# Optimal Parallel MPEG Encoding

Jeffrey Moore, William Lee, Scott Dawson, and Brian Smith

Department of Computer Science

4130 Upson Hall

Cornell University

Ithaca, NY 14853-7501 USA

{jmm, wwlee, spdawson, bsmith}@cs.cornell.edu

---

## Table of Contents



---

## Abstract

The Tcl/Tk extension, Tcl/Rivl, provides a suite of commands to manipulate audio and video data. Compressing long sequences of MPEG video requires a significant amount of computation power. This paper outlines a parallel algorithm that achieves real-time MPEG compression without specialized hardware.

# 1. Introduction

Because it can take 15 seconds to compress 1 second of MPEG video on a Sparc 20, using increasingly more affordable computers is a must. As shown in the 'seq_write_parallel' implementation, it is possible to use a network of workstations to greatly improve the effective rate of MPEG compression. Machines on the network compress temporally divided video to achieve parallelism, as illustrated in Figure 1.

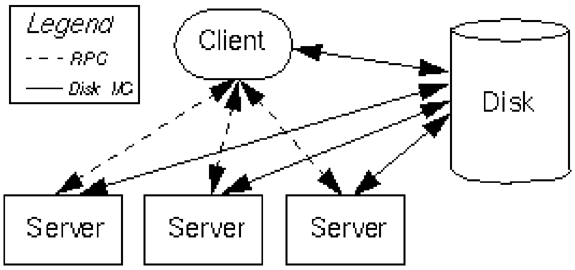



Figure 1 - General Overall System Architecture

# 2. Background

To best understand how everything works, it is good to have a basic understanding of Tcl/Rivl, Tcl/Dp, and MPEG Encode.

## 2.1 Tcl/Rivl

Tcl/Rivl, an extension to Tcl/Tk, treats images, audio, and video as first class data objects, making it ideal for multimedia data processing. Tcl/Rivl employs lazy evaluation, therefore it only manipulates data when necessary and only performs the operations visible in the final output. For example, if a user reverses, crops, and alters the length of a video sequence, the operations are not carried out until the user views or writes the data.

## 2.2 Tcl/DP

The Tcl/Dp extension to Tcl/Tk provides the networking functions to distribute work requests. Its implementation of remote procedure calls makes it ideal for transmitting work requests in the system.

## 2.3 MPEG Encode

Because Rivl incorporated the public domain University of California, Berkeley MPEG encoder, we used it.

# 3. Implementation

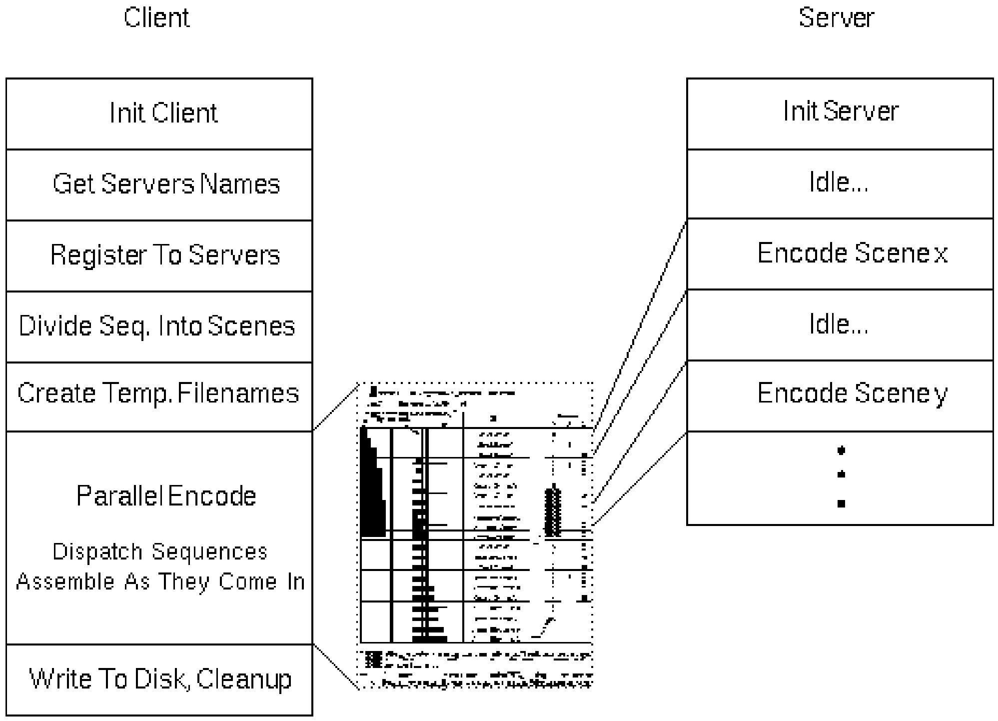



Figure 2 - Detailed Overall System Architecture

Figure 2 depicts the setup of the system. Each system receives a thorough discussion in the following sections.

