Memory Efficient Decoder Architectures for Quasi-Cyclic LDPC Codes

Yongmei Dai, Ning Chen and Zhiyuan Yan
Department of Electrical and Computer Engineering
Lehigh University, PA 18015, USA
E-mails: {yod304, yan, nic6}@lehigh.edu

Abstract

In this paper, we first propose a parallel turbo-sum-product (PTSP) and a parallel turbo-shuffled-sum-product (PTSSP) decoding algorithms for quasi-cyclic (QC) low-density parity-check (LDPC) codes assuming partly parallel decoder architectures. We show that our proposed algorithm not only achieves faster convergence and better error performances than the sum-product (SP) or overlapped SP (OSP) algorithm, but also needs less memory in implementation. Then we propose two partly parallel decoder architectures. Finally, proof-of-concept FPGA implementation results are presented.

Index Terms

low-density parity-check (LDPC) codes, quasi-cyclic (QC) codes, turbo decoding, shuffled decoding, sum-product decoding.

I. INTRODUCTION

Low-density parity-check (LDPC) codes [1, 2] and their efficient implementations have been receiving a lot of attention due to their near-capacity performances when decoded iteratively. Various (fully parallel, fully serial, and partly parallel) decoder architectures (see, e.g., [3–14]) have been proposed. In general, random code structure is susceptible to implementation problems, whereas regular code structure tends to ease the hardware implementation. Quasi-cyclic (QC) LDPC codes [15, 16], which not only have
relatively good error performances but also can be efficiently encoded with shift registers (see, e.g., [17]) and decoded with partly parallel decoder architectures [7–9, 11–14], are thus of great interest.

LDPC codes can be decoded efficiently by the sum-product (SP) algorithm (see, e.g., [18]). An LDPC code can be represented by its Tanner graph [19], wherein the codeword bits and the parity check equations are represented as variable and check nodes respectively. Each iteration of the SP algorithm consists of two sequential steps: check node updates (CNUs) and variable node updates (VNU). There is inherent parallelism among the CNUs and VNU respectively in each iteration of the SP algorithm, but fully parallel decoder architectures [3, 4] require a large number of processing elements and are susceptible to congested routing network. Hence, partly parallel decoders are considered [5–9, 20, 21]. In particular, the memory-banks based partly parallel decoder architectures for QC LDPC codes [7–9, 20, 21] have been intensively studied due to their simplicity and efficiency. For this kind of architectures, one problem is the high power consumption due to frequent memory access. To reduce memory access without sacrificing error performances, a power efficient architecture has been proposed in [21]. Another problem is low throughput and low hardware utilization efficiency (HUE) resulting from the sequential processing of the CNUs and VNU. To overcome this problem, overlapped SP (OSP) decoders [8, 9] have been proposed. For QC LDPC codes, the OSP decoder [9] improves the throughput and HUE of the non-overlapped SP decoder by up to 100% while having the same error performance and using the same amount of memory.

One disadvantage of the SP algorithm is its slow convergence. To improve the convergence rate, many variants of the SP algorithm (see, e.g., [11, 13, 22–25]), which essentially change the order of the CNUs and VNU to expedite the message passing, have been proposed. Of particular importance are the turbo [11] and shuffled [23] decoding algorithms. Other algorithms (see, e.g., [13, 24, 25]) can be viewed as variants of these two algorithms. The turbo and shuffled decoding algorithms require fewer (up to 50%) iterations to converge and achieves better error performance than the SP algorithm while leading to significant memory savings in implementation [11]. One potential drawback of the turbo (or shuffled) decoding algorithm is that the serialization of the decoding of supercodes may offset its advantage in convergence rate and the decoding throughput of the turbo (or shuffled) algorithm may be limited. Thus, in [25], a two-supercode based turbo-sum-product (TSP) and a two-group based shuffled-sum-product (SSP) algorithms have been proposed, which achieve a better tradeoff between throughput and convergence rate improvement while saving the same amount of memory as the turbo and shuffled decoding algorithms, respectively. Partly parallel decoders implementing the turbo decoding algorithm have also been proposed [11, 12, 14, 22]. For memory-banks based decoder architectures, Shimizu proposed a high-efficiency message-passing decoder [13] that implements a variant of the turbo decoding algorithm. However, their decoder achieves faster
convergence at the expenses of more VNUs and doubling the memory requirement compared to the OSP decoder [9].

In this paper, we focus on the decoding of QC LDPC codes assuming memory-banks based partly parallel decoder architectures. By exploiting the TSP and SSP decoding algorithms, our goal is to design new partly parallel decoders that can achieve higher throughput (due to faster convergence) and better error performances than the OSP decoder [9], but require less memory. The main contributions of this paper are as follows:

- First, we propose two decoding algorithms for memory-banks based partly parallel decoder architectures: parallel TSP (PTSP) algorithm and parallel turbo-shuffled-sum-product (PTSSP) algorithm. In comparison to the TSP algorithm, we show that our PTSP algorithm introduces marginal error performance loss while saving the same amount of memory and almost doubling the throughput. In comparison to the SP (OSP) algorithm, our PTSP algorithm achieves faster convergence, better error performance and requires less memory. To further reduce the memory requirement, we propose the PTSSP algorithm, which essentially combines the advantages of the TSP and SSP algorithms, leading to more memory saving and faster convergence than the PTSP algorithm.

- We then propose two partly parallel decoder architectures based on the PTSP and PTSSP algorithms, respectively. Compared to the OSP decoder [9], our PTSP decoder has the same computational complexity, but requires less memory and simpler control. In addition, our approach achieves more significant saving in power consumption than that in [21] since less memory leads to less memory access and power consumption.

- Finally, We present proof-of-concept FPGA implementation results for our PTSP decoder and compare it with previously proposed decoders [13,26] to show the advantages of our PTSP decoder.

