A scalable off-line MPEG-2 video encoding scheme using a multiprocessor system

Ishfaq Ahmad [a,1], Shahriar M. Akramullah [b], Ming L. Liou [b], Muhammad Kafil [a]

[a] *Department of Computer Science*

[b] *Department of Electrical and Electronic Engineering*

Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

Abstract

Video compression plays a central role in a vast number of multimedia applications but its computational requirements are beyond the capabilities of any present single processor system. In this paper, we explore the use of parallel machines like the Intel Paragon to compress MPEG-2 video sequences. The motivation is to build a production-based compression facility by exploiting the potential power of the available machine. Given a video sequence or a set of sequences, the aim of the parallel encoder is to achieve the maximum possible encoding rate. A collective scheduling scheme for the processors, I/O nodes, and disks is proposed that provides fast I/O, minimizes the idle times of processors, and enables the system to work in a highly balanced fashion. An efficient data layout scheme for storing video frames is also proposed in order for the I/O to sustain the desired data transfer rates. Using a small percentage of processors as the I/O nodes results in an efficient utilization of the system resources. As shown by experimental and analytical results, the encoding scheme is scalable and higher performance can be achieved with larger machines. The performance of the proposed scheme can be many times the real-time encoding rates with SIF (Standard Interface Format) and CCIR-601 video sequences. The experimental results indicate about two-fold gain in performance compared to the previous studies. Such a facility is useful for the conversion of analog videos to compressed digital form in large studios, digital libraries, and other multimedia database environments. The proposed scheme partitions the system into groups of compute nodes, and I/O nodes, and can be easily extended to other MIMD machines or a set of networked workstations.

Keywords: MPEG-2, video encoding, parallel processing, scheduling, multimedia, Intel Paragon.

[1] Corresponding author, E-mail: iahmad@cs.ust.hk

1 Introduction

The realm of high-performance parallel and distributed computing is expanding beyond the traditional scientific community because the current revolution of information technology has created a vast number of commercial applications that require massive computing power [5]. The need for high-performance computing in commercial applications is being recognized by researchers from the areas of both parallel processing and information technology. Large-scale databases, video servers, visualization tools, content-based search and retrieval, graphics rendering, etc., are typical applications that require large computing power.

Video encoding (also called compression) is another application which requires enormous computing power and thus can benefit from high-performance computing. Digitized video, which is a fundamental component of common multimedia applications, typically needs massive amount of data required to represent the audio-visual[2] information. A few minutes of video sequence requires Gbytes of data storage. Similarly, the amount of data for transmitting a video sequence over a network can easily overwhelm the network channels. For example, to transmit an uncompressed high quality digital television signal (CCIR-601) would require 126 Mbps. Even for a lower resolution signal suitable for video conferencing applications (Common Intermediate Format), the uncompressed bit rate is 36.5 Mbps [8]. These amounts become out of reach for HDTV (high-definition television) [9], which will be using digital video at a much higher resolution of 1920×1152.

Storage and transmission of a huge amount of data inevitably calls for compression and decompression of digital video. Fortunately, digital video contains ample redundancies in both the spatial and temporal domains, enabling encoding algorithms to achieve a high degree of compression with little degradation in quality. The entire compression/decompression process requires a *codec* consisting of an encoder and a decoder. The encoder compresses the data at the transmission or storage end while the decoder decompresses the data for reproducing the video to be viewed by the user. In order to guarantee exchange of the compressed video data between different systems such that a unique decompression is possible for a particular encoded bitstream regardless of the decoder configuration, video coding algorithms are standardized. Two recent international standards, known as MPEG-1 and MPEG-2, have been developed by the MPEG (Moving Pictures Expert Group) of the ISO (International Organization for Standards). These standards specify the syntax (representation) for the decoding of video and accompanying audio data.

[2] An audio data stream, having a smaller processing requirement as compared to video, does not appear as challenging, and is not addressed in this work.

Compression is considerably more complex as compared to decompression (which can possibly be done using a single processor machine). The objectives of an efficient video compression techniques include (assuming a defined bit rate) a good visual quality evaluated through subjective and objective assessment criteria, high compression ratio, and low complexity of the compression algorithm itself – the emphasis on any of these objectives can, of course, vary according to the target applications. Compression can be aimed for real-time or non-real-time encoding environments. In the former case, compression must be achieved *on-line*, for example in a system in which a video stream is being generated from a source such as a video camera; a real-time encoding rate is about 30 frames/sec. In the latter case, compression can be done *off-line* without strict requirements of real-time compression rate. Non-real-time compression is required in applications like digital library or production systems which require encoding a video sequence and storing it on a CD-ROM or DVD (Digital Versatile Disk).

There are two approaches to performing video compression: hardware-based [1], [23] and software-based [2], [3], [4], [7], [10], [11], [19], [20], [22], [24]. Both approaches have their own advantages and disadvantages. A hardware approach uses a special-purpose architecture, and its advantages include the ease of use and high compression speed. However, dedicated hardware is less flexible and can become obsolete. Furthermore, hardware is often optimized for a particular coding algorithm, and cannot be used for exploring other present and future video compression standards. A software solution using general-purpose computing platforms, on the other hand, is more flexible, and thus allows algorithmic improvements. In addition, for non-real-time applications, a software implementation can produce better quality video compression by tuning various parameters and by allowing multiple passes for optimization. However, the very high computation requirements of video applications can often overwhelm a single-processor sequential computer [2], [21]. Therefore, it is natural to exploit the potentially enormous computing power offered by parallel computing systems.

In this paper, we explore the use of parallel machines like the Intel Paragon to compress MPEG-2 video sequences. The motivation is to build a production-based compression facility by exploiting the potential power of the available machine. In addition, parallel machines which are normally available for scientific computing can be exploited for new multimedia applications. The compression facility is aimed to compress multiple and large video sequences. The environment is not real-time but the aim is to achieve the maximum possible encoding rate beyond the real-time speed. Such a facility is useful for the conversion of analog videos to compressed digital form in large studios, digital libraries, and other multimedia database environments. We propose schemes for data layout, efficient I/O, and load-balanced data distribution. These schemes provide fast data retrieval as well as efficient scheduling and

matching of I/O and computation rates such that the entire machine operates in a highly balanced fashion without any bottlenecks. Using a very small percentage of processors as the I/O nodes results in an efficient utilization of the system. More importantly, our scheme is scalable, that is, an increase in the number of processors will result in a proportional increase in the encoding rate. As a result, larger machines will yield higher encoding rates. Specifically, given any MIMD machine configuration (that is, the number of processors, I/O nodes, and disks), our proposed method will logically configure the machine for the best possible utilization and match the I/O and encoding rates to reach the ideal performance level.

The rest of this paper is organized as follows. Section 2 provides an overview of MPEG-2, followed by Sections 3 which briefly discusses the related work and gives a motivation for pursuing this research. Section 4 gives an overview of the Intel Paragon and its logical partitioning used in our encoding scheme. Section 5 includes a discussion of the proposed parallel encoder and various related issues. Section 6 presents the experimental results. Section 7 discusses the scalability of the encoder and the last section provides some concluding remarks.

2 Overview of MPEG-2

Video coding standards provide a common format and enable the sharing of technology among various industries. Some of the recent standards are JPEG (to compress still images for both storage and transmission applications) [13], H.261 [17] and H.263 (for video telephony and video conferencing applications at a low bit rate), and MPEG-1 (for applications requiring up to 1.5 Mbps bit rate) [6], [14]. The Moving Picture Experts Group of ISO has standardized the second international standard (MPEG-2 [15]), which is targeted for a variety of applications at a rate of 2 Mbps or above with a quality ranging from good quality NTSC to HDTV.

MPEG-2 is designed to be a *generic* standard to support a wide variety of applications and hence works in various modes, called *levels* and *profiles*, suitable for different environments. Both MPEG-1 and MPEG-2 employ three basic techniques for compression. The first is the reduction of spatial redundancies, which is done by transforming the spatial pixel values into the frequency domain by using the discrete cosine transform (DCT). The second is the reduction of temporal redundancies, which is carried out by using motion compensated prediction, that is, instead of transmitting an entire frame, only the changes in position of blocks of pixels with respect to the previous frame (along with the difference between the actual frame and predicted frame) are transmitted. The third technique for achieving additional compression is the

use of statistical properties to encode the data using fewer bits.

