Design and Implementation of Parallel Video Encoding Strategies Using Divisible Load Analysis

Ping Li, Bharadwaj Veeravalli, *Member, IEEE*, and Ashraf A. Kassim, *Member, IEEE*

***Abstract*—The processing time needed for motion estimation usually accounts for a significant part of the overall processing time of the video encoder. To improve the video encoding speed, reducing the execution time for motion estimation process is essential. Parallel implementation of video encoding systems using either the software or the hardware approach has attracted much attention in the area of real time video coding. In this paper, we attempt to implement a video encoder on a bus network. Usually, for such a parallel system, the key concern is associated with partitioning and balancing of the computational load among the processors such that the overall processing time of the video encoder is minimized. With the use of the *divisible load theory* (DLT) paradigm, a strip-wise load partitioning/balancing scheme, a load distribution strategy, two implementation strategies are developed to exploit the data parallelism inherent in the video encoding process. The striking feature of our design is that,both the granularity of the load partitions and all the associated overheads caused during parallel video encoding process can be explicitly considered. This significantly contributes to the minimization of the overall processing time of the video encoder. Extensive experimental studies are carried out to test the effectiveness of the proposed strategies. The performance of the parallel video encoder is quantified using the metrics *speedup* and *performance gain*, respectively. The experimental results show that our strategies are effective for exploiting the available parallelism inherent in the video encoding process and provide a theoretical insight on how to analytically quantify and minimize the overall processing time of a parallel system. The proposed strategies can be easily extended and applied to improve other existing parallel systems.**

***Index Terms*—Block matching motion estimation, bus network, divisible load theory (DLT), load distribution, load partitioning/balancing, parallel video coding.**

I. Introduction

DIGITAL video compression techniques have played a crucial role in the domain of telecommunications and multimedia systems. The way in which a video is encoded usually decides the cost of its storage and transmission. Besides, with the increasing usage of multimedia applications such as mobile communications, virtual reality, video phone/conferencing, etc., real time video coding has become essential. However, currently due to the complex nature of video encoding, achieving real time [24–30 frames per second (f/s)] video coding in a single-processor environment is often a difficult task. Parallel implementation of the video coding system becomes a natural option.

Parallel video coding can be achieved by either the special-purpose hardware or the software implementation using general-purpose computing platforms. Based on the overall consideration on cost, flexibility, portability, and scalability, software approach is more attractive and thus is used in this paper to design a parallel video encoder implemented on a bus network. Below, we give an introduction to some related work in this direction.

In [3], block matching motion estimation of a MPEG-2 [11] video encoder is parallelized using the *single-program multiple-data* (SPMD) programming paradigm. The video frame is equally partitioned into a number of blocks equal to the number of processors. This scheme is commonly referred to as *equal-partition scheme*. To eliminate the interprocessor communication that otherwise is necessary to exchange the decoded blocks between the slave processors, the video frame is partitioned and distributed in an overlapped fashion such that each processor is assigned all the required data in the initial communication phase. Thus, all the processors may perform their computation independently and concurrently. However, there is no indication about how the overall processing time is calculated and minimized in the paper. The results reported are completely based on experimental studies. A software-based MPEG-4 [12] video encoder using parallel processing is reported in [4]. A scheduling algorithm for exploiting the control parallelism between the coding processes of different video object planes (VOPs) and a shape-adaptive load partitioning method for exploiting the data parallelism inherent in the coding process of each VOP are developed. The proposed shape-adaptive load partitioning method is, in principle, similar to the equal-partition scheme. After taking into account the computation complexity of three kinds of macroblocks (MBs) (*contour MB*, *standard MB*, and *transparent MB*), this scheme guarantees that the actual computational load assigned to each processor is equal. Again, a quantitative analysis of the processing time is not reported. In [5], four parallel video encoding algorithms for H.261 [13] are proposed and evaluated. The first two algorithms referred to as *Spatial Parallel Algorithm-S1* and *Spatial Parallel Algorithm-S2* are purely spatial parallelism based. However, the latter two are referred to as *Spatial-Temporal Parallel Algorithm-ST1*, and *Spatial-Temporal Parallel Algorithm-ST2* attempt to exploit both the spatial and the temporal parallelism. The equal-partition scheme is used to partition the video frames and interprocessor communication between the slave processors is used in *S2*, *ST3*, and *ST4* to

Manuscript received August 14, 2003; revised Dedember 18, 2003 and July 19, 2004. This paper was recommended by Associate Editor K.-H. Tzou.

P. Li is with the Design Technology Institute, Faculty of Electrical Engineering, Eindhoven University of Technology, 5600 MB Eindhoven, The Netherlands (e-mail: p.li@tue.nl).

B. Veeravalli is with the Open Source Software Laboratory, National University of Singapore, Department of Electrical and Computer Engineering, 117576 Singapore (e-mail: elebv@nus.edu.sg).

A. A. Kassim is with the Vision and Image Processing Laboratory, National University of Singapore, 117567 Singapore (e-mail: eleashra @nus.edu.sg).

Digital Object Identifier 10.1109/TCSVT.2005.852627

minimize the communication latency. Again, the drawback is that the equal partitioning of the data among processors will not minimize the overall processing time. Furthermore, the inclusion of the interprocessor communication may complicate the software and hardware structures.

As observed from above, the equal partitioning of the data among the processors is predominantly used in most of the implementations thus far to realize a parallel video encoder. This is partly due to the fact that, by and large, the effects of communication delays and overhead incurred in data distribution phase, interprocessor communication and solution collection phase, are difficult to track and therefore they are usually neglected when designing a load partitioning/balancing scheme. However, for parallel implementation of a video encoder on a bus network, time delays for data exchange and solution collection can be substantial when large number of processors are to be used. When processing time minimization is to be carried out, we have to account for all possible delays that affect the time performance starting from the load distribution phase to the solution transfer back phase. Another point to note is that, the performance improvement in terms of the overall processing time achieved by the equal-partition scheme will increase with the number of processors when one neglects the communication and computation overheads. However, in real-life situations, this gain does not stretch too far as the effect of overheads tend to dominate as shown in [14] and [22] with the increase of the processors, amidst communication latencies and the associated overheads.

Since the predominantly used equal-partition scheme has certain disadvantages as described above, in this paper, we attempt to design a load partitioning/balancing scheme and a load distribution strategy to effectively partition the video frames among the processors using the divisble load theory (DLT) paradigm. This is very important, especially when the network speed is slow, distributing and collecting the data to and from the processors may consume quite a large amount of time. If the data assigned to the processors is not carefully balanced, the performance improvement by parallel processing may not compensate the performance loss caused by the communication latencies and associated overheads. In this paper, we also propose two implementation strategies of the video encoder to fully exploit the data parallelism inherent in the video encoding process. The DLT paradigm provides a generic, direct and intuitive approach for load partitioning and balancing of computation-intensive workloads on network-based computing systems and therefore is an ideal tool that we can use to design the parallel video encoder.

In the recent literatures, several practical applications were shown to gain a significant performance improvement using DLT paradigm. These include, image processing applications such as large-size image processing [20], database operations [19], matrix-vector product computations [18]. In [18] rigorous experimental implementation of the matrix-vector products on PC clusters as well as on a network of workstations (NOWs) were carried out and in [19] several other applications such as *pattern search*, *file compression*, *joining operation in relational databases*, *graph coloring and genetic search* were attempted using the DLT paradigm. In [17], a record search algorithm is

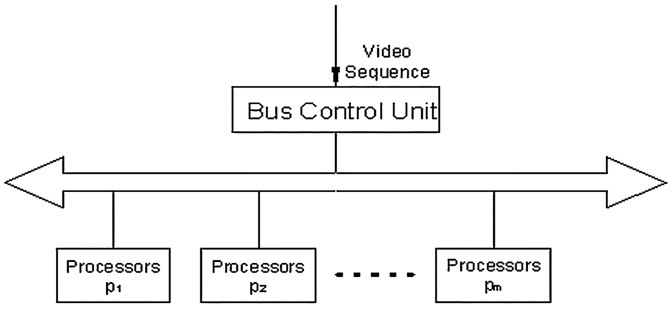


Fig. 1. Bus network topology.

designed for database applications. It must be noted that all the above attempted applications are of computationally intensive nature with very large size loads to process. In our problem context, the nature of loads to be handled are very large and demand a very large processing time. Thus, DLT becomes a natural choice in realizing a parallel video encoder.

A. Our Contributions

For the first time in domain of parallel video coding, we apply the DLT paradigm to the video encoding process. A strip-wise load partitioning/balancing scheme and a load distribution strategy are developed to effectively partition the video frames among the processors. Two implementation strategies that are able to effectively exploit the data parallelism inherent in the video encoding process are proposed. The striking feature of our design is that, using DLT paradigm, both the granularity of the load partitions and all the associated overheads caused during parallel video encoding process can be explicitly considered. This greatly contributes to the minimization of the overall processing time of the video encoder. The proposed strategies give us a theoretical insight on how to analytically quantify and minimize the processing time and all associated overheads caused in the parallel processing process. The parallel video encoder designed in this paper can and is shown to achieve a good performance.

The organization of this paper is as follows. In Section II, we present the system model and the video encoder implemented in this paper. In Section III, we describe the mathematical model that we use to construct the parallel video encoder. In Section IV, we describe two implementation strategies of the video encoder that are developed to effectively exploit the data parallelism inherent in the video encoding process. In Section V, we present the experimental results obtained in our implementations. The performance of the strategies is quantified using metrics *speedup* and *performance gain*, respectively. In Section VI, we conclude the paper with some possible future extensions to the problem addressed in this paper.

