MITSUBISHI ELECTRIC RESEARCH LABORATORIES
http://www.merl.com

# Toward Realtime Side Information Decoding On Multi-Core Processors

Svetislav Momcilovic, Yige Wang, Shantanu Rane, Anthony Vetro

TR2010-100 December 2010

**Abstract**

Most distributed source coding schemes involve the application of a channel code to the signal and transmission of the resulting syndromes. For low complexity encoding with superior compression performance, graph-based channel codes such as LDPC codes are used to generate the syndromes. The encoder performs simple XOR operations, while the decoder uses belief propagation (BP) decoding to recover the signal of interest using the syndromes and some correlated side information. We consider parallelization of BP decoding on general-purpose multi core CPUs. The motivation is to make BP decoding fast enough for realtime applications. We consider three different BP decoding algorithms: Sum-Product BP, Min-Sum BP and Algorithm E. The speedup obtained by parallelizing these algorithms is examined along with the trade-off against decoding performance Parallelization is achieved by dividing the received syndrome vectors among different cores, and by using vector operations to simultaneously process multiple check nodes in each core. While Min-Sum BP has intermediate decoding complexity, a "vectorized" version of Min-Sum BP performs nearly as fast as the much simpler Algorithm E with significantly fewer decoding errors. Our experiments indicates that, for the best compromise between speed and performance, the decoder should use Min-Sum BP when the side information is of good quality and Sum-Product BP otherwise.

*Multimedia Signal Processing Workshop*

# Toward Realtime Side Information Decoding on Multi-core Processors

Svetislav Momcilovic†, Yige Wang, Shantanu Rane, Anthony Vetro

Mitsubishi Electric Research Laboratories (MERL), Cambridge, MA 02140, USA.

***Abstract*— Most distributed source coding schemes involve the application of a channel code to the signal and transmission of the resulting syndromes. For low-complexity encoding with superior compression performance, graph-based channel codes such as LDPC codes are used to generate the syndromes. The encoder performs simple XOR operations, while the decoder uses belief propagation (BP) decoding to recover the signal of interest using the syndromes and some correlated side information. We consider parallelization of BP decoding on general-purpose multi-core CPUs. The motivation is to make BP decoding fast enough for realtime applications. We consider three different BP decoding algorithms: Sum-Product BP, Min-Sum BP and Algorithm E. The speedup obtained by parallelizing these algorithms is examined along with the tradeoff against decoding performance. Parallelization is achieved by dividing the received syndrome vectors among different cores, and by using vector operations to simultaneously process multiple check nodes in each core. While Min-Sum BP has intermediate decoding complexity, a "vectorized" version of Min-Sum BP performs nearly as fast as the much simpler Algorithm E with significantly fewer decoding errors. Our experiments indicate that, for the best compromise between speed and performance, the decoder should use Min-Sum BP when the side information is of good quality and Sum-Product BP otherwise.**

## I. Introduction

Distributed source coding is an attractive option for sensor networks and surveillance systems in which image or video is acquired and compressed using low-cost hardware. This method of compression involves encoding the acquired signal conditioned on some statistically correlated side information at the decoder. For example, in the case of video signals, the side information for the current video frame can be furnished by a motion compensated version of the previous decoded frame. Distributed source coding draws on information-theoretic results on lossless coding of correlated sources [1] and rate-distortion tradeoffs for encoding of correlated sources [2]. Recent years have seen a revival in distributed source coding, in particular distributed video coding [3] which has exploited graph-based channel codes.

Nearly all implementations of distributed compression systems involve appropriately quantizing the signal of interest, and then extracting parity or syndrome symbols from it by applying a channel code. The syndromes constitute the compressed bit stream which is transmitted to the decoding station where, they are combined with the side information and fed to a channel decoder. The channel decoder essentially treats the side information as an error prone version of the signal of interest and uses the received syndromes to correct the errors, thereby recovering the signal of interest. This process, encompassing the side information decoding as well as the distortion introduced by quantization, is referred to as Wyner-Ziv coding. In theory, any channel code can be used in this way. In practice, however, graph-based channel codes such as Low-density Parity Check (LDPC) codes or Turbo Codes are preferred over hard-decision algebraic schemes such as Reed-Solomon codes or BCH codes owing to very low encoding complexity, and availability of soft-decision decoding algorithms that achieve better channel coding performance. LDPC encoding, for example, involves simple XOR operations, while LDPC decoding involves running a Belief Propagation (BP) algorithm to recover the signal of interest. An example of a distributed video coding system using an LDPC code is shown in Fig. 1. BP decoding is much more complex than the operations at the encoder.