## 3.1 Server Design

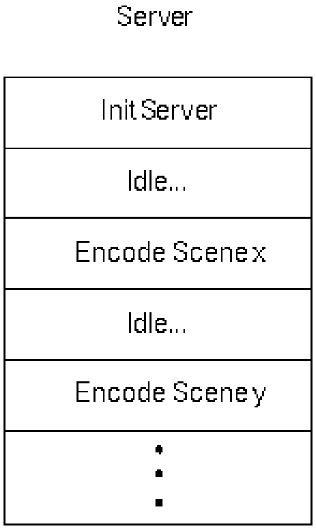



Figure 3 - Server Architecture

This is a concurrent server because it is able to accept and process multiple jobs at the same time, but issuing multiple simultaneous encoding requests decreases encoder performance linearly.

### 3.1.1 Initialization

The server is very simple. During initialization, the server creates a listening port to accept RPC connections. This mode of operation represents a security hole. For example, it is possible to send a malicious script like 'rm -rf /' to the server and execute it using the permissions of the user who started the server process.

## 3.2 Client Design

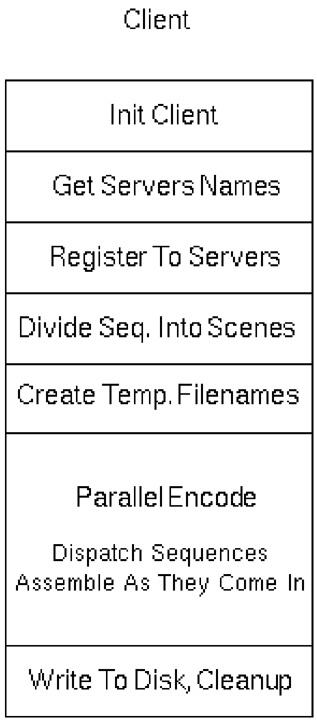



Figure 4 - Client Architecture

### 3.2.1 User Options

A variety of user options allows the user to customize the way that the client handles input and screen output.

#### 3.2.1.1 Scene Cut Detector

This option is on by default. The user can disable this command by using '-noCutDetect' on the 'seq_write_parallel' command line.

Performing cut detection on a video sequence before compression yields smaller, better quality output.

Output size improves because new scenes always begin with P frames. New scenes normally represent a change in backgrounds and actors, creating many differences between frames. This translates to a large number of error terms and propagated error through the video stream to the next P frame. If the first frame of every scene is an I frame, the MPEG compressor does not calculate motion vectors and error terms for the frame. This makes the resulting video sequence smaller and of better quality.

#### 3.2.1.2 Quiet

This option is off by default and is enabled by placing ’-quiet’ on the command line. The quiet option suppresses timing and progress information related to the encoding process, except error messages.

#### 3.2.1.3 Step Size

Step size is the amount, in seconds, of video sent to each machine for compression. In this command’s absence, ’seq_write_parallel’ calculates a value to evenly divide the sequence among all machines. To specify a step size, ’-stepSize <step size>’ must appear on the command line where <step size> represents the step size, in seconds. If all machines are of similar speed, automatic calculation works best.

### 3.2.2 Initialization

In the event driven Rivl environment, it is essential for the client to quickly access its state variables, therefore, all state variables are globally accessable. The client initializes these variables upon start-up in addition to creating a temporary directory for shared files.

### 3.2.3 Reading the Build Host File

To obtain a list of available compression servers, the client accesses the ".bld.host" file in the user’s home directory that contains a list of server names, performance benchmark, and an optional port number. The server name is a TCP/IP style machine name, the performance benchmark is the length of time the machine takes to encode one second of MPEG video, and the port number represents the TCP/IP port number the compression server listens to for work requests. In the absence of a port number, 1800 is assumed.

