Memory Efficient APP Decoders for Low-density parity-check codes

Velimir Ilić[1], Elsa Dupraz[2], David Declercq[3], Bane Vasić[4]
[1]Mathematical Institute SANU, Belgrade, Serbia
[2,3]ETIS laboratory, 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. 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 at least proportional to the number of edges. We derive the precise expressions for the memory and computational complexity of the decoders in terms of the the number of real operations and basic memory units required for the decoding and consider the decoders throughput dependency on the memory complexity.**

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 performances 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 applications with stringing constraints.

A posteriori probability (APP) decoder [3] is a suboptimal alternative to BP, in which the variable node processing is simplified by allowing variables to send messages in an intrinsic manner, and 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 presented as a message passing algorithm.

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 proposed in [3]. The algorithms use the advantages of different types of message passing scheduling, previously developed for the BP algorithm. The first one uses the flooding schedule over the variable nodes [4] and operates in a semi-parallel way by processing all the variable nodes per one time clock, which produces exactly the same output as the original APP decoder. The second one operates in a serial manner [5], by processing one variable node per time clock, with further decreasing of the memory requirements.

The computational and memory complexity analysis of the tree types of the APP decoders is conducted in terms of the number of operations and basic memory units required for the decoding. We also discuss the impact of the memory saving the decoding throughput, with the conclusion that, when the throughput is not an issue, the serial decoder should be used for the low memory decoding, otherwise, the semi-parallel decoder is applicable.

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.

II. APP decoding of LDPC codes

In this section we introduce basic definitions form LDPC codes theory and present the parallel message-passing APP decoder proposed in [3] .

*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).

A. LDPC codes

An regular LDPC code is a linear block code defined by a sparse parity-check matrix H. We denote by (M, N) the size of H. A codeword is a vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_N) \in \{0,1\}^N$ that satisfies $H\boldsymbol{x}^T = 0$, where $\boldsymbol{x}^T$ denotes the transposed (column) vector. 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 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 $\mathcal{M}(n)$ and the set of indices of variable-nodes connected to the check-node c_m is denoted with $\mathcal{N}(m)$.

Let $\boldsymbol{x} = (x_1, x_2, \ldots, x_N)$ be the transmitted binary codeword, and 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

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

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

B. APP decoders

The goal of the decoding is to compute the a posteriori probability $\Pr(x_n|\boldsymbol{y})$, which used for the decision making on bit values. APP decoder originally proposed in [3] is an iterative decoder which operates as follows.

Initialization: Variable-nodes are initialized to *a priori* values $(\gamma_1, \gamma_2, \ldots, \gamma_n)$, the received sequence $(y_1, y_2, \ldots, y_N)$, prior to the first iteration of the APP decoder:

$$\tilde{\gamma}_n^{(0)} = \gamma_n = \frac{p(x_n = 0|y_n)}{P(x_n = 1|y_n)}. \tag{1}$$

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;

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

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

$$x \boxplus y = \log \frac{1 + \mathrm{e}^x \mathrm{e}^y}{\mathrm{e}^x + \mathrm{e}^y} \tag{3}$$

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

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

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:

for each $\{v_n\}_{n=1,\ldots,N}$ **do** $\gamma_n = \log \dfrac{Pr(x_n = 0|y_n)}{Pr(x_n = 1|y_n)}$;

for each $\{v_n\}_{n=1,\ldots,N}$ **do** $\tilde{\gamma}_n^{(0)} = \gamma_n$;

