A Parallel Ultra-High Resolution MPEG-2 Video Decoder for PC Cluster Based Tiled Display Systems

Han Chen
Princeton University
Princeton, New Jersey
chenhan@cs.princeton.edu

Kai Li
Princeton University
Princeton, New Jersey
li@cs.princeton.edu

Bin Wei
AT&T Labs Research
Florham Park, New Jersey
bw@research.att.com

Abstract

This paper presents a hierarchical parallel MPEG-2 decoder for playing ultra-high-resolution videos on PC cluster based tiled display systems. To maximize parallelism while minimizing the communication requirements for a PC cluster, our algorithm uses a two-level splitter approach, where a root splitter splits an MPEG-2 video stream at the picture level and passes them to k second-level splitters, each of which splits the pictures into macroblocks and sends them to $m \times n$ *decoders according to their screen locations.*

Our experiments with various configurations show that this system is highly scalable and has a low and balanced communication requirement among the PC nodes. On a 4×4 *display wall system driven by 21 PCs, the implementation can play back a* 3840×2800 *video at 38.9 frames per second.*

1. Introduction

Driven by the rapid development of digital projector technology during the past decade, building a tiled display system using commodity projectors and a PC cluster has become an affordable approach to ultra-high-resolution wall-size display systems [7][10]. High resolution videos can be effective in applications such as remote data visualization, interactive collaboration, and immersive environments. However, most high-end video processor based video walls only support resolutions up to *High Definition Television* (HDTV). To play ultra-high-resolution videos with resolution matching that of a display wall calls for a novel parallel decoding system.

In a parallel MPEG-2 video decoder for PC cluster based tiled display wall systems (Figure 1 shows a generalized scheme), three major components work together. A splitter divides the input stream into small work units and sends them to the decoders. The decoders might need to communicate with each other to decode a picture. Finally, the decoded pixels might be redistributed before being displayed. There are two challenges in successfully building such a system: high performance and scalability. The system should be able to play ultra-high-resolution videos at an interactive frame rate while keeping the communication requirement at a minimum such that an off-the-shelf network can be used.

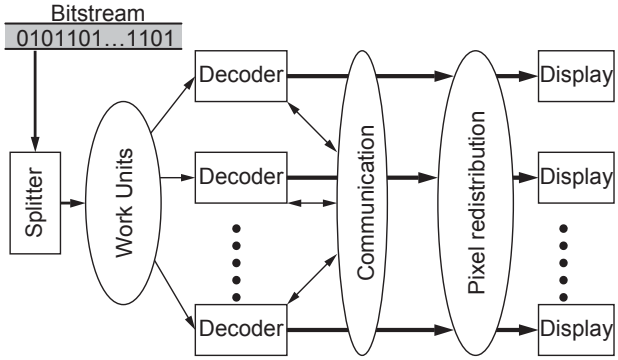

Figure 1. A Generalized Parallel Decoder for PC Cluster

In this paper, we present a hierarchical parallel MPEG-2 video decoder for the PC cluster based tiled display wall architecture. It is called a 1-*k*-(*m*,*n*) system with a root splitter splitting MPEG-2 video stream at picture level and passing them to *k* second-level splitters, each of which splits the pictures at macroblock level and sends macroblocks to one of the $m \times n$ decoders according to its screen location.

We show that such a system is highly scalable and has a low and balanced communication requirement among the cluster nodes. In a 1-4-(4,4) setup, it can play back a 3840×2800 MPEG-2 video stream at 38.9 frames a second, or an equivalent bit rate of 130 Mbps.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to the MEPG-2 video standard. Section 3 discusses the issues in parallel decoding for a tiled display system and related previous work. Section 4 presents our parallel decoder and discusses design issues. Section 5 reports our experimental results. Finally, Section 6 summarizes what we have learned in our study and future work.

2. MPEG-2 Overview

MPEG-2 is a set of ISO standards, consisting of a video standard, an audio standard, and a system layer standard for multiplexing. MPEG-2 video removes both spatial and temporal redundancies to achieve high compression ratios. It removes temporal redundancy by using motion estimation and compensation. An encoder specifies one or more motion vectors for macroblocks in some pictures. A motion vector

points to the closest prediction of the current macroblock from previous pictures. To further remove spatial redundancy, the prediction residual is transformed using *Discrete Cosine Transform* (DCT) on a block basis, and the resulting coefficients are quantized and run-length encoded.

There are three types of pictures in an MPEG-2 video stream: Intra (I), Predicted (P) and Bi-directional predicted (B). No motion compensation is used in I-pictures. In P-pictures, macroblocks are uni-directionally predicted using only the last I- or P-picture. In B-pictures, macroblocks can be bi-directionally predicted using both last and next I- or P-pictures. Figure 2 shows an example of a series of pictures.