### 3.2.4 Registering Clients to Servers

This step establishes communication links between the client and the servers using Tcl/Dp.

### 3.2.5 Dividing the Sequence

There are two different methods of dividing video sequences. A command line argument determines which method to use. The first method is cut detection; it divides a video along scene boundaries. This method is slow and should only be used when output quality, and to a lesser extent, size, are the main concerns. The second method uses a step size, in seconds, to divide the video. The user may specify a step size on the command line, but they risk degrading performance in the system by making the step size too large or too small. If the step size is too small, system performance suffers because of network contention and if the step size is too large, the work load does not balance properly and the user waits for a slow machine to finish compressing a large video sequence. In most circumstances, an automatic step size produces a well proportioned number. This receives further discussion later in the paper.

### 3.2.6 Creating Local File Patterns

Because Tcl/Dp cannot send binary data over a network, we used a shared file system. Each machine has easy access to all necessary files, but this technique has its share of problems as outlined later in the paper.

### 3.2.7 Parallel Encoding Scenes

#### 3.2.7.1 Overview

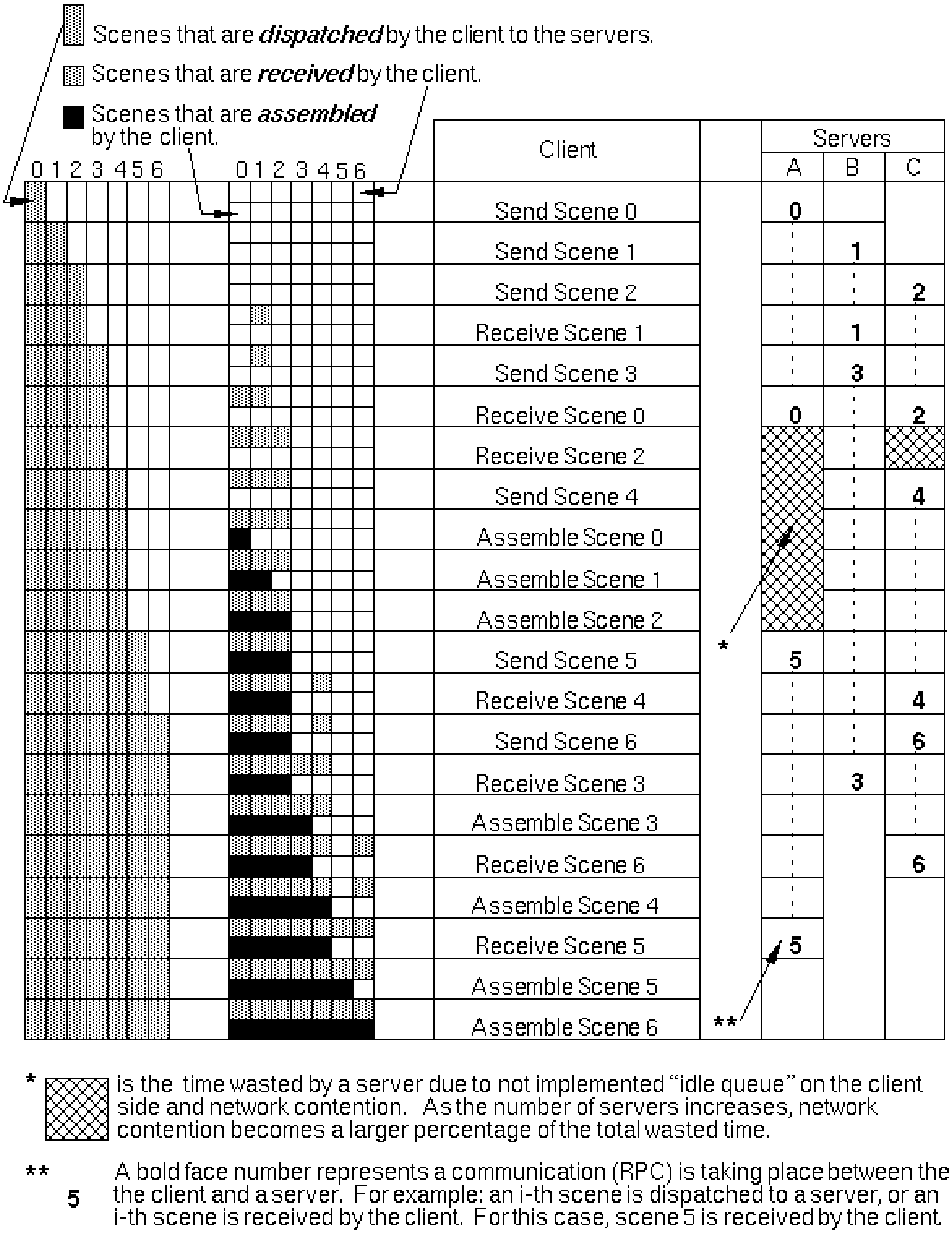



Figure 5 - Parallel Encode Process

The parallel encoder is the core of the program. Its job is to efficiently and reliably dispatch scenes to servers and assemble them as they return. A priority scheme for the three major tasks contributes to the efficiency of the system. The order of precedence is:

1. Receive scenes

2. Send scenes
3. Assemble scenes

The reliability of the system rests on the client. When the client detects a faulty server, its works gets redistributed to the next available, well server.

#### 3.2.7.2 Sequence Sending

Using the connections established during client registration, the client sends a Rivl script to a server using an asynchronous RPC call, then starts a timer. The client repeats this for each available server. Because receiving a scene has a higher priority than sending, the client checks the event queue for newly received scenes after each send. The client uses a round-robin algorithm to determine which server receives the next video segment. The server order is the same order as the servers listed in the Build Host file, described earlier.

##### 3.2.7.2.1 The RIVL Script

The Rivl script sent to the server contains a wealth of information. It tells the server where to find the necessary files for compression, which part of the scene to encode, where to put the results, and instructions on how to notify the client when it finishes. So, from the perspective of the client, it sends a command to the server to compress a sequence and the server returns the results in a temporary file.

