Real-Time Parallel MPEG-2 Decoding in Software

Angelos Bilas[1], Jason Fritts[2], Jaswinder Pal Singh[1]
{bilas@cs, jefritts@ee, jps@cs}.princeton.edu
[1]Department of Computer Science, [2]Department of Electrical Engineering
Princeton University, Princeton NJ 08544
Phone: (609) 258 - 5329
Fax: (609) 258 - 1771

Abstract

The growing demand for high quality compressed video has led to an increasing need for real-time MPEG decoding at greater resolutions and picture sizes. With the widespread availability of small-scale multiprocessors, a parallel software implementation provides an effective solution to the decoding problem.

We present a parallel decoder for the MPEG standard, implemented on a shared memory multiprocessor. The goal of this work is to provide an all-software solution for real-time, high-quality video decoding and to investigate the important properties of this application as they pertain to multiprocessor systems.

Both coarse and fine grained implementations are considered for parallelizing the decoder. The coarse-grained approach exploits parallelism at the group of pictures level, while the fine-grained approach parallelizes within pictures, at the slice level. A comparative evaluation of these methods is made, with results presented in terms of speedup, memory requirements, load balance, synchronization time, and temporal and spatial locality.

Keywords: Image processing, MPEG-1, MPEG-2, high-performance computing, video compression, real-time system, shared memory.

1 Introduction

Recent advances in network and microprocessor technology have placed video applications within our reach. High Definition Television (HDTV), Broadcast Satellite Service, Cable TV distribution on optical networks, Electronic Cinema, Interactive Storage Media, Multimedia Mailing, Networked Database Services, corporate Internet training and conferencing, Remote Video Surveillance and others are now becoming "practical" applications. The huge amount of data needed to make video available in all these cases has led to the adoption of the MPEG-1 and MPEG-2 standards for motion video compression and decompression. These standards greatly reduce the bandwidth and storage space required. Consequently, MPEG-1 and MPEG-2 are already being used in many video applications and their adoption continues to grow rapidly.

One factor limiting widespread use of the MPEG standard is its computational complexity. Video encoding and decoding under MPEG is expensive, too costly for uniprocessors to achieve real-time performance in software for displays of interesting resolution and size. The computational demands grow as users desire higher quality video. Encoding is more expensive than decoding, but can often be done off-line and hence may not need to be done in real-time. Decoding, however, demands real-time performance. Fortunately, real-time decoding is within the capabilities of bus-based multiprocessors, which are rapidly becoming commodity systems. Almost all major computer system vendors, including PC vendors, now provide such configurations directly, and desktop multiprocessors may not be far away. While special-purpose signal-processing hardware can be used to accelerate decoding [8], the ability to do it in software offers many advantages: it provides greater flexibility for accommodating new algorithms and enhancements as they evolve, does not require additional expensive hardware, and runs on general-purpose systems that can also be used for other purposes.

In this paper, we examine the extent to which increasingly popular cache-coherent bus-based shared memory multiprocessors can be used to speed up software MPEG decoding (we also present some initial results on a cache-coherent machine with physically distributed memory). We present two parallel implementations of the MPEG-2 decoder provided by the MPEG Software Simulations Group[1] [13]. The first version exploits very coarse-grained parallelism across groups of pictures in the video sequence, while the second exploits fine-grained parallelism within each picture. We evaluate their performance and resource requirements for different picture sizes and numbers of processors on a 16-processor Silicon Graphics Challenge multiprocessor, that, though being a fairly expensive multiprocessor, uses the same parallel

[1]Other sequential software MPEG encoders-decoders (codecs) are publicly available [6, 12, 11, 13].

algorithms and techniques as would be used by less expensive desktop servers. Detailed measurements with various performance monitoring tools are used to understand the role of potential bottlenecks to good performance. Finally, we use multiprocessor simulation to characterize the spatial and temporal data locality properties of the parallel versions to understand how they will interact with alternative memory system architectures and to understand how to design multiprocessors and their memory systems to obtain better parallel performance for this class of application.

We first present an overview of MPEG in Section 2, focusing on those aspects that are most relevant to parallelization. Following this, Section 3 describes the testbed that was used for implementing and benchmarking the different algorithms. In Section 4 we present the general methodology for parallelizing the decoder. Section 5 describes the parallel implementations along with the results. In the same section we also study the locality properties through simulation. Section 6 discusses related work and we discuss conclusions and future work in Section 7.

2 MPEG Overview

The MPEG coding standard defines a lossy[2] compression technique which takes advantage of spatial and temporal correlation to achieve high compression ratios. In exploiting spatial correlation, compression is achieved by finding the similarities within each picture and using those similarities to eliminate redundancy in the picture. Spatial correlation alone, however, provides only moderate compression, so temporal correlation must also be exploited. In elminating temporal redundancy, successive pictures within the video sequence are examined for similarities. These similarities are then used to remove redundancies across pictures. Using both spatial and temporal correlation, the MPEG standard provides high degrees of compression on video sequences.

Though both MPEG-1 and MPEG-2 use this compression technique, from a computational perspective MPEG-2 is more interesting due to its greater versatility. While most MPEG-1 video streams follow the *constrained parameters bit stream* format, which limits the the maximum bitrate to 1.75Mb/s and picture size to 352x240 pixels, MPEG-1 can actually be applied to a wide range of input formats. However, MPEG-2 is much more generally defined, providing a coding syntax that is a superset of the MPEG-1 syntax. Some of the additions to MPEG-2 include support for interlaced video sequences[3] and a scalable syntax which allows for layered coding of video streams. While this examination of the parallelism in MPEG-2 has not currently taken full advantage of these extensions, we chose the MPEG-2 decoder with the intent of exploring these avenues in the future.

The MPEG standard specifies the coded representation of picture information for digital storage media and digital video communication. Since video compression can be used in many different applications, and each application places a unique set of constraints upon the video stream, the MPEG committee created standards that are flexible enough to meet the demands of most applications. MPEG-2 provides the greatest versatility, supporting constant or variable bitrate transmission, random access, channel hopping, scalable decoding, bitstream editing, and special play functions such as fast forward playback, fast reverse playback, slow motion, pause, and still pictures (MPEG-1 offers only a subset of these functions). The complete MPEG standards are quite detailed and only a minimal overview may be given here. More comprehensive descriptions may be found in [3, 15].

Layered structure of an MPEG stream: An important aspect of the versatility of MPEG is its layered structure [3, 15]. The hierarchy of layers in an MPEG bitstream is arranged in the following order: *Sequence, Group of Pictures (GOP), Picture, Slice, Macroblock, and Block* (see Figure 1). The different parts of the stream are marked with unique, byte aligned codes called startcodes. These startcodes are used both to identify certain parts of the stream and to allow random access into the video stream. This random access ability is vital to parallelization.

The highest level in the layering is the *sequence* level. A sequence is made up of *groups of pictures (GOPs).* Each GOP is a grouping of a number of adjacent pictures. The purpose in creating such an identifiable grouping is to provide a point of random access into the video stream for play control functions (fast forward, reverse, etc.). In MPEG-1 the GOP is a required layer of the structure, but in MPEG-2 it was made optional because it was found that the sequence layer can serve in the same capacity.

