Parallel JPEG Algorithms for Still Image Compression

Peter Monnes
IBM Personal Computer Company
Boca Raton, Florida 33431, U.S.A

Borko Furht
Computer Science and Engineering
Florida Atlantic University
Boca Raton, Florida 33431, U.S.A

Abstract: **One of the hottest areas of development in the computer industry today is motion video on the desk top. Both motion video and still image video pose some new problems: huge storage and high bandwidth are required to send, view or store video data in a timely fashion. Compression techniques may help to reduce the amount of data by trading processing power for reduced data size. For both video compression and decompression, very large amounts of processing power are required to handle the video data quickly enough. Parallel processing is being used in some cases to solve this need for processing power.**

This paper will explore some of the parallel processing techniques currently being used and will explore some additional possibilities for parallelization. It will focus primarily on JPEG but many of the concepts are applicable to other compression schemes. Results from a simple PC based JPEG compression/decompression program will be used to explore some of the techniques.

1. Introduction

JPEG (Joint Photographic Experts Group) is an international standard for color image compression. It began as 10 independent competing proposals in 1987 and was finally adopted by CCITT in 1992. It provides a wide range of capabilities. It has "baseline" definition, those functions which every JPEG implementation must have. It also defines extended capabilities, lossless processes, progressive processes and hierarchical processes. See fig. 1 for a review or [5] or [10] for more detail on the standard. This paper will discuss only the baseline definition, since most mainstream implementations do not go beyond baseline capability.

2. Description of the Problem

Ideally, a computer user would receive a small data file which contains an image or video clip. A software application would be used to view and edit the image, possibly pulling in other image, video or graphics objects to add to the original in some way. The resulting file would be compact enough to be quickly sent on a network or phone line or stored economically on a disk. All the operations should be done quickly and easily and without special hardware.

There are some technical problems to be solved before this ideal situation can be attained. This paper assumes a target of 1/10 second response time to display a 320x240x24 bit per pixel image. A 10:1 compression ratio will be assumed to be adeqate in terms of quatity and compression rate for many applications. This is still an order of magnitude beyond what the best JPEG algorithms for personal computers are capable of today. Parallel processing is a possible solution.

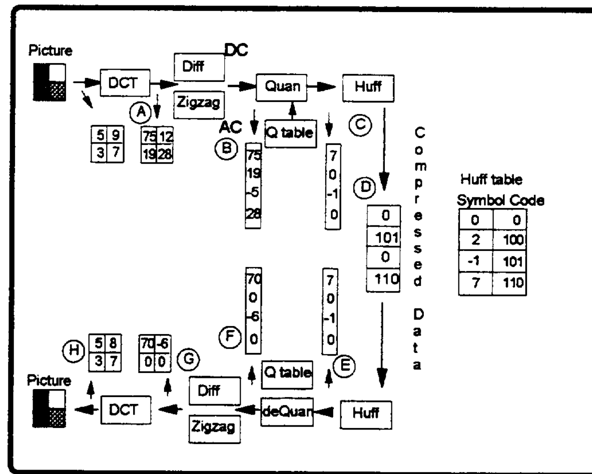

Fig. 1. Simplified JPEG Example.

3. Parallel JPEG Techniques

3.1 Overview of Techniques for Parallel JPEG

One approach to parallel processing of DCT based compression algorithms seems to be predominant in the literature. For compression, one or more blocks are handled by individual processors in parallel. At the end, a serial process performs the inherently serial portion of the job: Huffman encoding and writing to a file. For decompression, the serial process Huffman decodes the file and farms the blocks to individual processors. The processors do the inverse DCT and de-quantization. Finally, the decompressed blocks are placed into a data file or onto the display screen. Examples of this theme can be found in [3], [8], [16] and [17]. Figure 2 presents an overview of this type of architecture.

On most implementations, the parallel portions can achieve sufficient speed up so that the serial portion becomes the bottleneck. Thus, simply adding more processors will only go so far in solving the

0-7803-1797-1/94/$3.00 © 1994 IEEE

problem. The serial portion of the algorithm must be sped up by either a faster processor, or by some scheme to parallelize it as well.