The rest of the paper is organized as follows. To make this paper self-contained, Section II briefly reviews the QC LDPC codes and the decoding algorithms for LDPC codes including the SP, turbo, shuffled, TSP and SSP decoding algorithms. In Section III, we propose the PTSP and PTSSP algorithms for partly parallel decoder architectures. Simulation results for the proposed algorithms are also presented. In Section IV, we propose partly parallel decoder architectures based on our PTSP and PTSSP algorithms. Section V presents the FPGA implementation results for our PTSP decoder. Finally, some concluding remarks are given in Section VII.
A. QC LDPC Codes

A subclass of \((j, k)\) regular QC LDPC codes can be represented by a parity check matrix of \(j\) block rows and \(k\) block columns,

\[
H = \begin{bmatrix}
I_{d_{0,0}} & I_{d_{0,1}} & \cdots & I_{d_{0,k-1}} \\
I_{d_{1,0}} & I_{d_{1,1}} & \cdots & I_{d_{1,k-1}} \\
\vdots & \vdots & \ddots & \vdots \\
I_{d_{j-1,0}} & I_{d_{j-1,1}} & \cdots & I_{d_{j-1,k-1}}
\end{bmatrix},
\]

(1)

where \(I_{d_{s,t}}\) denotes an \(m \times m\) identity matrix with all the rows cyclically shifted to the right by \(d_{s,t}\) positions \((0 \leq s \leq j-1, 0 \leq t \leq k-1)\). \(H\) in (1) defines an \((m, j, k)\) QC LDPC code with total number of parity check equations \(L = jm\), block length \(N = km\), and rate \(R \geq 1 - j/k\).

B. Decoding Algorithms of QC LDPC Codes

Given an LDPC code \(C\) represented by a parity check matrix \(H = \{H_{l,i}\}_{l=1,j=1}^{L,N}\), we briefly review the SP algorithm and its variants, i.e., turbo, shuffled, TSP and SSP decoding algorithms. Please refer to [11, 23, 25] for details.

1) SP Decoding Algorithm: The SP algorithm (see, e.g., [18]) is a multi-iteration procedure that iteratively exchanges the extrinsic log likelihood ratio (LLR) information between check and variable nodes until either the parity check equations are satisfied or the pre-set maximum iteration number is reached. Each iteration of the SP algorithm consists of two sequential steps: CNUs and VNUs, and which step starts first depends on initialization. For \(i = 1, 2, \cdots, N\), let \(q_{l,i}^{(n)}\) denote the message from variable node \(i\) to check node \(l \in Q_i = \{1 \leq l \leq L : H_{l,i} = 1\}\) during the \(n\)-th iteration. Similarly, for \(l = 1, 2, \cdots, L\), let \(r_{l,i}^{(n)}\) denote the message from check node \(l\) to variable node \(i \in R_l = \{1 \leq i \leq N : H_{l,i} = 1\}\) during the \(n\)-th iteration. Let \(x = [x_1 x_2 \cdots x_N]\) denote the codeword that is modulated using BPSK modulation and transmitted over an additive white Gaussian noise (AWGN) channel, and let \(y = [y_1 y_2 \cdots y_N]\) denote the received signals. The SP decoding algorithm can be formulated as:

- Initialization: For \(i = 1, 2, \cdots, N\), calculate the intrinsic LLR information obtained from channel \(\lambda_i = \ln \frac{p(y_i \mid x_i = 0)}{p(y_i \mid x_i = 1)}\), and initialize \(q_{l,i}^{(0)} = \lambda_i, \forall l \in Q_i\).

- Iteration \(n (n \geq 1)\):
  1) CNUs: For \(l = 1, \cdots, L\) and \(\forall i \in R_l\),

\[
q_{l,i}^{(n)} = \left( \prod_{i' \in R_l \setminus \{i\}} \text{sgn}(q_{l,i'}^{(n-1)}) \right) \cdot \Psi \left( \sum_{i' \in R_l \setminus \{i\}} \Psi \left( |q_{l,i'}^{(n-1)}| \right) \right),
\]

2) VNUs: For \(i = 1, 2, \cdots, N\) and \(\forall l \in Q_i\),
where \( \Psi(x) = \ln \frac{e^x + 1}{e^x - 1} \).

2) VNU: For \( i = 1, 2, \cdots, N \) and \( \forall l \in Q_i \),
\[
q_{i,l}^{(n)} = \lambda_i + \sum_{l' \in Q_i \setminus \{l\}} r_{l',i}^{(n)}.
\]
(3)

For \( i = 1, 2, \cdots, N \), first compute the a posteriori LLR \( \Lambda_i^{(n)} = \lambda_i + \sum_{l \in Q_i} r_{l,i}^{(n)} \), and then set the hard decision \( \hat{x}_{i}^{(n)} \) either to 0 if \( \Lambda_i^{(n)} > 0 \) or to 1 otherwise.

3) Parity check: If either \( H[\hat{x}_1^{(n)}, \hat{x}_2^{(n)}, \cdots, \hat{x}_N^{(n)}]^T = 0 \) or the maximum iteration number is reached, stop the decoding. Otherwise, start the next iteration \( (n = n + 1) \).

For an \((m, j, k)\) QC LDPC code, the computational complexity of the SP algorithm is roughly \( 2j \) additions per variable node for VNU, and roughly \( 2k \) additions and \( 2k \) evaluations of \( \Psi(\cdot) \) per check node for CNU (see, e.g., [27]). Partly parallel decoder architectures based on the SP algorithm require memories of \( jkm \) units, each to store one extrinsic LLR [8].