In this paper, we consider BP decoding on general-purpose multi-core CPUs. The motivation is to make BP decoding fast enough for realtime decoding of time-sensitive signals such as surveillance videos. This is a practical requirement that has remained relatively unexplored in the literature; the emphasis has been on pure compression performance. In a detailed evaluation of the DISCOVER codec [4] on a general purpose dual core machine, the authors report that, to achieve high reconstruction quality even for QCIF-sized video frames, side information decoding of a single frame required 4-8 seconds depending upon the video content. By exploiting parallelization on multi-core CPUs used on consumer-level computers, we take a step toward real time decoding of Wyner-Ziv coded signals. The techniques in this paper also apply to the recent parallel implementations of BP on Graphics Processing Units (GPUs) based on NVIDIA's CUDA-based parallel computing architecture [5], [6], [7].

LDPC codes were invented by Gallager in the 1960s [8], but were ignored because of the limited processing capabilities at the time. They were rediscovered by MacKay and Neal [9] and since then, have received increased attention due to their near-Shannon-limit error performance. For distributed source coding, a class of rate-adaptive codes called LDPC Accumulate (LDPCA) codes [10] have become very popular. An LDPCA code is an LDPC code concatenated with an accumulator. The accumulator allows syndromes to be transmitted incrementally until decoding succeeds. For any set of accumulated syndromes, the decoding procedure is the same as that of a conventional LDPC code. In this paper, we focus on

† S. Momcilovic is with INESC-ID TU Lisbon, Portugal. This work was carried out when he was an intern at MERL.

978-1-4244-8112-5/10/$26.00 

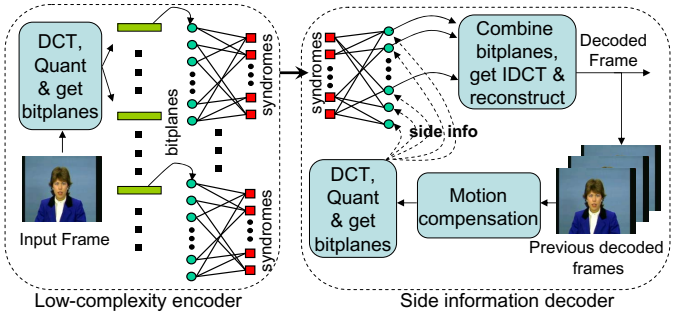



Fig. 1. The main components of a distributed video coding system. Decoding of only one bitplane is shown.

the LDPC decoding, keeping in mind that LDPCA decoding would need repeated invocations of LDPC decoding.

The remainder of the paper is organized as follows. Section II describes the three LDPC decoding algorithms evaluated in this paper. In particular, the calculations performed at the check nodes and variable nodes are described. Section III describes how parallelization is achieved by dividing decoding tasks among multiple processor cores and by incorporating vector instructions within each core. In Section IV, the speedup obtained via parallelization of the three BP decoding algorithms is discussed along with the tradeoff in decoding performance.

## II. LDPC Decoding Algorithms

An $(N, K)$ LDPC code is defined as the null space of a sparse parity check matrix $\mathbf{H}_{M\times N}$, where $N$ is the code length, $K$ is the code dimension, and $M \geq N - K$. The rate of the LDPC code, $R = K/N$. An LDPC code can also be represented by a bipartite Tanner graph with two types of nodes: *variable* nodes and *check* nodes. Each row in $\mathbf{H}$ corresponds to a check node and each column corresponds to a variable node; the $i^{\text{th}}$ check node is connected to the $j^{\text{th}}$ variable node if and only if $\mathbf{H}(i, j) = 1$.

Assume that the vectors being encoded are binary. For image and video applications, non-binary vectors constructed from blocks of pixels are converted into binary and the individual bitplanes are provided as inputs to the LDPC encoder, as shown in Fig. 1. Encoding consists of calculating the syndromes according to $\mathbf{s} = \mathbf{Hc}$ where $\mathbf{c}$ is the input binary sequence or bitplane and $\mathbf{s}$ is the syndrome vector. If $R < 1$, the syndrome vector $\mathbf{s}$ represents a compressed encoded version of the input vector $\mathbf{c}$, and is transmitted to the decoder.

To initialize side information decoding, the variable nodes in the LDPC code graph are populated with a hypothesis about the bits to be recovered. This hypothesis is obtained using a side information vector $\mathbf{v}$ which is correlated to the vector $\mathbf{c}$ which is to be recovered. In the simplest case, a starting hypothesis for $\mathbf{c}$ is the vector $\mathbf{v}$ itself. In general, the starting hypothesis expresses the likelihood that the bit $v_i$ in the $i^{\text{th}}$ variable node has value 0 or 1.

The check nodes are associated with the bits from the received syndrome vector $\mathbf{s}$. Each check node specifies a constraint equation satisfied by all the variable nodes connected to it. In BP decoding, messages are passed between the variable nodes and check nodes with the aim of enforcing these constraints. The messages propagate the beliefs at a given node to the other nodes connected to it. After a few iterations of message passing, the variable nodes should satisfy the constraints imposed by the check bits, in which case the decoding is deemed to be successful.

