ECE 451 Final Project

Parallel Video Compression through MPI and FFMPEG Libraries

Zac Brown, Praveen Chekuri, Pradnya Pisal
12/30/12

Files:

ffNT.avi – the video file we are compressing

*must download or wget from the link below

https://dl.dropbox.com/u/4619278/Parallel%20Project/ffNT.avi

test.mpg – the compressed video file. It is the output to our project

you can run the parallel_final.c code or download the file from the link below

http://dl.dropbox.com/u/19657747/test.mpg

sequential.c – sequential code for compressing a video to MPEG1

compiling:

```
 gcc -Wall -O2 -g   -c -o sequential.o sequential.c
gcc sequential.o  -pthread -lavdevice -lavfilter -lavformat -lavcodec -ldl -lasound -lSDL
-lpthread -lz -lrt -lswresample -lswscale -lavutil -lm    -o sequential
```

running:

```
./sequential [video file]
ex) ./sequential ffNT.avi
```

parallel_final.c – parallel code for compressing the video ffNT.avi to MPEG1

compiling:

```
mpicc -Wall -O2 -g   -c -o parallel.o parallel_final.c
mpicc parallel.o  -pthread -lavdevice -lavfilter -lavformat -lavcodec -ldl -lasound -lSDL
-lpthread -lz -lrt -lswresample -lswscale -lavutil -lm    -o parallel
```

running:

```
mpirun -np [num_procs] ./parallel_final
```

*hard coded to compress ffNT.avi so make sure its in the same directory

parallel_performance.c – code for the parallel perforance analysis

compiling:

```
mpicc -Wall -O2 -g   -c -o parallel_p.o parallel_performance.c
mpicc parallel_p.o  -pthread -lavdevice -lavfilter -lavformat -lavcodec -ldl -lasound -
lSDL -lpthread -lz -lrt -lswresample -lswscale -lavutil -lm    -o parallel_performance
```

running:

```
mpirun -np [num_procs] ./parallel_performance
```

OR run this loop to get the performance of multiple nodes:

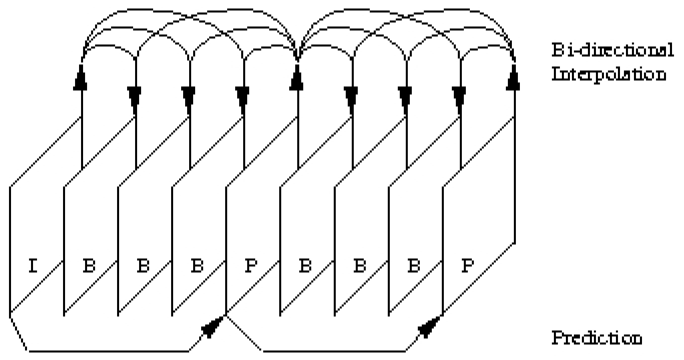
```
for ((  i = 2 ;  i <= 16;  i++  ))
do
rm test.mpg
mpirun -np $i ./parallel >> full.txt
done
```

Introduction

Video compression is a very useful and important application, especially in today's constant demand for streaming, uploading, and downloading video content from the internet. The process, however, can be very time consuming and can take some heavy processing power. This project takes on the task of increasing the efficiency of video compression through parallel computation using MPI libraries.

There are two types of video compression. Lossless compression tries to keep the quality of the video intact while lossy compression sacrifices the video quality for higher compression ratios. This project uses the Lossy MPEG1 video compression algorithm to compress video as small as possible, as fast as possible. MPEG1 Encoding begins by first transforming from the RGB color space to the YUV color space. This is because after experimentation, the creators of MPEG concluded that the Color Intensity (Y) of a frame is most noticeable to the human eye. The chrominance aspects (UV) are less noticeable and can be further compressed without drastically hurting the video quality.

MPEG next separates frames into 3 different types. Intra Frames (I-Frames) are considered reference frames. They are compressed using JPEG image compression. JPEG applies a Discrete Cosine Transform to 8x8 blocks of pixels. These 8x8 blocks are then divided by an 8x8 JPEG standard table. Dividing by this table actually makes most of the pixel values zero. JPEG then writes these blocks to a file by storing the coefficients of the blocks and the number of zeros between these non-zero values. These I-Frames are used as reference frames for the other two types of frames. Predicted Frames (P-Frames) use I-Frames and other P-Frames to predict motion. Instead of compressing these frames using the JPEG encoding, these frames instead try best to guess where the 8x8 blocks have moved since the last reference frame. Storing these motions vectors compresses the frame far more than just using JPEG for the entire video. The last type of frame is a Bidirectional Frame (B-Frame) and is the same thing as a P-Frame but uses past and future reference frames to predict motion.



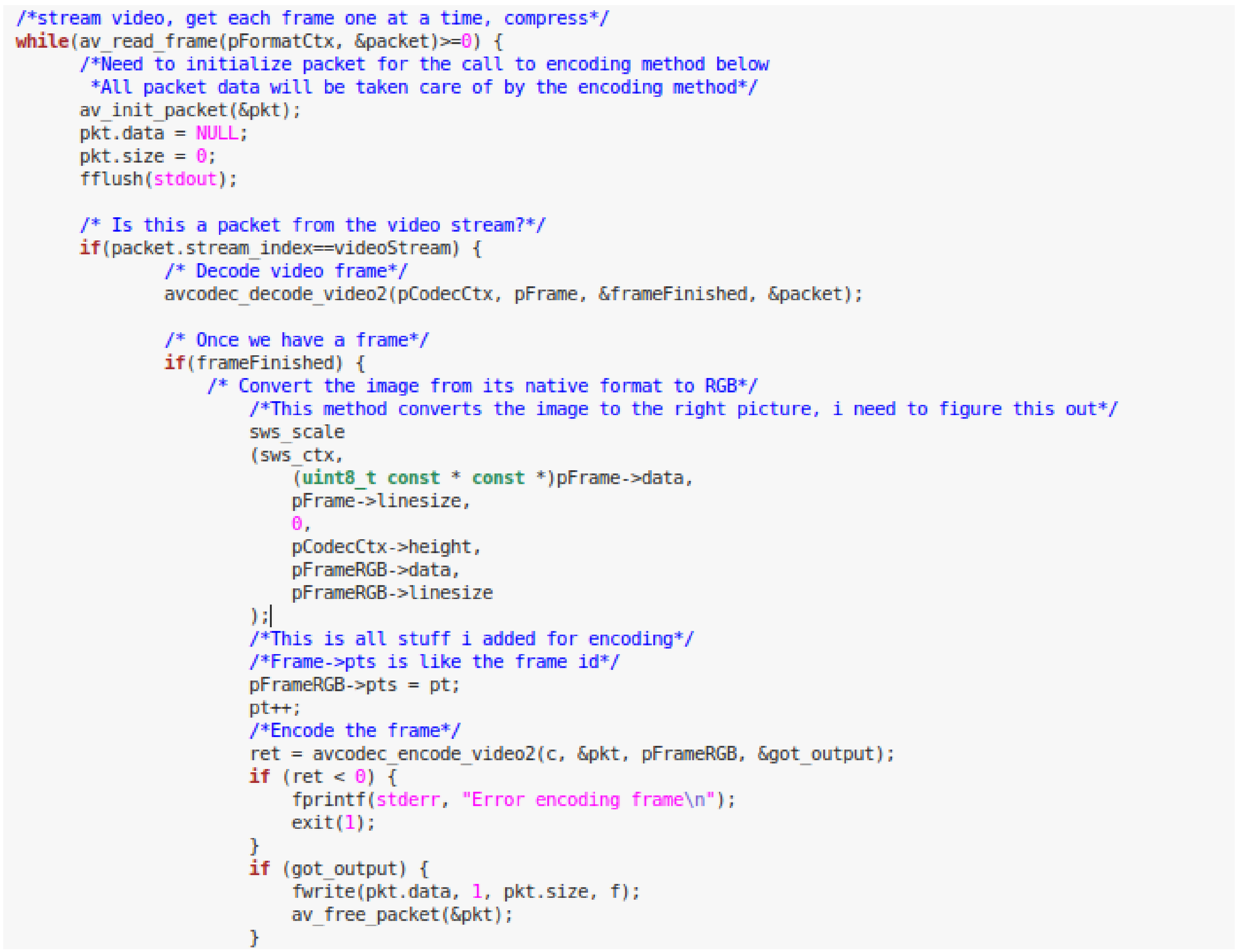
The process of encoding each frame is the bulk of the work for MPEG1 compression. This project implements FFMPEG libraries to take on this arduous encoding process. Below is an explanation of how our code uses both the MPI and FFMPEG libraries to encode a video.