MPEG-2 defines the syntax of its video which is structured in six hierarchical layers: Sequence layer, Group of Pictures (GOP) layer, Picture layer, Slice layer, Macroblock (16×16 pixel area) layer, and Block (8×8 pixel area) layer. To allow random access to the coded bitstream while achieving a very high compression ratio at the same time, pictures are classified as Intra-coded (I), Predictive coded (P), and Bidirectionally predictive coded (B) pictures. I-pictures provide good random access with moderate compression and are used as reference pictures for future prediction. P-pictures are coded more efficiently from a previous I- or P-picture and are generally used as reference pictures for further prediction. B-pictures provide the highest degree of compression but require both past and future reference pictures for motion compensated prediction. The algorithm first selects an appropriate spatial resolution for the signal, and then performs motion estimation by block matching. Motion estimation refers to finding the displacement, called *motion vector*, of a particular macroblock of the current frame with respect to a previous or future reference frame or combination of both. A search is based on *mean absolute difference* (MAD) matching criteria, i.e., a match is found that yields the minimum accumulated absolute values of the pel differences for all macroblocks.

Next, the algorithm performs motion-compensated prediction for the temporal redundancy reduction. The difference signal, i.e., the prediction error, is further compressed using the block transform coding technique which employs the two-dimensional 8×8 DCT. The resulting transform coefficients are quantized in an irreversible process that discards the less important information. To allow a smooth bit-rate control, an adaptive quantization is used at the macroblock layer. The motion vectors are combined with the residual DCT information, and transmitted using variable length codes.

MPEG-2 provides more advanced features but is compatible with MPEG-1, that is, an MPEG-2 decoder can decode an MPEG-1 compressed bit-stream. MPEG-2 supports interlaced video in addition to progressive video (which is also supported by MPEG-1). Besides, MPEG-2 takes other measures to improve the picture quality. Furthermore, MPEG-2 provides concealment motion vectors for I-pictures in order to increase robustness from bit errors. MPEG-2 introduces variable bit-rate along with usual constant bit-rate. Moreover, MPEG-2 presents two color spaces, namely 4:2:2 and 4:4:4, in addition to 4:2:0 used by MPEG-1.

3 Related Work

The problem of software-based video encoding using parallel processing is non-trivial, and cannot be solved by simply replicating multiple sequential encoders on different processors, because the local memory of a single processor is usually not large enough to hold more than a few frames and thus an efficient I/O methodology is required to bring the data in and take the compressed data out of processors. Since the video signal can be viewed as a 3-dimensional (3-D) signal, that is, two dimensions in the spatial domain and one in the temporal domain, various partitioning schemes are possible. The parallelism can be exploited at macroblock, slice, frame and/or GOP (group of pictures) level. Moreover, the best method of parallelization also depends on whether the encoding is on-line or off-line.

In on-line encoding the data arrives from a live source, presumably at a rate of 30 frames/sec., which must be compressed on-line at that speed. To avoid any delay, each incoming frame must be processed in real-time. A natural solution to on-line encoding using a software-based parallel processing approach is to partition a frame as it arrives among many processors [2]. All processors then concurrently encode their parts of the frame data. The degree of parallelism can be increased by making the problem granularity as small as allowed by the frame size and the available number of processors.

This kind of parallel encoding, nevertheless, is still non-trivial from the perspective of parallel processing, as the computations in the processors are not independent and the processors need to communicate with each other to exchange certain parameters. The encoding times are not the same for all data partitions and hence global synchronization can incur waiting times at some processors. In addition, the overhead of data-distribution and concatenation of results can saturate the speedup if the number of processors is increased. In [2], a similar approach is used. Although a real-time encoding rate is achieved for the first time, it used 330 processors on the Intel Paragon, with the granularity of the problem (the amount of data per processor) being equal to a macroblock.

The drawbacks of such an approach, in addition to using massive parallelism, is that the speedup saturates as one macroblock is the smallest unit of data that can be reasonably assigned to one processors. Decreasing the granularity beyond a macroblock (e.g. a simple block) would incur very heavy inter-processor communication due to the motion estimation operation that needs to search the data beyond the local block.

In the off-line approach, the data stream does not arrive from a live source, rather an entire video sequence may be already available. Such an approach

is useful for production systems such as encoding a large video sequence into a disk or CD-ROM. In this approach, one can partition the video data in the temporal domain and let each processors independently encode a sequence of frames, such as a GOP.

The pioneering work done for non-real-time environment has been reported in [21] and [22]. In [21], GOP-level temporal parallelism has been used for MPEG-1 using the Intel Touchstone Delta and the Intel Paragon. I/O contention has been found to be the major bottleneck, and an I/O management policy has been used to restrict the number of PEs (processing elements) that involve in I/O at the same time. Frame rates of over 41 frames per second for CIF (352×288 pixels) sequence have been reported using 100 processors on the Paragon or 144 processors on the Touchstone Delta. However, the performance of the scheme starts degrading when more processors are used due to I/O contention. While the overall performance is faster than real-time, the entire video needs to be available and statically partitioned in advance.

A subsequent study [22] has indicated that for the CCIR-601 (704×480 pixels) sequence, the performance curve of the scheme goes down when more than 400 processors are used. It is mainly due to the large number of processors reading from parallel file system (PFS) simultaneously. A novel scheme using a combination of spatial and temporal parallelism has been proposed to reduce the number of processors which are involved in the I/O at the same time. In this scheme, the Paragon processors are evenly divided into groups of size m, where one processor in each group is devoted for I/O while other processors are responsible for encoding and are considered as compute nodes. An I/O node reads in a video section (1 GOP) and sends it to compute nodes in the group, each frame in the section is divided into slices which are processed by the compute nodes in parallel. The compressed data is then sent back to the I/O node on a frame by frame basis. An I/O node assembles the data and then writes it to the PFS after it reads the next video section. A maximum frame rate of 33 frames/sec. for CCIR-601 (704×480 pixels) sequence is achieved with a group size of 5 using 512 processors. Using this group size, 102 processors are reserved as I/O nodes, that is, only 410 processors are encoding. A large number (102 in the above case) of I/O nodes also cause I/O contention because the actual number of disks used by the PFS is considerably smaller than that.

In addition to the above studies with notable results, there have been some other previous attempts ([20], [23], [24]) to parallelize video encoders. In [20], MPEG-1 is implemented on a set of Ethernet-connected workstations. Slice-, frame-, and GOP-level parallelism has been exploited and compared. In their implementation, one of the workstation is devoted as master workstation which reads and distributes the video streams while the rest of the workstations do encoding. GOP-level parallel implementation has been found to perform best because of its low communication overhead in the Ethernet environment. A

frame rate of 7.8 frames/second has been reported using 18 workstations.

I/O in parallel computer systems can be a bottleneck in a number of parallel applications. Removal of the I/O bottleneck requires an integrated approach which addresses the problem at all levels of the system, including the storage and parallel architecture [16]. In the context of video encoding, the objective is to accomplish the optimal encoding rate, which can be achieved if all the processors are kept busy (that is their waiting times are zero). This requires a careful I/O strategy that is highly pipelined and always provides data whenever a processors finishes encoding of its previously assigned data. This, in turn, requires a data layout scheme that can minimize all of the overhead and yield the desired I/O rate.

4 A Logical Partitioning of the Intel Paragon

The parallel machine we have used in our study is the Intel Paragon XP/S parallel computer. The architecture of the Paragon, which is a distributed-memory machine, has been documented well in various publications [12]. Here, we describe some details of its architecture that are relevant to our work. We also propose a logical partitioning of its processors that is the basis of our scalable MPEG-2 video encoding scheme.

