Optimizing a 3D-FWT Video Encoder for SMPs and HyperThreading Architectures

Ricardo Fernández José M. García Gregorio Bernabé
Manuel E. Acacio
Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia
Campus de Espinardo - 30080 Murcia (España)
{r.fernandez,jmgarcia,gbernabe,meacacio}@ditec.um.es

Abstract

In this work we evaluate the implementation of a video encoder based on the 3D Wavelet Transform optimized for HyperThreading technology and SMPs. We design several implementations of the parallel encoder with Pthreads and OpenMP using functional decomposition. Then, we compare them in terms of execution speed, ease of implementation and maintainability of the resulting code. Our experiments show that while Pthreads provides the best results in terms of execution time, OpenMP can provide a nearly optimal execution time without sacrificing the maintainability of code.

1. Introduction

Current trends in computer architecture seek to exploit increasing parallelism at every level. Nowadays, an increasing number of computers are shared memory multiprocessors or they have processors able to execute several threads at the same time using Simultaneous Multithreading (SMT [12] [15] [18]). However, successfully exploiting these architectures requires using several threads or processes, which forces us to rethink the implementation of many algorithms.

Examples of these architectures are the Intel® processors with HyperThreading technology [19]. This technology makes feasible that a single processor can execute simultaneously two processes or threads adding relatively little complexity to the processor design.

On the other hand, the Wavelet transform allows codifying video at high quality while achieving great compression rates. This property makes the Wavelet transform a good option for building a video encoder able to obtain better performance than other more general purpose algorithms such as MPEG [20] or MPEG-4 [4] [5], as shown in [13] [8]. One of the main problems when working with the video sequence as a tridimensional signal is the huge datasets that have to be dealt with. Therefore, memory accesses slow down the encoder execution.

Multimedia applications have an increasing importance in many areas. Namely, there is a growing need to store and transmit high quality video for applications where common coding schemes do not yield enough quality. An example of this is medical video, whose quality requirements are very high and there exist regulations which mandate to keep it stored for long time. This makes necessary good compression techniques, but due to quality requirements, lossless compression techniques are commonly used (JPEG-LS [2]), which are impractical due to the low compression rates that can be obtained using them.

In the case of parallelizing a video encoder, the automatic parallelization ([3] [10]) methods available to us do not yield any benefit. It is necessary to use manual parallelization, specially to take advantage of the benefits that HyperThreading technology provides [7].

Manual parallelization poses a considerable burden in software development. It would be desirable to minimize this increase in complexity. There are a number of alternatives for implementing a parallel algorithm, each one with a different level of difficulty, maintainability and flexibility. This paper deals with the implementation of a parallel video encoder using the 3D wavelet transform. We evaluate and compare two of these alternatives: OpenMP [11] and Pthreads [1], in terms of the execution time of the resulting programs, ease of implementation, maintainability and reusability of the produced code. We present an implementation of the encoder described in [7] using OpenMP which is easier to implement, more readable and nearly as efficient as the original implementation using Pthreads.

The rest of the paper is organized as follows: in section 2, the basics of the 3D wavelet and the technologies em-

ployed in this paper (i.e., OpenMP and Pthreads) are presented. Section 3 discuses previous work on the encoder, including several optimizations applied to the sequential algorithm and parallelization strategies explored. Section 4 exposes the new parallel implementations of the algorithm. Finally, Section 5 studies the execution time of each implementation and compares it with the sequential implementation, and Section 6 presents the conclusions and the future work of this paper.

2. Background

2.1. Pthreads

Pthreads (POSIX threads, [1]) is a commonly portable API used for programming shared memory multiprocessors. This API is a lower level than OpenMP. Hence, it allows a greater control about how to exploit concurrency at the expense of increasing difficulty to use it.

In the Pthreads programming model the programmer has to create explicitly all threads and take care of all the necessary synchronization. Pthreads has a rich set of synchronization primitives which include locks (mutex), semaphores and condition variables.

2.2. OpenMP

OpenMP [11] is an specification which allows the implementation of portable parallel programs in shared memory multiprocessor architectures using C, C++ or FORTRAN.