II. System Model and the Video Encoder

The system model is shown in Fig. 1. The network comprises of a bus control unit (BCU) and a set of processors $p_1, \ldots, p_m$ connected by a bus. The video sequence to be compressed is assumed to arrive at BCU. The network may have either a dedicated BCU that is only responsible for load distribution or a general-purpose scheduler that, besides partitioning and assigning

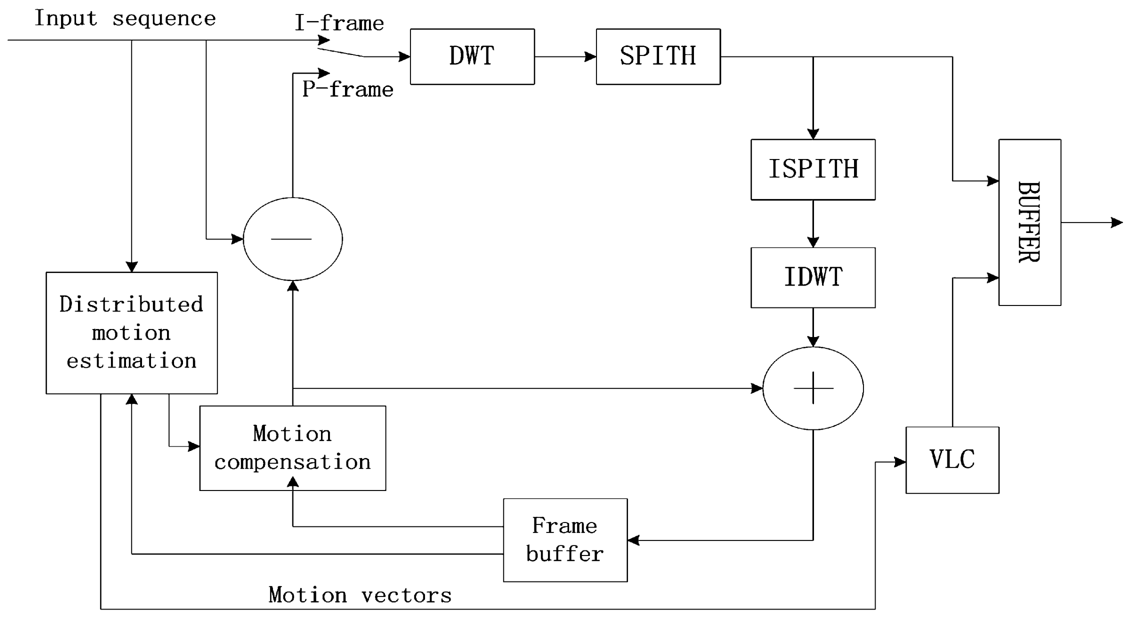


Fig. 2. System block diagram of the video encoder.

the data to the processors, will be also responsible for computing of the data. In this paper, we consider the case when the network has a general-purpose BCU that not only partitions the video frames and distributes them to the individual processors but also carries out the computations including discrete wavelet transform (DWT), set partitioning in hierarchical tree (SPIHT), inverse set partitioning in hierarchical tree (ISPIHT), inverse discrete wavelet transform (IDWT), motion compensation (MC) and variable length coding (VLC) of the motion vectors. Thus, in our system, all algorithms except the motion estimation are conducted by BCU, whereas the processors p_1 to p_m are responsible only for the motion estimation.

A. Video Encoder

Fig. 2 shows the system block diagram of the video encoder that is implemented in this paper, which is similar to the one proposed in [7]. The major difference is that, in our encoder, SPIHT is used for coding of the wavelet coefficients while in the codec reported in [7], the zero tree wavelet (ZTW) coding algorithm [8] is used. Below, we shall describe briefly the feature of our video encoder.

Block matching motion estimation (BMME) technique is used to detect the local motion and the block motion compensation technique is employed for block prediction. The video frame is partitioned into a number of 16×16 blocks and each block if estimated from the reference frame using one motion vector. After motion compensation, all blocks in current frame are predicted by the corresponding blocks from the reference frame. In order to remove the temporal redundancy, the blocks in current frame is subtracted by the predicted blocks in reference frame to obtain the residual blocks. The residue blocks are then pieced together to form a complete residual frame, upon which the two-dimensional DWT are applied to remove the spatial redundancy. The wavelet coefficients are efficiently encoded using SPIHT algorithm.

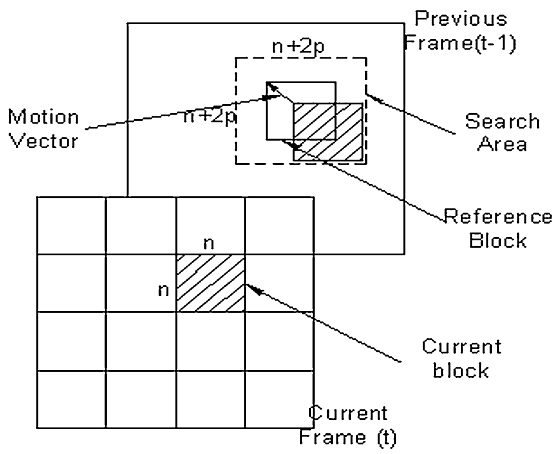


Fig. 3. BMME.

Although Fig. 2 shows the entire video encoder, our focus is on the BMME part. We shall now describe in brief on how BMME is conducted. In block matching process, two frames are required and they are the *current frame* and the *previous frame*, respectively. The current frame is divided into blocks of size $m \times n$. For each block (current block) in the current frame, the previous frame is searched within a neighborhood (search area) in order to determine the closest matching block with respect to a specified error criterion. This process is illustrated by Fig. 3. More details on specific algorithms can be found in [6].

From the above procedure, we observe that there is a considerable scope for exploiting data parallelism inherent in the BMME process. Given the search area and the current block, the BMME of each block is independent with each other and therefore can be conducted concurrently. The only restrictive aspect is that, besides the current block and the reference block, the individual processors also demand the surrounding blocks from other processors to form the search area. In our implementations, we assign such additional surrounding blocks in the initial communication phase. Thus, no interprocessor communication is required. This tremendously simplifies our design complexity.

III. Mathematical Model

In this section, the load partitioning/balancing scheme and load distribution strategy that are required for parallelization of the video encoder will be presented. The DLT paradigm on which our design is based will be briefly explained.

A. Load Partitioning/Balancing

In general, what affects processing time most is the size of the minimum data that can be processed by the processor, which is often referred to as *computational granularity* [2]. Also, in reality, there exists a minimum amount of data, referred to as *granularity*, that can be assigned to the processor for computation. Usually, the smaller the size of the granularity, the more processors (also referred to as more degree of parallelism) may be used which may improve the performance. However, the communication latency and the associated overheads incurred during the course of communication and computation increase with the number of the processors. If the network speed is not high enough compared to the processor speed, the latencies and overheads may dominate and results in a counterproductive effect on the overall processing time. Thus, the size of the granularity should be a serious consideration when we attempt to partition the data for any actual parallel system.

To strike a balance between the degree of parallelism and the communication overhead, a strip-wise load partitioning/balancing scheme is developed to partition the video frames among the processors. In this scheme, each frame is partitioned into rows of 16×16 blocks and one row of blocks in the frame is considered as the minimum data (granularity and referred to as one data unit in our schemes) that can be assigned to an individual processor. Fig. 4 illustrates the load partitioning process using the proposed scheme, where m is the number of processors used in the system.

There are several other load partitioning schemes existing in the literatures such as row-wise, column-wise, block-wise partitioning, etc. [23]. This paper does not examine the problem of optimal partitioning scheme. Since the main objective of the paper is focused on designing a theoretical or analytical method to quantify and minimize the overall processing time of parallel systems, determination of the optimal size of the basic unit is not our major concern. We believe the stripe-wise load partitioning scheme proposed in this paper is able to achieve a good balance between the overheads and the degree of parallelism for our parallel video encoding system, which is demonstrated by our results in Section V-C.

B. Load Distribution

We note that every processor needs to be assigned a portion of current frame and the corresponding portion of previous frame for block matching process. Since the strip-wise load partitioning scheme is used to partition the video frames, what we need to do is to determine size of the *strips*, i.e., the number of data units, to be assigned to the processors. To achieve this, the load distribution strategy proposed in [2] is used. It may be noted

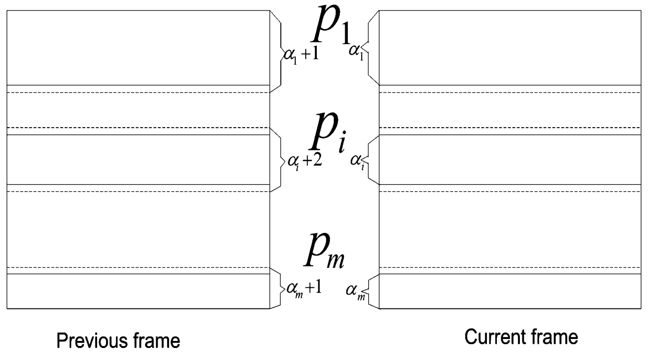


Fig. 4. Load partitioning process using the strip-wise scheme.

that the load distribution strategy proposed in [2] does not take into account the *solution-transfer-back* phase, during which the computed solutions obtained by the individual processors are transferred back to BCU. For our problem context, the solution-transfer-back phase is crucial because the resulting motion vectors in the processors must be transferred back to BCU for subsequent processing. Thus, the solution-transfer-back phase must be carefully planned so as not to clash with bus busy time.

Below, the load distribution strategy proposed in [2] is introduced and extended so that solution-transfer-back phase can be considered during the load distribution process. Some important notations that will be used throughout this paper are the following:

α_i	amount of the data (number of rows of blocks) assigned to processor p_i;
E_i	time for processor p_i to process one data unit;
C	time for BCU to transmit one data unit over the bus network;
θ_{cp}	additive computation overhead that includes all extra delays (the extra time for system switching, hardware initialization, software startup, etc.) associated with the computation process;
θ_{cm}	additive communication overhead that includes all extra delays (the extra time for system switching, hardware initialization, software startup, etc.) associated with the communication process;
m	number of processors involved in the parallel processing;
L	total number of data units to be scheduled;
λ_i	ratio of the data to be transferred back to BCU from processor p_i over the data assigned to it. In this application, we note that $0 \leq \lambda_i \ll 1$.

In the above, one data unit is defined as one row of blocks in the frame, i.e., the minimum data that can be assigned to the processor (granularity).

The load distribution process is described by means of a directed flow graph (DFG) [2] as shown in Fig. 5. In brief, DFG represents load distribution process and captures precedence relationships between communication and computation events. An in-depth description of DFG representation can be found in [2]. Note that besides *communication node* and *computation node* as explained in [2], a third type of node referred to as *solution-transfer-back* node is introduced in our DFG. The weight of the communication node i is given by the communication time to transfer a load α_i to p_i and weight of the computation

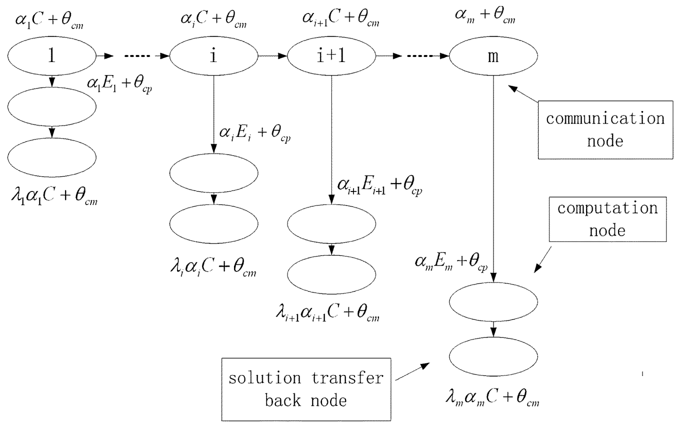

$\alpha_1 C + \theta_{cm}$
$\alpha_i C + \theta_{cm}$
$\alpha_{i+1} C + \theta_{cm}$
$\alpha_m + \theta_{cm}$
1
i
i+1
m
$\alpha_1 E_1 + \theta_{cp}$
communication node
$\alpha_i E_i + \theta_{cp}$
$\lambda_1 \alpha_1 C + \theta_{cm}$
$\alpha_{i+1} E_{i+1} + \theta_{cp}$
computation node
$\lambda_i \alpha_i C + \theta_{cm}$
$\alpha_m E_m + \theta_{cp}$
$\lambda_{i+1} \alpha_{i+1} C + \theta_{cm}$
solution transfer back node
$\lambda_m \alpha_m C + \theta_{cm}$

Fig. 5. Directed flow graph for load distribution in bus network with m processors.

node i is given by the computation time to process the load α_i by p_i. Similarly, the weight of a solution-transfer-back node i is the processing time for p_i to transfer the results to BCU. We shall now introduce some important definitions that will be used throughout this paper.