Iteration loop: $k > 0$

for each $\{c_m\}_{m=1,\ldots,M}$ **do** $\Psi_m^{(k)} = 0$
  total check-sum computation
  for each $v_n \in \mathcal{H}(c_m)$ **do**

$$\Psi_m^{(k)} = \Psi_m^{(k)} \boxplus \tilde{\gamma}_n^{(k-1)}$$

  check-to-variable messages computation
  for each $v_n \in \mathcal{H}(c_m)$ **do**

$$\mu_{k\to m}^{(n)} = \Psi_m^{(k)} \boxminus \tilde{\gamma}_n^{(k-1)}$$

a posteriori update
for each $\{v_n\}_{n=1,\ldots,N}$ **do** $\tilde{\gamma}_n^{(k)} = \gamma_n$
  for each $c_m \in \mathcal{H}(v_n)$ **do**

$$\tilde{\gamma}_n^{(k)} = \tilde{\gamma}_n^{(k)} + \mu_{k\to m}^{(n)}$$

hard decision
for each $\{v_n\}_{n=1,\ldots,N}$ **do** $\hat{x}_n = (1 - \mathrm{sign}(\tilde{\gamma}_n))/2$
if $\hat{\boldsymbol{x}}$ is codeword **then** exit the iteration loop

End iteration loop

3) Hard decision: Estimated (binary) 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 $\boxplus_{k\in\mathcal{N}(m)\setminus n} \tilde{\gamma}_k^{(k-1)}$, which can efficiently be computed using the inverse operation for $\boxplus$ called minus-box operator:

$$x \boxminus y = \log \frac{1 - \mathrm{e}^x \mathrm{e}^y}{\mathrm{e}^x - \mathrm{e}^y} \tag{5}$$

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

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

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

$$\mu_{m\to n}^{(k)} = \Psi_m^{((k))} - \tilde{\gamma}_n^{((k))}. \tag{7}$$

The pseudo-code for the message passing implemented APP deocder proposed in [3] is given in **Algorithm 1**.

Algorithm 1: SEMI-PARALLEL URAPP 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:

for each $\{v_n\}_{n=1,\ldots,N}$ **do** $\gamma_n = \log \dfrac{Pr(x_n = 0|y_n)}{Pr(x_n = 1|y_n)}$;

for each $\{v_n\}_{n=1,\ldots,N}$ **do** $\tilde{\gamma}_n^{(0)} = \gamma_n$;

Iteration loop: $k > 0$
Total check-sum computation
for each $\{c_m\}_{m=1,\ldots,M}$ **do** $\Psi_m^{(k-1)} = \boxplus\limits_{n \in \mathcal{N}(m)} \tilde{\gamma}_n^{(k-1)}$

partial a posteriori update
for each $\{c_m\}_{m=1,\ldots,M}$ **do**
  for each $v_n \in \mathcal{H}(c_m)$ **do**

$$\mu = \Psi_m^{(k-1)} \boxminus \tilde{\gamma}_n^{(k-1)}$$
$$\tilde{\gamma}_n^{(k)} = \tilde{\gamma}_n^{(k)} + \alpha\mu$$

a posteriori update
for each $\{v_n\}_{n=1,\ldots,N}$ **do** $\tilde{\gamma}_n^{(k-1)} = \tilde{\gamma}_n^{(k)}$
hard decision
for each $\{v_n\}_{n=1,\ldots,N}$ **do** $\hat{x}_n = (1 - \text{sign}(\tilde{\gamma}_n))/2$
if $\hat{\boldsymbol{x}}$ is codeword **then** exit the iteration loop

End iteration loop

Algorithm: SERIAL 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:

for each $\{v_n\}_{n=1,\ldots,N}$ **do** $\gamma_n = \log \dfrac{Pr(x_n = 0|y_n)}{Pr(x_n = 1|y_n)}$;

for each $\{v_n\}_{n=1,\ldots,N}$ **do** $\tilde{\gamma}_n^{(0)} = \gamma_n$;
Total check-sum computation
for each $\{c_m\}_{m=1,\ldots,M}$ **do** $\Psi_m^{(0)} = \boxplus\limits_{n \in \mathcal{N}(m)} \tilde{\gamma}_n^{(0)}$

