Available online at www.sciencedirect.com
ScienceDirect
Parallel Computing 33 (2007) 54–72
PARALLEL COMPUTING
www.elsevier.com/locate/parco

An efficient implementation of a 3D wavelet transform based encoder on hyper-threading technology

Gregorio Bernabé [a,*], Ricardo Fernández [a], Jose M. García [a], Manuel E. Acacio [a], José González [b]

[a] Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Campus de Espinardo, 30080 Murcia, Spain
[b] Intel Barcelona Research Center, Intel Labs, Barcelona, 08034 Barcelona, Spain

Received 3 June 2005; received in revised form 16 June 2006; accepted 28 November 2006
Available online 22 December 2006

Abstract

Video medical compression algorithms based on the 3D wavelet transform obtain both excellent compression rates and very good quality, at the expense of a higher execution time. The goal of this work is to improve the execution time of our 3D Wavelet Transform Encoder. We examine and exploit the characteristics and advantages of a hyper-threading processor. The Intel Hyper-threading Technology (HT) is a technique based on simultaneous multi-threading (SMT), which allows several independent threads to issue instructions to multiple functional units in a single cycle. In particular, we present two approaches: data-domain and functional, which differ in the way that the decomposition of the application is performed. The first approach is based on data division, where the same task is performed simultaneously by each thread on an independent part of the data. In the second approach, the processing is divided in different tasks that are executed concurrently on the same data set. Based on the latter approach, we present three proposals that differ in the way that the tasks of the application are divided between the threads. Results show speedups of up to 7% and 34% by the data-domain and functional decomposition, respectively, over a version executed without hyper-threading technology. Finally, we design several implementations of the best method with Pthreads and OpenMP using functional decomposition. We compare them in terms of execution speed, ease of implementation and maintainability of the resulting code.

Keywords: 3D Wavelet Transform; Hyper-threading; Pthreads; OpenMP

1. Introduction

In the last few years, there has been a considerable increase in the volume of medical video generated in hospitals. Its strict regulations and quality constraints make research into compression techniques specifically oriented to this class of video an interesting area. Furthermore, the application of the wavelet transform [18] has also become an important development. Wavelet transform has been mainly applied to image or video

* Corresponding author.
E-mail address: gbernabe@ditec.um.es (G. Bernabé).

0167-8191/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2006.11.011

compression. Several coders have been developed using 2D wavelet transform [2,21,23,30,32], or 3D wavelet transform to efficiently approximate 3D volumetric data [29] or to code video sequences [15,22]. Nowadays, the standard MPEG-4 [4,5] supports an ad-hoc tool for encoding textures and still images based on a wavelet algorithm.

In previous works [6,8–11], we have developed and improved an encoder for medical video based on the 3D Wavelet Transform. Our encoder achieves high-compression ratios with excellent quality, such that medical doctors are unable to distinguish between the original and the reconstructed video. Also, the execution time achieved with all presented optimizations allows the real-time video compression and transmission in the 3D wavelet transform step.

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 Multi-threading (SMT [19,24,27]). However, to exploit these architectures successfully requires several threads or processes, which forces us to rethink the implementation of many algorithms.

Current examples of these architectures are the Intel® processors with hyper-threading technology [28]. This technology makes it feasible for a single processor to execute simultaneously two processes or threads, and adds relatively little complexity to the processor design.

In the case of parallelizing a video encoder, the automatic parallelization [3,13] methods available to us do not yield any benefit. It is necessary to use manual parallelization, especially to take advantage of the benefits that hyper-threading technology provides [7]. Manual parallelization poses a considerable burden in software development and it would be desirable to minimize this increase in complexity. There are a number of alternatives for implementing a parallel algorithm, each with different levels of difficulty, maintainability and flexibility.

In this paper, we attempt to take advantage of hyper-threading technology to improve the execution time of our 3D Wavelet Transform Encoder. We present different proposals that differ in the way the decomposition of the application is performed: data-domain or functional. Both are based on a multi-threading model to exploit the advantages of the hyper-threading technology. Results show that hyper-threading technology, when properly used, offers speedups of 1.07 and 1.31 for the data-domain and functional decomposition, respectively, for the optimal block size with respect to a single processor system without hyper-threading technology. We develop different implementations of the better functional decomposition proposal using OpenMP and Pthreads. We evaluate and compare these alternatives in terms of the execution time of the resulting programs, ease of implementation, maintainability and reusability of the produced code. We present an implementation using OpenMP which is easier to implement, more readable and nearly as efficient as the original implementation using Pthreads.[1]

Recently, and independently to our work, Tenllado et al. [31] have developed a related work parallelizing the discrete Wavelet transform (DWT) for its execution using hyper-threading technology and SIMD. Some of their conclusions are similar to ours: they also found that a functional decomposition is better suited for hyper-threading than a data decomposition, although they did not use functional decomposition when comparing SMP against hyper-threading. Unlike our work, they only use OpenMP for making the parallelization and the working set of their application is smaller than ours, because they apply the DWT just to bidimensional images instead of a tridimensional signal representing a video sequence.

The rest of this article is organized as follows: the background is presented in Section 2. Section 3 describes several approaches in order to reduce the execution times in the 3D Wavelet Transform Encoder using the hyper-threading technology architecture. The main details of each method are presented in depth and different implementations of the best method are developed using Pthreads and OpenMP. Experimental Results on some test medical video are analyzed in Section 4, and the benefits and limitations of various methods for the 3D Wavelet Transform Encoder will be discussed. Finally, Section 5 summarizes the work and draws the main conclusions.

[1] A preliminary implementation was already presented in [20].

2. Background

In this Section, we review the framework on which our enhancements have been built. We first outline the principal steps of our 3D Wavelet Transform Encoder to compress medical video. Then, we describe the main characteristics of hyper-threading technology.

2.1. The proposed 3D Wavelet transform based encoder

In previous works [9,10] we presented an implementation of a lossy encoder for medical video based on the 3D Fast Wavelet Transform (FWT). This encoder achieves high-compression ratios with excellent quality (PSNR around 41 for all frames), such that medical doctors are unable to distinguish between the original and the reconstructed video.

The 3D-FWT is computed by successively applying the 1D wavelet transform to the value of the pixels in each dimension. We have considered Daubechies W_4 (Daub-4) [18] as the mother wavelet function. We have chosen this function because some previous works have proved its effectiveness [9,10]. Fig. 1 shows the key processing steps involved in this 3D Wavelet-based Transform Encoder. The thresholding step uses the x-percentile method to discard those coefficients whose value does not provide enough information. The quantizer transforms the floating points coefficients into unsigned integer coefficients. We have proposed a quantizer where the number of bits needed by each pixel coefficient to be encoded depends on the sub-cube that this pixel belongs to. In general, the number of required bits depends on the number of the wavelet transformations performed to each sub-cube. In the entropy encoder, a run-length compression is applied to the binary representation of the integer coefficients, thus exploiting the presence of large chains of zeros and using a hexadecimal representation. Finally, an arithmetic encoder is applied to increase the compression ratio even more without affecting the video quality.