The Paragon consists of compute, I/O, and service partitions (see Figure1), each with a certain number of processors. The nodes (processor/memory pairs) are connected in a two-dimensional rectangular mesh, with each node connected only to its nearest four neighbors in the mesh. Each compute node is powered by an i860 XP processor operating at 50MHz and has a peak floating point performance of 75 Mflops double-precision or 100 Mflops single-precision. Each node is equipped with 32 MB of DRAM where 24 MB is available to user programs. In addition, each i860 XP contains 16 KB of instruction cache and 16 KB of data cache. The machine used in our experiments contains 128 compute nodes, 12 service nodes, and 7 disks. However, in order to meet the requirements of 2-D mesh connection, we use 6 disks, 6 nodes for I/O and 120 nodes for encoding.

The file systems supported on the Intel Paragon are UNIX File System (UFS), Network File System (NFS), and Parallel File System (PFS). The PFS consists of multiple stripe directories where each directory is on a separate disk and is a mount point of a separate UFS. Each disk is controlled by a dedicated I/O node. Files stored in PFS are distributed or striped across all the stripe directories. The number of stripe directories in a PFS is called the *stripe factor*, and the amount of data from each file that is stored in each directory (disk) is called *stripe unit* [12]. For instance, if the stripe unit size is 64kbytes, the

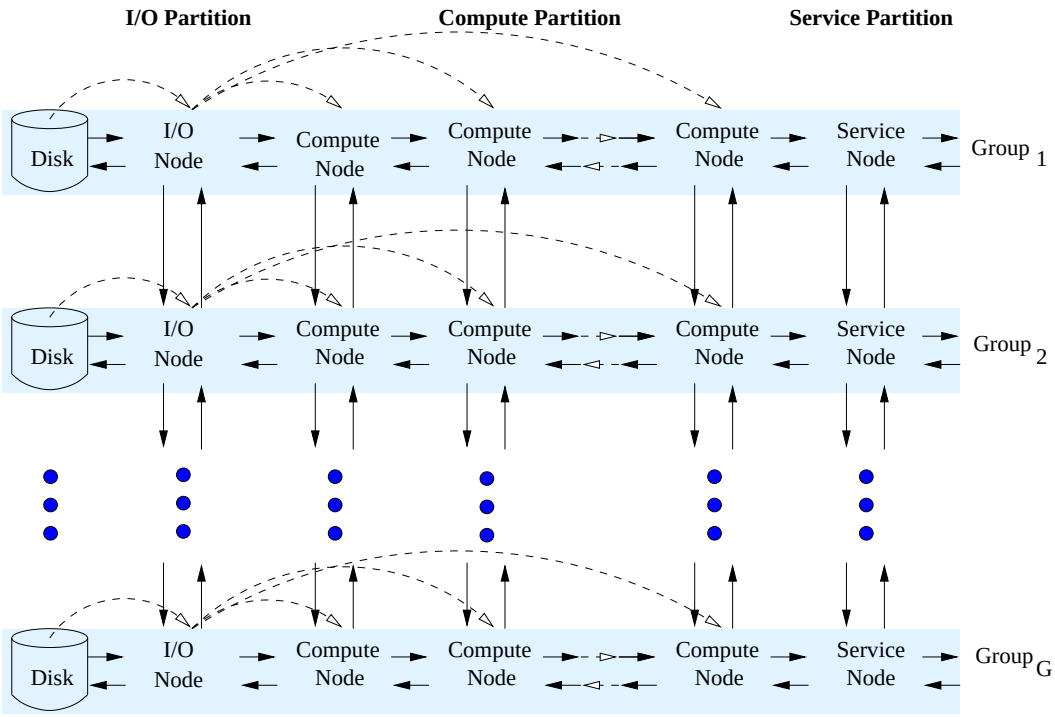

Fig. 1. The architecture of the Intel Paragon and the partitioning of the compute-processors into groups.

first 64kbytes of a file will be stored in the first UFS disk and the second 64k in the next UFS disk and so on. Parallel data access is thus provided when data are residing in multiple UFS disks. The product of the stripe factor and stripe unit is called the *full stripe size*.

For our encoding scheme, we logically divide the processors of the Intel Paragon into G groups, where the largest value of G is the total number of disks used by the Paragon parallel file system (PFS). Each group has one disk and one I/O processor which is responsible for reading the uncompressed data from the disk and delivering it to the compute processors which are responsible for encoding and writing the compressed data.

5 The Parallel Encoder

The objectives of our parallel encoder are:

- To achieve the maximum possible encoding rate, given any machine configuration (that is, the number of processors, I/O nodes, and disks).
- To achieve a complete scalability, in that, the encoding rate should increase linearly with an increase number of processors without reaching a saturation point in performance.

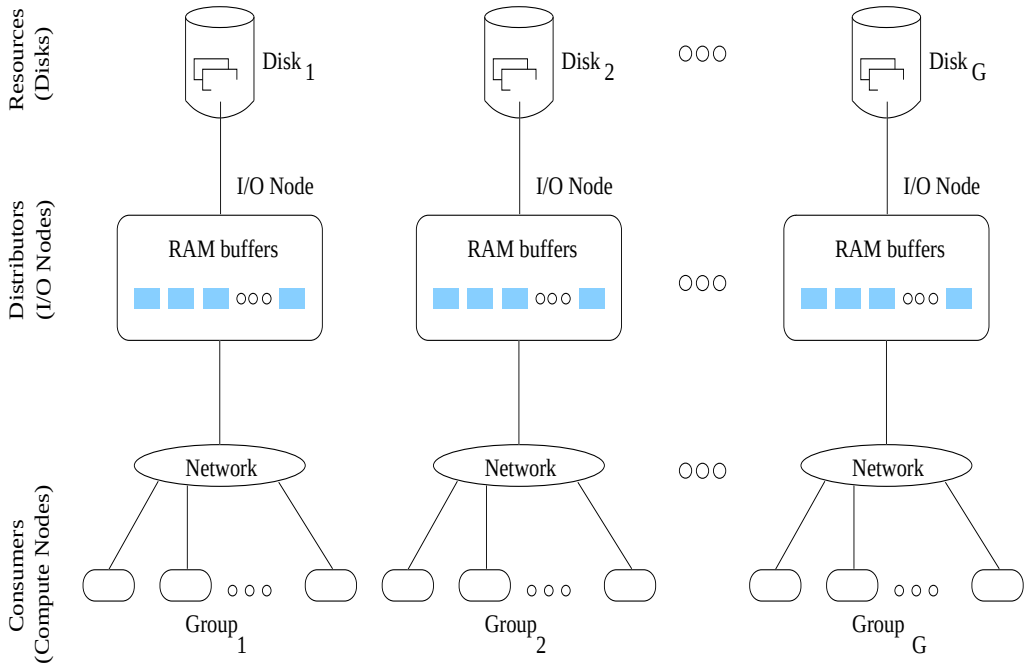

Fig. 2. MPEG-2 parallel encoder.

The objectives can be met if all of the processors are kept busy in reading the data, performing the encoding, and writing the coded bit stream. In other words, the processor waiting times are minimized and thus the encoding rate is the maximum possible. The proposed scheme achieves these objectives by scheduling the disks, I/O nodes, and the processors. The scalability issue of the proposed scheme is discussed in Section 7. An extra plus point of our parallel encoder is that the number of processors dedicated for the I/O is very small (about 5%) compared to that (about 25%) in one of the previous studies [22].

In this work, the entire system including the computing nodes, I/O nodes, and disks should work in a highly balanced fashion, and their operations should overlap with each other. As illustrated in Figure2, our encoding system can be viewed as a collection of producers (disks)[3], distributors (I/O nodes), and consumers (compute nodes). The goal is to sustain an uninterrupted supply of the product (data) needed by the consumers so that they do not starve. The consumer demand requires matching of the distribution rates with the consumption rate. The entire population of consumers is divided into groups and one distributor is assigned to each group. At the same time, the number of distributors should be kept the minimum. Furthermore, the producers should

[3] One can assume that an external source is providing the data for the disks. Then the disks can be assumed to be *buffer*s for the external source, which can be filled when the distributors are busy distributing the previous data to the consumers. In our scheme, this buffer size does not require the entire video and is very small.

keep up with the distribution and consumption rates.