- **Load distribution**, denoted by α, is defined as an m-tuple $(\alpha_1, \alpha_2, \ldots, \alpha_m)$ such that

$$\sum_{i=1}^{m} \alpha_i = L. \tag{1}$$

  This equation is referred to as normalization equation.
- **Computation finish time path** (CFTP) of processor p_i, denoted as $T_i^{\text{CFTP}}(\alpha)$, is defined as the sum of the weights of the nodes starting from the communication node 1 till computation node i along the directed arrows, which is computed as follows:

$$T_i^{\text{CFTP}}(\alpha) = C \sum_{j=1}^{i} \alpha_j + \alpha_i E_i + i\theta_{cm} + \theta_{cp}. \tag{2}$$

- **Transfer-back Finish time path** (TFTP) of processor p_i, denoted as $T_i^{\text{TFTP}}(\alpha)$, is defined as the sum of the weights of the nodes starting from the communication node 1 till solution-transfer-back node i along the directed arrows, which is computed as follows:

$$T_i^{\text{TFTP}}(\alpha) = C \sum_{j=1}^{i} \alpha_j + \alpha_i(E_i + \lambda_i C) + (i+1)\theta_{cm} + \theta_{cp}. \tag{3}$$

- **Overall processing time** of the video encoder, denoted as $T(\alpha)$, is defined as the processing time for the video encoder to process one video frame.

It has been rigorously proved [9] that for optimal processing of a divisible load it is *necessary and sufficient* that all the processors that are participating in the computation must stop computing at the same time instant. However, this is not true when solution-transfer-back phase is to be considered explicitly. In this case, to achieve the optimal processing time, the processors should stop computing one after another in such a manner that they could start their solution-transfer-back phase one after another. We will briefly explain this in later discussion. By equating the CFTP of processor p_{i+1} to the TFTP of processor p_i, we obtain the following recursive equations:

$$\alpha_{i+1} C + \alpha_{i+1} E_{i+1} = \alpha_i E_i + \lambda_i \alpha_i C, \quad i = 1, 2, \ldots, m-1. \tag{4}$$

Note the communication and computation overheads are cancelled out when we equate CFTP to TFTP and thus do not appear in (4) explicitly. From (4) together with the normalization equation (1), we have m equations with m independent variables. The load partitions that minimize the processing time can be obtained by solving these recursive equations. The solutions of the recursive equations are shown below.

Rewriting (4), we have

$$\alpha_i = \frac{E_{i+1} + C}{E_i + \lambda_i C} \times \alpha_{i+1} = \frac{M_i}{N_i} \times \alpha_{i+1}$$

where

$$M_i = E_{i+1} + C, \quad N_i = E_i + \lambda_i C.$$

Thus

$$\alpha_i = \frac{M_i}{N_i} \frac{M_{i+1}}{N_{i+1}} \cdots \frac{M_{m-1}}{N_{m-1}} \times \alpha_m = f_i \times \alpha_m \tag{5}$$

where

$$f_i = \frac{\prod_{k=i}^{m-1} M_k}{\prod_{k=i}^{m-1} N_k}.$$

Substituting (5) into the normalization equation, we obtain

$$\alpha_m = \frac{L}{\sum\limits_{i=1}^{m} f_i}. \tag{6}$$

Thus, as shown above, the individual load partition α_i is obtained by using (5) and (6). From Fig. 5, it is clear that the overall processing time of the parallel processing process is equivalent to the TFTP of processor p_m. With the load partitions derived above, the TFTP of processor p_m is calculated as follows:

$$T_m^{\text{TFTP}} = \sum_{i=1}^{m} (\alpha_i C + \theta_{cm}) + (\alpha_m E_m + \theta_{cp}) + (\lambda_m \alpha_m C + \theta_{cm})$$

where, α_i are given by (5) and (6).

We now explain why we equate the CFTP of processor p_{i+1} to the TFTP of processor p_i to derive (4) for optimal processing time. Suppose we change the load partitions $(\alpha_1, \ldots, \alpha_p, \ldots, \alpha_q, \ldots, \alpha_m)$ derived using (5) and (6) to $(\alpha_1, \ldots, \alpha_p + \Delta, \ldots, \alpha_q - \Delta, \ldots, \alpha_m)$, where Δ is a small positive value, it can be seen that the overall processing time will increase by $\Delta(C + E_p + \lambda_p C - \lambda_q C)$. Similarly, suppose the load partitions derived above are changed to $(\alpha_1, \ldots, \alpha_p - \Delta, \ldots, \alpha_q + \Delta, \ldots, \alpha_m)$, then we observe that the overall processing time will increase by $\Delta(E_q + \lambda_q C)$. Hence, any incremental change to the load paritition derived from the above equations lead to an increase in processing time. Thus, the load partitions computed by equating the CFTP of processor p_{i+1} to the TFTP of processor p_i are optimal for our problem where the single-installment load distribution strategy is used.

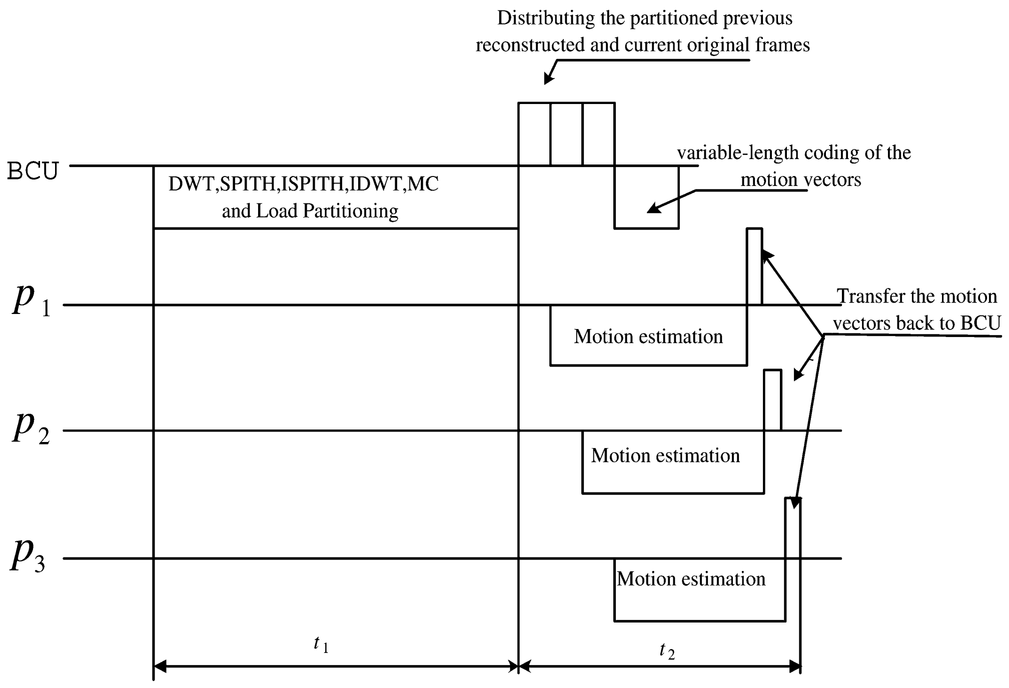

Distributing the partitioned previous reconstructed and current original frames
BCU
DWT,SPITH,ISPITH,IDWT,MC and Load Partitioning
variable-length coding of the motion vectors
p_1
p_2
p_3
Motion estimation
Transfer the motion vectors back to BCU
t_1
t_2

Fig. 6. Timing diagram of the video encoder using Strategy I.

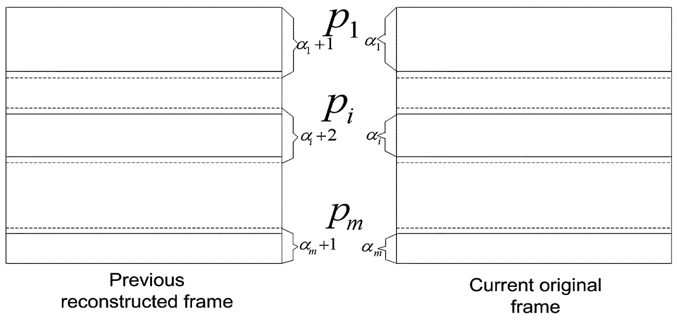


Fig. 7. Load partitioning process for Strategy I.

Obviously, the load partitions obtained by solving the recursive equations may not be integers. For our problem of parallelization of BMME process, the video frames must be partitioned and assigned to the processors in terms of integer number of data units, i.e., one row of the blocks. To restrict the non-integer load partitions derived above to integer values, an integer approximation algorithm proposed for the single-installment strategy in [2] is used.

IV. Implementation Strategies

The motion vectors obtained in the video encoder are of half-pel precision. To obtain the half-pel motion vectors, the entire BMME process is divided into two steps; the full-pel motion estimation (FPME) to get full-pel motion vectors and the half-pel motion estimation (HPME) to refine the full-pel motion vector to half-pel precision. Generally, the HPME is performed on the previous reconstructed frame. However, the FPME can be done either on the previous reconstructed frame or on the previous original frame [7]. This leads to our two implementation strategies of the video encoder. They are

Strategy I: both FPME and HPME are performed on previous reconstructed frame;

Strategy II: FPME is performed on previous original frame and HPME is performed on the previous reconstructed frame.

Below, we will give a more detailed discussion on these two implementation strategies.

A. Analysis of Strategy I

In this strategy, all algorithms except the BMME are carried out by BCU. Fig. 6 shows the timing diagram of the video encoder when three processors are used. We see the overall processing time of the encoder is the sum of t_1 and t_2, where t_1 is the execution time for the preprocessing phase[1] and t_2 the processing time for the parallel motion estimation process. t_1 can be computed by multiplying the processing speed of BCU with the number of data units processed by it. The processing speed of BCU can be measured and is observed to be a more-or-less constant value. The number of data units processed by BCU equals to the total number of the rows of the blocks in a video frame, which is a constant value. Thus, t_1 is more-or-less constant. It is the t_2 that varies with the amount of the data assigned to the individual processors. Thus, to minimize the overall processing time (t_1+t_2), focus is on minimizing t_2 by adjusting the amount of the data assigned to the processors.

The load partitioning process using the strip-wise load partitioning/balancing scheme is shown in Fig. 7, in which m is the number of processors (excluding BCU) used in the parallel system. The directed flow graph of the load distribution process is illustrated by Fig. 8. The weight of communication node i is given by $(2\alpha_i + 2)C + \theta_{cm}$, where α_i is the number of data units assigned to processor p_i; C is the processing time needed

[1]In Strategy I, preprocessing phase is defined as the processing before the load distribution process by BCU.