Within each GOP are a number of *pictures.* In MPEG-1, each picture corresponds to a frame, but in MPEG-2 interlaced video is supported so each picture corresponds to either a frame (for progressive or interlaced video streams) or a field (for interlaced video streams) in the original stream. Pictures are further subdivided into *slices*, each of which defines a fragment of a row in the picture. Slices comprise a series of *macroblocks*, which are 16x16 pixel groups containing the luminance and chrominance data for those pixels in the decoded picture[4]. Macroblocks are divided into *blocks* (6 to 12 depending upon format). A block is an 8x8 pixel group that describes the luminance or chrominance for that group of pixels. Blocks are the basic unit of data at which the decoder processes the

[2] A lossy compression scheme is one in which data are lost. Only partial recovery of the original unencoded data is possible.

[3] In interlaced video sequences, each picture is either a frame or a field, where two interlaced fields comprise a frame. A progressive video sequence has no concept of fields and therefore each picture is a frame. Consequently, in interlaced video, pictures that are coded as fields have half the number of horizontal lines as progressive pictures (frames) and must be displayed at twice the display rate.

[4] Luminance is the monochrome representation of the signal and chrominance provides the color information for the video.

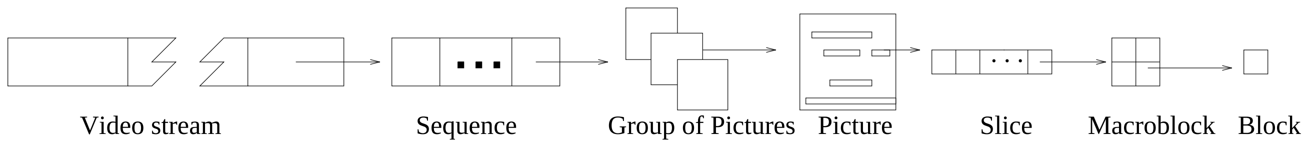

Figure 1: High level bitstream organization in MPEG

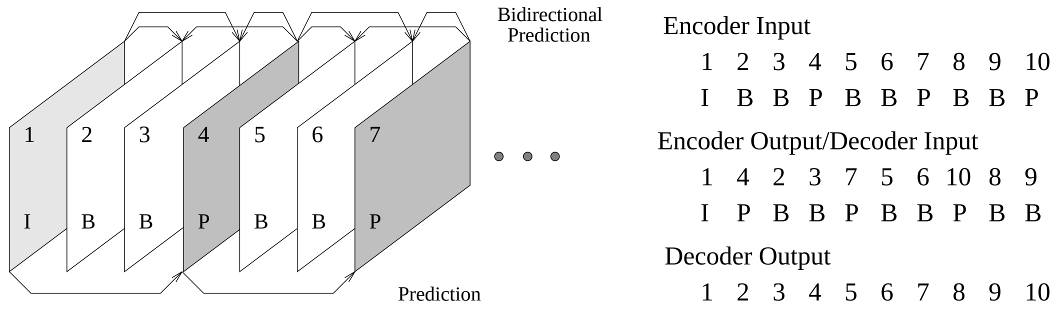

Figure 2: Example of picture structure. Notice that while the encoder input and decoder output both have pictures in the correct display order, the encoded stream has them out of order, such that each I-Picture comes before the P-Picture which references it (i.e. picture 1 comes before pictures 4,7), and each I and P-Picture comes before the B-Pictures that reference it (i.e. pictures 1 and 4 come before pictures 2,3).

encoded video stream. Macroblocks and blocks do not have startcodes associated with them; their boundaries are discovered implicitly while decoding.

Another important aspect of the structure of MPEG is the *picture type*. Pictures in MPEG are encoded into one of three types [5], as shown in Figure 2. All picture types use spatial correlation, but not all use temporal correlation. The first picture type, the *intra* coded picture (I-Picture), uses only spatial correlation. Since their decoding is independent of other pictures, I-Pictures provide access points into the coded stream where decoding can begin. However, using just spatial correlation, they achieve only moderate compression. The second type of picture, the *predictive* coded picture (P-Picture), is coded more efficiently by also using temporal redundancies from a past I or P-Picture. These P-Pictures are then used for reference in further prediction. The final picture type, the *bidirectionally-predictive* coded picture (B-Picture), uses temporal redundancies from both past and future reference pictures, and consequently achieves the highest degree of compression. B-Pictures are never used as references for prediction.

The organization of the three picture types in a stream is very flexible. The only stipulation upon their ordering is that any picture that depends upon temporal redundancies from a past or future picture must be decoded after its reference pictures. As a result, the order of the pictures in the input stream to the decoder (i.e. the output of the encoder) is not the order in which the pictures must be displayed. An example of this reordering in the encoded stream is shown in Figure 2.

Encoding: As described above, the block is the basic unit of data processing. For each block of data in the video sequence, the encoder performs five steps to produce an encoded block: motion estimation, discrete cosine transform (DCT), quantization, and run-length and Huffman coding (see Figure 3).

In the first stage, motion estimation, the encoder tries to take advantage of temporal redundancies among pictures. Motion estimation, required only for P and B-Pictures, is performed using the relevant reference pictures. During motion estimation, each macroblock [6] is moved within a variable-size search window around its position in the reference picture until the position of greatest similarity is found. This displacement is saved as a motion vector. For each block in the macroblock, the difference from the corresponding block at the position indicated by the motion vector in the reference picture is computed. This gives the prediction error for each element in that block.

The next four stages of the encoder take advantage of spatial correlation in compressing the video sequence. Following motion estimation, the encoder takes either the original block data (for I-Pictures) or the prediction error information (for P or B-Pictures) and performs a discrete cosine transform (DCT) on the block to find its frequency spectrum. Typically, in video sequences the higher frequency components are less prominent as they are less noticeable to the human eye, so much of the transformed block has

[5] MPEG-1 actually supports a fourth picture type, the DC-coded picture (D-Picture) type. However, this type is little used and was eliminated from MPEG-2

[6] Motion estimation is only performed on a macroblock basis. All blocks within that macroblock use the same set of motion vectors.

small values. The next step, quantization, zeroes out many of these smaller values, reducing the number of distinct frequency values. This quantization step is the lossy step of the encoding process and the degree of quantization is specified using both quantization matrices and a quantization coefficient.

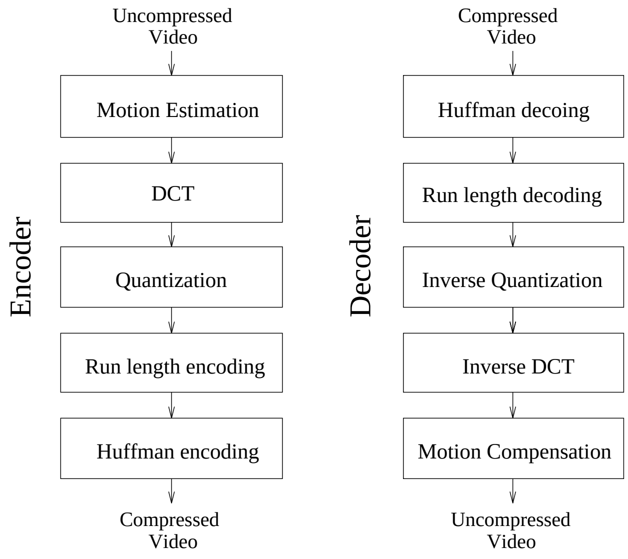

Figure 3: The MPEG model

After quantization, run-length and Huffman coding are performed. Quantized blocks have numerous zeros, and can be efficiently encoded using run-length coding. Only non-zero elements and the number of zeros between them are recorded, saving much space in the encoded result. Finally, the non-zero elements are Huffman coded, a process in which the elements with the most common values are given the shortest codes. The combination of run-length coding and Huffman coding minimizes the amount of data needed to describe the quantized block.

