

Parallelizing MPEG Video Encoding using Multiprocessors *

DENILSON M. BARBOSA¹

JOÃO PAULO KITAJIMA²

WAGNER MEIRA JR.¹

¹Departamento de Ciência da Computação
Universidade Federal de Minas Gerais
Caixa Postal 702, 30123-970
Belo Horizonte, MG, Brasil
{dmb,meira}@dcc.ufmg.br

²Laboratório de Bioinformática
Universidade Estadual de Campinas
Caixa Postal 6176, 13083-970
Campinas, SP, Brasil
jpk@lbi.dcc.unicamp.br

Abstract. Many computer applications are currently using digital video. Recent advances in digital imaging and faster networking infrastructure made this technology very popular, not only for entertainment, but also in health and education applications. However, high quality video requires more storage space and communication bandwidth than traditional data. To deal with this problem, most of the digital video encoding techniques use a compression scheme. The MPEG committee has defined widely used standards for digital video encoding providing high quality images and high compression rates. However, real-time MPEG encoding also demands high computational power, usually far beyond traditional sequential computers can provide. Fortunately, the algorithms compliant to the MPEG standard can be parallelized. In this work, we propose a novel and simple shared-memory parallel algorithm for MPEG-1 video encoding based on the parallel execution of different coarse grained tasks (read pictures, write coded frames, I, P, and B frames coding). Two synchronization strategies are implemented and experimental results show that encoding rates of 43 frames per second are achieved, which is almost 50% better than the real-time encoding standard rate (30 frames per second).

1 Introduction

Many applications are requiring nowadays high quality video encoding and transmission. HDTV (High Definition TV), Video on Demand services and telepresence applications, such as remote surgeries, are some important examples of these applications. Although the underlying telecommunication infrastructure allows real-time transmission of streaming media, real-time encoding of high quality digital video demand large amounts of computing power, usually far beyond traditional hardware can provide. Despite its increasing popularity, real-time encoding of digital video remains a challenge.

Digital video requires more storage space and communication bandwidth than traditional data. To deal with this problem, most video encoding standards define compressing algorithms to reduce these requirements. The MPEG (Moving Pictures Experts Group) committee is an international task force devoted to develop standards for digital video encoding. At the present moment, there are two MPEG standards concerning high quality video encoding: (1) MPEG-1 and (2) MPEG-2. The later is an evolution of the former: it handles more video formats, such as HDTV, and deals with transmission issues. However, the encoding algorithms are very similar in

both standards. There are two other MPEG standards: (1) MPEG-4, which deals with digital video in low communication capability environments; and (2) MPEG-7, which is currently under development and is devoted to storage and retrieval of descriptive information regarding the sequence as long as allowing search by contents in video sequences.

In this work, we are specifically interested in the MPEG-1 Video Standard [5] due to its relative simplicity. Moreover, it would be easy to extend a MPEG-1 encoder to take into account the MPEG-2 standard. Hereinafter we use the term MPEG to refer to the MPEG-1 standard.

The frames in a MPEG sequence are encoded inside Groups of Pictures (GOPs). Each frame has a decoding timestamp relative to the beginning of its corresponding GOP. The standard defines 4 models for frame encoding: I, P, B, and D. D-frames cannot be mixed with frames of other types and are only used in special purpose sequences. I frames are encoded independently of other frames in the sequence, while P and B frames are encoded as differences from one or two previous (or subsequent) *reference frames*, as depicted by Figure 1. These differences are obtained via motion estimation and compensation techniques [4].

The frame encoding dependencies turn into data dependencies for the encoder. Besides, they can hamper

*This Work was partially supported by CAPES and PROTEM-CC Almadem Project (CNPq grant 680072/95-0). João Paulo Kitajima has a FAPESP fellowship since May 1st, 1999 no. 99/01389-0.

