Available Parallelism in MPEG Processing
A Survey of Literature
David Foster <s0455368@sms.ed.ac.uk>
3 December 2004

Introduction

The Moving Picture Expert Group (MPEG) have defined a number of standards for digital video for use in many applications, such as digital television and computer movie files. Two such standards, MPEG-1 and MPEG-2, are widely adopted already and can be seen in many applications. The videos are compressed for efficiency in transmitting them, whether it be over the airwaves, over a computer network, or on a fixed medium such as a CD. While the process for decoding the videos from their compressed state is good enough, the process of encoding the videos to that compressed state is very computationally expensive. The design of the encoding algorithm contains several locations where parallelizing would greatly enhance performance in terms of time taken to encode a movie. This paper aims to survey the work that has been published regarding methods to parallelize the MPEG encoding procedure.

MPEG Background

As mentioned before, MPEG has created a series of standards for compressing video streams, such as MPEG-1 and MPEG-2. Both are very similar standards, with MPEG-2 building on MPEG-1 and offering a variety of sizes, among other features. MPEG-2 uses a far more enhanced motion prediction algorithm as well. [1] In video streams, adjacent frames tend to be very similar. MPEG compression makes use of this redundancy of data to allow videos to be compressed up to 100:1. [3]

The contents of an MPEG compressed video stream are arranged in a hierarchical fashion. At the top of this hierarchy are Groups of Pictures, also known as GOPs. Each GOP defines a sequence of individual frames, and contain timestamp information for decoding purposes. The frames themselves are of four types: I, P, B, and D-frames. D-frames are a highly specialized case, and cannot be used with other frame types, and we will ignore them. [4] I-frames are intra coded frames, also called image frames, because they consist of a compressed image in full, not dissimilar to a compressed JPEG image. They do not contain reference to any other frames, and may be used as a starting point in a stream. P-frames are predicted frames, which are predicated from another I or P-frame using a technique called motion compensation. Finally, B-frames are bidirectionally coded frames, which are predicted from the previous and the next I or P frame. The two types of predicted frames determine if smaller blocks have changed from a previous frame or will change in a future frame, and encode this information, yet the motion compensation is not perfect, and as a result, some intra blocks will occur in prediction frames. A GOP contains a pattern of these frames, starting with an I-frame or

a B-frame as a decoding start and having P and B-frames follow it, ending with another I-frame or P-frame. [1] Some GOPs may contain multiple I-frames, but those defeat the purpose of taking advantage of temporal redundancy.

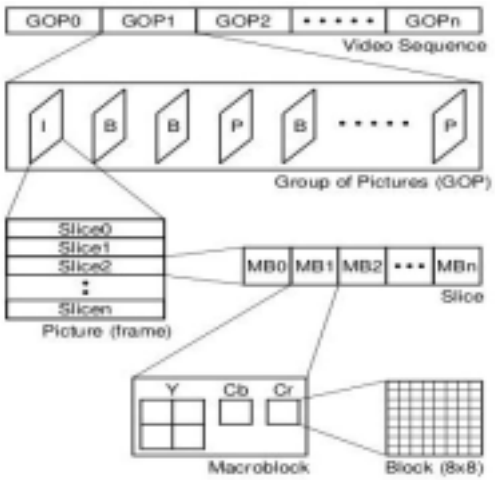

Figure 1. MPEG Heirarchy [5]

Diving further in the hierarchy, frames are composed of slices, which are groups of the next lower layer, the macroblock. In MPEG-2, these slices are entirely horizontal groups, while in MPEG-1 they can be rectangular. Macroblocks are made up of the final layer, blocks, and the both make up video information in native YUV format. [1]

Compression in MPEG encoding is based on this hierarchy. At the lowest level, the blocks, it is very common that elements will have the same color information. These are compressible elements, and are exposed by performing a two-dimensional Discrete Cosine Transformation and a few other operations, which highly compress the data at the block level. This transformation, however, is not lossy - these can be rebuilt from the compressed information. This takes care of spacial redundancy in streams. The other type of redundancy, temporal, is solved by using motion compensation. This step takes a portion of macroblock from one frame and tries to find its location in a sequential frame. If it finds it within a set of parameters, it can describe the block using a motion vector to save on space. [1]

Parallelization of Encoding Algorithms

The process of turning an video stream into an MPEG is extremely computationally expensive, due to the processing that occurs for each frame. Since most computations are highly math oriented from the result of hundreds of thousands of additions and multiplications, it follows that processing can be parallelized to take

advantage of other processors able to perform such calculations. The decision that needs to be made is on what level in the hierarchy can the process be split up on. Both the macroblock and block levels could be used, as they involve a high level of computation, but the sheer amount of communication needed between multiple processes would defeat the advantages that a parallel implementation would bring. [1]

Several implementations of parallelizing the MPEG encoding algorithm have been presented below. Some deal with parallelizing on the slice level, others on the frame level, and more on the GOP level. It is of note that all methods discussed were able to show performance increases, so long as the environments they were designed for are available. Some utilize a network based approach, while others use a multiprocessor shared-memory approach.