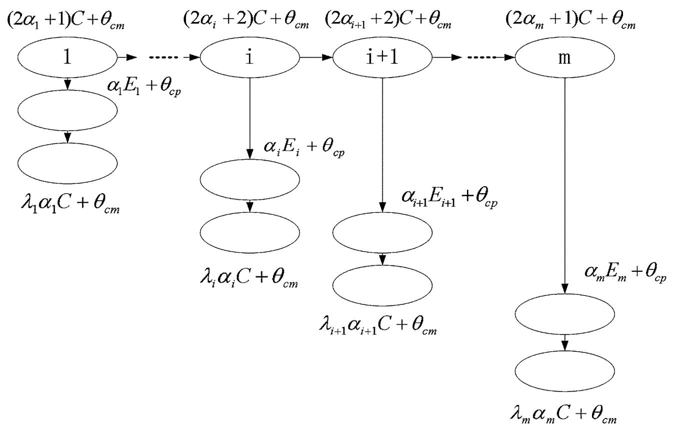

$(2\alpha_1+1)C+\theta_{cm}$
$(2\alpha_i+2)C+\theta_{cm}$
$(2\alpha_{i+1}+2)C+\theta_{cm}$
$(2\alpha_m+1)C+\theta_{cm}$
1
i
i+1
m
$\alpha_1E_1+\theta_{cp}$
$\alpha_iE_i+\theta_{cp}$
$\alpha_{i+1}E_{i+1}+\theta_{cp}$
$\alpha_mE_m+\theta_{cp}$
$\lambda_1\alpha_1C+\theta_{cm}$
$\lambda_i\alpha_iC+\theta_{cm}$
$\lambda_{i+1}\alpha_{i+1}C+\theta_{cm}$
$\lambda_m\alpha_mC+\theta_{cm}$

Fig. 8. Directed flow graph of the parallel BMME process for Strategy I.

for transferring one data unit over the network; θ_{cm} is the communication overhead. As explained in Section II, we assign the additional data that is required by processor p_i to conduct the BMME process at the initial communication phase. The second "2" in the weight means that, besides the α_i number of rows of blocks in both the current and reference frames, two additional rows of blocks above and below the strip should be assigned to processor p_i as well. The weights of the communication nodes 1 and m are slightly different from others. As shown by Fig. 7, since there is no block above the first strip assigned to processors p_1 and no block below the last strip assigned to processor p_m, the search areas required for the BMME process of the first and last rows of blocks in current frame are obtained simply by extending the reference frame. Thus, only one additional row of blocks needs to be transmitted to processors 1 and m at the initial communication phase, as is demonstrated by the "1" instead of "2" in the weights of the communication nodes 1 and m. The weight of computation node i is given by $\alpha_iE+\theta_{cp}$ and the weight of solution-transfer-back node i is given by $\lambda_i\alpha_iC+\theta_{cm}$, both of which have been explained in Section III.

From Fig. 8, by equating the CFTP of processor p_{i+1} to the TFTP of processor p_i, we obtain

$$\begin{aligned}&(2\alpha_{i+1}+2)C+\theta_{cm}+\alpha_{i+1}E_{i+1}+\theta_{cp}\\&\quad=\alpha_iE_i+\theta_{cp}+\lambda_i\alpha_iC+\theta_{cm},\quad i=1,2\ldots m-2 \quad (7)\\&(2\alpha_m+1)C+\theta_{cm}+\alpha_mE_m+\theta_{cp}\\&\quad=\alpha_{m-1}E_{m-1}+\theta_{cp}+\lambda_{m-1}\alpha_{m-1}C+\theta_{cm}. \quad (8)\end{aligned}$$

Again, together with the normalization equation, we have m equations with m independent variables. The load partitions can be obtained by solving these m recursive equations, as in Section III.

B. Analysis of Strategy II

From implementation perspective, Strategy I is simple and straightforward to implement. This is in fact an advantage of Strategy I. However, because of the strong data dependency between the preprocessing phase and the motion estimation phase, processors are not allowed to start the motion estimation process until BCU finishes its preprocessing and distributes the previous reconstructed frame to them. Similarly, BCU must wait for the processors to transfer back the motion vectors for subsequent processing. Obviously, such data dependency result in idle time for both BCU and processors, wasting the available CPU time. Strategy II is proposed to circumvent this problem.

In this strategy, the entire BMME is split into two steps, i.e., the FPME which is done on the previous original frame and the HPME which is conducted on the previous reconstructed frame. Fig. 9 shows the timing diagram of the video encoder using Strategy II when a fast BMME algorithm is used, where three processors are used. With this strategy, the data dependency between the part of preprocessing phase to obtain the *previous reconstructed frame* and the FPME that is done on the *previous original frame* is completely eliminated. After the previous original frame is partitioned and distributed to the processors, the preprocessing by BCU to obtain the previous reconstructed frame and the FPME by processors to obtain the full-pel motion vectors may be carried out concurrently. As such, not only the data parallelism inherent in the BMME process but also the parallelism between the preprocessing phase and FPME phase are exploited. It can be predicted that Strategy II will gain some advantages over Strategy I. It should be however carefully noted that the dependency between the preprocessing phase and the HPME that is performed on the previous *reconstructed* frame still exists. The time periods from t_1 to t_7 in Fig. 9 are defined as follows:

t_1	time for BCU to load the current original frame and partition the previous and current original frames;
t_2	time for BCU to distribute the previous and current original frames;
t_3	time for BCU to carry out the algorithms including DWT, SPIHT, ISPIHT, IDWT, MC, and partitioning of the previous reconstructed frame;
t_4	time for BCU to distribute the previous reconstructed frame;
t_5	time for BCU to encode the motion vectors using VLC;
t_6	time for processors to carry out the FPME;
t_7	time for processors to carry out the HPME.

1) Analysis of Strategy II When a Fast BMME Algorithm is Used: As shown by Fig. 9, when a fast BMME algorithm is used, the overall processing time of the video encoder is given by $(t_1+t_2+t_3+t_7)$. Again, since the time periods t_1, t_2, and t_3 for preprocessing phases are more or less constant, we attempt to minimize t_7 for HPME by adjusting amount of the data assigned to the individual processors.

The load partitioning process is illustrated by Fig. 10 and the directed flow graph of the load distribution process is shown in Fig. 11. The weight of the communication node i is given by (α_i+2) instead of $(2\alpha_i+2)$ as in Fig. 8. The reason is that we do not need to transfer the current original frame since it has already been partitioned and assigned to the processors in FPME. The weights of computation and solution-transfer-back nodes are no different from those in Fig. 8.

From Fig. 11, by equating the CFTP of processor p_{i+1} to the TFTP of processor p_i, we obtain

$$\begin{aligned}&\alpha_iE_i+\theta_{cp}+\lambda_i\alpha_iC+\theta_{cm}\\&\quad=(\alpha_{i+1}+2)C+\theta_{cm}+\alpha_{i+1}E_{i+1}+\theta_{cp}\\&\quad i=1,2\cdots m-2 \quad (9)\end{aligned}$$

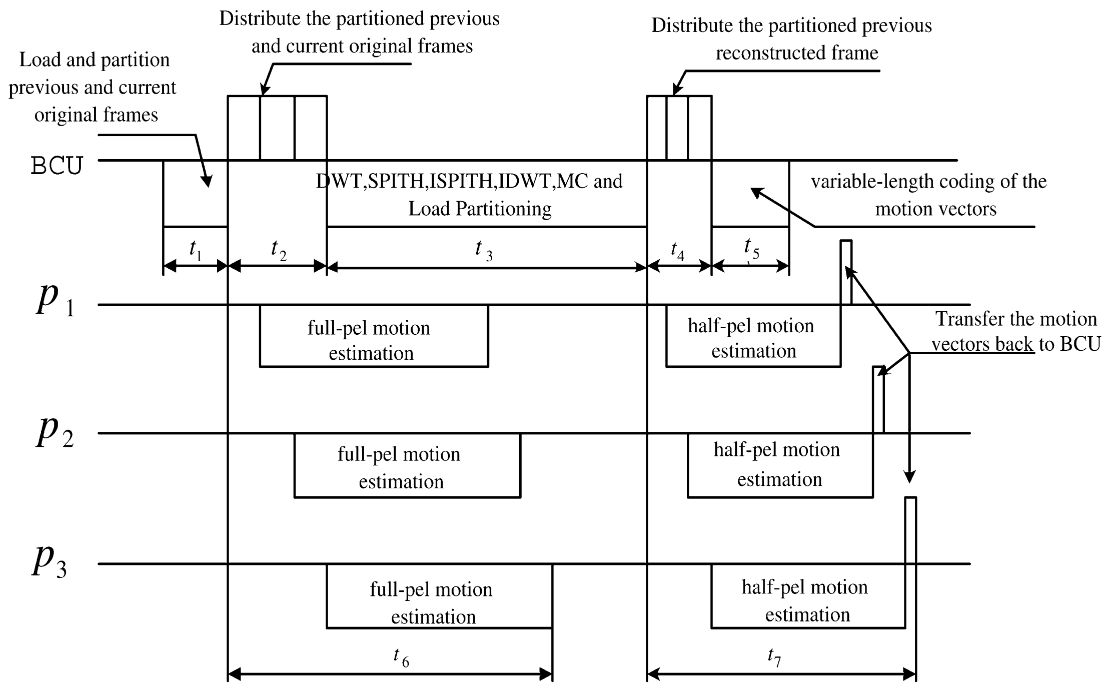

Load and partition previous and current original frames
Distribute the partitioned previous and current original frames
Distribute the partitioned previous reconstructed frame
BCU
DWT,SPITH,ISPITH,IDWT,MC and Load Partitioning
variable-length coding of the motion vectors
t_1
t_2
t_3
t_4
t_5
p_1
full-pel motion estimation
half-pel motion estimation
Transfer the motion vectors back to BCU
p_2
full-pel motion estimation
half-pel motion estimation
p_3
full-pel motion estimation
half-pel motion estimation
t_6
t_7

Fig. 9. Timing diagram of video encoder when a fast BMME algorithm is used for Strategy II.

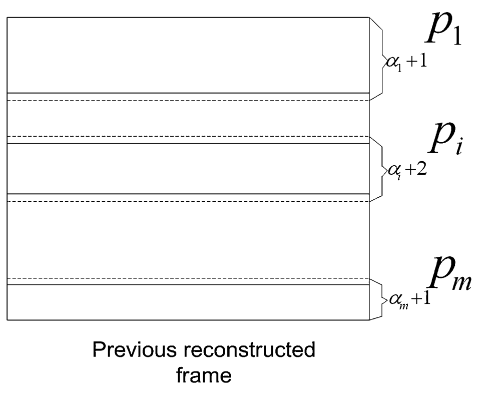


Fig. 10. Load partitioning process when a fast BMME algorithm is used for Strategy II.

$$\begin{aligned}&\alpha_{m-1}E_{m-1}+\theta_{cp}+\lambda_{m-1}\alpha_{m-1}C+\theta_{cm}\\&\quad=(\alpha_m+1)C+\theta_{cm}+\alpha_mE_m+\theta_{cp}.\end{aligned}\tag{10}$$

Similar to the Strategy I, the load partitions can be obtained by solving these recursive equations.

2) Analysis of Strategy II When a High-Complexity BMME Algorithm is Used: The above analysis is based on two assumptions, they are, $(t_2+t_3>t_6)$ and $(t_4+t_5<t_7)$, under which the overall processing time of the video encoder is given by $(t_1+t_2+t_3+t_7)$ where t_1, t_2 and t_3 for precessing phases are more-or-less constant and t_7 for HPME varies with the amount of data assigned to the processors.

