Engineering Journal issn: 0125-8281

Article

A Practical Nonbinary Decoder for Low-Density Parity-Check Codes with Packet-Sized Symbols

Usana Tuntoolavest[a,*] **and Visuttha Manthamkarn**[b]

Department of Electrical Engineering, Faculty of Engineering, Kasetsart University, Bangkok, Thailand
E-mail: [a,*]usana.t@ku.th (Corresponding author), [b]visuttha.m@ku.th

Abstract. This paper presents a practical decoder for regular low-density parity-check (LDPC) codes with flexible packet-sized symbols. The proposed hMP-VSD (Combined hard-decision message-passing with vector symbol decoding) is much less complex than the conventional VSD and has the same decoding performance. Regular LDPC codes with systematic encoding are selected for implementation. The channel is assumed to be the q-ary symmetric channel (q-SC). Different code lengths and column weights of LDPC codes are investigated. The results show that the codes with a column weight of 7 provide the best performance for hMP-VSD, while hMP works best with codes having a column weight of 5. With packet-sized symbols, even the rather short (60, 30) code structure has code lengths of 1,920 to 245,760 bits with symbol sizes of 32 to 4,096 bits. Both the decoder and its encoder were implemented on Raspberry-pi 4 model B boards and these results confirm that the computation time of hMP-VSD is 60% to 70% lower than that of VSD for p_e in the range 0.05 to 0.1.

Keywords: Nonbinary decoder, hard-decision message-passing, low-complexity decoder, packet-sized symbol, Raspberry-pi, low-density parity-check codes.

ENGINEERING JOURNAL Volume 26 Issue 9
Received 7 February 2022
Accepted 20 September 2022
Published 30 September 2022
Online at https://engj.org/
DOI:10.4186/ej.2022.26.9.35

DOI:10.4186/ej.2022.26.9.35

1. Introduction

Low-density parity-check (LDPC) codes, which are widely used error-correcting codes, were first proposed by Gallager in 1963 [1]. After being long forgotten, possibly because of a lack of computing power at that time, they were rediscovered by Mackay and Neal in 1996 [2]. These codes have many applications because they achieve performance close to Shannon's limit in the additive white-Gaussian noise channel [3], [4], [5]. Some applications of LDPC codes are: the 802.11n/ac (Wi-Fi) standard [6], ship-based satellite communications on the move [7], and 2D and 3D NAND flash memories [8]. The message-passing (MP) technique is a common decoder for LDPC codes [3], [9]. LDPC codes usually use binary symbols and soft-decision decoding. The code length can be extremely long, up to thousands of bits, and codewords can be used as a packet in the data network. However, MacKay, who rediscovered LDPC codes, stated that nonbinary LDPC codes in the binary symmetric channel and binary Gaussian channel had better decoding performance than the binary LDPC codes [10]. That study showed the results for $GF(q)$ for 2^i; i = 2, 3, 4, which were two bits per symbol to four bits per symbol. Nonbinary LDPC codes were also investigated by other researchers, such as Cho et al. [3], who proposed the design of nonbinary LDPC codes with message-passing algorithms. They considered $GF(q)$ for $q = 2^i$; i = 2, 3, 4, 5, 6, having two bits per symbol to six bits per symbol. Nonbinary LDPC codes were also considered in the fountain-coding scheme in the erasure channel by Kasai [11].

The current paper considers LDPC codes on a much larger scale by applying them to packet-sized symbols with a minimum of 32 bits per symbol and no upper limit on the symbol size. In this case, a packet is considered as a symbol in a codeword; therefore, a codeword consists of many data packets and parity-check packets. With packet-sized symbols, soft-decision message-passing (sMP) is no longer practical because the complexity increases rapidly with the symbol size. Hard-decision message-passing (hMP) becomes attractive because of its low-complexity. hMP requires less power, fewer hardware resources, and provides higher speed than sMP [12]. The tradeoff is that the performance of hMP is lower than sMP for all SNR levels [13].

The bit-flipping (BF) algorithm is well-known in hard-decision decoding for LDPC codes. The first BF method was proposed by Gallager in 1962 [14]. In each iteration, the syndromes were computed. Then, the most suspicious bit was selected and flipped if the number of corresponding unsatisfied check nodes was higher than or equal to a fixed threshold. However, its performance was not sufficient. To improve the performance of BF, soft-information-aided BF decoding, such as the gradient descent BF (GDBF) [15], noisy-GDBF (NGDBF) [16], and adjustment factor-aided-NGDBF (A-NGDBF) algorithms [17] have been proposed. In addition, multibit-flipping (MBF) [18] based on the hard-decision BF algorithm was proposed to improve the performance for NAND storage systems. For MBF, the decoder allowed multiple bits to be flipped during each iteration. Although the proposed method uses hard-decision decoding as does BF, it uses large nonbinary symbols from $GF(2^r)$ where $r \geq 32$ bits, while BF uses binary symbols. Therefore, it is based on a different concept from BF. The proposed method is based on verification-based decoding [19] and does not need information from the channel.

Even though hMP cannot provide excellent decoding performance by itself, it can be very helpful when used in combination with another nonbinary decoder called vector symbol decoding (VSD). VSD is a verification-based decoding technique for codes that use r-bit nonbinary symbols from $GF(2^r)$, where $r \geq 32$ bits [20]. Examples of 446, 892, and 1,784 bits per symbol were used in [21] for VSD with convolutional codes. Larger sizes can be extended with the same number of computations because the decoding does not depend on the symbol size. The complexity of VSD depends on the number of erroneous symbols.

The conventional method for encoding LDPC code uses a systematic generator matrix derived from the parity-check matrix by Gaussian elimination. However, this method is inefficient because its complexity is $O(n^3)$ for preprocessing and $O(n^2)$ for actual encoding, where n is the codeword length. Therefore, the encoding complexity is high for LDPC codes, which generally use a large block-size n [22]. Another encoding method proposed in [23] can be applied to any LDPC code with much lower complexity than the conventional method. In addition, the codeword is still separated into a data part and a parity part. However, its complexity still increases linearly with the block size. For packet-sized symbols, shorter codes can accommodate the same amount of binary data as long binary codes because each symbol consists of many bits. With systematic encoding [23], the codeword is separated into the data part and the parity part. Therefore, VSD can identify the positions of the data symbols without using a systematic parity-check matrix $\boldsymbol{H}$.

The current paper proposes a new suboptimal, low-complexity decoding algorithm for packet-sized symbols called combined hard-decision message-passing with vector symbol decoding (hMP-VSD). This hMP-VSD consists of two parts, namely, hMP (the pre-decoder) and VSD (the main decoder). The overall complexity of MP-VSD is lower than that of VSD, but the performance remains the same. The preliminary idea of using hMP to reduce the complexity of VSD has been presented in [24]. However, then, it was impractical, because only a nonsystematic code was investigated. Since the decoder outputs the decoded codeword, it is troublesome to map this to the decoded data. An example of a method to map decoded codewords into decoded data for convolutional VSD was patented in [25]. However, the encoding part was not addressed in [24]. To make it practical, systematic LDPC codes were considered in [26]

by converting the nonsystematic regular LDPC codes to their systematic pairs with row operations. However, only the hMP part was investigated in [26] without the VSD part. The implementation of hMP-VSD in Raspberry-pi boards for systematic LDPC codes was introduced in [27].

