LDPC codes - new methodologies

by

Jan Broulím

Submitted to the Department of Applied Electronics and Telecommunications
in fulfillment of the requirements for the degree of

Ph.D.

at the

UNIVERSITY OF WEST BOHEMIA

August 2018

Author . . . .
Department of Applied Electronics and Telecommunications
24 August 2018

LDPC codes - new methodologies

by

Jan Broulím

Submitted to the Department of Applied Electronics and Telecommunications
on 24 August 2018, in fulfillment of the
requirements for the degree of
Ph.D. in Electronics

Abstract

Low Density Parity-Check (LDPC) codes have become very popular because of their near Shannon limit performance when decoded using a probabilistic decoding algorithm. This work proposes several methodologies related to LDPC codes, including design of codes based on optimsation algorithms, mapping LDPC decoders onto parallel architectures, and improving performance of state-of-the-art decoders.

LDPC codes are random-based codes, defined in terms of parity-check matrices or Tanner graphs. Parameters of Tanner graphs, particularly a degree distribution and cycle occurrence, are crucial for probabilistic iterative decoders. Therefore, algorithms for producing good codes are needed. In this work, an algorithm for producing codes of large girth is proposed and evaluated. This algorithm is further utilsed for genetic optimzation methods accelerated by coarse grained parallelzation. The proposed methods are evaluated using different code lengths and redundancies.

The second part of this thesis is devoted to mapping LDPC decoders on parallel systems, which are becoming very popular in modern communications systems. A general method for mapping irregular LDPC codes is proposed and evaluated on GPU platform using OpenCL and CUDA frameworks.

The last main part introduces algorithms for improving performance of LDPC codes. Two main methods are proposed, a method based on backtracking codeword estimations and a method based on using several parity-check matrices. The second method, so called Mutational LDPC (MLDPC), utilses several parity-check matrices produced by slight mutations which run in parallel decoders. Information from all decoders is then used to provide the codeword estimation. The MLDPC is further modified using information entropy and so called radius which provide the additional improvement of the Bit Error Rate.

Thesis Supervisor: Doc. Dr. Vjačeslav Georgiev

Thesis Co-Supervisor: prof. RNDr. Tomáš Kaiser, DSc.

Acknowledgments

I would thank to my supervisors, Vjaceslav Georgiev and Tomas Kaiser, to take the responsibility of this work and for giving me the opportunity to learn and develop new skills in interesting fields.

I am particularly grateful for the assistance and useful discussions given by Alexander Ayriyan, Stefan Berezny, Nikolaos Boulgouris, Sima Davarzani, Hovik Grigorian, colleagues at the Faculty of Electrical Engineering in Pilsen, Institute of Experimental and Applied Physics in Prague, Joint Institute of Nuclear Research, especially the HybriLIT group, people involved in GBT Project at CERN, and specialists at Brunel Language Centre.

My thanks also go to my family for providing the support during my study.

Declaration

I hereby declare that this thesis is my original work and that I have not used any sources other than those listed. I further declare that only legal and licensed software has been used. This thesis has not been submitted at any other institution in order to obtain a degree.

....

Contents

List of Figures

List of Tables

List of Symbols

AWGN Additive White Gaussian Noise

BER Bit Error Rate

BFS Breadth-First Search

BPSK Binary-Phase Shift Keying

DVB-S Digital Video Broadcasting - Satellite

DVB-T Digital Video Broadcasting - Terrestrial

FPGA Field Programmable Gate Array

G Generator matrix

GPU Graphics Processing Unit

H Parity-Check matrix

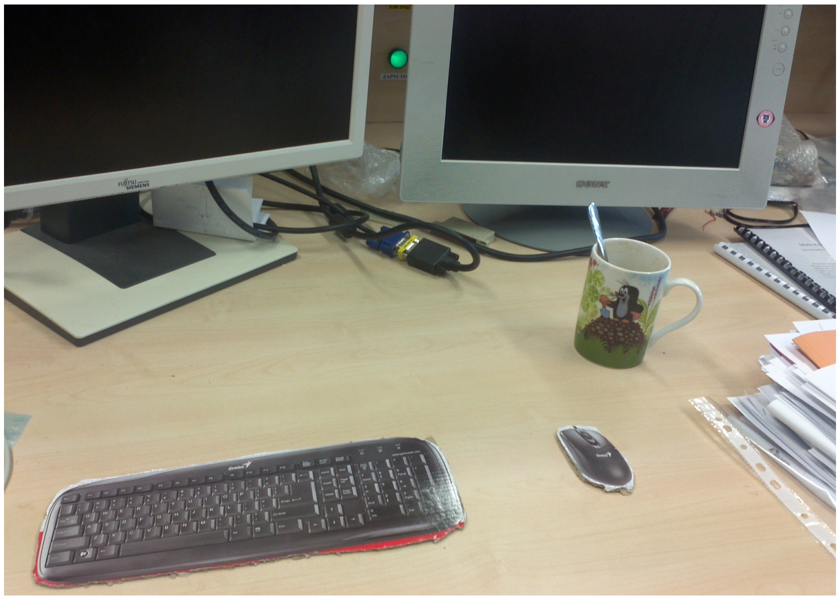
LDPC Low Density Parity-Check

MLDPC Mutational LDPC

MLDPCe MLDPC using information Entropy

MLDPCr MLDPC using Radius

SNR Signal-to-Noise Ratio



The place where this text has been mostly written :-)

Chapter 1

Introduction

With the increasing development of electronics and telecommunication technologies, there was a need for definitions, new terminology and a generalization of communication processes. The first theoretical contribution to the generalization of a data transmission was made by Shannon's work in 1948 [46], where he defined quantities connected with a new discipline called information theory, established limits for communication processes between a source and a destination and introduced a schematic diagram (Fig. 2-1) of a general communication model.

Since Shannon's work, the topic of error detection and error correction codes, related to channel coding, has seen significant growth. The first serious discussion of error correction codes emerged in Hamming's work in 1950 [22], where Hamming provided a method for the correction of single bit and the detection of double bit errors with minimum redundancy being added to the data transmitted. Since the second half of 20th century, error correction codes have attracted much attention in research work and have been utilized in many applications, including space photography transmission, television broadcasting services, Ethernet, or wireless communication networks.

Since the invention of correction codes and establishment of space programmes, space agencies, i. e. NASA and ESA, have become a common user of these codes. Reed-Muller codes [49],[44] found an application in transmission of Mars photographs in the Mariner spacecraft mission between 1969 and 1977. Reed-Solomon codes [53] and convolutional codes were used on the Voyager 1 space probe mission, whereas a Golay code [25] was used on the Voyager 2 (both were launched in 1977). The New Horizon spacecraft, currently enroute to Pluto, uses turbo codes [62] for space transmission and Reed-Solomon codes in an internal system.

Although Hamming codes are still used in random access memories, more robust codes are commonly deployed. Reed Solomon codes have become familiar on compact discs, digital versatile discs, Blu-Ray discs, or hard discs. They are also used as a coding technique at CERN for data transmission from detectors or as an outer code to convolutional codes in television broadcasting standards (DVB-T, DVB-S). However, DVB-S2 (ratified in 2005) [88] and DVB-T2 (2006) standards use a combination of BCH [7] and LDPC codes [86]. LDPC codes were further standardized as an option to other coding schemes for communication networks, such as 10 Gigabit Ethernet (10GBASE-T) in 2006, WiMAX (802.16e) in 2006 , WiFi (802.11n in 2009 or 802.11ac in 2014 [87]). As can be seen from the number of applications with LDPC codes, which is increasing, these codes have became the very popular coding technique used for an establishment of reliable data transmission. Another interesting coding technique includes polar codes [2]. Also these codes have seen growing interest because of their low coding complexity.

Low Density Parity-Check (LDPC) codes were introduced by Gallager [24] in 1962. Since MacKay's rediscovery [37] in 1995, the past two decades have seen increasingly rapid advances in the field of LDPC codes. There is a growing number of applications with LDPC codes and Sum-Product (SP) decoding [70]. Recently, the Progressive-Edge Growth (PEG) [74] construction has been becoming popular. It was said, "The PEG construction creates matrices with very large girth. This construction has proved to produce the best known Gallager codes." (MacKay, 2008). Moreover, LDPC codes can be constructed in a wide variety of block lengths or redundancies, and the decoder is able to report that the block has been incorrectly decoded, which is not a common behavior of all decoders associated with different types of codes.

This text is organized as follows. Chapter 2 provides general ideas of error correction coding. Chapter 3 presents LDPC codes in terms of their history, principles of contruction, encoding and decoding. Chapter 4 introduces innovative construction methods, applied in combination with genetic optimization algorithms. A performance of the proposed methods is evaluated and new codes are compared to the codes of the same length. Chapter 5 proposes a general method for the parallelization of irregular LDPC codes. Benchmarks are performed using GPU platform. In Chapter 6, several methodologies improving performance of decoders are proposed and the performance is compared to the state-of-the art decoders using the same LDPC code.

Chapter 2

Error correction coding

2.1 History of error correction coding

Since Hamming reported the first generalized correction codes there has been an increasing amount of interest in this field. In 1954, D. Muller published a paper on the application of Boolean algebra to electrical circuits and error detection [44], where he established the minterm terminology. Working independently of Muller, I. Reed reported codes with bit error correctability greater than two [49]. Codes that came from their work are currently known as Reed-Muller. In 1955, P. Elias introduced convolutional codes [17] and later, in 1957, E. Prange introduced cyclic codes [48]. This trend was followed by BCH and Reed-Solomon codes in 1960 [7], [53]. The first mention of LDPC codes appeared in Gallager's work in 1962 [24], but this work did not have a considerable impact for several decades. Afterwards, in 1993, there was a revolution with the introduction of turbo codes [62], whose decoding performance was significantly closer to Shannon's bound than any other codes published previously. In 1995, MacKay and Neal rediscovered Gallager's LDPC codes [37], which had been neglected for over 30 years (few mentions by Zyablov, Pinsker and Tanner [85], [56]). MacKay's rediscovery of LDPC codes in 1995 [37] has heightened the need for other studies on those codes. In 1998, Davey and MacKay proposed non-binary LDPC codes, which outperformed turbo codes [14], [15]. Spielman also showed that binary LDPC codes using heuristic construction can surpass turbo codes in terms of correction capability [54].

2.2 Shannon model

The Shannon model (Fig. 2-1) divides the communication process into separate blocks with specific functions. Data provided by an information source are transmitted and passed through a noisy channel to a receiver, where they are processed and transferred to a destination. Today, the transmitter is often divided into the source encoder and the channel encoder, according to the theory introduced by Shannon. Additionally, a modulator can be located between the channel encoder and the channel if it is not considered as a part of the channel encoder. Similarly, the receiver can be divided into a demodulator, a channel decoder, and a source decoder (Fig. 2-2).

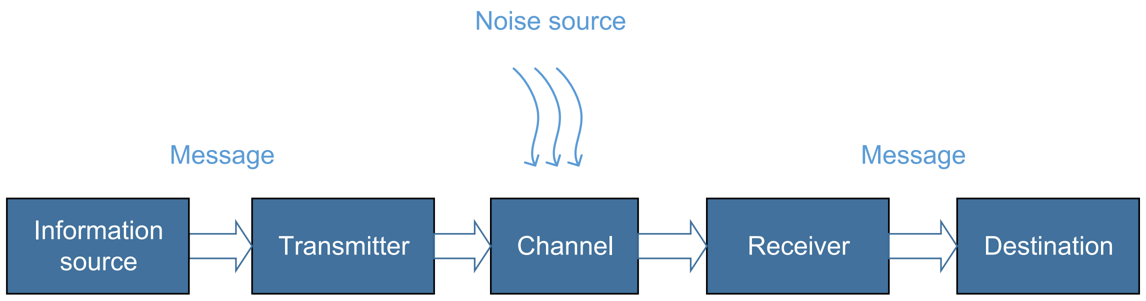


Figure 2-1: Shannon's conceptual diagram of the information transmission between the source and the destination.

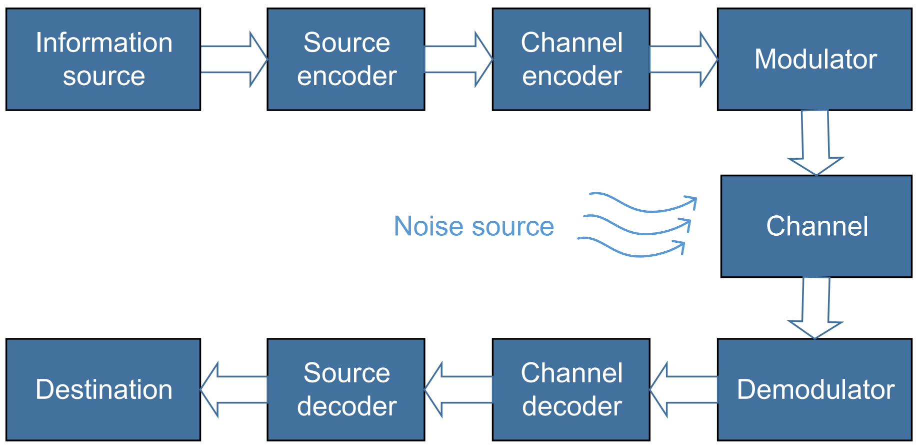


Figure 2-2: Extended diagram of the communication chain.