One of the main drawbacks of using the 3D wavelet transform to code and decode medical video is its excessive execution time. Since three dimensions are exploited in order to obtain high-compression rates, the working set becomes huge and the algorithm is limited by memory (memory bound).

Hence, we developed a memory conscious 3D FWT that exploits the memory hierarchy by means of blocking algorithms [11,8], thus reducing the final execution time. In particular, we proposed and evaluated two blocking approaches that differ in the way that the original working set is divided. In the first approach, we proposed to divide the original sequence into several sub-cubes, whereas in the second approach the original cube is divided into several rectangles. In both proposals, the wavelet transform is independently applied to each of these sub-cubes or rectangles, respectively, as we can see in Fig. 2. In [11,8], both a cube partitioning and rectangle partitioning blocking strategies were presented, which were able to introduce overlapping or not in each case.

Both approaches had the same drawback: they caused some quality degradation because they isolated the pixels at the borders of the sub-regions from those that surround them. In the case of a mother function with four coefficients [18], 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 sub-regions created by blocking, they depend on pixels which are outside the sub-region.

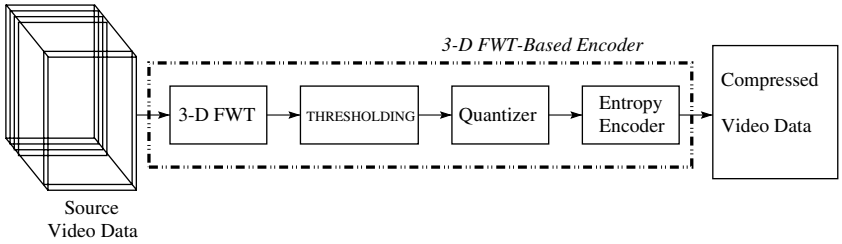

Fig. 1. 3-D FWT-based encoder processing steps.

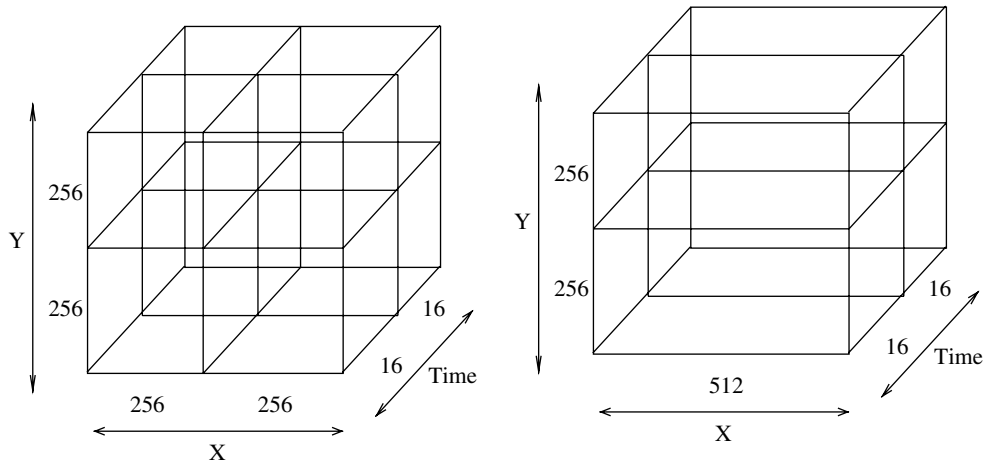

Fig. 2. Cube and Rectangular approaches.

The non-overlapping strategies use the values 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, especially in the limits between regions where image discontinuities appear.

The overlapping strategies use the pixels from the adjacent blocks, thus avoiding the quality loss but exploiting less spatial locality. This approach also implies some redundant calculations. It is possible to store these results and reuse them for the next block, and it has been shown that this is beneficial at least in the case of rectangle partitioning [11,8]. We call the last strategy rectangular blocking with overlapping and operation reuse, and it will be the strategy assumed for the rest of this work.

Moreover, we attempted to take advantage of the Streaming SIMD Extensions efficiently [12] by performing a manual vectorization using the compiler intrinsic instructions available in the Intel®[2] C/C++ Compiler [17]. 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 wavelet transform. The calculation of four low-pixels and four high-pixels of the 3D-FWT requires 32 floating point multiplications, 24 floating point additions and 56 instructions. Using SSE, it is possible to perform those operations with only 15 instructions [6,8].

We also employed other classical methods like data prefetching and loop unrolling (in the time dimension) [6,8]. Finally, we examined the source code in order to exploit the temporal and spatial locality in the memory cache. A method to enhance the locality of the memory hierarchy, based on the way that the wavelet transform is computed in the x and y dimensions, was presented taking into account that the mother wavelet function is the Daubechie's of four coefficients (Daub-4). This method, called *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. The speedup obtained with this optimization is very important, especially when combined with the use of SSE instructions for the calculations in the Y dimension.

Results showed that the rectangular overlapped blocking approach with the different optimizations provided the best execution times for among all tested algorithms, achieving a 5× speedup over the non-blocking non-overlapped wavelet transform, while maintaining both the compression ratio and the video quality of the original encoder [6,8].

2.2. SMT and hyper-threading technology architecture

An SMT processor can execute several threads simultaneously. It shares most hardware resources like caches, functional units, branch predictors, etc., and replicates only those resources needed to store the status

[2] Intel is a registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

of every process, such as architectural registers. In this way, a single physical processor is considered the operating system and applications as two or more 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. Moreover, threads can also execute in parallel if they use different functional units of the processor. It is possible that, in some cases, the execution time of a sequential application increases, but that the overall productivity of the system is improved.

Intel has introduced one implementation of SMT called hyper-threading [28] in its high-performance ×86 processors like Xeon and Pentium IV, obtaining improvements in execution time around 30% in some cases [25]. Hyper-threading technology allows one physical processor to appear as two logical processors.

From an architectural point of view, the introduction of hyper-threading technology means that the operating system and the user programs can schedule processes or threads on the logical processors as if they were multiple physical processors. Like dual-core or dual-processor systems, two applications or threads can be executed in parallel in a processor with hyper-threading technology. However, from a micro-architecture perspective, hyper-threading technology implies that instructions from different logical processors are executed on shared resources in a concurrent way.

From the programming point of view, hyper-threading technology represents a challenge to develop parallel applications sharing resources like execution units or caches. These applications can be developed taking into account the resources that are used by one thread and the resources that could be used by the another one.

2.3. Pthreads and OpenMP

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 on how to exploit concurrency at the expense of increasing the difficulty to use it.