The current paper presents the complete coding system, including both the encoder and hMP-VSD decoder with regular LDPC codes. Regular LDPC codes with systematic encoding are selected instead of the systematic codes in [26], [27] because they provide better decoding performance. The decoding performance of the complete hMP-VSD for regular LDPC codes with systematic encoding is investigated in detail for various code lengths and rates. From the effect of the density of bit 1s in $\boldsymbol{H}$ discovered in [26], this paper considers codes with different densities of 1s in $\boldsymbol{H}$ as well. The reduced complexity of the decoder is shown for these codes. In the simulations, the q-ary symmetric channel (q-SC) model is used to represent a packet-based erroneous channel [28], [29]. Furthermore, the encoder and decoder for regular LDPC codes with systematic encoding have been implemented on Raspberry-pi boards to show that the number of computations is low enough to be practical.

2. Materials and Method

2.1. Background

2.1.1. Hard-decision message-passing

MP is a decoding technique for LDPC codes [3], conveniently used with Tanner graph. Tanner graph is a bipartite graph consisting of two groups of disjointed nodes, namely, the set of variable nodes and the set of check nodes. Figures 1 and 2 show the relationship between $\boldsymbol{H}$ and its corresponding Tanner graph. For 1 in position (i, j) of $\boldsymbol{H}$, there is a line linking a check node $\boldsymbol{c}_i$ to the variable node $\boldsymbol{v}_j$. During the decoding with MP, messages are passed back and forth between the variable and check nodes. The decoding is done iteratively. For binary LDPC codes, sMP is normally used and the messages are probabilistic values. However, for packet-sized symbols, hMP is used instead of sMP because sMP is impractical. A packet contains many bits; therefore, numerous possible values could occur. sMP must calculate the probabilities for all values and pass them to another node. If 32 bits per packets are used, there are $2^{32} = 4.29 \times 10^9$ different possible values. For longer packets, the number of possible values increases substantially.

If hMP is used, the message to be passed is only the bit sequence of each packet. The sum of values from many nodes will use the modulo-2 addition and this can be done simply with any length of the packets, as illustrated in Fig. 2.

$$\boldsymbol{H} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & & 0 \end{bmatrix} \begin{matrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{matrix}$$
$$v_1 \; v_2 \; v_3 \; v_4 \; v_5 \; v_6 \; v_7 \; v_8 \; v_9 \; \dots \; v_n$$

Fig. 1. Relationship between $\boldsymbol{H}$ and nodes of Tanner graph in Fig. 2.

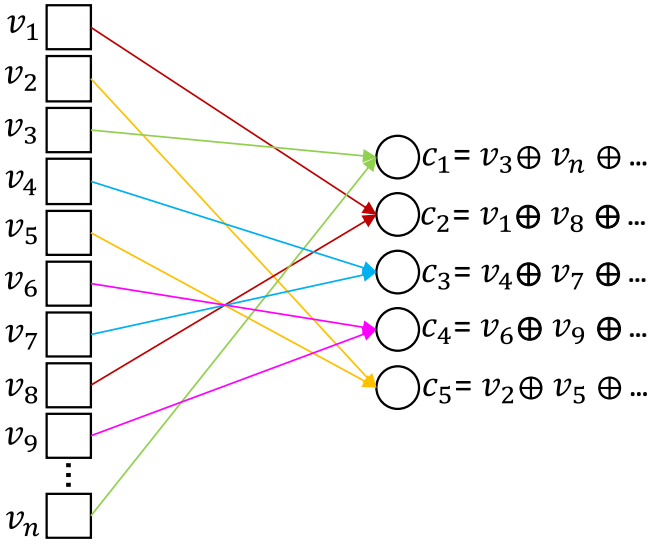

Fig. 2. Tanner graph with calculation of check node values from modulo-2 addition of variable node values.

2.1.2. Systematic encoding for nonbinary block codes

For any block code, the codeword can be computed from the data sequence and generator matrix. For binary codes, it involves the multiplication of a vector by a matrix as in Eq. (1). For nonbinary codes, it is the multiplication of a matrix by a matrix, as in Eq. (2).

$$\boldsymbol{v} = \boldsymbol{G} \cdot \boldsymbol{u} \tag{1}$$

$$\boldsymbol{V} = \boldsymbol{G} \cdot \boldsymbol{U} \tag{2}$$

where $\boldsymbol{v}$ is the column vector of a codeword, $\boldsymbol{u}$ is the column vector of a data sequence, $\boldsymbol{V}$ is the codeword matrix, $\boldsymbol{U}$ is the data matrix and $\boldsymbol{G}$ is the generator matrix.

Note that a bold small letter represents a vector, and a bold capital letter represents a matrix.

As $\boldsymbol{v}$ and $\boldsymbol{u}$ are column vectors, Eq. (1) is in a different form than the usual encoding equation normally seen where $\boldsymbol{v}$ and $\boldsymbol{u}$ are row vectors. These are chosen because it is more convenient to perform operations using $\boldsymbol{V}$ and $\boldsymbol{U}$ as matrices.

With systematic encoding, the data symbols can be transmitted out as the first part of the codeword. Then, the parity-check symbols can be computed from the vector modulo-2 sum of the data symbols specified by

each parity-check equation in $\boldsymbol{H}$, substantially simplifying the encoding of large nonbinary symbols.

2.1.3. VSD

VSD is suitable for linear codes with packet-sized symbols because its decoding complexity remains the same for any large symbol size. However, it is not designed for small-sized symbols because it assumes that all error patterns are linearly independent. Thus, the typical symbol size is 32 bits per symbol or more. The VSD algorithm has been described many times [24], [30]; therefore, it is only briefly explained here. The flowchart of the complete hMP-VSD and an example will be described later in the Method section.

VSD accepts the received symbols $\boldsymbol{y}_j$; j = 1, 2, …, n and places them as rows of the received matrix $\boldsymbol{Y}$. The syndrome matrix is then calculated from Eq. (3).

$$\boldsymbol{S} = \boldsymbol{H} \cdot \boldsymbol{Y} \tag{3}$$

where $\boldsymbol{S}$ is the syndrome matrix of size $(n - k) \times r$.
$\boldsymbol{H}$ is the parity-check matrix of size $(n - k) \times n$.
$\boldsymbol{Y}$ is the received matrix of size $n \times r$.
n is the number of total symbols in a code word of an (n, k) code.
k is the number of data symbols in a code word of an (n, k) code.
r is the size in bits of each symbol.

If $\boldsymbol{S}$ equals a zero matrix, the decoder concludes that there is no error. If it is not a zero matrix, an error-locating vector $\boldsymbol{\sigma}$ will be found to verify the correct symbol positions and the erroneous symbol positions. To find $\boldsymbol{\sigma}$, null combinations are discovered first because $\boldsymbol{\sigma}$ equals the result of the OR operations of null combinations. Each null combination can verify some symbols; therefore, the OR operations of the null combinations will verify all verifiable symbols. The null combination idea has also been applied in rateless code decoding [31].