Some implementations of parallelizing the algorithm at the slice level have been proposed. Bozoki et. al. have proposed two algorithms, the first of which deals with having a parallel system encode slices of a frame. Their system model is based on machines separated by an ethernet network, with on master server coordinating remote encoding processes, and putting together the results of the computation of those processes. In the slice level parallelization, the number of slices in a frame is determined by the number of encoding processes. Each encoding process receives a message with a slice, which are horizontal rows only, and more data around that slice in order to be able to compress more efficiently. Parallelizing on the slice level introduces a number of issues, such as motion which occurs into different slices, which one encoding process would not know about, unless some sort of communication between the encoding processes was devised, but that would introduce unnecessary overhead and would reduce the efficiency of the algorithm. Therefore, the slice level method introduces encoding problems in the video, especially when frames have lots of vertical motion. [1]

Iwata and Olukotun also proposed a parallelization at the slice level. Their simulation model was run in a simulated OS environment on a single machine, simulating several different multiprocessor architectures. They compared the slice level parallelization against parallelization at the macroblock level, and found the slice level to be worse, due to the overhead of synchronization required for a macroblock implementation being less than the synchronization required for a slice implementation. [5] This is due to the data dependencies highlighted with Bozoki et. al.'s implementation. In fact, Iwata and Olukotun's findings go against the assumption made with Bozoki et. al., but the different simulation environments they used explain the discrepancy. Communication between processes running on different machines separated by ethernet is much slower than a single machine using a shared memory

architecture which would be able to take advantage of the macroblock encoding parallelization.

Many researchers who implemented slice level parallelization also looked at a GOP level parallelization. Several implementations exist, using different system setups, all with advantages and disadvantages. Moore et.al. implemented a GOP level implementation using the scripting language TCL and some extensions, one for handling the video data and one for handling the network communication. They used a master/slave style configuration over a standard ethernet network setup. In this implementation, both the master and slaves all read the raw data to encode from an NFS mounted filesystem. The master process first communicates with the slave processes, instructing them which sequence of frames to encode, helped by the use of a cut detection mechanism to try to improve efficiency and to compress optimally. The master server then polls the state of the each slave to determine if it is ready to receive a new workload. Encoded sequences are stored on the NFS drive using a file naming scheme which includes information about the ordering of the sequences, where it is then reassembled by the master process. The system also exhibits fault tolerance, in that all work done up to a point of failure is kept and may be resumed. [2]

This method of parallelization is not without its faults. One such problem is one that is quite obvious in the communication model, in that the master process has to poll each slave process for its status, instead of the slave process announcing it has completed its work. This results in a lot of time being wasted in the slave processes that are done but waiting for the master to contact them. Moore et. al. acknowledge this in their paper. Another problem which they fail to mention is the fact that the encoding slaves seem to produce fully standalone MPEG files, in which headers and footers need to be stripped off in order to be combined. In fact, they mention that this is a major bottleneck. The obvious solution to this is to not have the slave processes produce a full MPEG file, however since they are using an external library they may not have control over this. Yet one more problem they acknowledge is the problems with disk I/O contention, in that all processes must read from the source, yet the disk must physically move the read head in order to serve these requests, resulting in time waiting for these I/O requests to be served. A solution could be to utilize communication to transfer the data from the master to the slave processes, which would also make it so the slaves could be anywhere. However, this introduces a problem of communication load and bandwidth which would have to be addressed. [2]

Wang also implemented a very similar style system, in which masters and slaves are used to communicate. The architecture used here is an 18 processor single machine, which allow the masters and slaves to communicate in a speedier manner. Sockets are used for communication instead of a shared memory implementation. Both sets of processes actually send their data to each other through messages instead of all reading from a disk. When the slave processes finish their work, they send their data and a done message to the master process, which then sends the data to a combination process. [3]

Wang was able to find a decent speed up using this process, which seemed to be capped as the number of slave processes went above 5. He felt that was due to his master process not being a concurrent server, and it blocked until one request was completely served. He has detailed preliminary work of making the master process concurrent, and thus has achieved significant speedup when the number of slave processes goes above 4. However, his implementation is not perfect. He notes that a scheme should be devised in which the slave processes communicate directly with the combination process, in order to save on message passing of large amounts of data. [3]

One problem in Wang's work is that it is not a formal paper, and is missing out on some details, such as how his system is accessing files and obtaining data, which could result in some performance implications.

Bozoki et. al. also describe a GOP level algorithm which uses an architecture similar to the ones above, with slave and master processes, with the master doing the combination and output. Like Wang's algorithm, the processes communicate their data through messages, however it is designed to work on a network infrastructure. The slaves do not send their encoded frames back however, the encoded frames are stored on either a common NFS drive or on their local filesystems, to be reassembled later at the request of the user. The master does a bit of processing of the raw data as well, as they expect the input to be from a hardware input digitizer, which provides YUV data in 4:2:2, and MPEG-1 expects the data to be in 4:2:0. The master assigns the slaves GOPs to work on via the network, and before the slaves encode the last frame, they request to the server for a new one. Using this method, they were able to achieve a nearly linear improvement in speed as the number of slave processes increased. [1]

As discussed above, this algorithm exhibits some processing in the master process in order to convert the YUV from 4:2:2 to 4:2:0. There is no mention in the paper if it was ever looked at to simply send the data to each slave and have the slaves perform the conversion prior to encoding. The counter argument to this is the increased

traffic in the messages to each slave. While the trade off may be a marginal gain or loss, it would've been nice to see investigation in this area in order to facilitate the master server being able to handle requests faster.

