On the memory complexity of APP decoders for LDPC codes

Velimir Ilić[1], Elsa Dupraz[2], David Declercq[3], Bane Vasić[4]
[1]Mathematical Institute SANU, Belgrade, Serbia
[2,3]ETIS ENSEA/Univ. of Cergy-Pontoise/CNRS, Cergy-Pontoise, France
[4]Department of ECE, University of Arizona, Tucson
[1]velimir.ilic@gmail.com, [2]dupraz@ensea.fr,
[3]declercq@ensea.fr, [4]vasic@ece.arizona.edu

***Abstract*—In this paper we propose two memory efficient a posteriori probability (APP) decoders for the decoding of low-density parity-check (LDPC) codes. The proposed decoders require memory that is linear in the number of nodes in the Tanner graph of the code. This is a significant saving compared to the existing APP decoder, which requires memory that is proportional to the number of edges. We derive the exact expressions for the memory and computational complexity of the decoders in terms of the number of real operations and basic memory units required for the decoding.**

I. Introduction

Belief propagation (BP) is an iterative message-passing algorithm for decoding Low Density Parity Check (LDPC) codes, widely used in many systems [1]. Despite its good error correction performance and capability of approaching the Shannon limit, BP suffers from large memory requirements for message processing and storage, proportional to the number of edges in the Tanner graph of the code [2]. Such large memory requirements, coupled with additional hardware resources needed for the message updating, make the BP less attractive in practical applications.

A posteriori probability (APP) decoder [3] is a sub-optimal alternative to BP, in which the variable node processing is simplified by allowing variables to send messages in an intrinsic manner. As a result, in the APP decoder, a message from a variable node corresponds to a posteriori value used to estimate that variable. Although this property admits a memory efficient implementation, the advantage has not been recognized in the original paper [3], where the APP decoder is represented in its parallel form.

In this paper we propose two memory-efficient APP decoders that require memory proportional to the number of nodes in the Tanner graph of the LDPC code, rather than to the number of edges, as in the parallel APP decoder proposed in [3]. The proposed algorithms use the advantages of different types of message passing scheduling, previously developed for the BP algorithm. The first algorithm uses the flooding schedule over check nodes proposed in [4] for BP decoding. It operates in a semi-parallel way, by processing all the check nodes in a parallel manner and variable nodes in a serial one. The second one is based on the shuffled schedule proposed in [5] and operates in fully serial manner. The computational and memory complexity analyses of the original parallel APP decoder, as well as of the two proposed decoders, are derived.

The paper is structured as follows. In section II we present the basic notions on LDPC codes and review the parallel APP decoder. The memory efficient APP decoders are derived in the section III and the complexity analysis is derived.

II. APP decoding of LDPC codes

In this section we introduce basic definitions of LDPC codes theory and present the original parallel APP decoder proposed in [3] .

A. LDPC codes

A regular LDPC code is a linear block code defined by a generator matrix G of size (K, N) and by a sparse parity-check matrix H of size (M, N), with $N = K + M$. The codeword $\boldsymbol{x} = (x_1, x_2, \ldots, x_N) \in \{0,1\}^N$ is constructed from the information sequence $\boldsymbol{u} = (u_1, u_2, \ldots, u_K) \in \{0,1\}^K$ as $\boldsymbol{x} = G\boldsymbol{u}$. The codeword $\boldsymbol{x}$ satisfies $H\boldsymbol{x}^T = 0$, where $\boldsymbol{x}^T$ denotes the transposed (column) vector. The rate of the code is denoted $R = K/N$. The Tanner graph [2] of an LDPC code is a bipartite graph whose adjacency matrix is the parity-check matrix of the code H. It contains two types of nodes: a set of variable-nodes $\mathcal{N} = \{v_1, v_2, \ldots, v_N\}$, corresponding to the N columns of H, and a set of check-nodes $\mathcal{M} = \{c_1, c_2, \ldots, c_M\}$, corresponding to the M rows of H. A variable-node v_n and a check-node c_m are connected by an edge if and only

*This work is supported by Ministry of Science and Technological Development, Republic of Serbia, Grant III044006, by the NSF under grants CCF-1314147 and CCF-0963726 and the Seventh Framework Programme of the European Union, under Grant Agreement number 309129 (i-RISC project).

if the corresponding entry of H is non-zero. The set of indices of check-nodes connected to the variable-node v_n is denoted with $\mathcal{H}(v_n)$ and the set of indices of variable-nodes connected to the check-node c_m is denoted with $\mathcal{H}(c_m)$. In the case of regular LDPC codes cardinalities of $\mathcal{H}(v_n)$ and $\mathcal{H}(c_m)$ are the same for all v_n and c_m, and denoted with d_v and d_c, respectively. In the case of regular codes, the rate of the code can be calculated as $R = 1 - \frac{d_v}{d_c}$.

