Hybrid Parallelization of an H.264/AVC Video Encoder

A. Rodríguez, A. González , and M.P. Malumbres

Abstract-- **Last generation video encoding standards increase computing demands in order to reach the limits on compression efficiency. This is particularly the case of H.264/AVC specification that is gaining interest in industry. We are interested in applying parallel processing to H.264 encoders in order to fulfil the computation requirements imposed by stressing applications like video on demand, videoconference, live broadcast, etc. Given a delivered video quality and bit rate, the main complexity parameters are image resolution, frame rate and latency. These parameters can still be pushed forward in such a way that special purpose hardware solutions are not available. Parallel processing based on off-the-shelf components is a more flexible general purpose alternative. In this work we propose a hierarchical parallelization of H.264 encoders very well suited to low cost clusters. Our proposal uses MPI message passing parallelization at two levels: GOP and frame and an OpenMP optimization at the lowest parallelization level. In previous work we found that GOP parallelism alone gives good speed-up but imposes very high latency, on the other side frame parallelism gets less efficiency but low latency. Combining both approaches we obtain a compromise between speed-up and latency. Finally, we extend the hierarchical parallelization by including OpenMP optimizations to the most expensive coding functions.**

Keywords—**advanced video coding, parallel and distributed programming, hierarchical parallelization, MPI, OpenMP, performance evaluation**

I. Introduction

THE bandwidth available nowadays in computer networks and also in the Internet allows video delivery applications to reach acceptable performance levels. An important aspect that gives support to video communications is video encoding and this will continue to be the case even with future bandwidth increments. This is because raw video requires a huge amount of data transmitted per second, particularly when high resolution and frame rates are involved.

Video compression is based on removing sensitive redundant information and in the high spatial and temporal correlation. Last generation video encoding techniques, particularly H.264/AVC [1], push the capabilities of these techniques to their limits. The result is a reduction on the bandwidth requirements in several orders of magnitude.

Encoding efficiency has a price that is computation power. H.264/AVC encoders have a very high CPU demand, the most critical case is encoding with latency and real time response requirements. When this is combined with high quality video formats, the only adequate platforms are those with supercomputing capabilities (i.e. clusters, multiprocessors and special purpose devices).

We are interested on cluster platforms because they are becoming a commonly available resource in an increasing number of companies and institutions that require high-performance systems able to cope with large-scale applications (i.e. high-performance web server platforms). Parallel programming on clusters is also very flexible and it allows the design of parallel video encoders adapted to almost any requirement.

Resources available on clusters vary from single to multiple CPU per node, and in every node we can have multimedia extensions in the CPUs and powerful graphic coprocessors. To make efficient use of all these computation resources we can combine different programming approaches:

- *Message passing parallelism.* Message passing runtimes and libraries (i.e. MPI [2]) allow using a cluster to develop distributed versions of a video encoder.
- *Multithread parallelism.* Multithreading (i.e. OpenMP [3]) permits using SMP cluster nodes to reduce response time of the local encoder MPI process.
- *Optimized libraries.* Sequential code can be optimized by using additional resources like SIMD extensions and GPUs to perform complex operations. This optimization approach can be applied by hand or using optimized libraries (i.e. Intel IPP [4], AMD ACML [5], OpenGL [6], etc).

These techniques can be combined hierarchically in such a way that the three levels (message passing, multithreading and optimization) are quite orthogonal. Analysis and implementation of every level is independently done and the individual improvements of each level sum up to improve the overall application performance.

A video sequence is a stream of frames generated at a certain frequency or frame rate. H.264/AVC specification allows the definition of a number of consecutive frames as an independent unit (GOP) to be encoded. H.264 also allows defining slices inside a frame as frame portions that can also be independently encoded. Then message passing level can be decomposed in these two levels.

A. Rodriguez and A.González are with the Technical University of Valencia (UPV). E-mail:abrodleo@doctor.upv.es, agt@disca.upv.es.