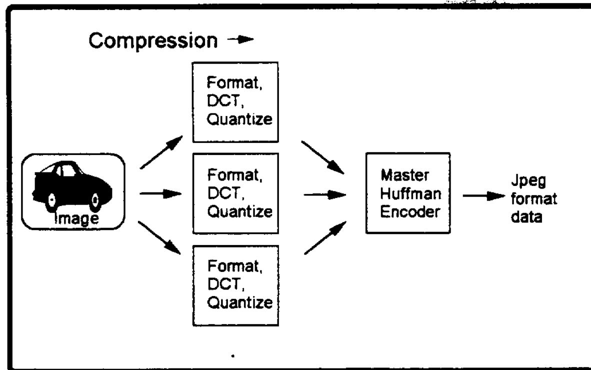

Fig. 2. Parallel architecture for the JPEG algorithm.

3.2 Description of Experiment

Rather than perform a parallel simulation for the entire JPEG algorithm, a paper analysis was performed. In order to get some representative numbers with which to work, a serial compression program and a serial decompression program were written in C and run on a personal computer. A fast DCT algorithm derived by E. Feig was incorporated. A compression ratio of 10:1 was achieved by using a quantization factor of 10. Figure 3 shows the original and final pictures. Any artifacts are due to the scaling to fit into this article.

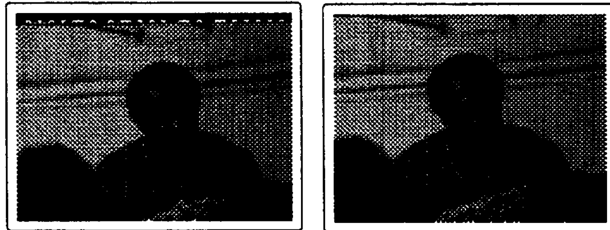

Fig. 3. Original and Final images(compressed then decompressed)

The programs were run on both an i80286 machine and an i80486 33 MHz machine. The i80486 was chosen to get an overall performance number representative of today's available general purpose mainstream microprocessor. The i80286 was chosen because it ran the program slowly enough to be able to measure individual subroutines with the system timer. The numbers were run on a 24 bit per pixel 320x240 Targa image, compressed then decompressed to a 24 bit per pixel 320x240 BMP image. The 6.9 seconds to compress and 5.7 seconds to decompress the image on the faster processor still require a speedup of at least 100. A speedup of 100 will be the goal of this exercise, independent of the processor.

3.3 Frames in Parallel

Multiple frames may be processed concurrently as suggested for DVI in [17]. This technique is referred to as "concurrent streams" in [6]. To get a speed up of 100, simply line up 100 processors and process 100 frames in parallel. Each frame is done completely on one processor. See figure 4. The software used to obtain the numbers in table 1 could be used to decompress concurrent streams with only slight modifications to its I/O routines. The first processor starts with the first frame, the second processor starts with the second and so on. Once the first frame is complete, the other 99 are at or near completion. The first processor then begins work on the 101st frame etc. Subsequent frames will then be completed on an average of 100 times faster than with a single processor. This does nothing to help speed up a single image, and even with motion JPEG sequences, the user must wait a while for the first frame.

Compression		Decompression
5.3	Read File I/O Write	2.4
9.8	YUV Conversion	8.0
2.0	Reorder from/for YUV conv	3.4
20.1	FDCT/IDCT	20.3
3.3	(de)Quantization	2.6
6.1	Huffman encoding/decoding	4.9
0.1	Write File I/O Read	0.4
46.7		42.0

Table 1. Time in seconds for execution of each step for JPEG. Compression times are in order listed, decompression is in reverse.

