y que no introduce problemas de mantenibilidad o reusabilidad.

B. Usando Pthreads

El mismo esquema de la figura 1 se ha implementado usando Pthreads (*pthreads1*). Se ha partido de la versión anterior y se ha realizado una versión equivalente que usa Pthreads en lugar de OpenMP. Los principales cambios realizados han sido:

- Extraer el código relativo al hilo esclavo y ponerlo en una función independiente. Esto es necesario porque Pthreads requiere que el punto de entrada de cada nuevo hilo sea una función.
- Crear una estructura de datos auxiliar para el paso de parámetros. Las funciones a utilizar como punto de entrada de un hilo en Pthreads deben tener una signatura determinada. En particular, tan sólo pueden recibir un parámetro y éste debe ser un puntero a `void`. El patrón habitual para pasar más parámetros consiste en

```
struct thr_data { int f, r, c, ...  };

int hilo_esclavo (struct thr_data *data)
{
    /* cuantización alta */
}

void fwt (int rows, int cols, int frames, ...)
{
    for (...)  /* bucle exterior */
    {
        struct thr_data data;
        pthread_t hilo;
        /* 1ª pasada 3D-FWT */
        data.f = f; data.r = r; data.c = c; ...;
        pthread_create (&hilo, NULL,
                        hilo_esclavo, &data);
        /* 2ª pasada 3D-FWT */
        /* cuantización baja */
        pthread_join (hilo, NULL);
    }
}
```

Fig. 3. Fragmento de código del esquema *pthreads1*.

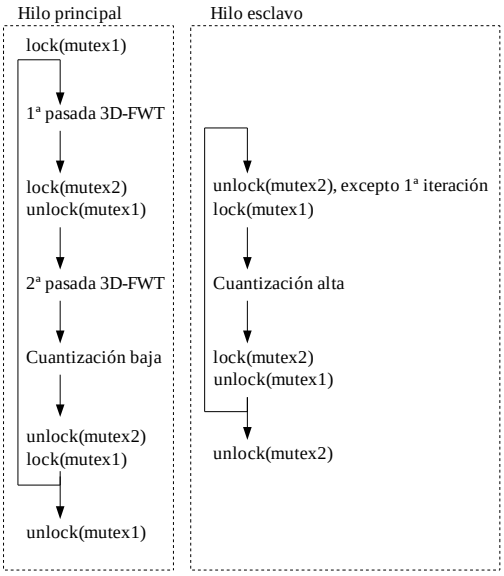

Fig. 4. Segundo esquema de paralelización.

crear una estructura auxiliar con un campo por cada parámetro necesario y pasar un puntero a dicha estructura. Este es nuestro caso, ya que al extraer el código del hilo esclavo a una función independiente en necesario comunicarle información que antes se almacenaba en variables locales.

El código que se obtiene después de las transformaciones manuales anteriores es bastante diferente a la versión secuencial, como puede verse en la figura 3. La implementación no resulta muy complicada por tratarse de cambios de la estructura del código siguiendo patrones conocidos, pero las nuevas estructuras de datos y funciones creadas con el único fin de permitir la paralelización dificultan la legibilidad y el mantenimiento del código.

C. Usando Pthreads más eficientemente

El uso de Pthreads implica una programación a más bajo nivel que OpenMP, lo que permite un mayor control y flexibilidad a la hora de decidir cómo paralelizar un programa. La implementación descrita en la sección anterior no aprovecha este hecho. Concretamente, se crea y se destruye un nuevo hilo para cada bloque, lo que podría resultar costoso.

Se ha diseñado un esquema alternativo (*pthreads2*) que evita la creación de varios hilos y usa únicamente dos para todos los bloques. Estos dos hilos se crean al principio del proceso de codificación y se usan candados para conseguir la sincronización entre ellos.

No es suficiente con el uso de una región crítica protegida por un candado, ya que es necesario que ambos hilos se ejecuten parcialmente en orden, es decir, el segundo hilo no puede comenzar con un nuevo bloque hasta que el hilo principal haya completado la primera pasada de la transformada Wavelet del bloque previo y el hilo principal no puede comenzar con un nuevo bloque hasta que el esclavo haya acabado el actual.

Para conseguir esta ordenación parcial se han utilizado dos candados que se adquieren alternativamente por ambos hilos. También se podría haber utilizado semáforos en su lugar. En la figura 4 se muestra un esquema de dicha implementación.