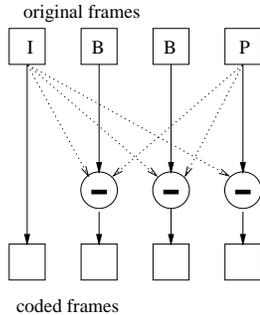


Figure 1: Frame encoding dependencies in a MPEG sequence.

the simultaneous encoding of the frames thus diminishing the desired maximum speedup when going to parallel [2].

There are some previous approaches of MPEG parallel encoding. Most of them employ either *ad hoc* hardware or data-parallelism software strategies. There are two major drawbacks in these approaches: (1) *ad hoc* hardware becomes obsolete and (2) the communication overhead is high in data-parallel algorithms [3].

In this paper we present a simple and novel approach to parallelize the MPEG encoding using shared-memory multiprocessors. This paper is organized as follows. Section 2 describes related work and Section 3 presents our approach for parallel MPEG encoding. Implementation issues are discussed in Section 4 while our experimental results are summarized in Section 5. Finally, some conclusions and future works are outlined in Section 6.

2 Related Work

Previous approaches to parallelize MPEG encoders use either *ad-hoc* hardware or message passing architectures. The major problem of using special hardware, as described in [12], is that the hardware becomes obsolete. This happens due to improvements in processing and memory components or in video encoding technology and standards¹ that cannot be incorporated in existent *ad hoc* hardware.

Regarding general purpose parallel machines, [1] presents a data-parallel algorithm for real-time MPEG-2 encoding. It implements an encoder for the Intel Paragon distributed memory parallel machine and for a network of workstations. The strategy described consists in dividing each frame and distributing the pieces among the processors. However, there are some portions of the frames that have to be sent to more than one processor

¹They also become obsolete.

due to block matching motion estimation techniques [7, 3]. Moreover, the frames cannot be arbitrarily divided, thus defining an upper bound on the number of processors that can be used [1], depending on the size of each frame in the sequence being encoded.

There is yet another problem with this approach. The amount of data exchanged can be very high. Furthermore, adding more processors makes the problem worse: the proposed solution does not scale [3]. The authors suggest an alternative approach where each processor gathers all data it needs before start encoding. This avoids communication during the encoding phase but does not reduce the amount of exchanged data.

Another strategy is to send whole frames for each processor, as described in [10]. The problem here is that the amount of data needed by a processor depends on the type of the frame being encoded. A processor encoding a B-frame will need more data than a processor encoding an I-frame. Clearly, the communication workload will not be homogeneous among the processors.

There are not many approaches using shared memory parallelism. In [11] the parallelization of the motion estimation process for a MPEG-4 encoder is described. In this case, the overall speedup is limited to the portion of time spent in this phase of the encoding. Furthermore, there are heuristic computations that produce good visual results and require much less computing power [13]. In our work, we use one such heuristic [4, 7] that, alone, improves the sequential encoding time by a factor of 74% [2].

3 Proposed Algorithm

We propose here a new algorithm [2] based on shared-memory parallelism to overcome communication problems of data-parallel approaches. This is achieved by (1) using a shared memory parallel machine, where communication is performed through primary memory, and (2) sharing the uncompressed original frames among the tasks. With this approach, the reference frames can be accessed directly from memory, avoiding communication among processors during encoding.

We define two kinds of task: (1) GOP encoding task and (2) picture encoding task, responsible respectively for GOP and frame encoding. The smaller task performed in our approach is the encoding of a single frame, regardless of its type. There are also the main task, which coordinates the GOP tasks, and the write task that flushes the encoded frames stored in memory to the final compressed file. The algorithm defines two equally sized buffers: (1) the read buffer, which holds uncompressed frames, and (2) the write buffer, where the coded frames are stored prior to be written in the compressed