In the proposed scheme, an efficient load-balanced scheme for distributing the data from the I/O nodes to the compute nodes is described. Whenever a compute processor finish encoding, it demands data from the I/O processor which is ready to provide the required data for the next round. To maintain such an I/O with little overhead, a data layout scheme on the disk is designed, as explained below. The notations and symbols used in the subsequent discussion are included in Table 1.

5.1 Data layout

The main issue is how to divide a video sequence into a set of files and to spread those files on the disks to reduce the I/O contention and file opening overhead. There are three simple ways of dividing a video. First, to store the entire sequence in one file. Second, to store each frame (including one luminance and two chrominance components) in one file. Third, to store the luminance and the two chrominance components of a video frame in three separate files. The first option is obviously impractical because of the large space requirement and is inefficient because of the potential contention. The second and the third options incur a substantial overhead for opening a large number of files. The division of the video sequence in our work is a trade-off strategy and is elaborated below.

Uncompressed frames of the video sequence are distributed to different disks on a GOP basis, that is, the video sequence is first divided into several GOPs which are then divided among the disks as evenly as possible. Without loss of generality we can assume that the total number of GOPs (G) is divisible by the number of disks (D). Each disk stores an array of GOPs. The size of the array can be a multiple of m (the group size). Further, within a group, each compute node is assigned at least one GOP, containing g frames (12 frames in our case).

An I/O node goes through a *disk read phase* in which it reads the data for all the compute nodes in the group, followed by a *sending phase* in which it sends the data read in the disk read phase. The combination of the two phases is considered as a *round*. The frames required for one compute node in one round are called a *batch*; a batch may contain k frames, where $k \in 1, \ldots, g$. All the batches which are required in one round for all the compute nodes in a group are combined in one file (we will call it a *vector* in the subsequent discussion). This clustering of frames is done to avoid the overhead of opening a large number of files. There are L layers of GOPs in a disk, each of which contains R vectors (see Figure3).

Table 1
Notations and parameters

Symbol	Meaning
N	Total number of nodes
D	Number of disks = Number of I/O nodes
m	Number of compute nodes in a group = $(N - D)/D$
M	Total available memory in one I/O node
B	Number of frames in a batch
R	Total number of rounds
n	Number of total frames in a video
g	Number of frames in a group of picture (GOP)
G	Total number of GOPs in the video sequence = n/g
L	Number of layers in a disk
F	Size of one frame in bytes
b	Size of buffer to contain one batch = $F \times B$
T_{open}	Time to open a file
t_{read}	Time to read one frame
T_{read}	Time to read one batch
T_v	Time to read one vector
t_{enc}	Average time to encode one frame
T_{enc}	Time to encode one batch (average over all compute nodes)
t_{write}	Time to write one frame to PFS
t_{send}	Time to send one frame to a compute node
T_{send}	Time to send a batch to a compute node
t_{recv}	Time to receive one frame by a compute node
$T_f(j)$	Minimum finish time of compute nodes in j^{th} round
$T_{k,i}$	Encoding time of the batch received in i^{th} round by the k^{th} compute node
$T_w(j)$	Waiting time for an I/O node for the j^{th} round
r	Data transfer rate from the disks to I/O node
f	Frame encoding rate

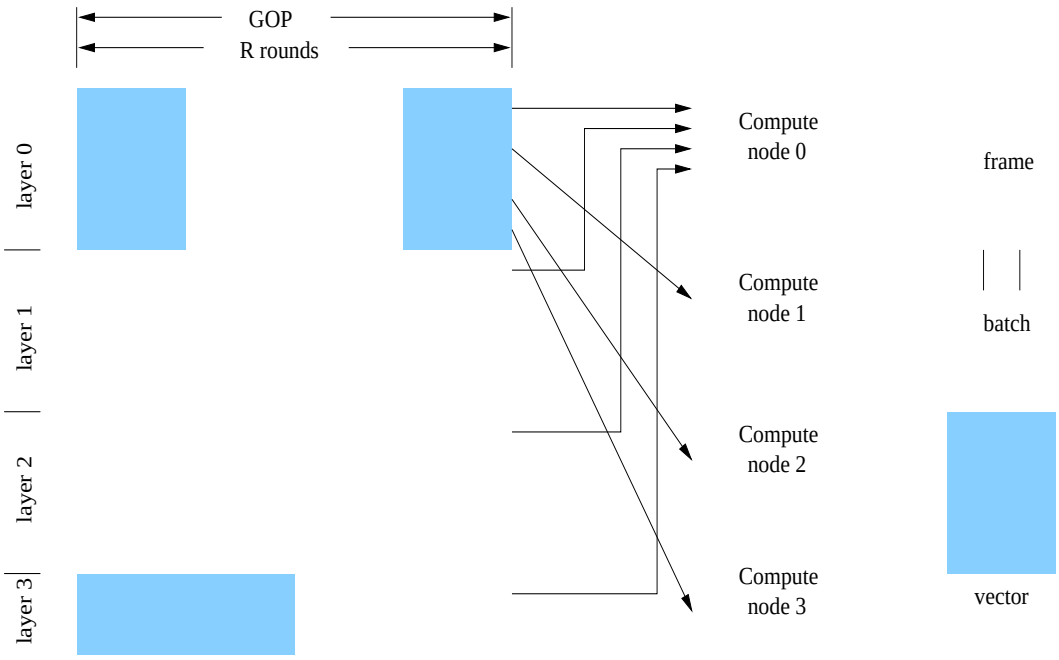

Fig. 3. An example of data layout on a disk for distribution among compute nodes by one I/O node.

If the size of the array of GOPs in one disk is a multiple of m, then the vector containing the batches is as follows: $v :< k, k >$, for $k = 0, 1, \ldots, m-1$ where $< i, j >$ represents that batch i in the vector that will be sent to processor j in the group and m is the group size. The same vector will be used in all the rounds.

Consider an example with the total number of frames equal to 48. We divide it into 4 GOPs and assume that there is only one group of 4 compute nodes. Each compute node will be assigned one GOP, and if a batch of 1 frame is used, there will be only one type of vector having 4 frames. Therefore, the I/O node will read and send 4 frames in a round, and there will be 12 such rounds. On the other hand if a batch of 3 frames is used, the vector will have 12 frames. In this case, the I/O node will read and send 12 frames in a round, and there will be 4 such rounds. The time required to read a vector v can be computed as:

$$T_v = T_{open} + m \times T_{read}, \tag{1}$$

where T_{open} is the average time to open a file, which is the time needed to read the file and the disk addresses of the data blocks from the disk, T_{read} is the average time to read one batch of frames.

However, when the size of the array of GOPs in one disk is not a multiple of m, there will be one layer (e.g. layer 3 in Figure3) with number of GOPs less than m. Since one compute node must encode at least one GOP, if there

is l GOPs in the a layer ($l < m$), we must use l compute nodes for that layer. Consequently, the vectors for that layer will be: $v :< k, mod(k, l) >$ for $k = 0, 1, \ldots, m-1$, where $< i, j >$ represents that batch i in the vector will be sent to processor j in the group of l compute nodes, and there are m batches in a vector. The time to read a vector v is the same as expressed in Equation (1).

5.2 The data distribution scheme

Our data distribution uses a dynamic approach. The I/O node of a group reads a vector from the associated disk and sends one batch of frames to each compute node in the group. While the compute nodes are encoding, the I/O node reads the next vector from the disk and waits. As soon as a compute node finishes the encoding of its earlier batch it sends a request message for the next batch. After receiving the request, the I/O node sends the next batch to the requester. After serving all the requests, the I/O node reads the next vector and waits for the requests. The compute nodes save the compressed data into a buffer, and write it to the PFS when it is full. The size of the buffer is set to be a multiple of the *full stripe size* of the PFS to optimize the writing. All of the disks in the PFS are used for writing as compared to one disk used by an I/O node for reading.

The overhead of writing the compressed data is very small as compared to reading the uncompressed data, therefore, no scheduling is employed for writing the compressed data. The operation of this scheme is shown in Figure4 with three compute nodes in a group.