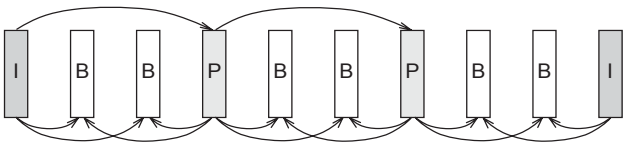

Figure 2. A Series of Pictures

In encoded video bitstreams (see Figure 3), a 32-bit byte-aligned start code is provided for each sequence, GOP, picture, and slice. However, a macroblock does not have a start code, nor is its start or end necessarily byte-aligned.

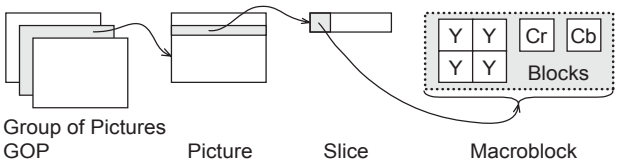

Figure 3. MPEG-2 Video Stream Syntactic Elements

3. Previous Work and Discussion

After Patel *et al.* first described their software MPEG video decoder [12], people have proposed various methods to increase its performance via parallelization at different levels.

Bilas *et al.* proposed a parallel decoder on a shared memory SMP [3] and investigated parallelism at both GOP and slice levels. The decoder can achieve real time frame rate for DVD resolution videos and a few frames per second for HDTV resolution videos. Kwong *et al.* designed a GOP level parallel decoder on an IBM SP parallel computer using a HIPPI network [8]. It only decodes 352×240 resolution videos at 30 fps with 16 nodes.

Bala *et al.* proposed a functional parallelization on a 2-CPU SMP [2]. One thread is used for run length decoding, IDCT, and motion compensation, while another does the dithering and display. This approach does not scale, because as the number of processor increases, it becomes harder to partition the decoder into smaller functional components.

In the hardware front, Lee *et al.* described an MPEG codec on a single-chip multiprocessor[9] exploiting both instruction level and functional parallelism. Yang *et al.* proposed a variable issue architecture for multi-threading MPEG processing [13], but no parallelism model and performance data were given. In the related MPEG encoding field, Yu *et al.* investigated macroblock and slice level parallelism in encoding [14]. Akramullah also proposed a data parallel encoder [1].

Most of these methods use a shared memory SMP or equivalent platform. They focus on improving performance with more processors, but have not considered the issues in scaling video resolutions. Because of the high memory bandwidth requirement of ultra-high-resolution videos, it is impossible for an SMP to display such videos even if it can decode them with enough number of processors. A PC cluster based display system is much better suited for this purpose.

Let's consider different levels of parallelization with regard to the following questions: cost of splitting, inter-decoder communication, and pixel redistribution cost.

• Sequence or GOP level: Because there are byte-aligned start codes for sequences and GOPs, the cost of splitting is minimum. Sequences and most GOPs (closed GOP) are self-contained, therefore the decoders do not need to communicate with each other. However, the cost of pixel redistribution is extremely high. A decoder needs to send out $(mn-1)/(mn)$ of every picture to other nodes for display.

• Picture level: The splitting cost is minimum due to the start code. The decoders need to fetch reference blocks remotely in order to decode a P- or B-picture: up to one entire picture for a P-picture, and two for a B-picture. The pixel redistribution cost is the same as in GOP level parallelism.

• Slice level: The splitting cost is also minimum. Inter-decoder communication is reduced. Because the splitter can group adjacent slices together to form a work unit, a decoder only needs to fetch remote data when a macroblock references data outside the region. The pixel redistribution cost is also reduced. Typically, a slice covers a row of macroblocks, thus only $(m-1)/m$ of a slice needs to be sent to other nodes.

• Macroblock or block level: The splitting cost is high. Due to the lack of start code a splitter has to parse an entire picture to access a macroblock or block. Inter-decoder communication is further reduced. A macroblock is sent to the node where it will be displayed, thus each decoder decodes a rectangle area of a picture. It needs to fetch remote blocks only when a motion vector crosses the boundary of the rectangle. In this arrangement, no pixel redistribution is needed.

Table 1. Comparison of Different Types of Parallelization

Level	Splitting Cost	Communication Requirement	Pixel Redistribution
Sequence	very low	none	very high
GOP	very low	none or low	very high
Picture	very low	very high	very high
Slice	very low	moderate to high	moderate to high
Macroblock or Block	high	moderate	none

As we can see from Table 1, none of these methods suffices in itself. In a coarse granularity parallelization, i.e., sequence, GOP, picture, slice, the splitting cost is very low, but the communication cost is high. In a fine granularity parallelism, i.e., macroblock level, the communication cost is low and distributed, but the splitting cost is high, which becomes a bottleneck when the number of decoders increases.

4. Hierarchical Decoding

An ideal parallel decoder should satisfy the following:

• The communication requirement is low and balanced, such that an off-the-shelf network is adequate;

• The decoding system is bottleneck free;

• It should achieve real-time decoding of higher resolution videos with more nodes. We call this resolution-scalability.

