Optimal Parallel MPEG Encoding

Jeffrey Moore, William Lee, Scott Dawson
Department of Computer Science
4130 Upson Hall
Cornell University
Ithaca, NY 14853-7501 USA
{jmm, wwlee, spdawson}@cs.cornell.edu

ABSTRACT

Moving Picture Expert Group (MPEG) encoding is a computationally expensive operation. This paper outlines a method of compressing video streams from within a Resolution Independent Video Language (RIVL) using the MPEG standard of compression in parallel. Two processes, a *client* and a *server*, work in tandem to parallelize the MPEG encoding process. The client dispatches jobs to each of the available servers for processing and the servers handle the compression of the video streams. The goal of Optimal Parallel MPEG Encoding is to derive a scalable, robust protocol to efficiently handle this computation.

1. INTRODUCTION

We implemented an algorithm which encodes MPEG[8] videos using a network of workstations on a shared file system. To accomplish this, the video to be encoded is divided at logical breaks in the video stream. These breaks are determined either by use of an existing cut detector or an automatic step size calculation. The major advantage of this implementation is increased processing speed. Furthermore, incorporation of cut detection may yield improved compression and output quality. Our choice of existing languages and networking protocols is also notable, providing us with assurance that our implementation will be as portable as they are.

2. BACKGROUND

2.1 RIVL

RIVL[5] is a resolution independent video language which is an extension to the Tcl[4] scripting language. The most attractive feature of RIVL is its treatment of images, audio, and video as first class data objects. This makes editing multimedia data easier. Another attractive trait of RIVL is its lazy evaluation strategy. Processing is done within RIVL only when absolutely necessary. For example, if a user reverses, crops, and alters the length of a video sequence, those operations are not carried out until the user wishes to either view the data or write it to a persistent storage device. To accomplish this lazy strategy, RIVL creates a "graph", the starting node of which is the original sequence. The graph progressively increases in size and complexity as more operations are applied to the sequence. In the given example, the reverse, crop, and length operations are made as the graph is traced back to the sequence. The important point to extract from this is that accessing the original sequence within the RIVL environment can be quite complex. The details of the "graph" are abstracted from the user, which is important to note in the explanation of our client-server communication algorithm.

2.2 Distributed Processing

Distributed Processing[6] (DP) is a networking extension to the Tcl scripting language. It provides the networking facilities used in the project. This extension has been exhaustively tested and debugged. Also, the extension is very portable, therefore working on any platform on which the Tcl scripting language works. DP also supports IP multi-cast, a relatively new capacity of internet routing. Each of these reasons makes DP a prime candidate for the networking portion of this project.

2.3 MPEG Encoding

Our implementation uses the public domain UC Berkeley MPEG encoder, which has been built into RIVL. Our parallel implementation temporally divides the video for processing, making it unnecessary to modify the existing encoder.

3. IMPLEMENTATION

The architecture is comprised of a client process and server processes, as illustrated in Figure 1. The client process is the controlling entity, while the server processes are those that carry out the actual encoding.

3.1 Client Design

The client module is the process that executes on the client machine, serving as a controller/dispatcher for the duration of the encoding process. Following is an overview of our infrastructure.

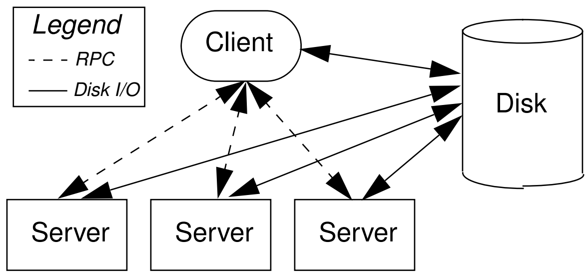

Figure 1: Encoder Architecture

3.1.1 Reading the Build Hosts File

The client accesses a file in the user's home directory entitled `.bld.host`. This contains a list of machines that the user desires to utilize for processing, followed by an optional port number that establishes a channel to that machine. If no port number is specified, the channel defaults to 1800. Once the build hosts are determined, the client-server relationships can be established.