Programming using OpenMP is based on the use of directives which show the compiler what to parallelize and how. These directives allow the programmer to create parallel sections, mark parallelizable loops and define critical sections. When a parallel region or parallel loop is defined, the programmer has to specify which variables are private for each thread, shared or used in reductions.

The programmer does not have to specify how many threads will be used or how the execution of a parallel loop will be scheduled. OpenMP includes a number of policies which can be chosen at run time.

OpenMP allows an incremental parallelization of programs. Therefore, it is specially well suited for adding parallelism to programs which were initially written sequentially. In most cases, it is possible to keep the same code for the sequential and parallel versions of the program just ignoring the OpenMP directives when compiling the sequential program. This is specially useful in the development phases to allow an easier debugging.

2.3. SMT and HyperThreading

An SMT processor can execute several treads simultaneously, sharing most hardware resources like caches, functional units, branch predictors, etc and duplicating only those resources necessary to store the status of every process, such as registers. This way, a single physical processor looks to the operating system and applications as two different virtual processors.

Resource sharing allows a better use of processor resources, permitting a thread to continue execution while another is blocked by a cache miss or branch misprediction. It is possible that, in some cases, the execution time of a sequential application increases, but the overall productivity of the system is improved.

Intel® has introduced an implementation of SMT called HyperThreading [19] in its high performance x86 processors like Xeon™ and Pentium™ IV obtaining improvements in execution time around 30% in some cases [16].

2.4. Fast Wavelet Transform

The Wavelet transform decomposes a signal extracting the information present at several resolutions [17]. The fast Wavelet transform can be implemented using a pair of QMF filters. The transform of a discrete signal is calculated by the convolution of the signal with each filter and downsampled by 2.

One of the filters, H, is a low-pass filter which extracts coarse grain information from the signal, while the other, G, is a high-pass filter which extracts the details from the signal. Hence, we obtain two signals, *low* and *high*, with half of the samples of the original signal.

Applying several times the transform to the coarse grain part produces a greater decorrelation of the original information. This increases the number of coefficients that can be discarded during the umbralization phase. Hence, a greater compression ratio can be achieved in the following phases. However, this fact has a significative impact on the quality of the reconstructed video. So, there is a tradeoff between a greater compression and better quality.

The Wavelet transform is generalized to two or more dimensions by applying successively the one dimensional Wavelet transform to each dimension. The particular order is not important [14].

3. Previous Work

We are interested in applying the fast 3D wavelet transform algorithm (3D-FWT) to the compression of medical video of size 512×512 and grayscale (8 bits per pixel) in sequences of 64 frames. From previous work [8], we have concluded that the best configuration uses the Daubechies

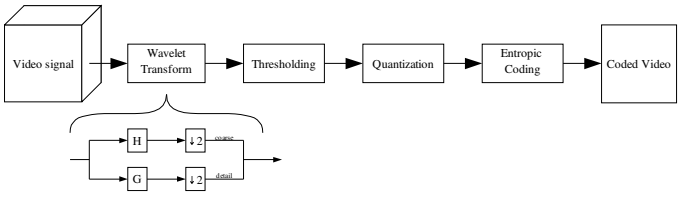

Figure 1. Encoder phases.

of 4 coefficients as mother function and does two passes in each dimension.

In figure 1, we show a diagram of the phases of the encoder that we have developed. After the Wavelet transform, we apply a thresholding step which discards wavelet coefficients whose absolute value is less than a certain constant.

Then, we make the most costly step in terms of CPU time: the quantization, which converts the floating point coefficients of the transform into unsigned integers with certain number of bits which is determined depending of the current cube.

Finally, we perform an entropic coding using binary 3D run-length coding and the result is compressed using an arithmetic coder.

3.1. Optimizing the sequential algorithm

Below we are going to present a number of techniques that we developed [6] [8] [9] to improve the execution time of our encoder.

3.1.1. Blocking From the data shown above, the size of the signal to be coded is 16 MB. Under these circumstances, the execution time is dominated by memory accesses due to the huge size of the working set and the low locality of the accesses. Blocking raises the locality of the algorithm and reduces the traffic between memory and the processor, increasing the number of hits at the higher levels of cache. In [9], both a cube partitioning and rectangle partitioning blocking strategies were presented, being able in each case to introduce overlapping or not.