The result of performing these five encoding stages on all blocks in a video sequence is an MPEG encoded video stream. A stream may be encoded once and then transmitted across a transmission media and/or stored as needed. However, decoding is necessary every time the stream is viewed.

Decoding: Decoding is a completely deterministic process. Given an MPEG stream, there is only one way to decode it into a sequence of pictures. The decoding process for an MPEG encoded stream performs the five encoding stages in reverse order. First Huffman and run-length decoding are applied to generate the quantized block. Then inverse quantization is performed to get the block's frequency spectrum. From this, the inverse discrete cosine transform (IDCT) is taken to obtain the block's spatial data (for I-Pictures) or prediction error data (for P or B-Pictures). Then, if necessary, motion compensation may be used to generate the final macroblock data from the prediction error, motion vectors, and reference pictures.

Since the order of the pictures in the encoded stream is not necessarily the display order, it is the job of the decoder to display the pictures in proper order (see Figure 2). However, before displaying a picture more processing (e.g. dithering) may be necessary.

A major difference between the encoding and decoding processes is that in the former, much time is spent in motion estimation as it is necessary to search for the most similar macroblock in the reference picture(s), whereas in the latter, motion vectors are already available, which makes motion compensation much cheaper than motion estimation.

3 System Environment

This section describes the hardware and software environment that was used in the implementation as well as the testing methodology we used to benchmark the different algorithms.

The Multiprocessor platform: The SGI Challenge multiprocessor is a cache-coherent, bus-based, centralized shared memory multiprocessor. The machine we use has 16 processors connected by a 256 bit-wide bus with peak bandwidth of 1.2GB/sec. Each processor is a 150MHz MIPS R4400 with peak performance of 75 MFlops. Each node has first level data and instruction caches of 16KB each (direct-mapped) and a unified second level cache of 1MB (2-way set-associative). The system has 1GB of main memory that is 8-way interleaved, out of which we could use up to 500MB for our program. The system can support up to 4 I/O buses, each 320MB/sec peak. The operating system is IRIX 5.3 and it incorporates substantial functionality from UNIX System V, Release 4.

Since the machine supports a shared address space programming abstraction, shared data can simply be allocated as such and then referenced directly by any processor. Our parallel programs are written in C, augmented with the `parmacs` parallel programming macros from Argonne National Laboratory [2]. Porting the program to other shared address space architectures is easily achieved by using the proper version of the `parmacs` system for the architecture under consideration. The macros are also supported by our multiprocessor simulation environment, so we can measure various characteristics of the parallel programs.

Performance analysis tools: To understand performance characteristics and bottlenecks, we used the `pixie` and `prof` tools. `pixie` instruments the code and counts basic blocks. The ideal execution time of each process in a program is then calculated given the cycle time of the system and a one cycle delay for every memory reference. `prof,` on the other hand, uses *program counter sampling (pc-sampling)* to determine where time is spent, and reports actual execution time including memory stall time. However, the timing results reported by `pixie` and `prof` include synchronization time, which must be computed separately by instrumenting the source code. The results of `pixie` and `prof` may then be combined to determine the time spent in the memory subsystem.

Stream	Resolution	GOP size	Picture size
1-4	176x120	4,13,16,31	22K
5-8	352x240	4,13,16,31	82.5K
9-12	704x480	4,13,16,31	330K
13-16	1408x960	4,13,16,31	1320K

Table 1: Description of test streams.

Test streams: We tried to be as consistent as possible in choosing the input sequences. Most public domain sequences are small, have random characteristics, and do not explore the parameter space in any systematic way. We therefore created our own set of test streams. Starting with a small public domain stream, a moving view of a flower garden with 150 pictures and resolution of 352x240 pixels (flowg.mpg, from Stanford), we created larger streams by repeating a number of pictures in a continuous video sequence and scaling each picture using interpolation. Each resulting stream is composed of a total of 1120 pictures, has a 30 pictures/sec display rate and 5 or 7 Mbits/sec bit rate. The I/P picture distance is 3, thus there are 2 B-Pictures between any two consecutive reference I or P-Pictures. Table 1 shows the characteristics of the streams[7].

We vary only two parameters in our test streams: the resolution and the number of pictures per GOP. These are important because they define the amount of processing required to decode a picture as well as the memory requirements of the system. As seen in Table 1, we use four different resolutions (176x120, 352x240, 704x480, 1408x960)[8] and four different numbers of pictures per GOP (4, 13, 16, 31) for a total of 16 streams.

The public domain MPEG-2 encoder [13] we used to create the streams creates one slice for each row of a picture. Thus the four different picture sizes have 8, 15, 30 and 60 slices respectively from the smallest to the largest. Similarly, most public domain video sequences we found also have a small number of slices per picture (usually one per row).

One other parameter of video streams that is of great importance is the bit rate. This is because the bit rate provides a measure of both the degree of compression and the relative quality of the video. All of the results presented in this paper used the aforementioned video streams which assumed a fixed bit rate of 5 Mb/s for the 352x240 and 704x480 streams and 7 Mb/s for the 1408x960 streams. But since bit rates can vary considerably according to the desired degree of compression or video quality, we also felt it important to examine the effect of different bit rates on parallelism. Using streams of widely varying bit rates, we found that the decoding times for streams of a given picture size are typically within 10%-15% of the time measured for our test streams. However, this decoding time differential is seen to a proportionate degree with an increasing number of processors. The overall result is that the speedups are consistent. So, even though the absolute execution time deviates slightly for video streams of different bit rates, there is no noticeable impact on parallel performance.

4 Exploiting Parallelism

The amount of work associated with decoding different pictures, and even with different parts of the same picture, is variable and unpredictable, so maintaining a balanced workload requires that we use some form of dynamic tasking mechanism. Static assignment of tasks to processes is also difficult because tasks are not known ahead of time but are created as the input is read, in parallel with the actual computation. We present two different methods for exploiting parallelism. In both methods the incoming stream is decomposed into tasks that are put in task queues and can be processed in parallel. The difference lies in the nature and granularity of the tasks, which affects the performance and characteristics of a parallel implementation.

The possible choices for a task in MPEG are: sequence, group of pictures (GOP), picture, slice, macroblock and block. Given the encoding scheme in MPEG, only a GOP and a slice are reasonable choices, as we shall see.

The first type of parallelism is across pictures. Since P and B-Pictures depend on other nearby pictures, assigning adjacent pictures to different processors leads to many serializing dependencies, and associated synchronization and communication among processors. A better solution is to parallelize across completely independent GOPs. Parallelizing at either the sequence or GOP level might work, however the sequence level may lead to tasks which are too large and create load imbalance[9]. Therefore, the GOP level is a more reasonable choice. At the GOP level, tasks can still be very coarse-grained, but since GOPs are relatively independent, there is essentially no inherent communication in the parallel algorithm except in accessing shared task queues. This forms the first approach, which we call the GOP-level implementation.

In the second type of parallelism, parallelism within a picture, the only plausible approach is to use slices

[7] In MPEG-2 terminology, all the streams have a *main* profile and a *high* level.

[8] The last two streams are more commonly found with pictures sizes of 720x480 and 1440x960. We used the uncommon sizes to maintain consistent picture size ratios.