C. Variants of SP Decoding Algorithm

1) Turbo Decoding Algorithms: For an \((m, j, k)\) QC LDPC code, the turbo decoding algorithm [11] splits \( H \) into \( j \) submatrices row-wise, i.e., \( H = [H_1^T \ H_2^T \ \cdots \ H_j^T]^T \). Each \( H_g \) corresponds to one block row in (1) and defines one supercode \( C_g \), and the original LDPC code \( C = C_1 \cap C_2 \cdots \cap C_j \). Each submatrix \( H_g \) \( (g = 1, 2, \cdots, j) \) satisfies the constraint that there is at most one 1 in each column to save memory. Let \( C \) and \( V \) denote the sets of check and variable nodes respectively in the Tanner graph of \( C \). The breakup of \( H \) partitions \( C \) into \( j \) disjoint subsets: \( C = \tilde{C}_1 \cup \tilde{C}_2 \cup \cdots \cup \tilde{C}_j \), where \( \tilde{C}_g \) \( (g = 1, 2, \cdots, j) \) denotes the set of check nodes corresponding to \( H_g \). As shown in Fig. 1(a), each iteration of the SP algorithm consists of two serial steps: first passes the messages from \( V \) to all \( \tilde{C}_g \)’s \( (g = 1, 2, \cdots, j) \) concurrently and then passes the messages from all \( \tilde{C}_g \)’s \( (g = 1, 2, \cdots, j) \) to \( V \) concurrently. In the turbo decoding, each iteration becomes \( j \) serial sub-iterations as shown in Fig. 1(b): first passes the messages from \( V \) to \( \tilde{C}_1 \) and from \( \tilde{C}_1 \) to \( V \), then passes the messages from \( V \) to \( \tilde{C}_2 \) and from \( \tilde{C}_2 \) to \( V \), and so on. Since within one iteration, one supercode can use the updated and hence more reliable information from preceding supercodes, the turbo algorithm requires fewer (up to 50\%) iterations and achieves better error performance than the SP algorithm. Since the decoding of one supercode only needs information from the other supercodes and the new information can overwrite the old information from the succeeding supercode, this eliminates the need to save the extrinsic information of one supercode and leads to memory saving of \( km \) units, but the number of additions per variable node for VNU increases roughly from \( 2j \) to \( j^2 \).
2) **TSP Decoding Algorithm:** For an \((m, j, k)\) QC LDPC code, the TSP algorithm [25] splits \(H\) row-wise into two submatrices: 
\[
H = [H_1^T \ H_2^T]^T
\]
where \(H_1\) is the first block row and \(H_2\) consists of the rest of the block rows in \((1)\). \(H_1\) satisfies the constraint that there is exactly one 1 in each column to save memory. \(H_1\) and \(H_2\) define two supercodes \(C_1\) and \(C_2\) respectively, and \(C = C_1 \cap C_2\). The breakup of \(H\) partitions \(C\) into two disjoint sets \(C_1\) and \(C_2\). In order to save memory and exploit the turbo effects, each iteration of the TSP algorithm has two serial sub-iterations, as illustrated in Fig. 1(c): first passes the messages from \(V\) to \(C_1\), from \(C_1\) to \(V\) and from \(V\) to \(C_2\); then passes the messages from \(C_2\) to \(V\). Since the new information from \(C_1\) is immediately passed to \(C_2\), there is no need to save it and this leads to the memory saving of \(km\) units. The TSP algorithm leads to faster convergence and better error performance than the SP algorithm, while having the same computational complexity. Since the TSP algorithm only exploits the turbo effects of two supercodes, it converges slightly slower than the turbo algorithm, but can achieve a higher throughput.

3) **Shuffled Decoding Algorithm:** The shuffled or group shuffled decoding algorithm [23] is essentially a column-version of the turbo decoding algorithm, and \(H\) is split into \(k\) submatrices column-wise, i.e., 
\[
H = [H_1 \ H_2 \ \cdots \ H_k],
\]
each corresponding to one block column in \((1)\). Correspondingly, \(V\) is partitioned into \(G\) disjoint subsets: 
\[
V = \tilde{V}_1 \cup \tilde{V}_2 \cup \cdots \cup \tilde{V}_k,
\]
where \(\tilde{V}_g\) \((g = 1, 2, \cdots, k)\) denotes the set of variable
nodes corresponding to $\bar{H}_g$. Alternatively, we may view each iteration of the SP algorithm as: first passes the messages from $C$ to all $\bar{V}_g$'s ($g = 1, 2, \cdots, k$) concurrently and then passes the messages from all $\bar{V}_g$'s ($g = 1, 2, \cdots, k$) to $C$ concurrently as shown in Fig. 1(a). In the group shuffled decoding, each iteration becomes $k$ serial sub-iterations as shown in Fig. 1(d): first passes the messages from $C$ to $\bar{V}_1$ and from $\bar{V}_1$ to $C$, then passes the messages from $C$ to $\bar{V}_2$ and from $\bar{V}_2$ to $C$, and so on. The shuffled decoding algorithm also achieves better error performances and faster convergence than the SP algorithm. At the cost of increasing the number of additions and $\Psi(\cdot)$ evaluations roughly from $2k$ to $k^2$ per check node for check node updates (CNUs), the group shuffled decoding algorithm leads to memory saving of $jm$ units [23].

4) **SSP Decoding Algorithm**: The SSP algorithm [25] is essentially the column version of the TSP algorithm and $H$ is split column-wise into two submatrices: $H = [\bar{H}_1 \bar{H}_2]$, where $\bar{H}_1$ is the first block column and $\bar{H}_2$ consists of the rest of block columns. $\bar{H}_1$ has exactly one 1 in each row to save memory. Correspondingly, $V$ is split into two disjoint sets $V_1$ and $V_2$, which correspond to $\bar{H}_1$ and $\bar{H}_2$ respectively. As illustrated in Fig. 1(e), each iteration of our SSP algorithm also has two serial sub-iterations: first passes the messages from $C$ to $V_1$, from $V_1$ to $C$, and from $C$ to $V_2$; then passes the messages from $V_2$ to $C$. Compared with the SP algorithm, the SSP algorithm has the same computational complexity, but achieves better error performance, faster convergence, and memory saving of $jm$ units.

### III. **Decoding Algorithms for Partly Parallel Decoder Architectures**

In memory-banks based partly parallel decoders [7–9], due to memory access limitation, the CNUs for the $m$ rows in one block row and the VNU's for the $m$ columns in one block column are both carried out sequentially. In this case, having more serial sub-iterations in each iteration as in the turbo or shuffled algorithm leads to low decoding throughput. Hence, we focus on the TSP and SSP algorithms, and propose the following two algorithms: PTST and PTSSP algorithms to further improve throughput or memory saving of memory-banks based partly parallel decoder architectures.