5.2.1 Buffer size at I/O nodes

Uncompressed data is read into the RAM buffers from the magnetic disk for all the compute nodes in a group and then sent to compute nodes (see Figure2). Two most important parameters are disk reading time per frame and average encoding time of the frames on compute node. Sending or receiving time for a frame is not significant as compared to reading and encoding times, therefore we will not consider them in the following discussion. The disks used in the Paragon are Maxtor 120 and their characteristics are given in Table 2. Since the average access time is average seek time + average rotational delay + transfer time + controller overhead, and disk reading overhead for Paragon PFS is per 64k block basis, reading more 64k blocks at one time will result in less overhead per frame. Disk reading time per frame using batch sizes of 1, 3 and 6 frames is shown in Table 3. The numbers within parenthesis in third column represent the actual number of 64k block being used. It may be

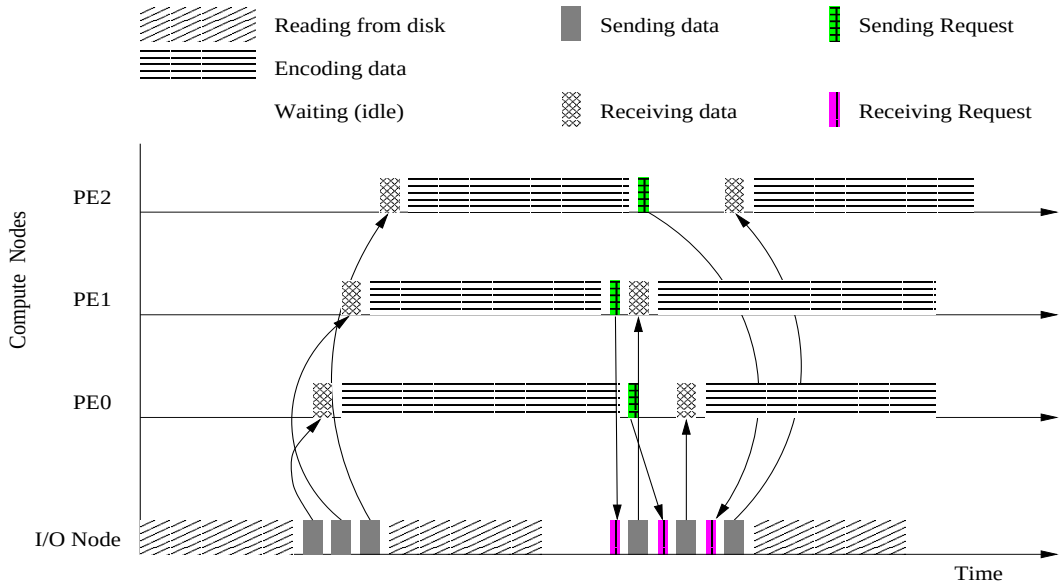

Fig. 4. An illustration of the data distribution scheme.

Table 2
Paragon disk (Maxtor) characteristics.

Transfer rate	3-4 MB/sec
Rotation speed	3200-3600 rpm
Average seek time	16-18 msec

Table 3
Observed frame reading times for SIF (352×240) *flower garden* sequence.

Batch size (frames)	Batch size (bytes)	No. of 64K blocks	Access time (msec.)	Transfer time (msec.)	reading time/ frame (msec.)
1	126720	1.93 (2)	63	30	66.0
3	380160	5.80 (6)	168	90	56.0
6	760320	11.60 (12)	300	180	50.0

noticed from column 4 that the disk reading overhead (access time) does not increase linearly with increase in the number of blocks in a batch.

The ideal situation is to have the serving rate (disk reading and sending) of the I/O node equal to the encoding rate of the compute nodes, so that neither the I/O node nor the compute nodes are idle at any time. The optimal buffer size for an appropriate group size (m) is the one that balances the disk reading rate with the encoding rate. The memory requirement for the m RAM buffers should be less than or equal to the total available memory M, i.e., $m \times F \times B \leq M$ where F is the frame size (no. of bytes in a frame) and B is the number of frames sent to a compute node in one round.

Therefore, the maximum batch size (in bytes) will be,

$$B_{max} = \left\lfloor \frac{M}{m \times F} \right\rfloor$$

and consequently, the buffer size b will be, $b_{max} = F \times B_{max}$.

This buffer size can be used for all the rounds if the following equation is satisfied:

$$T_{open} + m \times B \times (t_{read} + t_{send}) \leq T_f(j). \tag{2}$$

where $T_f(j)$ is the minimum finish time of the j^{th} round among all compute nodes in the group and is given by:

$$T_f(j) = B \times (t_{recv} + t_{write}) + \min_k T_{k,i}, k = 1, \ldots, m, \tag{3}$$

where $T_{k,i}$ is the encoding time for i^{th} batch by the k^{th} compute node. Equation (2) requires that the serving time for one round on the I/O node should be less than or equal to the encoding time of all compute nodes in the group. This requirement guarantees no waiting time on the compute nodes. A larger batch size (constrained by the memory size) is better so as to utilize the I/O node more efficiently and to reduce the communication overhead.

5.2.2 Waiting time of an I/O node

Disregarding the negligible values of Tsend and Trecv, we can determine the waiting time $T_w(j)$ for an I/O node in the j^{th} round as follows:

$$T_w(j) = \max_k \sum_{i=1}^{j} T_{k,i} - \max_k \sum_{i=1}^{j-1} T_{k,i} - m \times T_{read} - T_{open}, k = 1, \ldots, m, \tag{4}$$

and the total waiting time of an I/O node during R round is:

$$\sum_{j=1}^{R} T_w(j) = \max_k \left\{ \sum_{i=1}^{R} T_{k,i}, k = 1, \ldots, m \right\} - R \times m \times T_{read} - R \times T_{open}. \tag{5}$$

One can determine from the experimental data the expected value of the total encoding time of each compute node in a group (served by an I/O node) i.e. $E\left[\sum_{j=1}^{R} T_w(j)\right]$ in order to minimize the total waiting time of that I/O node. Since the I/O nodes are assumed to be identical, this approach will guarantee the optimal utilization of resources for any given number of compute nodes, I/O nodes and disks.

The ideal frame rate is defined as the maximum number of frames the encoder can encode in one second without any waiting times on compute nodes, given some numbers of processors and disks. As compared to the sequential encoder, the utilization of compute nodes in the parallel encoder is 100

$$f = (N - D)/t_{enc}, \tag{6}$$

where N is the total number of processors used and D is the number of disks or the processors reserved as the I/O nodes, t_{enc} is the average encoding time per frame. But as parallel implementation introduces some other overheads into the encoding time, the encoding overhead per frame will be:

$$T_o = t_{recv} + t_{write},$$

where t_{recv} is the average receiving time per frame and t_{write} is the average writing time of the encoded bitstream to the disk for one frame. Therefore, the equation for ideal frame rate becomes:

$$f = \frac{(N - D)}{(t_{recv} + t_{enc} + t_{write})}. \tag{7}$$

The average writing time per frame (t_{write}) is difficult to estimate, because it varies depending upon the number of processors writing to the PFS at the same time. The average across all the compute nodes will be used as t_{write} to estimate the ideal frame rate.

6 Experimental Results

The SIF (360×240) video sequences *flower garden*, *table tennis* and *football* were used as test sequences. Two CCIR-601 (720×480) sequences *susie* and *football* were also used for our experiments. The sequences were repeated to obtain about 4000 frames for SIF and 3000 frames for CCIR-601. The size of the GOP was 12 frames with I-P frame distance of 3. For motion estimation, the 2D-logarithmic search with the search windows of ± 11 for P frames and ± 10 for B frames were used. The sequential encoder used was the one from MPEG Software Simulation Group [18]. The encoder was compiled using the C compiler on the Paragon with all the optimizations enabled.

Table 4 include the results for the SIF football sequence. The first column shows the total number of processors while the second column is the number of compute nodes in one group. The obtained frame rates for buffer sizes of

Table 4
Encoding rates for SIF (360×240) *football* sequence.