3.1.2 Registering Client to Servers

This involves establishing communication links between the client and the servers as outlined in the `.bld.host` file. The communication links are established by use of DP commands. This step also initializes a table of server states that is dynamically updated during the encoding process.

3.1.3 Dividing the Sequence

In order to make the algorithm optimal, logical sequence division needed to take into account factors such as varied sequence length and the potential for server failure. Currently, there are two methods being utilized to divide sequences; (1) cut-detection and (2) a step-size calculation. The cut detector that we incorporated into our client code is the detector that will be included in the next distribution of RIVL. Using the cut detector yields better compression and enhanced quality, due to the fact that divisions are made in the pre-encoded sequence where a logical division occurs. However, the current speed at which the cut detector operates makes its use in the sequence division stage questionable for relatively small sequences. For now, the decision to use the scene detector is solely user-based. The step-size calculation yields a discrete value that is used when dividing a sequence. A stepsize that is too small is impractical, due to the amount of time that it takes to reassemble the sub-sequences after encoding. For example, when dealing with a 2 second video stream, piecing together the sub-images from 20 machines (where each machine receives 0.1 seconds) will take much longer than the total processing time on one machine. This is due to the stripping of headers and trailers from subsequences when creating the final MPEG. If the step-size is too large, the parallel performance will be inhibited by the speed of the last server that is processing. The logical stepsize choice is one that will enable faster processors to process more video than slower processors, without providing a slower server with an amount of data that may impose a bottleneck.

3.1.4 Creating Local File Patterns

Since we are assuming a shared file system for our implementation, we take advantage of the fact that we will be able to access that file system to retrieve all encoded data after processing. We create a consistent file pattern that each server will utilize to write their encoded data to disk. The file pattern is a combination of the subsequence number, start time, and end time of the encoded scene.

3.1.5 Encoding Scenes

The client algorithm is as follows:

- If the current server is "dead", or responds with "busy", repeat this step with the next server.
- If the server responds with "idle", it can receive encoding jobs. Initiating the encoding process is accomplished by using a DP remote procedure call (RPC in Figure 1).

To dispatch a job to a server, a RIVL script is sent that contains instructions on how to access the original video data. The script also contains information on how to recreate the desired video sequence.

3.1.6 Conclude Processing

At this point, we access the directory in the shared file system where the servers' output was written. Using the local file pattern convention previously outlined, we concatenate all of the MPEG files in that directory in sequential order, producing a final MPEG. This step is comparatively quite fast, because the only overhead of concatenation is the stripping of headers and footers from the smaller MPEGs, and joining the remaining data together into one MPEG file.

3.2 Server Design

The server uses the Berkeley MPEG encoder to compress a video sequence. When a job has been receive by the server, its status is set to "busy" until either processing has concluded or the client-server connection has been terminated. The server code is rather trivial, as all code related to processing is obtained from the client when a job is received. All other code in the server module relates to state management. Ideally, the server should be implemented as a UNIX daemon.

4. EVALUATION

4.1 Fault Tolerance

In today's heterogeneous computing environment, fault tolerance is becoming more important. Computer equipment has not reached a point where it can continuously run without encountering noticeable failures. In order to make our application robust, we've added fault tolerant code to both the client and the server.

4.1.1 Client Fault Detection

The client maintains knowledge about each server's state. It polls a server to determine its status in a round robin fashion. In the event that a response is not received from a queried server, the server is assumed to have crashed. The client will then reassign the lost sequence to another available server. In the event that all servers fault during processing, our implementation will salvage any completed data that was left in the designated temporary directory. The advantages of this become apparent when processing long video sequences. If, after an hour of processing in parallel, there is no chance of recovery, it is better to have some data to view than no data at all.

4.1.2 Server Fault Detection

The server can detect client crashes, an event in which the server will complete its assigned work-load and reset its internal state to "idle". This is an extremely vital component to the robustness of our program. Without this, a crashed client could leave some servers in a busy state.

4.3 Parallel Performance

Theoretically, this solution is capable of exhibiting a linear increase in performance over a sequential solution. Currently, our solution does not exhibit this linear performance gain. As is usually the case with programming distributed applications, there are certain interaction protocols that can be optimized on both the client and server to increase performance. Unfortunately, in such a distributed application, there is a certain amount of overhead both before, during, and after processing, that cannot be extracted from the system. We've found that the bottleneck for our system specifically relates to MPEG read/write overhead and event queue management.