#### A. **PTSP Decoding of QC LDPC Codes**

Similar to the non-overlapped SP decoding, the sequential processing of sub-iterations 1 and 2 in the TSP algorithm leads to low throughput and low HUE assuming memory-banks based partly parallel decoders. To improve the throughput and HUE, we propose a PTSP decoding algorithm, which essentially parallelizes the processing in sub-iterations 1 and 2 of the TSP algorithm.
We illustrate the difference between the TSP and PTSP algorithms during the $n$-th iteration of decoding in Figs. 2(a) and 2(b). For the TSP algorithm, we assume the memory banks are initialized with the channel intrinsic LLRs and hence sub-iteration 2 starts first. To simplify notation, we denote the set of messages from a set $A$ of check nodes $(A \subseteq C)$ to a set $B$ of variable nodes $(B \subseteq V)$ during the $n$-th iteration of the TSP decoding as $r_{A,B}^{(n)} = \{r_{l,i}^{(n)} : l \in A, i \in B\}$, and that from $B$ to $A$ as $q_{A,B}^{(n)} = \{q_{l,i}^{(n)} : l \in A, i \in B\}$. We denote those for the PTSP decoding as $\tilde{r}_{A,B}^{(n)}$ and $\tilde{q}_{A,B}^{(n)}$ respectively. At the $n$-th iteration of the TSP decoding, the decoding of $C_2$ uses the extrinsic information of the $(n-1)$-th iteration from $C_1$ and itself, i.e., $q_{C_2,V}^{(n-1)}$ is used when calculating $r_{C_2,V}^{(n)}$, and the decoding of $C_1$ uses the information of the $n$-th iteration from $C_2$, i.e., $r_{C_2,V}^{(n)}$ is used when calculating $q_{C_2,V}^{(n)}$. In the PTSP decoding, since the decoding of $C_1$ and $C_2$ is performed simultaneously, the most recent information can be exchanged during the decoding process. At the $n$-th iteration, at the beginning, the decoding of $C_2$ uses the extrinsic information of the $(n-1)$-th iteration from $C_1$ and itself, i.e., $\tilde{q}_{C_2,V}^{(n-1)}$ is used when calculating $\tilde{r}_{C_2,V}^{(n)}$, and the decoding of $C_1$ also uses the information of the $(n-1)$-th iteration from $C_2$, i.e., $\tilde{r}_{C_2,V}^{(n-1)}$ is used when calculating $q_{C_2,V}^{(n)}$. As the decoding proceeds, when part of the information of the $n$-th iteration from $C_2$ is available, the decoding of $C_1$ can use it, i.e., $\tilde{r}_{C_2,V}^{(n)}$ can be used when calculating $q_{C_2,V}^{(n)}$. The decoding of $C_2$ can then use the more reliable information from $C_1$, i.e., $\tilde{q}_{C_2,V}^{(n)}$ can be used when calculating $r_{C_2,V}^{(n)}$. Even more reliable information from $C_2$ is then produced and passed to $C_1$. Thus, in the PTSP algorithm, more and more reliable information is exchanged between the two supercodes as the decoding proceeds. However, each time only part of the information exchanged is from
Fig. 3. Split parity check matrix $H$ row-wise and column-wise for a $(3, 5)$-regular QC LDPC code

the $n$-th iteration. As a result, whether the PTSP algorithm leads to faster or slower convergence than the TSP algorithm depends on how much information is exchanged between the two supercodes.

Finally, same as the TSP algorithm, the PTSP algorithm also has the same computational complexity as the SP algorithm and leads to a memory saving of $km$ units.

B. PTSSP Decoding of QC LDPC Codes

To further improve the memory saving and convergence rate of the TSP algorithm, we can combine the TSP and SSP algorithms. Given a QC LDPC code, we split $H$ row-wise and column-wise simultaneously into four submatrices as shown in Fig. 3, where $H_1 = [H^{11} H^{12}]$ corresponds to the first block row; $H_2 = [H^{21} H^{22}]$ corresponds to the rest of the block rows; $H_1 = \begin{bmatrix} H^{11} \\ H^{21} \end{bmatrix}$ corresponds to the first block

Sub-iteration 1 Sub-iteration 2 Sub-iteration 3

Fig. 4. PTSSP decoding schedule
column and \( \bar{\mathbf{H}}_2 = \begin{bmatrix} \mathbf{H}^{12} \\ \mathbf{H}^{22} \end{bmatrix} \) corresponds to the rest of block columns. The breakup of \( \mathbf{H} \) partitions the set of check nodes as well as variable nodes in the Tanner graph of the given LDPC code into two disjoint sets: \( C_1 \) and \( C_2 \) corresponding to \( \mathbf{H}_1 \) and \( \mathbf{H}_2 \); and \( V_1 \) and \( V_2 \) corresponding to \( \mathbf{H}_1 \) and \( \mathbf{H}_2 \), respectively. \( \mathbf{H}_1 \) has only one 1 in each column and \( \bar{\mathbf{H}}_1 \) has only one 1 in each row. Between \( \mathbf{H}_1 \) and \( \mathbf{H}_2 \), we use the TSP decoding and this leads to the memory saving of the first block row, i.e., \( km \) units. Between \( \mathbf{H}_1 \) and \( \mathbf{H}_2 \), we use the SSP decoding, and this leads to the memory saving of the first block column, i.e., \( jm \) units.

As illustrated in Fig. 4, the PTSSP algorithm essentially changes the order of the messages passed between \( C_1, C_2 \) and \( V_1, V_2 \): first passes the messages from \( C_2 \) to \( V_1 \); then passes the messages from \( V \) to \( C_1, C_1 \) to \( V \) and \( V \) to \( C_2 \); finally passes the messages from \( C_2 \) to \( V_2 \). It is clear that sub-iterations 1 and 3 of the PTSSP decoding combined is the same as sub-iteration 1 of the TSP decoding, and sub-iteration 2 of the PTSSP decoding is the same as sub-iteration 2 of the TSP decoding. By TSP decoding, we can save the memory corresponding to \( \mathbf{H}_1 \). By SSP decoding, we can further save the memory corresponding to \( \mathbf{H}^{21} \) since the information in \( \mathbf{H}^{21} \) can always be obtained from or saved in \( \mathbf{H}^{22} \). Hence, only the information in \( \mathbf{H}^{22} \) needs to be saved. Furthermore, similar to the PTSP decoding, more and more reliable information can be exchanged during the decoding process. As illustrated in Fig. 4, instead of always using the information of the \((n-1)\)-th iteration, i.e., \( q_{C_2,V_2}^{(n-1)} \) and \( r_{C_2,V_2}^{(n-1)} \), the most recently updated information of the \( n \)-th iteration, i.e., \( q_{C_2,V_2}^{(n)} \) and \( r_{C_2,V_2}^{(n)} \), can be used when they become available during the decoding process.