Both the above two assumptions are true if a fast BMME is used. However, if a very computation-intensive motion estimation algorithm such as the full-search block matching algorithm is used, the assumption $(t_2+t_3>t_6)$ may be violated and the overall processing time becomes $(t_1+t_6+t_7)$. In that case, minimizing t_6 is a better solution to minimize the overall processing time of the video encoder since, generally, t_6 for FPME is much greater than t_7 for HPME. Therefore, to achieve a better performance, we should reschedule the data among processors when a complex BMME algorithm is to be used.

Fig. 12 shows the timing diagram of the video encoder when a very time-consuming BMME algorithm is used. The load partitioning process is illustrated by Fig. 13, in which both the previous and current *original* frames are partitioned. Fig. 14 shows the directed flow graph of the load distribution process for this case. The weights of the communication, computation nodes are the same as in Strategy I. The notable difference is that no solution-transfer-back node is in Fig. 14. The reason is that the full-pel motion vectors obtained in the processors in FPME are needed for the HPME by the same processor and thus need not to be transferred back to BCU upon the completion of the FPME. In this case, the optimal solution can be obtained if all the processors involved in the parallel processing will stop their computing at the same time instant.

From the directed flow graph, by equating the CFTP of processor p_{i+1} to the CFTP of processor p_i, we obtain

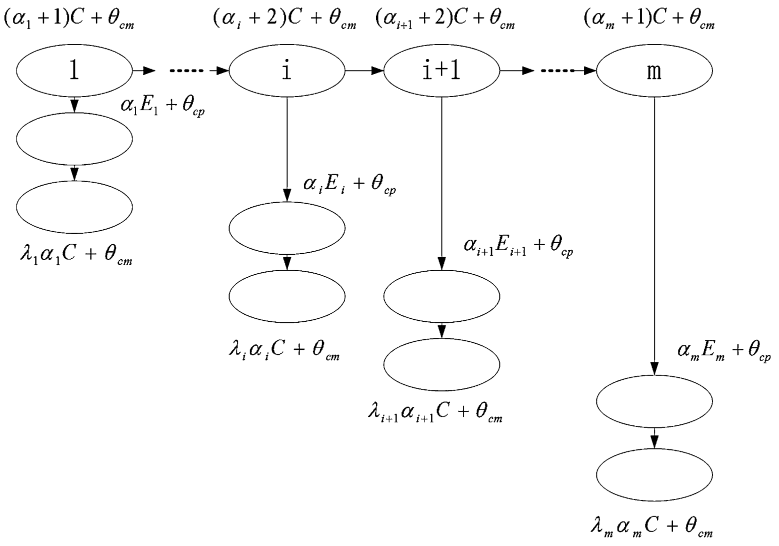
$$\begin{aligned}\alpha_iE_i+\theta_{cp}&=(2\alpha_{i+1}+2)C+\theta_{cm}+\alpha_{i+1}E_{i+1}+\theta_{cp}\\i&=1,2\ldots m-2\end{aligned}\tag{11}$$

$$\alpha_{m-1}E_{m-1}+\theta_{cp}=(2\alpha_m+1)C+\theta_{cm}+\alpha_mE_m+\theta_{cp}.\tag{12}$$

Thus, together with the normalization equation, the load partitions can be obtained by solving m recursive equations.

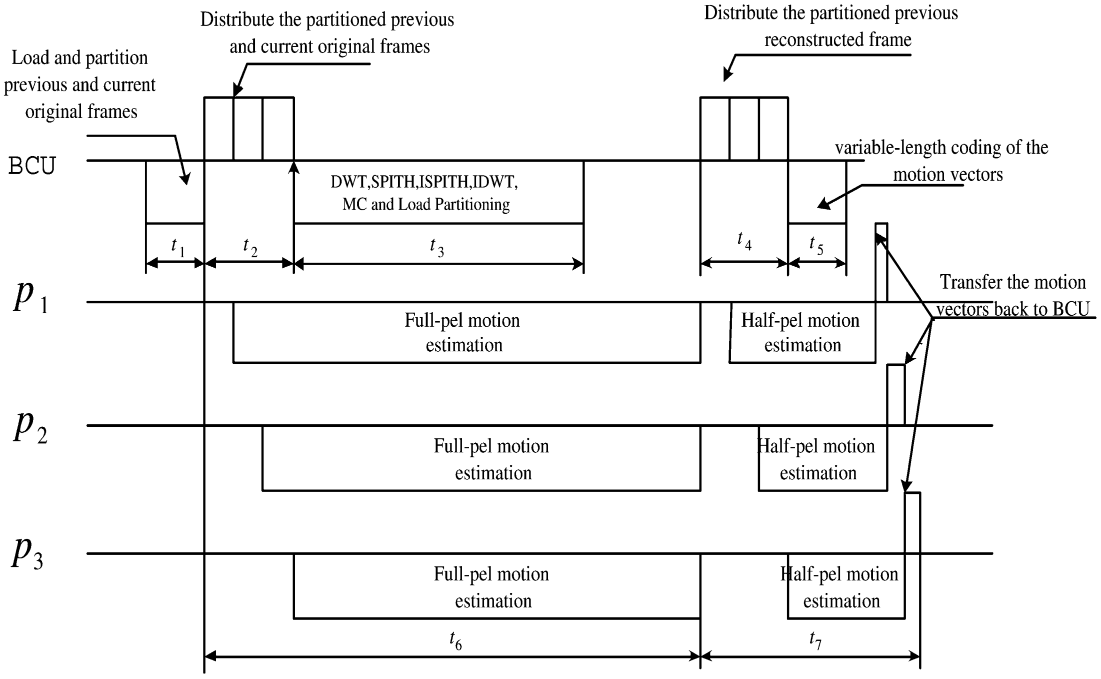
V. Performance Evaluation

In this section, we simulate the parallel video encoder on a PII350 MHz/128RAM computer using the proposed strategies. Obviously, we have homogeneous machines in our implementations, i.e., $E_i=E\,\forall\,i=1,\ldots,m$. All the program modules are developed under Microsoft Visual C++ version 6, in which each



$(\alpha_1+1)C+\theta_{cm}$
$(\alpha_i+2)C+\theta_{cm}$
$(\alpha_{i+1}+2)C+\theta_{cm}$
$(\alpha_m+1)C+\theta_{cm}$
1
i
i+1
m
$\alpha_1 E_1+\theta_{cp}$
$\alpha_i E_i+\theta_{cp}$
$\alpha_{i+1}E_{i+1}+\theta_{cp}$
$\alpha_m E_m+\theta_{cp}$
$\lambda_1\alpha_1 C+\theta_{cm}$
$\lambda_i\alpha_i C+\theta_{cm}$
$\lambda_{i+1}\alpha_{i+1}C+\theta_{cm}$
$\lambda_m\alpha_m C+\theta_{cm}$

Fig. 11. Directed flow graph of the load distribution process when a fast BMME algorithm is used for Strategy II.



Distribute the partitioned previous and current original frames
Distribute the partitioned previous reconstructed frame
Load and partition previous and current original frames
BCU
DWT,SPITH,ISPITH,IDWT, MC and Load Partitioning
variable-length coding of the motion vectors
t_1
t_2
t_3
t_4
t_5
p_1
p_2
p_3
Full-pel motion estimation
Half-pel motion estimation
Transfer the motion vectors back to BCU
t_6
t_7

Fig. 12. Timing diagram of the video encoder when a high-complexity BMME algorithm is used for Strategy II.

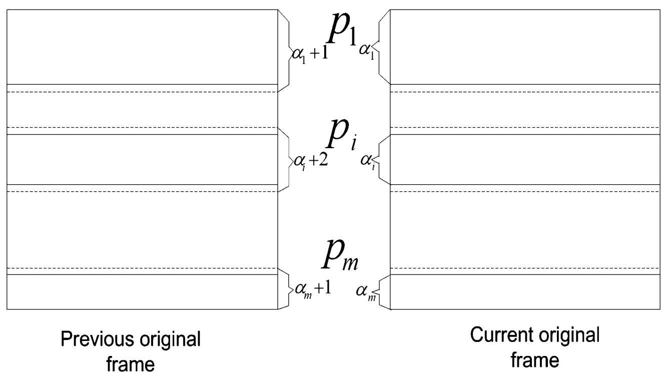


Fig. 13. Load partitioning process when a high-complexity BMME algorithm is used for Strategy II.

processor is represented by one corresponding process. The data communication between the processes is achieved by using the shared memory techniques. The output bit streams are written to the disk and could be decoded and displayed for visual inspection and comparison.

To demonstrate the advantages of the proposed strategies, it would be best if an actual implementation of the strategies on a real parallel video encoding system could be carried out. However, since our main objective is focused on illustrating how the DLT could be applied to parallelize the video encoding process, how the data parallelism inherent in a video encoder can be fully exploited and how the execution time and the associated overheads caused in the parallel processing process can be analytically quantified and minimized, we decided to go for a realistic simulation experiments with real-life values of the components to track the behavior of the strategies. We feel that it is beyond the scope of the current paper to carry out the actual implementation. The strategies proposed in this paper give us a theoretical

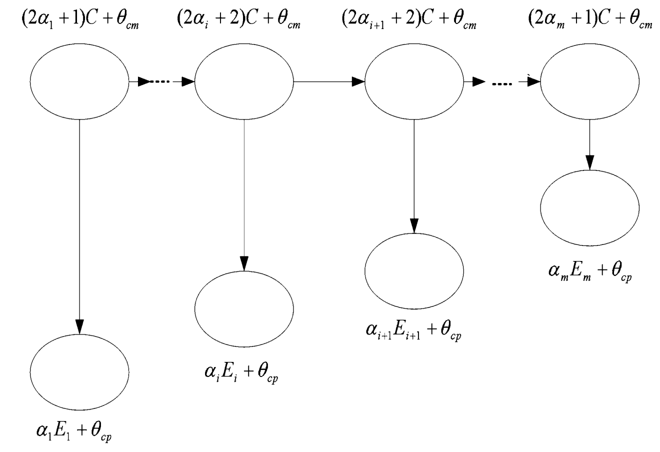

$(2\alpha_1+1)C+\theta_{cm}$
$(2\alpha_i+2)C+\theta_{cm}$
$(2\alpha_{i+1}+2)C+\theta_{cm}$
$(2\alpha_m+1)C+\theta_{cm}$
$\alpha_m E_m+\theta_{cp}$
$\alpha_{i+1}E_{i+1}+\theta_{cp}$
$\alpha_i E_i+\theta_{cp}$
$\alpha_1 E_1+\theta_{cp}$

Fig. 14. Directed flow graph of load distribution process for FBMA-II.

insight on how to quantify and minimize the overall processing time of a parallel system.

In our implementation, the *tempete.cif* (288×352 pixels) video sequence is used. we define six continuous frames a group of pictures (GOP). The first frame of a GOP is intracoded as I-frame and all five succeeding frames are intercoded from the preceding frame as P-frames. Each P-frame is encoded at 32 kbits and each I-frame is encoded at 96 kbits. BMME is performed on the 16×16 luminance blocks. The sum of absolute difference (SAD) is used as the distortion measure. The full search range is up to 15 pixels in all four directions from the center of the block. The motion vectors for the luminance blocks are divided by two, rounded to the nearest integers, and then taken as the motion vectors for the chrominance blocks.