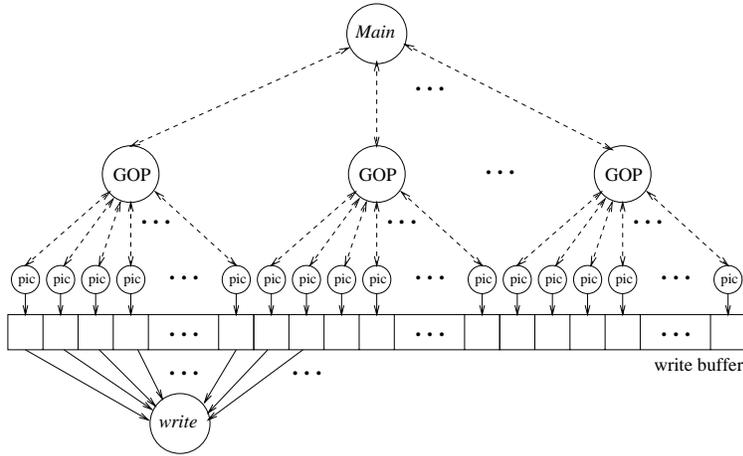


Figure 2: Distribution of tasks in the proposed parallel algorithm.

output file.

Figure 2 shows the structure of our algorithm. In the figure, circles represent tasks, dashed lines represent synchronization dependencies and solid lines represent data flows. The figure shows only the write buffer.

3.1 Synchronization Steps

There are two synchronization steps in our algorithm: (1) encoding synchronization and (2) writing synchronization. The first guarantees the correct encoding of the frames, provided the dependencies imposed by the MPEG standard. This is done by coordinating the GOP tasks with their corresponding picture tasks. Each GOP task encodes the appropriate GOP heading and creates as many picture tasks as there are frames inside the GOP. Those tasks are performed in parallel and, when they are all finished, the GOP task is considered completed. The main task coordinates the GOP tasks in a similar way.

The writing synchronization process prevents occupied write buffers from being incorrectly overwritten by other tasks. There are flags to indicate the state of the write buffers. When a frame is coded, the corresponding flag is set and no other frame can be encoded on that buffer. The write task, after dumping that buffer to disk, clears its flag. Another compressed frame can then be written on. The task of dumping frames into the output file must be serialized because the ordering of the frames is unique. Therefore, this synchronization step can hinder the full exploitation of the parallelism of our algorithm.

4 Implementation

The tasks defined in a parallel algorithm are mapped to code fragments in the real parallel program. This mapping process is largely dependent of the parallel architecture in use. Multithreaded programming environments are well suited for shared memory algorithms because the tasks are easily mapped onto threads of execution. These threads, by their nature, share every resource available for the program, including the main memory. We implemented our algorithm using the C language and the multithread programming library Pthreads [6].

In a multithreaded environment, the allocation of tasks to processors can be done automatically by the parallel machine's operating system. Thus, the programmer focus only in the synchronization and communication of tasks and the mapping of these tasks to threads. In this work we create different kinds of threads corresponding to the different kinds of tasks. However, a single thread can perform more than one task in different moments (e.g., a given picture encoding thread can encode more than one frame).

Tasks in a shared memory environment use shared portions of memory to communicate. In this work we define two structures, which we call task tags, for GOP tasks and picture tasks. A GOP tag holds information regarding the number and the position of the frames inside the GOP. These tags are created by the main task and processed by GOP tasks (i.e., written by the main thread and read by the GOP encoding threads). The picture tag holds informations on the read and write buffers to be used, the type and the location of the frame to be coded and, when necessary, the number of the buffers where the reference frames are stored. These tags are created by GOP tasks and used by picture tasks.