4.1. Hybrid Granularity Hierarchical Decoder

A macroblock level parallel decoder has low and balanced communication requirement, but the high splitting cost causes the splitter to become a bottleneck for videos with resolution higher than HDTV. As noted before, P- and B-pictures have dependencies on previous pictures when being decoded. However, this dependency is nonexistent when the pictures are being split, because a splitter does not motion compensate. Thus multiple pictures can be assigned to multiple independent splitters simultaneously, thereby eliminating the bottleneck. Based on this important observation, we designed a hybrid granularity hierarchical decoder.

The system consists of one or two levels of splitters and a set of decoders. For relatively low-resolution videos, such as DVD or HDTV, a single macroblock-level splitter is adequate. For high-resolution videos, the system uses a root splitter to split the input video at picture level and pass pictures to multiple second-level splitters, which split pictures at macroblock level to feed the decoders. We call it a 1-k-(m,n) system for a hierarchy of one root picture splitter, k second-level macroblock splitters, and $m \times n$ decoders in a tiled $m \times n$-projector display wall system. The root splitter runs on a console PC while the k second-level splitters run on k additional PCs. A single-level macroblock parallel decoder is a special case where $k = 1$; we call it a 1-(m,n) system.

Table 2 lists the high level algorithms for the root splitter, the second-level splitters, and the decoders.

• The root splitter scans an bit stream to find out where a picture starts and ends, copies the picture data to an output buffer, and then sends it to one of the k second-level splitters in a round-robin fashion to balance the workload.

• A second-level splitter parses a picture into macroblocks, and sorts them into $m \times n$ output buffers. We call them sub-pictures (SP). They do not necessarily conform to MPEG-2 syntax. The splitter then sends the sub-pictures to the corresponding decoders. Because there is no inter-picture dependency, the second-level splitters operate independently.

• A decoder (D) decodes the sub-picture it receives one macroblock at a time. When a macroblock's motion vector crosses the boundary of the decoder's screen rectangle, the decoder needs to fetch blocks remotely.

Table 2. High Level Algorithms

```
Root Splitter:
While there are more pictures in the stream
    Copy the current picture P into an output buffer
    Select a second-level splitter
    Send picture to the splitter
```

```
Second-level Splitter:
While splitting has not finished
    Receive a picture P from the root
    For each macroblock M
        For each decoder D(i)
            If M lies in the rectangle of D(i)
                Append M to buffer SP(i)
    Send SP's to D's
```

```
Decoder:
While decoding has not finished
    Receive SP from second-level splitter
    Decode SP; fetch blocks remotely if necessary
    Display the picture
```

The description of the algorithms above is straightforward, but it requires careful designs to be efficient. In the following subsections we discuss several design issues we faced, and the choices we made.

4.2. Pre-calculation of Remote Macroblocks

Fetching remote blocks on demand can be inefficient. First, the decoder has to block and wait. Second, a dedicate server thread is required, which introduces many context switches. An alternative to this is to pre-calculate who needs which macroblocks. Because a second-level splitter parses the entire picture during splitting, it has the knowledge of which macroblock on which decoder references remote blocks from which other decoder. To take advantage of this, we add a macroblock exchange instruction buffer called MEI for each decoder. When a motion vector of macroblock M in the rectangle of D(i) crosses the boundary and references blocks from D(j), the splitter appends an instruction SEND(x, y, i) to MEI(j) and RECV(x, y, j) to MEI(i), where (x, y) is the coordinate of the reference block. After the entire picture is split, the splitter sends MEI(i) and SP(i) to D(i).

When a decoder receives the MEI and SP, it executes the SEND instructions before decoding the picture. This is possible because the reference blocks are always in previously decoded pictures. During decoding, when a decoder needs a remote block, it verifies its coordinates with a RECV instruction and reads the block *locally.*

We call this method *pre-calculation of remote macroblocks*. It eliminates the need of multi-threading in decoders and reduces the overhead of blocking receive to a minimum. The message exchanges also serve as a way of synchronization, so that no two decoders are off by more than one frame.

4.3. Decoder State Propagation

Within a slice, the DC coefficients and motion vectors of a macroblock are predicted from those of the last macroblock. Because each decoder might only get a portion of an original slice, the splitter needs to send these predictors to the decoder so that it can decode the partial slice correctly.

To propagate the information efficiently, we create a *State Propagation Header* (SPH) for sub-picture bit streams. When two adjacent macroblocks in a sub-picture are not from the

same slice, we insert an SPH between them. The header contains the macroblock mode, current DC coefficients, motion vector predictors, and macroblock address increment.

As mentioned in the MPEG-2 overview, a macroblock does not necessarily start or end at byte boundaries. To avoid costly bit shifting operations for realigning partial slices, we copy all the bytes that contain a partial slice from the original stream, and specify in SPH how many bits (0 to 7) should be skipped at the beginning, see Figure 4. Although SPH and the unused bits increase the size of bit stream, every row of macroblocks in a sub-picture needs only one header. We show in the evaluation section that the overhead introduced is small.