BP decoding can be realized either via a fully parallel flooding-type scheduling, or a fully serial shuffled-type scheduling [11], or a partial parallel group-shuffled approach [11]. Here, we focus on the fully parallel scheme in which the flow of operations is as indicated in Fig. 2. We consider 3 decoding algorithms: Sum-Product BP, Min-Sum BP and Algorithm E. The decoding operations for each of these three algorithms are detailed below. As a setup step, using binary phase shift keying (BPSK), the sequence to be encoded, i.e., $\mathbf{c}$ is mapped into $\mathbf{x}$ according to $x_n = 1 - 2c_n$. The destination observes a side information sequence $\boldsymbol{y}$ where $y_n = 1-2v_n \in \{-1, 1\}$. Denote the set of variable nodes connected to check node $j$ by $\mathcal{N}(j) = \{k : H_{jk} = 1\}$ and the set of check nodes connected to variable node $k$ as $\mathcal{M}(k) = \{j : H_{jk} = 1\}$. Denote using $\mathcal{N}(j)\backslash k$ the set $\mathcal{N}(j)$ with variable node $k$ excluded, and $\mathcal{M}(k)\backslash j$ the set $\mathcal{M}(k)$ with check node $j$ excluded. The following notation is used for the $i^{\text{th}}$ iteration of message passing:

$u_{mn}^{(i)}$: message from check node $m$ to variable node $n$
$v_{mn}^{(i)}$: message from variable node $n$ to check node $m$
$v_n^{(i)}$: belief of variable node $n$
$u_n$: message from the side-information channel for variable node $n$

### A. Sum-Product BP Decoding Operations

1) Initialization: Set $i = 1$, and the maximum number of iterations to $I_{\text{MAX}}$. For $1 \leq n \leq N$, set $u_n = \ln \frac{P(x_n=1|y_n)}{P(x_n=-1|y_n)}$. For each $m$, $n$, set $v_{mn}^{(0)} = u_n$ .
2) Iterative decoding:
   (a) Perform check node calculations, i.e., for $1 \leq m \leq M$ and each $n \in \mathcal{N}(m)$,

$$u_{mn}^{(i)} = 2\tanh^{-1}\left(\prod_{n'\in\mathcal{N}(m)\backslash n} \tanh\left(\frac{v_{mn'}^{(i-1)}}{2}\right)\right) \quad (1)$$

   For details about the derivation of the above formula, the reader is referred to [12].
   (b) Perform variable node calculations, i.e., for $1 \leq n \leq N$ and each $m \in \mathcal{M}(n)$,

$$v_{mn}^{(i)} = u_n + \sum_{m'\in\mathcal{M}(n)\backslash m} u_{m'n}^{(i)} \quad (2)$$

3) Hard decision and stopping criterion test:
   - Set $v_n^{(i)} = u_n + \sum_{m\in\mathcal{M}(n)} u_{mn}^{(i)}$. Create $\hat{\mathbf{c}}^{(i)} = [\hat{c}_n^{(i)}]$ such that $\hat{c}_n^{(i)} = 1$ if $v_n^{(i)} < 0$, and $\hat{c}_n^{(i)} = 0$ otherwise.
   - If $\mathbf{H}\hat{\mathbf{c}}^{(i)} = \mathbf{s}$ or if the number of iterations has reached $I_{\text{MAX}}$, stop decoding and go to Step 4. Otherwise set $i := i + 1$ and go to Step 2.
4) Output $\hat{\mathbf{c}}^{(i)}$ as the decoded codeword.

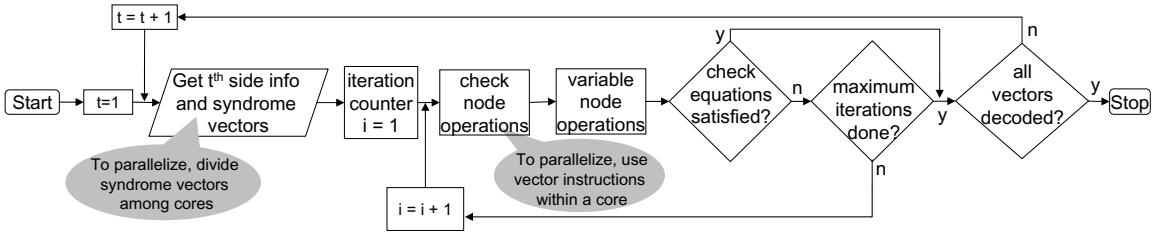



Fig. 2. A flow diagram containing a high-level summary of the sequence of operations performed in BP decoding.