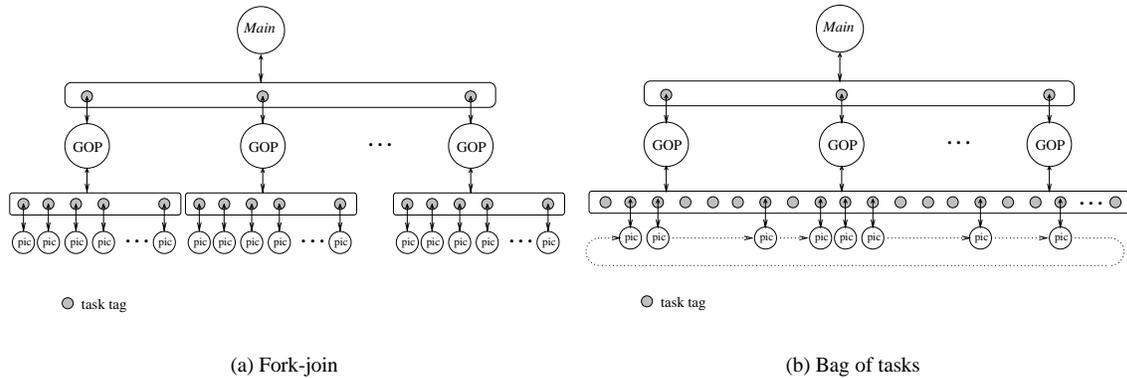


Figure 3: Synchronization strategies. The circles represent threads.

Moreover, every tag contains semaphores to signal the start and the completion of the corresponding task. Thus, both synchronization and communication among tasks are done using the tags. We use semaphores to avoid race conditions when accessing tags.

In this work, the main task (performed by the main program – which is also a thread) creates a number of GOP tasks and coordinates the execution of them by signaling the GOP encoding threads via the semaphores in the corresponding tags. The GOP encoding threads, by their turn, create picture tasks and fill the corresponding tags. Each GOP encoding thread encodes the GOP header, starts the picture encoding threads (signaling the tags) and blocks until all frames are coded. When the task is completed, the GOP encoding thread signals the main thread which creates a new GOP task to be performed.

Concerning the picture tasks synchronization, we implemented two different strategies: (1) fork-join and (2) asynchronous execution of threads, also known as a “bag of tasks”.

4.1 Fork-Join Synchronization

In this strategy, each picture encoding thread is attached to a unique GOP encoding thread. Moreover, there is a fixed number of picture threads to each GOP thread. Therefore, a given picture encoding thread will always encode frames of the same type. Each GOP task holds a private vector of tags as shown by Figure 3(a).

The major problem with this approach is the synchronization overhead imposed. After finishing a task, a given picture thread cannot encode another frame until every other thread inside the same GOP is finished. If this synchronization is not respected, the buffers assigned to the GOP task can be replaced by new frames

while still in use by other tasks.

4.2 Asynchronous Encoding

There is only one picture tags vector, shared by all GOP and picture threads, in this strategy. Each GOP task writes in a portion of this vector, creating the corresponding tasks. All picture threads read this vector in a circular circuit (Figure 3(b)) encoding the first available task found. Thus, the picture threads are no longer attached to particular GOP threads and eventually encode different types of frames.

With this approach, the number of picture threads no longer depends on the number of GOP tasks. The number of buffers, however, is still a function of the number of GOP tasks and the number of frames per GOP.

5 Experimental Results

Both versions of the algorithm were implemented and tested on a Sun Enterprise 10000 belonging to CENA-PAD²-MG/CO. This machine has 32 processors and 8 GB of primary memory. Unfortunately, exclusive access to this machine was not possible, although its load was stable during the experiments. The average workload of the machine during our experiments was of 15 CPUs fully busy. There were idle processors, but the processor-memory bus was shared, as well as the disk subsystem. The standard deviation of the measures were low, reinforcing the load stability of the multiprocessor.

5.1 The Parallel Encoder

The first step towards the intended parallelization was to isolate the encoding routines for I, P, and B frames.

²Centro Nacional de Processamento de Alto Desempenho.

sequence	frames	screen size
football	146	320 × 240
susie	148	320 × 240
sun	4653 ^a	352 × 240
leroy	2054	352 × 240

Table 1: Test sequences.

^aIn the first experiment only 1000 frames are used.