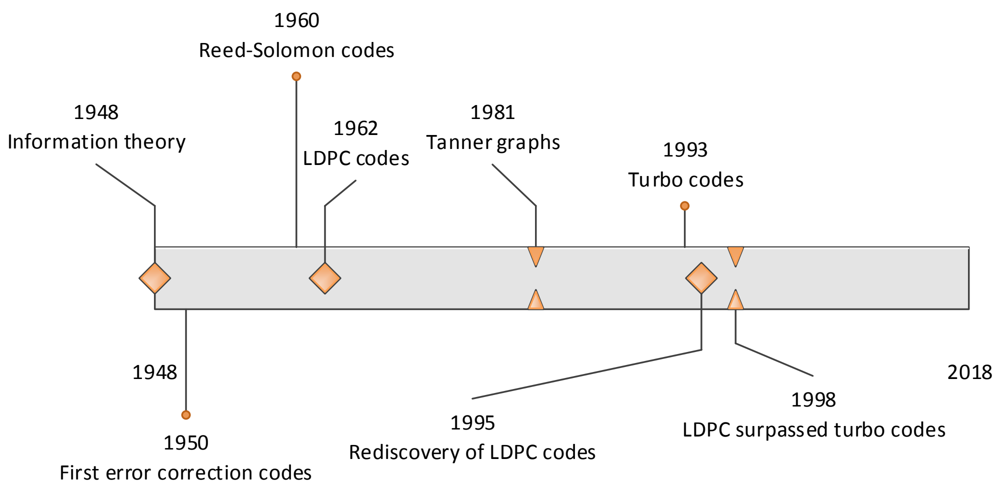


Figure 2-3: Historical milestones in coding theory.

2.3 Definitions

Alphabet $\mathcal{A}$ is said to be the set of q symbols, usually considered that $\mathcal{A}$ forms a field.

A block code over an alphabet $\mathcal{A}$ is a set of q^k vectors (codewords) of length n.

A block code C is a linear (n,k) code if and only if its q^k codewords form a k-dimensional vector subspace of the vector space $\mathcal{F}_q^n$ over a field $\mathcal{F}_q$, where $\mathcal{F}_q^n$ is a space of q^n vectors. The number n is the length of the code and the number k is the dimension of the code. The examples of linear and nonlinear codes can be seen in Table 2.1.

The Hamming distance d between vectors $\mathbf{x}$ and $\mathbf{y}$ of equal length ($|\mathbf{x}|$=$|\mathbf{y}|$) is the number of positions where the corresponding elements are different,

$$d(\mathbf{x}, \mathbf{y}) = \sum_{n=1}^{|\mathbf{x}|} d_i \tag{2.1}$$

$$d_i = \begin{cases} 0 \text{ for } x_i = y_i \\ 1 \text{ for } x_i \neq y_i \end{cases} \tag{2.2}$$

The minimum distance d_{min} of a block code C is the smallest Hamming distance between any two codewords in the code C.

Table 2.1: Hamming (7,4) code and 4B/5B coding

message $\mathbf{m}$	codeword $\mathbf{c}$	
	Hamming (7,4)	4B/5B
0000	0000000	11110
0001	0001111	01001
0010	0010110	10100
0011	0011001	10101
0100	0100101	01010
0101	0101010	01011
0110	0110011	01110
0111	0111100	01111
1000	1000011	10010
1001	1001100	10011
1010	1010101	10110
1011	1011010	10111
1100	1100110	11010
1101	1101001	11011
1110	1110000	11100
1111	1111111	11101

$$d_{min} = \min_{\mathbf{c_1},\mathbf{c_2} \in \mathcal{C} \wedge \mathbf{c_1} \neq \mathbf{c_2}} d(\mathbf{c_1}, \mathbf{c_2}), \tag{2.3}$$

The Hamming weight $w(\mathbf{c})$ of a codeword $\mathbf{c}$ is the number of nonzero elements of the codeword.

The entropy $H(X)$ is a measure of the amount of uncertainty (average amount of self-information) associated with a discrete random variable X . For a source X with probabilities $P(X = x_i)$, where $x_i \in \{x_1 ... x_N\}$, the entropy is:

$$H(X) = -\sum_{i=1}^{N} (P(x_i) \log_2 P(x_i)) \quad \text{[Sh]}. \tag{2.4}$$

The information rate R of code $\mathcal{C}$ is given by:

$$R = \frac{\log_q |\mathcal{C}|}{n}, \tag{2.5}$$

where q is the number of symbols in the alphabet and n is the length of the codeword.

For a block code $\mathcal{C} = (n, k)$ the information rate becomes:

$$R = \frac{\log_q \left|q^k\right|}{n} = \frac{k}{n}, \tag{2.6}$$

where k is the length of the information message.

The redundancy r of the code is the difference between the length of the codeword and the length of the information message, $r = n - k$.

2.3.1 Generator matrix

Every linear block code $\mathcal{C}$ can be described in terms of the k by n generator matrix $\mathbf{G}$. The rows of $\mathbf{G}$ generate $\mathcal{C}$. The codeword $\mathbf{c}$ can be represented as a linear combination of the row vectors of $\mathbf{G}$. The information vector (message) is denoted as $\mathbf{m}$. Then,

$$\mathbf{c} = \mathbf{mG}. \tag{2.7}$$

2.3.2 Parity-check matrix

A parity check matrix $\mathbf{H}$, associated with a linear block code $\mathcal{C}$, is a matrix satisfying the formula:

$$\mathbf{GH}^\top = \mathbf{0}. \tag{2.8}$$

Any vector $\mathbf{v}$ of length n is a codeword if and only if it meets the following condition:

$$\mathbf{vH}^\top = \mathbf{0}. \tag{2.9}$$

A product of the multiplication $\mathbf{vH}^\top$ is called the syndrome $\mathbf{s}$.

2.3.3 Systematic form of coding

Consider information symbols (elements of $\mathbf{m}$) as the elements of the information vector $\mathbf{m}$. When the systematic form of coding is applied, information symbols are then written in the same order both in the codeword and in the original message. The systematic form of the generator matrix is the following: $\mathbf{G} = [\mathbf{P}\,|\,\mathbf{I}]$ or $[\mathbf{I}\,|\,\mathbf{P}]$, where $\mathbf{I}$ is the k by k identity matrix and $\mathbf{P}$ is a part of $\mathbf{G}$ matrix.

2.3.4 Non-binary codes

So far, only binary representations of the generator and a parity check matrix have been considered. Linear block codes can be generalized using finite field arithmetic, which is illustrated in following paragraphs.

The modular arithmetic with congruence modulo n relation is considered. If n is a prime, such an arithmetic generates a field, which can be utilized for computations with linear block codes. Because of the binary channels generally used in communications, Galois fields (GF) of 2^p polynomials are generally utilized. Such a field is created by any irreducible polynomial.

2.3.5 Communication channel models

There are two groups of mathematical models of communication channels, analogue and digital. Analogue channels, including Additive White Gaussian Noise (AWGN) or Rayleigh fading model, work with analogue messages being transmitted. Digital channel models, involving Binary Symmetric Channel (BSC) or Binary Erasure Channel (BEC), consider the messages as digital variables. The capacity associated with the channel is a measure first used by Shannon [46]. The definition in terms of entropy is the following:

The channel capacity C is the upper bound on the information being transmitted between the input and output (maximum mutual information),

$$C = \max\{I(X,Y)\} = \max\{H(Y) - H(Y|X)\}, \tag{2.10}$$

where $H(Y)$ is the destination entropy, $H(Y|X)$ is the conditional entropy, and $I(X,Y)$ is the mutual information. If $H(Y|X) = 0$, there are no errors in the transmission and the capacity is maximum. As the symbol duration T_s is known, the maximum bandwidth in bits per second can be determined by:

$$C' = \frac{C}{T_s} \quad \text{[bit/s, Sh/s]}. \tag{2.11}$$

For BSC, the capacity is given:

$$C = 1 - H = 1 + p \log_2 p + (1-p)\log_2(1-p) \quad \text{[Sh]} \tag{2.12}$$

Related to the AWGN channel, the maximum rate of transmitted information is given by the Shannon-Hartley theorem, which can be written in the following formula:

$$C = B \log_2 \left(1 + \frac{P_S}{P_N}\right) \quad \text{[bit/s]}, \tag{2.13}$$

where B is the bandwith, P_S is the power of the signal, P_N is the power of the noise.

Chapter 3

LDPC codes

Low Density Parity-Check (LDPC) codes are defined in terms of sparse parity check matrices (described in Section 2.3.2). Suppose a parity-check matrix $\mathbf{H}$ associated with the linear block code $\mathcal{C}$. If the matrix $\mathbf{H}$ is sparse, the code $\mathcal{C}$ is said to be the LDPC code. If the column weights in the matrix $\mathbf{H}$ are all the same and row weights are all the same, the LDPC code is regular. If not, it is irregular. It has been shown that irregular codes perform better [35].

LDPC construction, encoders, decoders, and historical milestones are outlined in the following chapter.

3.1 Background

Although only the regular form of the parity-check matrix was considered by Gallager, irregular constructions have become familiar because of their better performance. The first report on irregular codes was published by Luby, Mitzenmacher, Shokrollahi and Spielman in 1998 [35], followed by MacKay, Wilson and Davey in 1999 [38]. In 2001, a powerful code design, which is based on appropriate degree distribution in a Tanner graph, was presented by Richardson, Shokrollahi and Urbanke [50]. However, regular codes have been studied comprehensively in MacKay's work [39]. In 2003, Fossorier studied Quasi-Cyclic (QC) codes, based on circulant permutation matrices [21]. Later, in 2004, Vasic showed several combinatorial constructions of parity-check matrices [64], and Tian, Jones, Villasenor and Wesel studied an effect of the graph connectivity on the correction capability [61]. Probably still the best known code was identified in the work of Chung, Forney, Richardson and Urbanke [13]. Further, in 2004, Thorpe, Andrews, Dolinar proposed the construction of LDPC codes based on protographs [60]. Hu, Eleftheriou and Arnold, discovered in 2001 and generalized in 2005 an algorithm (the Progressive Edge-Growth (PEG)) based on a Tanner graph construction in terms of a tree structure [73], [74]. During the past ten years, this algorithm has become a general standard for the design of good codes. Numerous studies have attempted to modify PEG construction in order to achieve better performance. In 2010, Zheng, Lau and Tse presented a PEG modification improving performance at the error floor region [84]. Uchoa, Healy, de Lamare and Souza [63] introduced PEG techniques for fading channels in 2011. Our work on LDPC codes started in 2012 [10] with the utilization in a microcontroller and it was followed with probably the first direct application of genetic optimization methods on code construction in 2013 [11]. This work has led to results pointed out in Section 4.2. Furthermore, the genetic construction can be naturally optimized to all possible channels and decoding algorithms, where it outperformes state-of-the art solutions

Decoding algorithms used for LDPC codes are suboptimal and the convergence to the minimum distance codeword is not guaranteed. Thus, the correction performance depends both on the code and the decoding algorithm used. Several decoding methods, which differed in the performance and required number of operations, were presented in the literature. Soft-decision algorithms working on Tanner graphs were introduced by Wiberg [70]. Nonbinary and Fast Fourier Transform (FFT) versions were pointed by Davey in 1999 [16] and later, in 2004, log-domain FFT decoding was described by Byers and Takawira [9]. The work of Yedidia, Freeman and Weiss from 2001 contributed to generalization of Belief Propagation (BP) algorithms [76], including the decoding algorithms above.

Recently, researchers have shown an increased interest in the connection of neural networks and LDPC codes. A relationship between message passing and continuous Hopfield networks was revealed by Ott and Stoop in 2007 [47]. In 2009 Karami, Attari and Tavakoli presented perceptron neural networks for LPDC decoders [27] and later, in 2013, Anton, Ionescu, Tutanescu, Mazare and Serban presented an application with parallel Hopfield networks [1].

However, far too little attention has been paid to the multi-edge codes discovered by Richardson and Urbanke in 2004 [52]. This work was followed by Liva and Chiani [34] in 2007, where they provided a novel extrinsic information transfer analysis for protograph and multi-edge codes. Obata, Yung-Yih, Kasai and Pfister showed properties of multi-edge codes on binary erasure channels in 2013 [45].