No. of Processors	Group Size	$b = F \times 1$		$b = F \times 3$	
		Frames/ sec.	Avg. idle time (msec.)	Frames/ sec.	Avg. idle time (msec.)
48	7	25.28	27.78	25.50	5.01
60	9	32.21	68.39	32.60	4.24
72	11	38.81	76.97	39.51	5.40
84	13	45.41	102.44	46.23	5.82
96	15	52.38	90.43	52.70	6.28
114	18	61.38	158.11	62.25	7.35
126	20	66.68	157.15	69.38	8.50

1 and 3 frames are given in columns 3 and 5, respectively. The average idle times on compute nodes are given in columns 4 and 6. The achieved frame rate is calculated by dividing the number of total frames by the encoder finish time; the encoder finish time includes the I/O time, communication time, idle (waiting) time, and the encoding time. The idle time is the time period during which a compute node waits for data from an I/O node.

From Table 4, the buffer size of 3 frames gives better performance, this is due to less communication and waiting overhead as compared to the buffer size of 1 frame. For the buffer size of 3 frames, the compute nodes need to send requests and wait for data for 4 times per GOP (as compared to 12 times for buffer size of 1) which results in less overall waiting time. Figure5 and Figure6 include plots of the frame rates for the scheme as well as the ideal frame rate (computed using Equation (7)) using the buffer size of 1 and 3 frames. The waiting times for buffer size of 3 frames are negligible as compared to those with 1 frame.

Table 5 and Table 6 give the results for the other two SIF sequences where the disk writing times are also shown; buffer size of 3 frames is used for these sequences. The maximum frame rate of 71 frames per second is achieved for *flower garden* sequence using 126 processors while reserving 6 processors as I/O nodes, that is, using all of the 6 PFS disks.

The frame rates for the CCIR-601 sequences *football* and *susie* are given in Figure7 and Table 7 respectively. The results for CCIR-601 sequences are obtained using buffer size of 1 frame. Buffer size of 3 frames could not be used for larger group sizes because of the limited available memory of an I/O node. For example, the memory required for a group size of 20 is more than 40MB

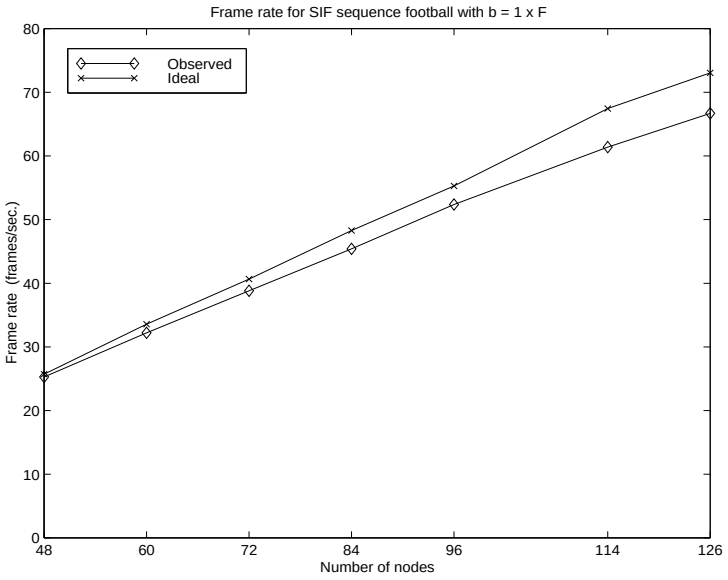

Fig. 5. The frame rate for SIF sequence *football* for $b = F \times 1$.

while the memory available to a user program on a Paragon node is about 25 MB.

Comparing the obtained results for our scheme with those of the schemes proposed in [22], the performance of our scheme is considerably higher. This is an approximate comparison, as [22] has used a frame size of 506880 bytes while in our experiments an even larger frame size of 691200 bytes (resulting from a frame size of 720×480 pixels with 4:2:2 chroma format) is used. Furthermore, our work is based on MPEG-2 while that of [22] is based on MPEG-1. For 125 Paragon processors, the frame rate reported in [22] is 8 frames/sec., while we achieved a frame rate of about 13 frames/sec. (which should be 17 frames/sec. if we normalize the two frame sizes) using 126 processors. Thus, our scheme yields a two fold increase in encoding rate.

7 Scalability of the Scheme

First, we will determine the largest possible group size or how many compute nodes can be supported by a single disk without any significant waiting time on compute nodes. Assuming the buffer size b to be equal to the size of 3 frames, Equation (2) can be written in terms of reading and encoding times as:

$$T_{open} + m \times (T_{read} + T_{send}) \leq T_{enc} + T_{recv}. \tag{8}$$

where T_{read}, T_{send}, T_{recv} and T_{enc} are the times to read, send, receive and encode the data of size b bytes, respectively. The overhead of receiving the request

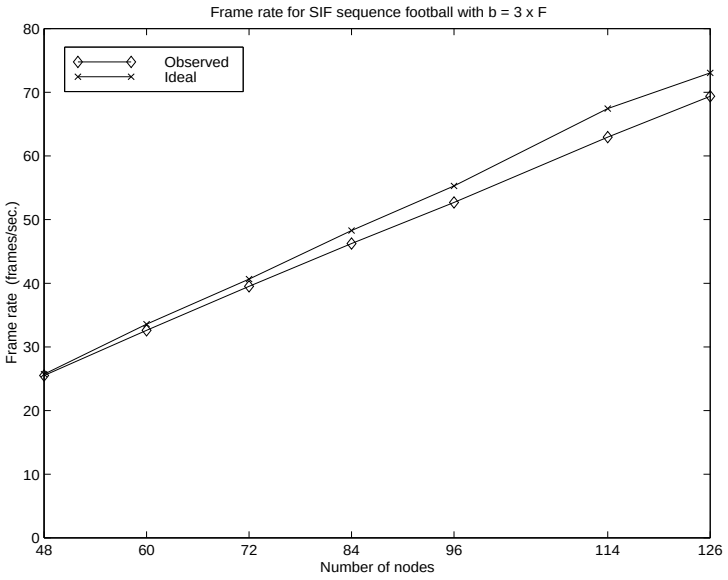

Fig. 6. The frame rate for SIF sequence *football* for $b = F \times 3$.

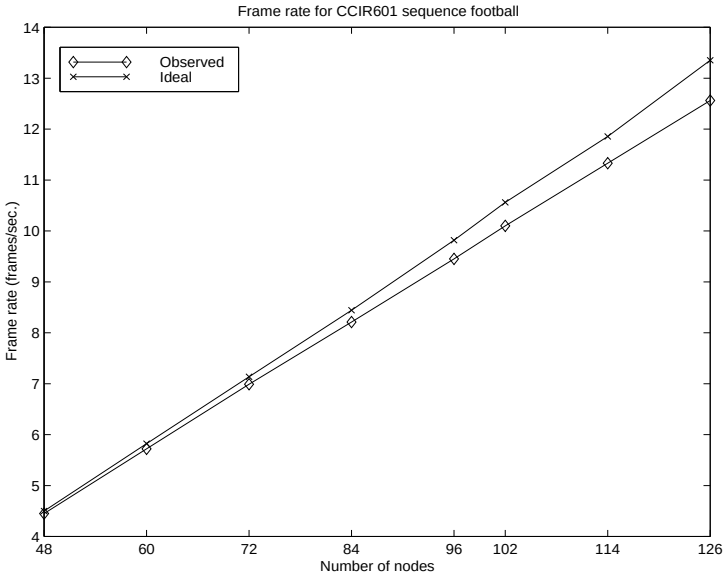

Fig. 7. The frame rate for CCIR-601 sequence *football*.

messages from compute nodes is assumed to be negligible and not taken into account in the above equation. The values of T_{read} and T_{enc} are independent of the group size. Experimental results show that the values of T_{send} and T_{recv} do not vary significantly (for group sizes of 7 to 20), therefore the maximum group size m_{max} can be estimated from the above equation as:

$$m_{max} = \frac{T_{enc} + T_{recv} - T_{open}}{T_{read} + T_{send}}. \tag{9}$$

If we keep the group size less than or equal to m_{max}, the specified constraint in Equation (2) will be preserved. Note that, by definition, $m = (N - D)/D, \Rightarrow D = \lceil N/(m+1) \rceil$, where D is the number of disks to use. If we use D disks

Table 5
Encoding rates for SIF (352×240) *flower garden* sequence.