M.P. Malumbres is with the Miguel Hernández University (UMH). E-mail: mels@umh.es.

In previous work we have implemented and evaluated GOP based and slice based parallel video encoders [7,8]. In this paper we present the results obtained with an H.264/AVC parallel encoder that combines GOP and slice parallelism using MPI on clusters. The paper is organized as follows: First we estimate the performance that we can expect by means of analytical tools: Little law [9] and PAMELA [10]. Next we discuss the design and main issues related to the implementation. Then we present the performance measurements obtained in two clusters and, finally, some conclusions and future work directions are drawn.

II. Performance analysis

As we show in [8] an efficient parallel video encoder can be implemented dividing the video stream in GOPs. We defined a GOP as 15 consecutive frames following the IBBPBBP… coding pattern. Considering the availability of enough computation resources (cluster nodes) then we can always achieve real time response but with a high latency (GOP encoding sequential time).

On the other hand in [7] we got good performance and latency in a slice based parallel encoder. Unfortunately the scheme gives limited scalability and then real time response is achievable only under limited circumstances.

Both approaches can be combined in order to get the better of every one: scalability and low latency. Processing several GOPs in parallel will contribute to increase throughput. When real time response is achieved, that is throughput is equal to frame rate, additional computational resources can be used to parallelize GOP encoding in order to reduce latency. This is done dividing frames in several slices and processing slices in parallel.

Processes in GOP level parallelization interact slightly with a master process to get GOP ids and at the end of the GOP encoding process to compose the video encoded bit stream. Slice level parallelization has a greater level of interaction among processes because after encoding one frame, we have to decode it in order to keep the DPB (Decoding Picture Buffer) updated with reference frames used in motion estimation.

To describe more precisely the aggregated effect on performance when GOP and slice parallelism are combined we will use Little's law: $N = X^*R$. In order to have a precise definition of the equation's terms, we have to define what a job is. We consider a job the encoding of one GOP. Then the equation terms are:

- N: Number of GOPs processed in parallel.
- R: Elapsed time between a GOP enters the system and the same GOP is completely encoded.
- X: Number of GOPs encoded per second.

If we have n_P nodes in the cluster and every GOP is decomposed in n_S slices, then the number of GOPs processed in parallel will be $N = n_P / n_S$.

If slices are processed in parallel with efficiency E_S and if the sequential encoding time of one GOP is R_{SEQ} then GOP parallel encoding time is:

$$R = R_{SEQ} /(n_S * E_S) \tag{1}$$

Here we suppose that GOP parallelization gets and efficiency close to 1, as experimentally found at [7,8]. Finally the GOP throughput of combined parallel encoder is:

$$X = \frac{\frac{n_P}{n_S}}{\frac{R_{SEQ}}{n_S \cdot E_S}} = \frac{n_P}{R_{SEQ}} \cdot E_S \tag{2}$$

The effect on performance of combining GOP and slice parallelism is to reduce response time (latency) but throughput if affected negatively if the efficiency of slice parallelization is significantly less than 1.

As an example let consider a 1 hour video sequence in HDTV format at 1280x720 and 60 frames/sec. We suppose that a H.264/AVC sequential encoder is able to encode one GOP (15 frames) in 5 seconds. If only one slice per frame is defined in a parallel encoder (no slice parallelism is present) then:

$$X = \frac{n_P}{R_{SEQ}} \tag{3}$$

To get real time response, X has to be equal to 60 frames/sec or 4 GOPs/sec then

$$n_P = 4 \cdot 5 = 20 \text{ nodes} \tag{4}$$

GOP parallelization gives real time response in a 20 nodes cluster but with 5 seconds latency. If the maximum allowed latency in the application is fixed to 1 second, then we can include slice parallelism to comply with this requirement. Let suppose E_S as 0.8 then ¿how many slices do we have to define and how many cluster nodes are required?