Several patents have been granted for encoding or decoding implementations in hardware for special purposes (e.g. US 7,543,212 B2 or US 7,499,490 B2 in 2009) and several hardware implementations have been published [67], [80]. Nevertheless, very few studies have examined scalable implementations of decoders for irregular codes in Field-Programmable Gate Arrays (FPGA) (e.g. [57]). Important milestones covering LDPC codes are presented in the work of Bonello, Chen and Hanzo in 2011 [8]. The selected milestones are pointed out in Fig. 3-1

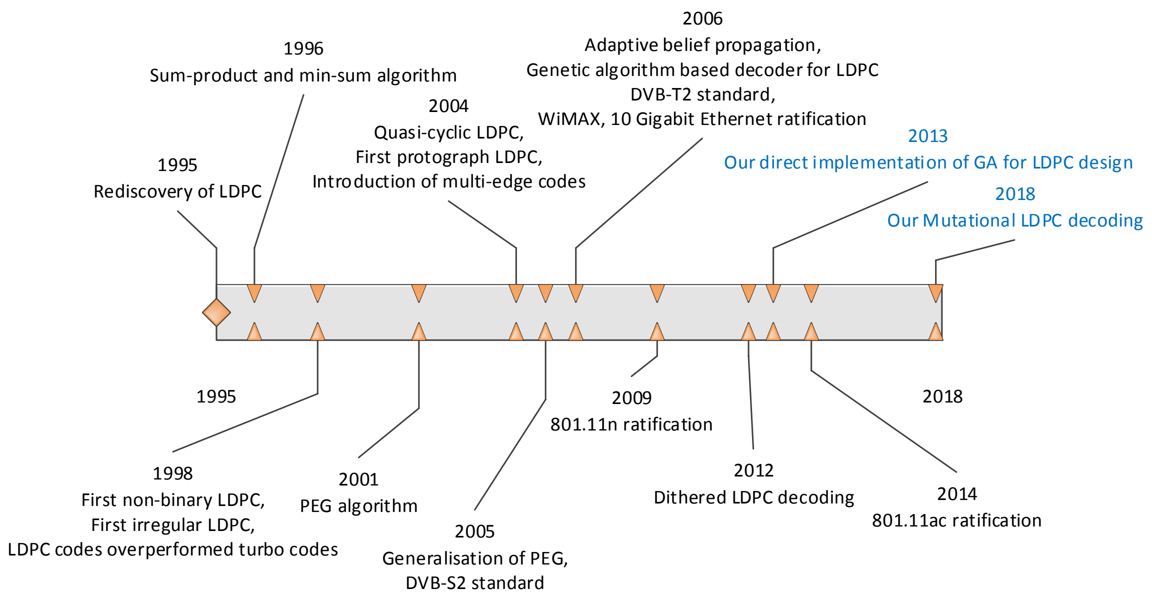


Figure 3-1: Historical milestones related to LDPC codes.

3.2 Encoding

The encoding of LDPC codes is performed by the multiplication of an information vector $\mathbf{m}$ with the generator matrix $\mathbf{G}$. Because LDPC codes are often defined by a parity-check matrix $\mathbf{H}$ solely, there is a need for finding the generator matrix $\mathbf{G}$, which satisfies Eq. 2.8. This can be done by the principle described below.

It is assumed that the parity-check matrix $\mathbf{H}$ is in the systematic form $\mathbf{H} = [\mathbf{P} \,|\, \mathbf{I}]$. A relation between $\mathbf{G}$ and $\mathbf{H}$ is given by:

$$\mathbf{G} = \left[\mathbf{I} \,|\, -\mathbf{P}^\top\right] \quad \Longleftrightarrow \quad \mathbf{H} = [\mathbf{P} \,|\, \mathbf{I}] \,, \tag{3.1}$$

where $\mathbf{I}$ is the k by k identity matrix.

3.3 Tanner graphs

The Tanner graph is a graph representation of linear block codes, which provides a support for decoding algorithms of LDPC codes. A parity-check matrix $\mathbf{H}$ of a linear block code $\mathcal{C}$ is considered as a part of an adjacency matrix of a bipartite undirected graph (Fig. 3.2). Nodes associated with columns of $\mathbf{H}$ are said to be the variable nodes, symbol nodes, or bit nodes, whereas nodes associated with rows of $\mathbf{H}$ are called the check nodes. Edges occur between these nodes. The edges are being used for passing messages between two sets of nodes. Note that the full adjacency matrix of the Tanner graph has the form:

$$\mathbf{A} = \begin{bmatrix} 0 & \mathbf{H} \\ \mathbf{H}^\top & 0 \end{bmatrix} . \tag{3.2}$$

The set of all edges in the Tanner graph is defined by:

$$E \triangleq \left\{ \{c_i, v_j\} \,:\, \mathbf{H}_{i,j} \neq 0 \right\} , \tag{3.3}$$

or in terms of node indices:

$$\mathcal{E} \triangleq \{(i,j) \,:\, \{c_i, v_j\} \in E\}. \tag{3.4}$$

A degree distribution function describes a distrubution of degrees in a graph. Associated to Tanner graphs, an ensemble of irregular codes is specified by two degree distributions, $\lambda(x)$ and $\rho(x)$,

$$\lambda(x) = \sum_{i=2}^{d_{vmax}} \lambda_i x^{i-1}, \tag{3.5}$$

$$\rho(x) = \sum_{j=2}^{d_{cmax}} \rho_j x^{j-1}, \tag{3.6}$$

where λ_i is the fraction of edges that belongs to degree-i variable nodes, ρ_j is the fraction of edges that belong to degree-j check nodes, d_{vmax} is the maximum variable node degree, and d_{cmax} is the maximum check node degree.

3.4 Decoding

The decoding of LDPC codes is usually performed by iterative Belief Propagation (BP) algorithms [70], [76] which work on the Tanner graph [56]. There are two groups of algorithms - hard decision and soft decision, which performs better in terms of correction capabilities. The soft decision decoding is described in the following sections.

Sum-Product algorithm

The Sum-Product (SP) algorithm is the algorithm based on probabilistic iterative decoding of LDPC codes. The algorithm works in several steps : Initialization, Iterative process, and Termination.

The first step of the algorithm is the initialization, when the decoder receives a possibly corrupted codeword from a communication channel. During the iterative process, message passing through edges is being performed. Values sent from variable nodes to check nodes are denoted as q_{ij}, values outgoing from check nodes to variable nodes are denoted as r_{ij}. The algorithm is terminated when the corrected codeword is achieved or after a certain number of iterations is reached. If the algorithm is interrupted after the specific number of iterations, the decoding is considered as unsuccessful.

Two sets of node indices are given, $\mathcal{M}_j$ and $\mathcal{N}_i$. $\mathcal{M}_j$ is the set of all check node indices that are connected with the variable node j, while $\mathcal{N}_i$ is the set of all variable node indices that are connected with the check node i.

$$\begin{aligned}\mathcal{M}_j &\triangleq \{i : (i,j) \in \mathcal{E}\},\\ \mathcal{N}_i &\triangleq \{j : (i,j) \in \mathcal{E}\}.\end{aligned} \tag{3.7}$$

Consider the codeword sent into the channel to be denoted as $\mathbf{c}$ and the possibly corrupted message received from the channel as $\mathbf{y}$. Bits in these vectors are denoted as c_j and y_j, where $j = 1 \ldots n$. In the initialization step, conditional probabilities $P(c_j \mid y_j)$ are calculated and these values are sent to check nodes.

The conditional probabilities are defined as:

$$p_j(0) \triangleq P(c_j = 0 \mid y_j), \tag{3.8}$$

$$p_j(1) \triangleq P(c_j = 1 \mid y_j), \tag{3.9}$$

after the calculation, these values are sent to check nodes as messages q_{ij},

$$q_{ij}(0) = p_j(0),\ q_{ij}(1) = p_j(1). \tag{3.10}$$

The calculation of conditional probabilities for several channels, Binary Symmetric Channel (BSC), Binary Erasure Channel (BEC), Binary Symmetric Channel with Errors and Erasures (BSEC), and AWGN channel with Binary-Phase Shift Keying (BPSK), can be performed using the formulas below.

For BSC channel, we can express the initial messages as:

$$p_j(0) = \begin{cases} 1-p, & \text{for } y_j = 0 \\ p, & \text{for } y_j = 1 \end{cases}, \tag{3.11}$$

$$p_j(1) = 1 - p_j(0) = \begin{cases} p, & \text{for } y_j = 0 \\ 1-p, & \text{for } y_j = 1 \end{cases}, \tag{3.12}$$

where $p \in [0,1]$ is the crossover probability (probability of an error) and $y_j \in \{0,1\}$ is the received symbol.

For BEC, the probabilities are given as:

$$p_j(0) = \begin{cases} 0, & \text{for } y_j = 1 \\ 1, & \text{for } y_j = 0 \\ 1/2, & \text{for } y_j = \mathrm{e} \end{cases}, \tag{3.13}$$

$$p_j(1) = \begin{cases} 1, & \text{for } y_j = 1 \\ 0, & \text{for } y_j = 0 \\ 1/2, & \text{for } y_j = \mathrm{e} \end{cases}, \tag{3.14}$$

where $y_j \in \{0,1,\mathrm{e}\}$ is the received symbol.

Calculation of the probabilities of BSEC is the following:

$$p_j(0) = \begin{cases} 1-p, & \text{for } y_j = 0 \\ p, & \text{for } y_j = 1 \\ 1/2 & \text{for } y_j = \mathrm{e} \end{cases}, \tag{3.15}$$

$$p_j(1) = 1 - p_j(0) = \begin{cases} p, & \text{for } y_j = 0 \\ 1-p, & \text{for } y_j = 1 \\ 1/2 & \text{for } y_j = \mathrm{e} \end{cases}. \tag{3.16}$$

AWGN channel with BPSK modulation, which is often used in simulations:

$$p_j(0) = \frac{1}{1 + \mathrm{e}^{2y_j/\sigma^2}}, \tag{3.17}$$

$$p_j(1) = 1 - p_j(0) = \frac{1}{1 + \mathrm{e}^{-2y_j/\sigma^2}}, \tag{3.18}$$

where $y_j \in \mathbb{R}$ is the received symbol.

Algorithm 1 Message passing - Sum Product algorithm

```
1: procedure VALUES TO CHECK NODES
   ▷ First half on an iteration
   Input: p, r
   Output: q
2:     for all $j \in [\mathbf{0}, |\mathcal{M}|)$ do
3:         for all $i \in [\mathbf{0}, |\mathcal{N}|)$ do
4:             $q_{i,j}(0) = p_j(0)$
5:             $q_{i,j}(1) = p_j(1)$
6:             for all $i' \in \mathcal{M}_j \setminus i$ do
7:                 $q_{i,j}(0) = q_{i,j}(0) r_{i',j}(0)$
8:                 $q_{i,j}(1) = q_{i,j}(1) r_{i',j}(1)$
9:             end for
10:        end for
11:    end for
12: end procedure

13: procedure VALUES TO VARIABLE NODES
    ▷ Second half on an iteration
    Input: q
    Output: r
14:    for all $j \in [\mathbf{0}, |\mathcal{M}|)$ do
15:        for all $i \in [\mathbf{0}, |\mathcal{N}|)$ do
16:            $r_{i,j}(0) = 1$
17:            $r_{i,j}(1) = 1$
18:            for all $j' \in \mathcal{N}_i \setminus j$ do
19:                $r_{i,j}(0) = r_{i,j}(0)(1 - 2q_{i,j'}(1))$
20:            end for
21:            $r_{i,j}(0) = 1/2 + 1/2 r_{i,j}(0)$
22:            $r_{i,j}(1) = 1 - r_{i,j}(0)$
23:        end for
24:    end for
25: end procedure
```

The values of p_j are passed to check nodes as q_{ij} messages. After passing q_{ij},

Algorithm 2 Soft-decision decoding

```
1: procedure DecodeAWGN                                  ▷ SP algorithm
   Input: $\mathbf{y}$ – output from a demodulator
   ITERATIONS – maximum number of iterations
   $\sigma$ – variance of the channel
   Output: $\widehat{\mathbf{c}}$
2:     $\mathbf{q}$ =Initialize($\mathbf{p}, \sigma$)                      ▷ See Algorithm 3
3:     $\mathbf{r}$ =Values to Variable Nodes($\mathbf{q}$)               ▷ See Algorithm 1
4:     $\widehat{\mathbf{c}}$ =Calculate Estimation($\mathbf{r}$)                ▷ See Algorithm 4
5:     if $\widehat{\mathbf{c}}\mathbf{H}^\top = \mathbf{0}$ then   return $\widehat{\mathbf{c}}$
6:     end if
7:     for $it \in (0, \text{ITERATIONS})$ do
8:         $\mathbf{q}$ =Values to Check Nodes($\mathbf{r}$)              ▷ See Algorithm 1
9:         $\mathbf{r}$ =Values to Variable Nodes($\mathbf{q}$)           ▷ See Algorithm 1
10:        $\widehat{\mathbf{c}}$ =Calculate Estimation($\mathbf{r}$)            ▷ See Algorithm 4
11:        if $\widehat{\mathbf{c}}\mathbf{H}^\top = \mathbf{0}$ then   return $\widehat{\mathbf{c}}$
12:        end if
13:    end for
14: end procedure
```