A. Combinations of the Schemes Implemented in This Paper

The computational complexity of the BMME algorithms varies largely from algorithm to algorithm. To examine the impact of the complexity of the motion estimation algorithm on the performance of the video encoder, two representative BMME algorithms are implemented using each strategy. They are: 1) full-search block matching algorithm (FBMA) [16], the most computation-intensive BMME algorithm and 2) new three step search (NTSS) [10], a fast BMME algorithm. Thus, totally four combinations are implemented in this paper, which are listed as follows.

1) Strategy I with NTSS, referred to as NTSS-I.
2) Strategy I with FBMA, referred to as FBMA-I.
3) Strategy II with NTSS, referred to as NTSS -II.
4) Strategy II with FBMA, referred to as FBMA-II.

B. Determination of the Parameters

To solve the m recursive equations, we must provide the parameters E_i, C, θ_{cp}, θ_{cm}, m, L, and λ_i. And, since the entire video encoding process involves several processing phases such as HPME, FPME, preprocessing, VLC of the motion vectors, etc., we need to capture the execution time of each of these phases. Thus, before implementing the four combinations, those parameters that are required for either solving the recursive equations or calculating the processing time of each of the processing phases should be first quantified.

Note in our implementation, the video sequence is of CIF format (288×352 pixels) and each video frame is divided into 18×22 number of 16×16 blocks. As explained in Section III, we consider the size of the granularity (data unit), the minimum size of the data that can be assigned to the processor, as one row of blocks, i.e., 22 blocks. Thus, L equals to 18, which is the total number of rows of blocks in a video frame (288/16). Because each 16×16 luminance block will generate one 3-byte motion vector, λ_i equals to $3/(256 \times 4)$. That is, the solutions obtained in the processor p_i that need to be transferred back to BCU only amount for $(\lambda_i \times 100)$ percent of the data α_i assigned to it. The luminance component of a pixel is stored in one 4-byte float type variable and one motion vector is stored in a 3-byte structure in our C++ code.

The computation overhead θ_{cp} and communication overhead θ_{cm} are difficult to track in real systems. In our simulation on a single-processor environment, due to following reasons, we set these two overheads to zero, i.e., $\theta_{cp} = \theta_{cm} = 0$. Communication and computation overheads mainly comprise of the extra time used for system switching, hardware initialization and software startup. The amount of this extra time remains independent of the data size and is more or less constant per any computation or communication task [14], [21]. For some parallel systems where the latency caused by the overheads is comparable with the computation latency or the communication latency, the overheads may become a critical issue. In that case, the overheads may consume much processing time and dominate the parallel processing and must be carefully considered. For the problem studied in this paper, where one row of the MBs is treated as a basic unit and a 5-PII 350-MHz/128-MB processor bus network is used as the building platform, the time for communication and computation overheads are negligible when compared to the actual computation and communication time. That is, the processing time for the processor to process a basic unit and the processing time for BCU to transmit a data unit over the bus network will be much larger than the communication and computation overheads. Thus, setting the two overheads to zero does not influence the load partitioning/balancing process and the overall processing time. Furthermore, as will be discussed later, since the extra time caused by overheads is implicitly considered as a part of speed parameters when they are measured, above assumption of the two overheads being zero remains valid.

The speed parameter E is dependent on the processor, algorithm, and the size of the granularity. Given the processor, algorithms and the size of the granularity in our experiments, the measured values of E will be given below. The speed parameters that are required for our implementations are defined as follows.

Speed parameters for Strategy I:

E_{ME}	time for processor to carry out the BMME with one data unit;
E_{mainI}	time for BCU to carry out the preprocessing with one data unit;

Speed parameters for Strategy II:

E_{mainII}	time for BCU to carry out the algorithms including DWT, SPIHT, IDWT, ISPIHT, MC, and partitioning of previous reconstructed frame with one data unit;

E_{MV}	time for BCU to encode the motion vectors with one data unit;
E_{LP}	time for BCU to partition the previous original frame and current original frame with one data unit;
E_{full}	time for processor to carry out FPME with one data unit;
E_{half}	time for processor to carry out HPME with one data unit.

Now we shall describe the methodology adopted to measure the speed parameter E. In our program, the function *clock( )*, which is defined in Microsoft Visual C++ version 6 run-time library, is used to measure the processing time. The return value of *clock( )* is the elapsed wall-clock time since the start of the process (elapsed time in seconds times CLOCKS_PER_SEC, where CLOCKS_PER_SEC is the operating frequency of the processor), which tells how much time the calling process has used. It takes into account whatever time the OS was used including the extra time for communication and computation overheads. Partly due to this reason, it is reasonable to set the computation and communication overheads to zero in the simulations. We encode the first 50 frames of *tempete*. The processing times for each of the frames are measured and then averaged. The average values are treated as the final results. Note the processing times for the I-frames are excluded from the computation of the final results.

In practice, the communication speed will be faster than computation speed in a small bus network. In our experiments, with $m = 5$, the time to transfer the partitioned data to the processors and the time to collect the motion vectors are calculated using an observed value of C as 5 ms/data unit. The determination of C is based on the experiment conducted in [20], in which an image processing application for large images is implemented on a real 10/100 Mb-Tx high-speed Ethernet network. To measure the communication speed, the entire image is transmitted over the network repeatedly for certain number of times. The average value is then computed to represent the communication speed C. In reality, the communication speed of a bus network varies over time. However, since the communication delays or the channel bandwidth of a small and dedicated bus network remains more-or-less constant in a practical system [14], [21], it is appropriate to represent the communication speed using a single average value, as done in our simulation.

The measured speed parameters for Strategy I are given as follows. E_{mainI} equals to 115 ms/data unit, E_{ME} for NTSS is 64 ms/data unit and for FBMA is 1538 ms/data unit. The measured speed parameters for Strategy II are tabulated in Table I, in which all the parameters are measured in ms/data unit.

As above, in our experiments, speed parameters are represented by the average values of the measured processing times of 50 continuous frames under the assumption that the speed parameters remains constant during the entire parallel processing. In reality, the speed parameters like E_{mainII}, E_{ME}, etc., vary from frame to frame and slice to slice. Table II lists the measured processing times for BCU and five processors to process their computation load for each of the 50 frames encoded in experiment NTSS-I. In our encoder, each intracoded I-frame is followed by five intercoded P-frames. The rows in the table where processing times of the five processors equal to zero denote the intracoded I-frames. When computing the speed parameters, the processing times for I-frames are excluded.

TABLE I
SPEED PARAMETERS FOR STRATEGY II

E_{LP}	E_{mainII}	E_{MV}	E_{full}		E_{half}
			NTSS	FBMA	
10	115	0.14	40	1581	27

TABLE II
MEASURED PROCESSING TIMES FOR BCU AND 5 PROCESSORS TO PROCESS THEIR COMPUTATION LOAD FOR EACH OF THE 50 FRAMES (EXCLUDING THE FIRST I FRAME) ENCODED IN NTSS-I; THE NUMBER OF DATA UNITS ASSIGNED TO 5 PROCESSORS ARE 6, 4, 3, 3, 2, RESPECTIVELY

Frame	Type	BCU (ms)	p_1 (ms)	p_2 (ms)	p_3 (ms)	p_4 (ms)	p_5 (ms)
1	P	2140	390	220	160	220	110
2	P	2040	380	220	160	220	110
3	P	1980	390	270	220	160	170
4	P	1920	390	270	170	160	170
5	P	1920	380	280	160	220	110
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
45	P	2260	380	220	220	160	170
46	P	1980	440	270	170	220	110
47	P	2030	380	280	160	220	110
48	I	1980	0	0	0	0	0
49	P	2310	430	280	160	220	110
50	P	2530	380	280	270	220	110

From Table II, we see the processing times for the processor to process the same amount of computation load do vary over time. However, the variation is small. For BCU, the ratio between standard deviation of the processing times and its average processing time is 0.07; for processor p_1, this value is also 0.07; for processor p_5, the value is 0.21. Since all the MBs use the same motion search strategy in our algorithm, the complexity and processing time demand of each MB remains close to each other. Thus, for the uniform system that is simulated in this paper, it will be appropriate to use the average speed parameter, which is measured in advance, to partition all the frames. Further, since the main objective of the paper is focused on designing theoretical or analytical methods to quantify and minimize the processing time and all associated overheads caused in the parallel processing process, accurate updating of the speed parameters according to past statistics is not very necessary.

C. Experimental Results

Up to now, we have clarified the methodologies and technical details that are required for parallel implementation of a video encoder. It is the time for us to put those ideas into realization. Below, we will describe in detail the implementations of the four combinations, referred to as *NTSS-I*, *FBMA-I*, *NTSS-II*, *FBMA-II*, respectively. The performance of the video encoders is evaluated using the metrics *speedup* and *performance gain*, which are defined as follows. Let us refer to the overall processing time of the conventional video encoder implemented on a single processor as t_{con}; and, the overall processing of the parallel video encoder implemented on a bus net-

TABLE III
THEORETICAL DME PROCESSING TIMES FOR NTSS-I

	p_1	p_2	p_3	p_4	p_5
Communication Time (ms)	65	115	155	195	220
Computation Time (ms)	384	256	192	192	128
Solution-transfer-back Time (ms)	0.09	0.06	0.04	0.04	0.02
Finish Time (ms)	449	371	347	387	348

TABLE IV
MEASURED DME PROCESSING TIMES FOR NTSS-I

	p_1	p_2	p_3	p_4	p_5
Communication Time (ms)	65	115	155	195	220
Computation Time (ms)	387	265	182	199	129
Solution-transfer-back Time (ms)	0.09	0.06	0.04	0.04	0.02
Finish Time (ms)	452	380	337	394	349

work as t_{par}. The *speedup* is defined as the ratio of t_{con} over t_{par}, i.e., speedup $= t_{\text{con}}/t_{\text{par}}$. The *performance gain* is defined as the ratio of $(t_{\text{con}} - t_{\text{par}})$ over t_{con}, i.e., performance gain $= (t_{\text{con}} - t_{\text{par}})/t_{\text{con}}$.

As follows, we present the experimental studies conducted in this paper to illustrate the parallel video encoding process using proposed strategies. The performance of the parallel video encoder is evaluated using the *speedup* and *performance gain* based on the experimental results.

1) NTSS-I: Substituting the processing speed E_{ME} for NTSS and C, m, L, θ_{cm}, and θ_{cp} that are given before into (7) and (8) and the normalization equation, by solving which we obtain $(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5) = (5.02, 4.20, 3.49, 2.87, 2.41)$. After integer approximation, we have $(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5) = (6, 4, 3, 3, 2)$.

Now, since we have the parameters E_i, C, θ_{cp}, θ_{cm}, and the load partitions α_i, the processing time for distributed motion estimation (DME) by each processor can be calculated using the definitions of $T_i^{\text{CFTP}}(\alpha)$, $T_i^{\text{TFTP}}(\alpha)$, and $T_i(\alpha)$, as defined in Section III. The the preprocessing time t_1 can be computed by multiplying the processing speed E_{mainI} and the number of data units processed by BCU, i.e., L.