Our first approach was to use available public domain encoders [8, 9]. However, these encoders did not suit our intended modularization. We then implemented a simple MPEG-1 encoder, based on the MPEG syntax described in [7]. Some optional features of the MPEG standard were not implemented. Although they are very important, we left them as future work because our current major concern is not video quality but evaluate both speedup and scalability of our algorithm.

In this work, we fixed the number and the distribution of the frames for the encoder. We used 12 frames per GOP, according to the following sequence: I-B-B-B-P-B-B-B-P-B-B-P. This generated a uniform workload in every experiment of 12 picture tasks for each GOP task.

5.2 Video Sequences Benchmark

The computational effort needed to encode a video sequence depends largely on the sequence itself. Besides screen size and number of frames, one of the most important factors is the complexity of the scenes [4]. Thus, in order to better determine the performance of our algorithm, we chose four different sequences to run the experiments. Table 1 shows some features of these sequences.

From these four sequences, two are considered simple: *susie* and *leroy*. The *susie* sequence shows a girl answering the phone and the *leroy* video shows an interview with a musician. The other two are considered more complex. *Football* shows a scene of an American football game while *sun* is a promotional sequence produced by Sun Microsystems. The *susie* and *football* sequences were used as test sequences by the MPEG committee. We decided to use also longer sequences to verify the impact of the size of a sequence over the encoder’s performance.

5.3 The Experimental Environment

Two experiments were devised in order to characterize the performance of our encoder according to the synchronization technique used. The first experiment compares the speedup and the efficiency of each synchronization strategy. By efficiency we mean the portion of

time the encoder spends doing useful work. More specifically, the time spent in file reading and synchronization steps are not considered useful. The second experiment is intended to verify the real-time encoding capability of our encoder. The benchmark in this experiment is the 30 frames per second threshold, accepted as the minimum for real-time encoding.

In the first experiment we used 1, 2, 3, and 4 simultaneous GOP tasks for the fork-join encoder (corresponding to 12, 24, 36 and, 48 picture tasks and threads). The bag of tasks version was tested with 1, 2, 4, 8, 16, and 32 independent picture encoding threads. In this configuration, we used 4 simultaneous GOP tasks or, in other words, 48 picture tasks.

In the second experiment, we used the configuration that exhibited the best performance: using bag of tasks synchronization with 16 threads. Different portions of the same sequence were encoded.

We performed 20 measures for each point. In order to reduce the impact of the machine load, the measures were compared to the average of these 20 points. The 5 points presenting the largest deviation were discarded.

5.4 Performance Evaluation

The overall speedup of both versions is depicted by Figure 4. The fork-join version is limited to a speedup close to 5, while the bag of tasks version presents almost linear speedups when up to 8 threads are used. Since we do not change the MPEG encoding algorithms, the above ideal speedup in the figure is due to the write and GOP tasks which are also performed in parallel.

Figure 5 presents the average encoding efficiency curves for both versions of the encoder. The figure shows that I/O and wait times due to synchronization operations are the major factors of performance degradation of the fork-join version. The graph for the bag of tasks version shows that these overheads have less impact over the encoder’s performance until 32 threads are used. The first advantage of the bag of tasks strategy can be seen by the comparison of these two figures. Since the fork-join version uses more picture threads, more requests to the I/O subsystem are performed. Therefore, it is more likely to overload the machine by using this approach.