Algorithm 3 Soft-decision decoding - initialization

```
1: procedure Initialize                                  ▷ Probabilities for AWGN
   Input: $\mathbf{y}$, $\sigma$
   Output: $\mathbf{q}$
2:     for all $y_j \in \mathbf{y}$ do
3:         $p_j = 1.0/(1 + \exp(-2y_j/\sigma^2))$
4:     end for
5:     for all $j \in [\mathbf{0}, |\mathcal{M}|)$ do
6:         for all $i \in [\mathbf{0}, |\mathcal{N}|)$ do
7:             $q_{i,j} = p_j$
8:         end for
9:     end for
10: end procedure
```

values r_{ij} are calculated and passed back to the variable nodes, as follows:

$$r_{ij}(0) = \frac{1}{2} + \frac{1}{2} \prod_{j' \in \{\mathcal{N}_i \setminus j\}} \left(1 - 2q_{ij'}(1)\right), \tag{3.19}$$

$$r_{ij}(1) = 1 - r_{ij}(0) \tag{3.20}$$

Algorithm 4 Soft-decision decoding - estimation

```
 1: procedure CALCULATE ESTIMATION
      Input: $\mathbf{p}$, $\mathbf{r}$
      Output: $\widehat{\mathbf{c}}$
 2:     for all $j \in [\mathbf{0}, |\mathcal{M}|)$ do
 3:         $Q_{i,j}(0) = p_j(0)$
 4:         $Q_{i,j}(1) = p_j(1)$
 5:         for all $i \in \mathcal{M}_j$ do
 6:             $Q_{i,j}(0) = Q_{i,j}(0) r_{i,j}(0)$
 7:             $Q_{i,j}(1) = Q_{i,j}(1) r_{i,j}(1)$
 8:         end for
 9:         if $Q_{i,j}(0) > Q_{i,j}(1)$ then    $\widehat{\mathrm{c}}_j = 0$
10:         else  $\widehat{\mathrm{c}}_j = 1$
11:         end if
12:     end for
13: end procedure
```

After the variable nodes receive the values of r_{ij}, a new estimation $\hat{\mathbf{c}}$ of the codeword $\mathbf{c}$ is calculated.

$$Q_j(0) = \mathrm{K_j} \cdot p(0) \cdot \prod_{i \in \mathcal{M}_j} r_{ij}(0), \tag{3.21}$$

$$Q_j(1) = \mathrm{K_j} \cdot p(1) \cdot \prod_{i \in \mathcal{M}_j} r_{ij}(1), \tag{3.22}$$

the constant $\mathrm{K_j}$ is chosen to satisfy $Q_j(0) + Q_j(1) = 1$,

$$\hat{c_j} = \begin{cases} 1, \text{ for } Q_j(1) > Q_j(0) \\ 0, \text{ otherwise} \end{cases} . \tag{3.23}$$

If $\hat{\mathbf{c}}$ is a codeword of the code (the product of $\hat{\mathbf{c}}\mathbf{H}^\top$ is equal to the zero vector), the decoding is stopped. If not, the decoding process continues with the next iteration. Then, values q_{ij} are calculated as follows:

$$q_{ij}(0) = \mathrm{K_{ij}} p(0) \prod_{i' \in \{\mathcal{M}_j \setminus i\}} r_{i'j}(0), \tag{3.24}$$

$$q_{ij}(1) = \mathrm{K_{ij}} p(1) \prod_{i' \in \{\mathcal{M}_j \backslash i\}} r_{i'j}(1), \tag{3.25}$$

where $\mathrm{K_{ij}}$ is the normalization constant to satisfy: $q_{ij}(0) + q_{ij}(1) = 1$.

These values are sent to check nodes and the second half of the iteration continues to Eq. 3.19 and 3.20.

Log-domain decoding

In order to decrease the number of multiplications being used in the decoding process, the log-domain decoding has been proposed. Assume the surjective function L,

$$L: \; L\left(x, y\right) = \ln \frac{x}{y}. \tag{3.26}$$

Consider this function applied on initial probabilities and messages being passed through the Tanner graph,

$$L(p_j) \triangleq L(p_j(0), p_j(1)) = \ln \frac{p_j(0)}{p_j(1)}, \tag{3.27}$$

$$L(r_{ij}) \triangleq L(r_{ij}(0), r_{ij}(1)) = \ln \frac{r_{ij}(0)}{r_{ij}(1)}, \tag{3.28}$$

$$L(q_{ij}) \triangleq L(q_{ij}(0), q_{ij}(1)) = \ln \frac{q_{ij}(0)}{q_{ij}(1)}. \tag{3.29}$$

For AWGN channel, the initial probability is given by:

$$L(p_j) = -\frac{2y_i}{\sigma^2}. \tag{3.30}$$

For Rayleigh fading channel, the initial probabilities can be expressed as (e. g. [28]):

$$L(p_j) = -\frac{2y_i}{\sigma^2}\alpha, \tag{3.31}$$

where α is the normalized Rayleigh fading factor with $E\left[\alpha^2\right] = 1$ and probability density function $p(\alpha) = 2\alpha \mathrm{e}^{-\alpha^2}$, and $y_j \in \mathbb{R}$ is the received symbol.

If no channel state information is available, we can calculate with the approximation:

$$L(p_j) = -\frac{2y_i}{\sigma^2}E[\alpha]. \tag{3.32}$$

It can be derived that $L(r_{ij})$ values are determined as:

$$\phi(x) \triangleq -\ln\left(\tanh\frac{x}{2}\right) = \ln\frac{e^x+1}{e^x-1}, \tag{3.33}$$

$$L(r_{ij}) = \prod_{j'\in\{\mathcal{N}_i\backslash j\}} \text{sign}\left(L(q_{ij'})\right)\cdot\phi\left(\sum_{j'\in\{\mathcal{N}_i\backslash j\}}\phi\left|L(q_{ij'})\right|\right). \tag{3.34}$$

The precise calculation of ϕ can consume a prohibitively long time. In order to decrease the number of operations, the following simplification is often applied:

$$L(r_{ij}) \approx \prod_{j'\in\{\mathcal{N}_i\backslash j\}} \text{sign}\left(L(q_{ij'})\right)\cdot\min\left|L(q_{ij'})\right|. \tag{3.35}$$

The algorithm using the equation above is usually referred as the Min-Sum algorithm [70].

Values of $L(q_{ij})$ are calculated by the summation:

$$L(q_{ij}) = L(p_j) + \sum_{i'\in\{\mathcal{M}_j\backslash i\}} L(r_{ij}), \tag{3.36}$$

and the estimation $\hat{\mathbf{c}}$ is given by:

$$L(Q_j) = L(p_j) + \sum_{i\in\mathcal{M}_j} L(r_{ij}), \tag{3.37}$$

$$\hat{c_j} = \begin{cases} 1, \text{ for } L(Q_j) < 0 \\ 0, \text{ otherwise} \end{cases}. \tag{3.38}$$

Sum-Product algorithm for non-binary codes

The SP algorithm for non-binary LDPC codes was generalized by Davey's and Mackay's work [14]. Their non-binary LDPC codes [15] were the first LDPC codes that surpassed turbo codes at decoding performance.

The values of $r_{ij}(a)$ are calculated as follows:

$$r_{ij}(a) = \sum_{\mathbf{c}\,:\,c_j=a\,\wedge\,\mathbf{c}\cdot\mathbf{h}_i^\top=0} \left(\prod_{j'\in\{\mathcal{N}_i\backslash j\}} (q_{ij'}(c_{j'})) \right), \tag{3.39}$$

which is the sum over all codewords which have the symbol a at the position j and satisfy the i-th parity check, and $\mathbf{c} \in \mathcal{C}$. The probability that the symbol at j-th position is equal to a is determined by this formula. The vector $\mathbf{h}_i$ denotes the i-th row of the parity-check matrix.

The values of $q_{ij}(a)$, which are being passed from the variable to the check nodes, are determined by:

$$q_{ij}(a) = \mathrm{K_{ij}} \cdot p(a) \cdot \prod_{i'\in\{\mathcal{M}_j\backslash i\}} r_{i'j}(a), \tag{3.40}$$

The constant $\mathrm{K_{ij}}$ is chosen to satisfy $\sum\limits_{a\in\mathcal{A}} q_{ij}(a) = 1$.

The estimation is realized by:

$$\hat{x_j} = \underset{a}{\operatorname{argmax}} \left(p(a) \prod_{i\in\{\mathcal{M}_j\}} r_{ij}(a) \right). \tag{3.41}$$

The initial probabilities are calculated proportional to the channel properties. For the AWGN channel using BPSK modulation, it is:

$$p_j(a) = p(y_j \,|\, c_j = a) \propto \prod_{i=1}^{k} \mathrm{e}^{\frac{((y_j\,\mathrm{BIN})_k - (-1)^{t_i})^2}{2\sigma^2}}, \tag{3.42}$$

t_i^a is chosen with respect to a and i.

Adaptive decoding, modifications

One of the most perspective methods of improving the correction performance of the decoder is the usage of an adaptive technique of decoding, firstly presented in [29], and tested on LDPC codes by Mobini in 2011 [40]. In this method, the outgoing messages are replaced by:

$$q'_{ij} = \alpha q_{ij} \mathrm{e}^{-\beta|q_{ij}|}. \tag{3.43}$$

where $\alpha \in (0, 1]$ and $\beta \in [0, \infty)$ are chosen by solving an optimization task for particular codes.

Gallager's construction

This is is the original Gallager's construction of regular **H** matrix. The weight of row vectors of the parity-check matrix is denoted as k, the weight of column vectors of the parity-check matrix is denoted as j, and the length of a codeword is denoted as n.

Considering the matrix $\mathbf{A}_0$,

$$\mathbf{A}_0 = \begin{bmatrix} 1 & 1 & 1 & 1 & & & & & & & & & & & & \\ & & & & 1 & 1 & 1 & 1 & & & & & & & & \\ & & & & & & & & 1 & 1 & 1 & 1 & & & & \\ & & & & & & & & & & & & 1 & 1 & 1 & 1 \end{bmatrix} \tag{3.44}$$

of the dimension n/k by n, the matrix **H** is then given as:

$$\mathbf{H} = \begin{bmatrix} \pi_1(\mathbf{A}_0) \\ \pi_2(\mathbf{A}_0) \\ \vdots \\ \pi_j(\mathbf{A}_0) \end{bmatrix} \tag{3.45}$$

where π_i are random column permutations of $\mathbf{A_0}$. However, irregular forms of **H** matrices perform better under the SP decoding. Some construction methods are summarized in the following chapter and methods based on large girth construction and genetic optimization algorithms are proposed in this work.

Chapter 4

Construction of LDPC codes

Numerous studies have been focused on construction algorithms of LDPC codes during past decades. In general, these algorithms create the parity-check matrices, which define LDPC codes and associated Tanner graphs. The construction of a parity-check matrix can be performed either in structured (e.g. geometry based methods) and unstructured ways (e.g. PEG [73] or by Gallager's construction (Section 3.4)).

Kou, Lin and Fossorie summarized the construction of LDPC codes based on finite geometries [30] in 2001. Large-girth construction based on graphical models was presented in work of Zhang and Moura [81] in 2003. Later, in 2004, Moura, Lu and Zhang summarized structured construction methods [43] and Vasic and Milenkovic introduced several combinatorical construction methods in [65]. Significant family of structured codes are Quasi-Cyclic (QC) LDPC codes [21], constructed by Fossorier in 2004.

In 2001, Richardson, Shokrollahi and Urbanke introduced a powerfull construction algorithm working with degree distributions in the Tanner graph [50]. In 2003, the algorithm using a new metric called Extrinsic Message Degree (EMD) in order to avoid short cycles was proposed by Tian, Jones, Villasenor and Wessel [58].

The Progressive Edge Growth (PEG) algorithm, firstly presented in 2001 [73] a summarized in 2005 [74], outperformed codes constructed by the methods above. Several modifications of PEG was published [84], [63]. Good degree distributions might by designed using the density evolution method [33], [13], [51] and passed as an input of the PEG algorithm.

This chapter proposes methodologies for designing LDPC codes and associated Tanner graphs of large girths, and direct application of genetic optimization algorithms.

4.1 Generating LDPC matrices with large girth

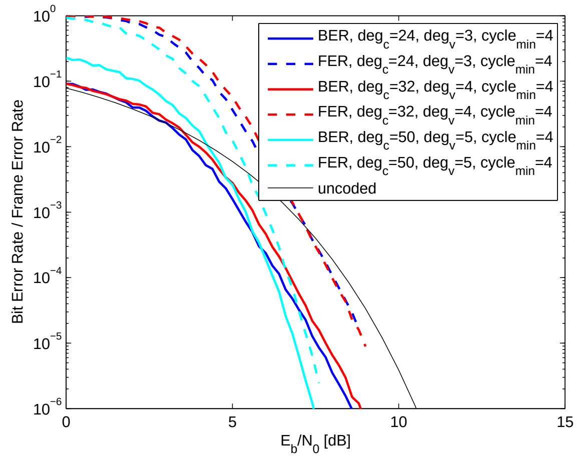
The proposed algorithm for generating Tanner graphs is based on sequential usage of Breadth-First Search (BFS) and addition of edges with avoidance of short cycles. The diversity of candidate solutions can be tuned by parameters σ_v, σ_c, and cycle lengths, as can be seen in Algorithm 5.

