, , 1–13 ()

A Portable Fault-Tolerant Parallel Software MPEG-1 Encoder

ISKENDER AGI AND R. JAGANNATHAN iskender@csl.sri.com

SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
Tel: 415-859-2868
FAX: 415-858-2844

Received May 1, 1991

Editor: Borko Furht

Abstract. MPEG video compression is quite difficult to achieve in real time, and hardware solutions for this problem are expensive. We present a portable, fault-tolerant, parallel, MPEG-1 encoder implemented in software. We detail the implementation strategy for the encoder and give performance results on a network of workstations and a massively parallel processor.

Keywords: MPEG encoding, parallel programming, fault tolerance

1. Introduction

The Motion Pictures Experts Group (MPEG) standard, MPEG1 [4], is widely used for video storage and transmission. MPEG video compression is difficult to achieve in real time as it is inherently computation-intensive. Although the compression scheme has hardware implementations that operate in real time (24-30 frames per second), these boards and chipsets, e.g. from Optivision, C-Cube, and LSI Logic, are expensive. Thus, there has been some interest in performing the compression in software on existing hardware (such as workstation networks and multiprocessors) by harnessing compression-inherent parallelism. In addition, software implementations can aid in the design of hardware by providing a platform for exploring implementation strategies.

The Berkeley MPEG-1 video encoder is the first parallel encoder that runs on a network of Unix workstations [2]. Both the specification and discovery of parallelism in the encoder are done explicitly by the application. Basically, the application manifests as multiple processes that interact with each other using TCP/IP-based communication and synchronization calls. The application architecture consists of a master process, several slave processes, and three other specialized processes — decoder process, combine process, and I/O process. The main source of parallelism is the simultaneous activity of the different slave processes (on different workstations). However, the encoder is not portable to different architectures such as symmetric multiprocessors or other distributed-memory parallel computers (like the CM5). Also, the encoder cannot tolerate failure of some of the workstations it runs on, nor can it take advantage of new workstations available during execution.

A version of the Berkeley encoder, hand-ported to Intel Delta and Paragon, acheives greater than 40 frames/second encoding rate [7]. However, without using a speciazed I/O scheme that we will discuss later, this encoder acheives an encoding rate of approximately 22 frames/second.

In this paper, we describe a parallel software implementation of an MPEG-1 encoder using the GLU[6] system for parallel programming, developed at SRI. We compare performance of the encoder to the Berkeley encoder on workstation networks. We also present encoder performance on the CM-5. We also show how well the encoder adapts to fluctuating processing power of a workstation network.

2. The GLU System for Parallel Programming

GLU is a system for constructing parallel applications using existing sequential code and executing them on diverse parallel computers without changing the applications themselves. Parallel computers programmable by GLU range from heterogeneous workstation clusters to shared-memory multiprocessors to massively parallel computers. Figure **??** illustrates the GLU system.

GLU embodies a very high-level approach to parallel applications programming, one that insulates the application from much of the low-level implementation details, such as partitioning, mapping, scheduling, communication and synchronization. An application in GLU consists of two parts: a set of sequential components (written in conventional languages such as C) and a dataflow-style application composition that uses the sequential components as building blocks. Parallelism is implicit in the composition of the dataflow-style application. Because the components themselves can be derived with only nominal stylistic changes to existing sequential code, much of the sequential code can be reused.

Multidimensional signal processing applications such as video encoders can be succinctly and naturally expressed in GLU. The composition language directly supports notions of causality and multidimensional data dependencies commonly used in signal processing and communications applications.

The GLU application is compiled beforehand to a specific application architecture which is then conventionally compiled to appropriate executables. The choice of application architecture has no bearing on the application itself, thus making the application portable. Parallelism latent in the application is discovered and exploited at run time using a demand-driven model of computation called *eduction* that is embodied by the application architecture.

The GLU compiler currently supports both the master/worker application architecture and the peer-based application architecture. The master/worker application architecture is well suited for applications with single program, multiple data (SPMD) parallelism, of which MPEG compression is an example.