[9] Recall that in MPEG-2 the GOP level is optional. When the GOP level is used, sequences are typically large, but when it is not used, sequence sizes are usually smaller. Hence, when the GOP level does not exist, the sequence level may be used for parallelization.

as tasks since they are marked with startcodes in the input stream. The problem with macroblocks and blocks is that they do not have startcodes to identify them without actually doing the decoding itself (if they did, we could used them as tasks as well). Parallelizing at these levels would make it necessary for one process to perform the decoding of the stream, detect the boundaries of each macroblock (including its motion vectors) while processing the stream, and assign the macroblock or its blocks to other processors for further processing. While this approach may be viable, it places a large load on one processor and would only pay off in cases where there is a significant problem with load imbalances at the slice level. Because slices are groups of adjacent macroblocks, they increase task granularity compared to macroblocks, potentially causing load imbalance, but help reduce communication when reconstructing blocks from motion vectors and other nearby blocks. Our other parallel implementation, which we call the slice-level implementation, defines the task unit to be a slice. We shall discuss both these parallel versions and their tradeoffs further in subsequent sections.

5 Experimental Results

5.1 Parallelism at group of pictures level

The first approach taken in parallelizing the decoder tries to exploit parallelism at the GOP level. In this implementation, we define the task unit to be a GOP.

Recall that a GOP consists of any number of pictures. By definition, a GOP must contain at least one I-Picture. Also, the first picture (in display order) in a GOP must be an I-Picture or a B-Picture, and the last picture in a GOP must be an I-Picture or a P-Picture. If the first picture is an I-Picture or a B-Picture that does not depend on the pictures of the previous GOP, then the GOP is defined as a closed GOP and it can be decoded independently.

Since consecutive GOPs can be decoded by different processors, GOPs need to be closed. Although the assumption that all GOPs in the stream are closed is not necessarily true of the streams generated by encoders, it is in fact not very restrictive. One way to overcome it even when the GOPs in the input stream are not closed is by taking advantage of the fact that the stream contains startcodes for pictures and identifies their type with a type field. This parallel design does not require that tasks be GOPs as defined in the input stream, but rather any closed set of pictures that can be decoded independently. The scan process could scan the stream and construct closed tasks. This would be somewhat complicated by the fact that each of these tasks would have to contain pointers to appropriate headers, so that the worker processes could find configuration information specific to the pictures that it would decode. For simplicity we assume that the GOPs in the input stream are themselves closed, and can be processed independently.

Figure 4 shows the architecture of the parallel decoder. We dedicate one process, the scan process, to read the stream from the disk (or network or other source) and identify the tasks. While reading the stream into memory, it scans the stream for startcodes that mark the beginning of each GOP. All but one of the other processes are worker processes which dequeue tasks from the task queue and decode the corresponding GOP. Each worker processor operates in a loop in which it removes the next task from the queue, decodes the corresponding GOP and places the decoded pictures in a display queue. The worker processes assume that the data for the GOPs in the task queue are already in memory. The last process is assigned as the display process. It is responsible for displaying the pictures which may have been processed and inserted in the display queue out of order - in the correct order. It is also responsible for dithering the pictures. In the measurements we present, we do not include dithering time since it is not a necessary part of the decoding. The dithering cost can vary greatly depending on the characteristics of the display device.

The scan process and the display process do not take part in the decoding process. Another approach would be to have them participate when free, but this would complicate the system. Moreover, as the requirements for better image quality, and consequently picture resolution and bit rate, increase, these processors will be busy most of the time anyway.

The speed at which the scan process is placing pictures in the task queue is shown in Table 2. Here we assume (quite reasonably) that the scan process can be fed with data at the required bit rate. Doing this under a variety of conditions is a topic of current research in the areas of networking and I/O.

Picture size	352x240	704x480	1408x960
File size(Mbytes)	25	25	45
Number of pictures	1120	1120	1120
Scan time(sec)	4.5-6.5	4.5-6.5	11-14
Scan rate(pics/sec)	170-250	170-250	80-100

Table 2: Scan rate in the scan process.

The motivation for the GOP level design is twofold. First, tasks are large and independent, so accesses to the task queue are infrequent and there is little synchronization or inherent communication among the worker processes. Second, the resulting system is easy to understand and implement.

Picture size	352x240	704x480	1408x960
Max pictures/sec	69.9	26.6	7.3

Table 3: Maximum number of pictures/sec decoded for each picture size.

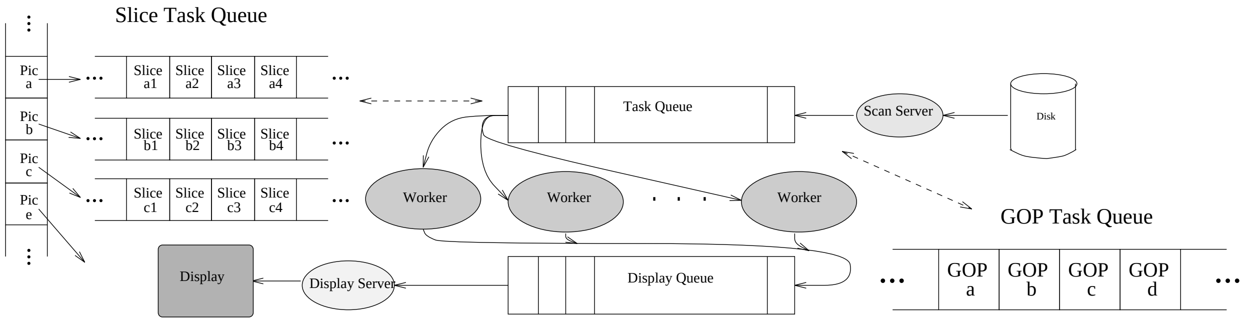

Figure 4: Architecture of the parallel decoder

5.1.1 Results

In our measurements on the Challenge, we tried to capture the behavior of the decoder in terms of speedups, memory requirements, load balance, and the components of execution time including memory overhead. We omit the results obtained for the smallest resolution (176x120) in all cases due to space limitations.

Performance and speedup: We measure speedup as the ratio of the number of pictures per second that P worker processors ($P+2$ total processors) can decode to the number of pictures per second that are decoded by one worker processor (3 total processors). This is different than the speedup obtained over a uniprocessor system, which would multiplex the scan and display processes with the worker process in the uniprocessor baseline and hence likely inflate the speedups. The results, in Figure 5, show that the speedup is almost linear in all cases. Table 3 gives the maximum number of pictures per second decoded for each picture resolution, using 14 worker processes.

Load imbalance: To capture load imbalance we measured the minimum, maximum and average computing times among the worker processes. Figure 6 shows that when the number of pictures per GOP is small, the minimum and maximum times are very close to the average. This means that all the worker processes spend approximately the same amount of time computing. As the number of pictures per GOP increases, the load imbalances become more apparent because the number of tasks decreases and the time to process a task increases with GOP size. Thus if one or more processors process even one more task than the rest they will appear to have computed significantly more. In reality even this is just an artifact of the relatively short input stream, and load imbalance among workers is not likely to be a problem for real streams that contain many GOPs.

Memory system and synchronization overheads: The time spent executing instructions stalled in memory and waiting at synchronization points was measured by `pixie`, `prof` and source level instrumentation as discussed earlier. The `pixie, prof` times in Figure 7, show that in all the cases 10%-30% (with an average of 20%) of the time is spent stalled in memory. We shall study cache miss rates and memory system interactions through simulation later.

Synchronization time among the worker processes is minimal. They only need to synchronize when accessing common resources. The time spent on locks was measured to be negligible compared to the processing time in each slave.