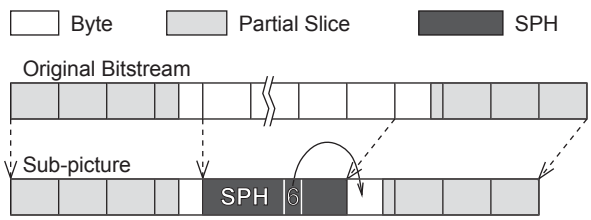

Figure 4. Partial Slices in a Sub-Picture

4.4. Zero-Copy Data Transfer

We use the GM library [11] over Myrinet [4] for fast user level communication. To avoid memory copies in sending and receiving network messages, we design the protocol such that the receiver always posts receive buffers before the sender sends any data. A receiver can be either a second-level splitter or a decoder, and a sender is either the root splitter or a second-level splitter, respectively.

We use two receive buffers to implement a simple flow control. Initially, a receiver posts two receive buffers. After it receives and consumes a message, the receiver recycles the previous receive buffer and sends an ack/go-ahead message to the sender to indicate that a new receive buffer is available. A sender first performs its work. Then, except for the first time, it waits for an ack from the receiver before sending the data. As we can see in Figure 5, this method eliminates memory copy and minimizes the time spent on blocking receives.

4.5. Ordering of Pictures

Because the GM library does not maintain the order of messages from different senders, we design a protocol to keep the proper ordering of pictures arriving at the decoders. A naive solution of using frame number requires the decoders to keep a queue of incoming pictures and reorder them. Instead, we make use of the ack/go-ahead messages in our protocol to eliminate queuing.

When multiple splitters exist, we re-direct a decoder's ack message to the next splitter instead of the sender of the picture. Consider k splitters. When splitter a sends a picture to the decoders, it also sends the node id of the splitter responsible for the next picture, i.e., $b = (a + 1) \bmod k$. We call it *ack-node-id* (ANID). When a decoder receives a message from any splitter, it first extracts the ANID; then instead of sending the ack to the sender, it sends the ack to node ANID. This allows splitter b to send the next picture.

To make the number of second-levels splitters flexible, we hide the information about second-level splitters from each other. Instead, the root splitter will send the next splitter id (NSID) along with the picture data to a second-level splitter.

Table 3. Refined Algorithms

```
Root splitter:
a = 0
While there are more pictures
   Copy a picture P to send buffer
   Wait for ACK from any splitter, except for the first picture
   Send P to splitter a, with NSID = (a+1) % k
   a = (a+1) % k
Second-level Splitter:
Post two receive buffers for incoming messages
While there are more pictures
   Recycle the previous receive buffer
   Receive picture P from root, with NSID = b
   Send ACK to root
   For each macroblock M
      For each decoder D(i)
         If M lies in the rectangle of D(i)
            Append M to buffer SP(i)
         If M references data at (x,y) of D(j)
            Append SEND(x,y,i) to MEI(j)
            Append RECV(x,y,j) to MEI(i)
   Wait for ACK from all decoders, except for the very first
   picture in a stream
   Send MEI's and SP's to D's, with ANID = b
Decoder:
Post two receive buffers for incoming messages
While there are more pictures
   Recycle the previous receive buffer
   Receive MEI and SP from splitter, with ANID = b
   Send ACK to splitter b
   Execute SEND instruction in MEI
   Decode SP; get remote macroblocks according to RECV's
   Display the picture
```

Table 3 lists the refined algorithms for the root splitter, the second-level splitters, and the decoders. Figure 5 shows the flow of work units and messages in a system where $k = 2$. Since message exchanges among decoders are irrelevant, we use one column for all the decoders.

4.6. Configuration Determination

Currently, we determine the configuration of the 1-k-(m,n) system empirically, based on the video stream resolution.

We determine m and n by matching the video resolution with the resolution of a tiled display wall. For example, a 3840×2800 video stream would require $m = 4$ and $n = 4$ if the resolution of each tile is 1024×768.

We determine k by matching the speed of splitters and decoders. Suppose it takes t_s to split a picture at macroblock

level, and t_d to decode and display a sub-picture, the overall frame rate is given by the following formula:

$$F = min(k/t_s, 1/t_d)$$

When $t_s > k \cdot t_d$ the splitters are the bottlenecks in the system, and the decoders are not running at full speed. When $t_s \le k \cdot t_d$ the decoders are running at full speed. Therefore the optimum value of k is $\lceil t_s/t_d \rceil$. If this value equals 1, we can save the second-level splitter by using a 1-(m,n) system.