Both approaches had the same drawback: they caused some quality degradation because they isolated the pixels at the borders of the subregions from those that surround them. In the case of a mother function with four coefficients the value of the transform in each pixel depends on the four adjacent pixels. Hence, in the case of the pixels near the limit of the subregions created by blocking, they depend on pixels which are outside the subregion.

The non overlapping strategies use the value from pixels from that same block, either replicating the pixels from the border or using those from the opposite side. This produces an appreciable quality loss, specially in the limits between regions where image discontinuities appear.

The overlapping strategies use the pixels from the adjacent blocks, avoiding the quality loss but exploiting less spatial locality. Also, this approach implies some redundant calculations. It is possible to store these results and reuse them for the next block and it has been shown that it is beneficial at least in the case of rectangle partitioning [9].

We call the last strategy *rectangular blocking with overlapping and operation reuse* and it will be the strategy assumed in the rest of this work. It has been determined (see previous paper [9]) that $32 \times 512 \times 16$ is the optimal blocking size.

3.1.2. Vectorization The target architecture for our optimizations (Intel® Xeon™) provides us with vectorial instructions specifically targeted to multimedia applications. SSE extensions allow us to exploit fine grain parallelism vectorizing loops which perform a simple operation over data streams. It is possible to exploit these instructions to reduce the number of floating point instructions executed to calculate the transform.

The calculation of four *low* pixels and four *high* pixels of the 3D-FWT requires 32 floating point multiplications and 24 floating point additions. Using SSE, it is possible to perform those operations with only 15 instructions [6].

3.1.3. Columns Vectorization Given that the signal is stored in row order, locality is exploited when the transform is applied in the X dimension. When the transform is applied in the Y dimension, the number of memory misses increases considerably even when the blocking strategies previously described are applied. This is due to the need of values from several successive rows to calculate each column transform.

Column vectorization consists of interleaving the calculations of the transform in the Y dimension and the transform in the X dimension, exploiting the fact that the results from the X dimension are the values needed to perform the calculations in the Y dimension [6].

The speedup obtained with this optimization is very important [6], specially when combined with the use of SSE instructions for the calculations in the Y dimension.

3.2. Parallelization

In [7], we developed two strategies for parallelizing the Wavelet transform specially targeted to the HyperThreading technology: using data decomposition and functional decomposition.

Data decomposition applies the transform simultaneously to two independent blocks of frames from the sequence. The strategy allows for good load balancing and incurs very little communication. However, the working set doubles and both threads perform identical functions, which

makes this technique poorly suited for the HyperThreading architecture.

Functional decomposition performs independent phases of the encoder in parallel. Namely, we can simultaneously perform the transform and quantization phases.

Wavelet transform takes less than half the time of the quantization to be executed, hence the workload is not balanced. Also, memory requirements increase as in the case of data decomposition because it is necessary to copy the results of the transform to use them in the quantization without being overwritten by the next operations.

It is possible to obtain a better functional decomposition noticing how coding is performed. When the first transform pass has been applied to a video block in the T dimension, half of its frames represent low frequencies and the other half high frequencies. After applying the transform to the three axis, only an eight part of the pixels are necessary to the second pass of the transform: those which represent low frequencies.

In the new parallelization scheme a thread performs the wavelet transform and the quantization of the low part while another thread performs the quantization of the high part. The second thread can start once the first pass has been finished, hence the overlapping happens between the wavelet transform plus the quantization of the low part and the quantization of the high part. This strategy is better balanced than the previous one and does not require any data copying.

In [7] we implemented the described strategy using Pthreads, which implies a low level programing which allows for a great control and flexibility to decide exactly how to parallelize the program. In that implementation, we exploited this fact to avoid the creation and destruction of a thread for each block, which could be quite costly. We designed a scheme (*pthreads1*) which used only two threads for all blocks. These threads were created at the beginning of the coding process and we used locks to achieve synchronization between them.

It is not enough to use a critical region protected with a single lock because we need both threads to be executed in partial order. That is, the slave thread can't start with a new block until the first thread has finished the first pass of the Wavelet transform for the previous block and the main thread cannot start a new block until the slave thread has finished it.