Sequential Implementation

The general idea behind the code is that our system will:

1) Decode the Video to extract each frame
2) Encode each frame sequentially
3) Write the encoded frame to a new file.

Decoding the video first is essential to our solution because it is the method used to extract the three types of frames. We began deriving the solution to optimizing video encoding by first creating a sequential algorithm for the process. This code can be found in sequential.c. While this code is quite long and somewhat confusing, the most important aspects will be explained below.



```c
/*stream video, get each frame one at a time, compress*/
while(av_read_frame(pFormatCtx, &packet)>=0) {
      /*Need to initialize packet for the call to encoding method below
       *All packet data will be taken care of by the encoding method*/
      av_init_packet(&pkt);
      pkt.data = NULL;
      pkt.size = 0;
      fflush(stdout);

      /* Is this a packet from the video stream?*/
      if(packet.stream_index==videoStream) {
              /* Decode video frame*/
              avcodec_decode_video2(pCodecCtx, pFrame, &frameFinished, &packet);

              /* Once we have a frame*/
              if(frameFinished) {
                  /* Convert the image from its native format to RGB*/
                      /*This method converts the image to the right picture, i need to figure this out*/
                      sws_scale
                      (sws_ctx,
                          (uint8_t const * const *)pFrame->data,
                          pFrame->linesize,
                          0,
                          pCodecCtx->height,
                          pFrameRGB->data,
                          pFrameRGB->linesize
                      );
                      /*This is all stuff i added for encoding*/
                      /*Frame->pts is like the frame id*/
                      pFrameRGB->pts = pt;
                      pt++;
                      /*Encode the frame*/
                      ret = avcodec_encode_video2(c, &pkt, pFrameRGB, &got_output);
                      if (ret < 0) {
                          fprintf(stderr, "Error encoding frame\n");
                          exit(1);
                      }
                      if (got_output) {
                          fwrite(pkt.data, 1, pkt.size, f);
                          av_free_packet(&pkt);
                      }
```

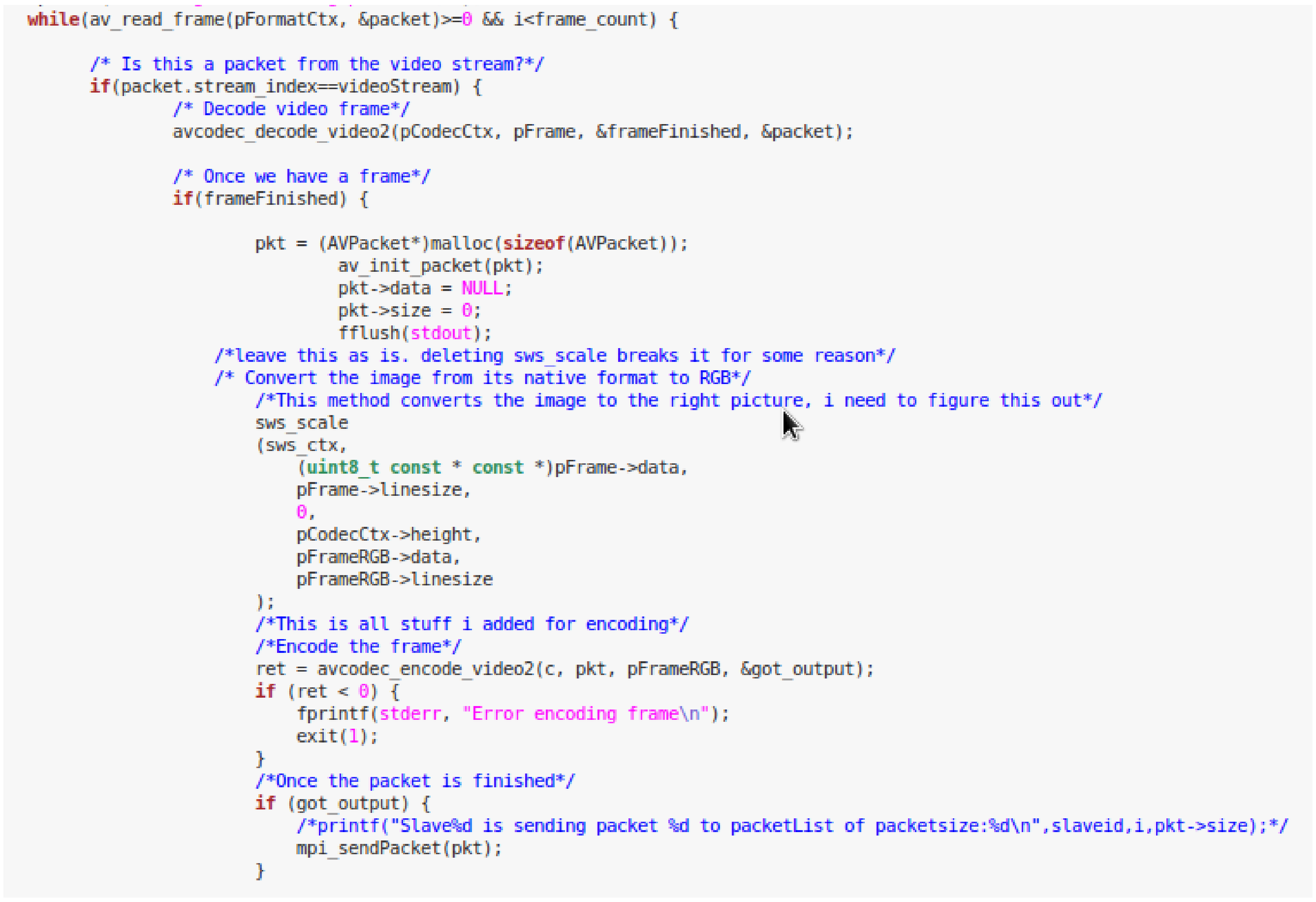
This loop streams the video until the method av_read_frame() fails to open another frame, suggesting the video has finished streaming. The most important data structure to this program is the AVPacket, which is the structure that contains all of the information we will be writing to our encoded file. There are two different AVPackets in this loop. The first (packet) is used for decoding the video and extracting frames and the second (pkt) is used for encoding the frames. The AVPacket packet contains multiple streams from the video file. We are only compressing video so the first if statement looks to find the video stream only. The next line calls avcodec_decode_video which actually decodes the video and stores the frame in the structure AVFrame pFrame. Once the frame has finished decoding, we call the function sws_scale() which converts the frame to the desired YUV color space. Lastly this loop then calls the function avcodec_encode_video2() which does the MPEG1 encoding described above. Once the frame is encoded, we write the data from the AVPacket pkt to the output file.