Table III shows the calculated (theoretical) processing times for each processor to process its DME data. Note that, after the integer load approximation, the processors may not stop computing exactly one after another. We select the maximum finish time of the processor as the processing time for the entire DME process. From Table III, we see processor p_1 takes the longest time and thus, t_2 equals to 449 ms. t_1 is the product of E_{mainI} and L, i.e., $t_1 = E_{\text{mainI}} \times L = 115 \times 18 = 2070$ ms. Thus, the *theoretical overall processing time* of the video encoder using DME is given by $(t_1 + t_2 = 2519$ ms$)$. Without DME, the overall processing time is given by $(E_{\text{mainI}} + E_{\text{ME}}) \times L = (115+64)\times 18 = 3222$ ms. Thus, a speedup of 1.28 and a 21.8% performance gain on the overall processing time is achieved by scheduling the BMME data on a bus network.

Table IV shows the measured DME processing times obtained in our experimental study, where we see the measured overall processing time t_2 for DME process is 452 ms. The measured processing time t_1 for preprocessing phase is 2067 ms. Thus, the *actual overall processing time* can be computed as $(t_1 + t_2 = 2067 + 452) = 2519$ ms, which happens to be equal to the *theoretical overall processing time*. Table V summarizes

TABLE V
COMPARISON BETWEEN THEORETICAL AND MEASURED RESULTS FOR NTSS-I

	t_1	t_2	t_{par}	t_{con}	*speedup*
Theoretical Results (ms)	2070	449	2519	3222	1.28
Measured Results (ms)	2067	452	2519	/	/

TABLE VI
THEORETICAL DME PROCESSING TIMES FOR FBMA-I

	p_1	p_2	p_3	p_4	p_5
Communication Time (ms)	45	95	135	185	220
Computation Time (ms)	6152	6152	4614	6152	4614
Solution Transfer back Time (ms)	0.06	0.06	0.04	0.06	0.04
Finish Time (ms)	6197	6247	4749	6337	4834

TABLE VII
MEASURED DME COMPUTATION TIMES FOR FBMA-I

	p_1	p_2	p_3	p_4	p_5
Computation Time (ms)	6138	6136	4614	6173	4625

TABLE VIII
COMPARISON BETWEEN THEORETICAL AND MEASURED RESULTS FOR FBMA-I

	t_1	t_2	t_{par}	t_{con}	*speedup*
Theoretical Results (ms)	2070	6337	8407	29754	3.54
Measured Results (ms)	2067	6358	8421	/	/

TABLE IX
THEORETICAL PROCESSING TIMES OF HPME FOR NTSS-II

	p_1	p_2	p_3	p_4	p_5
Communication Time (ms)	35	70	95	115	130
Computation Time (ms)	162	135	81	54	54
Solution Transfer back Time (ms)	0.09	0.07	0.04	0.03	0.03
Finish Time (ms)	197	205	176	169	184

TABLE X
THEORETICAL PROCESSING TIMES OF FPME FOR NTSS-II

	p_1	p_2	p_3	p_4	p_5
Communication Time (ms)	65	125	165	195	220
Computation Time (ms)	240	200	120	80	80
Finish Time (ms)	305	325	285	275	300

TABLE XI
MEASURED COMPUTATION TIMES OF HPME FOR NTSS-II

	p_1	p_2	p_3	p_4	p_5
Computation Time (ms)	155	141	84	60	55

TABLE XII
MEASURED COMPUTATION TIMES OF FPME FOR NTSS-II

Computation Time (ms)	233	197	116	85	83

TABLE XIII
COMPARISON BETWEEN THEORETICAL AND MEASURED RESULTS FOR NTSS-II

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_{par}	t_{con}	*speedup*
Theoretical Results (ms)	180	220	2070	130	2.52	325	205	2675	3279	1.23
Measured Results (ms)	183	220	2130	130	2.5	322	211	2744	/	/

the above comparison between theoretical and measured results for NTSS-I. As explained in Section V-B, in our simulation, the communication times are computed using an observed communication speed. Thus, the measured communication times in Table IV are exactly the same as the theoretical communication times in Table III.

2) FBMA-I: Similar to *NTSS-I*, using the processing speed E_{ME} for FBMA as given before, we obtain $(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5) =$ (3.6591, 3.6290, 3.5992, 3.5695, 3.5432). After integer load approximation, we have $(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5) = (4, 4, 3, 4, 3)$.

The theoretical DME processing times are shown in Table VI, where we see t_2 equals to 6337 ms. The theoretical preprocessing time is the same as that in *NTSS-I*, i.e., $t_1 = 2070$ ms. Thus, the *theoretical overall processing time* is given by $(t_1 + t_2) = 8407$ ms. Without DME, the theoretical overall processing time of the video encoder is given by $(E_{\mathrm{main}} + E_{\mathrm{MV}}) \times L = (115 + 1538) \times 18 = 29\,754$ ms. Thus, a speedup of 3.54, a 71.7% performance gain on the overall processing time of the video encoder, is achieved. Table VII shows measured DME computation times obtained in this experiment (the DME communication times are computed using the observed communication speed and are the same as those shown in Table VI). The measured preprocessing time t_1 equals to 2067 ms. Similar to NTSS-I, the *actual overall processing time* can be computed as 8421 ms. Again, the experimental results show a close match with the theoretical results. Table VIII summarize the above comparison.

3) NTSS-II: Substituting processing speed E_{half} as given in Table I and C, θ_{cm}, θ_{cp}, m, L into (9), (10) and the normalization equation, by solving which we obtain $(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5) =$ $(5.5963, 4.4119, 3.4121, 2.5680, 2.0117)$. After integer load approximation, we have $(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5) = (6, 5, 3, 2, 2)$.

The theoretical processing times of HPME and FPME under the above load partitions are given in Tables IX and X, respectively, from where we can see t_7 equals to 205 ms and t_6 equals to 325 ms.

The respective processing times from t_1 to t_5 indicated in Fig. 9 are calculated as follows:

$t_1 = E_{\mathrm{LP}} \times L = 10 \times 18 = 180$ ms;
$t_2 = C \times (2L + 2m - 2) = 5 \times (2 \times 18 + 2 \times 5 - 2) = 220$ ms;
$t_3 = E_{\mathrm{mainII}} \times L = 115 \times 18 = 2070$ ms;
$t_4 = C \times (L + 2m - 2) = 5 \times (18 + 2 \times 5 - 2) = 130$ ms;
$t_5 = E_{\mathrm{MV}} \times L = 0.14 \times 18 = 2.52$ ms.

Thus, the *theoretical overall processing time* of video encoder using DME is given by $(t_1 + t_2 + t_3 + t_7) = 2675$ ms. Without DME, the theoretical overall processing time equals to $(E_{\mathrm{mainII}} + E_{\mathrm{full}} + E_{\mathrm{half}} + E_{\mathrm{MV}}) \times L =$ $(115 + 40 + 27 + 0.14) \times 18 = 3279$ ms. Thus, a speedup of 1.23 and a 18.4% performance gain on the overall processing time of the video encoder is achieved. Tables XI and XII show the measured computation times of HPME and FPME processes obtained in the experiment. The measured values of t_1, t_3 and t_5 are 183 ms, 2130 ms and 2.5 ms, respectively. Similar to NTSS-I, the *actual overall processing time* can now be computed as $(t_1 + t_2 + t_3 + t_7) = 2744$ ms. Again, the experimental results demonstrate a good match with the theoretical results. Table XIII summarizes the above results.

TABLE XIV
THEORETICAL PROCESSING TIMES OF FPME FOR FBMA-II

	p_1	p_2	p_3	p_4	p_5
Communication Time (ms)	45	95	135	185	220
Computation Time (ms)	6324	6324	4743	6324	4743
Finish Time (ms)	6369	6419	4878	6509	4963

TABLE XV
THEORETICAL PROCESSING TIMES OF HPME FOR FBMA-II

	p_1	p_2	p_3	p_4	p_5
Communication Time (ms)	25	55	80	110	130
Computation Time (ms)	108	108	81	108	81
Transfer-back Time	0.06	0.06	0.04	0.06	0.04
Finish Time (ms)	133	163	161	218	211

TABLE XVI
MEASURED COMPUTATION TIMES OF FPME FOR FBMA-II

	p_1	p_2	p_3	p_4	p_5
Computation Time (ms)	6149	6079	4985	6070	5001

TABLE XVII
MEASURED COMPUTATION TIMES OF HPME FOR FBMA-II

	p_1	p_2	p_3	p_4	p_5
Computation Time (ms)	97	107	79	105	77

4) FBMA-II: Substituting processing speed E_{full} for FBMA as given in Table I and C, θ_{cm}, θ_{cp}, m, L into (11) and (12) and the normalization equation, by solving which we obtain $(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5) =$ $(3.6576, 3.6283, 3.5992, 3.5703, 3.5447)$. After integer load approximation, we have $(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5) = (4, 4, 3, 4, 3)$.

The theoretical processing times of FPME and HPME under the above load partitions are shown in Tables XIV and XV respectively, where we see t_6 equals to 6509 ms and t_7 equals to 218 ms. Theoretical processing times from t_1 to t_5 for preprocessing phase are calculated the same as in *NTSS-II*. Thus, the *theoretical overall processing time* of the video encoder using DME is given by $(t_1 + t_6 + t_7) = 6907$ ms. Without DME, the theoretical overall processing time is $(E_{\mathrm{main}} + E_{\mathrm{full}} + E_{\mathrm{half}} + E_{\mathrm{MV}}) \times L = (115 + 1581 + 27 + 0.14) \times 18 = 31,016$ ms. Thus, a speedup of 4.49 and a performance gain of 77.8% on the overall processing time is achieved.

TABLE XVIII
COMPARISON BETWEEN THEORETICAL AND MEASURED RESULTS FOR FBMA-II

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_{par}	t_{con}	*speedup*
Theoretical Results (ms)	180	220	2070	130	2.52	6509	218	6907	31016	4.49
Measured Results (ms)	186	220	2019	130	4.0	6255	215	6656	/	/

Tables XVI and XVII show the measured computation times of the FPME and HPME processes. The measured values of t_1, t_3 and t_5 are 186, 2019, and 4 ms, respectively. Similar to NTSS-I, the *actual overall processing time* can be computed as $(t_1 + t_6 + t_7) = 6650$ ms. Again, we observe that the experimental results demonstrate a good match with the theoretical findings. Table XVIII summarizes the above results.

D. Discussions on the Results

Table XIX summarizes all the results obtained in our experiments. From the results obtained in four tests, we can see using the proposed load partitioning scheme and two implementation strategies, a significant performance improvement on the overall processing time of the video encoder is obtained. With only five processors used in our parallel implementation, the lowest performance gain on the processing time is 18.4% for Strategy I with NTSS and the highest is 77.8% for Strategy II with FBMA. The results reported in this paper provide considerable hope to implement these strategies for handling real time encoding, using divisible load paradigm.