To achieve this partial ordering we have used two locks which are acquired alternately by both threads. In figure 2, we show an scheme of the implementation.

The changes made to the code with respect to the sequential version are large. The main changes are:

- Extract the code related with the slave thread and put it in an independent function. This is necessary because Pthreads requires the entry point for every thread to be a function.

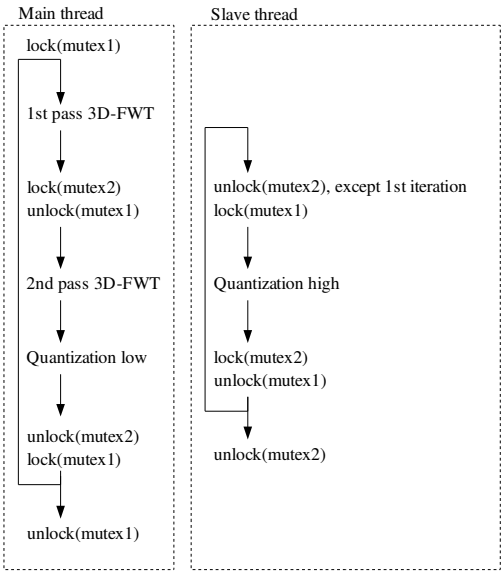

Figure 2. First parallelization scheme.

- Create an auxiliary structure for parameter passing. The functions used as thread entry points must have a specific signature. Namely, they can only take one parameter. The usual pattern for passing more than one parameter consists of creating an auxiliary structure with a field for each necessary parameter and pass a pointer to that structure. Because we extract the slave thread's code into an independent function, it is necessary to communicate some information that was previously available in local variables.
- Duplicate the loops of the encoder in each thread and add the synchronization points using locks.

These changes imply a deep restructuring of the code with the associated drop in legibility and maintainability. The new code, which can be seen in figure 3, is very different to the sequential code and it does not seem obvious what things are related to the algorithm and what are related with the parallelization.

Recently, independently to our work, Tenllado et al. [21] have developed a related work parallelizing the discrete Wavelet transform (DWT) for its execution using HyperThreading technology and SIMD. Some of their conclusions are similar to ours: they also found that a functional decomposition better suited for HyperThreading than a data decomposition, although they did not use functional decomposition when comparing SMT against HyperThreading. Unlike our work, they only use OpenMP for doing the parallelization and the working set of their application is smaller than ours, because they apply the DWT only to bidimensional images instead of a tridimensional signal representing a video sequence.

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

int slave_thread (struct thr_data *data)
{
    for (...) /* outer loop */
    {
        if (/* not in the 1st iteration */)
        {
            pthread_mutex_unlock (&mutex2);
        }
        pthread_mutex_lock (&mutex1);
        /* quantization 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 thread;
    pthread_mutex_lock (&mutex1);
    data.rows = rows; data.cols = cols; ...;
    pthread_create (&thread, NULL,
                    (slave_thread, &data);
    for (...) /* outer loop */
    {
        /* 1st pass 3D-FWT */
        pthread_mutex_lock (&mutex2);
        pthread_mutex_unlock (&mutex1);
        /* 2nd pass 3D-FWT */
        /* quantization low */
        pthread_mutex_unlock (&mutex2);
        pthread_mutex_lock (&mutex1);
    }
    pthread_mutex_unlock (&mutex1);
    pthread_join (thread, NULL);
}
```

Figure 3. Code snippet for the *pthreads1* scheme.

4. Using OpenMP to ease the implementation

Next, we describe an OpenMP alternative implementation with the same performance characteristics than the previous one but significantly easier to code and more maintainable.

4.1. Using OpenMP like Pthreads

OpenMP also allows using critical regions and locks. Hence, it is possible to use the same scheme previously described and implement the algorithm using OpenMP as described in figure 2 (*openmp1*). However, this approach is not much better and the result is not more comprehensible.

To make this implementation, it is necessary to do part of the manual transforms used in the previous scheme. It is not necessary to create new structures and it is not necessary to extract the code of the slave thread to another function either. However, it is still necessary to duplicate the loops, restructuring the code considerably, as can be seen in figure 4.