However, Figure 5 shows that wait times due to synchronization steps are the major limiting factor to the performance of the fork-join version. Picture threads in this version of the algorithm spend more than 50% of their time in these operations. Again, the bag of tasks version performs better, as detailed by Figure 6. There are two main reasons for this fact: (1) the number of threads used and (2) the flexibility of the scheme employed. Since synchronization is done by using shared

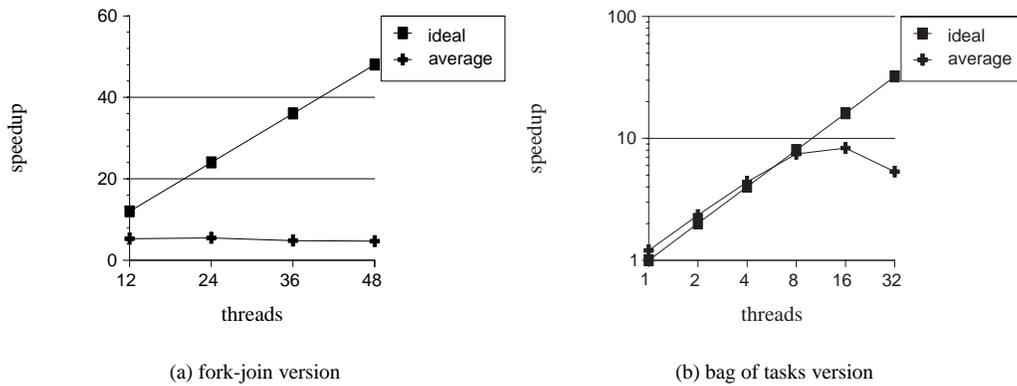


Figure 4: Overall speedup.

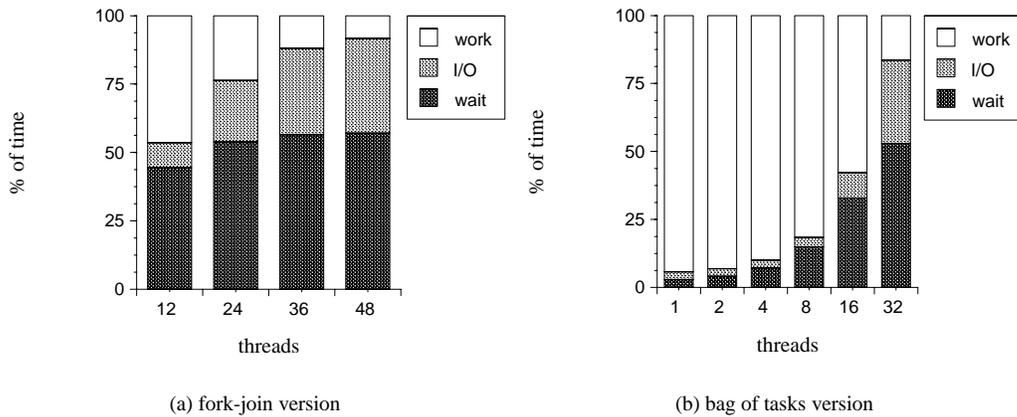


Figure 5: Average efficiency.

semaphores, more threads represent more synchronization operations and greater synchronization times.

The fork operation consists in the division of the GOP task in a number of picture tasks. The encoding process is then blocked until all picture tasks are done (the join operation). With this strategy, when a thread finishes encoding a frame, it remains blocked until all other threads (attached to the same GOP) also finish their work. Since encoding times are different for different frames, the longest encoding time will determine the encoding time for the entire GOP. There is another similar synchronization step concerning the main thread and the GOP threads which increases the synchronization overhead of this strategy.

The bag of tasks strategy is a much more flexible approach. Although the synchronization for the GOP threads remains the same, the picture threads search for

new tasks independently of GOP threads signals. Also, the picture threads do not wait for other picture threads. However, by increasing the number of threads in the encoder, we diminish the relative number of tasks per thread. The higher synchronization times in Figure 6 for 16 and 32 threads are a consequence of this fact. At the cost of adding more simultaneous GOP tasks, thus increasing the number of picture tasks and buffers used, the efficiency can be further improved. Since the largest buffer used in our experiments occupied 115 kilobytes of data, adding more buffers is not a concern.