The algorithm progressively iterates over nodes and calculates distances (denoted as $d(c, v)$) between nodes. If any distance is higher than a given *cycle* parameter and a degree is lower than the desired degree, a new edge is added between the nodes c and v. The parameter *cycle* is being changed from the value $cycle_{start}$ to $cycle_{stop}$ and it is decreasing.

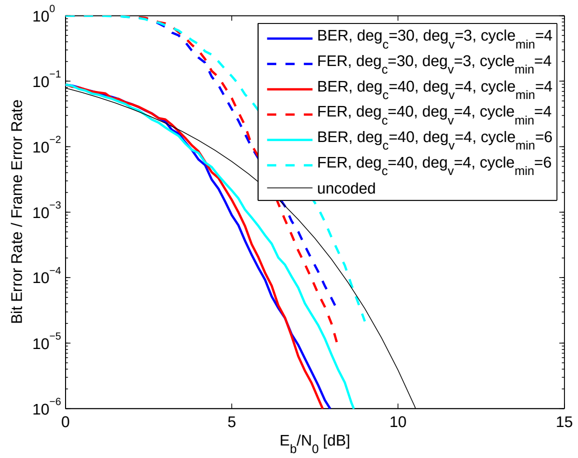
The proposed algorithm can be used solely for a construction of large girth graphs or as a part or genetic optimization tasks. The performance evaluation of codes produced by this algorithm is also presented. For genetic optimization tasks, this algorithm is used for producing the initial population, as described in the following sections.

Algorithm 5 Large girth graph construction

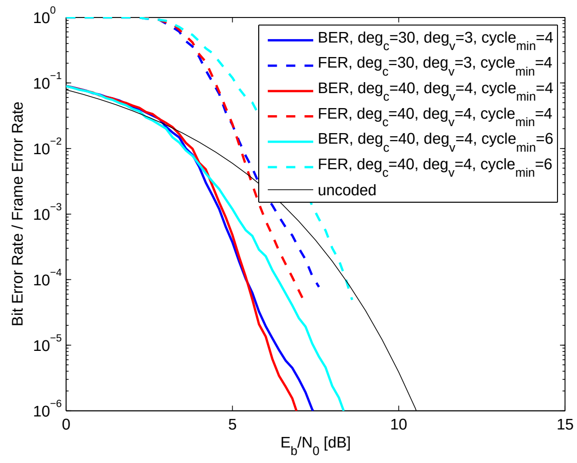
```
1: procedure GENERATE GRAPH
    Input: $v_{req}$ - requested variable nodes,
   $c_{req}$ - requested check nodes,
   $cycle_{start}$ - length of the longest cycle in the first iteration
   $cycle_{stop}$ - length of the longest cycle in the last iteration
   $d_{v0}$ - design degree of the variable nodes
   $d_{c0}$ - design degree of the check nodes
   $\sigma_v$ - standard deviation of variable node degrees
   $\sigma_c$ - standard deviation of check node degrees
    Output: (CN,VN)-bipartite graph $G$
2:     for $cycle = cycle_{start}$ downto $cycle_{stop}$ do
3:         $d_v = d_{v0} + \sigma_v r, r \in \mathcal{N}(0,1)$
4:         $d_c = d_{c0} + \sigma_c r, r \in \mathcal{N}(0,1)$
5:         for all $v \in VN$ do
6:             for all $c \in CN$ do
7:                 if $deg(v) < d_v$ and $deg(c) < d_c$ then
8:                     if $d(c,v) > cycle$ then
9:                         add $e = (c,v)$ to the set $E$
10:                    end if
11:                end if
12:            end for
13:        end for
14:    end for
15:    for all $v \in VN$ do
16:        if $deg(v) < 2$ then
17:            $e = (c : (c \in G, \max\{k|v \to^k c\}), v)$
18:            add $e$ to the set $E$
19:        end if
20:    end for
21:    for all $v \in CN$ do
22:        if $deg(c) < 2$ then
23:            $e = (c, v : (v \in G, \max\{k|c \to^k v\}))$
24:            add $e$ to the set $E$
25:        end if
26:    end for
27: end procedure
```



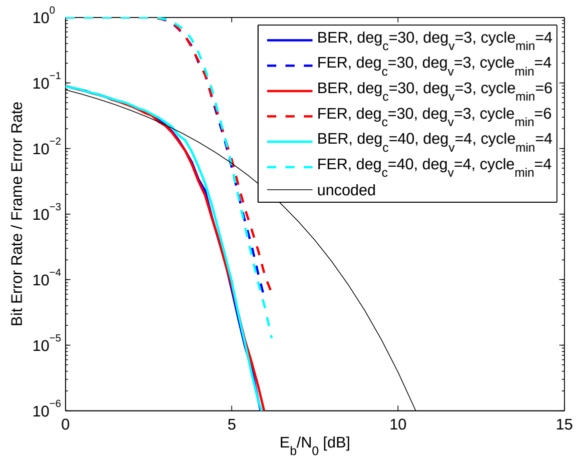
(a) LDPC (64,56).



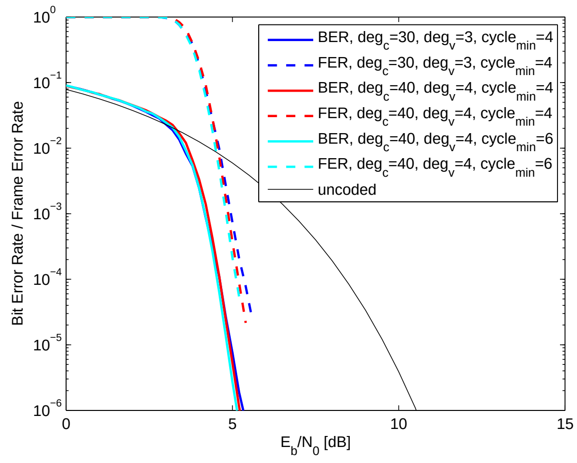
(b) LDPC (128,115).



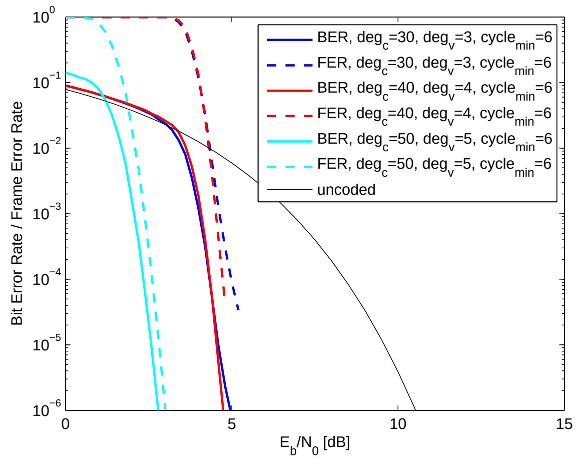
(c) LDPC (256,230).



(d) LDPC (512,461).

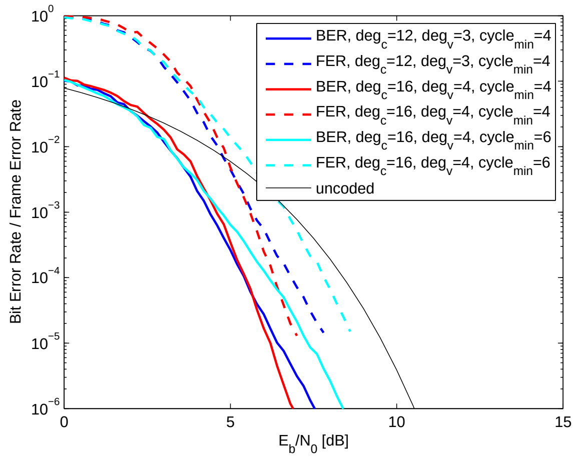


(e) LDPC (1024,922).

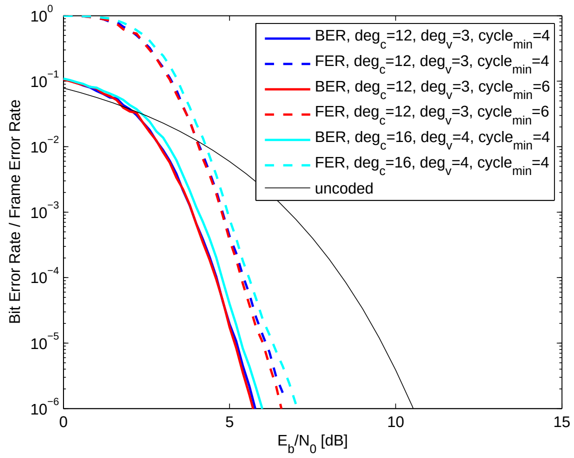


(f) LDPC (2048,1843).

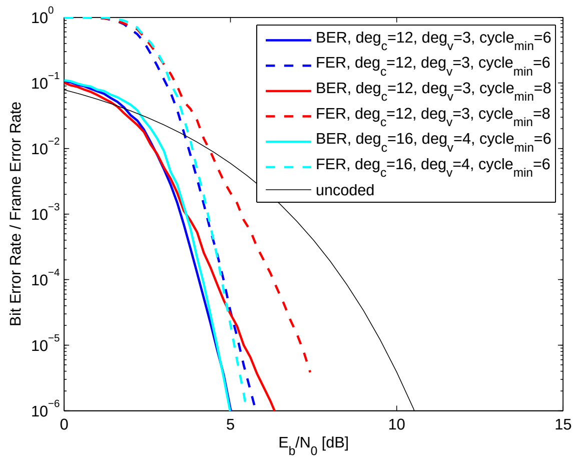
Figure 4-1: Correction performance of 10% redundancy codes (12.5% for n=64).



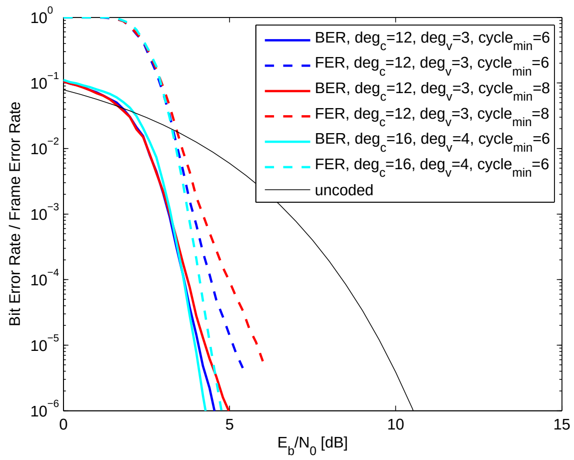
(a) LDPC (64,48).



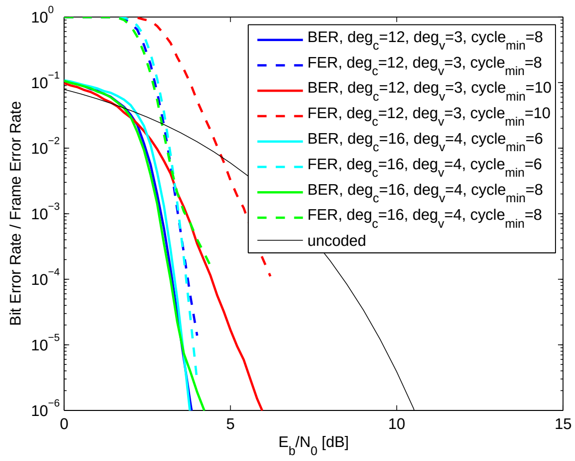
(b) LDPC (128,96).



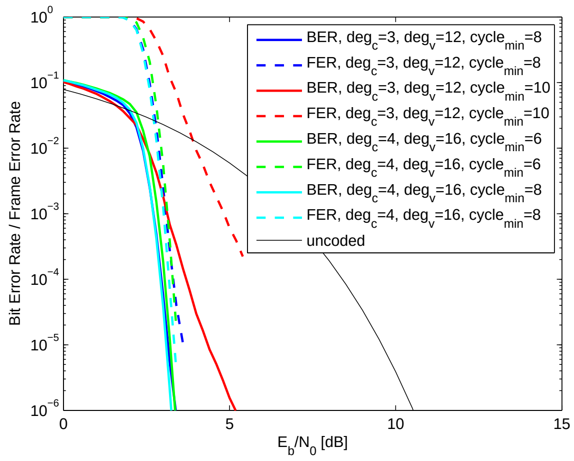
(c) LDPC (256,192).



(d) LDPC (512,384).

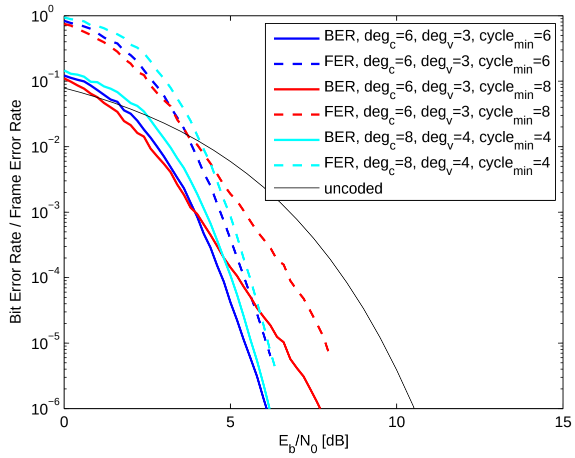


(e) LDPC (1024,768).