$$n_P = \frac{4 \cdot 5}{0.8} = 25 \; nodes \tag{5}$$

$$n_S = \frac{R_{SEQ}}{R \cdot E_S} = \frac{5}{1 * 0.8} = 6.25 \text{ slices} \tag{6}$$

The number of slices and the number of GOPs have to be integers. Then, we set n_S to 7 and N to 4, so the number of required nodes is adjusted to 28. The estimated performance indexes are:

$$X = \frac{28}{5} \cdot 0.8 = 4.48 \text{ GOPs/sec} \tag{7}$$

$$R = \frac{R_{SEQ}}{n_S \cdot E_S} = \frac{5}{7 * 0.8} = 0.89 \text{ sec} \tag{8}$$

In this example, we have obtained that the combined parallel encoder gives real time encoding with latency less than 1 sec on a cluster with 28 nodes.

As shown before, the efficiency of the slice parallelization scheme is very important because it has a direct effect in throughput and latency. We are going to estimate E_S by means of a PAMELA model parameterized with measurements taken on a conventional cluster.

Slice parallelization consists of partitioning every frame in a GOP in a fixed number of slices. Then every slice is encoded in parallel. Before proceeding with the next frame, the actual frame has to be composed and decoded to update de DPB in every node [1]. We implemented this synchronization by means of MPI_Allgather [2].

In our PAMELA model we suppose that MPI_Allgather is implemented efficiently using a binary tree. The number of slices processed in parallel is n_S and the mean slice encoding time is t_S. We call t_W the mean wait time due to variations in t_S and the global synchronization forced by allgather MPI operation. The communication parameters are t_L (start up time) and t_C (transmission time of one encoded slice). Then, the PAMELA model to parallel encode one frame is:

```
L = par (p=1..$n_S$)
      delay($t_S$); delay($t_W$)
    seq (i=0..$\log_2(n_S)-1$)
      par (j=1.. $n_S$)
        delay($t_L$ + $t_C$*$2^i$)
```

The parallel time obtained solving this model is:

$$T(L) = t_S + t_W + t_{AG}$$
$$t_{AG} = \log_2(n_s)\cdot t_L + (n_S - 1)\cdot t_C \tag{9}$$

So, efficiency can be computed as:

$$E_S = \frac{\frac{T_{SEQ}}{T}}{n_S} = \frac{\frac{n_S \cdot t_S}{t_S + t_W + t_{AG}}}{n_S} = \frac{1}{1+\frac{t_W + t_{AG}}{t_S}} \tag{10}$$

We have obtained experimental estimations of t_S and t_W using a sequential H.264 encoder on the Foreman CIF video sequence. Communications parameters t_L and t_C have been measured running IMB ping pong benchmark [11] on a cluster with four dual Opteron nodes interconnected by a Gigabit Ethernet switch. The measurements correspond to message sizes around the size of one encoded slice. With 4 slices we got a mean encoded slice size of 4056 bytes. The parameter values obtained appear on table I.

TABLE I

MEASURED PARAMETERS VALUE (TIME VALUES ARE SHOWN IN MICROSECONDS)

t_L	t_C	t_S	t_W	t_{AG}
60	0.0133*4056	840000	20586	421

Estimated efficiency for a slice based parallel encoder running in a 4 nodes cluster is:

$$E_S = \frac{1}{1+\frac{20586+421}{840000}} = 0.976 \tag{11}$$

A better estimation is obtained by running IMB allgather benchmark [11] on the cluster configuration we are going to wok on. The allgather time obtained experimentally on the cluster with 4056 bytes messages is 408 microseconds. Then, the estimated efficiency is:

$$E'_S = \frac{1}{1+\frac{20586+408}{840000}} = 0.976 \tag{12}$$

We notice that efficiency is not limited by allgather communication overhead, but it is limited by synchronization wait due to differences among slice encoding time. The percentage of wait time related to encoding time increases when the slice size decreases. This penalizes efficiency when we increase the number of slices. Another negative effect of increasing the number of slices, and then decreasing their size, is a reduction on encoding performance. Both factors limit the number of slices we can define and then the amount of available parallelism.