The null combinations are a row of $\boldsymbol{H}$ or the modulo-2 sum of rows of $\boldsymbol{H}$. The row or sum of rows to be null combinations can be identified by the index of a zero-syndrome vector or by the index of the set of syndrome vectors that sum to a zero vector, explained with the example in the Method section. More examples of VSD can be found in [24]. Once the error-locating vector is discovered, the bit 1s in this vector refer to the correct symbols and the bit 0s refer to the unverified or apparent erroneous symbols. The decoder will compute the erroneous symbol values from Eq. (4).

$$\boldsymbol{E}_{sub} = \boldsymbol{H}_{sub}^{-1} \cdot \boldsymbol{S}_{sub} \tag{4}$$

where
$\boldsymbol{E}_{sub}$ is the submatrix of the error matrix $\boldsymbol{E}$ that contains only the erroneous symbol values.
$\boldsymbol{H}_{sub}$ is the square submatrix of the parity-check matrix $\boldsymbol{H}$ that contains only the columns corresponding to the error positions and the rows corresponding to linearly independent rows of $\boldsymbol{S}$.
$\boldsymbol{H}_{sub}^{-1}$ is the inverse matrix of $\boldsymbol{H}_{sub}$.
$\boldsymbol{S}_{sub}$ is the submatrix of the syndrome matrix $\boldsymbol{S}$ containing only the linearly independent rows of $\boldsymbol{S}$ that correspond to the row of $\boldsymbol{H}_{sub}$.

After $\boldsymbol{E}_{sub}$ is known, the error matrix $\boldsymbol{E}$ is also known because, for a correct symbol, the row of $\boldsymbol{E}$ for the symbol position is a zero vector. The decoder will obtain the decoded codeword matrix $\boldsymbol{V}$ from Eq. (5) and the decoding process is finished.

$$\boldsymbol{V} = \boldsymbol{Y} \oplus \boldsymbol{E} \tag{5}$$

where $\oplus$ is the vector modulo-2 sum.

2.1.4. Channel model

Assume a q-ary symmetric channel (q-SC) model for packet-based erroneous channel. This channel model is suitable for LDPC codes with symbols from $GF(q)$; $q = 2^m$; $m \in \mathbb{N}$ for large q [28], [29].

Let the random variable X be the input and random variable Y be the output of the q-SC with transition probabilities of:

$$P(Y = y \mid X = x) = 1 - p; x = y \tag{6}$$

$$P(Y = y \mid X = x) = p / (q - 1); x \neq y \tag{7}$$

where $x, y \in GF(2^m)$.

2.2. Method

2.2.1. Flowchart

The hMP-VSD algorithm is described with the help of the flowchart in Fig. 3. The hMP is the first part of the decoding process and could be considered as the pre-decoder. It can easily correct some simple error patterns. Then, the output of hMP will be input to VSD. Thus, the decoding performance of hMP-VSD is the same as VSD. The benefit of adding hMP is to reduce the number of erroneous symbols that VSD needs to correct. The algorithm description shows that VSD is required to invert a square matrix of size $e \times e$, where e is the number of erroneous symbols. Since the complexity of matrix inversion is $O(n^3)$ for a $n \times n$ matrix [32], reducing the matrix size substantially reduces the complexity.

The flowchart in Fig. 3 shows the proposed hMP-VSD decoder. The algorithm starts with the received matrix $\boldsymbol{Y}$, containing the received symbols $\boldsymbol{y}_j$; j = 1, 2, …, n as its row vectors. Then, the decoder evaluates if there are any new $\boldsymbol{y}_j$ values. At the beginning of the hMP decoding process, all $\boldsymbol{y}_j$ values are new; therefore, each check node will compute its value using the vector modulo-2 sum of all variable node values connected to it.

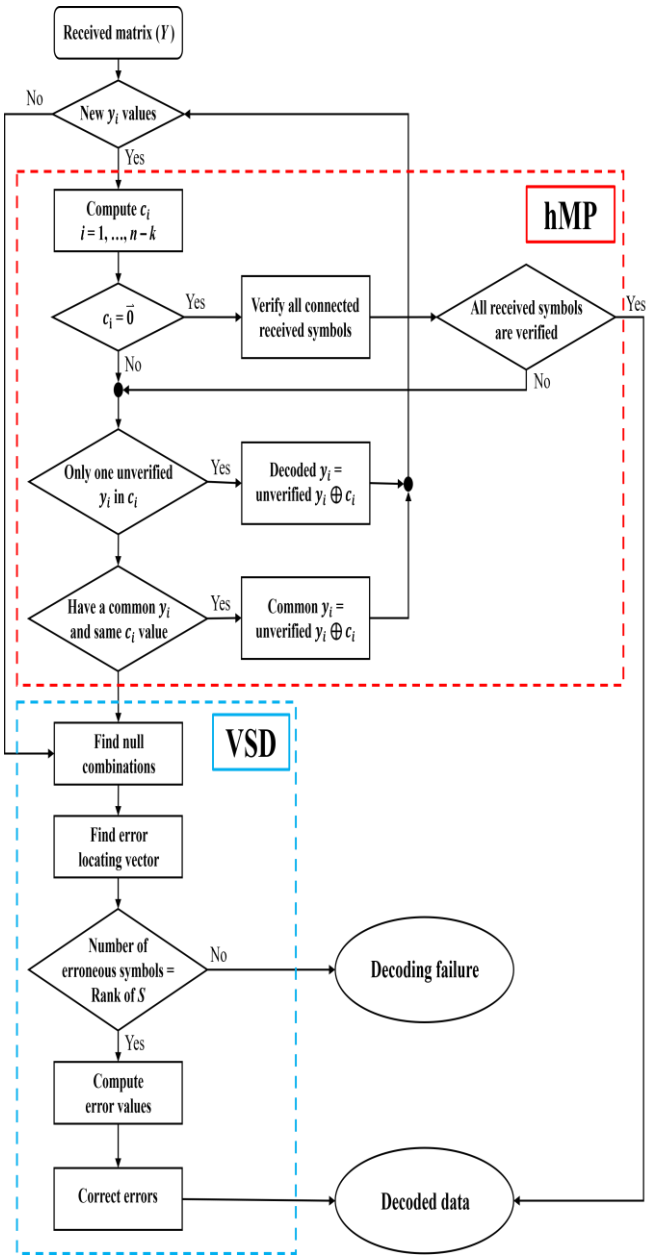

Fig. 3. Flowchart of hMP-VSD algorithm.

The check node value is the same syndrome value as the linear block codes. If a check node value $\boldsymbol{c}_i$ is equal to a zero vector, all variable nodes connected to it are verified to have correctly received symbol values. If all $\boldsymbol{c}_i$; $i = 1, 2, \ldots, n - k$ are zero vectors, all received symbols are verified to be correct, and the decoder can output the decoded data directly for systematic encoding.

For each nonzero $\boldsymbol{c}_i$, evaluation only occurs if there is only one unverified symbol connected to it. If there is, the decoded received symbol value $\boldsymbol{y}_j$ will be corrected as shown in Eq. (8).

$$\boldsymbol{y}_j = \boldsymbol{y}_j \oplus \boldsymbol{c}_i \tag{8}$$

This new received symbol value will be used to calculate the values of all check nodes connected to it.

If there is more than one unverified symbol connected to a check code, the decoder will evaluate whether there is only one common unverified symbol connected to at least two identical check nodes. If there is, the common received symbol value will be decoded as the received symbol $\boldsymbol{y}_j$ and corrected using Eq. (8). This new received symbol value will be used to calculate the values of all check nodes connected to it.

If there is no more correctable unverified symbol, the hMP part will be completed and the received matrix will be input to the VSD part. Since the check node values (the syndromes) have been calculated in the hMP part, VSD can start with finding the null combinations. These syndrome values are placed as row vectors of a syndrome matrix $\boldsymbol{S}$ for VSD. If the number of unverified symbols, which are considered erroneous symbols, from the null combinations is the same as the rank of $\boldsymbol{S}$, VSD can correct the errors with Eq. (4). If the number of erroneous symbols from the null combinations is not equal to the rank of $\boldsymbol{S}$, the decoder fails.