```
void fwt (int rows, int cols, int frames, ...)
{
    omp_lock_t lock1;
    omp_lock_t lock2;
    omp_set_lock (&lock1);
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            for (...) /* outer loop */
            {
                /* 1st pass 3D-FWT */
                omp_set_lock (&lock2);
                omp_unset_lock (&lock1);
                /* 2nd pass 3D-FWT */
                /* quantization low */
                omp_unset_lock (&lock2);
                omp_set_lock (&lock1);
            }
            omp_unset_lock (&lock1);
        }
        #pragma omp section
        {
            for (...) /* outer loop */
            {
                if (/* not the 1st iteration */)
                {
                    omp_unset_lock (&lock2);
                }
                omp_set_lock (&lock1);
                /* quantization high */
                omp_set_lock (&lock2);
                omp_unset_lock (&lock1);
            }
            omp_unset_lock (&lock2);
        }
    }
}
```

Figure 4. Code snippet for the *openmp1* scheme.

The resulting code (with the introduction of locks) means the loss of one of the advantages of OpenMP: the parallelized code cannot be compiled and run as sequential code correctly. This way of using OpenMP is not typical and does not take advantage of the features provided by it.

4.2. Easily using OpenMP

It is actually easier to create a parallel version using OpenMP if we start directly from the sequential implementation and just add the needed directives.

In figure 5, we show the scheme of the implementation (*openmp2*) of a block codification. In each iteration for each block of $32 \times 512 \times 16$ pixels, an independent slave thread

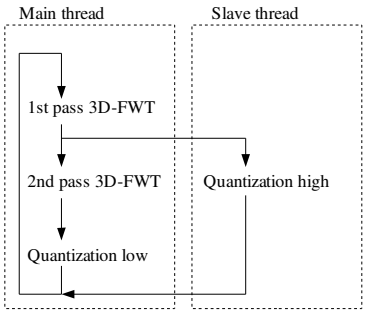

Figure 5. Second parallelization scheme.

```
for (...) /* outer loop */
{
    /* 1st pass 3D-FWT */
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            /* 2nd pass 3D-FWT */
            /* quantization low */
        }
        #pragma omp section
        {
            /* quantization high */
        }
    }
}
```

Figure 6. Code snippet for the *openmp2* scheme.

performs the quantization of the high part while the main thread performs the second pass of the Wavelet transform and the quantization of the low part. Thresholding is performed by the same thread than the transform in order to achieve better load balancing.

The changes that we have introduced to the sequential code are minimal thanks to OpenMP's high expressive level. Namely, it has not been necessary to use any explicit synchronization primitive.

Specifically, we have simply used a `#pragma omp parallel sections` directive which defines two blocks that are executed in parallel. Each block is specified with a `#pragma omp section` directive. One of the blocks (main thread) contains the code for the second pass of the transform and the quantization of the low part while the other block contains the code for the quantization of the high part. The result is shown in figure 6.

It is important to note that this implementation creates a new thread for each block, unlike the *pthreads1* and *openmp1* implementations and does not use any manual synchronization.

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

int slave_thread (struct thr_data *data)
{
    /* quantization high */
}

void fwt (int rows, int cols, int frames, ...)
{
    for (...) /* outer loop */
    {
        struct thr_data data;
        pthread_t thread;
        /* 1st pass 3D-FWT */
        data.f = f; data.r = r; data.c = c; ...;
        pthread_create (&thread, NULL,
                        slave_thread, &data);
        /* 2nd pass 3D-FWT */
        /* quantization low */
        pthread_join (thread, NULL);
    }
}
```

Figure 7. Code snippet for the *pthreads2* scheme.

The resulting code is very similar to the sequential implementation. Actually, if we ignore the OpenMP directives, it is possible to execute it as a sequential program obtaining correct results. This proves the simplicity of the parallelization using OpenMP and its maintainability and reusability properties.

4.3. Using Pthreads like OpenMP

The same scheme from figure 5 has been implemented using Pthreads (*pthreads2*) for comparison purposes. We have taken the previous version as a starting point and we have derived an equivalent version using Pthreads instead of OpenMP.