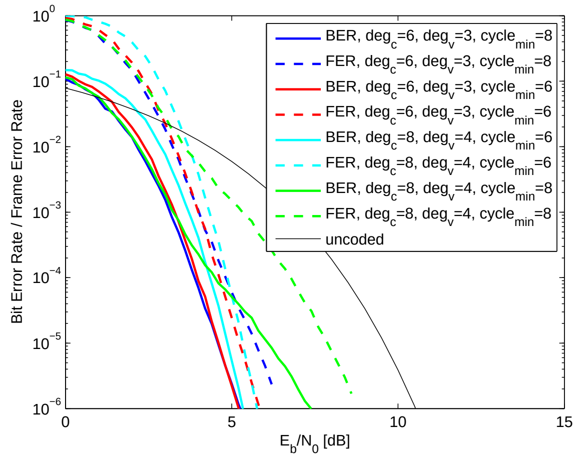


(f) LDPC (2048,1536).

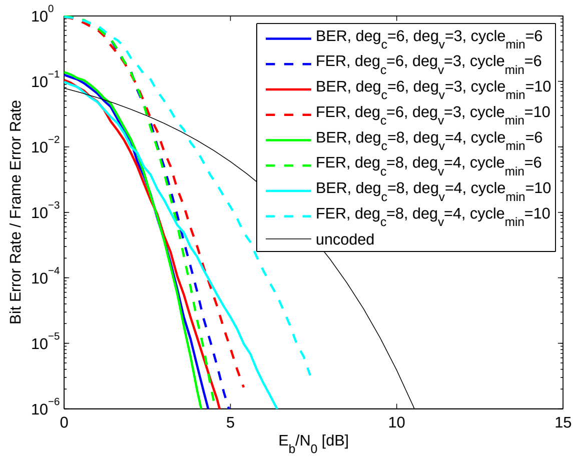
Figure 4-2: Correction performance of 25% redundancy codes.



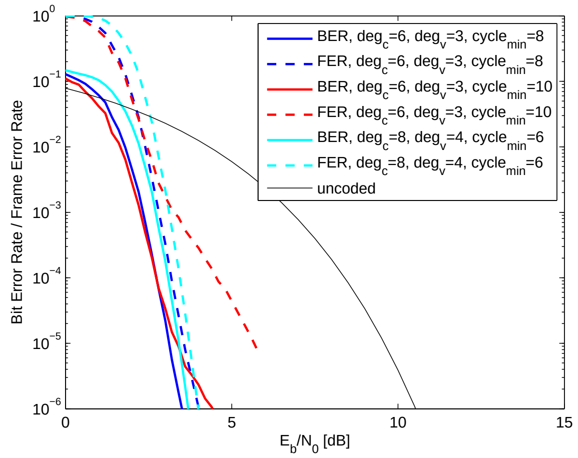
(a) LDPC (64,32).



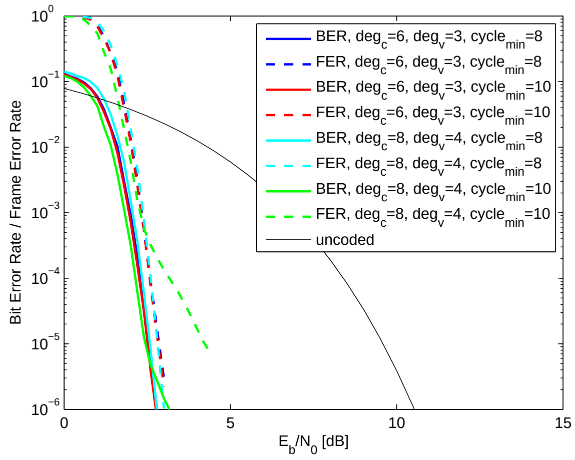
(b) LDPC (128,64).



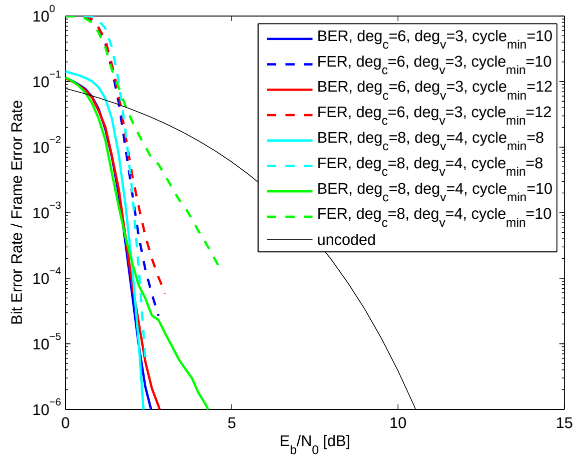
(c) LDPC (256,128).



(d) LDPC (512,256).

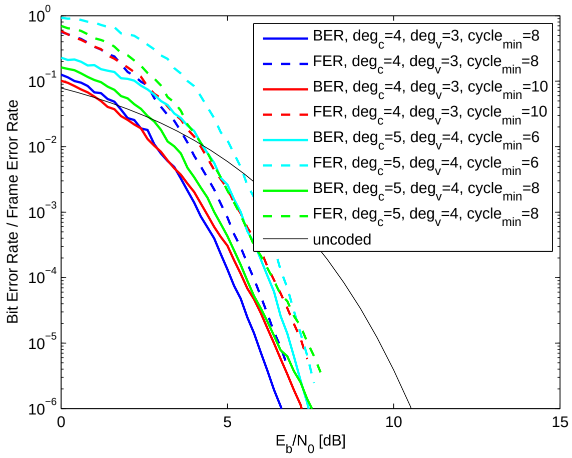


(e) LDPC (1024,512).

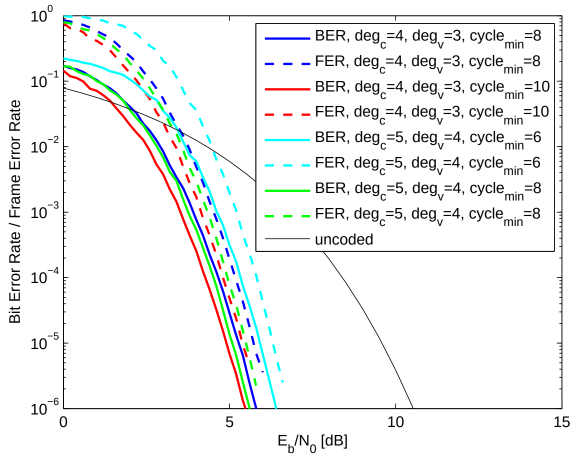


(f) LDPC (2048,1024).

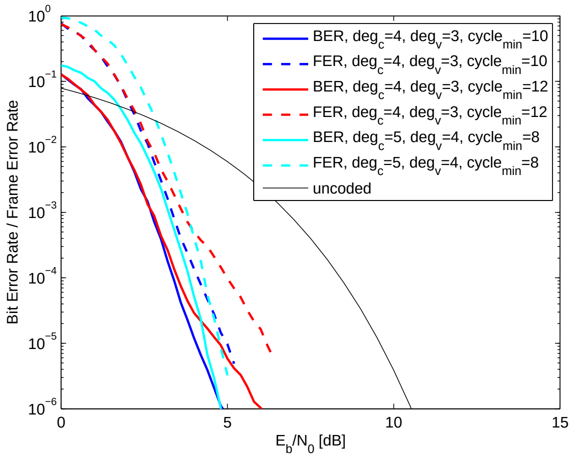
Figure 4-3: Correction performance of 50% redundancy codes.



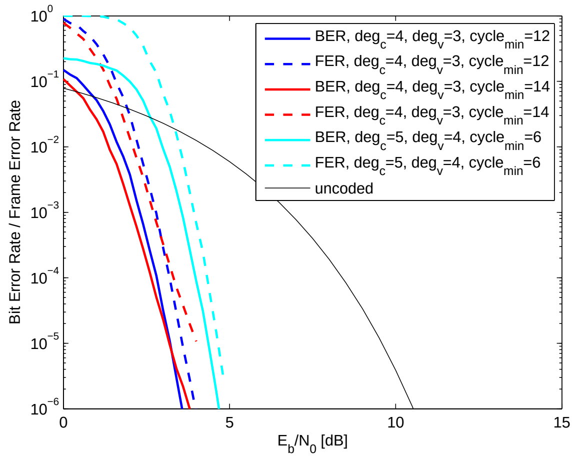
(a) LDPC (64,16).



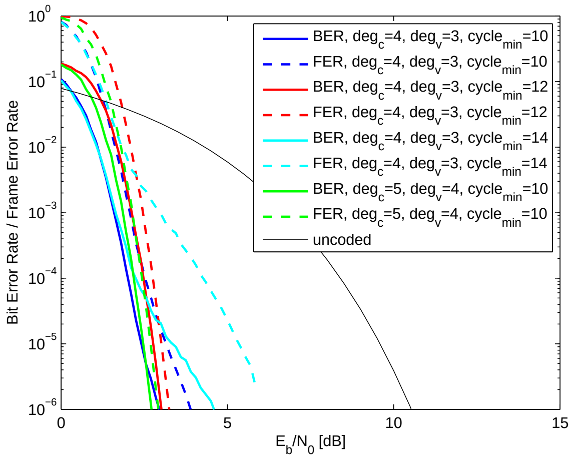
(b) LDPC (128,32).



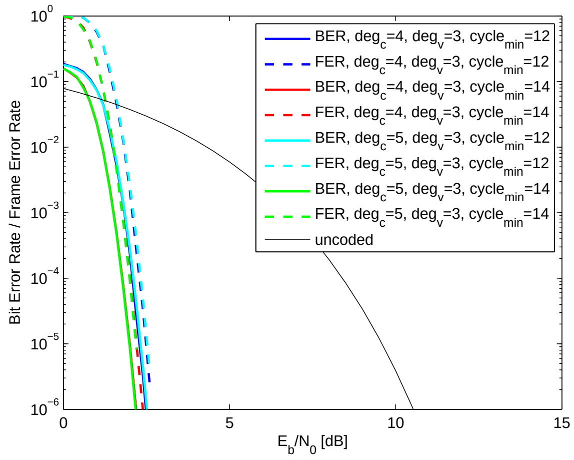
(c) LDPC (256,64).



(d) LDPC (512,128).



(e) LDPC (1024,256).



(f) LDPC (2048,512).

Figure 4-4: Correction performance of 75% redundancy codes.

4.2 Genetic optimization of LDPC codes

4.2.1 Optimization algorithm principle

Considering a linear block code $\mathcal{C} = (n, k)$ as a candidate solution of an optimization problem and the objective function $f_0 : \mathcal{C} \to \mathbf{R}$, the goal is to find the optimal solution C^* such that $f_0(C^*) \leq f_0(C)$ as a local minimum $||C - C^*|| \leq \delta$, where δ determines the subspace size. According to the parameters of the algorithm and the optimization problem, the local minimum can be equal to the global minimum. A constraint function $g_i(\mathcal{C}) \leq b_i, i = 1, ..., m$ can be also considered.

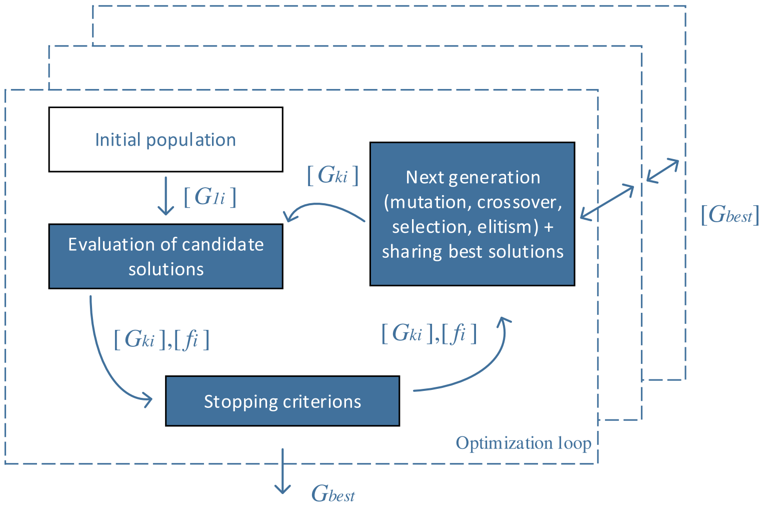


Figure 4-5: A diagram of the optimization process.

The genetic optimization algorithm requires generating the initial population, mutation and crossover operators, and methods used for selection and evaluation candidate solutions. The proposed operators and methods are described in the folllowing sections.