C. Simulation Results

We consider a \((211,3,5)\) SFT code [15] with block length \( N = 1055 \) and rate \( R \approx 0.4 \). Figs. 5 shows the bit error rate (BER) performance comparisons of the SP (OSP), turbo, group shuffled, TSP, SSP, PTSP and PTSSP algorithms after 5, 15, and 50 iterations. CNU’s are carried out according to (2) and we use floating point representation. Setting the maximum number of iterations as 50, Tables I compares the average numbers of iterations, \( N_t \), all the algorithms take to converge. For the turbo decoding algorithm, we assume 3 supercodes. For the group shuffled decoding algorithm, the five block columns are split into three groups with two, two, and one block columns respectively. Note that the group shuffled decoding cannot save memory with the assumed splitting. The turbo, TSP and PTSP algorithms can all save \( 1/j \), i.e., 33\% of the total memory required by the SP algorithm. The SSP algorithm can save \( 1/k \), i.e., 20\% of the total memory required by the SP algorithm. The PTSSP algorithm can save \( 1/j + 1/k - 1/jk \), i.e., 46.7\% of the total memory required by the SP algorithm.
As can be seen clearly from Fig. 5 and Table I, our PTSP and PTSSP algorithms achieve better BER performances and faster convergence than the SP (OSP) algorithm. At 2.5 dB, our PTSP and PTSSP algorithms require 25.4% and 35.8% fewer iterations respectively than the SP algorithm. The faster convergence of the two algorithms potentially may lead to higher throughput. Furthermore, Tables I shows that our PTSP algorithm converges slightly slower than the TSP algorithm. Notice that our PTSSP algorithm combines the advantages of the TSP and SSP algorithms, and achieves faster convergence than the PTSP algorithm. We also performed simulations for the (421, 4, 7) and (281, 5, 8) SFT codes and similar observations can be made.

IV. DECODER ARCHITECTURES FOR QC LDPC CODES

A. PTSP Decoder Architecture

For an \((m, j, k)\) QC LDPC code, we propose a partly parallel decoder architecture based on our PTSP algorithm. Our PTSP decoder adopts an approach similar to that in [8] and uses one variable-check node function unit (VCNFU) and \((j - 1)\) check node function units (CNFUs). It also uses \(k\), \((j - 1)k\), and \(k\) memory banks to store the channel intrinsic LLRs, extrinsic LLRs, and hard decisions respectively. Fig. 6 shows the proposed decoder architecture for a \((3, 5)\) regular QC LDPC code, where the memory
one bit whereas the wordlength for the memory banks for intrinsic and extrinsic messages depends on
implementation. Each unit of the channel intrinsic LLRs, extrinsic LLRs, and hard decisions respectively.

<table>
<thead>
<tr>
<th>SNR</th>
<th>SP</th>
<th>Turbo</th>
<th>TSP</th>
<th>PTSP</th>
<th>Shuffled</th>
<th>SSP</th>
<th>PTSSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>47.2</td>
<td>45.7</td>
<td>46.2</td>
<td>46.3</td>
<td>46.1</td>
<td>46.7</td>
<td>45.8</td>
</tr>
<tr>
<td>1</td>
<td>34.9</td>
<td>29.5</td>
<td>31.4</td>
<td>31.6</td>
<td>30.9</td>
<td>33.2</td>
<td>30.1</td>
</tr>
<tr>
<td>1.5</td>
<td>17.9</td>
<td>11.2</td>
<td>13.5</td>
<td>13.9</td>
<td>13.1</td>
<td>15.7</td>
<td>12.2</td>
</tr>
<tr>
<td>2</td>
<td>9.5</td>
<td>5.1</td>
<td>6.4</td>
<td>6.9</td>
<td>6.3</td>
<td>8.0</td>
<td>5.9</td>
</tr>
<tr>
<td>2.5</td>
<td>6.7</td>
<td>3.6</td>
<td>4.5</td>
<td>5.0</td>
<td>4.6</td>
<td>5.7</td>
<td>4.3</td>
</tr>
</tbody>
</table>

Fig. 6. PTSP decoder architecture for a (3, 5) regular QC LDPC code

banks labeled $\Lambda_t$ ($0 \leq t \leq k - 1$), $m_{s,t}$ ($1 \leq s \leq j - 1$, $0 \leq t \leq k - 1$), and $\hat{x}_t$ ($0 \leq t \leq k - 1$) save
the channel intrinsic LLRs, extrinsic LLRs, and hard decisions respectively.

Each memory bank consists of $m$ units — labeled with addresses $\{0, 1, 2, \cdots, m - 1\}$. The three
types of memory banks differ. The $(j - 1)k$ memory banks for extrinsic LLRs are dual-port while the $k$
memory banks for intrinsic LLRs are single-port. Whether the hard decision memory banks are single-port
or dual-port depends on implementation. Each unit of the $k$ memory banks for hard decisions has exactly
one bit whereas the wordlength for the memory banks for intrinsic and extrinsic messages depends on
implementation. Modulo-$m$ counters are used to generate the memory addresses. As mentioned before, there is no need to save the information from $C_1$. Hence the $(j-1)k$ dual-port memory banks correspond to the block rows in $H_2$. The $m$ units in each $m_{s,t}$ correspond to the $m$ 1’s in $I_{d_s,t}$ $(1 \leq s \leq j-1$, $0 \leq t \leq k-1$). Each 1 in $I_{d_s,t}$ has a row number and a column number, and hence each memory unit in the memory bank has a row (memory) address and column address, between which the difference is $d_{s,t}$.