Let $\boldsymbol{y} = (y_1, y_2, \ldots, y_N)$ be the received sequence as defined in [6]. The channel is defined by the probabilistic model

$$p(\boldsymbol{x}, \boldsymbol{y}) \propto \prod_{n=1}^{N} \Pr(y_n|x_n) \prod_{m=1}^{M} \mathbb{1}(\sum_{n \in \mathcal{N}(m)} x_n)$$

where $\Pr(y|x)$ is the channel likelihood, $\mathbb{1}$ is the indicator function and $\sum_{n \in \mathcal{N}(m)} x_n$ are modulo 2 sums determined by the parity check matrix H.

.... Code rate is defined as $R = K/N$. ......

B. Parallel APP decoder

The goal of the decoding is to compute the a posteriori probability $\Pr(x_n|\boldsymbol{y})$, which is used for the decision making on bit values. APP decoder originally proposed in [3] computes the a posteriori probability in an iterative message passing manner, by processing all the check and variable nodes in parallel. In one half-iteration, messages from check nodes are computed according to previously computed (or initialized) values in variable nodes. After that, all the variable nodes take incoming message at same time and update its values, which completes one iteration. Parallel APP decoder operates as follows.

Initialization: Variable-nodes are initialized to a priori values $(\gamma_1, \gamma_2, \ldots, \gamma_N)$ from the received sequence $(y_1, y_2, \ldots, y_N)$ as

$$\tilde{\gamma}_n^{(0)} = \gamma_n = \frac{p(y_n \mid x_n = 0)}{p(y_n \mid x_n = 1)}. \qquad (1)$$

Iterative processing:

1) Check-node processing: consists in computing the check-to-variable messages $\mu_{m \to n}^{(k)}$, for all check-nodes m and their neighbor variable-nodes v_n;

$$\mu_{m \to n}^{(k)} = \boxplus_{k \in \mathcal{N}(m) \backslash n} \tilde{\gamma}_k^{(k-1)}, \qquad (2)$$

where $\boxplus_{i \in \mathcal{N}(m) \backslash n}$ stands for the summation over the set $\mathcal{N}(m) \backslash n$ induced by the box-sum operation defined as

$$x \boxplus y = \log \frac{1 + e^x e^y}{e^x + e^y} \qquad (3)$$

Algorithm 1: PARALLEL APP DECODER

```
Input: $\boldsymbol{y} = (y_1, \ldots, y_N) \in \mathcal{Y}^N$        ▷ received word
Output: $\hat{\boldsymbol{x}} = (\hat{x}_1, \ldots, \hat{x}_N) \in \{0,1\}^N$        ▷ estimated
    codeword

    Initialization:
 1: for each $\{v_n\}_{n=1,\ldots,N}$ do $\gamma_n = \log \frac{Pr(x_n = 0|y_n)}{Pr(x_n = 1|y_n)}$;
 2: for each $\{v_n\}_{n=1,\ldots,N}$ do $\hat{\gamma}_n = \gamma_n$;

    Iterative processing loop
 3: for each $\{c_m\}_{m=1,\ldots,M}$ do $\Psi_m = 0$
 4:     for each $v_n \in \mathcal{H}(c_m)$ do $\Psi_m = \Psi_m \boxplus \hat{\gamma}_n$
 5: for each $\{c_m\}_{m=1,\ldots,M}$ do
 6:     for each $v_n \in \mathcal{H}(c_m)$ do $\mu_{k \to m} = \Psi_m \boxminus \hat{\gamma}_n$

 7: for each $\{v_n\}_{n=1,\ldots,N}$ do $\hat{\gamma}_n = \gamma_n$
 8:     for each $c_m \in \mathcal{H}(v_n)$ do $\hat{\gamma}_n = \hat{\gamma}_n + \mu_{k \to m}^{(n)}$

 9: for each $\{v_n\}_{n=1,\ldots,N}$ do $\hat{x}_n = (1 - \text{sign}(\tilde{\gamma}_n))/2$
10: if $\hat{\boldsymbol{x}}$ is codeword then exit the iteration loop
    End iterative processing loop
```