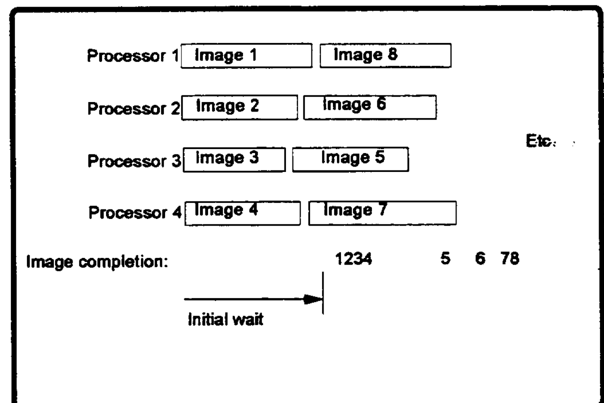

Fig.4. Image Level Granularity

Even though this method can achieve the desired speedup and would be straight forward to implement, it does not solve the problem of a long wait to decode a single JPEG image. This approach would require enough storage for each processor to buffer a frame of input and a frame of output. This could add up to a large amount of storage and thus be expensive to implement. For these reasons, another approach is needed.

4. Parallelization of One Frame

In order to quickly process a single image, some form of parallelization within the frame will be necessary. Since JPEG images are constructed of 8x8 pixel blocks, this is the obvious place to begin. A nice feature of JPEG is that the 8x8 blocks are almost independent. That is, each block can be encoded or decoded without knowledge of information in the other blocks. Thus an array of processors, potentially up to the number of blocks in the image, could each be assigned one block to (de)compress.

First the image or compressed data must be input. It is assumed for this paper that the input and output mechanisms are adequate for whatever speed is required by the JPEG algorithm. Second, the format of the image must be converted to the desired transmission format. JPEG doesn't dictate any particular image format, but typically the input is not in the format the user wants to send. The sample code used for this paper did an RGB to YUV conversion. This conversion takes a healthy chunk of time and may also need to be parallelized. This should be straight forward to parallelize and will not be addressed in this paper.

4.1 Encode

Reading the Image

Once the uncompressed image is loaded, individual processors may be assigned one or more 8x8 blocks to compress. This could be done by accessing the data from a shared memory or by passing the data in messages. An interleaved memory design could reduce a possible memory contention delay. Or, individual processors could read data in some staggered fashion so that memory access is balanced over time. In this way, the demand at the peak memory usage time could be reduced.

Passing the data to individual processors in messages requires one or more processors to read the data and send it to the other processors. A group of processors could read a shared memory in the fashion described above, and send it to other processors to actually do the encoding.

Parallel JPEG Encode

At this point, each processor has a portion of the image, and has it converted to the desired color space format. Since each block is independent, the individual processor can now perform the DCT, the quantization and Huffman encoding on its own data without any interplay with other processors.

Parallel Huffman Encode

The Huffman encoding step presents some complications. Huffman is a form of variable length coding, which means that the resulting length of encoded bits of some data is dependent on the data. This means that the Huffman encoded data from each processor will be some arbitrary length in bits. Statistically, in 7 of 8 cases, it will not align on a byte boundary. Each processor will need to know the alignment of the resulting bit stream of all of the processors before it in the output stream. It can then properly shift its output before the output is combined with the other processors' output. A similar concept applies to the DC difference coding. Each processor needs to know the value of the DC difference value of the processor before it. The processors could broadcast their alignment and DC values as soon as they finish coding their streams. Since each will finish at different times due to the variable length nature of the Huffman coding, the communication bottle neck should be minimized. This information could be conveyed either by messages or by storage in shared memory.

Writing the Data

Once each processor has the data properly aligned and coded, the data needs to be combined and either stored or transmitted. In most architectures, one processor has I/O responsibility. All of the other processors send their data to it in turn.

4.2 Decode

Reading the Compressed Data

In the next section, it will be shown that the Huffman decode is essentially a serial process. That means that one processor will be responsible for decoding the entire image. This makes reading the image trivial. One processor gets and decodes all of the data.