4.3.1 System Diagnostics

Through statistical load analysis, we have determined the performance bottlenecks in our system. The wasted processing time, W_i in Figure 2, is the largest performance inhibitor, often wasting 50% of the server's time. From analysis of the subsystems that comprise our solution, we can isolate potential locations for improvement and speculate about the performance that will result. Referring to the equations eq1 and eq2, K_i and W_i represent the primary bottlenecks in the system. K_i is the overhead in reading and writing video data, while W_i is the communication overhead involved in scheduling video. Using these two measurements, we derive a utilization value for a given server (eq3). If we minimized K_i and W_i, the speedup would approach N. This would be ideal.

We set up an experiment using 30 servers (N), with varying configurations and processor clock speeds (Table 1), to encode a one minute long, 160 by 120 MPEG video. In order to maximize the use of machines, we split the MPEG into 120 subsequences (stepsize of 0.5 seconds). Using this division of work, faster machines would be able to process more subsequences than slower machines. We found that the processing time per second of data is 1.5 times slower for the parallel implementation than the serial implementation. This disparity between the two implementations is justified by K_i.

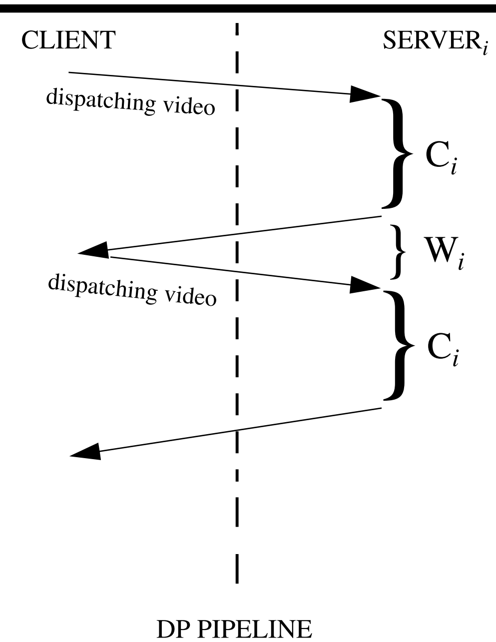

Figure 2 - Client-Server Interaction

$$Speedup = \frac{ST}{Max(C_i + W_i)} \quad \text{(eq1)}$$

$$C_i = \left(\frac{ST}{N}\right) + K_i \quad \text{(eq2)}$$

$$U_i = \frac{C_i}{C_i + W_i} \quad \text{(eq3)}$$

ST = Server Processing Time (serial)
N = Number of Servers
$i = 1...N$
K_i = MPEG Read/Write Overhead
W_i = Communication Overhead
C_i = Server Processing Time (parallel)
U_i = Utilization

Workstation	Speed (MHz)	Count
HP735	99	3
Sun Sparc20	60	5
Sun Sparc10	50	5
Sun Sparc5	70	3
Sun Sparc2	40	4
Sun ELC	33	7
Sun IPX	40	3

Table 1 - Server Selections

Testing demonstrates that faster servers exhibit only 50% MPEG processing utilization (U_i). This is because they are spending a great deal of time awaiting client processing requests (W_i). The slower servers are highly utilized (90% on average) because they are computing data most of the time. In short, these slower servers cannot process quickly enough to warrant receiving more work.

With relatively large W_i and K_i, our overall improvement for 30 machines is only 8-fold. According to our experimental data, two things must occur to increase performance: (1) eliminate W_i and (2) lower K_i. Eliminating W_i for 30 machines will result in a 20-fold improvement. The remaining 10-fold is achievable by reducing K_i. Intelligent reduction of these bottlenecks is the next step in our research.

4.3.2 Disk I/O Contention

As the number of servers being used for compression increases, so does the number of the servers concurrently requesting data from the shared file system. As the number of simultaneous requests increase, I/O becomes a bottleneck. This occurs because the data is only stored on one disk, thus having only one read head. This single head can only process a certain amount of data at a time, therefore making other processes await data retrieval. However, this drawback exists in most shared system architectures (i.e. memory, file, etc.) regardless of the application.