### B. Min-Sum BP Decoding Operations

The Min-Sum algorithm [13] is a simplified version of Sum-Product BP. All decoding steps are the same as those in Sum-Product BP except the check node update in (1), which is now approximated by

$$u_{mn}^{(i)} = \prod_{n' \in \mathcal{N}(m) \backslash n} \operatorname{sgn}(v_{mn'}^{(i-1)}) \cdot \min_{n' \in \mathcal{N}(m) \backslash n} |v_{mn'}^{(i-1)}| \,. \quad (3)$$

As decoding primarily involves additions and comparisons, Min-Sum BP is less complex than Sum-Product BP.

### C. Algorithm E Decoding Operations

Algorithm E was proposed and analyzed in [14], [15]. It quantizes all the messages in Sum-Product BP to $-1$, $0$, or $+1$ and can be carried out as follows:

1) Initialization: Set $i = 1$ and the maximum number of iteration to $I_{\text{MAX}}$. For each $m$, $n$, set $v_{mn}^{(0)} = y_n$.
2) Iterative Decoding:
   (a) Perform check node calculations as follows: For $1 \leq m \leq M$ and each $n \in \mathcal{N}(m)$,
   $$u_{mn}^{(i)} = \prod_{n' \in \mathcal{N}(m) \backslash n} v_{mn'}^{(i-1)}$$
   (b) Perform variable node calculations as follows: For $1 \leq n \leq N$ and each $m \in \mathcal{M}(n)$,
   $$v_{nm}^{(i)} = \operatorname{sgn}\left( w^{(i)} \cdot y_n + \sum_{m' \in \mathcal{M}(n) \backslash m} u_{m'n}^{(i)} \right)$$
   where $\operatorname{sgn}(x)$ takes values -1, 0 or +1 for $x < 0$, $x = 0$, and $x > 0$ respectively, and $w^{(i)}$ is a weight chosen to optimize performance. For example, in [15], $w^{(1)} = 2$ and $w^{(i)} = 1$ for $i \geq 2$ is found to optimize the decoding performance for a regular $(3, 6)$ LDPC code.
3) Then, for the stopping criterion test, evaluate the variable node beliefs as
   $$v_n^{(i)} = \operatorname{sgn}\left( w^{(i)} \cdot y_n + \sum_{m' \in \mathcal{M}(n)} u_{m'n}^{(i)} \right)$$
   and proceed as in the Sum-Product BP algorithm

In terms of processing time per iteration, Algorithm E is faster than Sum-Product BP and Min-Sum BP owing to its simpler decoding operations. In the high signal-to-noise ratio (SNR) regime, where the side information is accurate and only a few bits are estimated in error, a even faster algorithm called Active-Set Algorithm E, or "Fast Algorithm E" has been proposed [16]. The rationale is that when SNR is large, most messages converge quickly. Thus, it is not necessary to update every variable/check node at each iteration. The decoder just checks whether the messages entering a node are different from their values in the previous iteration. If none of the messages has changed, the node is not updated and overall decoding time is reduced with no loss of performance.

## III. Implementation on multi-core CPUs

### A. Processor-level Parallelization

To speed up the execution of BP decoding, two kinds of parallelism are used. At the level of the processor cores, a single program/multiple data (SPMD) approach is used. This approach is useful in scenarios where the same set of instructions is executed in multiple iterations on *different* data. As there are no data dependencies between the iterations, they can be implemented independently and in any order on separate processor cores. The parallelization of such loops requires the creation of multiple threads and new thread contexts, by means of replication of the variables, e.g., counters, that will be private in each thread. All other variables are shared between the threads, and may be accessed concurrently by multiple threads. The creation of new thread contexts, and simultaneous access of the same variables by different processor cores results in a *parallelization overhead*. The speedup obtained via parallel execution needs to be large enough to overcome the effect of parallelization overhead.

In BP decoding, as shown in Fig. 2, there are loops at three levels. The top-most loop is on the "block" level, where the decoding algorithm is repeated for each received $M$-length block of syndromes. With the SPMD approach, this loop can be executed in parallel by allocating the syndrome blocks to the multiple cores, as there are no data dependencies between blocks. The mid-level loop is on the iteration level, where at each iteration, messages are exchanged between variable nodes and check nodes. The exchanged messages are different in each iteration and dependent on the messages from the previous iteration. Therefore, they can not be executed in any arbitrary order, and this loop cannot be parallelized via the SPMD approach. The innermost-level consists of two loops for computing the variable node messages and check node messages. In these loops, the same calculations are performed at each variable or check node, and with small modifications, these loops can be executed in parallel using the SPMD approach. The proposed scheme parallelizes the check node loop rather than the variable node loop, because check node processing was found experimentally to occupy a larger fraction of the processing time. Vector instructions