After the erroneous symbol values are found, the decoder will add them to the received symbol values. The result will be the decoded codeword. For systematic encoding, the data symbol positions are known; therefore, the decoded codeword can output the decoded data symbols directly from the decoded codeword.

Ex1. Consider a regular LDPC code with the parity-check matrix $\boldsymbol{H}$ in Eq. (9) and suppose the received matrix $\boldsymbol{Y}$ containing sixteen 5-bit symbols in Eq. (10) is received.

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{h}_1 \\ \boldsymbol{h}_2 \\ \boldsymbol{h}_3 \\ \boldsymbol{h}_4 \\ \boldsymbol{h}_5 \\ \boldsymbol{h}_6 \\ \boldsymbol{h}_7 \\ \boldsymbol{h}_8 \end{bmatrix} = \begin{bmatrix} 0010010110010001 \\ 0100001010001011 \\ 0001010000111001 \\ 1100001001100100 \\ 1010100001100010 \\ 0101000110010100 \\ 1000111101000000 \\ 0011100000001110 \end{bmatrix} \tag{9}$$

$$\boldsymbol{Y} = \begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \\ \boldsymbol{y}_3 \\ \boldsymbol{y}_4 \\ \boldsymbol{y}_5 \\ \boldsymbol{y}_6 \\ \boldsymbol{y}_7 \\ \boldsymbol{y}_8 \\ \boldsymbol{y}_9 \\ \boldsymbol{y}_{10} \\ \boldsymbol{y}_{11} \\ \boldsymbol{y}_{12} \\ \boldsymbol{y}_{13} \\ \boldsymbol{y}_{14} \\ \boldsymbol{y}_{15} \\ \boldsymbol{y}_{16} \end{bmatrix} = \begin{bmatrix} 01101 \\ 10100 \\ 00100 \\ 10001 \\ 00111 \\ 01110 \\ 11001 \\ 10100 \\ 10110 \\ 00010 \\ 01001 \\ 00100 \\ 11100 \\ 00100 \\ 00001 \\ 01011 \end{bmatrix} \tag{10}$$

Part I: hMP

Figure 4 illustrates how the check node values are calculated in the hMP algorithm. From the vector modulo-2 sum of the connected variable nodes, the

check node values are found to be $\boldsymbol{c}_1$ = 00111, $\boldsymbol{c}_2$ = 01101, $\boldsymbol{c}_3$ = 00101, $\boldsymbol{c}_4$ = 01111, $\boldsymbol{c}_5$ = 00100, $\boldsymbol{c}_6$ = 00111, $\boldsymbol{c}_7$ = 01011, and $\boldsymbol{c}_8$ = 01011.

No check node is equal to a zero vector; therefore, no received symbol is verified. However, there is a common unverified symbol $\boldsymbol{y}_5$ connecting to both $\boldsymbol{c}_7$ and $\boldsymbol{c}_8$. These two check nodes also have the same value. Consequently, the decoder will correct $\boldsymbol{y}_5$ by adding the value of $\boldsymbol{c}_7$ or $\boldsymbol{c}_8$ to it, as in Eq. (11), showing the decoded $\boldsymbol{y}_5$.

$$y_5 = y_5 \oplus c_7 = 00111 \oplus 01011 = 01100 \quad (11)$$

Next, the check node values $\boldsymbol{c}_7$ and $\boldsymbol{c}_8$ connected to $\boldsymbol{y}_5$ are recomputed to be zero vectors. These zero vector check nodes verify all variable symbols connected to these check nodes. In this case, $\boldsymbol{y}_1$, $\boldsymbol{y}_5$, $\boldsymbol{y}_6$, $\boldsymbol{y}_7$, $\boldsymbol{y}_8$, and $\boldsymbol{y}_{10}$ are verified by $\boldsymbol{c}_7$. In addition, $\boldsymbol{y}_3$, $\boldsymbol{y}_4$, $\boldsymbol{y}_5$, $\boldsymbol{y}_{13}$, $\boldsymbol{y}_{14}$, and $\boldsymbol{y}_{15}$ are verified by $\boldsymbol{c}_8$. After these verifications, there is one unverified variable symbol, $\boldsymbol{y}_{11}$, connected to $\boldsymbol{c}_5$. Then, the decoder corrects $\boldsymbol{y}_{11}$ by adding the value of $\boldsymbol{c}_5$ to it, as shown in Eq. (8). Then, the new check node value $\boldsymbol{c}_5$ will be a zero vector, verifying all symbols connected to it.

Since there are no more unverified symbols that can be corrected by hMP, hMP will output these updated received symbols to VSD, which is the part II of this hMP-VSD algorithm.

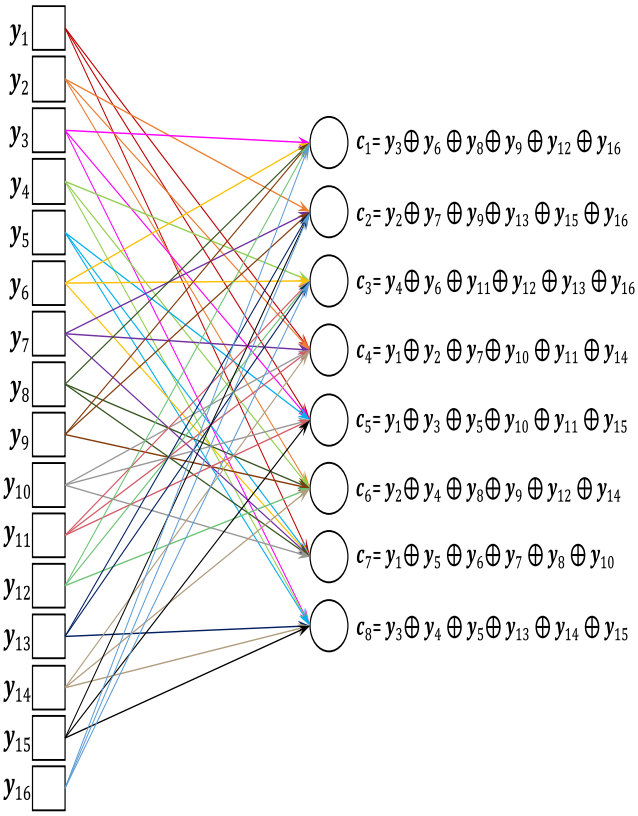

Fig. 4. Calculations of check node values.