2) A posteriori information update: consists in computing the a posteriori messages $\tilde{\gamma}_n^{(k)}$, for all variable-nodes v_n,

$$\tilde{\gamma}_n^{(k)} = \gamma_n + \sum_{m \in \mathcal{M}(n)} \mu_{m \to n}^{(k)}. \qquad (4)$$

3) Hard decision: Estimated values of sent bits, $\hat{\boldsymbol{x}} = (\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_N)$, according to the rule: $\tilde{\gamma}_n^{(k)} > 0$ then $x_n^{(k)} = 0$, otherwise $x_n^{(k)} = 1$. The decoder stops when either $\hat{\boldsymbol{x}}$ is a codeword or a maximum number of decoding iterations is reached.

Check to variable messages requires the computation of all partial sums $\boxplus_{k \in \mathcal{N}(m) \backslash n} \tilde{\gamma}_k^{(k-1)}$, which can efficiently be computed using the inverse operation for $\boxplus$ called minus-box operator:

$$x \boxminus y = \log \frac{1 - e^x e^y}{e^x - e^y} \qquad (5)$$

It is easy to check that $x \boxplus y \boxminus y = x$. Using the $\boxminus$ operator, the sum

$$\Psi_m^{(k)} = \boxplus_{k \in \mathcal{N}(m)} \tilde{\gamma}_k^{(k-1)} \qquad (6)$$

can be computed once per iteration and node, and all the messages can be computed for all $n \in \mathcal{N}(m)$ as

$$\mu_{m \to n}^{(k)} = \Psi_m^{(k)} \boxminus \hat{\gamma}_n^{(k-1)}. \qquad (7)$$

III. Memory efficient APP decoding

In this section we consider implementation aspects of the APP decoder. First, we present parallel APP decoder proposed in [3]. After that we propose two alternative memory efficient variants, semi-parallel and serial APP decoders. We derive the exact expressions for the memory and computational complexity of the decoders in terms of the number of real operations and basic memory units required for the decoding.

A. Parallel APP decoder

In the parallel APP decoder proposed in [3], each edge and each node of the Tanner graph uses its own processor to perform computations. The pseudo-code for the parallel decoder is given in **Algorithm 1**.

The computation of check sums Ψ_m is done in lines 3 and 4, with M processor for $\boxplus$ operation which works in parallel. In accordance, "for each loop" in the line 3 can be performed at once and each of M processors performe $\boxplus$ addition in the line 4 for d_v times.The check-to-variable messages are computed in the lines 5 and 6, where the "for each loops" run over the all edges in the Tanner graph. All of the d_vN processors associated to the edges run in parallel each of them performing $\boxminus$ operation for once. The computation of the a posteriori values $\hat{\gamma}_n$ is done in the lines 7 and 8. Similarly as for the check sums, we need N processors which work in parallel and performe real additions for d_c times. The discussion is summarized in the first row of Table I.

Let us discuss now the memory requirements of the parallel implementation. During the computations, we need M registers for storing the check-sums Ψ_m, d_vN registers for storing check-to variable messages $\mu_{m\to n}$ and N registers for the estimated a posteriori values $\hat{\gamma}_n$. This results in the total number of $(d_v+1)N+M$ registers, as given in Table II.

In accordance, parallel APP decoder requires a memory which is proportional to the number of edges in the Tanner graph. In the following sections we propose two decoders which require memory that is proportional to the number of the nodes.

B. Semi-parallel APP algorithm

A semi-parallel APP decoder uses the flooding schedule over the check nodes, proposed in [4] for BP decoding. In the semi-parallel version the first half-iteration is done in parallel, while the second one is processed in a serial manner. The semi-parallel APP decoder is presented in **Algorithm 2**.

In the same manner as in parallel decoder, each check node uses its own processor to perform computations. In accordance, "for each loops" in the line 3 and 4 are performed using M processors which runs in parallel and each processor computes $\boxplus$ addition for d_v times.