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

On the other hand, OpenMP [14] is a 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. The OpenMP programmer can specify the appropriate scheduling policy in the program through the schedule clause in work sharing directives. If the schedule is not specified, static is assumed in most implementations. In addition, the scheduling policy can be specified at run-time

OpenMP allows an incremental parallelization of programs. Therefore, it is especially 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 by just ignoring the OpenMP directives when compiling the sequential program. This is especially useful in the development phases to allow easier debugging.

3. Design alternatives to properly exploit the hyper-threading technology

In this section we present two different approaches to improve the execution times of the 3D Wavelet Transform Encoder using the hyper-threading technology.

3.1. Multi-threading using data-domain decomposition

The data domain decomposition method involves partitioning the application data such that each thread works concurrently on an independent part of the data. In this method, each thread is statically assigned to

process a block of frames of the sequence of video. Assuming that the computational load corresponding to different blocks is similar, the several threads will finish the task at roughly the same time.

For example, an input sequence of 64 frames of video can be divided into two blocks of 32 frames to apply each of the steps of the 3D Wavelet Transform Encoder. Each block, consisting of frames of 512×512 pixels, is a unit that can be encoded independently. The sequence is split in blocks of frames because the time dimension in the wavelet transform should be applied on a set of frames and can not be applied on a frame independently because the good characteristics of the wavelet transform would be lost.

3.2. Multi-threading using functional decomposition

In functional decomposition, the processing is divided into different tasks that are executed in a pipelined way on the same data set. This functional decomposition implemented on a parallel architecture requires one to look for the functionally independent tasks of the 3D Wavelet Transform Encoder. Each thread is assigned to a different task so they can run in parallel. At the end, the final results are combined if necessary.

Here, we propose different approaches to perform a multi-threaded functional decomposition on the 3D Wavelet Transform Encoder.

3.2.1. Memory prefetch and 3D Wavelet transform

Memory latency has become one of the major bottlenecks in achieving high-performance on modern processors. In the computation of the wavelet transform, the working set is quite huge but with predictable access patterns. In [6,8] we proposed to insert prefetch instructions to decrease memory latency in a single-threaded application. In an HT architecture this proposal can be extended to two threads: a computational and a prefetcher.

We therefore propose to split into functionally independent two tasks that are statically assigned to each thread to run concurrently. One thread executes the 3D Wavelet Transform Encoder, the master thread, while the second thread, the helper thread [16,33], carries out the prefetch instructions to put the necessary pixels for computing the wavelet transform into the cache. The helper thread must be activated by the master thread in order to perform the prefetch instructions when necessary. In order to synchronize both the master and helper thread, spin-waiting loop with a conditional variable or critical regions have been considered. The first option may involve long delays because the helper thread is constantly consuming resources of the processor without executing any useful instructions that can make performance worse in a multi-threaded processor.

With this in mind, we have implemented the helper thread using mutexes. The master thread creates the helper one, which consists of an infinite loop performing prefetching instructions. The helper thread is based on a critical region locked by the master thread, which is waiting for the address where it starts to perform the prefetch instructions. When the master thread unlocks the critical region, it provides the address of the data to be prefetched by the helper thread.

3.2.2. 3D Wavelet transform and quantization

In the 3D Wavelet Transform Encoder, the main computational cost falls in the following tasks: the computation of the 3D Wavelet Transform and the transform of the floating points wavelet coefficients into unsigned integers, specifying the number of bits needed to encode each of the wavelet layers in each block of frames in the quantizer step. Moreover, we note that the workload of the two tasks is quite different. The 3D Wavelet Transform is entirely floating point computation-bound and many of the instructions to compute the wavelet coefficients are calculated in the MMX/SSE/SSE2 execution unit. However, the integer execution unit is idle most of the time. On the other hand, in the quantizer step, once the floating points coefficients have been transformed into unsigned integers, most of the instructions are computed in the integer unit whereas the MMX/SSE/SSE2 execution unit is idle most of the time.

Hence, different threads are assigned to the 3D Wavelet Transform and to the quantizer step. The thresholding phase has been integrated in the 3D Wavelet Transform step because it takes less CPU time than the quantizer step.

The wavelet thread acts as a master thread, creates the quantizer thread and computes the wavelet transform in each block of the current video sequence. Once each block of data has been set up, the wavelet

thread wakes up the quantizer thread, which operates on the resulting wavelet coefficients as illustrated in Fig. 3b. As the quantizer thread has a higher processing time than the wavelet thread, the wavelet thread takes roughly one-quarter of the CPU time while the quantizer thread takes three-quarters of the CPU time. Therefore, the output of the wavelet thread needs to be copied in a different buffer from that where the wavelet transform has been calculated, because the wavelet transform always operates on the same buffer to compute the different blocks of the video sequence. In this way, we can avoid stopping the wavelet thread until the quantizer threads completely executes the current block, as in the serial program as we can observe in Fig. 3a. This implies an increase in the memory requirements of the multi-threaded version. On the other hand, we can implement an equivalent approach avoiding the data copying between the buffers used by the two threads. We use two buffers (A, B), which are used in an alternating way by the two threads. When the wavelet thread works on buffer A, the quantizer works on buffer B. At the next step (given that the 2 threads work synchronized, in steps) the quantizer works on block A (using the results produced at the previous step by the wavelet thread) and the wavelet starts working on a new data block on buffer B. In the same way as the previous proposal, we have used mutexes to synchronize the two threads.