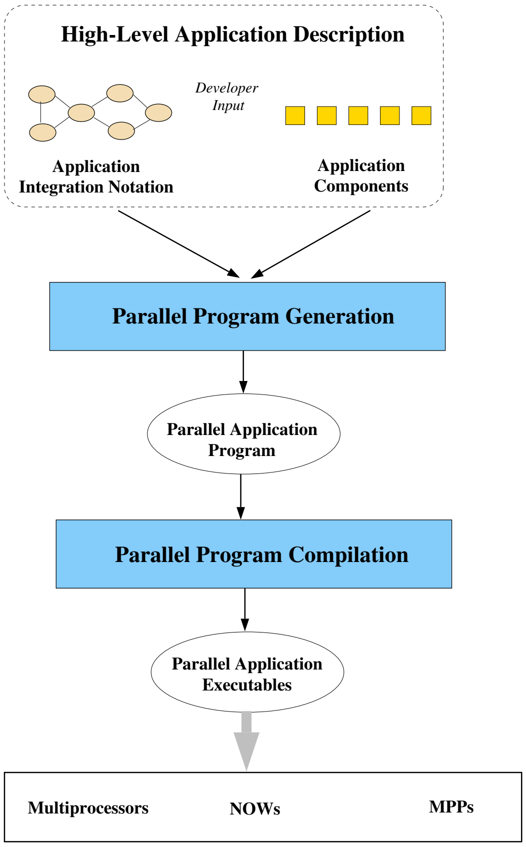

Figure 1. GLU system diagram.

2.1. GLU Master/Worker Application Architecture

The GLU master/worker application architecture consists of the master and several workers. The master interprets the composition of the dataflow-style application using eduction. In doing so, it generates executable functions, which can be executed locally, in the master itself or remotely in one of the workers. By overlapping executions of functions in different worker units, GLU exploits application-inherent SPMD-parallelism. Functions are executed locally in the master when there is no advantage to remote execution or when the function is bound to the master for reasons such as I/O.

The master also detects the entry of a new worker or the failure a worker, and acts accordingly—providing a new worker with outstanding work (in the form of a function execution) or rescheduling failed work on a remaining worker. GLU uses a dynamic load distribution scheme, which is well suited for situation when execution times of remote functions are nonuniform or when the number of workers fluctuates.

The master transports the name of a function and its arguments to a worker and the worker transports the result of function execution to the master. The manner in which master and workers communicate and synchronize depends on the target system and is transparent to the application. When the target is a network of workstations, remote procedure call (RPC) is employed; message queues are used on shared-memory multiprocessors.

3. MPEG-1 Encoding

A video sequence consists of a string of indexed frames. Each frame is a still-color image of a scene taken at a particular instant in time. The MPEG encoder takes a color video sequence, compresses that sequence using a combination of compression schemes, and produces a serial bit-stream that can be sent over a communication medium. The conversion of the three-dimensional video sequence into a one-dimensional bit stream allows the communication medium to be any of a number of possibilities including satellite, coaxial cable or even CD-ROM. An MPEG decoder receives this bit stream and decodes it to reproduce a facsimile of the original video sequence.

MPEG-1 video sequences typically use a frame size of 352×288 pixels displayed at a rate of 24-30 frames per second (FPS). Once encoded, the bit-stream of this typical video sequence is designed to reproduce a VHS-quality video at a transmission rate of 1.5Mb/sec. The MPEG encoder is not limited to these typical values and can encode larger or smaller image sizes and at varying quality settings, which affect the transmission rate accordingly.

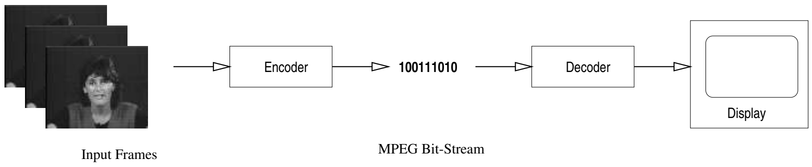

Figure 2. MPEG video codec.

While a number of decoders are available, the number of software encoders is few. Some of the encoders are written to analyze MPEG implementations[5], and only one is written to optimize speed and also run in parallel on a network of workstations. Our implementation builds on the the Berkeley encoder by adding fault-tolerance, increased portability, and modularity.