The CNFUs can be implemented in the same way as in [7] or [8], and CNFU-$s$ $(0 \leq s \leq j-1)$ performs CNU’s for the $s$-th block row, one row at a time. The VCNFU essentially functions as one CNFU and $k$ variable node function units (VNFUs), each of which consists of the components inside the dashed rectangular box as shown in Fig. 7. VNFU-$t$ $(0 \leq t \leq k-1)$ performs the VNU’s for the $t$-th block column, one column at a time. Each VNFU has two parts: P1-VNFU inside the dotted rectangular box to the left of CNFU-0, and P2-VNFU inside the dotted rectangular box to the right of CNFU-0. The VCNFU has four components: $k$ P1-VNFUs before the CNFU performing the VNU’s for $C_1$, CNFU-0 performing the CNU’s for $C_1$, $k$ P2-VNFUs after the CNFU performing the VNU’s for $C_2$ and making hard decisions. The VCNFU and the $(j-1)$ CNFUs simultaneously implement sub-iterations 1 and 2, respectively.

Let $c = \{c_0, c_1, \ldots, c_{j-1}\}$ denote the row addresses the $j$ CNFUs operate on in their respective block rows. Let $v = \{v_0, \ldots, v_{k-1}\}$ denote the column addresses the VNFUs operate on in their respective
block column. In order to perform PTSP decoding, it requires that $v_t = c_0 + d_{0,t} \pmod{m}$ ($0 \leq t \leq k-1$).

We can choose $c_s$ ($0 \leq s \leq j - 1$) to maximize the information exchange between the two supercodes. However, our simulation results show that different addressing schemes have only marginal effects on error performances and convergence rate. Hence, we simply choose $c_s = 0$ ($0 \leq s \leq j - 1$). Fig. 8 shows the starting addresses for a $(3, 5)$ regular QC LDPC code, where each box, also labeled $m_{s,t}$ ($0 \leq s \leq j - 1$, $0 \leq t \leq k - 1$), corresponds to one $I_{d_{s,t}}$ and the diagonal line inside each box denotes the position of 1’s. The dots denote the data CNFU-1 and CNFU-2 are updating, and the squares denote the data the VCNFU is updating.

Since the VCNFU performs VNUs first and CNFU-1 and CNFU-2 perform CNU first and they operate on the memory banks simultaneously, the memory banks need to be initialized accordingly. That is, initialize the memory units where CNFUs operate on first with the channel intrinsic LLRs and the memory units where the VCNFU operates on first with 0’s.

PTSP Decoding Schedule:

1) Initialization: Set $c_s = 0$ ($0 \leq s \leq j - 1$) and $v_t = c_0 + d_{0,t}$ for ($0 \leq t \leq k - 1$). For each memory bank $m_{s,t}$ ($1 \leq s \leq j - 1$, $0 \leq t \leq k - 1$), initialize the shaded part shown in Fig. 6 — the memory units with row addresses from 0 to $v_t - d_{s,t} - 1 \pmod{m}$ — with the channel intrinsic LLRs, and initialize the unshaded part with 0’s.

2) Repeat for $m$ times:
In this case, the memories are all initialized to channel intrinsic LLRs and only single-port memory banks are needed since sub-iterations 1 and 2 are to be carried out sequentially.

Fig. 9. PTSSP decoder architecture for a (3, 5) regular QC LDPC code

i) VCNFU: VNFU-\(t\) \((0 \leq t \leq k - 1)\) simultaneously reads one data from each of the memory units with column address \(v_t\) in the \(j - 1\) memory banks in the \(t\)-th block column, denoted as \(m_{s,t}(v_t)\) \((1 \leq s \leq j - 1)\) in Fig. 7, performs VNUs for \(C_1\), and obtains the \(k\) data corresponding to row address \(c_0\) in the 0-th block row, denoted as \(m_{0,t}(c_0)\) \((0 \leq t \leq k - 1)\) in Fig. 7. CNFU-0 takes as input the \(k\) data, performs the CNUs. Then the VNUs for \(C_2\) are performed and the updated messages are written back to the same memory units.

ii) CNFU-\(s\) \((1 \leq s \leq j - 1)\) simultaneously reads one data from each of the \(c_s\)-th memory units of the \(k\) memory banks in the \(s\)-th block row, performs CNUs, and writes the updated messages back to the same memory units.

iii) \(c_s = c_s + 1 \pmod{m}\) \((0 \leq s \leq j - 1)\), \(v_t = v_t + 1 \pmod{m}\) \((0 \leq t \leq k - 1)\).

Note that steps 2(i) and 2(ii) are carried out in parallel, and step 2 constitutes one iteration of the PTSP decoding. Also note that the same decoder architecture can also be used to implement the TSP algorithm.
Fig. 10. CVCNFU for a (3, 5) regular QC LDPC code

B. PTSSP Decoder for QC LDPC Codes

For a \((j,k)\) regular QC LDPC code, our PTSSP decoder uses one check-variable-check node function unit (CVCNFU) and \(k, (j-1)(k-1)\), and \(k\) memory banks to store the channel intrinsic LLRs, extrinsic LLRs, and hard decisions respectively. Fig. 9 shows the proposed decoder architecture for a \((3, 5)\) regular QC LDPC code, where the memory banks labeled \(\Lambda_t\) \((0 \leq t \leq k-1)\), \(m_{s,t}\) \((1 \leq s \leq j-1, 1 \leq t \leq k-1)\), and \(\hat{x}_t\) \((0 \leq t \leq k-1)\) save the intrinsic LLRs, extrinsic LLRs, and hard decisions respectively. As mentioned before, there is no need to store the information corresponding to the first block row and block column, thus the \((j-1)(k-1)\) memory banks for extrinsic LLRs correspond to \(H^{22}\).

The CVCNFU essentially functions as two CNFUs and one VCNFU, as shown in Fig. 10, where CNFU-s \((1 \leq s \leq j-1)\) performs the CNU's for the \(s\)-th block column, one row at a time. The CVCNFU has three components: a network of adders and \(\Psi(\cdot)\) before the VCNFU implementing sub-iteration 1, a VCNFU implementing sub-iteration 2, a network of adders (subtractors) and \(\Psi(\cdot)\) after the VCNFU implementing sub-iteration 3. The two networks of adders and \(\Psi(\cdot)\) combined together form 2 CNFUs. It is clear that the PTSSP decoder has exactly the same computational complexity as the SP algorithm.