There are a couple of things that should be noted from the sequential code. First, this algorithm only compresses video, and the output file will be silent because we do not write any audio information to it. Therefore, to truly see the compression ratio, a silent video file will be used as an input. Audio compression greatly complicates this system. You must first separate the file into two streams (demux) and then compress both the video stream and audio stream separately. Then you have to recombine these two compressed streams (mux) back together and write it to a file. Unfortunately, we did not have enough time to research the audio compression process or a way to implement it.

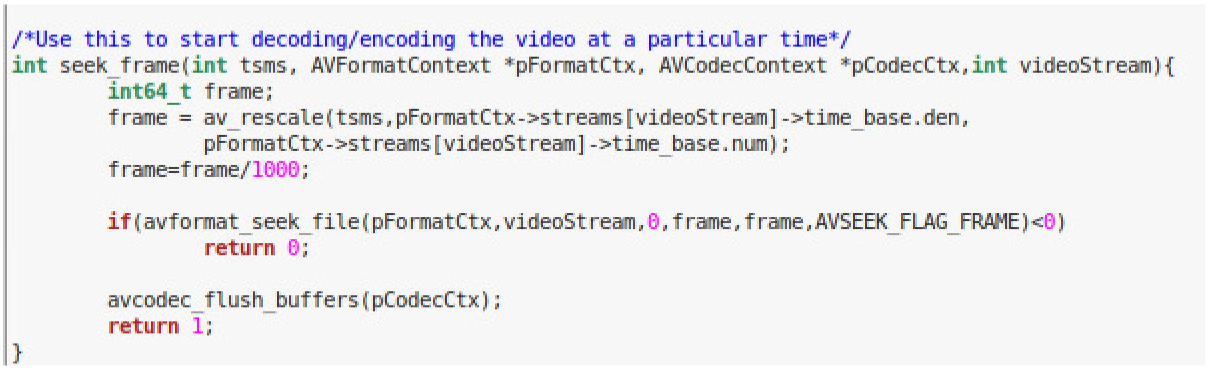
The second, and more important observation, is that the encoded AVPackets must be written in the proper order for the video to be a real video. Our original idea for parallelizing the code was to have each node receive a frame one at a time. Whenever a slave finished the encoding process, it would send the frame back to the master, which would then write the encoded frame to the file, and wait for the next frame to encode. This method will not work because if slave 2 finishes frame 20 before slave 1 finishes frame 19, the file will not be written in the proper order.

Our solution to optimizing the sequential code is to statically divide the original video file into chunks. Each slave will receive a time to begin decoding the original video, and a frame count signifying the number of frames to encode. The slave will then send all of its encoded AVPackets back to the master node. The key to this solution, however, is that the master node receives the packets from the slaves in the proper order. This is done by first receiving all of the packets from slave 1 and writing them to the file, and repeating this process for all subsequent slaves. So our code works by having each slave perform the sequential algorithm explained above only for a portion of the video.

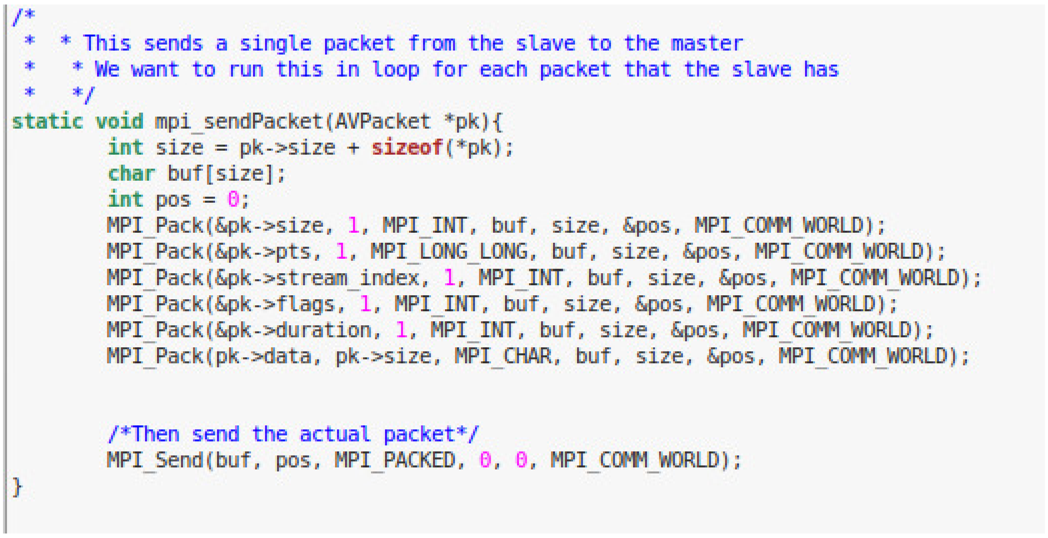
Parallel Implementation


```
while(av_read_frame(pFormatCtx, &packet)>=0 && i<frame_count) {

      /* Is this a packet from the video stream?*/
      if(packet.stream_index==videoStream) {
              /* Decode video frame*/
              avcodec_decode_video2(pCodecCtx, pFrame, &frameFinished, &packet);

              /* Once we have a frame*/
              if(frameFinished) {

                      pkt = (AVPacket*)malloc(sizeof(AVPacket));
                              av_init_packet(pkt);
                              pkt->data = NULL;
                              pkt->size = 0;
                              fflush(stdout);
                  /*leave this as is. deleting sws_scale breaks it for some reason*/
                  /* Convert the image from its native format to RGB*/
                      /*This method converts the image to the right picture, i need to figure this out*/
                      sws_scale
                      (sws_ctx,
                          (uint8_t const * const *)pFrame->data,
                          pFrame->linesize,
                          0,
                          pCodecCtx->height,
                          pFrameRGB->data,
                          pFrameRGB->linesize
                      );
                      /*This is all stuff i added for encoding*/
                      /*Encode the frame*/
                      ret = avcodec_encode_video2(c, pkt, pFrameRGB, &got_output);
                      if (ret < 0) {
                          fprintf(stderr, "Error encoding frame\n");
                          exit(1);
                      }
                      /*Once the packet is finished*/
                      if (got_output) {
                          /*printf("Slave%d is sending packet %d to packetList of packetsize:%d\n",slaveid,i,pkt->size);*/
                          mpi_sendPacket(pkt);
                      }
```