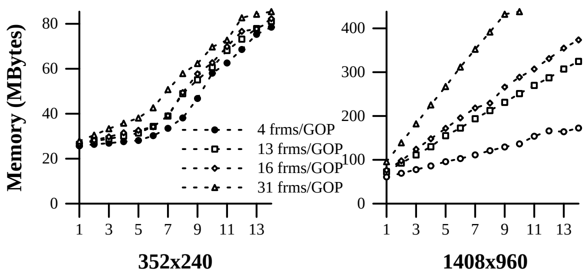

Figure 8: Actual memory requirements for the GOP approach

Memory requirements: The maximum memory required by the system depends on the number of processors and the picture size. Each processor needs to keep up to three decoded pictures in memory (two reference pictures and the one currently being decoded). The speed of the scan server affects the memory requirements as well. It allocates memory to store the data it reads from disk, potentially increasing memory requirements when it outpaces the decoding. In Figure 8 we plot the maximum amount of memory used by the decoder for each test stream. We see that the memory used by the system grows with the size of the GOP, the size of the picture and the number of processors used. In many applications this is not a problem because the size of a GOP is relatively small. For instance, applications that require random access, fast-forward playback, or fast-reverse playback favor the use of relatively short GOPs.

To predict memory requirements for different

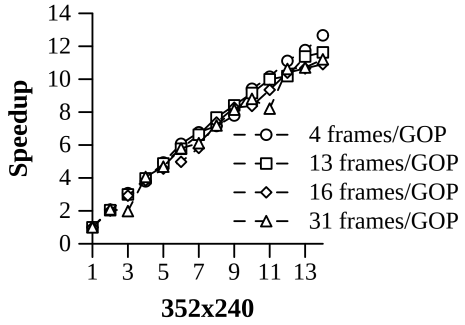

Figure 5: Speedups for the GOP approach.

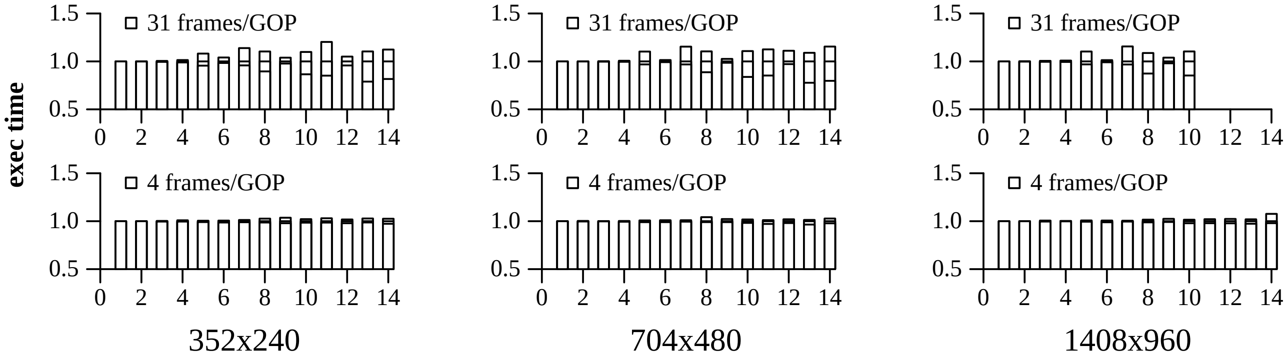

Figure 6: Load balance for the GOP approach. The x axis is the number of worker processors used and the y axis the minimum, average and maximum execution times among the worker processes (normalized to the average).

streams and system configurations we derived an analytical model that gives the memory requirements of the program as a function of execution time, taking into account the speeds of the scan server and the decoding process. Figure 9 shows the predicted requirements in memory for three cases as a function of time. The third case (1408x960, 31 pictures/GOP, 11 processors) cannot be run on the system because the memory requirements exceed the available memory. Despite the dynamic allocation and deallocation of memory as tasks are created and completed the model has been verified to be very close to the actual behavior of the system.

In addition to the large memory requirements, this method also has the problem that it has large random access latency for play functions. For example, should the user fast-forward to a later section of the video sequence, decoding must begin anew at that point, with each processor grabbing one GOP. Because only one processor processes a GOP, the speed at which the video begins to display at that point is dependent upon one processor, not all the processors. As a result, the GOP parallel method is better suited to continuous play.

5.2 Slice level parallelism

Addressing these problems led us to consider another approach to parallelizing the decoder. In our second parallel decoder implementation, we define the task unit to be a slice.

As defined by the standard, a slice in MPEG is a series of an arbitrary number of macroblocks within one row of the picture. Each slice must contain at least one macroblock, and consecutive slices may not overlap. Slices occur in the bitstream in the order in which they are encountered, starting at the upper-left of the picture and proceeding by raster-scan order from left to right and top to bottom. However, slices do not need to maintain the same structure from picture to picture: there may be different numbers of slices and/or different slice sizes.

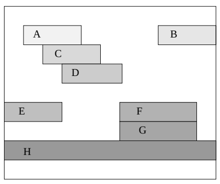

Figure 10: The general slice structure in a picture.

In the most general case, it is not necessary for slices to cover the entire picture (Figure 10). Areas not enclosed in a slice are not encoded. However in all the profiles defined so far by the standard a *restricted slice structure* is used, in which every macroblock in

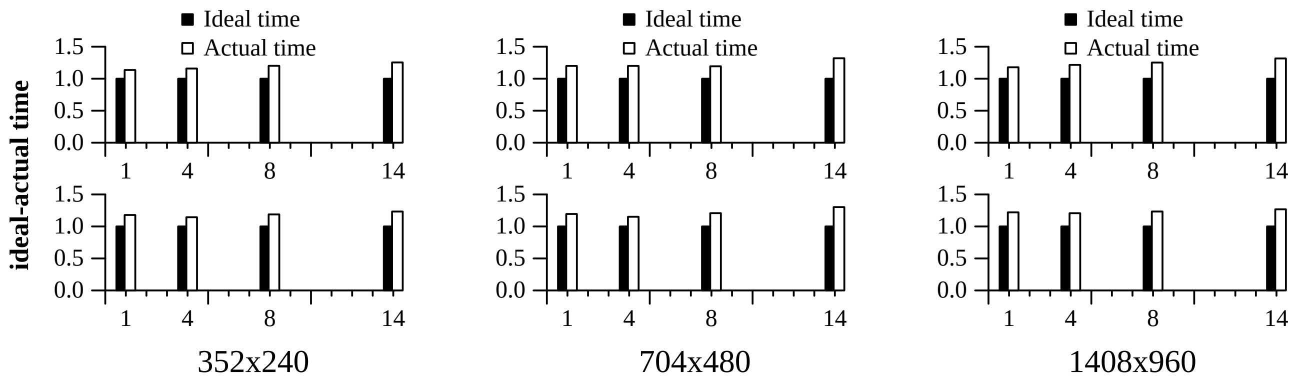

Figure 7: Ideal and actual time for the GOP approach. The x axis is the number of processors and the y axis the ideal(black bars) and actual(white bars) time as given by pixie and prof respectively, summed over all worker processes. The results are normalized to the ideal time.