One problem they noted in their paper was the high acquisition time of the frames. Since they were on a NFS mounted drive, they had to be transferred over a local network, and at the time of their experiment they couldn't achieve faster bandwidth than 400 kilobytes a second. They also said that having the data on local hard disks tot he encoding processes wouldn't help that much since hard disks of that time were capped at around 1 megabyte a second. [1] This added some upper limits to the performance of their algorithms, because it was not entirely dependent on how fast the slaves encoded, it was capped by how fast the master could read the frames, which was around 2 frames per second. Nowadays, hard disk and network performance have increased much more, and it would be nice to see the results of this algorithm on more modern equipment in order to determine an upper bound.

Barbosa et. al. provide an algorithm which is based on a shared memory implementation. Processes communicate through primary memory and are implemented using a thread model. The tasks are broken up into a set of tasks on two levels. The upper level is the GOP level, which coordinates the breakup of frames, and a frame level, which encodes the frames. The GOP level is manages by a coordinating task, which also communicates with a writing task. Every task has access to read and write buffers, which means that dependent frames are able to look at other frames that are not necessarily done by that encoding process. [4]

Barbosa et. al. implemented two forms of syncronization within the encoding processes. The first one deals with implementing a fork/join approach, in which the GOP task spawns a number of encoding processes and assigns them frames to encode. They found that this method was not optimal because when one encoding process finished its work, it had to wait until all other encoding processes in that GOP group finished before the GOP task would give it more work to do. Instead, they took advantage of the shared memory setup and defined a method in which encoding processes were allowed access to any frame that needed to be encoded, which they called the bag of tasks method. This ensured that encoding processes were never kept idle, and since the processes had access to other frames that they needed to encode the P and B-frames, they were able to accomplish their task. This is not without its faults however, as tasks taking responsibility for a new P or B-frame would have to wait to make sure the decoding was done for the previous frame, which can result in time that those processes are waiting. Due to this communication overhead, as the number of threads increased past a certain

saturation point, the efficiency of the method went down. However, if the number of threads are capped at around 16, the noticed a significant speedup, far better than that of the fork/join version. [4]

Future Work

One area that could be improved upon in many of the algorithms presented is their use of cut detection mechanisms. Only one paper reviewed [2] even mentions it, and they do use it, but they are not convinced it is worth it. A cut detection mechanism works by finding a worthwhile breakpoint in the frames in order to divide GOPs up. Many of the algorithms discussed above use a fixed number of frames, which all start with an I-frame. This means that two frames, although incredibly similar, could be divided into different GOPs and thus would result in a full I-frame being encoded where a P or B-frame would have saved on a lot of space. The obvious advantage to this method is optimizing for size, as it would minimize even more redundant information in the stream. However, its not without its faults. It would involve an extra layer of processing that would have to be done prior to the actual encoding of the stream. It would also require extra work to ensure processes capable of more processing would encoding larger GOPs to ensure optimal load balancing, which if done right would be a large benefit. Since the goal of many of the algorithms is to improve the overall speed of encoding rather than the size efficiency, it is generally not discussed. However, the ability to optimize for size or speed can already be seen in other areas, such as the GNU C compiler, as a user defined option.

It is also worth looking into the possibility of moving MPEG based algorithms to the current evolution of modern graphics hardware. The programmable nature of these architectures brings them closer to a vector or stream processor, which are well suited or media applications because they exhibit high levels of parallelism, as well as other features. [6] Advances in tools like the high level general purpose language Cg for graphics hardware allows rapid development of algorithms, which are portable to most major implementations of graphics hardware.

Bibliography

[1] S. Bozoki, S.J.P. Westen, R.L. Langendijk, J. Biemond. (1996) **Parallel Algorithms for MPEG Video Compression with PVM.** EUROSIM.

[2] Jeffrey Moore, William Lee, Scott Dawson. **Optimal Parallel MPEG Encoding.** Department of Computer Science, Cornell University.

[3] F. Wang. **Parallelization of Software MPEG Compression.** http://www.evl.uic.edu/fwang/mpeg.html

[4] D.M. Barbosa, J.P. Kitajima, W. Meira Jr. (1999) **Parallelizing MPEG Video Encoding using Multiprocessors.** Proceedings of the XII Brazilian Symposium on Computer Graphics and Image Processing, pp. 215-222.

[5] E. Iwata, K. Olukotun. **Exploiting Coarse-Grain Parallelism in the MPEG-2 Algorithm.** Technical Report CSL-TR-98-771, Stanford University.

[6] K. Hillesland, S. Molinov, R. Grzeszczuk. (2003) **Nonlinear Optimization Framework for Image-Based Modeling on Programmable Graphics Hardware.** SIGGRAPH.