5. RELATED WORK

5.1 Faster Than Real-Time MPEG Encoding

In a joint effort between U.C. Berkeley and Purdue University, the Intel Touchstone Delta and Intel Paragon achieved faster than real time MPEG compression[1]. The encoder that made this possible is the U.C. Berkeley MPEG encoder with extensive modifications. Achieving faster than real time performance on a massively parallel machine requires major source code modifications to increase scalability across processors. Source code changes of this magnitude greatly reduce the portability of the encoder, thus making it a special purpose encoder. Also, the supercomputers used for this implementation are prohibitively expensive, making the purchase of one difficult to justify. Alternatively, our encoder may use inexpensive computers to perform compression at an increased rate.

5.2 Comparison

Our implementation of parallel MPEG encoding compressor is much more portable than the above implementation. Because we use RIVL for the video encoding process, our encoder works anywhere that RIVL works. Many ports of RIVL have already taken place. The Berkeley/Purdue encoder, on the other hand, may require a complete source code rework.

6. EXTENSIONS

6.1 Increasing Performance

We've discussed the reasons for the degraded performance of our system. Currently, the client polls all servers in a quasi round-robin style. Modification of our algorithm may decrease the communication overhead (W_i). As an improvement, the client can assume that a server is active until otherwise notified by way of either a closed connection or server notification. This may also lower K_i, due to the fact that a server will not need to constantly respond to client queries about its status.

6.2 IP Multi-cast

In order to obtain an up to date list of all servers that are running, we have decided to incorporate IP multi-cast. Using IP multi-cast will remove the reliance of our implementation on the build host file.

6.3 Binary Data Transfer

Binary data transfer is a feature scheduled to be included in DP in the near future. This will remove the shared file-system reliance on our current implementation. A client then can use any machine on the internet, running our server, to compress videos.

6.4 Improved Step-Size Calculation

Step size calculation could be dynamically recalculated during the course of video compression. The calculations could be based on the amount of time taken to complete video compression on one subsequence.

6.5 Load Balancing

Assigning larger subsequences to servers that complete processing in a short amount of time is a fairly accurate measure of load balancing. This may replace the automatic step size calculation.

7. Conclusion

Our parallel implementation exhibits an eight-fold increase in speed over serial implementations. An optimal implementation dramatically reduces the amount of time required to compress video without using super-computers. Also, our parallel encoder is no more difficult to use than its serial counter-part.

Acknowledgments

This research was conducted for the Multimedia Systems course and a Masters research project at Cornell University. The authors are grateful to Jonathan Swartz and Brian Smith for their assistance throughout the project.

References

[1] K. Shen, L. A. Rowe, E. J. Delp, *A Parallel Implementation of an MPEG1 Encoder: Faster Than Real-Time!,* Proceedings of the SPIE - The International Society for Optical Engineering, vol.2419, pp. 407-418.

[2] G. W. Cook, E. J. Delp, *An Investigation of JPEG Image and Video Compression Using Parallel Processing,* 1994 IEEE International Conference on Acoustics, Speech and Signal Processing, p. 6 vol. 3382, V/437-40 vol.5.

[3] K. Shen, G. W. Cook, L. H. Jamicson, E. J. Delp, *An Overview of Parallel Processing Approaches to Image and Video*, Proceedings of the SPIE Conference on Image and Video Compression, vol. 2186, February 1994, San Jose, California, pp. 197-208.

[4] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Professional Computing Series.

[5] J. Swartz, B.C. Smith, *A Resolution Independent Video Language*, ACM Multimedia '95 Proceedings, pp. 179-188.

[6] B. C. Smith, *Tcl-DP Information,* Personal Communication.

[7] K. L. Gong, *Parallel MPEG-1 Video Encoding*, MS Thesis, Department of EECS, University of California at Berkeley, May 1994, Issued as Technical Report 811.

[8] D. Le Gall, *"MPEG: A video compression standard for multimedia applications"*, Communication of the ACM, vol. 34, no. 4, pp. 46-58, April 1991.