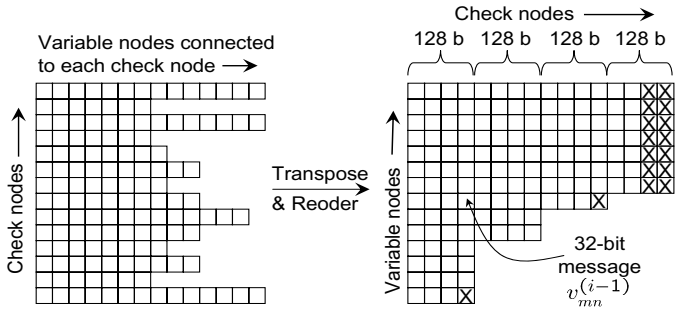



Fig. 3. Remapping the node indices to enable simultaneous processing of multiple check nodes. In this example, each small square is a 32-bit message.

are used in order to achieve parallelization of the check node calculations within a processor core. This is elaborated below.

### *B. Parallelization via Vector Instructions*

Vector instructions allow each processor core to process several check nodes simultaneously, thereby reducing the processing time per iteration. In order to exploit vector instructions to the fullest, it is necessary that the calculation performed at each check node is simple and similar to the calculations performed at every other check node. Unfortunately, Sum-Product BP evaluates the $\tanh(\cdot)$ function via direct computation or a table lookup, for which there is no efficient implementation using vector instructions. Also, Fast Algorithm E processes a node only if the messages entering it have changed since the last iteration and this non-uniformity makes it unsuitable for vector instructions. On the other hand, Min-Sum BP and Algorithm E can both use vector instructions because, in these algorithms, every check node is processed in nearly the same way as every other check node. One interesting observation, elaborated in Section IV is that, by using vector instructions for Algorithm E, the speedup obtained is enough to rival the speed of Fast Algorithm E.

To apply vector instructions, it is necessary to remap the messages between the check nodes and variable nodes, as shown in Fig. 3. In particular, it is necessary to arrange the messages such that they occupy $W$-bit blocks, where $W$ is the size of the largest block on which additions and logical operations can be performed. Thus, for a given data-type,

$$\text{No. of calculations in parallel} = \frac{W}{\text{sizeof(datatype)}}$$

Fig. 3 shows the data organization in memory for the case in which $W = 128$ and the messages are each 32 bits long, allowing a block of 4 check nodes to be processed in parallel. Each individual square contains a message from a variable node to the appropriate check node. The transposition ensures that messages related to a given check node are placed in successive locations in memory. Reordering as shown in the figure provides the most efficient way to process groups of 4 check nodes while inserting a minimum number of neutral messages. For the chosen word size of 32 bits, these neutral messages, marked "X" are placed to ensure that the number of messages being processed is a multiple of 4. The value of these neutral messages is set to zero for additions and to one for multiplications so that they do not affect the calculations.

TABLE I
DIMENSIONS OF LDPC MATRICES

| Matrix | Size | #edges | rate |
|---|---|---|---|
| $\mathbf{H}_G$ | 5940×1020 | 32635 | 0.83 |
| $\mathbf{H}_H$ | 5940×3000 | 32635 | 0.49 |
| $\mathbf{H}_I$ | 5940×5340 | 32635 | 0.1 |

To implement parallel check node operations in the above fashion for Min-Sum BP, the SSE[1] Vector Instruction Set from Intel is used. In Algorithm E, on the other hand, the check node calculations involve messages that take values -1 or 0 or +1 and they are implemented entirely using logical operations on $char$, i.e., 8-bit values. Thus, in the vector implementation of Algorithm E, 16 check nodes can be processed simultaneously.

## IV. EXPERIMENTAL RESULTS

All experiments were conducted on an Intel Core 2 Quad CPU Q9650 running at 3 GHz with 4 GB of RAM. The OpenMP Application Program Interface Version 3.0 [17] was used to implement parallelization of various BP decoding algorithms. This API provides C/C++ compiler directives and library routines to support shared-memory parallelism. The simulations were conducted on nine LDPC codes at various rates; we report results on the three largest codes in this paper. The parity check matrices are labeled $\mathbf{H}_G$, $\mathbf{H}_H$ and $\mathbf{H}_I$, and their dimensions are shown in Table I. These codes are all derived from a single LDPCA code with 5940 variable nodes. The code dimensions are motivated by a distributed video compression application[2]. The syndrome vectors have to be decoded with the help of side information, e.g., the previous video frame or a motion compensated version of it.

Video studies have shown that a Laplacian model is close to the observed dependency between the source bitplane and the side information bitplane. In this work, we are interested primarily in speedup from parallelization per iteration of LDPC decoding, not in choosing the LDPC code with the smallest number of check nodes or the LDPC code that converges in the smallest number of iterations. Since the parallelization speedup per iteration is independent of the channel model used, we assume a much simpler Binary Symmetric Channel (BSC) model between the source bitplane and the side information bitplane. Thus, if the crossover probability of the BSC is too large, the LDPC code will not be able to recover the source bitplane from the side information bitplane even after the maximum number of iterations, $I_{\text{MAX}}$, is reached. In all our simulations, $I_{\text{MAX}} = 100$.