We have had to perform some of the same changes that we did for the *pthreads1* version. We have not had to duplicate the loops because the threads are created inside each iteration and we have been able to avoid the use of manual synchronization, but we have had to extract the body of the slave thread to an independent function and we have had to create an auxiliary structure for parameter passing.

The resulting code after the manual transformation is quite different to the sequential version, as can be seen in figure 7. The implementation is not very complicated because it consists mostly of structural changes to the code following some well known patterns, but the new data structures and functions created degrade the legibility, maintainability and reusability of the code.

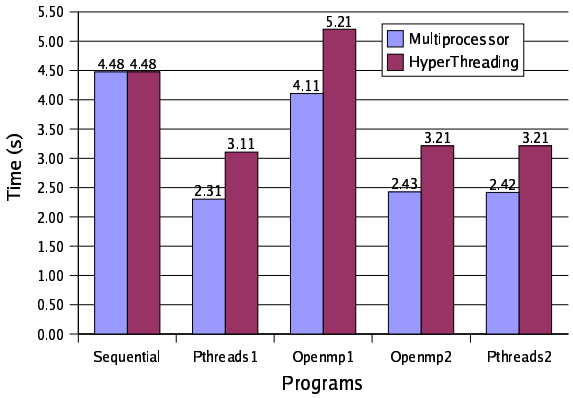

Figure 8. Execution times for the different implementations.

Hardware	
Processors	2 Intel® Xeon™
Frequency	2 GHz
TLB	64 entry, fully associative
Level 1 Instruction Cache	Trace Cache 12 K micro-ops
Level 1 Data Cache	8 KB, 4 way, 64 byte/line
Level 2 Unified Cache	512 KB, 8 way, 128 byte/line
RAM memory	512 MB DRAM
Software	
Operative System	Linux 2.4.18-3smp
Compiler	Intel® v7.1

Table 1. Evaluation environment characteristics

5. Evaluation and Analysis

In this section, we present the evaluation of the performance of each one of the previous techniques with respect to their execution time and its maintainability. In figure 8, we show the execution times for the different implementations using two independent processors and a single processor with HyperThreading technology. In table 1 we show the relevant hardware and software used for our tests. For the HyperThreading tests we have used the same machine enabling the HyperThreading capability of one of the two processors while completely disabling the other one.

First, we notice that when we get an improvement using two processors, we obtain a proportional improvement using HyperThreading. Even if the improvement is smaller, it is still significative. As mentioned in section 3.2, this is due to the parallelization scheme used that does different operations in each thread to avoid contention in the functional units of the processor.

Three of the implementations achieve enough coding speed to be used for real-time video coding (more than 24 frames per second) when they are used in a multiprocessor architecture. However, this is not the case using HyperThreading.

The *pthreads1* implementation is the one which obtains the best performance, and it is also the most difficult one to implement and the one which generates the most maintainability problems due to the great code restructuring necessary with respect to the sequential version. The performance advantage that it obtains with respect to the other simpler implementations is very small and does not justify the increased complexity. The programming effort needed to obtain a little speedup with respect to *openmp2* is too big to be justified. Specifically, the introduction of explicit synchronization is problematic due to the difficulty to obtain a correct implementation and the danger of introducing hard to detect errors like data races.

The *openmp1* implementation obtains the worst results, taking even more time than the sequential version to execute. It is important to note that this version is equivalent to the *pthreads1* version, so the decrease in performance is caused only by the transformations carried out by the compiler and the OpenMP support library. Additional tests suggest that the locks provided by the Intel OpenMP support library have worse behaviour than Pthreads mutexes under heavy contention, like in our parallelization scheme.

The *openmp2* and *pthreads2* implementations obtain practically the same times. Amongst these two, the first one is considerably easier to implement and more legible because, as seen above, the needed changes to the sequential code for *openmp2* are limited and easily comprehensible, while the *pthreads2* requires a certain restructuring and the introduction of new data structures. Also, the same code of *openmp2* can be used for both a sequential and a parallel versions, which is very useful for debugging during the development phase.

6. Conclusions

In this work we have improved the implementation of the video encoder presented in [7]. The applied techniques, Pthreads and OpenMP, have been evaluated and compared in terms of execution time of the produced program, difficulty to accomplish the parallelization, and readability and maintainability of the resulting code.