4.2.2 Tanner graph mutations

For reproducing the next generations of candidate solutions in the optimization process, operators performing mutations or crossover are needed. A mutation of the Tanner graph is reproduced with the use of the following operations:

- Addition of R random edges,
- Erasure of R random edges,
- Addition of an edge connected to a chosen node $u \in G$ and some node $v \in G$: $d(u, v) > M$ and $d(u, v) \equiv 1 \pmod 2$ (connection of two distant nodes),
- Erasure of an edge in order to break a cycle of length $l < Q$; the example can be seen in Fig. 4-6,
- Addition of an edge between nodes $u, v \in G$: $(d_G(u) = 1 \wedge d_G(v) = 1)$ and $d(u, v) \equiv 1 \pmod 2$ (connection of nodes with a degree equal to one),
- Addition of an edge between nodes $u, v \in G : d_G(v) = 1$ and $d(u, v) = \max\{d(u, v) | d(u, v) \equiv 1 \pmod 2\}$; the example can be seen in Fig. 4-7,

where G is the Tanner graph, E are edges of the graph G, and parameters R, M, Q are chosen randomly with regard to the Tanner graph size (it is convenient to tune them experimentally for each optimization task). The operation performed for reproducing one mutant is also chosen randomly. The operator works with graph search algorithms for finding cycles in the Tanner graph.

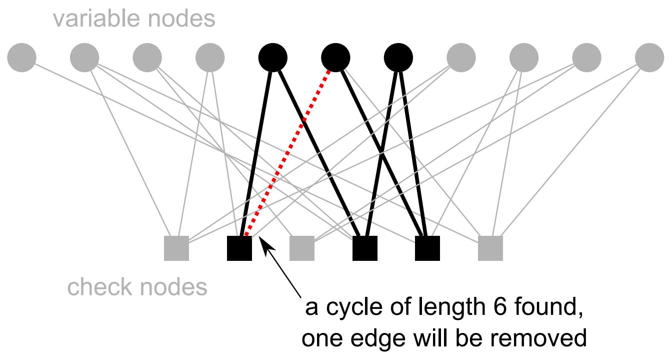


Figure 4-6: Breaking cycles in the Tanner graph as an example.

4.2.3 Recombination of Tanner graphs

Supposing (VN, CN) - bipartite graph G as the Tanner graph of the LDPC code $\mathcal{C}$, a coloured subgraph H^β (called the chromosome), such $H \subseteq G$, we define algorithms

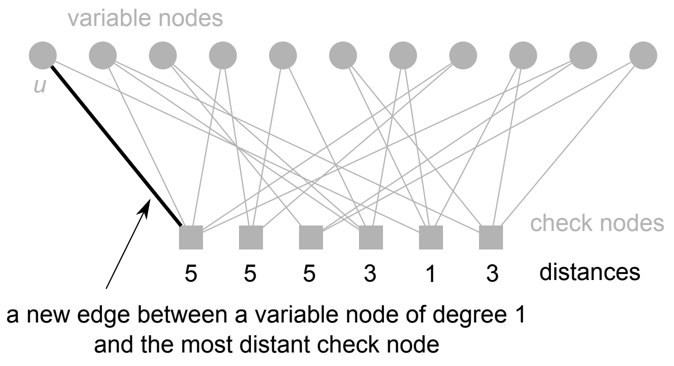

variable nodes
u
check nodes
5 5 5 3 1 3 distances
a new edge between a variable node of degree 1 and the most distant check node

Figure 4-7: Adding a new edge connected with the node $u : d_G(u) = 1$ as an example.

to combine chromosomes H_i^β in order to create a new Tanner graph G_{TG},

$$G_{TG} = \left(U = \bigcup U_i, E = \left\{\bigcup E_i, \{E_j\}\right\}\right), \tag{4.1}$$

where E_i are edges of chromosomes H_i, E_j are edges ensuring that graph G_{TG} is connected. U_i are node chromosomes.

A vertex colouring function α,

$$\alpha : v \in G \rightarrow [0, 1], \tag{4.2}$$

is used to mark nodes which are included in the chromosome H. The algorithm implementing the vertex colouring works with 3 parameters, v_{req} (the number of requested variable nodes), c_{req} (the number of requested check nodes), and v_{start} (the node where the colouring function starts). The nodes of the graph G are selected with the use of the Breadth-First Search (BFS) starting at v_{start} and terminated when the number of selected nodes reaches the values v_{req} and c_{req}, as can be seen in Algorithm 6. The algorithm partly keeps the degree distribution and other parameters, e. g. cycle occurrence, of the graph G. The reproduced chromosome H^β uses vertex colouring function β,

$$\beta : v \in H \rightarrow [0, \max(d_{vmax}, d_{cmax})], \tag{4.3}$$

where d_{cmax} is the maximum check node degree and d_{vmax} is the maximum variable node degree, as the difference between the original node degree and the node degree in the chromosome H, as defined in Algorithm 7. These values are then applied in the recombination function for edge addition to connect the input chromosomes H_i among each other, as described in Algorithm 8. The recombination function reproduces a new Tanner graph G_{TG}, consisting of chromosomes $H_{set} = \{H_i^\beta\}$, $i = 1, ..., k$.

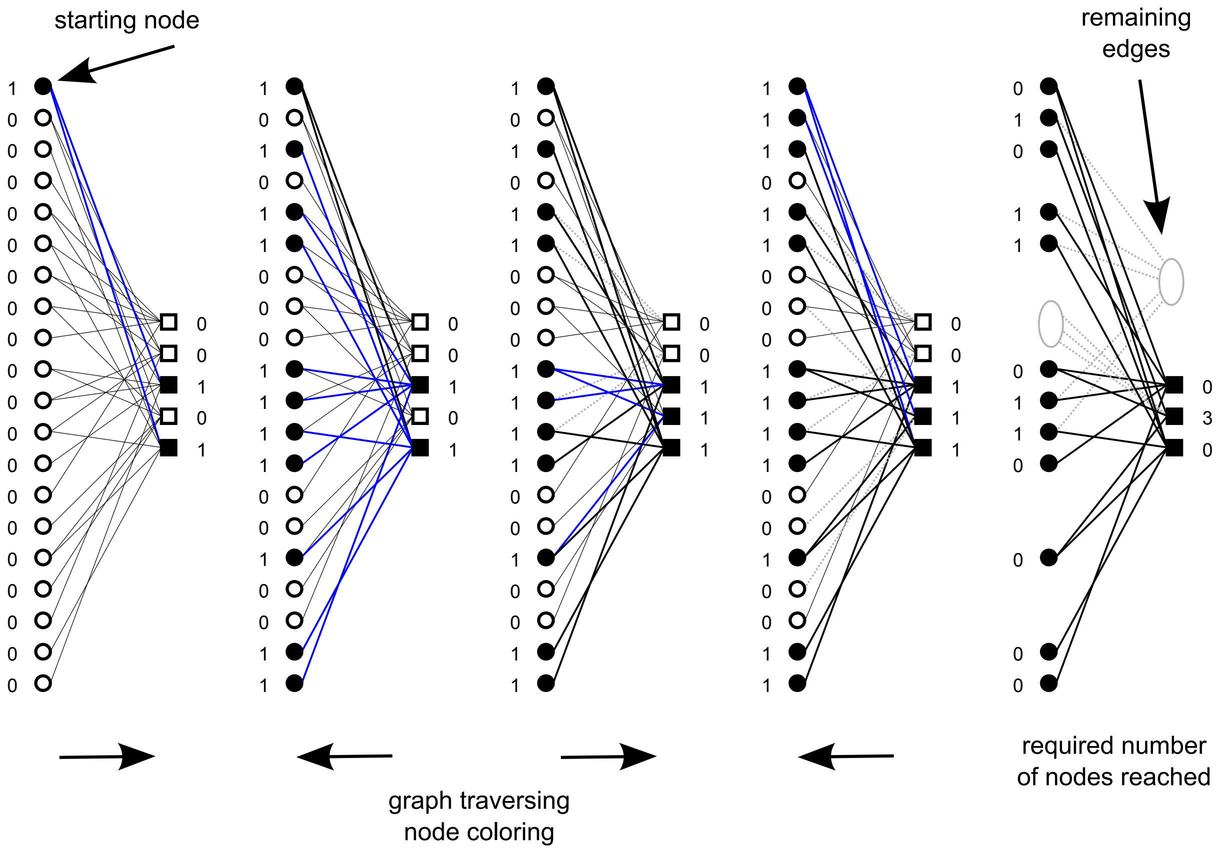
The colouring algorithm implements BFS in order to partly keep the distribution of cycles in the resulting chromosome H. By combining several graphs together in the optimization process, we can get codes with interesting correction capabilities measured by the fitness function.

Algorithm 6 Node colouring

```
1: procedure COLOUR NODES
     Input: $v_{req}$ - requested variable nodes,
     $c_{req}$ - requested check nodes,
     $(VN, CN)$ - bipartite graph $G$,
     $v_{start} \in G$ - starting node for traversing
     Output: coloured graph $G^\alpha$
2:     for all $u \in G$ do
3:         $\alpha(u) = 0$
4:     end for
5:     $\alpha(v_{start}) = 1$
6:     for $k = 0$ to $(v_{req}+c_{req})$ do
7:         for all $u \in \{v | v_{start} \rightarrow^k u\}$ do
     ▷ Implemented via BFS algorithm
8:             if $u \in VN$ and $|\{u|\alpha(u) = 1\}| < v_{req}$ then
9:                 $\alpha(u) = 1$
10:            end if
11:            if $u \in CN$ and $|\{u|\alpha(u) = 1\}| < c_{req}$ then
12:                $\alpha(u) = 1$
13:            end if
14:            if $|\{u \in CN|\alpha(u) = 1\}| = c_{req}$
                   and $|\{u \in VN|\alpha(u) = 1\}| = v_{req}$ then
15:                return $(G, \alpha)$
16:            end if
17:        end for
18:    end for
19: end procedure
```

Algorithm 7 Generating chromosome

```
1: procedure CHROMOSOME GENERATION
     Input: $(VN, CN)$-bipartite coloured graph $G^\alpha$
     Output: chromosome graph $H^\beta$
2:     $U_H = \{\forall u \in G^\alpha : \alpha(u) = 1\}$
3:     $E_H = \{\forall e = (u, v) \in G^\alpha : \alpha(u) = 1 \bigwedge \alpha(v) = 1\}$
4:     for all $u \in U_H$ do
5:         $\beta(u) = |\{(u, v) : v \in G^\alpha, \alpha(u) = 1 \bigwedge \alpha(v) = 0\}|$  ▷ number of unconnected edges
6:     end for
     return $((U_H, E_H), \beta)$
7: end procedure
```



starting node
remaining edges
graph traversing node coloring
required number of nodes reached

Figure 4-8: Generating the chromosome. Numbers are node degress corresponding to the selected part of the graph.

4.2.4 Fitness function

The fitness function is used for evaluating candidate solutions (a measure of correction capabilities) during the optimization process. Several types of fitness functions are

Algorithm 8 Combining chromosomes

```
1: procedure COMBINE CHROMOSOMES

    Input: $H_{set} = \{H_1^\beta, H_2^\beta, ..., H_k^\beta\}$ coloured graph, $d_{min}$
    Output: bipartite graph $G_{TG}$
2:      $U_{TG} = \{U_1, U_2, ..., U_k : U_i \in H_i^\beta\}$
3:      $E_{TG} = \{E_1, E_2, ..., E_k : E_i \in H_i^\beta\}$
4:      $G_{TG} = (U_{TG}, E_{TG})$                          ▷ new Tanner Graph
5:      while $\exists v \in VN : \beta(v) > 0$ do
6:          $k = \beta(v)$                                   ▷ node $v$ chosen randomly
7:          for $i = 0$ to $k$ do
8:              if $\exists c : \beta(c) > 0, c \in CN$ then
9:                  $e = (c, v : (\beta(c) > 0, \max\{k|v \to^k c\}))$
10:                 $\beta(c) = \beta(c) - 1$
11:             else
12:                 $e = (c, v : (c \in CN, \max\{k|v \to^k c\}))$
13:             end if
14:             $\beta(v) = \beta(v) - 1$
15:             add $e = (c, v)$ to the set $E_{TG}$
16:         end for
17:     end while
18:     while $\exists c \in CN : \beta(c) > 0$ do
19:         $k = \beta(c)$
20:         for $i = 0$ to $k$ do
21:             if $\exists v : \beta(v) > 0, v \in VN$ then
22:                 $e = (c, v : (\beta(v) > 0, \max\{k|c \to^k v\}))$
23:                 $\beta(v) = \beta(v) - 1$
24:             else
25:                 $e = (c, v : (v \in VN, \max\{k|c \to^k v\}))$
26:             end if
27:             $\beta(c) = \beta(c) - 1$
28:             add $e = (c, v)$ to the set $E_{TG}$
29:         end for
30:     end while
         return $(U_{TG}, E_{TG})$
31: end procedure
```

considered in this work,

- required E_b/N_0 to achieve the specific Bit Error Rate,
- correction capability for specific error patterns (e. g. bursts),
- correction capability for specific number of corrupted bits in the codeword.