III. SCALABILITY OF SLICE PARALLELISM

H264/AVC [12] encoders use a 16x16 pixel macroblock (MB) as data encoding unit. A slice is composed by a specific number of MBs. So, we can define several slices in a frame. Every slice is encoded and transmitted independently, limiting error propagation. In our context slices define slightly coupled tasks that can be performed in parallel.

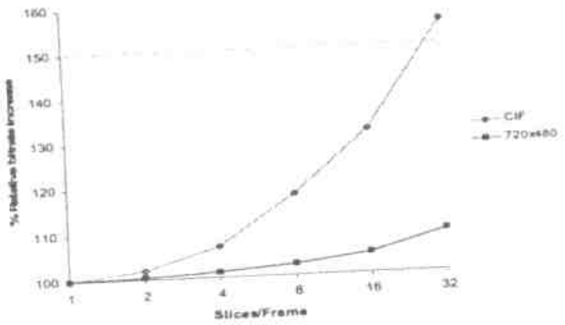

Fig. 1. Bitrate overhead vs. number of slices/frame.

However, defining several slices in a frame reduces encoding efficiency. The most adverse effect is a significant bit rate increment that is inversely proportional to the number of MBs per slice. That means that the feasible number of slices will depend on the video resolution, as shown at figure 1.

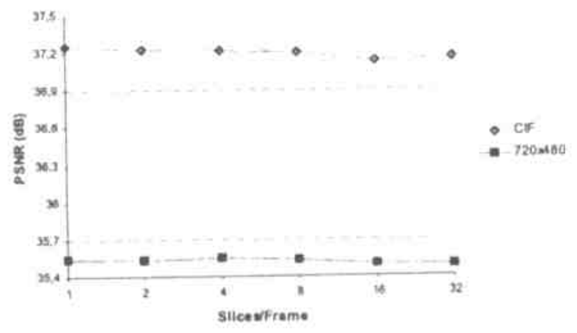

Fig. 2. PSNR loss vs number of slices per frame.

We can see also that in CIF resolution with 8 slices we have a bit rate increment close to 20% which is clearly inadmissible. When we apply slice parallelism we get good speedup with 8 processors then scalability is not limited by the parallel algorithm but by the increase on bit rate. Other encoder quality parameters like PSNR do not behave so adversely. We can se in figure 2 that PSNR has as small variation around 35.5 dB in CIF resolution and around 37 dB in 720x480.

Finally we have noticed that encoding time decreases when the number of slice increases. In figure 3 we can see a significant encoding time reduction in both CIF (33%) and 720x480 (27%) video formats.

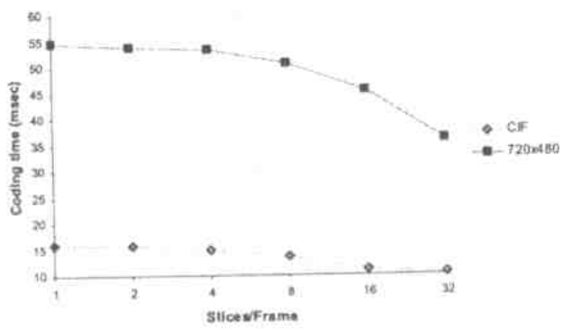

Fig. 3. Encoding time vs number of slices per frame.

We conclude that bitrate is the encoding efficiency parameter that limits the number of slices we can define. Particularly, a number between 4 and 12 slices seems to be adequate depending on video resolution. Parallel encoders based on slice parallelism give good speedups up to 8 slices as we will show in experimental results section.