3.1. Computational Complexity

The encoder is by far the most computationally intensive part of the MPEG-1 coder/decoder. Currently, MPEG bit-streams of reasonable image sizes can be decoded in near real-time, typically 10-30 FPS, by software running on modern workstations. However, a sequential software encoder running on the same machine may take roughly two orders of magnitude longer to generate the same sequence.

MPEG encoding of a video sequence requires many compression steps. Each frame in the video sequence is compressed using a combination of block-based motion compensation to take advantage of interframe temporal redundancy, and discrete cosine transform (DCT)-based compression to take advantage of intraframe spatial redundancy [1]. Each frame (or picture) in the sequence is broken into 8×8 pixel blocks for interframe DCT compression and 16×16 pixel macroblocks for intraframe motion compensation. A 16×16 macroblock also includes two 8×8 chrominance blocks in addition to the four luminance blocks for a total of six blocks per macroblock.

A number of these blocks strung together form a slice (of a frame) that acts as a resynchronization unit. For standard size frames the number of slices per frame is one. Larger frame sizes may use more slices per frame. Then these slices combine to form a picture.

A number of these pictures are grouped together to form a random access unit so that the video can be viewed either forward or backward. This group of pictures (GOP) has little or no dependence on the pictures in adjacent GOPs. A GOP consists of three types of frames: intracoded frames (I), motion-estimated forward predicted frames (P), and motion-estimated bidirectional predicted frames (B).

I-frames are encoded block by block, without regard to previous or future frames. The encoder takes the DCT of each 8×8 block, transforming that block into its frequency-based representation. Since spatial signals have high local autocorrelation, most of the frequency information is located in the lower frequency terms. The encoder eliminates many of the higher-order terms using quantization. The use of lossy compression increases the compression tremendously. The I-frame encoding uses the same scheme as JPEG encoding of still frames. This encoded I-frame is used as the motion-estimation reference for the other forward- and bidirectional-predicted frames. Figure 3 shows the frame dependency of the various frames in the MPEG frame sequence.

Forward-predicted (P) frames are encoded with reference to the most recent previous I- or P-frame. Each macroblock undergoes motion-estimation search to find the so-called *best-fit* vector relative to the reference frame. This vector is then transmitted along with the error between the macroblock and the motion-compensated macroblock from the previous frame encoded using the DCT. The use of exhaustive

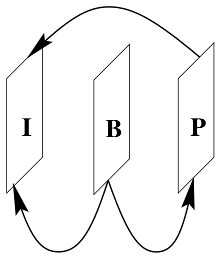

Figure 3. MPEG frame dependency.

search algorithms for motion estimation makes MPEG encoding much more computationally demanding than decoding. Macroblocks with no motion and no error are marked as *skipped* blocks, and no further coding is performed. If the error is too large, then the block is encoded as an I-block.

Bidirectional-predicted (B) frames are coded with reference to both the immediately previous I or P reference frame and the immediately future I or P reference frame. The motion-estimation and encoding procedure for B-frames is similar to the procedure used to encode the P-frames: it includes the calculation of the best-fit motion vector and the coding of the error terms using the DCT.

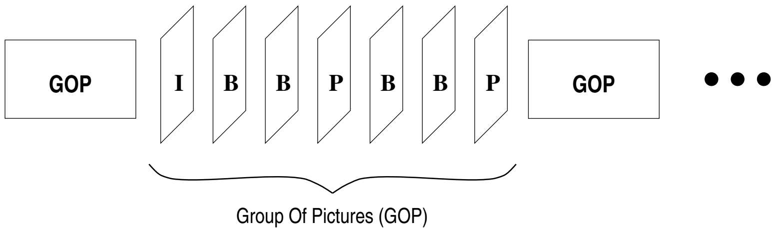

Figure 4. Illustration of an MPEG stream and group of pictures (GOP).

The sequence of I-, P- and B-frames defines what is called the *frame pattern* of a GOP, shown in Figure 4. The GOP always begins with an I-frame. The frame pattern length is the number of P- and B-frames until the next I-frame. A GOP that ends with a P-frame has no encoding dependence on the frames in the next GOP. However, if a GOP ends in a B-frame, then the encoding of that B-frame requires access to the first frame of the next GOP in the sequence.