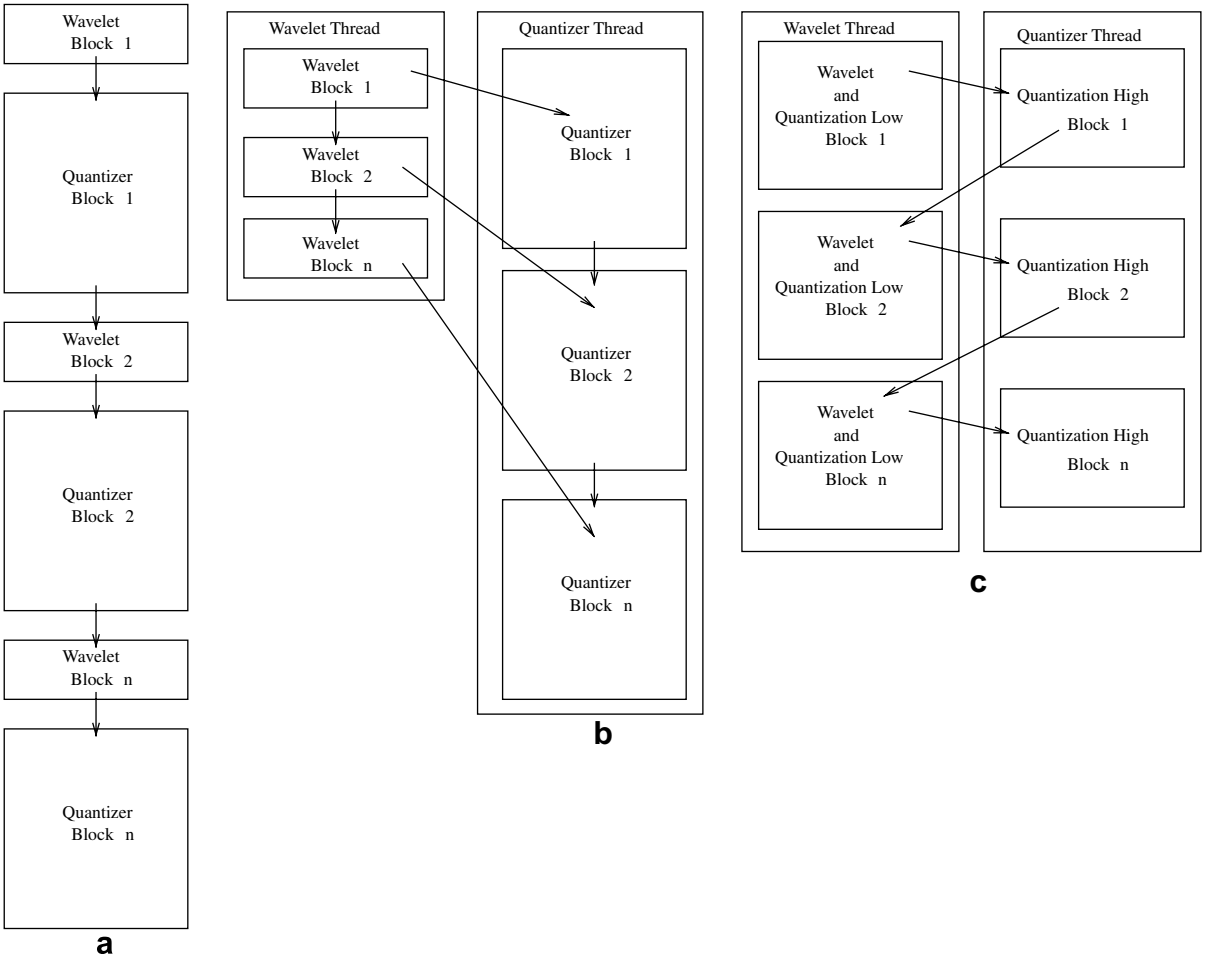

Fig. 3. Three methods for 3D Wavelet Transform Encoder. (a) Sequential mode; (b) Functional two-threaded mode; (c) Functional better-balanced two-threaded mode.

3.2.3. 3D Wavelet transform plus quantization low and quantization high

This proposal is based on the previous one. Moreover, the fact that the workload is not well balanced between the wavelet thread and the quantizer thread requires deep study on how the wavelet transform and quantizer work on the video sequence.

The wavelet transform can be implemented by quadrature mirror filters (QMF), $G = g(n)$ and $H = h(n) n \epsilon Z$. H corresponds to a low-pass filter and G is a high-pass filter. For a more detailed analysis of the relationship between wavelets and QMF see [26]. The filters H and G correspond to one step in the wavelet decomposition. Given a discrete signal, s, with a length of 2^n, at each stage of the wavelet transformation, the G and H filters are applied to the signal and the filter output down-sampled by two, generating two bands: low and high. The process is then repeated on the low-band to generate the next level of decomposition and so on. This procedure is referred to as the 1D Fast Wavelet Transform (1D-FWT). It is not difficult to generalize the one-dimensional wavelet transform to the multi-dimensional case [26]. The wavelet representation of an image, $f(x, y)$, can be obtained with a pyramid algorithm. It can be achieved by first applying the 1D-FWT to each row of the image and then to each column. That is, the G and H filters are applied to the image in both the horizontal and vertical directions. The process is repeated several times as in the one-dimensional case. This procedure is referred to as the 2D Fast Wavelet Transform (2D-FWT). As in 2D, we can generalize the one-dimensional wavelet transform for the three-dimensional case. Instead of one image, there is now a sequence of images. Thus a new dimension has emerged, time (t). The 3D-FWT can be computed by successively applying the 1D wavelet transform to the value of the pixels in each dimension.

When the first wavelet transform iteration is applied on a block of the video sequence, from the time dimension point of view, one half of the original frames represents a low-band and the other half is a high-band. After applying the wavelet transform to the X and Y dimensions, only an eight of the pixels of the original video sequence (the one that represents the low–low–low band), is needed for the second iteration of the wavelet transform, while the quantizer can work in the rest of the sequence without dependencies and risk of races. For example, for a block of a video sequence of 16 frames of 512×32 pixels, the first iteration of the wavelet transform to the time dimension generates two bands, low and high, of eight frames of 512×32 pixels. Then, the application of the wavelet transform to the X and Y dimensions outputs four bands (low–low, low–high, high–low and high–high) of 256×16 pixels in each frame. Therefore, the second iteration of the 3D-FWT only needs the pixels which belong to the band low–low–low of eight frames of 256×16 pixels and the quantizer can work in the rest of the sequence.

Now, once the first iteration of the wavelet transform is computed by the wavelet thread, it activates the quantizer thread on the high-band. Then, while the high-band wavelet coefficients are being quantized, the wavelet thread computes the second iteration of the wavelet transform and the respective quantization. Therefore, the wavelet thread performs the wavelet transform plus the quantization on the low-band whereas the quantizer thread executes the quantization on the high-band.

In the previous approach, the output of the wavelet thread had to be copied in a different buffer from that where the wavelet transform had been computed. Now this copy is not necessary because the quantizer thread ends its work on the high-band before the wavelet thread starts with a new block. Therefore, both threads can run on the same buffer in a concurrent way and the CPU time is better balanced between the two threads than in the previous proposal, as we can observe in Fig. 3c. Furthermore, the buffer is protected with a mutex to assure that the wavelet thread does not start with another block until the quantizer thread finishes with the current block.

In an N-level wavelet decomposition, the scheme proposed would be maintained. In general, each iteration of the wavelet transform creates a low- and a high-band. The wavelet thread is always devoted to apply the wavelet transform and the quantization of the low-band, and the other thread is in charge of the quantization of the high-band. Although the workload of the two threads decreases for each iteration of the wavelet transform, the proportion between the two threads is maintained and the workload is well balanced.

3.3. Implementation issues

We have implemented the previous proposals using Pthreads, which implies a low-level programming which allows for great control and flexibility to decide exactly how to parallelize the program. In the 3D

Wavelet Transform plus Quantization High and Quantization Low 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 worker thread can not 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 worker thread has finished it.

To achieve this partial ordering we have used two locks which are acquired alternately by both threads. In Fig. 4, we show a scheme of the implementation.