IV. Hierarchical H.264 parallel encoder

As we mentioned in the introduction we are going to combine two levels of parallelism in order to achieve scalability and low latency. In the first level we divide the input video sequence in blocks of 15 consecutive frames (GOPs) following the popular IBBPBBP... coding pattern. Every GOP is assigned to a processor group inside the cluster. Every group encodes its GOP independently of other groups. When one GOP is completely encoded, the processor group is ready to encode the next video GOP.

Every processor group has a local manager (P0) that communicates with the global manager (P0'). The local manager asks for a new GOP to be encoded by its group when the current one is completely encoded. The global manager informs about the GOP assignment by sending a message with the assigned GOP number to the requesting local manager. The on demand GOP assignment method is quite simple and gives a good load balance behaviour.

Inside a processor group, the assigned GOP is processed decomposing its frames in slices, in such a way that every processor in the group processes a slice. Once a processor has the next GOP number to encode, it can read and encode its corresponding slice on the frames belonging to that GOP.

Slices are defined getting MBs in scan order in such a way that the number of MBs per slice is as much balanced as possible. When all the slices belonging to a frame are encoded they have to be integrated to build the encoded frame. Next, all the encoded frames corresponding to a GOP are put together to form the output bit stream.

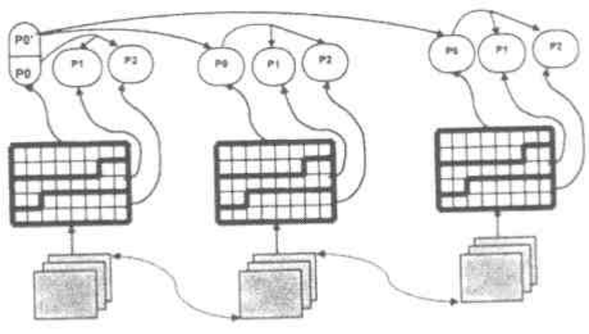

Fig. 4. Hierarchical H.264 parallel encoder.

V. Multithreading with OpenMP

Clusters based on multiprocessor nodes are becoming very common. Nowadays, the use of multicore processors permits to have two processor SMP nodes at a very low price. In the near future the number of cores will increase to 4 and 8 opening the field for applying effective multithreading parallelization techniques on clusters in combination with message passing.

OpenMP is a multithread programming standard that is designed to easily include thread level parallelism in existing sequential code. The parallelization technique consists of identifying the execution time bottlenecks in sequential code by means of profile analysis and then to parallelize the most time consuming loops using OpenMP pragmas.

In our case we profiled H.264 sequential encoder over the Foreman CIF video sequence. To generate the profiling data we use –pg compiler option and gprof tool. We got the results shown in table II over a total encoding time of 1095 seconds:

PROFILE RESULTS FROM FOREMAN CIF VIDEO SEQUENCE

Function	T (%)	CumT	Time	#Calls	T/Call
SetupFastFullPelSearch	46.6	510.65	510.65	14652	3.49E-02
FastFullPelBlockMotionSearch	6.23	578.95	68.30	600732	1.14E-04
SetupLargerBlocks	3.83	620.90	41.95	14652	2.86E-03
SubPelBlockMotionSearch	3.76	662.15	41.25	600732	6.87E-05
UnifiedOneForthPix	1.13	674.49	12.34	6	2.06
ModeDecision_4x4IntraBlocks	1.11	686.64	12.15	198144	6.13E-05
dct_chroma	1.09	698.61	11.97	116532	1.03E-04
...	...	...	...	...	...

The speedup achievable by OpenMP can be estimated by means of the next equation:

T

speedup can exploit otherwise wasted processor hyperthreading capabilities.