Part II: VSD

The decoder computes the syndrome matrix $\boldsymbol{S}$ from Eq. (3). For this example, the calculation and the result are shown in Eq. (12) and Eq. (13), respectively.

$$\boldsymbol{S} = \begin{bmatrix} 0010010110010001 \\ 0100001010001011 \\ 0001010000111001 \\ 1100001001100100 \\ 1010100001100010 \\ 0101000110010100 \\ 1000111101000000 \\ 0011100000001110 \end{bmatrix} \cdot \begin{bmatrix} 01101 \\ 10100 \\ 00100 \\ 10001 \\ 01100 \\ 01110 \\ 11001 \\ 10100 \\ 10110 \\ 00010 \\ 00110 \\ 00100 \\ 11100 \\ 00100 \\ 00001 \\ 01011 \end{bmatrix} \quad (12)$$

$$\boldsymbol{S} = \begin{bmatrix} \boldsymbol{s}_1 \\ \boldsymbol{s}_2 \\ \boldsymbol{s}_3 \\ \boldsymbol{s}_4 \\ \boldsymbol{s}_5 \\ \boldsymbol{s}_6 \\ \boldsymbol{s}_7 \\ \boldsymbol{s}_8 \end{bmatrix} = \begin{bmatrix} 00111 \\ 01101 \\ 01010 \\ 00000 \\ 00000 \\ 00111 \\ 00000 \\ 00000 \end{bmatrix} \quad (13)$$

The null combinations are shown by one or more syndromes that sum to the zero vector with modulo-2 addition. These are $\boldsymbol{s}_4$, $\boldsymbol{s}_5$, $\boldsymbol{s}_7$, $\boldsymbol{s}_8$, $\boldsymbol{s}_1 \oplus \boldsymbol{s}_6$, $\boldsymbol{s}_1 \oplus \boldsymbol{s}_2 \oplus \boldsymbol{s}_3$, and $\boldsymbol{s}_2 \oplus \boldsymbol{s}_3 \oplus \boldsymbol{s}_6$. The corresponding null combinations are $\boldsymbol{h}_4$, $\boldsymbol{h}_5$, $\boldsymbol{h}_7$, $\boldsymbol{h}_8$, $\boldsymbol{h}_1 \oplus \boldsymbol{h}_6$, $\boldsymbol{h}_1 \oplus \boldsymbol{h}_2 \oplus \boldsymbol{h}_3$, and $\boldsymbol{h}_2 \oplus \boldsymbol{h}_3 \oplus \boldsymbol{h}_6$. The values of these null combinations are shown in Eq. (14)–(20). The error-locating vector ($\boldsymbol{\sigma}$) is computed from the OR operations of null combinations, as shown in Eq. (21).

$$\boldsymbol{h}_4 = 1100001001100100 \quad (14)$$
$$\boldsymbol{h}_5 = 1010100001100010 \quad (15)$$
$$\boldsymbol{h}_7 = 1000111101000000 \quad (16)$$
$$\boldsymbol{h}_8 = 0011100000001110 \quad (17)$$
$$\boldsymbol{h}_1 \oplus \boldsymbol{h}_6 = 0111010000000101 \quad (18)$$
$$\boldsymbol{h}_1 \oplus \boldsymbol{h}_2 \oplus \boldsymbol{h}_3 = 0111001100100011 \quad (19)$$
$$\boldsymbol{h}_2 \oplus \boldsymbol{h}_3 \oplus \boldsymbol{h}_6 = 0000011100100110 \quad (20)$$
$$\boldsymbol{\sigma} = 1111111101101111 \quad (21)$$

For larger matrices, the syndrome matrix is modified to contain rows of at most one bit of "1" with column operations starting from the first row of $\boldsymbol{S}$ until it can no longer generate new linearly independent rows. The new matrix is called a modified syndrome matrix $\boldsymbol{S'}$. An example of $\boldsymbol{S}$ and its corresponding $\boldsymbol{S'}$ are

$$\boldsymbol{S}=\begin{bmatrix}11100\\11010\\01001\\01001\\00000\\01001\\01001\end{bmatrix} \text{ and } \boldsymbol{S}'=\begin{bmatrix}10000\\01000\\00100\\00100\\00000\\00100\\00100\end{bmatrix}$$

The detail on how to obtain $\boldsymbol{S'}$ is in [33]. Notice that it is now obvious, which set of rows in the modified syndrome matrix $\boldsymbol{S'}$ sums to a zero vector. They are row 3 ⊕ row 4, row 5, row 3 ⊕ row 6, row 3 ⊕ row 7 etc. The same sets of rows in the syndrome matrix will also sum to a zero vector. This technique was used in the actual programming of the decoder.

The number of erroneous symbols is equal to the number of 0s in $\boldsymbol{\sigma}$ that is equal to 2, as shown in Eq. (21). Since this is equal to the rank of $\boldsymbol{S}$, the decoder can compute the error values from Eq. (4) as follows:

$$\boldsymbol{S}_{sub}=\begin{bmatrix}\boldsymbol{s}_1\\\boldsymbol{s}_2\end{bmatrix}=\begin{bmatrix}00111\\01101\end{bmatrix} \quad (22)$$

$$\boldsymbol{H}_{sub}=\begin{bmatrix}11\\10\end{bmatrix} \quad (23)$$

$$\boldsymbol{H}_{sub}^{-1}=\begin{bmatrix}0 & 1\\1 & 1\end{bmatrix} \quad (24)$$

$$\boldsymbol{E}_{sub}=\boldsymbol{H}_{sub}^{-1}\cdot\boldsymbol{S}_{sub}=\begin{bmatrix}01101\\01010\end{bmatrix}=\begin{bmatrix}\boldsymbol{e}_9\\\boldsymbol{e}_{12}\end{bmatrix} \quad (25)$$

The decoded codeword matrix ($\boldsymbol{V}$) can be computed by adding the error matrix ($\boldsymbol{E}$) to the $\boldsymbol{Y}$ with the modulo-2 sum, as shown in Eq. (26).

$$\boldsymbol{V}=\boldsymbol{Y}\oplus\boldsymbol{E}=\begin{bmatrix}01101\\10100\\00100\\10001\\01100\\01110\\11001\\10100\\10110\\00010\\00110\\00100\\11100\\00100\\00001\\01011\end{bmatrix}\oplus\begin{bmatrix}00000\\00000\\00000\\00000\\00000\\00000\\00000\\00000\\01101\\00000\\00000\\01010\\00000\\00000\\00000\\00000\end{bmatrix}=\begin{bmatrix}01101\\10100\\00100\\10001\\01100\\01110\\11001\\10100\\11011\\00010\\00110\\01110\\11100\\00100\\00001\\01011\end{bmatrix} \quad (26)$$

2.2.2. LDPC codes

The parity-check matrix in this paper is constructed using the (γ, ϱ)-regular LDPC code [34], where γ is the column weight and ϱ is the row weight. The simulation results use the column weights γ = 3, 5, 7, and 9 because γ = 3 is one of the most common column weights for regular LDPC codes [34] and $\gamma \geq 5$ is suitable for the bit-flipping algorithm [35].

2.2.3. Codes with packet-sized symbols

In different data networks, packets can differ in size, but they are at least several bytes. Given that the symbol is at least 32-bits long to satisfy the linearly independent erroneous symbols of VSD, the number of computations hMP-VSD remains the same when the symbol size increases. Specifically, r-bit symbols with higher values of r only lead to vector modulo-2 arithmetic of longer vectors. The main complexity depends on the matrix inversion part of VSD to recover error values, with the size of the matrix inversion being the number of erroneous symbols.

To limit the decoder complexity, rather short codes, such as (60, 30) and (120, 60) LDPC codes, are selected. This will keep the number of erroneous symbols in each codeword quite small for a given channel condition. For example, if the probability of packet errors from the channel is 0.1, a (60, 30) code will have an average of six erroneous symbols, whereas a (120, 60) code will have an average of 12 erroneous symbols. Inverting a 12 × 12 matrix is much more complex than inverting two 6 × 6 matrices. Since there is usually a built-in standard binary code to correct random errors in each packet, the remaining erroneous packets are burst errors or too many random errors. Therefore, the probability of packet errors around 0.1 reflects a channel with problems including excessive noise and/or fading and/or interference.