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

- Extract the code related with the worker thread and put it in an independent function. This is necessary because Pthreads requires the entry point for every thread to be a function.
- 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 passing a pointer to that structure. Because we extract the worker 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 Fig. 5, is very different to the sequential code and it does not seem obvious which things are related to the algorithm and which are related to the parallelization.

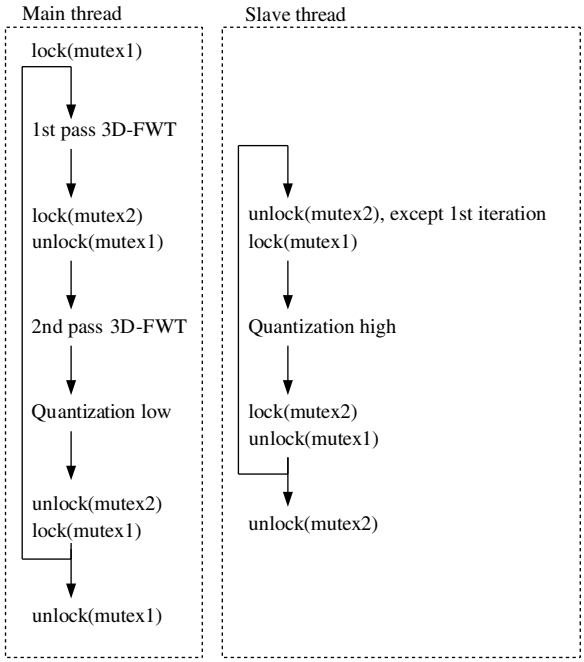

Fig. 4. First parallelization scheme.

```
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);
}
```

Fig. 5. Code snippet for the *pthreads1* scheme.

3.4. Using OpenMP to ease the implementation

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

3.4.1. Using OpenMP like pthreads

OpenMP also allows using critical regions and locks. Hence, it is possible to use the same scheme as previously described and implement the algorithm using OpenMP, as described in Fig. 4 (*openmp1*). However, this approach is not much better and the result is not more comprehensible.

To make this implementation, it is necessary to perform some of the manual transforms used in the previous scheme. It is not necessary to create new structures nor to extract the code of the worker thread to another function. However, it is still necessary to duplicate the loops, restructuring the code considerably, as can be seen in Fig. 6.

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 correctly as sequential code. This way of using OpenMP is not the intended and does not take advantage of the features provided by it.

```
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);
    }
  }
}
```

Fig. 6. Code snippet for the *openmp1* scheme.

3.4.2. 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 required directives.

In Fig. 7, we show the scheme of the implementation (*openmp2*) of a block codification. In each iteration for each block of pixels, (it has been determined, see previous papers [11,8], that $512 \times 64 \times 16$ is the optimal blocking size), an independent worker thread 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.

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 Fig. 8.

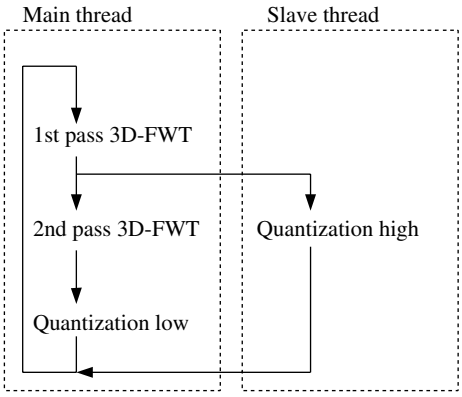

Fig. 7. Second parallelization scheme.

```
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 */
    }
  }
}
```

Fig. 8. Code snippet for the *openmp2* scheme.

Depending on the OpenMP support of the compiler, this implementation may create a new thread for each block, unlike the pthreads1 and openmp1 implementations. Fortunately, most compilers reuse threads when possible instead of destroying and recreating them unnecessarily. In particular, the compiler that we have used for our tests uses a threadpool of worker threads to implement parallel sections.

The resulting code is very similar to the sequential implementation. Indeed, if we ignore the OpenMP directives, it is possible to execute it as a sequential program and obtain correct results. This proves the simplicity of the parallelization using OpenMP and its maintainability and reusability properties. Notably, this implementation does not need any manual synchronization, unlike *pthreads1* and *openmp1* implementations which required locks managed by the programmer.

3.4.3. Using Pthreads like OpenMP

The same scheme from Fig. 7 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 had to perform some of the same changes that we made for the *pthreads1* version. We had not 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 had to extract the body of the worker thread to an independent function and we 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 Fig. 9. The implementation is not very complicated because it consists mostly of structural changes

```
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);
  }
}
```

Fig. 9. Code snippet for the *pthreads2* scheme.

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.

Note that this implementation does create a new thread for each iteration, unlike *openmp2* when compiled with most OpenMP compilers. To avoid that, we would need a much more complex design which creates a thread and uses manual synchronization to maintain the partial order, like *pthreads1*.

4. Experimental results

In this section we present the experimental results we have obtained for the different versions of the 3D Wavelet Transform. The evaluation process has been carried out on a 2 GHz Intel Xeon™[3] processor with hyper-threading technology. The main properties of the memory hierarchy are summarized in Table 1. The operating system kernel was Linux 2.4.18-3smp. The programs have been written in the C programming language and compiled with the Intel C/C++ Compiler v.6.0.

We have measured the elapsed completion time of the different proposals presented in Section 3 using operating-system provided timers for the following configurations:

1. Single-threaded execution on a single processor HT system, disabling the hyper-threading technology.
2. Dual-threaded execution on a single processor HT system.
3. Dual-threaded execution on a dual processor HT, disabling the hyper-threading technology. We have performed this configuration to evaluate the potential of hyper-threading technology. However, these results cannot be achieved by a HT processor because it only duplicates the architectural state whereas the processor execution resources are shared between the two logical processors. Therefore, results obtained on a dual processor constitute an upper bound for a HT processor. The theoretical limit of the speedup using both threads of a hyper-threaded processor is 1.66. It has to do with the number of the instruction queues and the rate with the instructions can be issued.

[3] Xeon is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Table 1
Description of the memory hierarchy

Level 1	L1 inst cache, Trace Cache 12 KB
	L1 data cache, 8 KB, 4-way, 64 byte line
Level 2	L2 cache, 512 KB, 8-way, 128 byte line
Level 3	1024 Mbytes DRAM

In order to compute the speedup, first of all we computed the arithmetic mean of 100 executions for each one of the configurations. Then, the speedup is computed as the ratio of the arithmetic means. The variation coefficient (the typical deviation divided by the arithmetic mean), which quantifies the dispersion of the samples is always smaller than 2.5% for all configurations.

4.1. Data-domain decomposition

The tests have been performed on a *heart* video medical sequence of 64 frames of 512×512 pixels coded in gray scale (8 bits per pixel).