Regardless of the thread synchronization strategy used, two facts were observed: (1) the write synchronization overhead was insignificant and (2) the algorithm performs slightly better with longer sequences. Also, as expected, the complex sequences (football and sun) demanded more average encoding time per frame than the

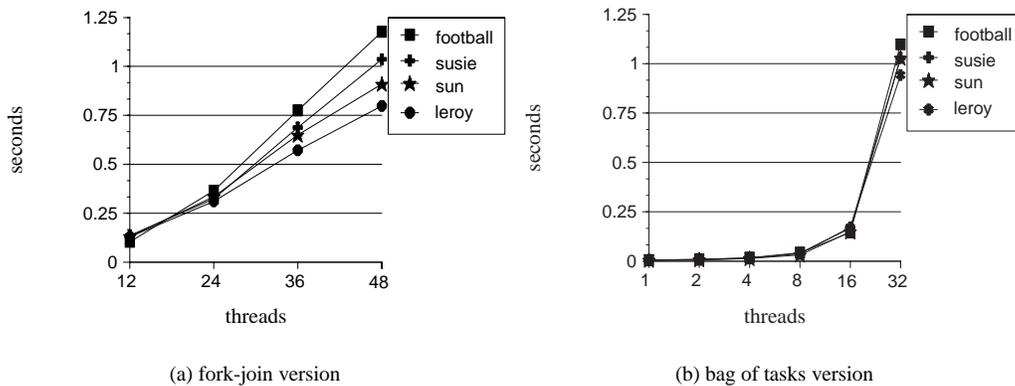


Figure 6: Synchronization times.

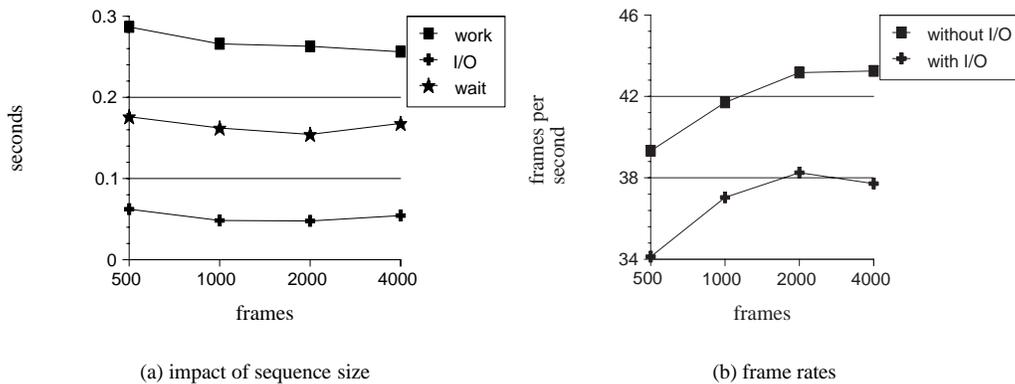


Figure 7: Real-time encoding.

simple ones.

5.5 Real-time encoding

The second experiment, concerning real-time encoding, has two goals: (1) verify the impact of sequence size over the encoder’s performance and (2) verify the capability of real-time encoding (i.e., more than 30 frames per second). The first goal is hard to achieve because the complexity of different portions of a sequence may vary. In this experiment, we used 500, 1000, 2000, and 4000 frames of the sun sequence, because its scene complexity is approximately the same across the sequence. This was also verified by our experiment.

As mentioned, the encoder used in this experiment has 16 picture threads and uses the bag of tasks strategy. The results are summarized by Figure 7. The graph in Figure 7(a) shows that the encoder performance seems not to be affected by the size of the sequence. As men-

tioned however, the encoder performed slightly better with longer sequences. This is also confirmed by the slowly decreasing average frame encoding times in the figure.

In Figure 7(b) we show the frame encoding rates achieved, either with and without considering I/O times for frame reading. We overlooked I/O times considering that an ideal real-time encoder would grab the uncompressed digital frames from a digitizer attached to a camera or a VCR, would store them in a temporary buffer and would encode them directly from the main memory.