The total size of the codeword in binary depends on the symbol size. For example, with 32-bit symbols, the codeword will have 60 × 32 = 1,920 bits. The vector modulo-2 sum will be performed on 32-bit vectors. If a packet contains 100 bytes, it will be an 800-bit symbol and the codeword will have 60 × 800 = 48,000 bits. The vector modulo-2 sum will be performed on 800-bit vectors. Since the symbol size is adjustable, there is no need to use a long code structure.

2.2.4. Implementation in Raspberry-pi board

The Raspberry-pi4 Model B uses Broadcom BCM2711, Quad-core Cortex-A72 (ARM v8) 64-bit SoC and runs at a frequency of 1.5 GHz. It has 4 GB LPDDR4 of SDRAM and a Micro-SD card slot for data storage and its operating system. The operating system, Raspbian, is optimized for the Raspberry-pi board and based on Debian, which is a Linux distribution. This paper uses Raspberry-pi boards as an encoder and a decoder of hMP-VSD. The implementation of hMP-VSD on the Raspberry-pi board has been presented in [27] with a systematic code. In the current paper, a regular LDPC code with systematic encoding is implemented. In addition, the channel model is changed from the Gilbert-Elliot model to q-SC. Moreover, the computational time for decoding is investigated.

DOI:10.4186/ej.2022.26.9.35

3. Results

This section presents a performance comparison between hMP and hMP-VSD in terms of the probability of decoding failure. The computational time for decoding using a Raspberry-pi 4 board is presented to show the effectiveness of using hMP as the pre-decoder.

Table 1 shows the density of (γ, ϱ)-regular LDPC code where γ = 3 and 5. The density of a code is shown as the percentage of bits 1s in its $\boldsymbol{H}$ matrix. For instance, a (60, 30) LDPC code with (3, ϱ) contains 60 × 3 = 180 bits of 1s in 30 × 60 = 1,800 total elements in $\boldsymbol{H}$, which equates to a density of 10%.

Figure 5 shows the probability of decoding failure of hMP and hMP-VSD for (60, 30) with γ = 3, 5, 7, and 9. The simulations were applied in the q-SC model with a range of p_e from 0.05 to 0.20. Figure 6 shows the probability of decoding failure of the decoder with the same parameters as Fig. 5, except that the codes are changed to (120, 60). For both the (60, 30) and (120, 60) codes, codes with γ = 3 provide almost the same decoding performance for hMP and hMP-VSD. Its decoding performance of hMP-VSD is the worst compared to the other γ values. hMP works best with the codes having γ = 5; however, codes with γ = 7 provide the best performance for hMP-VSD.

Figure 7 shows the probability of decoding failure of hMP and hMP-VSD for (60, 30) codes with γ = 7 for three different symbol sizes (32 bits per symbol, 1,024 bits per symbol, and 4,096 bits per symbol). In all cases, the performance results for both hMP and hMP-VSD are the same. Only the green lines are seen in Fig. 7 because all hMP plots overlap and all hMP-VSD plots overlap. This emphasizes that the decoding performance of VSD does not depend on the symbol size, given that the symbol size is large enough to satisfy the assumption that the erroneous symbols are linearly independent. A typical symbol size for VSD is at least 32 bits per symbol.

The main complexity of VSD is the number and size of the matrix inversion required to obtain the error values. The size of the matrix inversion is equal to the number of erroneous symbols that VSD must correct. By adding the hMP part, the number of erroneous symbols input to VSD is decreased and this reduces the overall complexity. Figure 8 illustrates the percentage reduction in the number of matrix inversions required for VSD when hMP is used as the pre-decoder for the two different codes (60, 30) and (120, 60) with γ = 3, 5, 7, and 9. Different numbers of erroneous symbols from 2 to 10 symbols in each received sequence are investigated. The channel model is the q-ary symmetric channel with p_e = 0.05.

Figure 9 shows the comparison between the hMP-VSD and VSD decoders for an average decoding time per code word of the 32-bit symbol case using the Raspberry-pi 4 Model B board with a range of p_e from 0.05 to 0.1 for an (60, 30) code with γ = 7. In all cases,

Table 1. Density of $\boldsymbol{H}$ for each γ.

$\boldsymbol{H}$	γ	Density (%)
(60, 30)	3	10
	5	16.67
	7	23.33
	9	30
(120, 60)	3	3
	5	5
	7	11
	9	15

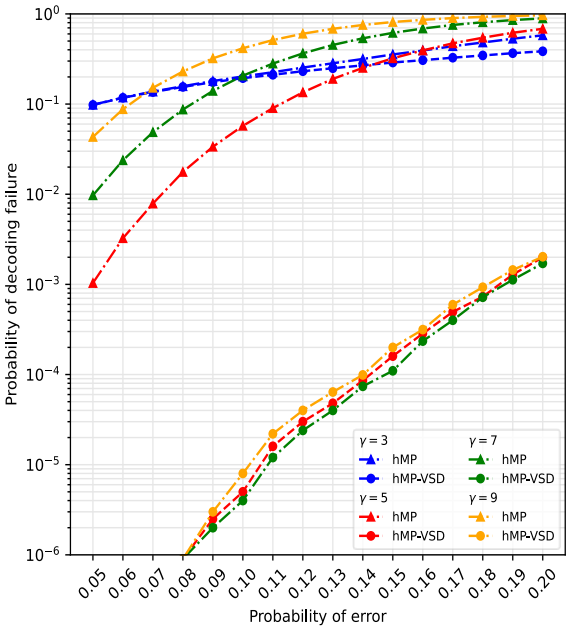

Fig. 5. Probability of decoding failure with decoders of hMP and hMP-VSD for (60, 30) codes.

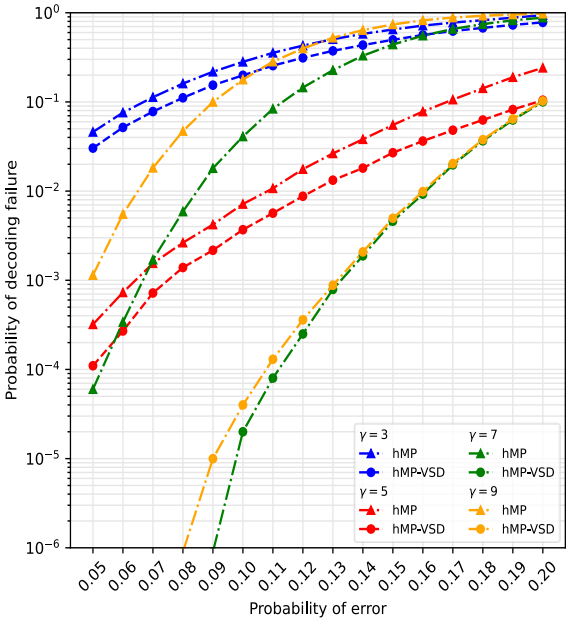

Fig. 6. Probability of decoding failure with decoders of hMP and hMP-VSD for (120, 60) codes.