Using OpenMP instead of manual Pthreads parallelization has clear advantages with respect to ease of use and legibility of the resulting code. This is specially evident when converting sequential code into parallel code.

From our results, we can infer that the optimum implementation strategy depends on the technology that is going to be used. The more natural forms to solve the problem in

each one of the tested technologies have obtained the best results (*openmp2* y *pthreads1*). The technique which best result have obtained in general has got a very poor result when applied to a technology that is not usually used in that way and is not optimized for it.

The poor result obtained by *openmp1* points to a deficient implementation of the synchronization primitives in Intel's OpenMP support library.

On the other hand, the good results obtained by *openmp2* and *pthreads2*, practically the same and very near to those of the best implementation makes us believe that the cost of creating and destroying many threads is not very big compared to its alternative based in synchronization.

As future work, it would be interesting to compare the result obtained using other compilers which support OpenMP as well as SSE intrinsics.

References

[1] IEEE P1003.1c-1995: Information technology-portable operating system interface (POSIX), 1995.

[2] FCD 14495, lossless and near-lossless coding of continuous tone still images (JPEG-LS), 1997. ISO/IEC JTC1/SC29 WG1 (JPEG/JBIG).

[3] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[4] S. Battista, F. Casalino, and C. Lande. MPEG-4: A multimedia standard for the third millenium, part 1. *IEEE Multimedia*, 6(4):74–83, October 1999.

[5] S. Battista, F. Casalino, and C. Lande. MPEG-4: A multimedia standard for the third millenium, part 2. *IEEE Multimedia*, 7(1):76–84, January 2000.

[6] G. Bernab´e, J. Garc´ıa, and J. Gonz´alez. Reducing 3D Wavelet transform execution time through the streaming SIMD extensions. In *11th Euromicro Conference on Parallel Distributed and Network based Processing*, February 2003.

[7] G. Bernab´e, J. Gonz´alez, and J. M. Garc´ıa. An efficient 3D Wavelet transform on Hyper-Threading technology. Technical report, Universidad de Murcia, 2004.

[8] G. Bernab´e, J. Gonz´alez, J. M. Garc´ıa, and J. 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*, pages 226–231, November 2000.

[9] G. Bernab´e, J. Gonz´alez, J. M. Garc´ıa, and J. Duato. Memory conscious 3D Wavelet transform. In *28th Euromicro Conference, Multimedia and Telecommunications Track*, pages 108–113. IEEE, September 2002.

[10] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu. Automatic detection of parallelism: A grand challenge for high performance computing. *Parallel and Distributed Technology: Systems and Applications, IEEE*, 2(3):37–47, 1994.

[11] O. A. R. Board. OpenMP C and C++ application program interface, version 2.0, March 2002.

[12] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE Micro*, 17(5):12–24, 1997.

[13] B.-J. Kim and W. A. Pearlman. An embedded wavelet video coder using three-dimensional set partitioning in hierarchical trees (SPIHT). In *Designs, Codes and Cryptography*, pages 251–260, 1997.

[14] R. Kutil and A. Uhl. Optimization of 3D Wavelet decomposition on multiprocessors. *Journal of Computing and Information Technology (Special Issue on Parallel Numerics and Parallel Computing in Image Processing, Video Processing, and Multimedia)*, 8(1):31–40, 2000.

[15] J. L. Lo, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, 1997.

[16] W. Magro, Petersen, P., and S. Shah. Hyper-Threading technology: Impact on compute-intensive workloads. *Intel Technology Journal*, 6(1):58–66, February 2002.

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

[18] P. Marcuello and A. Gonzalez. Exploiting speculative thread-level parallelism on a SMT processor. In *HPCN Europe*, pages 754–763, 1999.

[19] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–15, February 2002.

[20] T. Sikora. MPEG digital video–coding standards. *IEEE Signal Processing Magazine*, 14(5):82–100, September 1997.

[21] C. Tenllado, C. Garcia, L. Piñuel, M. Prieto, and F. Tirado. Exploiting multilevel parallelism within modern microprocessors: DWT as a case study. In *Int. Meeting on Vector and Parallel Processing (VECPAR'04)*, June 2004.