No. of Processors	Group Size	$b = F \times 3$		
		Frames/sec.	Avg. Idle time (msec.)	Avg. Writing time (msec.)
48	7	26.49	8.81	729.33
60	9	33.48	12.08	827.48
72	11	40.93	14.99	717.81
84	13	48.37	15.10	991.23
96	15	55.82	44.58	980.42
114	18	64.76	103.63	1707.96
126	20	71.96	326.17	2056.83

Table 6
Encoding rates for SIF (352×240) *table tennis* sequence.

No. of Processors	Group Size	$b = F \times 3$		
		Frames/sec.	Avg. Idle time (msec.)	Avg. Writing time (msec.)
48	7	26.51	4.43	161.13
60	9	33.52	4.21	525.54
72	11	40.98	5.47	378.54
84	13	48.04	5.29	867.72
96	15	54.99	5.37	801.98
114	18	65.35	6.35	1458.02
126	20	69.60	6.63	2454.08

subject to m_{max}, the degree of parallelism will be maximum, and the ideal frame rate can be determined theoretically.

On the other hand, with an increase of available number of disks, one I/O node needs to serve less number of compute nodes in a group, thereby reducing the disk access time, data reading time and communication time. In that case, waiting time would be introduced in the I/O node. Therefore, the frame rate would be determined by the minimum finish time of compute nodes (which would be larger than the serving time of I/O node).

In order to determine the ideal frame rate beyond 126 nodes (the number of nodes we have used for experiments), we note that from Equation (6),

Table 7
Encoding rates for CCIR-601 (720×480) *susie* sequence.

No. of Processors	Group Size	$b = F \times 1$		
		Frames/sec.	Avg. Idle time (msec.)	Avg. Writing time (msec.)
48	7	4.47	18.28	62.43
60	9	5.75	28.71	72.26
72	11	7.00	27.55	156.32
84	13	8.31	40.76	121.68
96	15	9.59	57.57	161.47
114	18	11.46	81.11	275.51
126	20	12.73	363.33	464.20

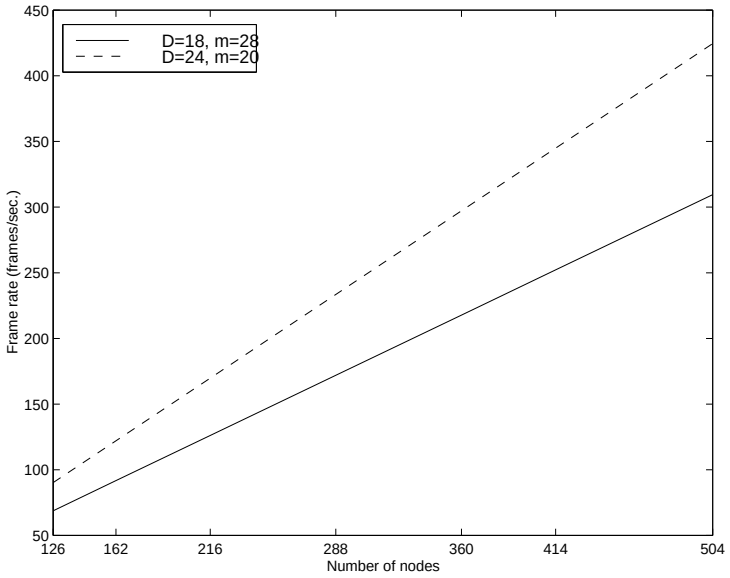

Fig. 8. Scalability for the SIF sequence *football*, no degradation of degree of parallelism.

$t_{enc} = (N - D)/f$. Therefore Equation (9) can be rewritten as:

$$m = \frac{B \times t_{enc} + T_{recv} - T_{open}}{T_{read} + T_{send}} \tag{10}$$

$$\Rightarrow f = \frac{B \times (N - D)}{m \times (T_{read} + T_{send}) + T_{open} - T_{recv}}. \tag{11}$$

In Figure8 and Figure10, frame rates are plotted against the number of processors using different number of disks. In Figure8, we consider the case of SIF sequence *football* for 512 nodes with $m = 28$ and $m = 20$, which gives

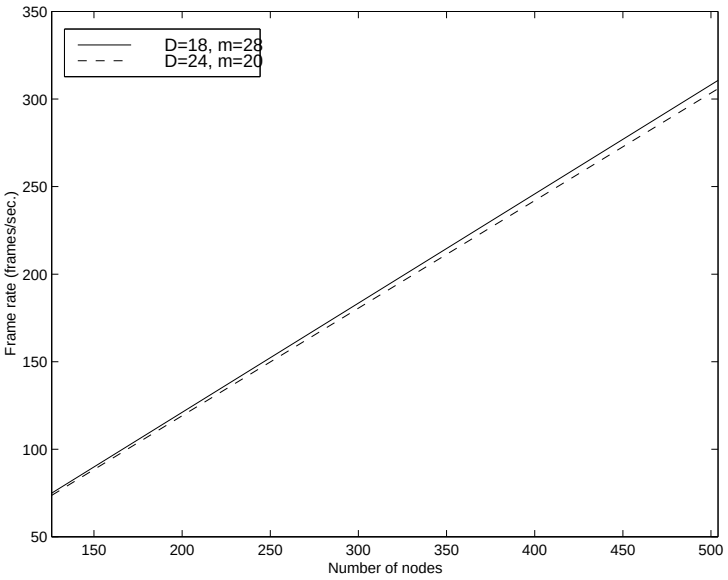

Fig. 9. Scalability for the SIF sequence *football*, degree of parallelism degraded.

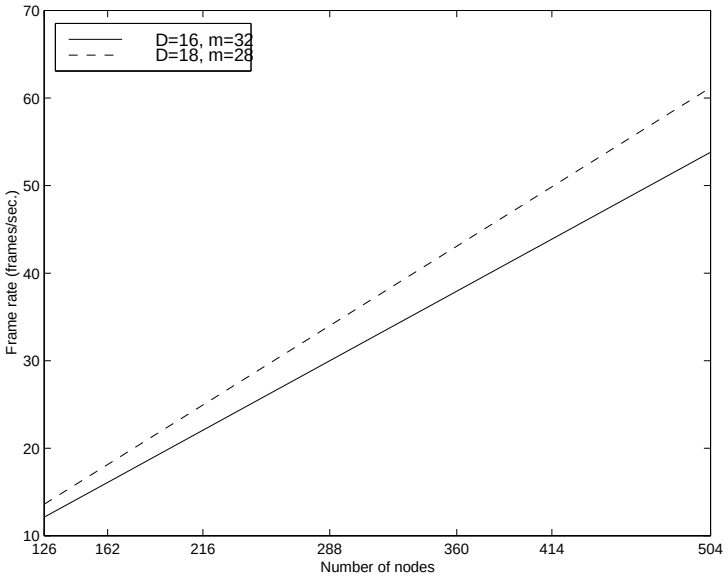

Fig. 10. Scalability for the CCIR-601 sequence *football*, no degradation of degree of parallelism.

$D = 18$, and $D = 24$, respectively. As long as D disks are available, it can be seen from Figure8 that the frame rate increases linearly with an increase in the number of processors. If the number of disks used is larger than D, assuming no degradation of degree of parallelism (that is, serving time of I/O node is equal to the minimum finish time of compute nodes), the scalability of the scheme can likewise be observed from Figure8. Similar observation can be made from Figure10 which depicts the case of CCIR-601 sequence *football*. Figure9 and Figure11 depict the scalability, if the above assumption does not hold, and the minimum finish time of compute nodes is larger than the serving time of I/O node.