The compression, using spatial and motion techniques, yields a bit stream that is compressed even further by the encoder using a variable-length (Huffman) coding. It is the combination of the quantization and the run-length encoding that yields the greatest compression in MPEG. This resulting bit stream becomes the MPEG sequence with the addition of the necessary sequence and GOP headers.

3.2. Input Format

The MPEG encoder assumes that the input is a set of YUV-formatted frames. The YUV input format differs from the standard format by using 2:1 subsampled chrominance planes. A special portable pixmap (PPM) to YUV converter is used to convert the input frames into the proper format for the encoder. This converter is also built into the read routine of the encoder so that the PPM-formatted frames may be read directly. Using PPM-formatted input files increases the encoding time because of the added time required for conversion to YUV format. The MPEG encoder requires the input frames to be located in the same network file system (NFS) directory. The encoder checks to see that there are enough frames for all the processors to begin work. If not, it begins processing of the existing frames and the rest of the assigned processors wait until the required frames arrive.

3.3. Parameters

The parameters that control the operation of the the MPEG encoder are either set on the command line or in a specified parameter file. These parameters control the quality and bit-rate and, subsequently, the encoding rate. These parameters are set at the beginning of encoding and do not vary.

Some of the parameters that can be set are the name of the output file, frame pattern, motion-estimation search techniques, the use of decoded or original frames for prediction, and the quantization scale. The use of wild cards allows the specification of numerically indexed input-file names in a compact syntax. The size and type of input file is either automatically detected or specified directly in the parameter file. The name of the output file is used as the root for the name of the GOP files, and to determine the output directory. The output-file name can be relative to the current directory for sequential processing or can specify the full NFS path for parallel processing.

The frame pattern specified in the parameter file sets the frame pattern for the entire MPEG sequence. This pattern cannot be changed until the beginning of another sequence. The quantization of each of the I-, P- and B-frames is set individually.

Four motion-estimation search techniques are implemented. The exhaustive search algorithm searches all possible motion vectors to the nearest pixel or half-pixel and picks the best vector based on maximum absolute difference criteria. The subsample search performs an exhaustive search on a subsampled macroblock. The two-level search performs an exhaustive search on a macroblock and then performs an exhaustive search on half-pixel region surrounding the best vector in the first search. The logarithmic search extends the concept of the two-level search to the entire macroblock, beginning with the largest possible motion first and refining the search to a finer level with each search. For more details on these parameters refer to [2], [3].

3.4. Output Format

The encoder calculates a GOP at a time on a number of machines simultaneously and writes the files to the file system. As the GOPs are written to the files, the master concatenates these GOP bit streams in serial order to form the MPEG bit stream. A decoder that accepts a bit stream from standard input can view the progress of the encoder by having the encoder output the MPEG bit stream simultaneously to standard output.

4. Software Architecture

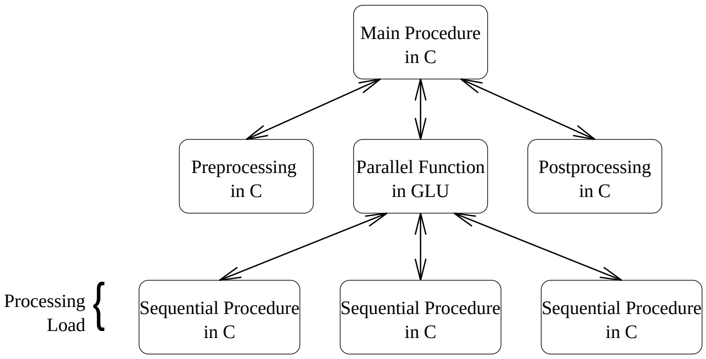

Figure 5. Software architecture for the MPEG encoder.

The MPEG encoder was implemented by modifying existing C procedures taken from the Berkeley encoder and using GLU code to handle the parallel processing. Figure 5 shows the software architecture for the MPEG encoder. At the top level, we have the main procedure in C, which calls preprocessing C routines that parse the arguments, allocate memory, etc. The main procedure then calls a GLU function that in turn calls the C procedures to be run in parallel. The sequential procedures in C called by the GLU function bear the bulk of the processing effort.