Algorithm 2: Semi-parallel APP decoder

```
Input: $\boldsymbol{y} = (y_1, \dots, y_N) \in \mathcal{Y}^N$    ▷ received word
Output: $\hat{\boldsymbol{x}} = (\hat{x}_1, \dots, \hat{x}_N) \in \{0,1\}^N$    ▷ estimated
    codeword

    Initialization:
 1: for each $\{v_n\}_{n=1,\dots,N}$ do $\gamma_n = \log \frac{Pr(x_n = 0|y_n)}{Pr(x_n = 1|y_n)}$;
 2: for each $\{v_n\}_{n=1,\dots,N}$ do $\hat{\gamma}_n^{\text{old}} = \gamma_n$;

    Iterative processing loop
 3: for each $\{c_m\}_{m=1,\dots,M}$ do $\Psi_m = 0$
 4:     for each $v_n \in \mathcal{H}(c_m)$ do $\Psi_m = \Psi_m \boxplus \hat{\gamma}_n^{\text{old}}$

 5: for each $\{v_n\}_{n=1,\dots,N}$ do $\hat{\gamma}_n^{\text{new}} = \gamma_n$;
 6: for each $\{c_m\}_{m=1,\dots,M}$ do
 7:     for each $v_n \in \mathcal{H}(c_m)$ do $\mu_{m\to n} = \Psi_m \boxminus \hat{\gamma}_n^{\text{old}}$
 8:     for each $v_n \in \mathcal{H}(c_m)$ do $\hat{\gamma}_n^{\text{new}} = \hat{\gamma}_n^{\text{new}} + \mu_{m\to n}$

 9: for each $\{v_n\}_{n=1,\dots,N}$ do $\hat{\gamma}_n^{\text{old}} = \hat{\gamma}_n^{\text{new}}$

10: for each $\{v_n\}_{n=1,\dots,N}$ do $\hat{x}_n = (1 - \text{sign}(\hat{\gamma}_n))/2$
11: if $\hat{\boldsymbol{x}}$ is codeword then exit the iteration loop
    End iterative processing loop
```

Unlike the parallel decoder, the estimated a posteriori values $\hat{\gamma}_n^{\text{new}}$ are initialized to the channel values in the line 5 and updated during the serial processing of check nodes in the second half-iteration, which is performed in the lines 6–8. The updating is performed using d_c processors for the $\boxminus$ and d_c processors for real additions. During the processing in one check node, all outgoing messages from the check node are computed at once in the line 7. After that all neighbors of the check node are updated at once in the line 8. This procedure is performed for all check nodes. As a result, at the end of the second half iteration, the a posteriori values are the same as the values computed in the parallel decoder. The computed a posteriori values are saved in the line 9, and used in the next iteration. In this way, computations are done using M processors for $\boxplus$ operation, d_c processors for the $\boxminus$ operation and d_c processors for real additions. The discussion is summarized in the second row of Table I.

Note that in semi-parallel APP decoder we do not need to store the messages from all check nodes, but only the messages from the processed one, which results in lower memory complexity than in the parallel decoder. We need M registers for storing the check-sums Ψ_m, N registers for the estimated a posteriori values $\hat{\gamma}_n^{\text{new}}$, d_c registers for storing check-to variable messages $\mu_{m\to n}$ and N registers for saved values $\hat{\gamma}_n^{\text{old}}$. This results in the total number of $M + 2N + d_c$ registers, as given in Table II.

	$+$			$\boxplus$			$\boxminus$		
	P_+	C_+	E_+	$P_\boxplus$	$C_\boxplus$	$E_\boxplus$	$P_\boxminus$	$C_\boxminus$	$E_\boxminus$
Parallel APP	N	d_v	d_vN	M	d_c	d_vN	d_vN	1	d_vN
Semi-parallel APP	d_c	M	d_vN	M	d_c	d_vN	d_c	M	d_vN
Serial APP	1	d_vN	d_vN	d_v	N	d_vN	d_v	N	d_vN

TABLE I: Computational complexity of APP decoders. P_* is number of processors, C_* time for the performing of all operations * if P_* works in parallel, and $E_* = P_* \cdot C_*$ the total number of * operations performed during one iteration

	Processors	Memory	Energy	Throughput
Parallel APP	$(\alpha_m N + M + d_vN)m$	$(d_v+1)N + M$	$d_v \cdot N \cdot E \cdot K_{par}$	$N/K_{par}(\alpha_c d_v + d_c + 1)T$
Semi-parallel APP	$(\alpha_m d_c + M + d_c)m$	$2N + 2d_c + M$	$d_v \cdot N \cdot E \cdot K_{par}$	$N/K_{par}(\alpha_c M + d_c + M)T$
Serial APP	$(\alpha_m + 2d_v)m$	$2d_v + 1$	$d_v \cdot N \cdot E \cdot K_{par}$	$N/K_{par}(\alpha_c d_v N + 2N)T$

TABLE II: Complexity of APP decoders.