Parallel Huffman Decode

The most difficult part of parallel JPEG is in parsing a compressed image. There is one major complication, finding the 8x8 blocks. Since the data has been run length encoded (Huffman is a type of run length encoding), each block's length is variable. The length depends on the nature of the data within the block. It is not possible to index directly to a particular block. The data must be Huffman decoded up to the point where the block begins in order to know where the block is. This is a potential bottleneck, if one processor must either decode the entire image or must orchestrate the decoding of it by other processors.

One solution is to use restart markers as allowed by the JPEG standard. Restart markers would be inserted into the data stream and would break up the encoded data into smaller units on block boundaries. They are numbered modulo 8 and would greatly assist in the parsing of the image. Since they are not themselves encoded, the image can be easily scanned for them prior to any decoding. Now, multiple processors could begin scanning the input data at various points, looking for one of the modulo 8 markers. In this way the decoding can be done in parallel. The image is broken up into as many subsection as there are processors available for parsing. Each processor begins scanning its subsection for a marker. It begins Huffman decoding when it finds a marker, it stops decoding when it finds the first marker after its subsection. Figure 5 illustrates this idea.

Restart markers are not the cure all. Not all implementations use restart markers. Additionally, there is no rule on how often restart markers must be inserted into the image. It may be that they are too sparsely spaced to give the desired parallelism.

Another possible solution is to use the 'bit positioning' method described in [6]. This method can be used with any number of

processors and can achieve any level of parallelism. Unfortunately, it has high computational complexity.

In bit positioning, each processor begins decoding at some interval in the Huffman bit stream, similar to the restart method above. The decoding begins in the overlapping region (see figure 6) and starts decoding the stream. The size of the overlapped region is the length of the longest Huffman code. Since the beginning of the overlapped region might not fall on the boundary of a code, every bit in the overlap region could potentially be the first bit in the first code in the overlap region. So that all possible decodes are covered, the decode must be done several times, each time beginning with a different bit in the overlapped region. In the code used for this paper, as in the JPEG recommended implementation, the maximum code length was 16. In this case the interval would have to be decoded 16 times!

Once the processors progress in decoding one or more intervals, the 'stitching' together of the results begins. This stitching can begin between any two processors with adjacent decode intervals. The processors pair their decode strings. Those strings on one processor which end in the trailing overlapped region one bit before the string which begins in the leading overlapped region of the next processor are paired. Any strings which do not match any other are discarded. Ultimately, this hierarchical technique selects only one stream as the correct stream.

Bit positioning doesn't work for the following reason. A JPEG bit stream is made up of DC components intermixed with the AC components; chrominance components combined in the same MCUs with luminance components. All of these components use different Huffman tables. This makes bit positioning unworkable. Any bits which are to be decoded could possibly be found in one of four tables. Which table exactly is not known until the exact position in the bit stream is known. And the position in the bit stream is not known until the stream is decoded from the beginning.

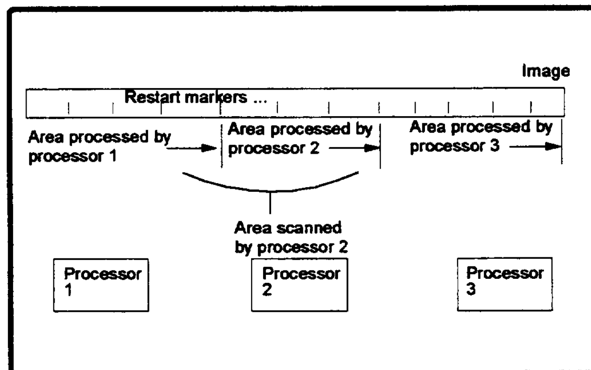

Fig. 5. Parsing with restart markers

The lack of any method to decode the Huffman bit stream in parallel is the Achilles heel of parallel JPEG. Without a parallel Huffman decode technique, a single processor must Huffman decode the entire image and then parcel out the blocks to the other processors for further JPEG decode. The Huffman decode would then become the limiting factor as shown by Amdahl's law. Focusing only on the steps which are part of the JPEG standard, Figure 7 demonstrates Amdahl's Law. The total serial time includes only IDCT time plus de-quantization time plus Huffman decode time.