The motivation for parallelizing an application is to get a proportional increase in speed for each additional processor used. We have found that the most natural way to break up the calculation of the MPEG encoder is to look for a *for*-loop at the highest level possible that will allow independent processing in keeping with SPMD parallelism. The advantage of this approach is that it saves extensive recoding of existing applications. Using other techniques or finer-grain processing may yield more elegant code, yet often will not improve performance on most conventional parallel computers or networks of workstations.

The MPEG encoder can be parallelized at many levels, from the pixel level up to the GOP level. Our implementation breaks up the calculation at the level of the GOP. One reason that we use the GOP as the basic parallel unit is the lack of frame dependence between adjacent GOPs which minimizes bookkeeping and reduces the read time to a single read per frame. It also reduces the amount of data output by the workers since the GOP is a compressed bit stream. Coding MPEG sequences using frames as their basic unit results in increased bookkeeping and data transferred between processors during interframe compression[3]. Although there is nothing inherent in our implementation that would prevent us from encoding at the picture level, we chose to encode the MPEG bit stream at the GOP level for the sake of simplicity. Currently, we are investigating the implementation of the encoder at a lower level, but the advantage seems minimal. Another reason we use the GOP as a basic parallel unit is that the length of the GOP, and therefore the amount of work, is user-definable at the beginning of the encoding sequence. Therefore the computation time of each process can be changed indirectly by the frame pattern if the latency of the bit stream is important. Figure 6 shows the mapping of the functions to masters and workers.

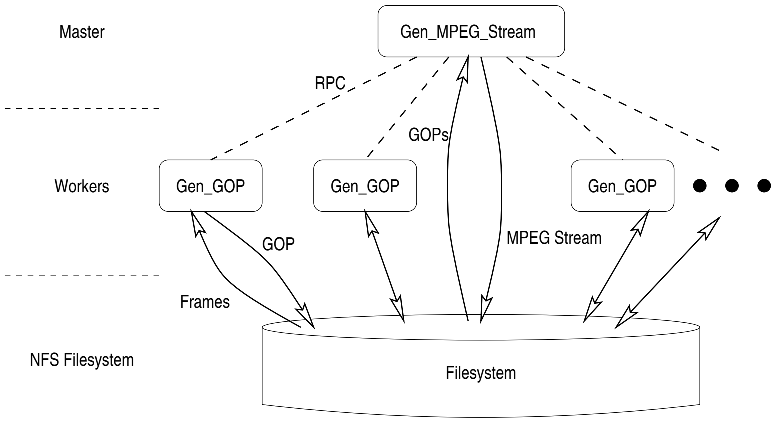

Figure 6. Software Implementation of the MPEG encoder.

5. Encoder Performance

We characterized the performance of our encoder through a number of experiments. We compare these results against that of the Berkeley encoder. We performed parallel encoding over a network of workstations connected by ethernet, and on a massively parallel processor. We also experimented with the fault tolerance of the encoder by varying the number of processors available on the network.

5.1. Network of Workstations

We performed encoding experiments on a network of Sun SPARCstation 2-class machines over a standard ethernet that has an effective data rate of approximately 4 Megabits per second. We used the Miss America sequence as the input frame set. The size of the frames in the Miss America sequence is 360×288 pixels. We used 288 frames for the input sequence, and set the frame pattern to a length of six. We ran the encoding on as many as twelve processors on a heterogeneous network, so that each processor encoded an average of four GOPs.

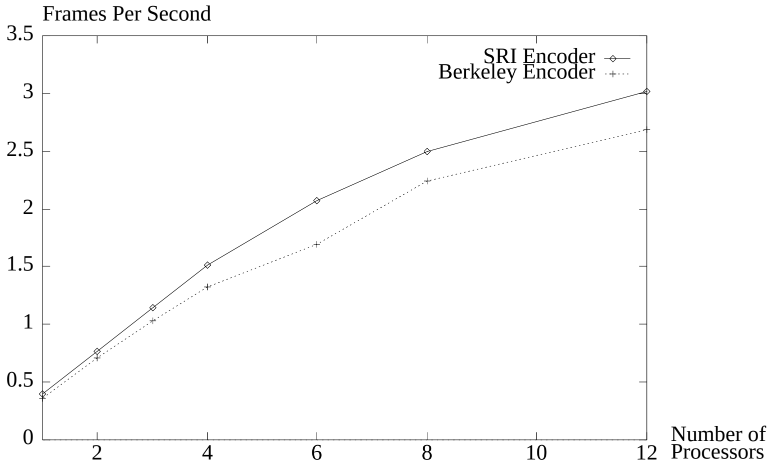

Figure 7. Frame encoding rate of the MPEG encoder on a network of Sun SPARCstation 2-class machines.