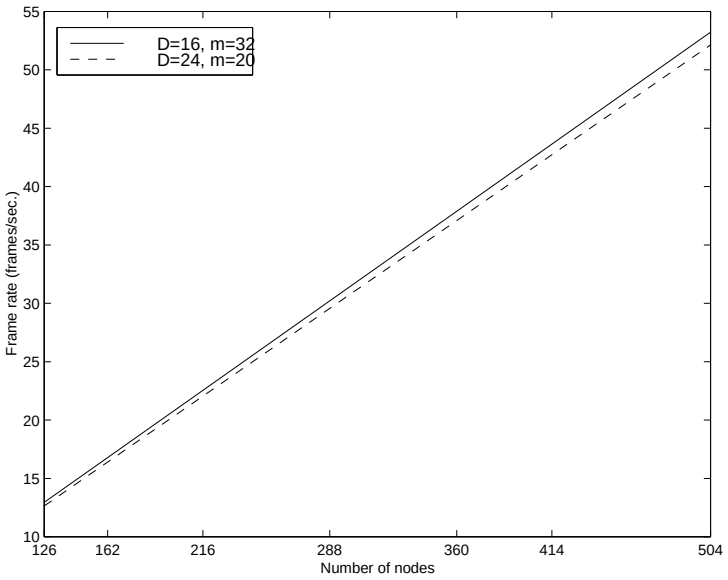

Fig. 11. Scalability for the CCIR-601 sequence *football*, degree of parallelism degraded.

However, if D disks are not available, an inevitable waiting time will be introduced in the compute nodes. Waiting time in the compute node is determined as:

$$T_{wait} = T_{open} + m \times (T_{read} + T_{send}) - (T_{enc} + T_{recv}). \tag{12}$$

In that case,

$$f = \frac{B \times (N - D)}{T_{open} - T_{recv} + T_{wait} + m \times (T_{read} + T_{send})}. \tag{13}$$

Therefore, the scheme scales gracefully to any number of processors as long as the required disks are available and the communication network provides enough bandwidth for the communication of all the I/O nodes with the compute nodes in their groups.

Example 1. Let us consider an example for the SIF (360×240) sequence football. Using the buffer size b equal to the size of 3 frames, we experimentally determine $T_{open} = 0.101$ sec., $T_{read} = 0.16$ sec., $T_{send} = 0.0048$ sec., $T_{recv} = 0.0048$ sec., and $T_{enc} = 4.77$ sec. Therefore, from Equation (9), the maximum group size m_{max} is 28.

Example 2. Let us consider another example for the CCIR-601 sequence football. Using the buffer size b equal to the size of 1 frame, we experimentally determine $T_{open} = 0.101$ sec., $T_{read} = 0.226$ sec., $T_{send} = 0.056$ sec., $T_{recv} = 0.056$ sec., and $T_{enc} = 9.20$ sec. Therefore, from Equation (9), the maximum group size m_{max} is 32.

Example 3. Given a total number of 512 nodes, we can find the minimum number of disks required. Consider the sequence in Example 1. For $m_{max} = 28$, we determine $D = 18$. Similarly, in case of example 2, for $m_{max} = 32$, the calculated $D = 16$.

8 Conclusions

We have proposed a parallel MPEG-2 encoder that has been implemented on the Intel Paragon. The proposed scheme partitions the system into groups of compute nodes, and I/O nodes, and can be easily extended to other MIMD machines or a set of networked workstations. The proposed encoder optimizes the system performance by balancing the computation, I/O, and the disk usage. The proposed encoder has achieved the highest level of performance reported for such a problem, with a frame rate of 71 frames per second for SIF (352×240 pixel) sequence using 126 Paragon processors. Using the CCIR-601 sequence, we have achieved about two times better performance compared to a previous study with the best results. The performance of the proposed scheme scales with the number of processors.

9 Acknowledgments

This research was supported by the Hong Kong Telecom Institute of Information Technology and Hong Kong Research Grants Council under grant number HKUST 759/96E.

References

[1] T. Akiyama, et al., MPEG2 Video Codec Using Image Compression DSP, *IEEE Trans. on Consumer Elec.*, 40 (3) (1994), 466-472.

[2] S. M. Akramullah, I. Ahmad, and M. L. Liou, A Data-parallel Approach For Real-time MPEG-2 Video Encoding, *Journal of Parallel and Distrib. Comp.*, 30 (2) (1995),129-146.

[3] S.M. Akramullah, I. Ahmad, and M. L. Liou, Performance of Software-Based MPEG-2 Video Encoder on Parallel and Distributed Systems, *IEEE Trans. on Circuits and Syst. for Video Tech.*, 7 (4) (1997), 687-695.

[4] A. C. Downton, Generalized Approach to Parallelising Image Sequence Coding Algorithms, *IEE Proc.-Vis. Image Signal Process.*, 141 (6) (1994), 438-445.

[5] B. Furht, *Multimedia Tools and App.*, (Kluwer, 1996).

[6] D. J. Le Gall, MPEG: A Video Compression Standard for Multimedia Applications, *Comm. of the ACM*, 34 (4) (1991), 46-58.

[7] K. L. Gong and L. A. Rowe, Parallel MPEG-1 Video Encoding, in *Proc. of 1994 Picture Coding Symp.*, Sacramento, CA, Sept. 1994.

[8] B.G. Haskell, A. Puri, and A.N. Netravali, *Digital Video: An Introduction to MPEG-2*, Digital Multimedia Standards Series, (Chapman and Hall, 1997).

[9] R. Hopkins, Digital Terretrial HDTV for North America: the Grand Alliance HDTV System, *IEEE Trans. on Consumer Elec.*, 40 (3) (1994), 185-198.

[10] Z. Huang, Y. Takeuchi and H. Kunieda, Distributed Load Balancing Schemes for Parallel Video Encoding System, *IEICE Trans. Fundamentals* E77-A (5) (1994), 923-930.

[11] H.-C. Huang and J.-L. Wu, New Generation of Real-time Software-based Video Codec: Popular Video Coder II (PVC-II), in *Proc. of the SPIE*, 2419 (1995), 329-339.

[12] Intel Scalable Systems Division, Intel Corporation, *Paragon System User's Guide*, (1996).

[13] ISO/IEC, *Draft International Standard ISO/IEC 10918-1*, Digital compression and coding of continuous-tone still images, (1991).

[14] ISO/IEC, *Draft International Standard ISO/IEC 11172-2*, Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to 1.5 Mbits/s, (1991).

[15] ISO/IEC, *ISO Committee Draft 13818-2*, Generic Coding of Moving Pictures and Associated Audio, (1993).

[16] R. Jain, K, Somalwar, J. Werth and J. C. Browne, Heuristics for Scheduling I/O Operations, *IEEE Trans. on Parallel and Distrib. Syst.*, Vol. 8., No. 3, Mar. 1997, pp. 310-320.

[17] M. L. Liou, Overview of the $p \times 64$ kbits/s video Coding Standards, *Comm. of the ACM*, 34 (3) (1991), 59-63.

[18] MPEG Software Simulation Group, *MPEG-2 Video Encoder, Version 1.1a*, (1994).

[19] P. Moulin, A. T. Ogielski, G. Lilienfeld and J. W. Woods, Video Signal Processing and Coding on Data-Parallel Computers, *Digital Signal Processing*, 5 (1995), 118-129.

[20] J. Nang and J. Kim, An Effective Parallelizing Scheme of MPEG-1 Video Encoding on Ethernet-Connected Workstations, in *Proc. of Intl. Conf. on Adv. in Parallel and Distrib. Comp.*, (1997), 4-11.

[21] Ke Shen, Lawrence A. Rowe, Edward J. Delp, A Parallel Implementation of an MPEG 1 Encoder: Faster Than Real-Time, in *Proc. of the SPIE*, 2419 (1995), 407-418.

[22] Ke Shen and Edward Delp, A Spatial-Temporal Parallel Approach For Real-Time MPEG Video Compression, in *Proc. of 1996 Intl. Conf. on Parallel Processing*, 2 (1996), 100-107.

[23] H. H. Taylor, et al., An MPEG Encoder Implementation on the Princeton Engine Video Supercomputer, in *Proc. of Data Compression Conf. 1993*, (Los Alamitos, CA, 1993), 420-429.

[24] Y. Yu and D. Anastassiou, Software Implementation of MPEG-II Video Encoding Using Socket Programming in LAN, in *Proc. of the SPIE*, 2187 (1994), 229-240.