Iteration loop: $k > 0$

for each $\{v_n\}_{n=1,\ldots,N}$ **do** $\tilde{\gamma}_n^{(k)} = \gamma_n$
  for each $\{c_m\}_{m=1,\ldots,M}$ **do**

$$\mu_{m \to n}^{(k)} = \Psi_m^{(k-1)} \boxminus \tilde{\gamma}_n^{(k-1)}$$
$$\tilde{\gamma}_n^{(k)} = \tilde{\gamma}_n^{(k)} + \mu_{m \to n}^{(k)}$$
$$\Psi_m^{(k)} = \mu_{m \to n}^{(k)} \boxplus \tilde{\gamma}_n^{(k)}$$

for each $\{v_n\}_{n=1,\ldots,N}$ **do** ▷ hard decision

$$\hat{x}_n = (1 - \text{sign}(\tilde{\gamma}_n))/2$$

if $\hat{\boldsymbol{x}}$ is codeword **then** exit the iteration loop

End iteration loop

III. MEMORY EFFICIENT APP DECODING

In a common, parallel message passing implementation of the APP decoder, all the variable nodes take the message at same time, the a posteriori values are computed, which completes one iteration. Although this version provides high throughput, it suffers from the high memory requirements, proportional to the number of edges in the Tanner graph, since an iteration requires the storing of all check to variable messages for one iteration. In the next section we propose two memory efficient variants of the APP decoder.

A. Semi-parallel APP algorithm

Semi-parallel APP decoder uses the flooding schedule over the variable nodes [4] and operates by processing all the variable nodes per one time clock, producing exactly the same output as the original APP decoder (see **Algorithm 2**). Instead of the messages, we store only the values $\Psi_m^{(k-1)} = \boxplus\limits_{n \in \mathcal{N}(m)} \tilde{\gamma}_n^{(k-1)}$ which are used for the computation of the posterior values $\tilde{\gamma}_n^{(k)}$ and, at one iteration, all variable nodes can be partially updated during the computations in all variable nodes.

Although the semi-parallel version the smaller throughput than the parallel one, since two check nodes might try to access the same variable node to update its a posteriori value, it needs the storing only the values in variable nodes. As a result it have the complexity proportional to the number of nodes, which is a significant saving.

B. Serial APP algorithm

In the semi-parallel APP decoder, the values $\Psi_m^{(k-1)}$ are computed according to the values $\tilde{\gamma}_n^{(k-1)}$ from the $k-1$-iteration and are further used for the computation of the a posteriori values $\tilde{\gamma}_n^{(k)}$ from the k-th iteration. Instead of this, it is possible to update the values $\Psi_m^{(k-1)}$ according to newly computed a posteriors $\tilde{\gamma}_n^{(k)}$, immediately after they have become available. This is the underlying idea for the serial APP decoder (**Algorithm 3**), which is based on the ideas from [5] and [7] proposed for the BP decoding. As it is shown in [5], [7], the serial processing increases the convergence speed and provides better bit error rates, but the cost is paid by lowered throughput and coding latency. Accordingly, when the throughput and latency are not an issue, the serial decoder should be used for the low memory decoding, otherwise, the semi-parallel decoder is applicable.

C. Computational and memory complexity analysis

The computational and memory complexities of the three APP algorithms are given in Table 1. The time complexity is defined as the number of operations (real addition, box-plus

	$+$	$\boxplus$	$\boxminus$	Memory
Parallel APP				
Ψ	$-$	dN	$-$	1
$\mu_{m\to n}^{(k)}$		$-$	dN	dN
$\tilde{\gamma}_n^{(k)}$	dN	$-$	$-$	N
Total	dN	dN	dN	$(d+1)N+1$
Semi-parallel APP				
$\Psi_m^{(k-1)}$	$-$	dN	$-$	M
μ	$-$	$-$	dN	1
$\tilde{\gamma}_n^{(k)}$	dN	$-$	$-$	N
$\tilde{\gamma}_n^{(k-1)}$	$-$	$-$	$-$	N
Total	dN	dN	dN	$M+2N+1$
Serial APP				
μ	$-$	$-$	dN	1
$\tilde{\gamma}_n^{(k)}$	dN	$-$	$-$	N
$\Psi_m^{(k)}$	$-$	dN	$-$	M
Total	dN	dN	dN	$M+N+1$

TABLE I: PS APP and GS APP complexities per iteration

and box-minus) required for the performing of the algorithm for a given pseudo code. The memory complexity is defined as the number of infinite precision bit registers needed to store variables during algorithm performing. To simplify analysis, we consider a case of a regular code with the variable degree $d = |\mathcal{M}(n)|$ (in this case, $N \cdot |\mathcal{M}(n)| = M \cdot |\mathcal{N}(m)| = dN$), and we assume that all groups are of the same size $|\mathcal{G}_l| = N/L$.