This is the main loop each slave performs for encoding their chunk of video. The slave chooses a time to begin this loop by calling the method seek_frame(). Seek frame changes the start time of the video by specifying a time in milliseconds, which is sent by the master to the slave.

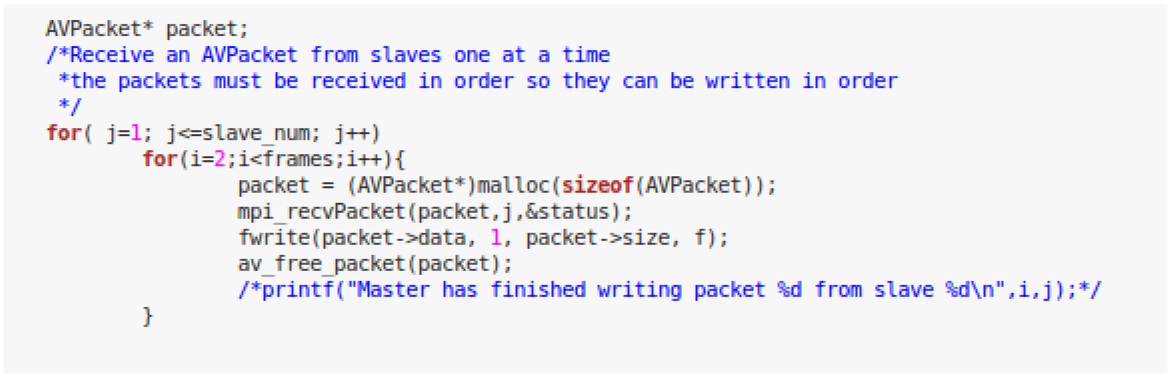

```
/*Use this to start decoding/encoding the video at a particular time*/
int seek_frame(int tsms, AVFormatContext *pFormatCtx, AVCodecContext *pCodecCtx,int videoStream){
        int64_t frame;
        frame = av_rescale(tsms,pFormatCtx->streams[videoStream]->time_base.den,
                pFormatCtx->streams[videoStream]->time_base.num);
        frame=frame/1000;

        if(avformat_seek_file(pFormatCtx,videoStream,0,frame,frame,AVSEEK_FLAG_FRAME)<0)
                return 0;

        avcodec_flush_buffers(pCodecCtx);
        return 1;
}
```

The only difference between the sequential code and the slave's code is that once the packet has finished, instead of writing to a file, the slave calls the method mpi_sendPacket().



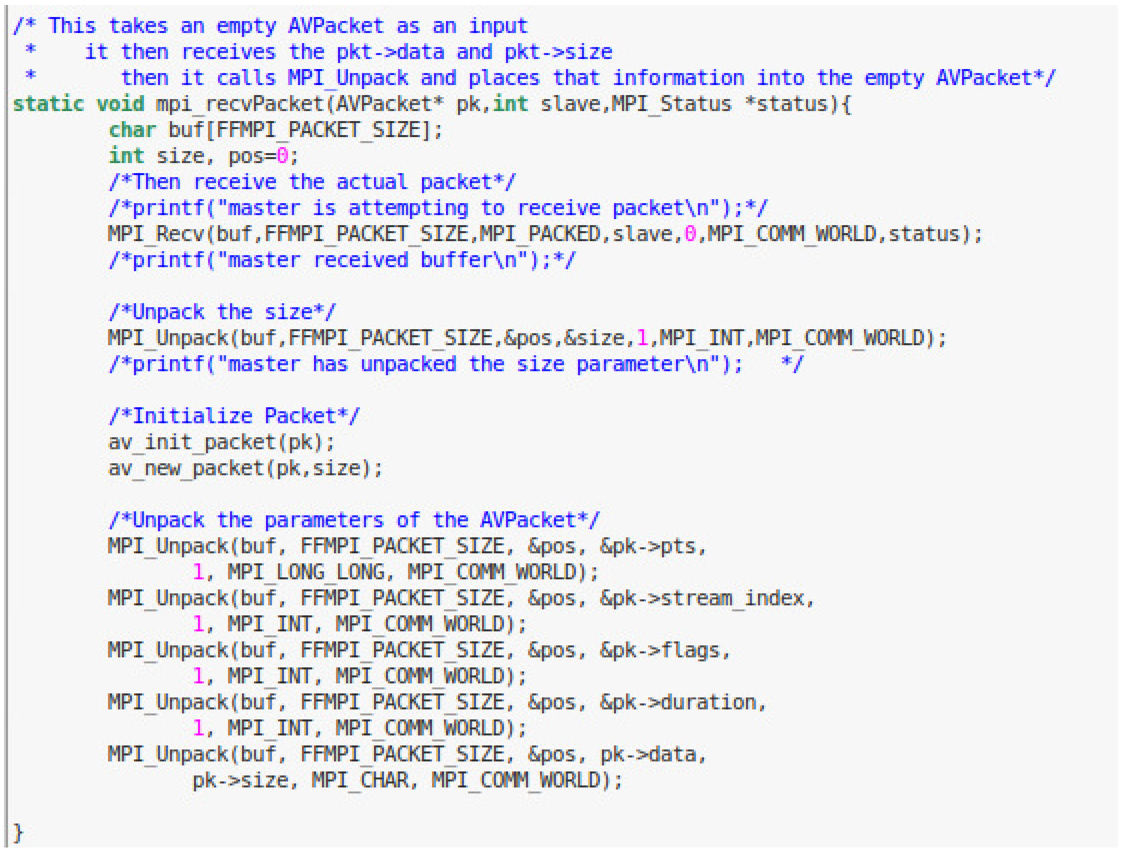
```
/*
 *  * This sends a single packet from the slave to the master
 *   * We want to run this in loop for each packet that the slave has
 *   */
static void mpi_sendPacket(AVPacket *pk){
        int size = pk->size + sizeof(*pk);
        char buf[size];
        int pos = 0;
        MPI_Pack(&pk->size, 1, MPI_INT, buf, size, &pos, MPI_COMM_WORLD);
        MPI_Pack(&pk->pts, 1, MPI_LONG_LONG, buf, size, &pos, MPI_COMM_WORLD);
        MPI_Pack(&pk->stream_index, 1, MPI_INT, buf, size, &pos, MPI_COMM_WORLD);
        MPI_Pack(&pk->flags, 1, MPI_INT, buf, size, &pos, MPI_COMM_WORLD);
        MPI_Pack(&pk->duration, 1, MPI_INT, buf, size, &pos, MPI_COMM_WORLD);
        MPI_Pack(pk->data, pk->size, MPI_CHAR, buf, size, &pos, MPI_COMM_WORLD);


        /*Then send the actual packet*/
        MPI_Send(buf, pos, MPI_PACKED, 0, 0, MPI_COMM_WORLD);
}
```