Los cambios realizados al código respecto a la versión secuencial para esta implementación son mucho mayores que en los casos anteriores. Por ejemplo, hay que duplicar los bucles del codificador en cada hilo, además de añadir las estructuras de datos y funciones adicionales descritas en la sección anterior.

Estos cambios implican una reestructuración profunda del código con el consiguiente deterioro de la legibilidad y mantenibilidad del código. El nuevo código, que se puede ver en la figura 5, es muy distinto al secuencial, y no resulta obvio para un programador que no haya seguido el proceso de desarrollo qué aspectos son relativos al algoritmo en sí y cuáles a la paralelización.

D. Usando OpenMP como Pthreads

OpenMP también permite el uso de regiones críticas y candados. Por tanto, es posible utilizar el mismo esquema anteriormente descrito e implementar el algoritmo con OpenMP según la figura 4 (*openmp2*). Sin embargo, este enfoque es poco natural y el resultado es más difícil de comprender.

Para conseguir esta implementación, es necesario realizar parte de las transformaciones manuales utilizadas para el esquema anterior. No es necesario crear nuevas estructuras y tampoco hace falta extraer el código del hilo esclavo a otra función. Sin embargo, sigue siendo necesario duplicar los bucles, reestructurando el código considerablemente, como se puede ver en la figura 6.

La necesidad de reestructuración y la introducción de los candados provoca la pérdida de una de las ventajas de OpenMP: que el código paralelizado pueda ejecutarse secuencialmente ignorando las directivas OpenMP obteniendo resultados correctos. De hecho, el resultado es un código más similar a la versión *pthreads2* que a *openmp1* en cuanto a estructura y con problemas similares de legibilidad.

```
struct thr_data { int rows, cols, frames, ...  };
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;

int hilo_esclavo (struct thr_data *data)
{
    for (...)  /* bucle exterior */
    {
        if (/* no es la primera iteración */)
        {
            pthread_mutex_unlock (&mutex2);
        }
        pthread_mutex_lock (&mutex1);
        /* cuantización alta */
        pthread_mutex_lock (&mutex2);
        pthread_mutex_unlock (&mutex1);
    }
    pthread_mutex_unlock (&mutex2);
}

void fwt (int rows, int cols, int frames, ...)
{
    struct thr_data data;
    pthread_t hilo;
    pthread_mutex_lock (&mutex1);
    data.rows = rows; data.cols = cols; ...;
    pthread_create (&hilo, NULL,
                    (hilo_esclavo, &data);
    for (...)  /* bucle exterior */
    {
        /* 1ª pasada 3D-FWT */
        pthread_mutex_lock (&mutex2);
        pthread_mutex_unlock (&mutex1);
        /* 2ª pasada 3D-FWT */
        /* cuantización baja */
        pthread_mutex_unlock (&mutex2);
        pthread_mutex_lock (&mutex1);
    }
    pthread_mutex_unlock (&mutex1);
    pthread_join (hilo, NULL);
}
```

Fig. 5. Fragmento de código del esquema *pthreads2*.

V. Evaluación y análisis

Estamos interesados en evaluar el rendimiento de cada una de las técnicas anteriores tanto en cuanto al tiempo de ejecución como en la facilidad de implementación y la mantenibilidad del código. En la figura 7 se muestra el tiempo de ejecución de las distintas implementaciones usando dos procesadores independientes, y usando un sólo procesador con HyperThreading.

Se ha usado el compilador de C++ de Intel® v7.1 para compilar y evaluar las distintas versiones del algoritmo. Para las pruebas con multiprocesadores se ha utilizado un Intel® Xeon™a 2 GHz con 2 procesadores. Para las pruebas con HyperThreading se ha utilizado el mismo equipo activando la capacidad de HyperThreading de uno de los dos procesadores y desactivando completamente el otro.

En primer lugar, observamos que cuando obtenemos una mejora en las prestaciones usando varios procesadores también la obtenemos usando HyperThreading, y aunque la mejora es siempre mucho menor no deja de ser significativa. Como se ha mencionado en la sección IV, el esquema de paralelización utilizado permite que sea así al realizar operaciones de distinto tipo para evitar contención en las unidades funcionales compartidas del procesador.