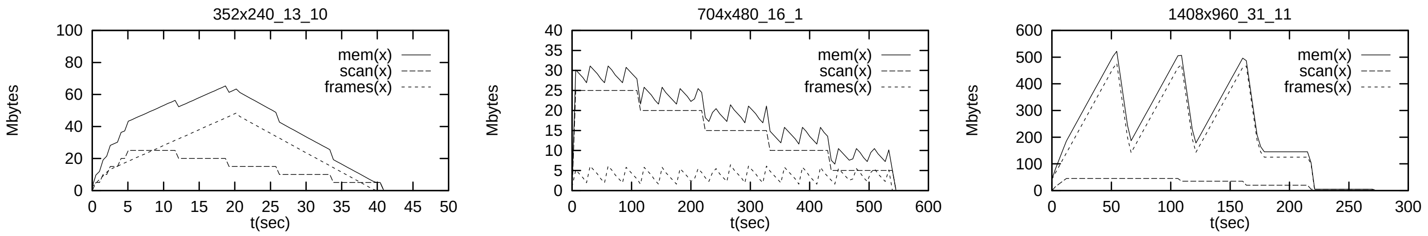

Figure 9: Predicted memory requirements for three cases. `mem(x)` is memory usage, `scan(x)` is the portion that corresponds to the scan process and `frames(x)` the memory allocated for pictures. Note that `mem(x) = scan(x) + frames(x)`.

the picture is enclosed in a slice.

The architecture of the decoder is basically the same as in the first approach. However, because of the need of the processors to access picture header information while decoding slices and the need to synchronize at picture boundaries, we found that a 2-D task queue facilitates a simpler implementation. The first level of the task queue holds pictures, while the second level holds the slices within those pictures, as can be seen in Figure 4. The scan process, while scanning the input stream, adds a task to the picture task queue for each picture startcode found, and then, for each slice startcode found within that picture, it adds a task to the slice task queue associated with that picture. When decoding a picture, worker processes remove tasks from the slice queue assocated with that picture. When the current slice task queue empties and the slices are all decoded, the picture is placed in the display queue and the next task from the picture task queue is accessed. The slice queue associated with that picture then becomes the new task queue. In our first implementation (simple slice version), processors synchronize at the end of every picture, so parallelism is only exploited within a picture. Also, no attempt is made to preserve locality across the slices from different pictures that are assigned to the same processor.

Two important differences from the GOP-level approach is that the memory requirements are much lower and the closed GOP assumption is not necessary. Since all the processors in the system work on the same picture, which is in shared memory, at most three pictures in all need to be in memory at a time (versus three pictures per processor as required by the GOP version).

The other benefit of the slice version is that it does not have the random access latency problem for play control functions. When a play control function causes play to begin from a new position in the video stream, all worker processors, not just one, immediately begin decoding the new GOP, slice by slice, in parallel.

The disadvantages of the slice approach are synchronization and inherent interprocess communication. Processes communicate as they access the same macroblocks from the reference pictures, particularly if those macroblocks (slices) were assigned to and written by other processes in the reference picture.

As noted above, the worker processes in this implementation synchronize at the end of each picture. The number of slices per picture is not defined by the standard, but most test sequences we found used only one slice per row of macroblocks. Thus each picture usually contains a small number of slices (the vertical resolution divided by 16, the vertical size of a macroblock). This has an important impact on the load balance and the performance of this approach. For example, a 704x480 picture has 30 slices. If we use 14 workers to decode such a picture, two workers will get three slices while the other twelve only get two and will be idle while the first three are decoding their final slice. Figure 11 shows how this creates a serious problem in speedups. Particularly since it happens in every picture. There is an improvement in execution time as processors are added only when the load is divided equally between all the processors.

To remedy this, we improved the implementation to synchronize the workers only after certain picture types, not after every picture (improved slice version). The key observation is that all B-pictures in a series use the same reference pictures and are not themselves used as reference pictures. Thus, since the next picture does not depend on the picture currently being decoded, available workers can begin decoding the next picture after completing all their tasks on the current picture. Synchronization is needed only at the end of an I or P-picture. This does not exploit the maximum concurrency, but that would require complex synchronization at the slice level.

In the following paragraphs we present and explain the behavior of these two fine grained implementations.

5.2.1 Results

Results for the slice implementation are presented in the form of speedups, synchronization time (load imbalance) and ideal versus actual time. Compared to the results for the GOP approach, memory requirements are very low and practically independent of the number of processors and the GOP size. We present synchronization time results for load imbalance rather than max-min-average results over the execution since worker processes synchronize at picture boundaries. Since the effectiveness of the slice approach does not depend on the number of pictures in a GOP, we only vary the size of the pictures and keep the GOP size constant at 13 pictures in this case.

Performance and speedup: Speedups are measured in the same way as in the GOP approach. Figure 11 shows the performance of both slice implementations. We see that if the processes synchronize at every picture, then speedups are nearly linear only for large pictures (that contain many slices). The knees in the simple version, especially in the 704x480 and 352x240, happen when the integer ratio of the number of slices in a picture over the number of processors, is reduced by one. In the 352x240 case each picture

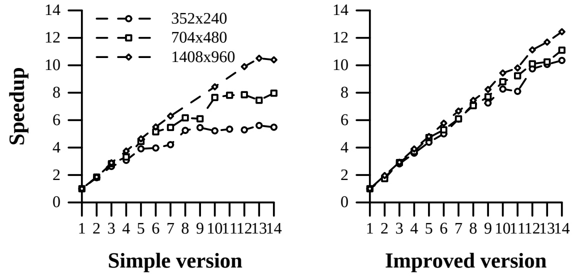

Figure 11: Frames/sec for the slice approach

contains 15 slices so performance doesn't increase for more than 8 processors.

The improved version greatly reduces this imbalance. The number of slices that can be processed is larger, depending on the I/P distance of the stream. This implementation exposes enough slice level concurrency for the numbers of processors used and achieves very good speedups for all picture resolutions.

Frame size	352x240	704x480	1408x960
Simple version	27.7	15.1	5.6
Improved version	54.7	21.6	6.8
GOP version	69.9	26.6	7.3

Table 4: Maximum number of frames/sec decoded for each picture size.

Table 4 gives the maximum number of frames per second decoded for each picture resolution. From from this table we see that the improved slice version has comparable performance to the GOP version. It is slower due to the increased overhead in managing the finer tasks (reading picture headers multiple times, etc.) and the additional synchronization time needed at picture boundaries.

Synchronization overhead: Figure 12 shows the average ratio of synchronization wait time to execution time for all workers as a function of the number of worker processes. It clearly shows that the improved version performs better. The times reported include both time spent in the task queue and at synchronization points, though the former is very small compared to the synchronization time. Thus, although task granularity is much smaller for this version, using a centralized task queue does not constitute a problem for the slice approach either.

The knees that were mentioned before are again visible in Figure 12, only now they are reversed. The ratio of synchronization to execution time generally increases or remains constant. However, it does drop whenever the ratio of slices per picture to the number of the worker processes decreases.

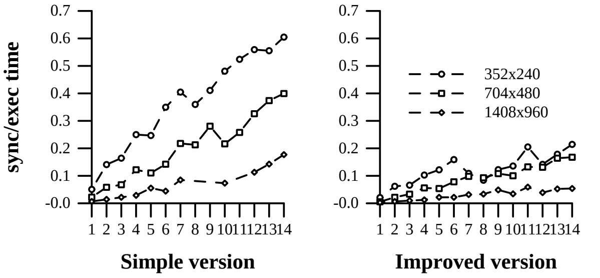

Figure 12: The average (sync time/exec time) of all worker processes versus the number of processors in the slice method. The more slices/#procs the less the sync time. For certain numbers of processors this diagram reflects that more processors stay idle longer in the simple version. In the improved version synchronization does not increase as much with the number of processors.

Memory system: Using `prof` and `pixie` we found the actual time to be on average within 95% of the ideal, which means that cache misses cost very little in this approach as well.