This is the method we created for sending an AVPacket from the slave back to the master. It converts the important data from the AVPacket to the type MPI_PACKED, which can be sent between nodes. This code is meant only for sending data from the slaves back to the master which will then receive the packet and write it to a file. Note that this code only sends one packet at a time, so much be called for each packet to be sent.



```
AVPacket* packet;
/*Receive an AVPacket from slaves one at a time
 *the packets must be received in order so they can be written in order
 */
for( j=1; j<=slave_num; j++)
        for(i=2;i<frames;i++){
                packet = (AVPacket*)malloc(sizeof(AVPacket));
                mpi_recvPacket(packet,j,&status);
                fwrite(packet->data, 1, packet->size, f);
                av_free_packet(packet);
                /*printf("Master has finished writing packet %d from slave %d\n",i,j);*/
        }
```

Above is a portion of code for the Master node. It is a loop that calls mpi_recvPacket(), which simply unpacks the MPI_PACKED structure from the slave, to retrieve the encoded packets in order from the slaves. After the packets are retrieved and unpacked, the packet is then written to a file.

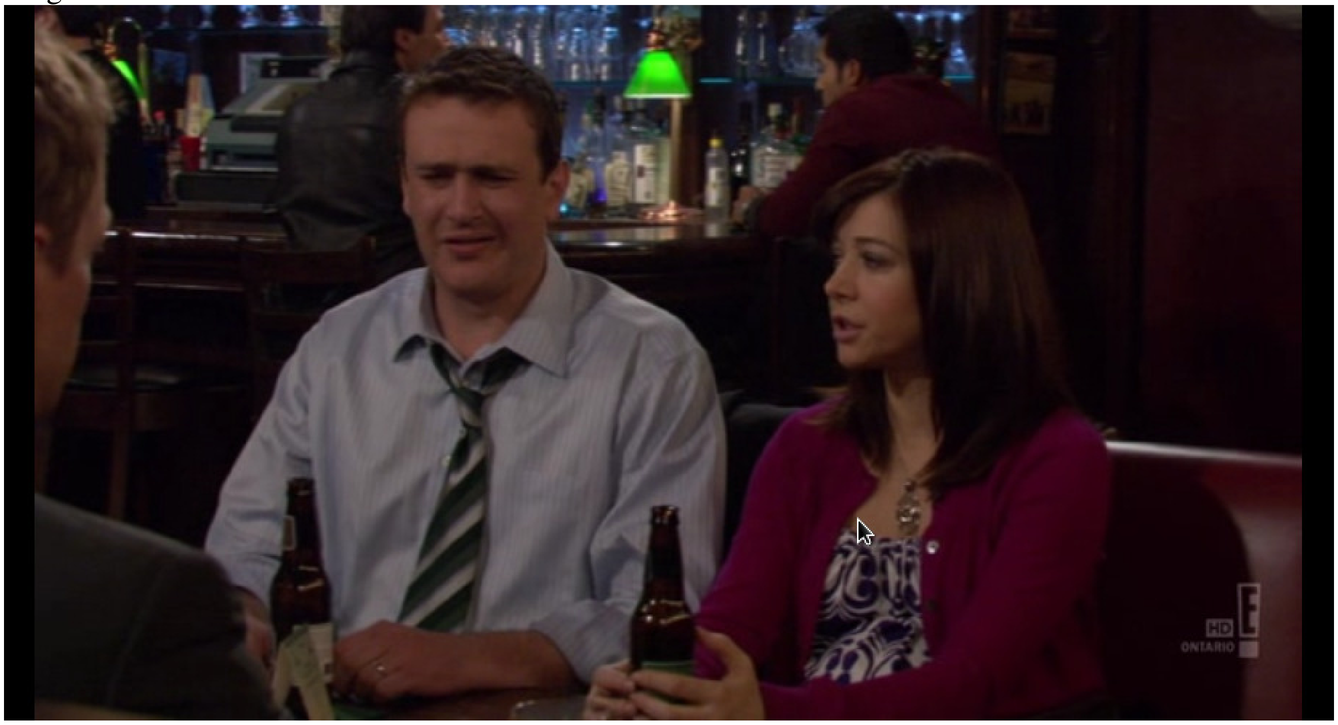


```c
/* This takes an empty AVPacket as an input
 *    it then receives the pkt->data and pkt->size
 *       then it calls MPI_Unpack and places that information into the empty AVPacket*/
static void mpi_recvPacket(AVPacket* pk,int slave,MPI_Status *status){
        char buf[FFMPI_PACKET_SIZE];
        int size, pos=0;
        /*Then receive the actual packet*/
        /*printf("master is attempting to receive packet\n");*/
        MPI_Recv(buf,FFMPI_PACKET_SIZE,MPI_PACKED,slave,0,MPI_COMM_WORLD,status);
        /*printf("master received buffer\n");*/

        /*Unpack the size*/
        MPI_Unpack(buf,FFMPI_PACKET_SIZE,&pos,&size,1,MPI_INT,MPI_COMM_WORLD);
        /*printf("master has unpacked the size parameter\n");    */

        /*Initialize Packet*/
        av_init_packet(pk);
        av_new_packet(pk,size);

        /*Unpack the parameters of the AVPacket*/
        MPI_Unpack(buf, FFMPI_PACKET_SIZE, &pos, &pk->pts,
               1, MPI_LONG_LONG, MPI_COMM_WORLD);
        MPI_Unpack(buf, FFMPI_PACKET_SIZE, &pos, &pk->stream_index,
               1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(buf, FFMPI_PACKET_SIZE, &pos, &pk->flags,
               1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(buf, FFMPI_PACKET_SIZE, &pos, &pk->duration,
               1, MPI_INT, MPI_COMM_WORLD);
        MPI_Unpack(buf, FFMPI_PACKET_SIZE, &pos, pk->data,
               pk->size, MPI_CHAR, MPI_COMM_WORLD);

}
```