Las implementaciones *openmp1* y *pthreads1* ob-

```
void fwt (int rows, int cols, int frames, ...)
{
    omp_lock_t lock1;
    omp_lock_t lock2;
    omp_set_lock (&lock1);
    #pragma omp sections
    {
        #pragma omp section
        {
            for (...)  /* bucle exterior */
            {
                /* 1ª pasada 3D-FWT */
                omp_set_lock (&lock2);
                omp_unset_lock (&lock1);
                /* 2ª pasada 3D-FWT */
                /* cuantización baja */
                omp_unset_lock (&lock2);
                omp_set_lock (&lock1);
            }
            omp_unset_lock (&lock1);
        }
        #pragma omp section
        {
            for (...)  /* bucle exterior */
            {
                if (/* no es la primera pasada*/)
                {
                    omp_unset_lock (&lock2);
                }
                omp_set_lock (&lock1);
                /* cuantización alta */
                omp_set_lock (&lock2);
                omp_unset_lock (&lock1);
            }
            omp_unset_lock (&lock2);
        }
    }
}
```

Fig. 6. Fragmento de código del esquema *openmp2*.

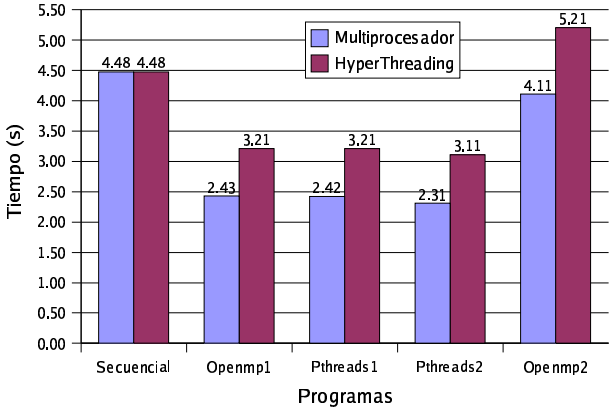

Fig. 7. Tiempos de ejecución de las distintas implementaciones.

tienen prácticamente los mismos tiempos. De estas dos, la primera es bastante más fácil de implementar y legible ya que, como se ha mostrado antes, los cambios realizados respecto al código secuencial para *openmp1* son escasos y muy poco invasivos, mientras que la versión *pthreads1* requiere una cierta reestructuración y la introducción de nuevas estructuras de datos. Además, en la versión *openmp1* se puede utilizar el mismo código para una versión secuencial y una paralela, lo que resulta muy útil para la depuración en la fase de desarrollo.

La implementación *pthreads2* es la que mejor rendimiento obtiene, siendo también la de mayor dificultad de implementación y la que genera mayo-

res problemas de mantenibilidad debido a la gran reestructuración necesaria respecto a la versión secuencial. La ventaja que se obtiene respecto a las otras implementaciones más sencillas y mantenibles es pequeña, por lo que no justifica el aumento de complejidad. Es decir, el esfuerzo de programación necesario para obtener la pequeña mejora respecto a *openmp1* es demasiado grande para ser justificado. En especial, la introducción de sincronización explícita es problemática debido a la dificultad de conseguir una implementación correcta y el peligro de introducir errores difíciles de detectar, como condiciones de carreras.

La implementación *openmp2* obtiene los peores resultados, peores incluso que la versión secuencial. Es importante señalar que esta versión es equivalente a la versión *pthreads2*, por lo que la disminución de prestaciones resulta inesperada. Pruebas realizadas indican que la implementación de las primitivas de sincronización en la librería de soporte de OpenMP de Intel se comporta peor que los mutex de Pthreads en presencia de mucha contención, como es el caso en nuestro esquema de paralelización.

VI. Conclusiones

En este trabajo se han aplicado varias técnicas de paralelización manual a un codificador de vídeo basado en la transformada Wavelet 3D. Las técnicas empleadas, Pthreads y OpenMP, se han evaluado y comparado desde el punto de vista del tiempo de ejecución del programa obtenido, la dificultad para realizar la paralelización y la legibilidad y mantenibilidad del código obtenido.

Las implementaciones sencillas, en especial la primera implementación basada en OpenMP, nos han permitido obtener la mayoría del beneficio posible gracias a la paralelización. Un mayor esfuerzo de implementación ha permitido obtener un beneficio algo mayor, pero el aumento de complejidad es, proporcionalmente, mucho mayor que el aumento en prestaciones.

El uso de OpenMP en lugar de la paralelización manual usando Pthreads tiene evidentes ventajas en cuanto a facilidad de uso y legibilidad del código producido. Esto es especialmente cierto en el caso de convertir un código diseñado originalmente como secuencial a código paralelo.