We performed the same encoding on the same set of processors with both our encoder and the original Berkeley code. Figure 7 shows the speed-up of the encoder as a function of the number of processors. This graph shows that we get linear speed-up and nearly seven times the speed using eight remote processors. Figure 8 shows that the efficiency soon starts to drop off beyond eight processors, and using twelve processors increases the speed-up factor to only slightly above eight. Beyond eight processors, network communication rate becomes the bottleneck.

A single SPARCstation 2 achieves 0.4 FPS encoding rate on a 360×288 image, and twelve remote workers achieve greater than 3.0 FPS. Therefore to achieve real-time encoding rates (30 FPS) requires a minimum of 75 workers. Using SPARCstation

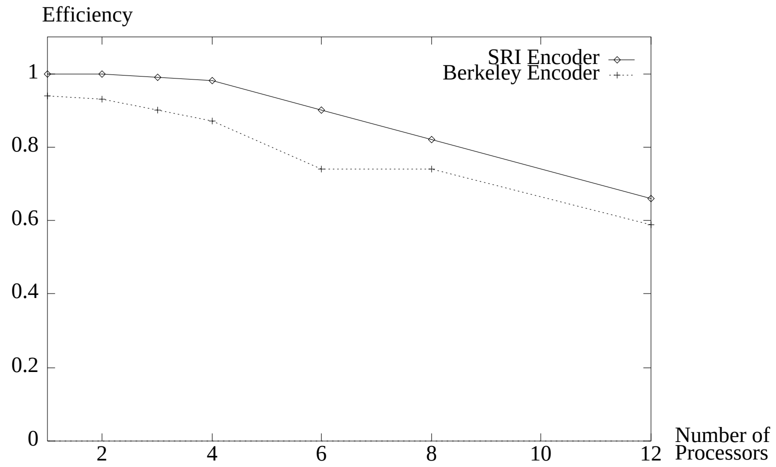

Figure 8. Processing efficiency of the MPEG encoder on a network of Sun SPARCstation 2-class machines.

20 processors, we can achieve real-time encoding speeds (30 FPS) with half as many processors. The major factor that would keep these data rates from being reached would be the communications bandwidth. To obtain real-time operation for this frame size, the interprocessor communication has to have enough bandwidth to sustain a rate of 38 Megabits/sec. Larger images would require a larger bandwidth. With the use of high-speed networks, such as ATM, and ever-increasing processor speeds and multiprocessor machines, it should be possible in the near future to achieve real-time encoding rates with our encoder on a single workstation or a small network.

5.2. Massively Parallel Processor

Our implementation can be ported to any architecture that has the GNU C compiler (gcc). We chose to implement our encoder on a massively-parallel CM5 machine. The encoder for the CM5 uses the same code as the implementation on the network of workstations.

Figure 9 shows the measurements of the frame encoding rate of our encoder on the CM5. The encoding speed increases linearly up to 16 processors. Using 16 processors, the encoder achieves a coding rate of 4.5 FPS. Beyond 16 processors, the file I/O bandwidth on the CM-5 becomes the bottleneck and adding more processors does not improve the frame rate significantly.