Thus, both GOP level and slice level approaches give good speedups, though the latter uses much less memory. While the memory requirements of the former increase linearly with GOP size, picture resolution and number of processors, the requirements of the latter depend only on picture resolution. Additionally, the GOP level approach has long random access latency for play control functions. On the other hand, the slice level version has somewhat higher synchronization and communication needs, which reduces its speedup. We shall now examine the memory system interactions of these versions more closely to determine the expected scaling when using larger machines and larger picture sizes, and to see how different cache organizations impact performance.

5.3 Locality properties

To understand the temporal and spatial data locality, we performed software simulations of the multiprocessor execution for the program. The simulations were done using the TangoLite execution-driven reference generator coupled to a memory system simulator [5]. The simulator models a cache-coherent multiprocessor with one level of cache per processor and is flexible in setting architectural parameters. In the results we present miss rates (ratio of misses to memory references), not timing numbers, because the impact of some types of cache misses on performance can be hidden by using relaxed memory consistency models and other latency tolerance techniques.

For spatial locality, Figure 13 shows the read miss rate of the GOP version versus the size of the cache line for a 1 Mbyte, fully associative, cache. We see that the miss rate halves whenever the cache line size doubles, which indicates that the program has very good spatial locality. The results are for the GOP version, but the slice version has similar behavior.

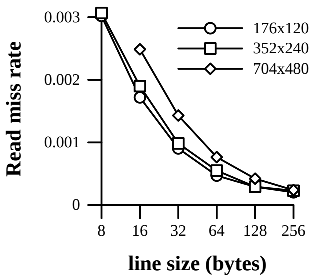

Figure 13: Read miss rate versus line size for an eight-processor execution and 1M , fully associative cache.

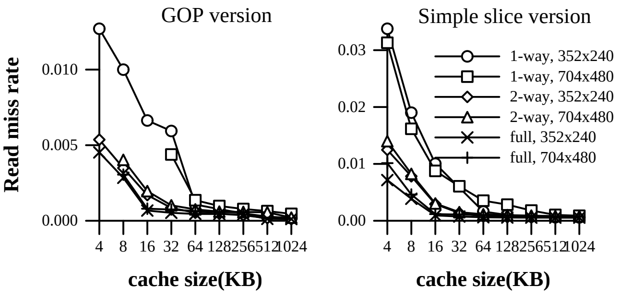

Figure 14: Miss rate versus cache size. Left: GOP version for 1 processor and a 64-byte cache line. Right: Simple slice version for 8 processors and a 64-byte cache line.

The size and scaling of a program's working sets (i.e. its temporal locality) are important to understanding how data traffic and performance will scale to larger problems and machines, and for determining what cache sizes will be necessary for good performance. We measure the working sets of the program by plotting the read miss rate versus the cache size (per-processor) used in the simulations. Since the GOP approach doesn't have any sharing among the worker processes (which all do similar work) we assume a one processor execution and one-way, two-way and fully associative caches with a 64-byte cache line size. For the slice level approach we present results using

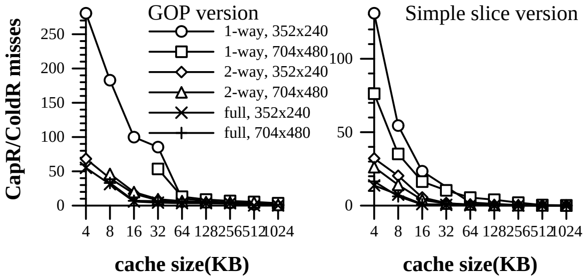

Figure 15: Ratio of read capacity misses over read cold misses versus cache size. Left: GOP version for 1 processor and a 64-byte cache line. Right: Simple slice version for 8 processors and a 64-byte cache line.

eight worker processors. The results for a single worker processor will be essentially the same as for the GOP approach, because the slice level approach incurs inherent interprocessor communication, and because with parallelism within a picture the access patterns change as the number of processors changes. The simulation results indicate that the miss rate for large cache sizes is dominated by cold misses rather than communication misses even in this case. The number of true sharing misses is small in comparison, and false sharing negligible.

As for capacity misses, we find (Figure 14) that the read miss rate drops dramatically for caches larger than 16K or 32K bytes as long as the caches have some associativity. Direct mapped (one-way associative) caches may need to be larger than 64K bytes to fit the working set. This suggests that the working sets are relatively small. To verify this we plot the ratio of the read capacity to the read cold misses versus the cache size, Figure 15. This figure shows that the number of capacity misses is small compared to the cold misses. Thus the capacity miss rates and traffic do not constitute a bottleneck and increasing the cache size further, beyond the working set size, does not significantly improve performance. The results also show that the working set size does not change with the picture size or the number of processors, even for the slice-level version, suggesting that it is determined by the data used for the reconstruction of a single macroblock or set of macroblocks, which is independent of these parameters. The working sets are therefore expected to fit in the caches of modern multiprocessors and capacity misses are not likely to be a problem even for very large pictures in either approach. We can expect to continue to obtain good parallel performance on MPEG decoding at least on small to moderate scale multiprocessors.

6 Related Work

We are not aware of any work so far that has demonstrated successful parallelization of MPEG-2 decoding on shared memory multiprocessors, or examined the performance bottlenecks and characteristics. The work on parallel MPEG-2 has focused on message-passing systems, and mostly on the encoding process with its much greater computational costs. Reported work has not analyzed the bottlenecks or the important data locality characteristics.

[1] presents a parallel MPEG-2 encoder for large scale multiprocessors. Parallelism is exploited at the block and macroblock level. They report real time encoding at rates higher than 30 pictures/sec for an Intel Paragon XP/S with 330 processors. Their test streams have relatively low resolution (maximum 360x288) and there is little performance analysis to enable extrapolation.

[7] describes a parallel decoder that exploits parallelization at the GOP level in a message-passing environment, but deals only with MPEG-1 streams.

[14] presents an MPEG-2 video encoder for a LAN of workstations. They conclude that for their approach the best parallel scheme should be based on the slices. Parallelizing at the macroblock level is found to be impractical for the general case. Real-time encoding is achieved for relatively small picture sizes.

Work has also been done in designing hardware or combined hardware-software codecs that achieve real-time performance. [8] describes a software solution on the Multimedia Video Multiprocessor (TMS320C80), and reports real-time results for small picture encoding-decoding. They also mention that the proposed solution achieves real time decoding for larger pictures as well.

7 Discussion and Conclusions

7.1 Conclusions

We have investigated the behavior of two parallel implementations of the MPEG-2 decoding process on shared memory multiprocessors. Parallelization was performed at the GOP level and slice level, respectively, using the `parmacs` parallel environment. Both methods demonstrated very good speedups. The results we presented were obtained using a SGI Challenge system, but the two implementations are portable across a large number of platforms.

Parallelism across GOPs reduces the synchronization required among processors, but has large random access latency for play control functions and can lead to extreme memory requirements that increase linearly with the GOP size (number of pictures per group), picture resolution, and number of processors. On the other hand, parallelizing at the slice level increases synchronization and communication somewhat but essentially eliminates the random access latency problem and dramatically reduces memory requirements. The memory requirements for this approach do not increase with GOP size or number of processors, but only with picture resolution. We also used a simulator to investigate the cache behavior of the decoding process, and found excellent spatial and temporal locality.