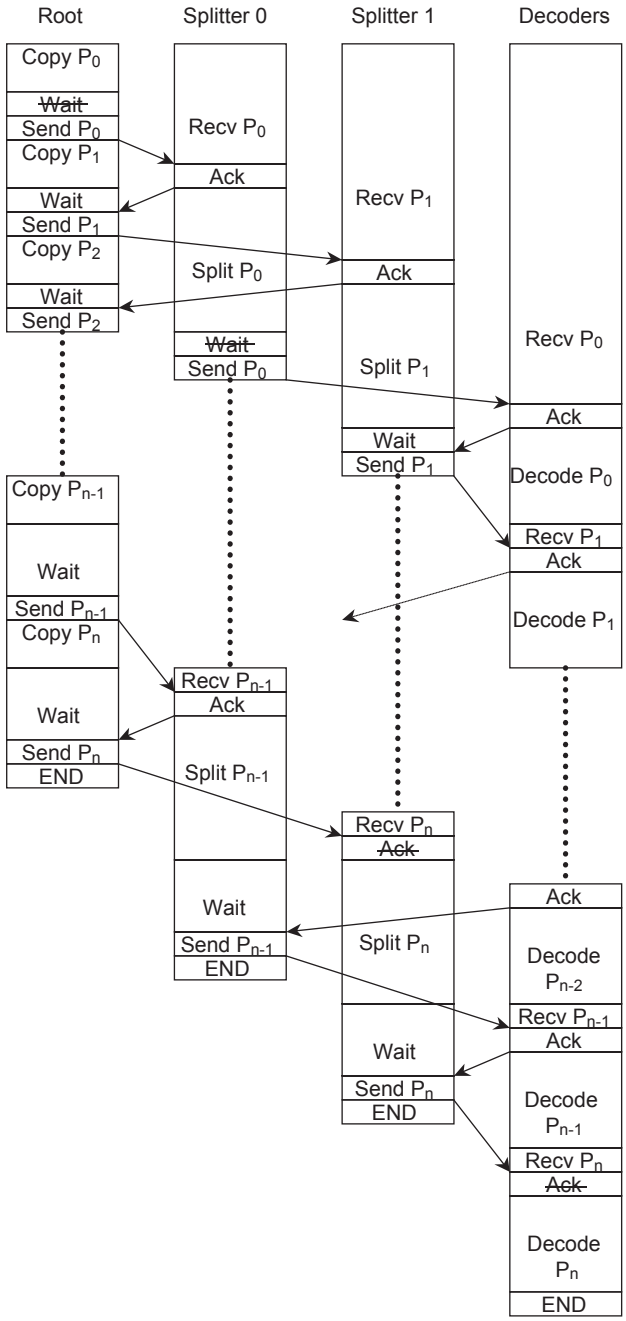

Figure 5. Flow of Work Units and Messages in a Two-Level System

5. Experiments and Results

We implemented the hierarchical parallel MPEG-2 decoder and evaluated its performance on the Princeton Scalable Display Wall system. We focus on these four questions:

- When do we need two-level splitting?
- For a given video stream, does the decoder scale in performance with increasing number of PCs in the cluster?
- Does the decoder scale with increasing video resolution?
- How much communication bandwidth is required and is that well balanced among all the nodes?

5.1. Test Platform

The display wall system consists of 24 DLP projectors in a 6×4 configuration with an $18' \times 8'$ rear projection screen. The resolution of each projector is 1024×768, and there is roughly a 40 pixel overlap between adjacent projectors for edge blending. The total resolution is about 6000×3000. The display wall has 25 PCs in a cluster connected by Myrinet. The console PC has a 550 MHz Pentium III processor and 1 GB of SDRAM. Each projector is driven by a PC workstation with a 733 MHz Pentium III processor, 256 MB of RDRAM, and an NVIDIA GeForce2 GTS graphics card.

Our experiments used a fraction of the display wall system. We used the console PC as the root splitter, up to 5 PCs as second-level splitters, and up to 16 PCs as decoders. The screen configurations in our experiments include 1×1, 2×1, 2×2, 3×2, 3×3, 4×3, and 4×4.

Because of the overlapping regions between adjacent projectors, some macroblocks are sent to multiple decoders. This causes some overhead, especially for low resolution videos, as we will notice in the evaluations.

Table 4. Characteristics of Test Video Streams

Stream	Name	Resolution	Average Frame Size (Byte)	Bit Per Pixel
1	spr	720 x 480	28873.75	0.668
2	matrix	720 x 480	25032.65	0.579
3	t2	720 x 480	32810.20	0.759
4	anim7.5	1000 x 750	31734.24	0.338
5	fish2	1280 x 720	35223.68	0.306
6	fish3	1280 x 720	35185.45	0.305
7	fish6	1280 x 720	35168.18	0.305
8	fish8	1280 x 720	35239.03	0.306
9	fox5	1280 x 720	30925.03	0.268
10	nbc4	1920 x 1080	74249.08	0.286
11	cbs3	1920 x 1080	75261.80	0.290
12	anim30	2000 x 1500	125693.42	0.335
13	orion40	2880 x 1440	166852.45	0.322
14	orion60	2880 x 2160	250388.21	0.322
15	orion80	3840 x 2160	333758.54	0.322
16	orion100	3840 x 2800	416968.77	0.310

5.2. Test Video Streams

We used 16 MPEG-2 video streams with resolutions ranging from DVD to near IMAX to test the performance and scalability of the system, see Table 4. Stream 1 to 3 are clips from the movie Saving Private Ryan, The Matrix and Terminator 2, respectively. Stream 4 is a scene from a short animation made by Adam Finkelstein. Streams 5 through 8 are shots of a fish tank taken with an HDTV video camera, courtesy of Intel MRL. Stream 9 is a video clip recorded from the HDTV broadcast of FOX5 in 720p format. Streams 10 and 11 are clips recorded from NBC4 and CBS3 in 1080i format respectively. Stream 12 has the same content as stream 4 but is rendered at quadrupled resolution. Streams 13 through 16 are compressed from results of fly-through visualization of the Orion Nebula, courtesy of UCSD supercomputing center.