##### 3.2.7.2.2 The Timer

When a server's timer expires, the client knows that the server is faulty and its job must be given to a healthy server. The time out value is the length of time required to compress that amount of video, as determined from the benchmark value in the Build Host File, multiplied by three. The calculation of this time-out value receives more attention later.

#### 3.2.7.3 Sequence Receipt

The client must receive scenes from servers as soon as encoding finishes, otherwise the server sits idle and underutilized. When a scene finishes compression, the server notifies the client via a RPC call that sets a flag in the client used by the assembler and job dispatcher. To fully utilize servers, the client proactively checks the network queue for finished jobs because Rivl is a single-threaded, event driven environment. To ensure that scene receiving has the highest priority, the scene assembler checks the queue often. When the client receives an encoded scene, the flag set by the server notifies the scene assembler of the new scene. It is possible for multiple RPC messages to arrive at the client event queue at the same time. Figure 5 demonstrates this phenomenon where server A finishes scene 0 at the same time server C finishes scene 2, so server C sits idle until the client finishes processing the RPC from server A. This demonstrates the importance of receiving messages in a timely fashion.

#### 3.2.7.4 Sequence Assembly

As scenes return, the scene assembler begins putting together the final video sequence. The scene assembler pieces the sequence back to together, starting at the beginning of the sequence, until it encounters a bubble or it must dispatch another scene. A bubble is a missing scene within a list of received scenes. For example, if scenes 1, 2, and 4 returned from a server, the bubble lies between 2 and

4. Because scene assembly is the lowest priority task, it often checks the event queue for higher priority tasks. For example, in figure 5, the client can continuously assemble scenes through number 4, but server C completes scene 6 and returns it to the client. This causes the client to interrupt the assembler, retrieve scene 6, and continue assembling scenes.

The scene assembler uses a left to right algorithm to concatenate scenes. This means the algorithm begins at the left, scene n, and assembles video segments to its right, scene n + 1. When the algorithm begins, n is equal to zero. This is a simple and very fast. We considered more complex schemes to avoid bubbles, but the overhead associated with more complex methods outweighed the gains.

Sending a scene to a server preempts scene assembly to maximize machine utilization, therefore when a client receives notification that server X finished a scene, the client sends a new scene to server X. For example, in figure 5, as soon as the client receives scene 2 from server C, it sends scene 4 to server C even though it could perform sequence assembly. The flag in the client guarantees this, but can cause performance degredation as discussed later.