Fig. 10 shows speedups with respect to the single-threaded application of the 3D Wavelet Transform Encoder presented in [6,8] to compute the *heart* video sequence. *X*-axis represents the different block sizes evaluated from $512 \times \mathbf{16} \times 16$ to $512 \times \mathbf{512} \times 16$. Baselines times for the sequential case are 4.32, 4.31, 4.44, 4.54, 4.67 and 4.81 for each block size, respectively.

First of all, we can observe that the application receives a small benefit from the introduction of hyper-threading technology achieving a speedup between 1.02 and 1.07 compared to the single-threaded execution. The optimal block size ($512 \times 16 \times 16$) obtains a speedup of 1.07. This optimal block size is the configuration which we have obtained the lowest execution time.

However, this data domain decomposition has a limited potential of performance improvement in a HT architecture. Some of the problems that limit the performance include:

- The threads work on different and independent parts of the data at the same time. Therefore, enough memory must be allocated to store and process different blocks of frames, independently. This implies that the two simultaneously operating threads both enter their memory intensive stage at approximately the same time and put more pressure on the memory hierarchy.
- Each thread performs the same tasks in a concurrent way. This produces a resource contention because different threads compete for the same resources in a processor with hyper-threading technology.

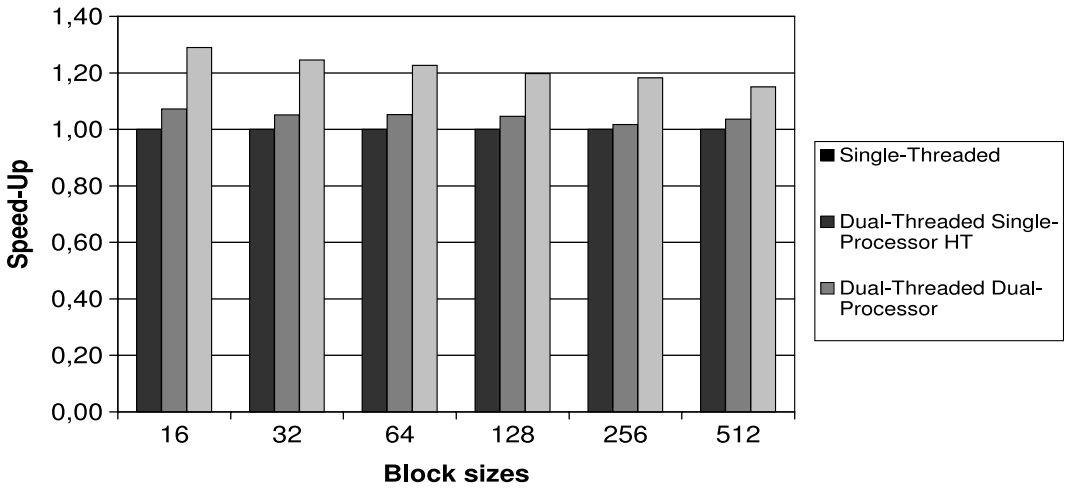

Fig. 10. Speedups for multi-threading using data-domain decomposition.

However, the workload is well balanced between the two logical processors (50%), and the programming is easy to develop because both threads work on independent parts of the data, without the necessity of mutexes to protect critical sections.

The dual-threaded execution on a dual processor achieves a speedup between 1.15 and 1.29 (1.29 for the optimal block size $512 \times 16 \times 16$) compared to the single-threaded execution. Hence, HT contribution, although smaller, is still substantial, taking into account that only the architectural state has been duplicated in a HT processor.

4.2. Functional decomposition

The tests have been performed on a *heart* video medical sequence of 64 frames of 512×512 pixels coded in gray scale (8 bits per pixel).

Fig. 11 shows speedups reached by the 3D Wavelet Transform Encoder with the rectangular overlapped blocking approach computing the *heart* video sequence with the two functional proposals described in Section 3: 3D Wavelet Transform and Quantization referred to the figure as *High&Low* and 3D Wavelet Transform plus Quantization High and Quantization Low, called *High*. *X*-axis represents the different block sizes evaluated from $512 \times \mathbf{16} \times 16$ to $512 \times \mathbf{512} \times 16$. Baselines times for the rectangular overlapped blocking approach are 4.32, 4.31, 4.44, 4.54, 4.67 and 4.81 for each block size, respectively.

Results for Memory Prefetch and 3D Wavelet Transform approach are not presented because, although different alternatives changing the distance of the prefetch have been implemented and tested, the speedups were tiny (1.01 or 1.02) or the single threaded execution even achieved better results. This is due to the fact that the master thread of the application is actively working on most 100% of the time and the helper thread is created and waiting, consuming a part of the processor resources. This causes resource contention and slows down the execution of the master thread. In addition, the Intel Xeon processor has a built-in hardware prefetch mechanism for loading data blocks from memory which conflicts with our *_mm_prefetch* instructions. Memory accesses are quite regular in the wavelet algorithm, hence the hardware prefetch unit works very well.

With regard to *High&Low*, we can observe that speedups with hyper-threading technology are between 1.12 and 1.14 compared with the single-threaded execution. The optimal block size ($512 \times 16 \times 16$, the one that provides the lowest execution time), obtains a speedup of 1.13. These figures are higher than the data-domain decomposition proposal, but the workload is not well-balanced between the two logical processors. However, in this approach it is not necessary to allocate memory to store and process two independent blocks of frames and there are not two concurrent tasks conflicting for the same resources, like in data-domain decomposition. Now, most of the time, the two tasks are using different execution units (the wavelet thread uses mainly the

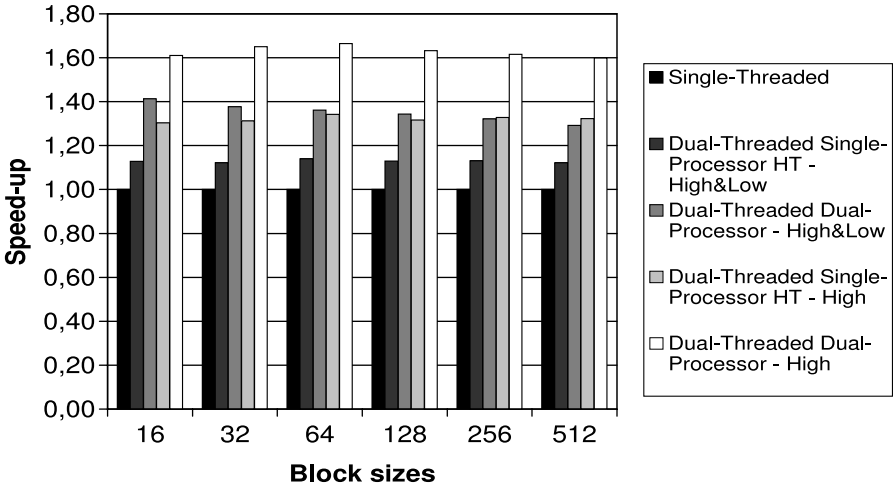

Fig. 11. Speedups for multi-threading using functional decomposition.