Algorithm 3: SERIAL APP DECODER

```
Input: $\boldsymbol{y} = (y_1, \dots, y_N) \in \mathcal{Y}^N$          ▷ received word
Output: $\hat{\boldsymbol{x}} = (\hat{x}_1, \dots, \hat{x}_N) \in \{0,1\}^N$          ▷ estimated codeword

   Initialization:
 1: for each $\{v_n\}_{n=1,\dots,N}$ do $\gamma_n = \log \frac{Pr(x_n = 0|y_n)}{Pr(x_n = 1|y_n)}$!!!!!!;
 2: for each $\{v_n\}_{n=1,\dots,N}$ do $\hat{\gamma}_n = \gamma_n$;
 3: for each $\{c_m\}_{m=1,\dots,M}$ do $\Psi_m = \boxplus_{n \in \mathcal{N}(m)} \hat{\gamma}_n$
   Iterative processing loop
 4: for each $\{v_n\}_{n=1,\dots,N}$ do
 5:     for each $c_m \in \mathcal{H}(v_n)$ do $\Psi_m = \Psi_m \boxminus \hat{\gamma}_n$
 6:     $\hat{\gamma}_n = \gamma_n$
 7:     for each $c_m \in \mathcal{H}(v_n)$ do $\hat{\gamma}_n = \hat{\gamma}_n + \Psi_m$
 8:     for each $c_m \in \mathcal{H}(v_n)$ do $\Psi_m = \Psi_m \boxplus \hat{\gamma}_n$
 9: for each $\{v_n\}_{n=1,\dots,N}$ do $\hat{x}_n = (1 - \text{sign}(\hat{\gamma}_n))/2$
10: if $\hat{\boldsymbol{x}}$ is codeword then exit the iteration loop
   End iterative processing loop
```

C. Serial APP algorithm

A serial APP decoder is based on schedule proposed decoding [5] and [7] for BP decoding. The decoder is presented in **Algorithm 3**.

In the serial APP decoder variable nodes are processed one by one. The check sums are initialized in the line 3, and after that updated during the whole iterative process. After the update in the line 5, the values Ψ_m become the messages $\mu_{m\to n}$ to the proceeded variable node, which are used for the variable node updating in the lines 6 and 7. The newly computed a posteriori values are after that added to Ψ_m in the line 8, so that Ψ_m again correspond to the check sums.

Note that, unlike for the semi-parallel decoder, in the serial one newly computed a posteriori values are used immediately after computation, for the computations of all subsequent a posteriori values in the same iteration. In the case of BP decoder, it has already been known that this updating improves the convergence [5], [7].

During the processing of one variable node, the decoder uses only one processor which performs d_v real additions, and d_v processors for the $\boxplus$ and $\boxminus$ operations which do updates at once. After that, the processors become available for the processing of the next variable node. This is summarized in the third row of Table I. Unlike the semi-parallel decoder, the message are stored in the check-sum registers and the copies of a posteriori values are not made, which results in total number of $M + N$ registers for storing the values Ψ_m and $\hat{\gamma}_n$, as shown in Table II.

D. Throughput analysis

Let $E_\boxdot$ denote the total number of $\boxdot$ operations which should be performed in one iteration ($\boxdot$ may stand for $+$,

$\boxplus$ or $\boxminus$). If the computations are performed using using $P_{\boxdot}$ processors which work in parallel and each of them runs $C_{\boxdot}$ times per one iteration, than we have $E_{\boxdot} = P_{\boxdot}\ C_{\boxdot}$.

Note that $+$ operation are performed with the processors which are associated to variable nodes, $\boxplus$ operation is performed with the one associated to check nodes and $\boxplus$ with the one associated with edges of the Tanner graph. Also note that, in one iteration, $\boxplus$ processors can start running only if all $+$ has finished the computation, and $\boxminus$....

In accordance, the average time, T, needed for finishing one iteration can be computed as

$$T = C_+\ t_+ + C_{\boxplus}\ t_{\boxplus} + C_{\boxminus}\ t_{\boxminus} \tag{8}$$

where $t_{\boxdot}$ stands for time needed for performing $\boxdot$ operations. Although, $\boxplus$ and $\boxminus$ requires higher time, an efficient and very accurate approximation have previously been proposed, and in our considerations we consider that all operations are performed for same time t for all the operations, in one time clock $f = 1/t$ so that

$$T = (C_+ + C_{\boxplus} + C_{\boxminus})\ t = \frac{C}{f}. \tag{9}$$

with $C = T/t = C_+ + C_{\boxplus} + C_{\boxminus}$.