**3.2.7.5 Fault Detection**

The client uses two fault detection mechanisms: detecting broken connections and time-out. Although the result of a time-out is a broken connection, it is more efficient for the client to deal with faulty server and reassign its work as soon as possible. If, by some chance, all servers crash, the client outputs as much of the final output as possible and terminates. Nothing gets salvaged from a client crash because it is not replicated.

**3.2.7.5.1 Broken Connection**

There is one way to detect a dead server; when the connection between the client and server breaks. The broken connection causes an exception in Tcl/Dp, calls a predetermined cleanup handler, and terminates. When the server terminates, the client dispatches the dead server's work to an idle server and removes it from the list of eligible servers.

**3.2.7.5.2 Time Out**

Sometimes in Rivl, a server not capable of encoding a scene can still accept RPC work requests. The time-out mechanism tolerates these failures. The timer for the current machine begins when the client sends a work request to the server and ticks until the client receives the completed scene or the timer expires. If the timer expired, the client closes the connection such that a Tcl/Dp exception gets generated. This is the same exception handler that handles a broken connection.

A time-out occurs when the encoder runs out of temporary disk space while encoding a large sequence. In this case, the server runs fine, but the encoder will not run. Because the server spawns a new process for the encoder, it has no way of knowing the encoder is not working properly, thus the server times-out, the client closes the connection, and its work is redistributed.

The length of the time-out period for each server is based on the predicted rate stored in the build host file. The predicted rate is the length of time, in seconds, the machine requires to compress 1 second of MPEG video. The empirical formula used to calculate time-out values is three times the predicted value. For example, if the predicted value is 16 seconds, the time-out period for 0.5 seconds of video is 24 seconds as shown in Equation 1.

$$\overbrace{\frac{16\ seconds}{MPEG\ second}}^{\text{Given Predicted Rate}} \times \underbrace{\frac{0.5\ MPEG\ second}{scene}}_{\text{Size of a Scene}} \times 3 = \overbrace{\frac{24\ seconds}{scene}}^{\text{Time-out Period}}$$

Equation 1 - Time-out Equation

The user disables the time-out mechanism by using zero as the predicted value.

### 3.2.8 Conclude Processing

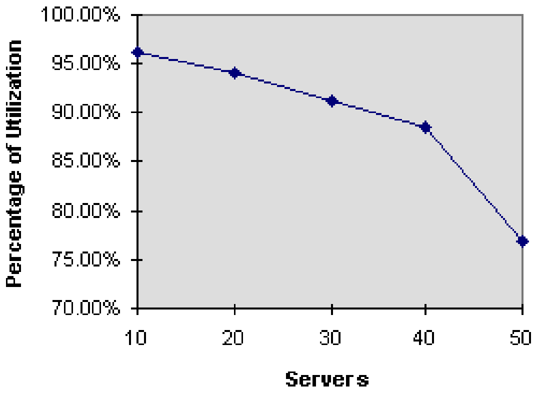
After sending, receiving, and reassembling all scenes, the client concludes execution by closing all Tcl/Dp connections, writing the output to disk, and removing temporary files. Closing all Tcl/Dp connections and writing output to disk is a fast process. Temporary file removal, on the other hand, requires a significant amount of time, therefore we spawn a background process for this task.

# 4. Evaluation

We ran two sets of experiments with 51 HP-725 workstations representing 1 client and 50 servers. The first experiment was to encode a one minute, 160x120 MPEG video without parallelism. These results calibrated our speedup comparison. The second experiment used the parallel encoder with 120, 0.5 second video segments. We tested beginning with 10, 20, 30, 40, and 50 machines.

In the single machine experiment, each machine took 1,030.3 seconds, with a standard deviation of 57.08 seconds, to encode the sequence. The speedup using the parallel encoder increased fairly linearly, peaking at a 19 times speedup.