The results show very good parallel speedups for both versions, and that we can achieve real time decoding for reasonable sized pictures (352x240, 704x480) on small-scale shared memory multiprocessors. For larger pictures (e.g. 1408x960), close to real-time performance may be achievable with high end systems using the latest processors, and perhaps further optimization of the serial uniprocessor code (we have not tried to optimize the code from the Software Simulation Group other than through the compiler). Also, since communication is small, and since our results show that capacity misses are not a problem and working sets do not grow with picture size, more and more processors can be brought to bear as larger resolutions are used, even in the slice level version. The trend in microprocessors toward the use of long cache lines also helps speed up decoding.

7.2 MPEG-2 on Distributed Shared Memory Architectures

New issues arise when we move to moderate-scale cache-coherent machines that have physically distributed main memory. Will a fully dynamic scheme, with no attempt to preserve locality in task assignment, work well as communication costs get relatively higher? For example, in our slice level implementation we make no attempt to ensure that the processor decoding a given slice is also assigned slices from later frames which reference that slice. And will data placement across physically distributed memories be important and manageable? We are currently investigating these issues on cache-coherent distributed memory machines. Preliminary results on the Stanford DASH multiprocessor [9] show that both the GOP and the improved slice version scale as long as there are enough tasks to keep the worker processes busy. However, for these programs written without any attention to data distribution speedups are not as good as on a centralized shared memory system.

For instance, for 704x480 pictures, the improved slice version runs 1.8, 3.4 and 5.2 times faster on 8, 16 and 30 processors as compared to 4 processors (one cluster in DASH), and the GOP version speeds up a little less. Data placement on a distributed shared memory system affects the number of remote misses each process experiences, and consequently the performance of the system. Using a hardware performance monitor, we observed the latency (not contention) of a lot of remote misses as being the major impediment to speedup on DASH (synchronization wait time and hence load imbalance is not a problem for either the GOP or improved slice versions), so smart data placement across main memories may be necessary to improve speedups. This is difficult for the slice level scheme, but can be done for the GOP level by using a task queue per processor, having a processor be assigned the tasks corresponding to GOPs that are loaded into its local memory (GOPs may be loaded in round-robin order among memories), and then have them steal tasks from other queues for load balancing. Our simulation results for the low communication miss rate and the small working sets suggest that this scheme should work well on moderate-scale machines, but this bears more detailed investigation.

7.3 Future Work

As mentioned in the MPEG Overview, there are many extensions over MPEG-1 that were added when creating MPEG-2. Foremost among these were support for interlaced video and a scalable syntax which allows layered coding of video sequences. In our initial work, we primarily explored parallelization as it applied to the general MPEG standard (both MPEG-1 and MPEG-2). While the scalable syntax is not often used in today's applications, this is primarily due to the difficulty in handling the large amount of data that accompanies the upper level of the scalable video sequence. This will likely change as technology grows to meet these needs. Unlike the scalable syntax, interlaced video is already commonly used in MPEG-2. Hence, it will be necessary to explore the parallelization of both these extensions to provide a complete multiprocessor solution to the problem of real-time MPEG-2 decoding.

Clearly the 16-processor Challenge system used in our experiments is not an inexpensive commodity system. However, it gave us the ability to investigate different aspects of MPEG-2 decoding with a significant number of processors. Given that processor technology has already moved to the next generation of processors (R10000, P6, UltraSparc, etc.), small scale SMP systems that are becoming commodity systems, will outperform the Challenge system we used. This work shows that an all software solution to the MPEG-2 decoding problem is both viable and interesting. To verify how well these new systems (especially those processors with extended instructions sets for graphics support, e.g. UltraSparc, P6, etc.) will do, we plan to port and run the algorithms presented in this work on a 4-processor P6 system.

Exploring different architectures is another future direction. Work on software shared memory systems has shown that medium scale systems that support a shared memory abstraction in software may provide users with an easy to program and highly cost effective solution. Investigating the behavior of the parallel decoder under these systems, for example shared virtual memory in which coherence is maintained at page granularity, is an interesting problem for further study.

7.4 Acknowledgments

We thank Somnath Ghosh for his help in understanding the structure of the decoder. We are indebted to NEC and particularly to James Philbin for generously providing the Challenge system we used for the measurements as well as to Kasinath Anupindi and Henry Cejtin for their help in managing the disk space and the cpu time we used. We would also like to thank Liviu Iftode for his help in using the Challenge system.

References

[1] Shahriar M. Akramullah, Ishfaq Ahmad, and Ming L. Liou, "A Data-Parallel Approach for Real-Time MPEG-2 Video Encoding", Journal of Parallel and Distributed Computing, 1995 Nov. 1. 30(2), pp. 129-146.

[2] Argonne National Laboratory, 9700 South Call Avenue, Argonne, IL 60439-4801, "Parmacs Parallel Programming System".

[3] V. Bhaskaran, and K. Konstantinides, "Image and Video Compression Standards: Algorithms and Architectures", Kluwer Academic Publishers, 1995, pp. 161-194.

[4] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", Proceedings of the 17th Annual International Symposium on Computer Architecture, 15-26, 1990, May

[5] Stephen Goldschmidt, and Helen Davis, "Tango Introduction and Tutorial", Computer Systems Laboratory, Stanford University, Stanford, CCA 94305.

[6] Andy C. Hung, "PVRG-MPEG CODEC 1.1", Portable Video Research Group (PRVG), Stanford University, June 14, 1993.

[7] M. K. Kwong, P. T. P. Tang, and B. Lin, "A Real-Time MPEG Software Decoder Using a Portable Message-Passing Library", Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439-4844.

[8] W. Lee, J. Golston, R. J. Gove, and Y. Kim, "Real-time MPEG Video Codec on a Single-Chip Multiprocessor", Proceedings of the SPIE, Digital Video Compression on Personal Computers: Algorithms and Technologies, Vol. 2187, Feb. 1994, pp. 32-42.

[9] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam, "The Stanford Dash Multiprocessor", IEEE Computer, Vol. 25, No. 3, March 1992, pp. 63-79.

[10] K. Shen, L. A. Rowe, and E. J. Delp, "A parallel Implementation of an MPEG-1 Encoder: Faster Than Real-Time", Proceedings of the SPIE conference on Digital Video Compression: Algorithms and Technologies, Vol. 2419, 7-10 February 1995, San Jose, California, pp. 407-418.

[11] Plateau Research Group, "Berkeley MPEG-1 Video Encoder User's Guide", Computer Science Division, University of California, Berkeley.

[12] Ketal Patel, Brian C. Smith, and Lawrence A. Rowe, "Performance of a Software MPEG Video Decoder", Computer Science Division-EECS, University of California, Berkeley.

[13] MPEG Software Simulation Group, "MPEG-2 Encoder/Decoder, Version 1.1", June 1994.

[14] Y. Yu, and D. Anastassiou, "Software Implementation of MPEG-2 Video Encoding Using Socket Programming in LAN", Proceedings of the SPIE, Conference on Digital Video Compression on Personal Computers: Algorithms and Technologies, San Jose, CA, Feb. 6-10, 1994, Vol. 2187, pp. 229-240.

[15] "ISO Committee Draft 13818-2, Generic Coding of Moving Pictures and Associated Audio: Recommendation H.262", ISO/IEC JTC1/SC29 WG11/602, Seoul, Nov.1993.

[16] S. Eckart, and C. Fogg, "ISO/IEC MPEG-2 Software Video Codec", Proceedings of the SPIE conference on Digital Video Compression: Algorithms and Technologies, Vol. 2419, 7-10 February 1995, San Jose, California, pp. 100-109.