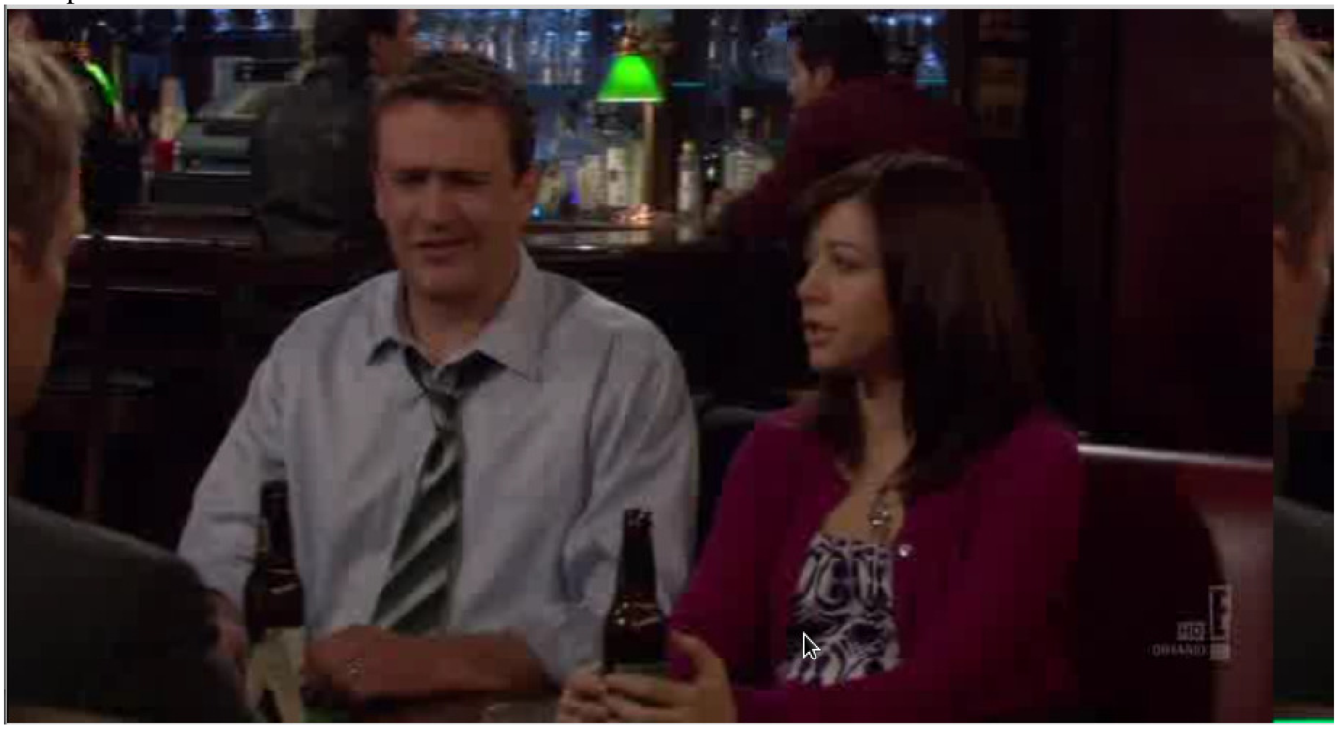
While this code is far more efficient than the original sequential code, it does have some issues. The main problem with this code is that for example, slave8 may finish decoding/encoding its entire chunk, and sending all of its data to the master before slave1 finishes its chunk of work. All of slave8's data can't be written until the master receives and writes the data from slaves 1-7. This means that slave8 is purely idle while the master receives all other packets. Unfortunately, we could not derive a dynamic scheme for dividing the work up to remove this issue without writing the packets in an improper order.

Here are the results from the compressed video file.