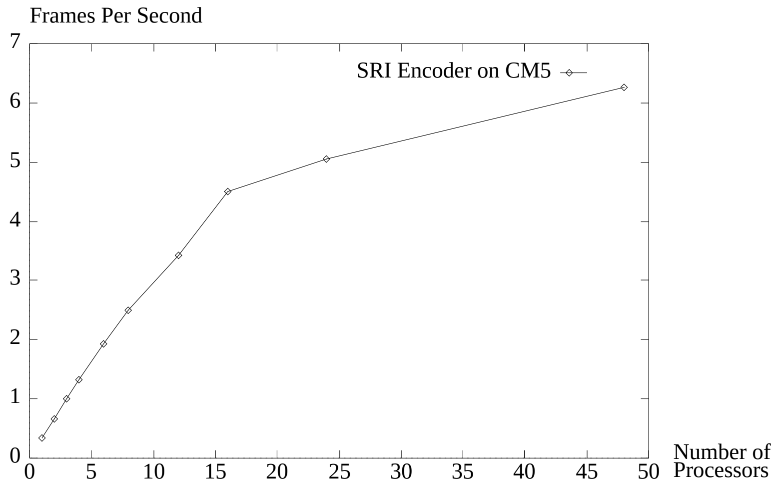

Figure 9. Frame encoding rate of the MPEG encoder as a function of the number of processors on a CM5

5.3. Fault-Tolerance and Network Adaptability

The encoder, by the fact that it is developed in GLU, is capable of adapting to the dynamically changing processing resources thereby providing both fault tolerance and dynamic scalability. We conducted a simple experiment that illustrates the adaptive nature of the encoder.

We performed a number of experiments that varied the number of processors available to perform encoding as a function of time for a 288-frame video sequence. There were three sets of processor availability patterns, and for each pattern, two different time constants (T) were used.

The three patterns we used are:

pulse-train 8 worker workstations up for T seconds and down for next T seconds,

ramp-train Add worker every t seconds until there are 8 workers; After t seconds, drop worker (in order they were added) until there are no workers; after t seconds repeat. ($t = T/8$).

first-in-first-out Start off with 4 workers; After t seconds, add one more; After t seconds, drop one (in order) After t seconds, drop one (in order) After t seconds, add one more (in order) ($t = T/8$).

Figure 10 illustrates these patterns.

Thus, we can expect the encoder not only to adapt to fluctuating resources but to do so without significant degradation.

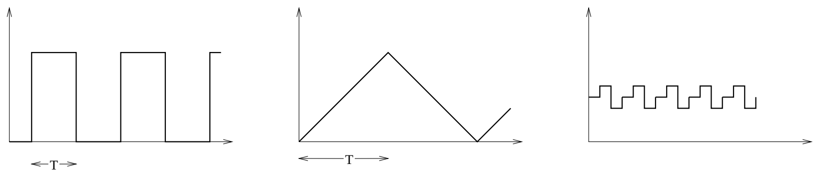

Figure 10. Processor availability.

6. Conclusion

MPEG video encoding is quite difficult to achieve in real time on a single workstation. We have presented a portable, fault-tolerant, parallel, MPEG-1 encoder implemented in software. We showed how a platform like GLU could aid in the implementation of parallel multimedia applications, such as our encoder. We showed that our encoder performs slightly better than the Berkeley encoder on a network of workstations and achieves 3 FPS with 12 Sun SPARCstation 2 machines. Our encoder can also perform MPEG encoding on a massively-parallel processor, for example a CM5 where it achieves 4.5 FPS using 16 processors. Finally, our encoder can dynamically adapt to the available network resources during execution.

References

1. D. Le Gall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.
2. K. L. Gong. *Berkeley MPEG-1 Video Encoder: User's Guide.* Computer Science Division, University of California, Berkeley, CA, 1993.
3. K.L. Gong and L.A. Rowe. Parallel MPEG-1 video encoding. In *1994 Picture Coding Symposium*, Sacramento, CA, September 1994.
4. MPEG (Moving Pictures Expert Group). Final text for ISO/IEC 11172, information technology — coding of moving pictures and associated audio for digital storage media at up to 1.5 mbit/s. Technical report, ISO/IEC, 1993.
5. A. C. Hung. *PVRG-MPEG CODEC 1.1.* Stanford University, Stanford, CA, July 1994.
6. R. Jagannathan and C. Dodd. *GLU Programmer's Guide (version 0.9).* Computer Science Laboratory, SRI International, Menlo Park, CA, 1994. Technical Report SRI-CSL-94-06.
7. Ke Shen, Lawrence A. Rowe, and Edward J. Delp. A parallel implementation of an MPEG1 encoder: Faster than real-time! In *Proceedings of SPIE Conference on Digital Video Compression: Algorithms and Technologies*, San Jose, February 5-10 1995. ftp://skynet.ecn.purdue.edu/pub/dist/delp/spie95-coding/mpeg.ps.Z.