VI. Experimental results

In order to evaluate the proposed H.264 hierarchical parallel encoder we choose two different clusters of workstations named Mozart and Aldebaran. Mozart has 4 biprocessor nodes with AMD Opteron 246 at 2 GHz interconnected by a switched Gigabit Ethernet. On the software side it runs Linux SuSE 9.1, Intel C compiler 8.1 and MPI 2.0. Aldebaran is an SGI Altix 3700 with 44 nodes Itanium II interconnected by a high performance proprietary network giving a NUMA architecture. It runs Linux RedHat 9.0 with GNU tools and MPICH. Performance measurements are obtained

TABLE II

PROFILE RESULTS FROM FOREMAN CIF VIDEO SEQUENCE

Function	T (%)	CumT	Time	#Calls	T/Call
SetupFastFullPelSearch	46.6	510.65	510.65	14652	3.49E-02
FastFullPelBlockMotionSearch	6.23	578.95	68.30	600732	1.14E-04
SetupLargerBlocks	3.83	620.90	41.95	14652	2.86E-03
SubPelBlockMotionSearch	3.76	662.15	41.25	600732	6.87E-05
UnifiedOneForthPix	1.13	674.49	12.34	6	2.06
ModeDecision_4x4IntraBlocks	1.11	686.64	12.15	198144	6.13E-05
dct_chroma	1.09	698.61	11.97	116532	1.03E-04
...	...	...	...	...	...

The speedup achievable by OpenMP can be estimated by means of the next equation:

$$Sp = \frac{T_{Sec}}{T_{Sec} + T_{Ov} - T_{Par} + \dfrac{T_{Par}}{N}} \qquad (13)$$

Where T_{Sec} is the sequential encoding time, T_{Par} is the time contribution of parallelized functions and T_{Ov} is the parallelization overhead. Considering two processors nodes the condition to pay off the parallelization overhead is $T_{Ov} < T_{Par} / 2$. We have estimated T_{Ov} in 50 microseconds then only functions with execution time per call clearly above 100 microseconds are candidates to OpenMP parallelization. The candidates with more significant contribution to total encoding time will be considered in first place.

From the profiling results, it is clear that the first candidate is function ***SetupFastFullPelSearch*** which is related with the setup of motion estimation process, being in charge of performing the initial MV prediction, and other time-consuming procedures. This function is responsible of around 46% of total encoding time. Its time per call is 34900 microseconds that clearly pays off the parallelization overhead. To implement the OpenMP parallelization there are some conditions to be satisfied: few and controlled data dependencies and no inline functions. The first function candidate is a long one with data dependencies in 4 data structures. A parallel for was implemented controlling the former mentioned dependencies and suppressing internal inline functions.

The second function on the contribution to sequential encoding time is ***FastFullPelBlockMotionSearch*** which is the main function that performs the motion estimation process for a particular MB, using the greedy Full Search method. This function has an execution time per call too much close to T_{Ov} then it was dismissed.

The next candidate that complies with the selection criteria is function ***SetupLargerBlocks*** which is related with SAD computations for larger blocks than the basic 4x4 block size (it is composed of a lot of sequential loops). The contribution of this function to global execution time is only 4%. In this case parallelization was implemented using OpenMP sections.

This first approach to OpenMP parallelization was integrated with the hierarchical MPI implementation. Only one MPI process was launched in every biprocessor node and two OpenMP threads were created in every parallel section. If multithreading contribution to speedup is low, then this processor assignment is clearly not adequate. However, the small gains in speedup can exploit otherwise wasted processor hyperthreading capabilities.

VI. EXPERIMENTAL RESULTS

In order to evaluate the proposed H.264 hierarchical parallel encoder we choose two different clusters of workstations named Mozart and Aldebaran. Mozart has 4 biprocessor nodes with AMD Opteron 246 at 2 GHz interconnected by a switched Gigabit Ethernet. On the software side it runs Linux SuSE 9.1, Intel C compiler 8.1 and MPI 2.0. Aldebaran is an SGI Altix 3700 with 44 nodes Itanium II interconnected by a high performance proprietary network giving a NUMA architecture. It runs Linux RedHat 9.0 with GNU tools and MPICH. Performance measurements are obtained encoding the 720x480 standard sequence Ayersroc composed by 16 GOPs. The combination of number of active GOPS (or processor groups) and number of slices per frame are described in table III.