To perform PTSSP decoding, it requires that \(c_s = c_0 + d_{0,0} - d_{s,0} \pmod{m} \) \((1 \leq s \leq j-1)\) and \(v_t = c_0 + d_{0,t} \pmod{m} \) \((0 \leq t \leq k-1)\). Set \(c_0 = 0\), Fig. 11 shows the starting addresses for a \((3, 5)\) regular QC LDPC code. The dots denote the data CNFU-1 and CNFU-2 are working on, and the squares
denote the data the VCNFU are working on. In $m_{1,0}$ and $m_{2,0}$, dots and squares coincide.

Similar to the PTSP decoder, since the VCNFU performs VNUs first and CNFU-1 and CNFU-2 perform CNU's first and they operate on the memory banks simultaneously, the memory banks need to be initialized in a similar way as in the PTSP decoder.

PTSSP Decoding Schedule:

1) Initialization: Initialize $c_0 = 0$, $c_s = c_0 + d_{0,0} - d_{s,0} \pmod{m}$ ($1 \leq s \leq j - 1$) and $v_t = c_0 + d_{0,t} \pmod{m}$ ($0 \leq t \leq k - 1$). For each memory bank $m_{s,t}$ ($1 \leq s \leq j - 1$, $1 \leq t \leq k - 1$), initialize the shaded part shown in Fig. 9 — the memory units with row addresses from $c_s$ to $v_t - d_{s,t} - 1 \pmod{m}$ — with the intrinsic LLRs, and initialize the unshaded part with 0’s.

2) Repeat for $m$ times:
   i) CNFU-s ($1 \leq s \leq j - 1$) simultaneously reads one data from each of the $c_s$-th memory units of the $k - 1$ memory banks in the $s$-th block row, performs CNU’s for $H^{21}$ and obtains the $j - 1$ data corresponding to address $v_0$ in the first block column, denoted as $m_{s,0}(v_0)$ in Fig. 10.
   ii) VCNFU: VCNFU simultaneously reads one data from each of the memory units with column address $v_t$ $1 \leq t \leq k - 1$ in the $j - 1$ memory banks in the $t$-th block column, denoted as $m_{s,t}(v_t)$ ($1 \leq s \leq j - 1$, $1 \leq t \leq k - 1$) in Fig. 10, together with the output from step 2(i),
Fig. 12. BER performance for the (256, 3, 6) QC LDPC code

\( m_{s,0}(v_0) \) (1 ≤ \( s \) ≤ \( j - 1 \)), performs VNU’s for \( C_1 \), CNU’s for \( C_1 \), and VNU’s for \( C_2 \). The output data corresponding to \( m_{s,t}(v_t) \) (1 ≤ \( s \) ≤ \( j - 1 \), 1 ≤ \( t \) ≤ \( k - 1 \)) are written back to the same memory units, and the output corresponding to \( m_{s,0}(v_0) \) (1 ≤ \( s \) ≤ \( j - 1 \)) are fed to step 2(iii).

iii) CNFU-\( s \) (1 ≤ \( s \) ≤ \( j - 1 \)) takes as input \( m_{s,0}(v_0) \) and performs the CNU’s for the \( H_{22} \). The updated data are written back to the same memory units.

iv) \( c_s = c_s + 1 \pmod m \) (0 ≤ \( s \) ≤ \( j - 1 \)), \( v_t = v_t + 1 \pmod m \) (0 ≤ \( t \) ≤ \( k - 1 \)).

Step 2 constitutes one iteration of the PTSSP decoding. Since steps 2(i), 2(ii) and 2(iii) are performed sequentially, each iteration of the PTSSP decoding takes more time than the PTSP decoding. This can be improved by inserting registers and pipelining the processing in the three steps. Totally, three registers, each having \((j - 1)k\) units, are needed for the pipelined processing. After pipelining, the number of clock cycles required for each iteration of the PTSSP decoding is dominated by the processing in VCNFU, which is the same as the PTSP decoder. Since each iteration of the PTSSP decoding takes approximately the same time as the PTSP decoder, but achieves faster convergence, higher throughput can be achieved. Also, the PTSSP decoder can save more memory than the PTSP decoder, and hence leads to more significant power savings.
V. IMPLEMENTATION OF PTSP DECODER

A. Quantization Scheme

The $\Psi(\cdot)$ function in the SP algorithm is a non-linear function and is susceptible to quantization noise. For finite precision implementations, a low-complexity approximation of the SP algorithm, min-sum (MS) algorithm is often used. In the MS algorithm, VNU’s are the same as in the SP algorithm, and CNU’s in (2) become

$$r_{l,i}^{(n)} = \prod_{i' \in R_l \setminus \{i\}} \text{sgn}(q_{l,i'}^{(n-1)}) \cdot \min_{i' \in R_l \setminus \{i\}} |q_{l,i'}^{(n-1)}|. \quad (4)$$

The MS algorithm can be directly applied to all the variants of the SP algorithm.

In order to compare with previously proposed decoder architectures, we consider the same $(256, 3, 6)$ QC LDPC code as in [13]. We also use MS approximation for the PTSP algorithm, referred to as PTSP-MS algorithm, and use 8-bit uniform quantization with 1 sign bit, 4 integer bits, and 3 bit fractional bits. Figs. 12 and 13 compare the simulated BER performance and the average numbers of iterations to converge between the MS and PTSP-MS algorithms using both floating point and fixed point. It is clear that the PTSP-MS algorithm achieves better BER performance and faster convergence than the MS algorithm for both the floating point and fixed point. For both algorithms, the BER performances using fixed point outperform those using floating point. Note that similar observations have been made in [27, 28]. Also, the number of iterations using fixed point are almost the same as those using floating point.
B. Pipeline Scheme

As shown above, the PTSP-MS algorithm leads to faster convergence and hence less iteration number than the MS algorithm. However, whether the faster convergence can lead to higher throughput or not depends on the critical path in the implementation of the PTSP decoder. In the following, we first show that the critical path of the PTSP decoder is determined by the VCNFU and then propose a pipeline scheme to reduce it to a similar value as that of the SP (OSP) decoder [26].