If we introduce an effective processing power P, so that $E = P\ C$, we have

$$P = \frac{E}{C} = \frac{E}{\frac{E_+}{P_+} + \frac{E_{\boxplus}}{P_{\boxplus}} + \frac{E_{\boxminus}}{P_{\boxminus}}} \tag{10}$$

and

$$\frac{C}{N} = \frac{E}{N\ P} = \frac{3d_v}{P} \tag{11}$$

In each iteration in each of N variable nodes, $+$ operation is performed for d_v times, which is $E_+ = N\ d_v$ in total. In similar manner we get $E_{\boxplus} = E_{\boxminus} = M\ d_c = N\ d_v$ i.e. $E_+ = E_{\boxplus} = E_{\boxminus} = \frac{E}{3}$:

$$P = \frac{3}{\frac{1}{P_+} + \frac{1}{P_{\boxplus}} + \frac{1}{P_{\boxminus}}} \tag{12}$$

The average number of iterations needed I depends of the type of the decoder. Since the parallel and semi-parallel APP decoder yields exactly the same output after each iteration, their average iterations numbers are the same, $I_{par} = I_{s-par}$. The serial decoder has better convergence, and the convergence increasing depends on the type of code considered. We use the assumption from [8] by which $I_{ser} = 1/2\ I_{par}$. Finally, the total average time needed for the decoding of of one code-word can be expressed as $T\ I_{par}$ for the parallel and semi-parallel decoder and as $T\ I_{par}$

Throughput of a decoder is defined as a number of information bits per time unit which can proceeded:

$$D = \frac{K}{T\ I} = \frac{R\ N}{T\ I} = \frac{R\ N\ f}{C\ I} = \frac{R\ N\ f\ P}{E\ I} = \frac{R\ f\ P}{3\ d_v\ I} = . \tag{13}$$

$$D = \frac{R\ f}{d_v\ I} \cdot \frac{1}{\frac{1}{P_+} + \frac{1}{P_{\boxplus}} + \frac{1}{P_{\boxminus}}}. \tag{14}$$

IV. Conclusion

Two memory efficient APP algorithms for decoding of LDPC codes were presented. The decoders are based on semi-parallel and serial node processing and require memory that is linear in the number of nodes in the Tanner graph corresponding to the LDPC code. The decoders have the same computational complexity as the parallel version [3] which requires memory that is proportional to the number of edges. We provided precise expressions for the memory and computational complexity of the decoders in terms of the number of real operations and basic memory units required for the decoding. The dependency of the

References

[1] D. J. C. Mackay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inf. Theory*, vol. 45, no. 2, pp. 399–431, Mar. 1999.

[2] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inf. Theory*, vol. 27, no. 5, pp. 533–547, May 1981.

[3] M. Fossorier, M. Mihaljević, and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *Communications, IEEE Transactions on*, vol. 47, no. 5, pp. 673–680, May 1999.

[4] F. Guilloud, E. Boutillon, J. Tousch, and J.-L. Danger, "Generic description and synthesis of ldpc decoders," *Communications, IEEE Transactions on*, vol. 55, no. 11, pp. 2084–2091, Nov 2007.

[5] J. Zhang and M. Fossorier, "Shuffled belief propagation decoding," in *Signals, Systems and Computers, 2002. Conference Record of the Thirty-Sixth Asilomar Conference on*, vol. 1, Nov 2002, pp. 8–15 vol.1.

[6] V. Savin, "Chapter 4 - {LDPC} decoders," in *Academic Press Library in Mobile and Wireless Communications*, D. Declerq, M. Fossorier, and E. Biglieri, Eds. Oxford: Academic Press, 2014, pp. 211 – 259. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780123964991000042

[7] H. Kfir and I. Kanter, "Parallel versus sequential updating for belief propagation decoding," *Physica A: Statistical Mechanics and its Applications*, vol. 330, no. 1–2, pp. 259–270, 2003.

[8] J. Kim and W. Sung, "Rate-0.96 ldpc decoding vlsi for soft-decision error correction of nand flash memory," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 22, no. 5, pp. 1004–1015, May 2014.