Original

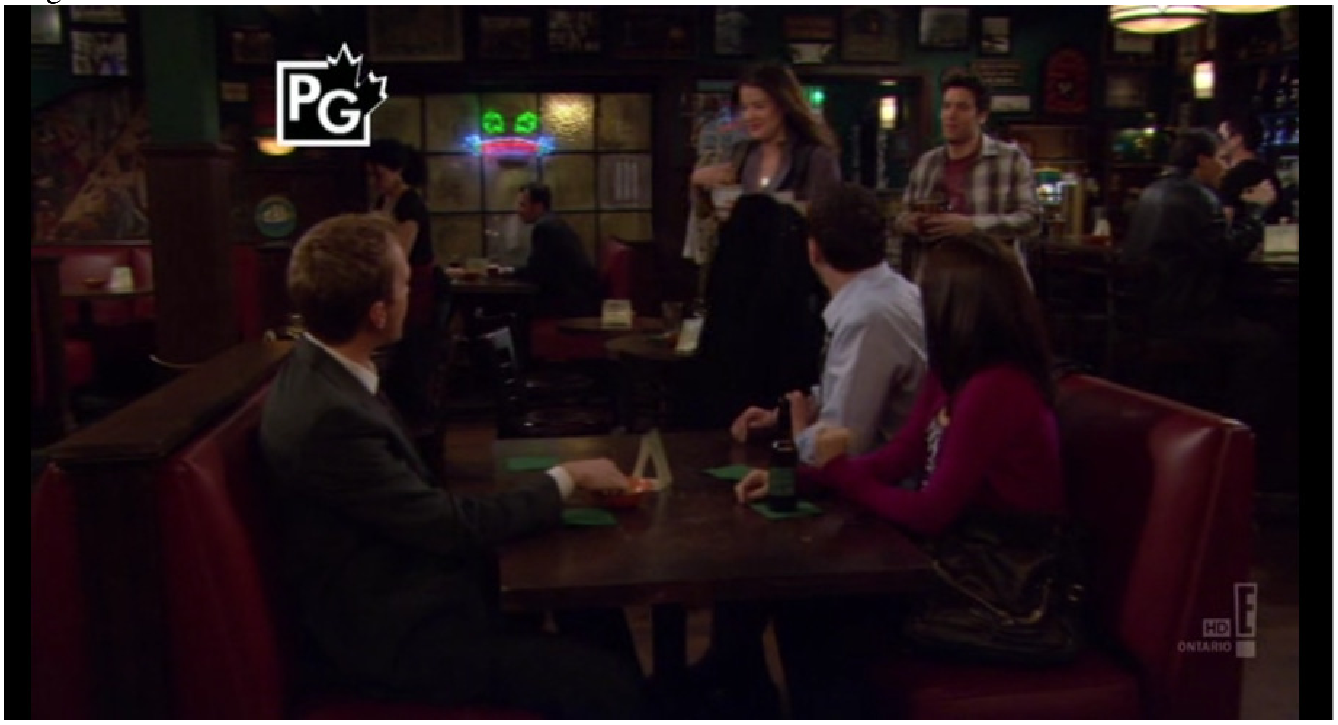


Compressed

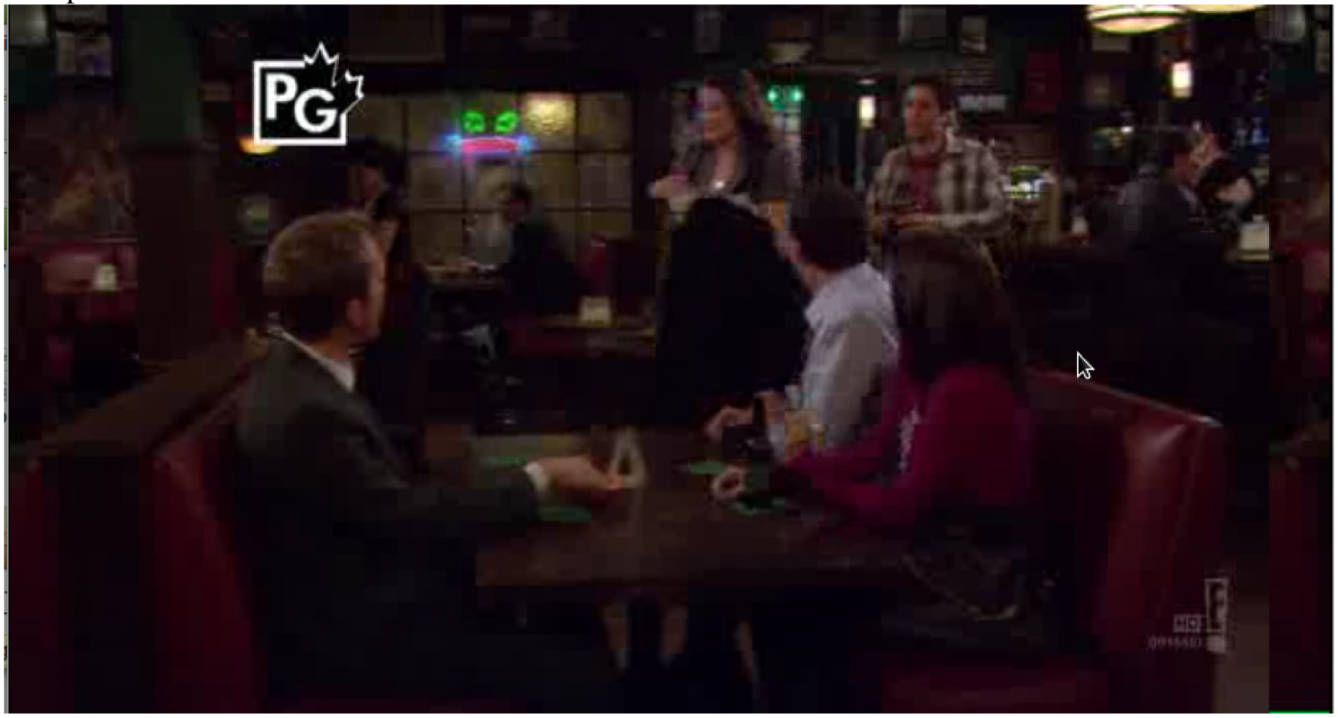


Notice how the compressed file seems a little blocky. These are the 8x8 blocks from the JPEG compression. The frame will seem blockier and blockier with more motion between frames, as seen below.

Original



Compressed

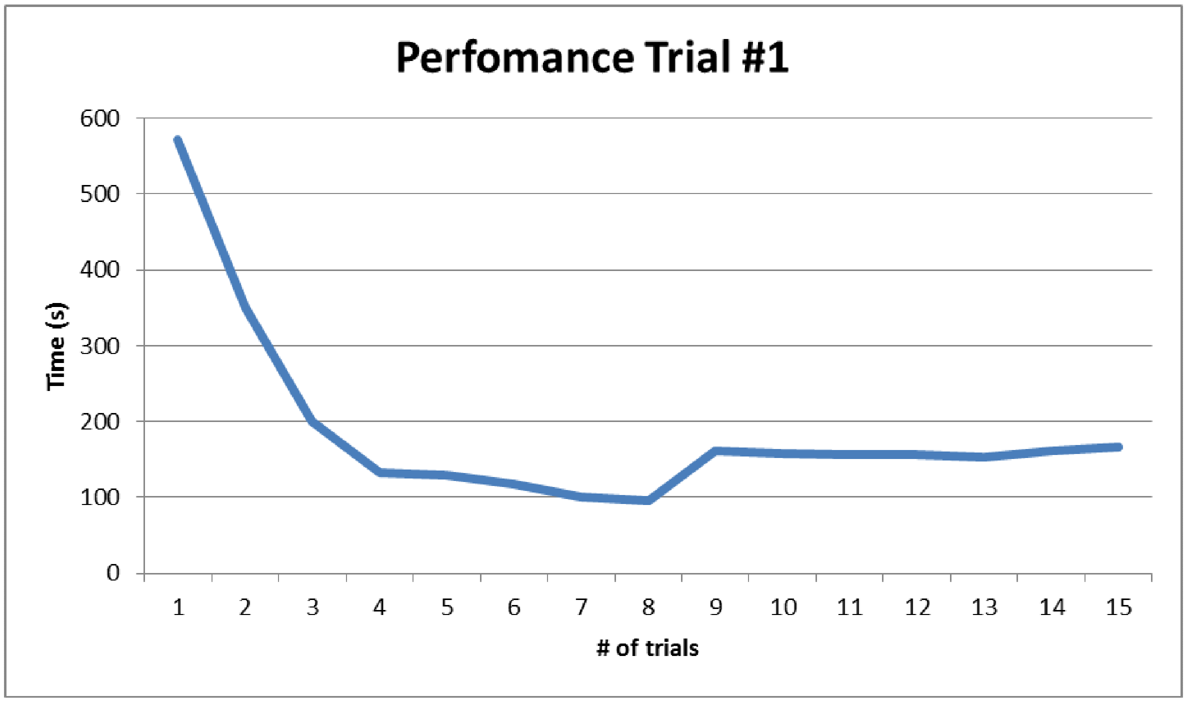


Performance Analysis

Once we had our MPI Implementation ready. The next step was to find out how many cores optimized the compression the most. We ran 2 trials in order to test for consistency. Both our tests returned similar results as we can see below.