TABLE III

WORKING MODES AT BOTH CLUSTERS.

Configuration	*Cluster*	*#Groups*	*#Slices*
01Gr_08Sl	Mozart	1	8
02Gr_04Sl	Mozart	2	4
04Gr_02Sl	Mozart	4	2
08Gr_01Sl	Mozart	1	8
01Gr_16Sl	Aldebaran	1	16
02Gr_08Sl	Aldebaran	2	8
04Gr_04Sl	Aldebaran	4	4
08Gr_02Sl	Aldebaran	8	2
16Gr_01Sl	Aldebaran	16	1

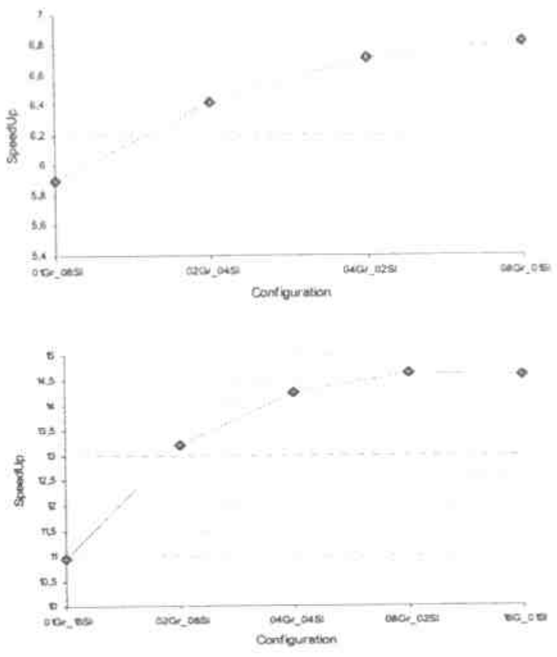

Fig. 5. Speedup in Mozart (top) and Aldebaran (bottom).

In figure 5 we can see that increasing the number of slices per frame has a significant adverse effect on speedup that reduces GOP throughput. This effect is the same in both clusters.

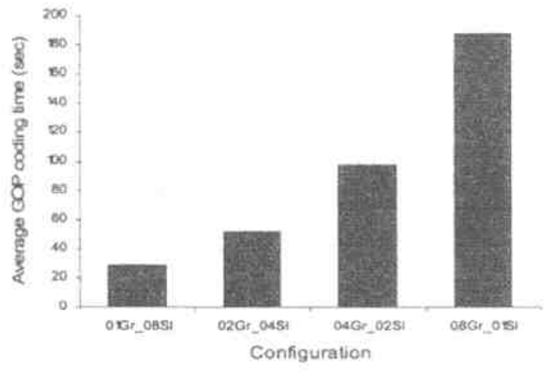

Fig. 6. Mean GOP encoding time

However, we show in figure 6 that the mean GOP encoding time is effectively (linearly) reduced when the number of slices per frame increases. The speedup loss is bigger than the one we have expected. After performing some additional experiments, we discovered that the speedup loss is mainly due to the synchronization wait time associated to MPI-allgather operation. This operation is performed at the end of each frame (as mentioned before). Also, as the number of slices per frame increase, the synchronization wait time becomes larger, reducing the overall application performance (as shown in figure 5). So, we have to further analyze their causes, in order to improve the overall performance of slice parallelism.

Performance obtained by the hybrid MPI-OpenMP approach is shown in table IV.