### 4.1 Server Utilization / Network Contention

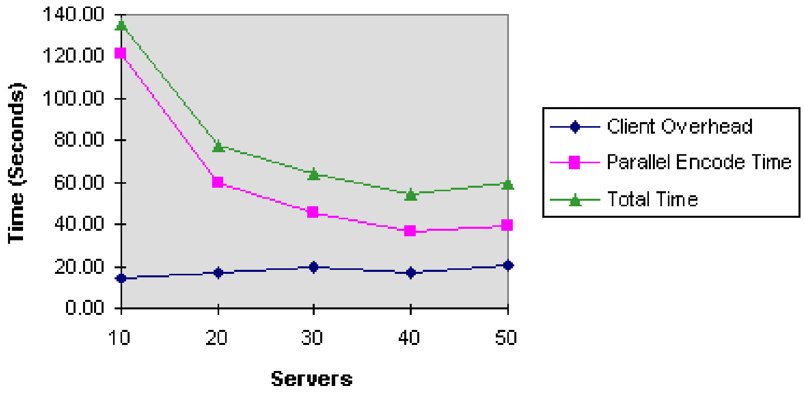



Graph 1 - Server Utilization

Server utilization is the percentage of time a server spends encoding video versus sitting idle. Idle time may be caused by network contention or slow client response to dispatching more work. Figure 5 shows an example of this with the wasted time shown as the cross hatch.

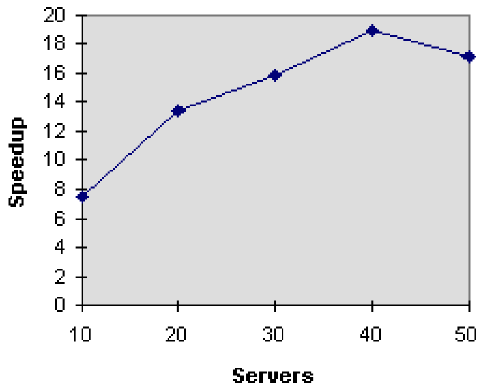
Graph 1 shows server utilization as the number of servers increases. The system experiences a 5% decrease in performance gain for every 20 servers added. After about 40 servers, the performance gain begins drops off as the number of servers increases.

### 4.2 Scalability



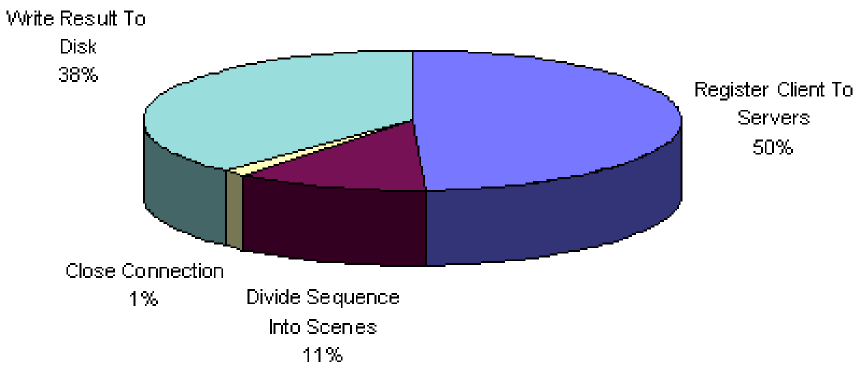

Graph 2 - Client Speed vs. Number of Servers

Graph 2 shows the results of the experiment with 10, 20, 30, 40, and 50 servers. As the number of servers increases, the amount of time required to encode the 60 second MPEG decreases quadratically with peak performance around 40 machines. These results are in line with those shown in graph 1.




Graph 3 - Speedup vs, Number of Servers

As graph 3 shows, the parallel encoder performs up to 19 times faster than the sequential encoder using 40 servers. Using 10 servers yields a 7 times increase in speed. However, using more than 40 servers results in reduced speedup as shown in graph 3.




Graph 4 - General Overhead on Client

Graph 2 shows that client overhead remains fairly constant no matter how many servers are present in the system. We can break the client overhead down into serveral categories, shown in graph 4. For example, connection establishment representss about 50%, or approximately 10 seconds, of total client overhead as shown by cross referencing with graph 2.

### 4.3 Disk I/O Contention

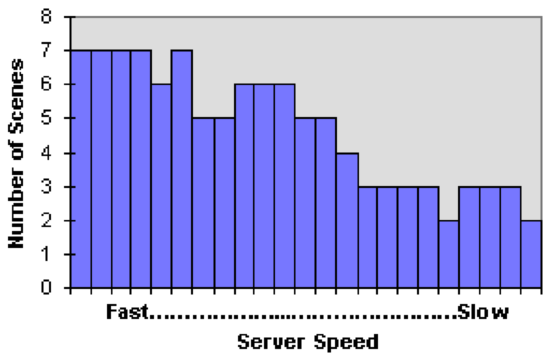
The number of servers concurrently requesting information from the shared file system increases with the number of servers used for compression. Because data is only stored on one disk, disk I/O contention increases because there is only one read head to satisfy many requests. As a direct result, processes must wait longer to receive results from the disk. This phenomenon infects all shared resource system architecture, and therefore is not specific to this application.