The results show that our encoder was able to encode 43 frames in a second, which is almost 50% better than the 30 frames per second threshold, widely accepted as adequate for video encoding.

6 Conclusions and Future Works

This paper presented a simple and novel approach to parallelizing MPEG video encoding using shared-memory multiprocessors. Also, two implementation strategies were discussed. Our parallel encoder was able to encode 43 frames per second using only 16 threads. This figure was achieved using a *non dedicated* Sun multiprocessor with 32 processors. Although this machine is still expensive, in a medium term, it will sell as a deskstation (QuadPentium PCs are already sold by less than US\$ 10,000.00).

Two versions of this algorithm were implemented. The encoders were compared regarding speedup and average efficiency. Our experimental results show that the bag of tasks strategy performs better than the fork-join approach. Regardless of the synchronization strategy, our algorithm performed better with longer sequences.

The performance exhibited by our parallel encoder is promising and we believe in further improvements. The more-than-necessary encoding rate suggests that the 30 frames per second threshold can be achieved when considering other overheads (e.g., due to capture and digitization processes) and other standards (e.g. MPEG-2 video). Moreover, it suggests that similar results can be achieved using less powerful and expensive machines. If the multiprocessor is dedicated to the encoding process, 30 frames per second would be sustained without difficulty.

Future work includes (1) implementation of adaptive quantization to improve video quality, (2) audio encoding, and (3) further code improvement in order to achieve 30 frames per second using a less expensive platform (e.g., a multiprocessor Intel PC).

Finally, we would like to acknowledge CENAPAD-MG/CO for making their multiprocessor available and Almadem PROTEM-CC project.

References

- [1] S. M. Akramullah, I. Ahmad, and M. L. Liou. A Portable and Scalable MPEG-2 Video Encoder on Parallel and Distributed Computing Systems. In *Symposium on Visual Communications and Image Processing '96*, pages 973–984, Orlando, FL., 1996.
- [2] Denilson M. Barbosa. MPEG Encoding of Digital Video in Parallel. Master's thesis, Federal University of Minas Gerais, March 1999. in Portuguese.
- [3] Denilson M. Barbosa, João Paulo Kitajima, and Wagner Meira Jr. Real-Time MPEG Encoding in Shared-Memory Multiprocessors. To appear in the 2nd International Conference on Parallel Computing Systems, 1999.
- [4] Vasudev Bhaskaran and Konstantinos Konstantinides. *Image and Video Compression Standards: Algorithms and Architectures*. Kluwer Academic Publishers, second edition, 1997.
- [5] ISO/IEC 11172-2: Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5 Mbit/s - Part II: Video, 1993.
- [6] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads*. Prentice Hall PTR, 1997.
- [7] Joan L. Mitchell, William B. Pennebaker, Chad E. Fogg, and Didier J. LeGall. *MPEG Video Compression Standard*. Chapman and Hall, 1996.
- [8] MPEG-2 Video Encoder, Version 1.1a. MPEG Software Simulation Group, 1994.
- [9] PVRG MPEG codec, Version 1.2.1. Portable Video Research Group, Stanford University.
- [10] K. Shen, L. Rowe, and E. Delp. A Parallel Implementation of an MPEG-1 Encoder: Faster than Real-Time! In *SPIE Conference on Digital Video Compression: Algorithms and Techniques*, San Jose, CA, 1995.
- [11] M. K. Steliaros, G. R. Martin, and R. A. Packwood. Parallelisation of Block Matching Motion Estimation Algorithms. Technical Report CS-RR-320, Department of Computer Science, University of Warwick, Coventry, UK, January 1997.
- [12] H. H. Taylor et al. An MPEG Encoder Implementation on the Princeton Engine Video Supercomputer. In *Proceedings of Data Compression Conference*, pages 420–429, Los Alamitos, CA, 1993.
- [13] A. Murat Tekalp. *Digital Video Processing*. Prentice Hall PTR, 1995.