Clearly, to achieve a 100x speedup, the Huffman portion of the algorithm must be parallelized as well as the other parts of the algorithm.

One additional idea related to Huffman decoding is worth mentioning. All of the data compression in JPEG actually occurs in the Huffman stage. The DCT and quantization simply prepare the data for more efficient Huffman coding. Once the data is Huffman decoded, the amount of data is equal to the full image size (except for a possible color space conversion, which is not part of the JPEG standard). This size of data might be a problem to send around to the individual processors for the later JPEG steps.

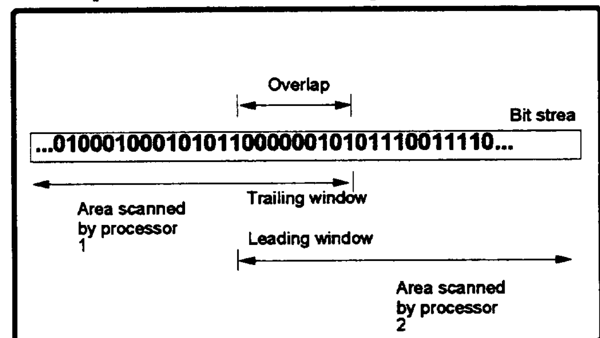

Fig. 6. Parsing with bit positioning method

I propose that the single processor which does the Huffman decode not really do the decode, but simply walk the bit stream. It would sift through the bit stream, find the MCU and block boundaries and then send the compressed data to the individual processors. The individual processors would each have to decode its own Huffman stream in parallel. The extra Huffman decode time on the individual processors would be small, and should be more than compensated for by a much reduced data transmission bottleneck. This method also saves on buffer size on the single Huffman processor, since it does not really decompress all of the data.

Speedup=T/Tn

Tn=A*T+((1-A)*T/N)

Where T is the total serial execution time

Tn is the total time with parallelization
A is the proportion of non-parallel code
N is the number of processors

From table 1, for decompression, T=27.8,
Huffman decode time = 4.9 A=4.9/27.8=0.18

For 100 processors: Tn=0.18*27.8+((1-0.18)*27.8/100)=5.13

Speedup=27.8/5.13=5.42

Fig. 7. Amdahl's Law and speedup

Parallel JPEG Decompression

Once the processors are allocated a block or blocks to decode, things are relatively straight-forward. Except for the DC coefficients, the blocks are independent. Each processor can proceed to perform the de-quantization, DCT and any reformatting of the data required (e.g. YUV to RGB etc.). For these steps, with no interaction with other processes, it is expected that a near ideal speed up could be achieved.

The DC coefficients cause some inherent serialization. Since they are difference coded, the value of the first DC component is required to find the second, the second required for the third etc. With restart intervals, the difference coding resets at every restart interval, and is not a problem.

In case restart markers are not used, the DC components would be decoded during the serial Huffman decode. As each DC component is decoded, its value is calculated based on the previous DC value. Either way, DC components are not a problem for decode.

Outputing the Data

The completed, decompressed data is then ready of output. It is either placed in shared memory or sent to the processor responsible for I/O. Some bottlenecks may occur here, since the individual processors are all handling similar tasks, and no variable length decoding. This is not a problem. Since the data would normally be Huffman decoded in a serial manner, it would be sent to individual processors in a staggered sequence as it is decoded. A consequence is that the data would be correspondingly staggered upon completion. The I/O processor would see the data spaced roughly at the same rate that it was sent to the individual processors

5. Application

Two possible appllications of parallel JPEG are foreseen. One is on the multiprocessor PCs which are just now appearing in the industry. To support the hardware, operating systems such as Windows NT and operating systems based on the Mach kernel which are designed to handle multiprocessor PCs are expected to be popular. A speedup of 100x is possible if a 10 processor system which has 10x the performance of an i80486 implements parallel JPEG at ideal efficiency. Some of the higher end desktop machines are already approaching this performance range. In the next few years systems of this type will be available which will be able to run motion JPEG with no special hardware.