Graph 5 - Encoding Time Per Second of MPEG by Server

Graph 5 shows the average compression time for a one second MPEG using different numbers of machines. The given compression times exclude idle time in the server, therfore this graph represents actual encoding time. No matter how many servers the client uses, graph 5 should remain flat (slope 0), but this is not the case. The only explanation for the performance degredation in graph 5 is disk contention, a 5% degredation for every 20 servers.

### 4.4 Load Balancing

We modelled an experiment after organizations with machines of various speeds by using 25 different servers to compress the same video mentioned above with a 0.5 second step size. The machines varied from a 60Hhz Sparc 20 to a 33Mhz Sparc ELC. This experiment shows how well our algorithm load balances encoding work in a non-uniform environment.




Graph 6 - Number of Scenes Encoded vs. Speed of Server

Graph 6 shows that the client dispatches more scenes to faster machines than it does to the slower machines. It turns out that the use of a relatively small step size is an easy way to perform load balancing. However, if the step size is too small, performance degrades because of increased communication between the client and the servers. If the scene size is too large, the system must wait for the last machine to finish its compression job. It is likely that a slow machine recently received a new segement and will take a long time to complete, thus hindering performance.

# 5. Extensions

## 5.1 IP Multi-cast

IP Multi-cast is a relatively new feature added to the TCP/IP protocol that allows a machine to join an IP Multi-cast group to make its presence know to everyone. By supporting this, machines may dynamically enter and exit the MPEG compression group while a client compresses a video.

## 5.2 Increase Performance

Using a flag to notify the client that a server needs more work has scalability problems. If there are multiple messages in the network queue, the client knows of only one message. If there are multiple messages, the client will process all of them, but there is a delay because assembler only dispatches one new job, then continues to assemble. All other servers are made to wait. For example, in figure 5, after the client sends scene 4 to server C, the idle server flag is clear. Now, server A is still idle, but the assembler does not know this and assembly continues while server A sits idle. One way to solve this problem is to have a queue of idle servers. Each time a client sends a scene to an idle server, the machine gets dequeued. Then, the client can only perform assembly when the idle server queue is empty.

## 5.3 Parallel Scene Cut Detector

Scene detection is a compute intensive operation. The detector built into Rivl is a Tcl script and would benefit from parallelism just as MPEG encoding did.

### 5.4 Non Destructive Time Out Mechanism

Currently, when a server times out, the client kills the connection to the server and resends its request to another server. This scheme gets the desired work done, but may kill a good server permaturely because it is slow. If the slow machine was almost finished, its work is gone and the new server must start it from the beginning. For example, if the client, upon receiving notification that server A timed out, sent the request to server B, but did not kill the connection to server A, server A has a chance of finishing before server B because it had a head start.

### 5.5 Use Isis

Isis is a toolkit for developing reliable, high performance distributed computing applications in a client-server environment. Tcl/Dp could be replaced by Tcl/Isis to provide a more robust communication layer for the parallel encoder. Building reliability into the communication layer leaves more time to improve the multimedia aspects of the encoder.

### 5.6 Use Rivl Handles

The scipt passed on the command line contains a list of commmands to reconstruct the MPEG video sequence on each of the servers. This enables the servers to create the correct output, but requires the user to remember each command issued to produce a video sequence. To make things easier on the user, the encoder could use a Rivl handle, which contains all of the necessary information, to help the servers reconstruct the proper sequence. This adds a step to the encoder, but makes the command easier to use.

## 6. Conclusion

Our parallel encoder exhibits a 19 times speedup over its serial counterpart using 40 workstations. It also achieves faster than real-time compression for a 60 second, 160x120 MPEG video by finishing in 55 seconds. Because the speedup is not proportional to the number of servers, 20 servers can reach a 14 times speedup, dramatically reducing the amount of time required to compress video without supercomputers. In addition, our parallel encoder is as easy to use as its serial sibling.

## 7. 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.

---

**Last Modified: 23:50pm EDT, April 09, 1996**