All streams except the first three have about the same bit rate per pixel, 0.3 bpp. This translates to a bitstream rate of about 20 Mbps for HDTV stream 720p at 60 fps, 1080i at 30 fps, and about 100 Mbps for the highest resolution Orion flyby sequence at 30 fps. The first three streams are compressed for DVD, and have a higher bit rate. Each sequence is trimmed to contain 240 frames.

5.3. Performance of One-Level Splitting

This experiment shows when the splitter in a 1-(m,n) system becomes a bottleneck. We played stream 1 (DVD) and stream 8 (720p HDTV) on a one-level system with screen configurations ranging from 1×1 to 4×4. The frame rates are listed in the left half of Table 5. A plot of frame rate versus total number of nodes ($1 + m \cdot n$) are shown as the dashed lines in Figure 6.

We can see that when the number of decoders is greater than 4, the splitter can not keep up with the decoders and becomes a bottleneck. When the number of decoders further increases, the frame rates drops slightly for reasons that will be explained in the next subsection.

Table 5. Frame Rate of One-Level and Two-Level Systems

One-Level System				Two-Level System			
Stream 1		Stream 8		Stream 1		Stream 8	
config	fps	config	fps	config	fps	config	fps
1-(1,1)	97.8	1-(1,1)	49.9	1-(1,1)	97.8	1-(1,1)	49.9
1-(2,1)	160.3	1-(2,1)	90.7	1-(2,1)	160.3	1-(2,1)	90.7
1-(2,2)	242.2	1-(2,2)	153.2	1-2-(2,2)	245.1	1-2-(2,2)	155.4
1-(3,2)	260.2	1-(3,2)	142.4	1-2-(3,2)	292.0	1-2-(3,2)	191.1
1-(3,3)	243.8	1-(3,3)	133.2	1-2-(3,3)	330.0	1-3-(3,3)	242.3
1-(4,3)	231.4	1-(4,3)	125.0	1-3-(4,3)	360.3	1-4-(4,3)	274.8
1-(4,4)	219.8	1-(4,4)	116.6	1-3-(4,4)	398.4	1-5-(4,4)	315.9

5.4. Two-Level Splitting Frame Rate Scalability

In this experiment, we try to remove the bottleneck with second-level splitters. We determine k by increasing it until the overall frame rate stops increasing. The results are shown in the right half of Table 5. The plots are shown as the solid lines in Figure 6, with number of nodes being $1 + k + mn$. The bottleneck is clearly removed, and the system achieves good frame rates as the number of decoders increases.

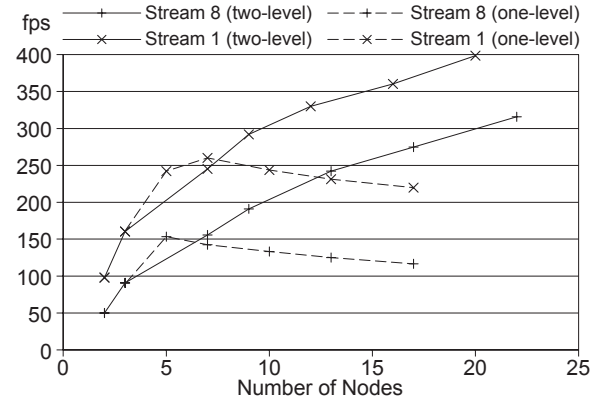

Figure 6. Frame Rate of One-Level and Two-Level systems

However, the acceleration is less than linear. This occurs for the same reason that the frame rate of a one-level system drops after saturation. Given a fixed resolution video stream, each decoder is responsible for less macroblocks when the number of decoders increases. Thus, the percentage of macroblocks that reference remote blocks increases. As a result, the decoders spend more time in fetching remote blocks. To illustrate this, we profile the decoders in both 2×2 and 4×4 settings for stream 8 and break down the runtime into five parts:

- Work: the time to decode and display a picture
- Serve: the time to prepare data for remote decoders
- Receive: the time waiting for sub-picture from splitters
- Wait: the time waiting for remote blocks
- Ack: the time to send acks to splitters

Figure 7 shows the runtime breakdown of each decoder and their average for both 2×2 and 4×4 setups. We can see that while about 80% of the runtime is spent in decoding in a 1-2-(2,2) system, only about 40% of the runtime is spent in decoding in a 1-5-(4,4) system. The percentage of serving remote decoders increases significantly because more macroblocks reference remote blocks. Also, increased contention in the network causes the receiving time to increase slightly.