Another application of parallel JPEG is on parallel arrays of DSPs or micro controllers. This solution would be desirable since these arrays could be used for any number of other functions.

6. Conclusion

A 100x speedup above current microprocessor performance is necessary to run JPEG 'fast enough' for today's multimedia expectations. While waiting for processors that are 100x as fast as today's, we can implement parallel algorithms. The techniques summarized in this article show how JPEG can be parallelized to achieve a moderate degree of speedup. While massive parallelism is not feasible for most applications, some implementations on multiprocessor workstations or small DSP arrays might suffice. The main technical issue to be solved is the serial nature of Huffman decoding. There is still plenty of opportunity for parallel JPEG, since the mainstream desktop processor is still 2 or 3 microprocessor generations away from running JPEG on a single processor.

REFERENCES

[1] P. Ang, "Video Compression Makes Big Gains," IEEE Spectrum, October 1991, pp 16-19.

[2] A. Antola and M. Tellarini, "Definition and Evaluation of a Transputer-based Architecture for Image Compression and Reconstruction," Microprocessing and Microprogramming, Vol. 31, No. 1-5, April 1991, pp. 127-131.

[3] D. Chen and R. Price, "A Real-Time TMS320C40 Based Parallel System for High Rate Digital Signal Processing," Proceedings - ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing, Vol. 3, 1991, pp. 1573-1576

[4] R. Cypher, J. Sanz and L. Snyder, "Algorithms for Image Component Labeling on SIMD Mesh-Connected Computers," IEEE Transactions on Computers, Vol. 39, No. 2, February 1990, pp. 276-281.

[5] JPEG Draft Specification JPEG-8-R8, August 14, 1990.

[6] H. Lin, and D. Messerschmitt, "Designing a High-Throughput VLC Decoder Par II-Parallel Decoding Methods," IEEE Transactions on Circuits and Systems for Video Technology, Vol. 2, No. 2, June 1992, pp. 197-206.

[7] V. Milutinovic and A. Car, "A Survey of JPEG Implementations," Technical Report, Institure for Advanced Computer Technology, TR-93-NCR-006, March, 1993.

[8] J. Normile and D. Wright, "Image Compression Using Course Grain Parallel Processing," Proceedings -ICASSP,IEEE International Conference on Acoustics, Speech and Signal Processing Vol. 2, 1991, pp. 1121-1124.

[9] K. Parhi, "High Speed Huffman Decoder Architectures," Conference Record of the Twenty Fifth Asilomar Conference on Signals, Systems and Computers, November 1991, pp. 64-68.

[10] W. Pennebaker and J. Mitchell, "JPEG Still Image Data Compression Standard," Van Nostrand Reinhold, 1993.

[11] R. Quinnell, "IC Acts as JPEG Image-Compression Coprocessor," EDN, March 4,1993, pp. 68-69.

[12] R. Quinnell, "Image Compression Part 2," EDN, March 4,1993, pp 102-126.

[13] S. Ramaswamy and G. Miller, "Multiprocessor DSP Architectures that Implement the FCT Based JPEG Still Image Compression Algorithm with Arithmetic Coding," IEEE Transactions on Consumer Electronics, Vol. 39, No. 1, February 1993, pp. 1-5.

[14] P. Ruetz and D. Auld, "Video Compression Makes Big Gains," IEEE Spectrum, October 1991, pp. 16-19.

[15] D. Shear, "ESN's DSP-Chip Directory," EDN, October 1, 1991, pp 104-133.

[16] F. Sijstermans, "CD-I Full-Motion Video Encoding on a Parallel Computer," Communications of the ACM, April 1991, pp. 81-91.

[17] M. Tinker, "DVI Parallel Image Compression," Communications of the ACM, Vol. 32, No. 7, July 1989, pp. 844-851.