MMX/SSE/SSE2 execution unit while the quantizer thread uses the integer unit). This avoids the resource contention between the two threads. However, the workload is not well-balanced because the quantizer thread takes most of the CPU time (three-quarters). Moreover, it is necessary to copy the output of the wavelet thread in a different buffer from that where the wavelet transform has been computed, allowing it to be processed later by the quantizer thread.

In order to obtain a well-balanced workload between the wavelet thread and the quantizer thread we proposed the *High* approach, where the wavelet thread consumes 60% of the CPU time whereas the quantizer thread consumes the remaining 40%. Speedups with hyper-threading technology are between 1.30 and 1.34, and 1.31 for the optimal block size $512 \times 32 \times 16$. These results indicate a significant improvement over the previous proposals. This is due to the effective use of the shared resources of the processor between the two threads. We also avoid copying the output of the wavelet thread in a different buffer, as in the previous proposal, and most of the time the two threads can run concurrently. In fact, the *High* approach produces less L1 and L2 misses (speedups of 1.81 and 2.13, respectively) than the *High&Low* approach justifying the decrease in the execution time. This strategy implies significant changes in the original program code, which are not easy to make without a deep study.

In Fig. 11, we can also observe speedups for the dual-threaded execution on a dual processor for the two previous approaches. Obviously, results are better than dual-threaded execution on a single processor HT system, and obtain speedups from 1.29 to 1.41 for the *High&Low* approach and from 1.60 to 1.67 for the *High* approach, compared with the single-threaded execution. The optimal block size obtains 1.41 ($512 \times 16 \times 16$) and 1.65 ($512 \times 32 \times 16$), respectively. These results confirm that the HT's throughput increase is substantial and significant compared to the resources that have been duplicated. As we have shown in the *High* approach, the main advantage of HT lies in its ability to boost utilization by dynamically scheduling functional units among multiple threads.

4.3. Implementation issues

In this section, we present the evaluation of the performance of each one of the *High* implementations developed in Section 3.3 with respect to their execution time and maintainability. In Fig. 12, we show the execution times for the different implementations using two independent processors and a single processor with hyper-threading technology for the optimal block size $512 \times 32 \times 16$.

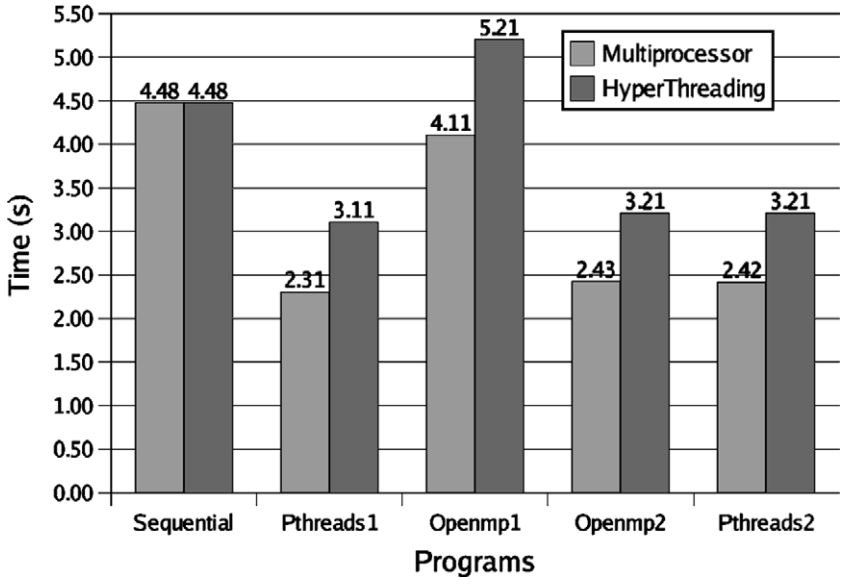

Fig. 12. Execution times for the different implementations.

Table 2
Additional lines of code required for each parallel implementation, compared with the sequential code

pthreads1	78
pthreads2	66
openmp1	29
openmp2	15

Firstly, as we showed in the previous section, we obtain a proportional improvement using hyper-threading. Even if the improvement is smaller, it is still significant. This is due to the parallelization scheme used, which performs different operations in each thread to avoid contention in the functional units of the processor.

The *pthreads1* implementation is the one which obtains the best performance, and it is also the most difficult to implement and the one which generates the most maintainability problems, due to the great code restructuring necessary with respect to the sequential version. In fact, as we can observe in Table 2, the *pthreads1* version is the parallel implementation which needs the highest number of additional lines of code compared with the sequential program. 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 of obtaining a correct implementation and the danger of introducing hard-to-detect errors like data races.

The *openmp1* implementation gives 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. Of the two, the first is considerably easier to implement and more legible because, as seen before, the required 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. In fact, the *openmp2* only adds 15 lines of code with respect to the sequential version, while the *phthreads2* adds 66 lines of code. Also, the same code of *openmp2* can be used for both sequential and parallel versions, which is very useful for debugging during the development phase.

5. Conclusions

In this work, we have focused on using the hyper-threading technology on our 3D Wavelet Transform Encoder when it is applied to encode medical video. To the best of our knowledge, this is the first work that evaluates a 3D Wavelet Transform Encoder on an SMT architecture like Intel's processors with HT. We have presented and evaluated four alternative schemes for the decomposition.

Firstly, we have used the data-domain decomposition to allow two threads to run concurrently on independent parts of the data. Results show an improvement from 1.02 to 1.07 compared to a single processor system, but this proposal is limited by the decomposition way.

In the second place, we have proposed the functional decomposition where the threads execute different tasks of the application concurrently on the same data set. Based on this decomposition, we have presented three methods: Memory Prefetch and 3D Wavelet Transform, 3D Wavelet Transform and Quantization, and 3D Wavelet Transform plus Quantization High and Quantization Low. In the first method, while one thread computes the wavelet transform another thread prefetches the pixels needed for this computation. The second method is based on a functional primary division where one thread performs the wavelet transform while another executes the quantization step. Finally, the last method is an optimization of the previous one, achieving a better workload balance and higher parallelism between the two threads. The first thread carries out the wavelet transform and the quantization on the low-band while the second one performs the quantization on the high-band. This method obtained the best results, from 1.30 to 1.34 of speedup, compared to a single-threaded execution on a single processor without hyper-threading technology.

We can conclude that the functional decomposition obtains better results than the data decomposition on HT processors for the 3D Wavelet Transform Encoder evaluated in this paper. This conclusion may apply to similar applications, such as MPEG-2 or MPEG-4. This is an open research area and it is part of our current and future work.