De los resultados obtenidos se puede deducir que la estrategia óptima de implementación depende de la tecnología a utilizar. Las formas más naturales de resolver el problema con cada una de las tecnologías empleadas han sido la que mejor resultado han obtenido (*openmp1* y *pthreads2*). La técnica que mejor resultado obtiene en general ha obtenido resultados desastrosos cuando se ha intentado aplicar en una tecnología que no suele usarse con ese estilo y no está por tanto optimizada para ello.

Los malos resultados obtenidos por *openmp2* pueden indicar una implementación deficiente de las primitivas de sincronización en la librería de soporte de OpenMP de Intel. Sería interesante como trabajo futuro comparar el resultado obtenido usando otros compiladores que soporten tanto OpenMP como el uso de intrínsecos para utilizar la instrucciones SSE.

Por otro lado, los buenos resultados obtenidos por *openmp1* y *pthreads1*, prácticamente iguales y muy cercanos a los de la mejor implementación dan a entender que el coste de crear y destruir muchos hilos no es muy grande en comparación con su alternativa basada en sincronización.

Referencias

[1] Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, Rebecca L. Stamm, and Dean M. Tullsen, "Simultaneous multithreading: A platform for next-generation processors," *IEEE Micro*, vol. 17, no. 5, pp. 12–24, 1997.

[2] Jack L. Lo, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, and Dean M. Tullsen, "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 322–354, 1997.

[3] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua, "Automatic program parallelization," *Proceedings of the IEEE*, vol. 81, no. 2, pp. 211–243, 1993.

[4] William Blume, Rudolf Eigenmann, Jay Hoeflinger, David Padua, Paul Petersen, Lrence Rauchwerger, and Peng Tu, "Automatic detection of parallelism: A grand challenge for high performance computing," *Parallel and Distributed Technology: Systems and Applications, IEEE*, vol. 2, no. 3, pp. 37–, 1994.

[5] ISO/IEC 14495-1 and ITU Recomendation T.87, "Lossless and near-lossless coding of continuous tone still images (jpeg-ls)," 1999.

[6] Beong-Jo Kim and William A. Pearlman, "An embedded wavelet video coder using three-dimensional set partitioning in hierarchical trees (SPIHT)," in *Designs, Codes and Cryptography*, 1997, pp. 251–260.

[7] Gregorio Bernabé, José González, José M. García, and José Duato, "A new lossy 3d wavelet transform for high-quality compression of medical video," in *IEE EMBS International Conference on Information Technology Applications in Biomedicine*, November 2000, pp. 226–231.

[8] T. Sikora, "Mpeg digital video–coding standards," *IEEE Signal Processing Magazine*, vol. 14, no. 5, pp. 82–100, September 1997.

[9] S. Battista, F. Casalino, and C. Lande, "Mpeg-4: A multimedia standard for the third millenium, part 1," *IEEE Multimedia*, October 1999.

[10] S. Battista, F. Casalino, and C. Lande, "Mpeg-4: A multimedia standard for the third millenium, part 2," *IEEE Multimedia*, vol. 6, no. 4, pp. 4–83, January 2000.

[11] Gregorio Bernabé, José González, and José M. García, "An efficient 3d wavelet transform on hyper-threading technology," Tech. Rep., Universidad de Murcia, 2004.

[12] OpenMP Architecture Review Board, "Openmp c and c++ application program interface, version 2.0," March 2002.

[13] "Ieee p1003.1c-1995: Information technology-portable operating system interface (posix)," 1995.

[14] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton, "Hyper threding technology architecture and microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, pp. 4–15, February 2002.

[15] W. Magro, P. Petersen, and S. Shah, ," *Intel Technology Journal*, vol. 6, no. 1, pp. 58–66, February 2002.

[16] Stephane G. Mallat, "A theory for multiresolution signal decomposition: the wavelet representation," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. PAMI-11, pp. 674–693, July 1989.

[17] Gregorio Bernabé, José González, José M. García, and J. Duato, "Memory conscious 3d wavelet transform," in *28th Euromicro Conference, Multimedia and Telecommunications Track*. IEEE, September 2002, pp. 108–113.

[18] Gregorio Bernabé, José M. García, and José González, "Reducing 3d wavelet transform execution time through the streaming simd extensions," in *11th Euromicro Conference on Parallel Distributed and Network based Processing*, February 2003.