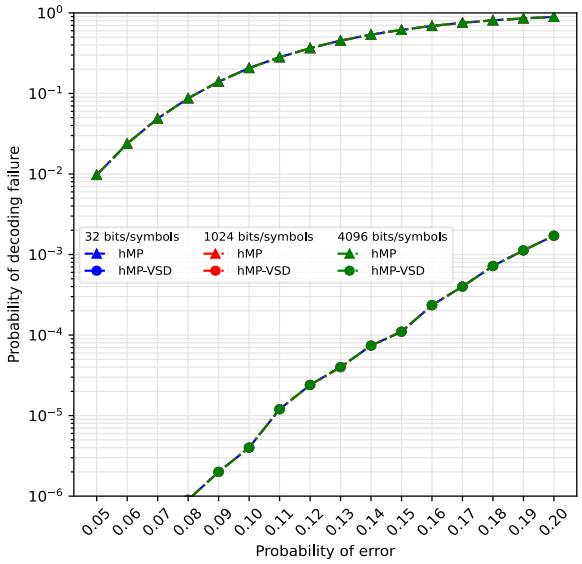

Fig. 7. Performance of hMP and hMP-VSD for (60, 30) code with $\gamma = 7$ for three sets of symbol sizes (32 bits per symbol, 1,024 bits per symbol, and 4,096 bits per symbol).

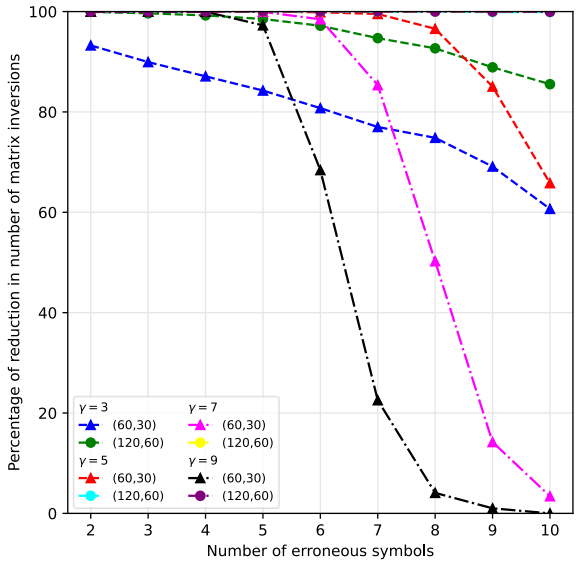

Fig. 8. Percentage of reduction in number of matrix inversion for different numbers of erroneous symbols when hMP is added as pre-decoder to VSD.

the hMP-VSD decoder takes substantially less time than the VSD decoder because hMP corrects errors using only the vector modulo-2 sum, whereas VSD must perform the matrix inversion to find the error values, which is considerably more complex than the vector modulo-2 sum. Based on these results, hMP-VSD takes approximately 60% less time than VSD (around 55% when $p_e = 0.1$ and up to 65% when $p_e = 0.05$).

An example of the results from the implementation of hMP-VSD on the Raspberry-pi board is shown using the original picture from Pexels [36] and a (60, 30) code with $\gamma = 7$. Figure 10 shows the data part of the received pre-decoded image from the Raspberry-pi board with $p_e = 0.2$. The received picture contains 46,010 erroneous data symbols in the 230,400 total data symbols or an input error probability of 1.997×10^{-1}. These erroneous data symbols are seen in the corrupted image in Fig. 10. After decoding using the hMP-VSD decoder, the number of erroneous data symbols substantially reduces to only 198 symbols or an output error probability of 8.59×10^{-4}, resulting in the good quality image shown in Fig. 11.

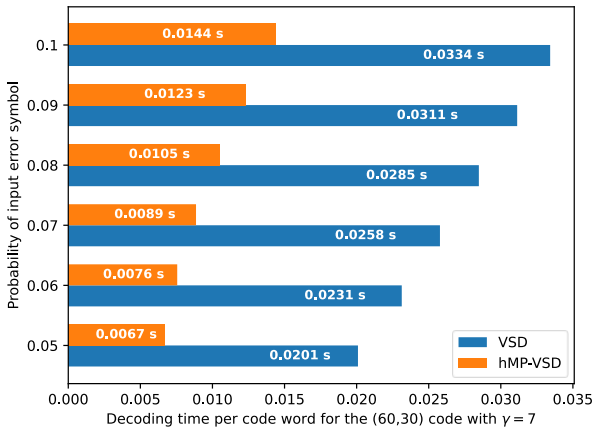

Fig. 9. Decoding time per code word with (60, 30) code with $\gamma = 7$ and 32 bits per symbol.

Fig. 10. Image of pre-decoded data part using Raspberry-pi board.

Fig. 11. Image of post-decoded data using Raspberry-pi board.

4. Discussions

Figures 5 and 6 compare the hMP part and hMP-VSD for the two sets of code lengths of 60 and 120 symbols and four values of γ = 3, 5, 7, and 9 for each code length. The simulation results in q-SC show that both the (60, 30) and (120, 60) codes with γ = 7 provide the best performance in the group. Furthermore, the closer the two graphs of the hMP part and hMP-VSD for each code, the more useful hMP will be as the pre-decoder because it can correct many erroneous symbols and leave smaller remaining erroneous symbols for VSD to correct. Even though these codes are low-density parity- check codes, the results from Fig. 5, Fig. 6, and Table 1 show that too low-density may not be preferable because VSD uses verification-based decoding, which can verify only the symbols involved in each parity-check equation. A very low-density results in each parity-check equation involving only a few received symbols. Therefore, VSD requires a higher number of parity-check equations to verify all the correctly received symbols. With a higher density, each received symbol has a greater chance of being in many parity-check equations and this increases the chance of VSD verifying all the correct symbols. Although the code density directly affects the performance, code with a too high-density does not provide good performance, as shown in Fig. 5 and 6. The results show that the decoding performance starts to drop when γ = 9 for both the (60, 30) and (120, 60) codes with regular $\boldsymbol{H}$.

Increasing the symbol size could increase the codeword length in bits. Because the proposed decoder allows the symbol size to increase as desired, the codeword length in bits can also be increased as desired. The simulation results in Fig. 7 emphasize that the decoding performance is maintained with any larger symbol size. The complexity of hMP-VSD is substantially lower than for VSD due to the reduction in the matrix inversion required. This is reflected clearly in the decreased computational time. An implementation result on a Raspberry-pi board is illustrated with a pre-decoded image and a post-decoded image (Fig. 10 and 11, respectively) to assist with easy visualization of the results.

5. Conclusions

Using hMP as the pre-decoder of VSD substantially reduces the complexity of VSD while maintaining the same decoding performance. Regular LDPC codes with systematic encoding are proposed for the practical use of hMP-VSD because VSD outputs the decoded codeword, not the decoded data sequence. The decoder can be used with flexible packet-sized symbols. Therefore, it can be easily adjusted to fit various channel conditions and network requirements. The proposed decoder and its corresponding encoder were successfully implemented on Raspberry-pi boards. The computational time confirms that hMP-VSD is considerably less complex than VSD.

Acknowledgement

The Kasetsart University Research and Development Institute (KURDI), Bangkok, Thailand provided support with proof reading and comments.

References

[1] R. G. Gallager, "Low-density parity-check codes," *in Research Monograph Series*. Cambridge: MIT Press, 1963.

[2] D. MacKay and R. Neal, "Near shannon limit performance of low density parity check codes," *IEEE Power Electron Lett.*, vol. 32, no. 18, pp. 1645-1646, 1996.

[3] S. Cho, K. Cheun, and K. Yang, "Design of nonbinary LDPC codes based on message-passing algorithms," *IEEE Trans. Commun.*, vol. 66, no. 11, pp. 5028-5040, 2018.

[4] T. J. Richardson and R. L. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 599-618, 2001.

[5] T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke, "Design of capacity-approaching irregular low density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 619-637, 2001.