D. Discussion

Note that all of the three algorithms have the same computational complexity. On the other side, the memory complexity is directly proportional to the decoder throughput.

IV. Conclusion

Two memory efficient architectures for APP decoding of LDPC codes was presented, based on semi-parallel and serial node processing. Unlike the exiting, parallel APP decoder [3] that require memory that is at least proportional to the number of edges in the Tanner graph of the code, the proposed algorithms require memory that is linear in the number of nodes, wile keeping the same computational complexity as the parallel version. We provided precise expressions for the memory and computational complexity of the decoders in terms of the the number of real operations and basic memory units required for the decoding. The dependency on the memory complexity on the decoders throughput is also considered.

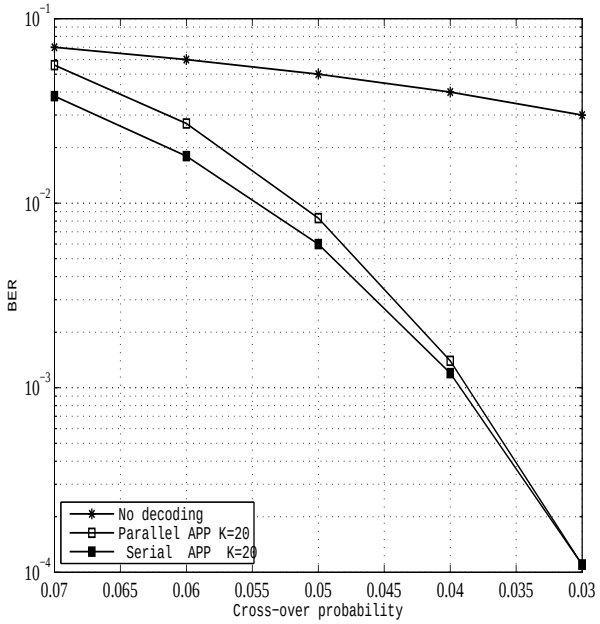

Fig. 1: Bit error rate of APP algorithms: regular (504,252) LDPC code.

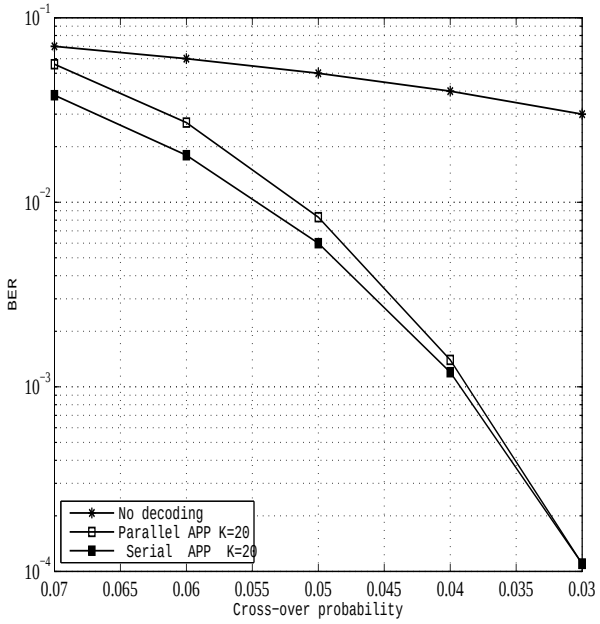

Fig. 2: Bit error rate of APP algorithms: regular LDPC code (PEGReg504x1008).

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.