Finally, we have encoded the last functional decomposition using Pthreads and OpenMP. The different implementations 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 especially 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* and *pthreads1*). The technique which has obtained the best result in general has a very poor result when applied to a technology that is not usually used in that way, and therefore, 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 high 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.

Acknowledgements

We thank the anonymous reviewers for their valuable comments that have helped us to improve the quality of the paper. This work has been funded in part by the Spanish Ministry of Science and Technology and the Feder European Funds under grant TIC 2003-08154-C06-03. The video sequences have been donated by the Hospital Recoletas (Albacete, Spain).

References

[1] IEEE P1003.1c-1995: information technology-portable operating system interface (POSIX), 1995.
[2] M. Antonini, M. Barlaud, Image coding using wavelet transform, IEEE Transactions on Image Processing 1 (2) (1992) 205–220.
[3] U. Banerjee, R. Eigenmann, A. Nicolau, D.A. Padua, Automatic program parallelization, Proceedings of the IEEE 81 (2) (1993) 211–243.
[4] S. Battista, F. Casalino, C. Lande, Mpeg-4: a multimedia standard for the third millenium. Part 1, IEEE Multimedia 6 (4) (1999) 74–83.
[5] S. Battista, F. Casalino, C. Lande, Mpeg-4: a multimedia standard for the third millenium. Part 2, IEEE Multimedia 7 (1) (2000) 76–84.
[6] G. Bernabé, J.M. García, J. González, Reducing 3D wavelet transform execution time through the streaming simd extensions, in: Proceedings of the 11th Euromicro Conference on Parallel Distributed and Network based Processing, Genoa, Italy, February 2003.
[7] G. Bernabé, J.M. García, J. González, An efficient 3D Wavelet transform on Hyper-Threading technology. Technical report, UM-DITEC-2004-02. Universidad de Murcia, 2004.
[8] G. Bernabé, J.M. García, J. González, Reducing 3d wavelet transform execution time using blocking and the streaming simd extensions, Journal of VLSI Signal Processing 41 (2005) 209–223.
[9] G. Bernabé, J. González, J.M. García, J. Duato, A new lossy 3-d wavelet transform for high-quality compression of medical video, Proceedings of IEEE EMBS International Conference on Information Technology Applications in Biomedicine (November) (2000) 226–231.
[10] G. Bernabé, J. González, J.M. García, J. Duato, Enhancing the entropy encoder of a 3D-FWT for high-quality compression of medical video, Proceedings of IEEE International Symposium for Intelligent Signal Processing and Communication Systems (November) (2001).
[11] G. Bernabé, J. González, J. M. García, J. Duato, Memory conscious 3D wavelet transform, in: Proceedings of the 28th Euromicro Conference. Multimedia and Telecommunications, September 2002.
[12] A. Bik, M. Girkar, P. Grey, X. Tian, Efficient exploitation of parallelism on Pentium III and Pentium IV processor-based systems. Available from: <http://developer.intel.com/>.

[13] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, P. Tu, Automatic detection of parallelism: a grand challenge for high performance computing, Parallel and Distributed Technology: Systems and Applications, IEEE 2 (3) (1994) 37–47.
[14] O.A.R. Board, OpenMP C and C++ application program interface, version 2.0 (March) 2002.
[15] Y. Chen, W.A. Pearlman, Three-dimensional subband coding of video using the zero-tree method, Proceedings of SPIE - Visual Communications and Image Processing (March) (1996) 1302–1310.
[16] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, J. Shen, Speculative precomputation: long-range prefetching of delinquent loads, in: Proceedings of 28th International Symposium on Computer Architecture, 2001, pp. 14–25.
[17] I. Corporation, Intel C/C++ compiler for Linux. Available from: <http://www.intel.com/software/products/compilers/>.
[18] I. Daubechies, Ten Lectures on Wavelets, Society for Industrial and Applied Mathematics, 1992.
[19] S.J. Eggers, J.S. Emer, H.M. Levy, J.L. Lo, R.L. Stamm, D.M. Tullsen, Simultaneous multithreading: a platform for next-generation processors, IEEE Micro 17 (5) (1997) 12–24.
[20] R. Fernández, J.M. García, G. Bernabé, M.E. Acacio, Optimizing a 3D-FWT video encoder for SMPs and HyperThreading architectures, in: Proceedings of the 13th Euromicro Conference on Parallel Distributed and Network based Processing, 2004.
[21] M.L. Hilton, B.D. Jawerth, A. Sengupta, Compressing still and moving images with wavelets, Multimedia Systems 2 (3) (1994).
[22] B.J. Kim, W.A. Pearlman, An embedded wavelet video coder using three-dimensional set partitioning in hierarchical trees (spiht), Proceedings of Data Compression Conference (1997).
[23] A.S. Lewis, G. Knowles, Image compression using the 2-d wavelet transform, IEEE Transactions on Image Processing 1 (2) (1992) 244–256.
[24] J.L. Lo, J.S. Emer, H.M. Levy, R.L. Stamm, D.M. Tullsen, Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading, ACM Transactions on Computer Systems 15 (3) (1997) 322–354.
[25] W. Magro, P. Petersen, S. Shah, Hyper-threading technology: impact on compute-intensive workloads, Intel Technology Journal Q1 (2002).
[26] S. Mallat, A theory for multiresolution signal decomposition: the wavelet representation, IEEE Transactions on Pattern Analysis and Machine Intelligence 11 (7) (1989) 674–693.
[27] P. Marcuello, A. Gonzalez, Exploiting speculative thread-level parallelism on a SMT processor, in: Proceedings of HPCN Europe, 1999, pp. 754–763.
[28] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, M. Upton, Hyper-threading technology architecture and microarchitecture, Intel Technology Journal Q1 (2002).
[29] S. Muraki, Multiscale volume representation by a dog wavelet, IEEE Transactions on Visualization and Computer Graphics 1 (2) (1995) 109–116.
[30] J.M. Shapiro, Embedded image coding using zerotrees of wavelets coefficients, IEEE Transactions on Signal Processing 41 (12) (1993) 3445–3462.
[31] C. Tenllado, C. Garcia, L. Piuel, M. Prieto, F. Tirado, Exploiting multilevel parallelism within modern microprocessors: DWT as a case study, in: Proceedings of International Meeting on Vector and Parallel Processing (VECPAR'04), June 2004.
[32] M.w. Marcellin, M.J. Gormish, A. Bilgin, M.P. Boliek, An overview of JPEG-2000, in: Proceedings of Data Compression Conference, March 2000.
[33] C. Zilles, G.S. Sohi, Execution-based prediction using speculative slices, in: Proceedings of 28th International Symposium on Computer Architecture, 2001, pp. 2–13.