First, consider the speedup obtained simply by dividing all the received syndrome vectors among the available processor cores. As shown in Fig. 4, the decoding time is the least for Algorithm E and the highest for Sum-Product BP. These

[1]SSE = Streaming SIMD Extensions

[2]Consider a video frame of size $720 \times 528$ pixels. An $8 \times 8$ blockwise Discrete Cosine Transform (DCT) is applied to the frame, each DCT coefficient is separately quantized and the resulting bitplanes are input to the LDPC encoder. For a single LDPC code to be applied to a particular DCT coefficient, the number of variable nodes in the LDPC code must be $\frac{720 \times 528}{8 \times 8} = 5940$. There is one LDPC code for every coded bitplane of each of the 64 DCT coefficients, and each of these codes transmits a syndrome vector to the decoder.

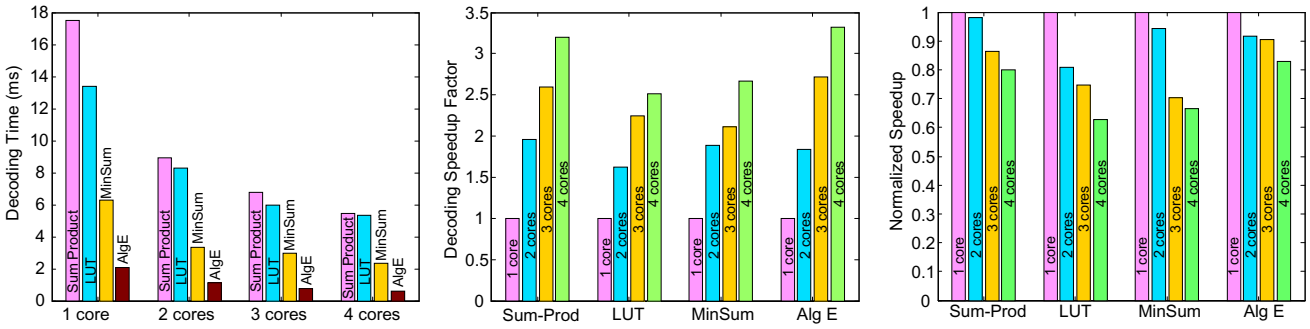



Fig. 4. As new cores are added, the speed of BP decoding increases, but parallelization overhead prevents a normalized speedup of one per core.

results are for the parity check matrix $\mathbf{H}_I$ with a low BSC crossover probability of 0.05. For each algorithm, the decoding times are averaged over 1000 decodings, i.e., the code is run 1000 times with the same syndrome vectors but with side information randomly perturbed according a BSC. The bar labeled "LUT" refers to an implementation of Sum-Product BP in which the $\tanh(\cdot)$ function is read from a look-up table with 32 bit precision. The LUT variant runs faster than Sum-Product BP, which uses a C-math function to compute $\tanh(\cdot)$ but has worse performance than Sum-Product BP, i.e., a larger number of uncorrected errors. The second bar graph plots the decoding speedup factor $S(y)$ while the third graph plots the normalized speedup $\bar{S}(y)$. These factors are given by:

$$S(y) = \frac{\text{Decoding time with 1 core}}{\text{Decoding time with } y \text{ cores}}, \quad \bar{S}(y) = \frac{S(y)}{y}$$

where $y$ is the number of cores used in parallel. The results show that, as more cores are added, the normalized speedup reduces because of the parallelization overhead associated with replicating thread contexts, and the contention that occurs when two threads access the same portions of memory.

Now, we describe the benefits of using vector instructions which speeds up the check node decoding operations within each core, as explained in Section III-B. Note that, for a fixed BP decoding algorithm, using vector instructions *does not* change the number of iterations needed for convergence, thus the coding performance of a BP decoding algorithm and its "vectorized" version are identical; the latter version just executes faster per iteration. As explained in Section III-B, Min-Sum BP and Algorithm E can be profitably vectorized as shown in Figs 5(a), (b) for the code matrices $\mathbf{H}_G$ and $\mathbf{H}_H$.