[6] I. Tsatsaragkos and V. Paliouras, "A reconfigurable LDPC decoder optimized for 802.11n/ac applications," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 1, pp. 182-195, 2018.

[7] B. Wang, P. Chen, Y. Fang, and F. C. M. Lau, "The design of vertical RS-CRC and LDPC code for ship-based satellite communications on-the-move," *IEEE Access*, vol. 4, pp. 44977-44986, 2019.

[8] T. Nakamura, Y. Deguchi, and K. Takeuchi, "Adaptive artificial neural network-coupled LDPC ECC as universal solution for 3-D and 2-D, charge-trap and floating-gate NAND flash memories," *IEEE J. Solid-State Circuits*, vol. 54, no. 3, pp. 745-754, 2019.

[9] X. Zhang and P. H. Siegel, "Quantized iterative message passing decoders with low error floor for LDPC codes," *IEEE Trans. Commun.*, vol. 62, no. 1, pp. 1-14, 2014.

[10] M. Davey and D. MacKay, "Low density parity check codes over GF(q)," *IEEE Commun. Lett.*, vol. 2, no. 6, pp. 165-167, 1998.

[11] K. Kasai, D. Declercq, and K. Sakaniwa, "Fountain coding via multiplicatively repeated non-binary LDPC codes," *IEEE Trans. Commun.*, vol. 60, pp. 2077-2083, 2012.

[12] N. Mobini, A. H. Banihashemi, and S. Hemati, "A different binary message-passing LDPC decoder," *IEEE Trans. Commun.*, vol. 57, no. 9, pp. 2518-2523, 2009.

[13] R. Jose and A. Pe, "Analysis of hard decision and soft decision decoding algorithms of LDPC codes

in AWGN," in *2015 IEEE International Advance Computing Conference (IACC)*, Banglore, India, 2015.
[14] R. Gallager, "Low-density parity-check codes," *IRE Trans. on Inf. Theory*, vol. 8, pp. 21-28, 1962.
[15] T. Wadayama, K. Nakamura, M. Yagita, Y. Funahashi, S. Usami, and I. Takumi, "Gradient descent bit flipping algorithms for decoding LDPC codes," *IEEE Trans. Commun.*, vol. 58, no. 6, pp. 1610-1614, 2010.
[16] G. Sundararajan, C. Winstead, and E. Boutillon, "Noisy gradient descent bit-flip decoding for LDPC codes," *IEEE Trans. Commu.*, vol. 62, no. 10, pp. 3385-3400, 2014.
[17] B. Dai, R. Liu, C. Gao, and Z. Mei, "Noisy gradient descent bit-flipping decoder based on adjustment factor for LDPC codes," *IEEE Commun. Lett.*, vol. 22, no. 6, pp. 1152-1155, 2018.
[18] J. Jung and I.-C. Park, "Multi-bit flipping decoding of LDPC codes for NAND storage systems," *IEEE Commun. Lett.*, vol. 21, pp. 979-982, 2017.
[19] M.G. Luby and M. Mitzenmacher, "Verification-based decoding for packet-based low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 51, no. 1, pp. 120-127, 2005.
[20] U. Tuntoolavest, "A simple method to improve the performance of convolutional vector symbol decoding with small symbol size," in *2004 IEEE Region 10 Conference TENCON 2004*, Chiang Mai, Thailand, 2004.
[21] U. Tuntoolavest and K. Limchaikit, "Scaling method to increase data rate with no degradation in vector symbol decoding performance," in *The 20th Asia-Pacific Conference on Communication (APCC2014)*, Pattaya, Thailand, 2014.
[22] J. Lu and J. M. F. Moura, "Linear time encoding of LDPC codes," *IEEE Trans. Inf. Theory*, vol. 56, no. 1, pp. 233-249, 2010.
[23] T. J. Richardson and R.L. Urbanke, "Efficient encoding of low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 638-656, 2001.
[24] U. Tuntoolavest, C. Athanan, and K. Panwong, "Message passing-vector symbol decoding for LDPC codes with nonbinary symbols," in *2018 International Conference on Embedded Systems and Intelligent Technology & International Conference on Information and Communication Technology for Embedded Systems (ICESIT-ICICTES)*, Khon Kaen, Thailand, 2018.
[25] U. Tuntoolavest and T. Chaitanarit, "Mapping circuits for non-systematic convolutional codes and the procedure to implement them," *Petty Patent No. 17134*, 2018.
[26] U. Tuntoolavest, V. Manthamkarn, and A. Maheshwari, "Systematic low density parity check codes with hard decision message passing algorithm for non-binary symbols," in *2020 8th International Electrical Engineering Congress (iEECON)*, Chiang Mai, Thailand, 2020.
[27] A. Maheshwari, U. Tuntoolavest, and K. Fukawa, "Implementation of the nonbinary encoder and decoder for systematic low density parity check codes on Raspberry-pi boards," in *2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, Vancouver, BC, Canada, 2020.
[28] C. Weidmann, "Coding for the q-ary symmetric channel with moderate q," in *2008 IEEE International Symposium on Information Theory*, Toronto, Canada, 2008.
[29] F. Zhang and H. D. Pfister, "Analysis of verification-based decoding on the q-ary symmetric channel for large q," *IEEE Trans. Inf. Theory*, vol. 57, no. 10, pp. 6754-6770, 2011.
[30] J. J. Metzner, "Vector symbol decoding with list inner symbol decisions," *IEEE Trans. Commun.*, vol. 51, pp. 371-380, 2003.
[31] U. Tuntoolavest, N. Shaheen, and V. Manthamkarn, "Verification-Based Decoding for Rateless Codes in the Presence of Errors and Erasures," *Eng. J.*, vol. 26, no. 4, pp. 37-44, Apr. 2022.
[32] L. Ma, K. Dickson, J. McAllister, and J. McCanny, "QR decomposition-based matrix inversion for high performance embedded MIMO receivers," *IEEE Trans. Signal Process*. vol. 59, no. 4, pp. 1858-1867, 2011.
[33] P. Vanichchanunt *et al.*, *Channel Coding Theory*, 1st ed. (in Thai) Thailand: TRIDI (Telecommunications Research and Industrial Development Institute), 2009.
[34] X. Liu, F. Xiong, Z. Wang, and S. Liang, "Design of binary LDPC codes with parallel vector message passing," *IEEE Trans. Commun.*, vol. 66, pp. 1363-1375, 2018.
[35] S. K. Chilappagari and B. Vasic, "Error-correction capability of column-weight-three LDPC codes," *IEEE Trans. Inf. Theory*, vol. 55, pp. 2055-2061, 2009.
[36] Skitterphoto. (2017). *White daisy closeup photography* [Online]. Available: https://www.pexels.com/photo/white-daisy-closeup-photography-597055/ [Accessed 1 December 2020].

Usana Tuntoolavest received the B.S. degree in electrical engineering from Chulalongkorn University in 1995, and the M.S. and Ph.D. degrees from the Pennsylvania State University, PA, USA in 1997 and 2002, respectively. She is currently an Associate Professor at Kasetsart University, Bangkok, Thailand. She has received many awards including the Excellence Lecturer of the 5th ASAIHL-Thailand Award (2017) from the Association of Southeast Asian Institute of higher learning, Thailand.

Visuttha Manthamkarn received the B.S. degree in electrical engineering from Kasetsart University in 2019. He is now working as a researcher at Kasetsart University, Bangkok, Thailand. His current research interests include channel coding and implementation for the LDPC encoding/decoding.