Obviously, by increasing the number of the processors in FBMA-II, t_6 for FPME can be further reduced and thus contribute to the overall processing time $(t_1 + t_6 + t_7)$ of the video encoder. However, note that $(t_2 + t_3)$ is the lower bound to which t_6 could be decreased because any further reduction in t_6 below $(t_2 + t_3)$ will not contribute to the overall processing time $(t_1 + t_2 + t_3 + t_7)$. As stated in [2], [14], and [22], given a m-processor system, it may not be necessary that an optimal solution exist when one attempts to utilize all the m processors. In the case of *Strategy II*, a more restrictive requirement on the maximum number of processors is posed. For the fast BMME algorithm, the optimal solution exists when t_7 for HPME is minimized.[2] However, for the high-complexity BMME algorithm $(t_6 > t_2 + t_3)$, the optimal number of processors should guarantee that t_6 approximates to $(t_2 + t_3)$ so that the idle time for both BCU and processors is minimized.

As we know, we use a PII350 Hz/128 MB computer to do the simulation. In terms of the up-to-date computer technology, this processor is certainly slow. However, since our main objective is not to achieve a real-time video encoder but to design a theoretical or analytical method to quantify the execution time and associated overheads in any parallel video encoding system, the processor speed is not our serious consideration. Our concern is more on how to effectively partition and balance the computation load among the processors for any given parallel system no matter what are the communication and computation speeds. Thus, as long as there are certain communication latency and computation latency in the system, our algorithm can be applied to minimize the parallel processing.

[2]In this case, refer to [14] for details on how to determine the optimal number of processors

TABLE XIX
PERFORMANCE GAIN AND SPEEDUP ACHIEVED BY OUR SCHEMES

	performance gain	*speedup*
Strategy I with NTSS	21.8%	1.28
Strategy I with FBMA	71.7%	3.54
Strategy II with NTSS	18.4%	1.23
Strategy II with FBMA	77.8%	4.49

When the computation latency is significantly greater than the communication latency, the load partitions by our scheme approximates to those by conventional equal-partitioning scheme. As we see from FBMA-I and FBMA-II, where the exhaustive full-search motion estimation algorithm is used, all the processors are assigned almost the same amount of data. However, from optimization point of view, our scheme certainly has the advantage over the simple partitioning scheme since it gives us a theoretical insight on how to quantify and minimize the processing time and associated overheads of any parallel system no matter what are the communication and computation delays. Furthermore, as we observed from our experiments, the DLT computation overhead is negligible especially when the number of processors involved is small.

VI. CONCLUSION

In this paper, we parallelize a video encoder on a bus network. Using DLT paradigm, a strip-wise load partitioning/balancing scheme, a load distribution strategy, and two implementation strategies are developed to exploit the available data parallelism inherent in the video encoding process.

The advantages of our strategies include the following. 1) With our approaches, the precise modeling and minimization of the execution time of each phase of the video encoding process becomes an easy task. The proposed strategies give us a theoretical insight on how to analytically quantify and minimize the overall processing time of a parallel video coding system. 2) Our strategies are easy to implement. No interprocessor communication is required in our implementations because we assign the additional data in the initial communication phase. This largely simplifies the complexity of our strategies. 3) The proposed strategies can be easily extended and applied to improve other parallel applications. Although a homogeneous platform is used in this paper, a heterogenous platform where the processors' speeds are different from each other are allowed. Furthermore, our strategies also allow different algorithms being used in processors. The data partitions that minimize the processing time can be obtained by solving the same set of recursive equations as that used in this paper. The only difference is on the measurement of the speed parameters.

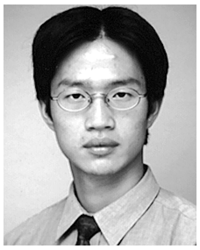
The performance of the strategies is rigorously tested in our experimental studies. Four combinations of the schemes referred to as NTSS-I, FBMA-I, NTSS-II, FBMA-II are implemented using each of the proposed strategies. The performance of our parallel video encoder is quantified using metrics *speedup* and *performance gain*. As presented in Section V, a significant performance gain on the overall processing time of the video encoder is obtained by using our strategies. The proposed strategies are shown to be effective in reducing the overall processing time of the video encoder and the DLT paradigm is shown to be an effective and viable solution to parallelizing the video encoder on a bus network.

ACKNOWLEDGMENT

The authors would like to thank Professor K. R. Rao, Department of Electrical Engineering, University of Texas at Arlington, for his suggestions in enriching Section I on video encoding schemes.

REFERENCES

[1] P. Kuhn, *Algorithms, Complexity Analysis and VLSI Architectures for MPEG-4 Motion Estimation*. Dordrecht, Germany: Kluwer, 1999.
[2] V. Bharadwaj and N. Viswanadham, "Suboptimal solutions using integer approximation techniques for scheduling divisible loads on distributed bus networks," *IEEE Trans. Syst., Man, Cybern. A, Syst. Humans*, vol. 30, no. 6, pp. 680–691, Nov. 2000.
[3] S. M. Akramullah, I. Ahmad, and M. L. Liu, "Parallelization of MPEG-2 video encoder for parallel and distributed computing systems," in *Proc. Midwest Symp. Circuits and Systems*, Aug. 1995, pp. 834–837.
[4] Y. He, I. Ahmad, and M. L. Liou, "A software-based MPEG-4 video encoder using parallel processing," *IEEE Trans. Circuit Syst. Video Technol.*, vol. 8, no. 7, pp. 909–920, Nov. 1998.
[5] N. H. C. Yung and K.-K. Leung, "Spatial and temporal data parallelization of the H.261 video coding algorithm," *IEEE Trans. Circuit Syst. Video Technol.*, vol. 11, no. 1, pp. 91–104, Jan. 2001.
[6] E. Chan and S. Panchanathan, "Review of block matching based motion estimation algorithms for video compression," in *Proc. Canadian Conf. Electrical and Computer Engineering*, vol. 1, Sep. 1993, pp. 151–154.
[7] S. A. Martucci, I. Sodagar, T. Chiang, and Y.-Q. Zhang, "A zerotree wavelet video coder," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 7, pp. 109–118, Feb. 1997.
[8] J. M. Shapiro, "Embedded image coding using zerotrees of wavelet coefficients," *IEEE Trans. Signal Process.*, vol. 41, no. 12, pp. 3445–3462, Dec. 1993.
[9] V. Bharadwaj, D. Ghose, V. Mani, and T. G. Robertazzi, *Scheduling Divisible Loads in Parallel and Distributed Systems*. Los Alamitos, CA: IEEE Computer Society Press, 1996.
[10] R. Li, B. Zeng, and M. L. Liou, "A new three step search algorithm for block motion estimation," *IEEE Trans. Circuit Syst. Video Technol.*, vol. 4, no. 4, pp. 438–442, Aug. 1994.
[11] *Generic Coding of Moving Pictures and Associated Audio*, Draft International Standard ISO/IEC 13 818, 1993.
[12] *MPEG-4 Video Verification Model*, ISO/IEC JTC1/SC29/WG11 N1796, 1997.
[13] *Video Codec for Audiovisual Services at $p \times 64$ kbits*, ITU-T Recommendation H.261, 1990.
[14] V. Bharadwaj, X. Li, and C. C. Ko, "On the influence of start-up costs in scheduling divisible loads on bus networks," *IEEE Trans. Parallel Distributed Syst.*, vol. 11, no. 12, pp. 1288–1305, Dec. 2000.
[15] A. Said and W. A. Pearlman, "A new, fast, and efficient image codec based on set partitioning in hierarchical tree," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 6, no. 3, pp. 243–250, Jun. 1996.
[16] P. Baglietto, M. Maresca, A. Migliaro, and M. Migliardi, "Parallel implementation of the full search block matching algorithm for motion estimation," in *Proc. Int. Conf. Application Specific Array Processors*, 1995, pp. 182–192.
[17] K. Ko and T. G. Robertazzi, "Record search time evaluation," presented at the Conf. Information Sciences and Systems, Princeton, NJ, Mar. 2000, Euro-Par 2000.
[18] S. K. Chan, V. Bharadwaj, and D. Ghose, "Matrix-vector products on distributed bus networks with communication delays using the divisible load paradigm: Performance analysis and simulation," *Math. Comput. Simul.*, vol. 58, pp. 71–79, 2001.
[19] M. Drozdowski and P. Wolniewicz, *Experiments With Scheduling Divisible Tasks in Clusters of Workstations*. New York: Springer-Verlag, 2000, vol. LNCS 1900, pp. 311–319.
[20] B. Veeravalli and S. Ranganath, "Theoretical and experimental study on large size image processing application using divisible load paradigm on distributed bus networks," *Image Vis. Comput.*, vol. 20, pp. 917–935, 2002.
[21] G. Barlas, "Collection-aware optimum sequencing of operations and closed-form solutions for the distribution of a divisible load on arbitrary processor trees," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 5, pp. 429–441, May 1998.
[22] M. Drozdowski, *Selected Problems of Scheduling Tasks in Multiprocessor Computer Systems*, ser. Monographs. Poznon, Poland: Poznan Univ. Technology Press, 1997.
[23] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr, and M. E. Zosel, *The High Performance Fortran Handbook*. Cambridge, MA: MIT Press, 1994.



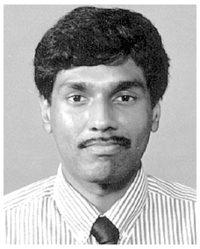
Ping Li received the B.Eng. and M.Eng. degrees in mechanical engineering from Xi'an Jiaotong University, Xi'an, China, in 1998 and 2000, respectively, and the M.Eng. degree in computer engineering from National University of Singapore in 2003.

He worked on H.264/AVC video coding at the Institute for Infocomm Research, Singapore, from March 2003 to August 2004. His research interests include video/image processing and multimedia computing. He is currently with the Design Technology Institute, Faculty of Electrical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands.

Bharadwaj Veeravalli (M'99) received the B.Sc. degree in physics from Madurai-Kamaraj University, Maduria, India, in 1987, the M.S. degree in electrical communication engineering and the Ph.D. degree from the Department of Aerospace Engineering, Indian Institute of Science, Bangalore, India, in 1991 and 1994, respectively.

He was a Postdoctoral Researcher in the Department of Computer Science, Concordia University, Montreal, Canada, in 1996. He is currently with the Department of Electrical and Computer Engineering at The National University of Singapore, Singapore, as an Assistant Professor. His research interests include high-speed heterogeneous computing, scheduling in parallel and distributed systems, multimedia computing, cluster/grid computing, bioinformatics, and manufacturing systems. He has served as a Technical Committee Member in several international conferences and had published more than 50 technical papers in archival journals and conferences. He is one of the earliest researchers in the field of divisible load theory and is the principal author of the book entitled *Scheduling Divisible Loads in Parallel and Distributed Systems* (Los Alamitos, CA: IEEE Computer Society, 1996).

He is currently serving as an Associate Editor for *International Journal of Computers and their Applications* (IJCA), and the IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS—A: SYSTEMS AND HUMANS.



Ashraf A. Kassim received the B.Eng. degree (first-class hons.) in electrical engineering from the National University of Singapore (NUS), Singapore, in 1985 and the Ph.D. degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, in 1993.

He worked on the design and development of machine vision systems at Texas Instruments until 1988. Since 1993, he has been with the Electrical and Computer Engineering Department, NUS, where he is currently an Associate Professor and Deputy Head of Department. His research interests include image analysis, machine vision, video/image processing and compression.