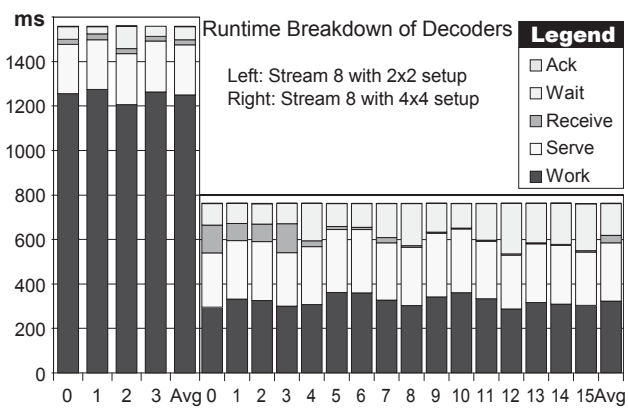

Figure 7. Runtime Breakdown of Decoders

As stated before, frame rate scalability is not critical once we achieve the real time frame rate. The more important question is whether the system can decode higher resolution streams with more decoders.

5.5. Two-Level Splitting Resolution Scalability

To test the resolution scalability of our two-level system, we run each of the 16 streams with an appropriate number of decoders such that the video resolution matches the screen resolution. As in 5.4, the number of second-level splitters is chosen to keep the decoders running at full speed. Table 6 shows the screen configuration, frame rate, and the rate of total decoded pixel for each stream. The apparent drop of frame rate (after stream 10) is due to the increased number of pixels per decoder.

Table 6. Frame Rate of All Streams in Two-Level System

Stream	Config	num of nodes	Frame Rate (fps)	Pixel Rate (Mpps)
1	1-(1,1)	2	97.8	33.8
2	1-(1,1)	2	105.0	36.3
3	1-(1,1)	2	99.8	34.5
4	1-(2,1)	3	102.1	76.6
5	1-(2,1)	3	94.6	87.2
6	1-(2,1)	3	87.4	80.5
7	1-(2,1)	3	89.2	82.2
8	1-(2,1)	3	90.7	83.6
9	1-(2,1)	3	105.0	96.8
10	1-2-(2,2)	7	88.8	184
11	1-2-(2,2)	7	74.0	154
12	1-2-(3,2)	9	60.2	181
13	1-2-(3,2)	9	45.3	188
14	1-3-(3,3)	13	43.0	268
15	1-3-(4,3)	16	38.9	323
16	1-4-(4,4)	21	38.9	418

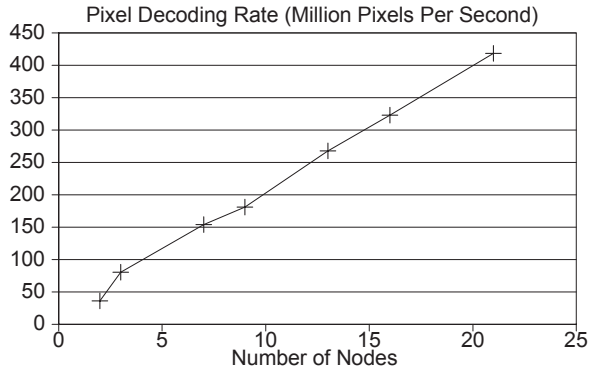

Figure 8. Pixel Decoding Rate of Two-Level System

Figure 8 shows a plot of pixel decoding rate versus number of nodes. When multiple streams exist for one configuration, we use the average value. The two-level system achieves a near linear acceleration, and scales well.

There is a slight drop of performance for the four highest resolution videos. We notice that in these videos, the majority of motion and visual details are located in a portion of the entire screen. Because an MPEG-2 video encoder allocates bits according to the scene complexity within a picture, in a tile containing complex motion and detail, the bit-rate of the decoder is much higher than that of a decoder with less motion and detail. When the decoders are synchronized, the overall frame rate is determined by the slowest decoder.

5.6. Bandwidth Requirement

In this experiment, we measure the send and receive bandwidth of each decoder and splitter in a 1-4-(4,4) system decoding stream 16. The results are shown in Figure 9. We can see that even for an ultra-high-resolution video with localized detail, the communication requirement is still low and balanced. It is well within the range of current commodity network technologies. The SPH headers in sub-pictures cause the send bandwidth of a splitter to be larger than its receive bandwidth. However, the overhead is only about 20%.

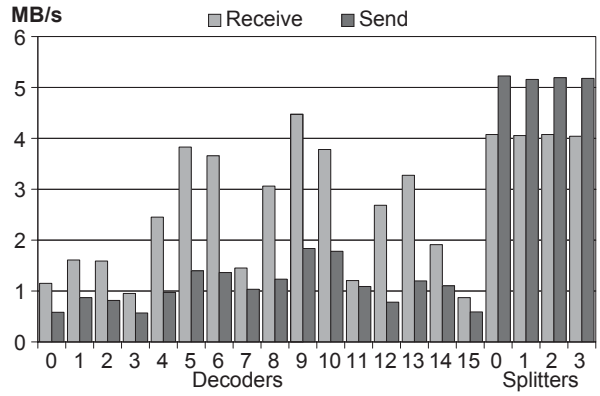

Figure 9. Send and Receive Bandwidth of Each Node in a 1-4-(4,4) System Decoding Stream 16

6. Conclusion and Future Work

This paper presents a hierarchical parallel MPEG-2 video decoder for ultra-high-resolution video streams on a PC cluster based tiled display system. The approach combines picture level splitting with macroblock level splitting to avoid the splitting bottleneck in a parallel decoder. Our experiments on a display wall system shows that our decoder has high performance and is highly scalable. It not only supports very high frame rate decoding of normal resolution video, but also can achieve real time frame rate for ultra-high resolution video streams.