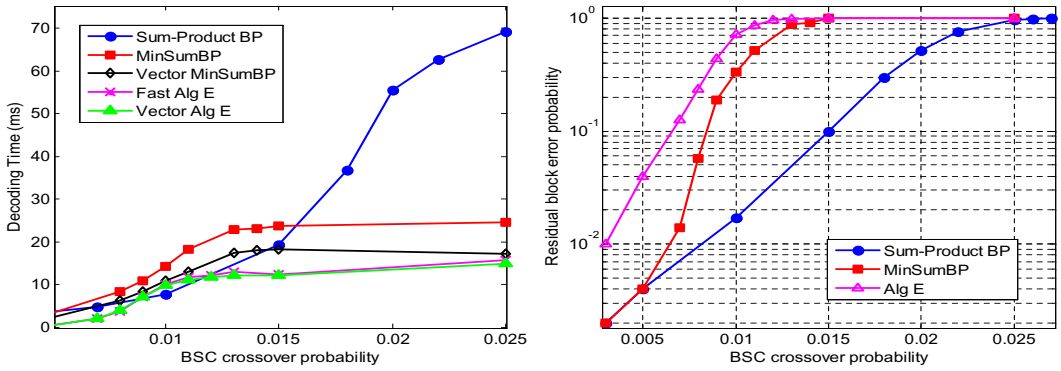
Firstly, the coding performance of Sum-Product BP is the best, followed by Min-Sum BP, followed by Algorithm E. This is expected because the latter two algorithms are approximations of Sum-Product BP. Further, as the crossover probability of the BSC between the source and side information bitplanes increases, the probability of uncorrected errors increases until it plateaus at 1, which means that there are undetected or uncorrected errors in every decoded vector. As there are more check nodes in $\mathbf{H}_H$ than $\mathbf{H}_G$, the plateau occurs at a higher crossover probability for the $\mathbf{H}_H$ code. When the crossover probability increases, more BP iterations are needed to recover the encoded vector, so the decoding time increases until the number of iterations maxes out at 100. A fact that is not visible from these plots is that Sum-Product BP converges in the fewest iterations but each iteration consumes more time.

Secondly, recall that Fast Algorithm E executes fewer iterations by first checking whether a node needs updating. The decoding time graphs show that, by using vector instructions in the plain Algorithm E, the decoding speed approaches and, in some cases, exceeds that of Fast Algorithm E. Note that, owing to the conditional checks, Fast Algorithm E is not suitable for implementation using vector instructions.

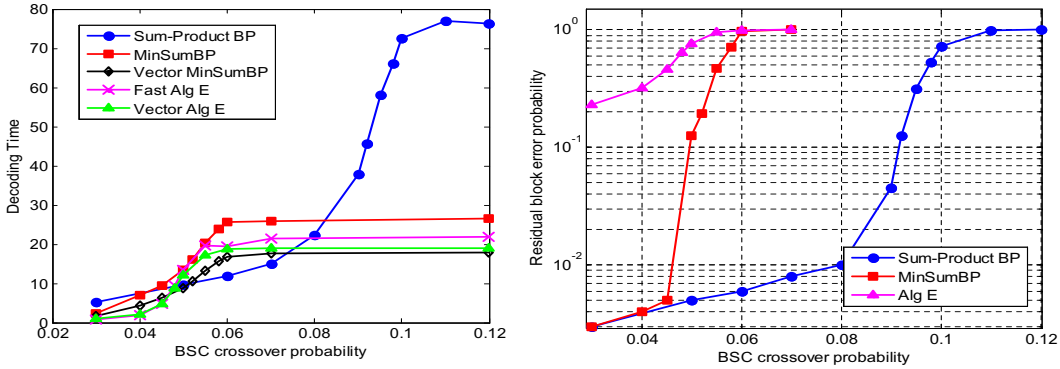
Thirdly, Min-Sum BP provides intermediate decoding performance and decoding speed between Sum-Product BP and Algorithm E. Interestingly however, with vector instructions, the Min-Sum BP decoding time is nearly as small as that of Algorithm E while retaining its superior decoding performance. The reason for this is that, check node operations consume 55-65% of the decoding time in Min-Sum BP, but only 35-45% of the decoding time in Algorithm E. Since vectorization reduces check node processing time, Min-Sum BP benefits more from vectorization than Algorithm E. We conclude that Vector Min-Sum BP is nearly always to be preferred over Algorithm E for side information decoding. When the crossover probability is low, e.g., while decoding the higher significant bitplanes of image pixels, Vector Min-Sum BP is to be preferred over Sum-Product BP because it gives the same performance in less time. However, when the crossover probability increases, e.g., while decoding the middle bitplanes of image pixels, Sum-Product BP gives significantly fewer decoding errors and must be preferred over Min-Sum BP even though it is slower.