TABLE IV

MPI-OPENMP IMPLEMENTATION RESULTS

	MB x Slice	T.Sec. 120 frm	T.Par. 120 frm	Avg. PSNR	Bits	Tpro x Gop	SpUp
4Gr-1Sl	1350	1962.04	443.13	39.26	5033880	216.6	4.43
2Gr-2Sl	675	1917.12	448.32	39.24	5082008	108.98	4.28
1Gr-4Sl	337	1878.96	460.06	39.22	5177752	53.81	4.08

As it can be seen, the use of OpenMP multithreading capabilities slightly increase the overall speedup, taking into account that only four MPI processes were launched with different number of processor's groups (Grs) and number of slices per groups (Sl). The reducing speedup performance (last column of table III) as the number of slices per group increases is due to the effect of slice synchronization as explained before. Although these are the initial OpenMP optimization steps, more work should be done to fully exploit the benefits of multithreading in those nodes with multithreading hardware support.

VII. CONCLUSIONS AND FUTURE WORK

A hierarchical parallel video encoder based on H.264/AVC specification was proposed. After performing some analysis about the convenience of a hierarchical parallel approach and developing a GOP/slice parallel version, experimental results confirm the results from previous analysis, showing the ability of getting a scalable and low latency H.264 encoder. This is performed adjusting the cluster configuration by setting up the number of processor groups (or parallel encoded GOPs) and the number of processor in a group (or number of slices in a frame). Depending on the application requirements we can reach an adequate balance between throughput and latency.

However, some issues remain open, as mentioned in previous section. So, we are analyzing the causes that lead to speedup loss when increasing the number of slices per frame. If we find a solution, the number of slices per frame will not be such limiting factor to obtain acceptable speedups.

We also have included a first approach to multithread parallelism using OpenMP. Our experience shows that it is difficult to develop OpenMP optimizations in the H.264 reference coding software, since it has not been designed having in mind multithread optimizations. Particularly data dependencies are difficult to identify and to resolve.

As future work, we plan to push on OpenMP parallelization and SIMD optimizations in order to make profit of the parallelism available in CPU resources.

VIII. ACKNOWLEDGMENTS

This work was funded by the Spanish Ministry of Education and Science under grant TIC2003-00339.

IX. REFERENCES

[1] ISO/IEC 14496-10:2003, "*Coding of Audiovisual Objects—Part 10: Advanced Video Coding,*" 2003, also ITU-T Recommendation H.264 "Advanced video coding for generic audiovisual services."

[2] Pacheco, P.S.: *Parallel Programming with MPI*, Morgan Kaufman Publishers, Inc

[3] R. Chandra et al., "*Parallel Programming in OpenMP*", Morgan kaufmann, 2000.

[4] *Intel Integrated Performance Primitives*, http://www.intel.com/cd/software/products/asmo-na/eng/perflib/ipp/index.htm

[5] *AMD Core Math Library (ACML)*, http://developer.amd.com/acml.aspx

[6] OpenGL Architecture Review Board et al., "*OpenGL(R) Reference Manual* ", 5th edition, Ed. Dave Shreiner, 2005.

[7] J.C. Fernández and M. P. Malumbres, "*A Parallel implementation of H.26L video encoder*", in proc. of EuroPar 2002 conf. (LNCS 2400), pp. 830, 833, Padderborn, 2002.

[8] A. Rodriguez, A. González and M.P. Malumbres, "*Performance evaluation of parallel MPEG-4 video coding algorithms on clusters of workstations*", IEEE Int. Conference on Parallel Computing in Electrical Engineering, pp. 354, 357, Dresden, 2004

[9] E.D. Lazowska, J. Zahorjan,G.S. Gaham, K.C. Sevcik, *Quantitative System Performance*, Prentice-Hall, 1984

[10] Arjan J.C. van Gemund, "*Symbolic Performance Modeling of Parallel Systems*", IEEE Transactions on Parallel and Distributed Systems, vol 14, no 2, February 2003.

[11] Intel MPI Benchmarks: *Users Guide and Methodology Description*, Intel GmbH, Germany, 2004.

[12] *H.264/AVC Reference Software*: http://iphome.hhi.de/suehring/tml/.