Because of the low bandwidth requirement, we expect our system to perform well beyond the scales and resolutions reported in this paper. This can be useful for scientific visualization, and immersive and collaborative environments.

Several aspects of the parallel decoding can be further studied and improved. First, each of our graphics card drives

a single projector. It would be useful to experiment with graphics cards that can drive multiple displays to further evaluate the performance.

Second, the decoding system currently balances workloads statically. We expect that a dynamic load-balancing method can help the splitter distribute work more evenly to fully utilize the decoders.

Finally, the configuration of the system is currently chosen empirically for each sequence to maximize the performance. This is an trial and error process. We could make the root splitter automatically choose the number of splitters when a target frame rate is given. Together with dynamic loading balancing, this can make the system more flexible.

7. Acknowledgements

This project is supported in part by Department of Energy under grant ANI-9906704 and grant DE-FC02-99ER25387, by Intel Research Council and Intel Technology 2000 equipment grant, and by National Science Foundation under grant CDA-9624099 and grant EIA-9975011. Han Chen is supported in part by a Wu Fellowship. An early stage of the work was supported in part by AT&T Labs Research.

8. References

[1] S. M. Akramullah, I. Ahmad, and M. Liou. A Data-Parallel Approach for Real-Time MPEG-2 Video Encoding. *Journal of Parallel and Distributed Computing*, 30(2):129-146, Nov. 1995.

[2] A. Bala, D. Shah, U. Feng, and D. K. Panda. Experience with Software MPEG-2 Video Decompression on an SMP PC. *Proceedings of the 1998 ICPP Workshop on Architectural and OS Support for Multimedia Applications/Flexible Communication Systems/ Wireless Networks and Mobile Computing*, pp29-36, 1998.

[3] A. Bilas, J. Fritts, and J. P. Singh. Real-Time Parallel MPEG-2 Decoding in Software. *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, Apr. 1997.

[4] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE MICRO*, 15(1):29-36, February 1995.

[5] A. Chimienti, M. Lucenteforte, D. Pau, and R. Sannino. A Novel Co-decoding Scheme to Reduce Memory in MPEG-2 MP@ML Decoder. *1998 URSI International Symposium on Signals, Systems, and Electronics*, pp272 -277, 1998.

[6] S. Eckart and C. E. Fogg. ISO/IEC MPEG-2 Software Video Codec. *Proc. Digital Video Compression: Algorithms and Technologies 1995*, 100-109, SPIE, 1995.

[7] M. Hereld, I. R. Judson, R. L. Stevens. Introduction to Building Projection-based Tiled Display System, *IEEE Computer Graphics and Applications*, Vol. 20, No. 4, pp22-28, July/August 2000.

[8] M. K. Kwong, P. T. Peter Tang, and B. Lin. A Real Time MPEG Software Decoder Using a Portable Message-Passing Library. *Mathematics and Computer Science Division ANL, Preprint MCS-P506-0395*, April 1995.

[9] W. Lee, G. J. Golston, and Y. Kim. Real-time MPEG Video Codec on a Single-Chip Multiprocessor. *Proc. of the SPIE, Conference on Digital Video Compression on Personal Computers: Algorithms and Technologies*, 2187:32-42, Feb. 1994

[10] K. Li, H. Chen, Y. Chen, D. W. Clark, P. Cook, S. Damianakis, G. Essl, A. Finkelstein, T. Funkhouser, T. Housel, A. Klein, Z. Liu, E. Praun, R. Samanta, B. Shedd, J. P. Singh, G. Tzanetakis, and J. Zheng. Building and Using a Scalable Display Wall System, *IEEE Computer Graphics and Applications*, Vol. 20, No. 4, pp29-37, July/August 2000.

[11] Myricom Inc. GM Library Reference, http://www.myricom.com/scs/GM/doc/gm_toc.html

[12] K. Patel, B. C. Smith, and L. A. Rowe. Performance of a Software MPEG Video Decoder. Proc. *ACM Intl. Conf. On Multimedia*, 75-82. ACM, Aug. 1993.

[13] W. Yang, H. Kim, M. Shin, I. Park and C. Kyung. A Multi-Threading MPEG Processor with Variable Issue Modes. *The 6th International Conference on VLSI and CAD*, pp545 -548, 1999.

[14] Y. Yu, and D. Anastassiou. Software Implementation of MPEG-2 Video Encoding Using Socket Programming in LAN, *Proc. of the SPIE, Conference on Digital Video Compression on Personal Computers: Algorithms and Technologies*, 2187:229-240, Feb. 1994.