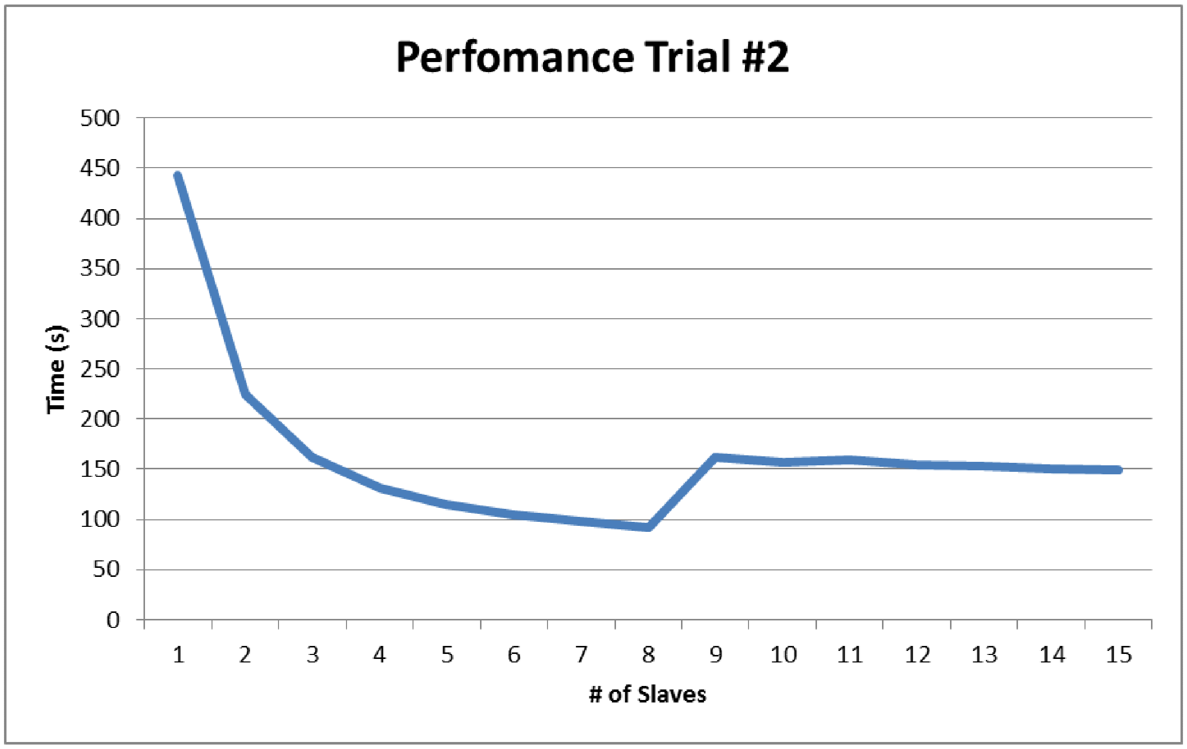
Trial 1

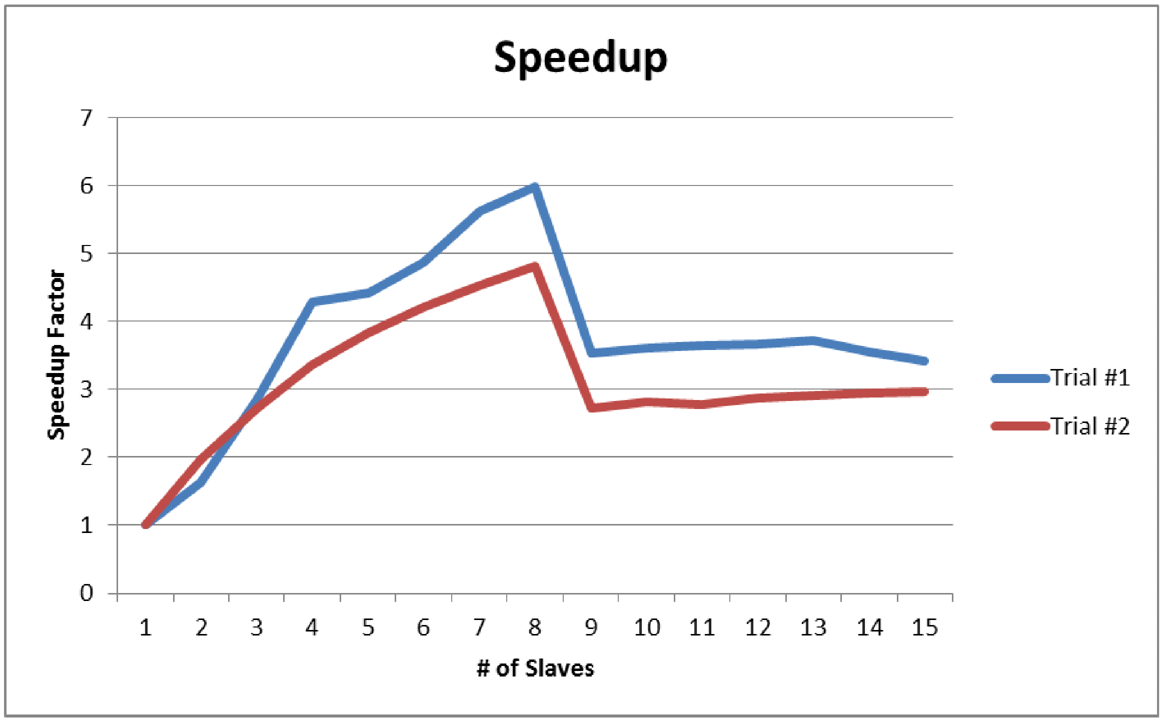
# of Slaves	Frames/Slave	Time (s)	Speedup
1	30220	571.358604	1
2	15110	351.278278	1.626512767
3	10073	199.77997	2.859939382
4	7555	133.471101	4.280766396
5	6044	129.06989	4.426738134
6	5036	117.118116	4.878481857
7	4317	101.417841	5.633709004
8	3777	95.501504	5.982718387
9	3357	161.64197	3.534716906
10	3022	158.594224	3.60264447
11	2747	156.902778	3.641481759
12	2518	156.080525	3.66066557
13	2324	153.749247	3.716171722
14	2158	160.936067	3.550220996
15	2014	167.207038	3.417072695



Trial 2

# of Slaves	Frames/Slave	Time (s)	Speedup
1	30220	443.041228	1
2	15110	223.959001	1.978224702
3	10073	162.565966	2.725301236
4	7555	131.974912	3.357010975
5	6044	115.431692	3.838124698
6	5036	105.133262	4.214091902
7	4317	97.881384	4.526307352
8	3777	91.894477	4.82119538
9	3357	162.361949	2.728725731
10	3022	156.779706	2.825883779
11	2747	159.15858	2.783646524
12	2518	153.952706	2.877774867
13	2324	152.575942	2.903742374
14	2158	150.565334	2.94251815
15	2014	149.590095	2.961701629





Conclusion

As we can see from the above charts, the project was a success as our goal was to speedup video compression using MPI. We attain maximum speedup by using 9 cores, i.e 1 master and 8 slave nodes. In both our trials 8 slaves attained the highest speedup, therefore we can conclude that a video containing approximately 30220 frames can be optimally compressed using 9 cores.

Overall, we are very pleased with the results from this experiment. The code compressed the 161.2 MB file down to 62.2 MB, making it nearly a third of the size of the original. The parallel code also incredibly decreased the time needed to compress the file from about 9 minutes to 1:30. These results are absolutely amazing to us. This experiment was truly an eye opening experience to the power of parallel computing and how its concepts can be applied to increase the speed and efficiency of many computer tasks.