In FPGA and RAM compilers for ASIC with deep sub-micron technology, only synchronous static memory is available [29]. A drawback of typical synchronous memory is that one or two waiting clock cycles are needed in every read operation. We assume exactly one waiting clock cycle is added for each read operation, but our approach can be easily extended to synchronous memory that requires two clock cycles. To achieve high throughput, pipelined memory access is usually employed. In the PTSP decoder shown in Fig. 6, the VCNFU and CNFUs work concurrently and accessing the memory banks concurrently through different port. The CNFUs are the same as those in the SP (OSP) decoder and hence we adopt the same pipeline scheme as that in [26], shown in Fig. 14(a). Each step of CNFUs updates takes 4 clock cycles and 2 clock cycles are used for the calculations required in CNFUs. After pipelining, we can achieve the maximum of 2 clock cycles per step throughput, where each clock cycle has either a read or a write operation. The VCNFU as shown in Fig. 7 essentially combines the function of one CNFU and one VNFU serially, hence the time required to perform one step of VCNFU updates will be longer than that for either CNFU or VNFU. If we use the same pipeline scheme and fit the VCNFU into 4 clock cycles, we need to perform the calculations required in one CNFU plus one VNFU in 2
clock cycles. Compared to the case for CNFUs, more calculations are required in each clock cycle. The critical path of the PTSP decoder is thus determined by the longer path in the VCNFU update. To reduce the critical path while maintaining the 2 clock cycles per step throughput, we use a different pipeline scheme for VCNFU as shown in Fig. 14(b).

VI. IMPLEMENTATION RESULTS

We have implemented our PTSP decoder using a Xilinx Vertex II xc2v6000-5ff1152 FPGA, which has 33792 slices and 144 block RAMs (each block RAM is 18 Kbits). The synthesis and implementation are carried out using Xilinx ISE Version 8.1. The results are compared with those from [13, 26], and in Table II, we specify the numbers of occupied slices, flip flops, 4-input LUTs, and block RAMs as well as critical path delays, average iteration numbers, and estimated decoding throughput. The first two rows of Table II, reproduced from [13, Table I], itemize implementation details of the two decoder architectures in [13]. These two architectures, referred to as the MS and HEMS architectures respectively henceforth, are based on the MS algorithm and the high-efficiency min-sum algorithm proposed in [13] respectively.

The HEMS algorithm in [13], a variant of the MS algorithm, needs more VNUs and more memory banks to achieve faster convergence. The third row, reproduced from [?, Table I], itemize the implementation details of the OSP-MS architecture. The OSP-MS decoder architecture doubles the throughput of non-overlapped decoder by scheduling the CNUs(VNUs) optimally to minimize the waiting time [?]. We note that the four implementation in Table II use the same FPGA, the same quantization scheme, the same check node update algorithm (Min-Sum). The estimated decoding throughputs for all four decoder architectures are computed based on the same assumptions made in [13]: The input interface can get intrinsic information by groups of $k$ nodes from different block columns, and the clock frequency is the inverse of the critical path delay. The decoding throughput of all four decoder architectures can be calculated by the same formula as in [26] given by

$$\text{Throughput} = \frac{k \times m \times f}{m + 2 \times m \times (N_t + 1)},$$

where $f$ is the clock frequency and $l$ is the average number of iterations. The difference between Eq. (5) and [13, (8)] is due to the different scheduling schemes.

Comparing all four decoder architectures itemized in Table II, it is clear that our PTSP decoder architecture requires less memory and achieves a higher throughput. The number of block RAM is reduced to 67% of the OSP architecture and 24% of the MS and HEMS architecture. Although our PTSP decoder architecture requires more combinational logic than the other architectures, the number
TABLE II
IMPLEMENATION RESULTS OF THE (256, 3, 6) QC LDPC CODE ON XILINX VIRTEXII xc2v6000-5ff1152 FPGA

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Slice</th>
<th>F/F</th>
<th>4-input LUT</th>
<th>Block RAM</th>
<th>Critical Path Delay [ns]</th>
<th>$N_t$</th>
<th>Throughput (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MS [13]</td>
<td>1144</td>
<td>1089</td>
<td>1705</td>
<td>102</td>
<td>9.894</td>
<td>3.4</td>
<td>23</td>
</tr>
<tr>
<td>OSP-MS [26]</td>
<td>1167</td>
<td>664</td>
<td>2066</td>
<td>36</td>
<td>10.893</td>
<td>3.0</td>
<td>61.2</td>
</tr>
<tr>
<td>PTSP-MS</td>
<td>1463</td>
<td>819</td>
<td>2534</td>
<td>24</td>
<td>11.933</td>
<td>2.3</td>
<td>66.1</td>
</tr>
</tbody>
</table>

of flip-flops is less than those of the MS and HEMS architectures. Memory efficiency of our decoder architecture will enable implementation on small FPGAs and simplify the placement and routing in ASIC implementation. As shown in Table II, the critical path delay of our PTSP architecture is greater than those of the other architectures, but it still provides the highest throughput. Nearly triple the throughput of the MS architecture, the throughput of our PTSP decoder architecture is 74% higher than that of the HEMS architecture despite fewer iterations required by the latter.

In comparison to the OSP decoder architecture proposed in [8, 9], it is clear that our PTSP decoder has exactly the same computational complexity, but requires less memory, thus less memory access and less power consumption. Also, our PTSP decoder requires simpler control than the OSP decoder since in the OSP decoder [9], in order to satisfy the data dependency constraints, the starting addresses need to be shifted at each iteration, and the hardware needs to be turned on and off regularly when $w^* \geq \lfloor m/2 \rfloor + 1$ [9]. Finally, due to the faster convergence of the PTSP algorithm as shown in Section III-C, our PTSP decoder may lead to higher throughput than the OSP decoder.

VII. CONCLUSIONS

REFERENCES