## V. Discussion

To see the decoding time results in the context of a video viewing application, consider the following very rough calculation: Suppose that Min-Sum BP decoding is performed and we can tolerate error-prone decoded blocks with probability less than 0.01. From Fig. 5, this requirement is satisfied for BSC crossover probability 0.05 for the code $\mathbf{H}_H$, for example. The decoding time for vector Min-Sum BP at this probability is 8.8 ms. With an $8 \times 8$ block DCT transform, there are 64 coefficients to be coded. However, not all bitplanes of each DCT coefficient are significant. At 40 dB quality in natural images, we found experimentally that, out of $64 \times 8 = 512$ bitplanes, it is necessary to code about 300 bitplanes. Thus, 300 Min-Sum BP decodings must be carried out per video frame. For the code $\mathbf{H}_H$, this gives a total decoding time of




(a) Parity Check Matrix $\mathbf{H}_G$, code rate 0.83




(b) Parity Check Matrix $\mathbf{H}_H$, code rate 0.49

Fig. 5. A comparison of the speeds and performance of Sum-Product BP, Min-Sum BP and Algorithm E at various crossover probabilities.

$300 \times 0.0088 = 2.64$ seconds. This implies that the decoding speed is 0.38 frames/s for standard definition video, or 1.52 frames/s for CIF video, or 6.06 frames/s for QCIF video on a general-purpose Quad Core machine. There are many simplifying assumptions made above: Firstly, different code matrices would be required for each bitplane. More reliable bitplanes would decode faster than $\mathbf{H}_H$ and less reliable bitplanes would decode slower. Secondly, motion compensation is needed to generate good side information and this incurs additional delay. Nevertheless, it is encouraging to see that realtime Wyner-Ziv decoding is within reach on multicore CPUs and certainly on massively parallel GPUs. Our current work consists of combining parallelized BP decoding, parallelized motion compensation and improved side information decoding into a realtime distributed video decoder. In addition to side information decoding, the benefits of parallelization and vector instructions reported herein are expected to be useful in many other applications that use BP decoding - disparity estimation in multiview images/video, traditional digital communications, and speech recognition to name a few.

## REFERENCES


[1] D. Slepian and J. K. Wolf, "Noiseless Coding of Correlated Information Sources," *IEEE Trans. Information Theory*, pp. 471–480, July 1973.

[2] A. D. Wyner and J. Ziv, "The rate-distortion function for source coding with side information at the decoder," *IEEE Trans. Information Theory*, vol. 22, pp. 1–10, Jan. 1976.

[3] B. Girod, A.Aaron, S. Rane, and D. Rebollo-Monedero, "Distributed video coding," *Proceedings of the IEEE, Special Issue on Advances in Video Coding and Delivery*, vol. 93, no. 1, pp. 71–83, Jan. 2005.

[4] X. Artigas, J. Ascenso, M. Dalai, S. Klomp, D. Kubasov, and M. Ouaret, "The DISCOVER codec: Architecture, techniques and evaluation," in *Picture Coding Symposium*, Lisbon, Portugal, Nov. 2007.

[5] S. Grauer-Gray, C. Kambhamettu, and K. Palaniappan, "GPU implementation of belief propagation using CUDA for cloud tracking and reconstruction," in *5th IAPR Workshop on Pattern Recognition in Remote Sensing (PRRS)*, Tampa, FL, Dec. 2008.

[6] S. Wang, S. Cheng, and Q. Wu, "A parallel decoding algorithm of LDPC codes using CUDA," in *Proc. Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA, Oct. 2008.

[7] A. D. Copeland, N. B. Chang, and S. Leung, "GPU accelerated decoding of high performance error correcting codes," in *2009 High Performance Embedded Computing (HPEC)*, Lexington, MA, Sept. 2009.

[8] R. G. Gallager, "Low-density parity-check codes," *M.I.T. Press*, 1963.

[9] D. J. MacKay and R. M. Neal, "Near Shannon-limit performance of low density parity check codes," *Electronics Letters*, vol. 32, pp. 1645–1646, 1996.

[10] D. Varodayan, A. Aaron, and B. Girod, "Rate-adaptive codes for distributed source coding," *EURASIP Signal Processing Journal*, vol. 86, no. 11, pp. 3123–3130, Nov. 2006.

[11] J. Zhang and M. Fossorier, "Shuffled iterative decoding," *IEEE Transactions on Communications*, vol. 53, no. 2, pp. 209–213, 2005.

[12] F. R. Kschischang, B. J. Frey, and H. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Information Theory*, vol. 47, pp. 498–519, Feb. 2001.

[13] N. Wiberg, *Codes and Decoding on General Graphs*. Studies in Sci. and Technol., Dissertation no. 440, Linköping, Sweden, 1996.

[14] M. Mitzenmacher, "A note on low density parity check codes for erasures and errors," in *SRC Tech. Note 1998-017, COMPAQ*, 1998.

[15] T. J. Richardson and R. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Information Theory*, vol. 47, pp. 599–618, Feb. 2001.

[16] Y. Wang, J. S. Yedidia, and S. C. Draper, "Multi-stage decoding of LDPC codes," in *IEEE Int. Symp. Inform. Theory*, June 2009, pp. 2151–2155.

[17] *OpenMP Version 3.0 Application Program Interface*. OpenMP Architecture Review Board, May 2008.