Considering a Bit Error Rate (BER) simulation in a discrete sequence of E_B/N_0 with the common difference, which starts at a given value and terminates when the BER is lower than a given THRESHOLD value, the fitness value is calculated as the following:

$$f = \left|\frac{E_B}{N_0}\right| + BER\left(\left|\frac{E_B}{N_0}\right|\right) \quad . \tag{4.4}$$

where $\left|\frac{E_B}{N_0}\right|$ is the length of the sequence. The computational time increases with a lower THRESHOLD value and also with the precision of the BER simulation.

For burst optimization tasks, a burst error pattern is used as an optimization parameter. It is given by $\mathbf{E_P} = (x_1, x_2, ..., x_k)$, where x_i are lengths of error bursts in the codeword and $|\mathbf{E_P}|$ is the number of burst errors. The total number of corrupted bits is then

$$\sum_{i=1}^{k} E_P(i), \tag{4.5}$$

where $E_P(i) = x_i$.

The fitness function checks all combinations of error bursts given by burst lengths as elements of E_P and calculates the number of corrupted codewords after the correction. The product of the sum or the ratio of corrupted codewords to all possible combinations of error patterns is then used as the evaluation value. It should be noted that this method also evaluates all combinations of error bursts given by the sum of all possible combinations of $x_i \in E_P$ (e. g. all possible error burst combinations for $E_P = (3, 2)$ are also burst errors of length 5). This type of evaluation does not include any stochastic process and it is faster than the previous one.

Another type of evaluation considered in this work is the calculation of correction performance for a given sequence of corrupted bits. Then, error patterns of given

Hamming weights are randomly generated. Usually, an interval of Hamming weights (the number of corrupted bits) where the correction performance changes rapidly can be found,

$$f = \sum_{i=\mathrm{e}rr_{\min}}^{\mathrm{e}rr_{\max}} 2^{i-\mathrm{e}rr_{\min}} N(i), \tag{4.6}$$

where $\mathrm{e}rr_{\min}$ is the minimal number of corrupted bits, $\mathrm{e}rr_{\max}$ is the maximal number of corrupted bits and $N(i)$ is a measure of the correction performance at i corrupted bits (e. g. the number of codewords failed in correction when the total number of tested codewords is the same for any i).

4.2.5 Parallelization

In this section, the application of parallelization for accelerating the optimization convergency is presented. The algorithm of coarse grained parallelism, also known as the island model [55], [71], uses several computing threads (denoted as L) for running genetic optimization processes. These threads share their best solutions among each other. During every certain time period of generations (denoted as K), an exchange of the best solutions is made among threads performing the genetic optimization. This method significantly improves the convergence speed of the proposed optimization process. However, the improvement depends on both the number of computing threads and the period of generations. In the implementation, all optimization threads regularly send their best solutions to the thread responsible for the maintenance. When a certain period of generation is reached, each thread sends a query for a code created from the combination of all best solutions. After receiving the best solution, the solution is included in the optimization process performed by the particular thread. The principle is illustrated in Fig. 4-10.

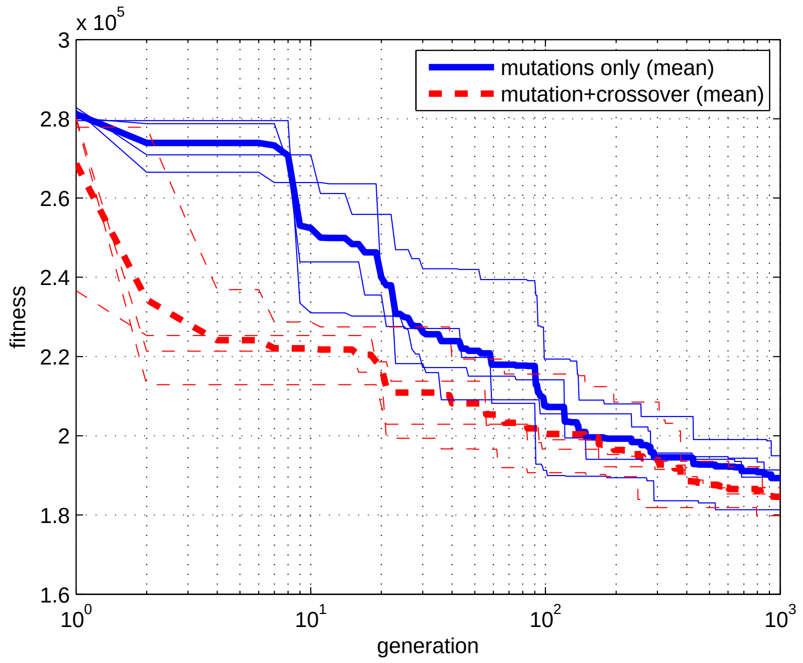
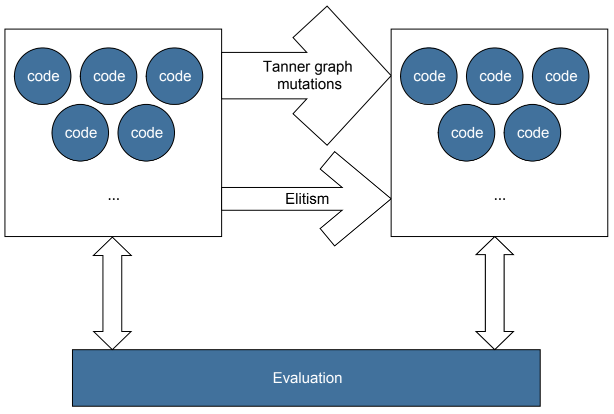
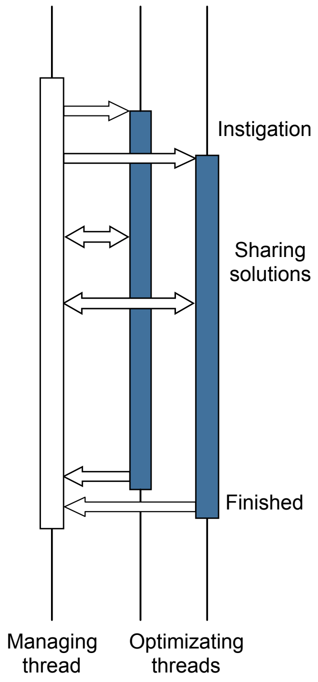


Figure 4-9: A demonstrative evolution of the fitness function. The optimization using only mutations is compared to the optimization using the combination of the mutations and crossover operators.



(a) Creating a new generation.



(b) Thread instigation process.

Figure 4-10: The principle of the optimization algorithm.

4.3 Application of optimization tasks

4.3.1 Sample optimization using mutations

An optimization of LDPC (128,64) code is considered. The LDPC (128,64) code was optimized for the best performance on AWGN channel. The optimization were running for 10 generations with 7 codes in the population and used the elitism concept, as can be seen in Fig. 4-11. The best improvement of E_B/N_0 is 1,15 dB (listed in Table 4.1). The same experiment was running in 76 independent simulations. The fitness function used for evaluating was based on finding the minimum BER threshold, as described in Eq. 4.4 on page 52. The initial set contained the population of 30 codes, produced according to Algorithm 5. Various plots regarding the BER and fitness values are depicted in Fig. 4-12.

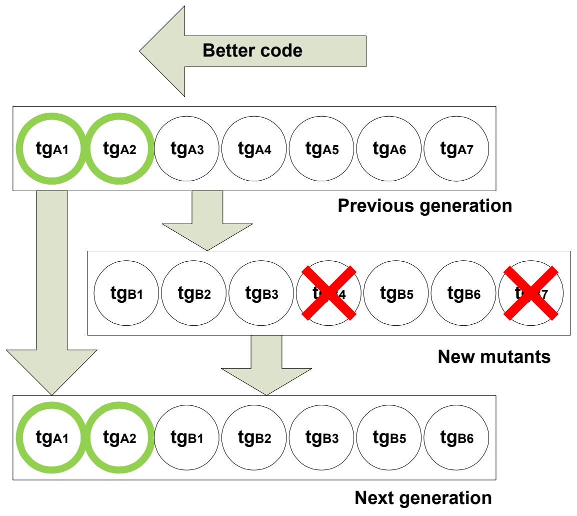
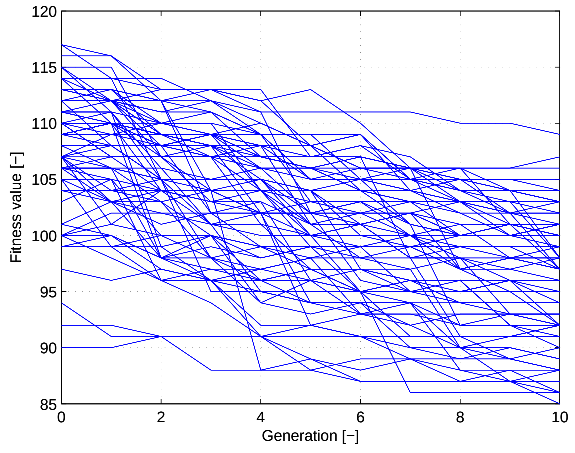


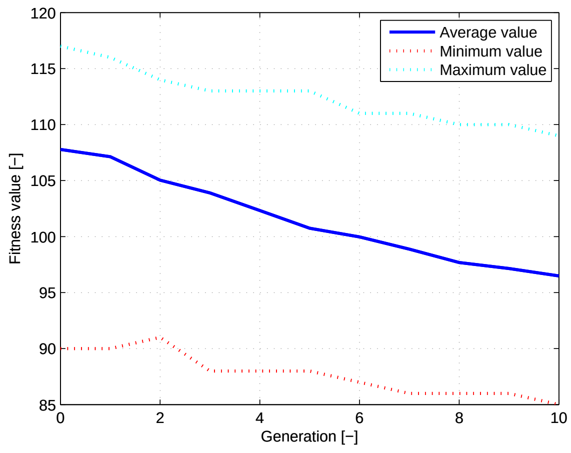
Figure 4-11: Reproducing the next generations of Tanner graphs.

Table 4.1: Minimum E_b/N_0 thresholds for a given $BER = 10^{-4}$ before and after the optimization.

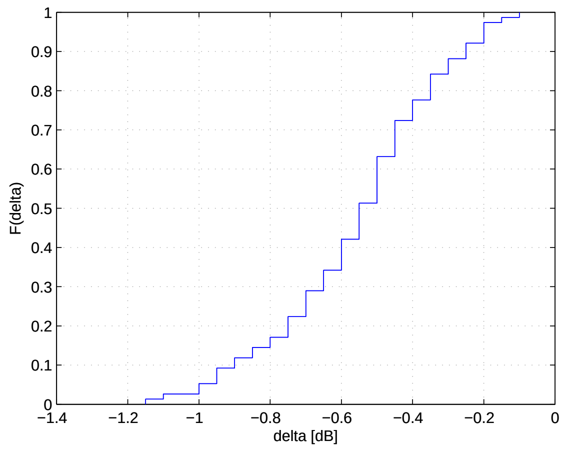
generation	value	threshold
intial	best value - best resulting thread	4.50 dB
inital	best value - worst resulting thread	5.80 dB
10th	best value - best resulting thread	4.25 dB
10th	best value - worst resulting thread	5.50 dB



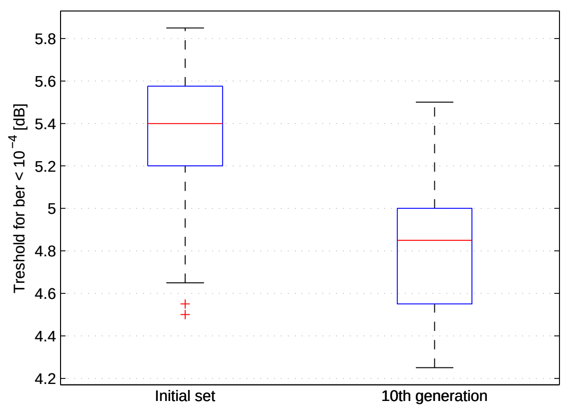
(a) Fitness values for all 76 simulations



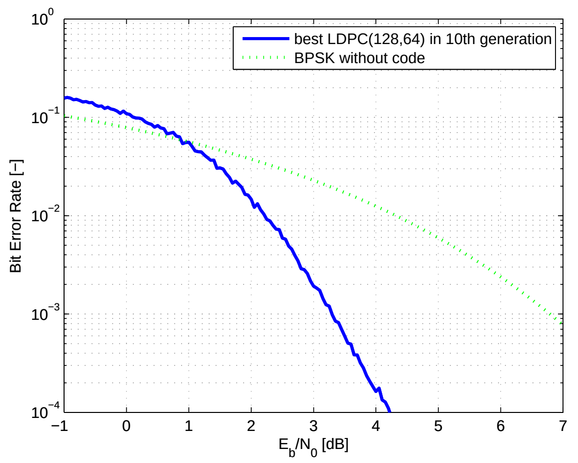
(b) Box plot E_b/N_0 for $BER = 10^{-4}$



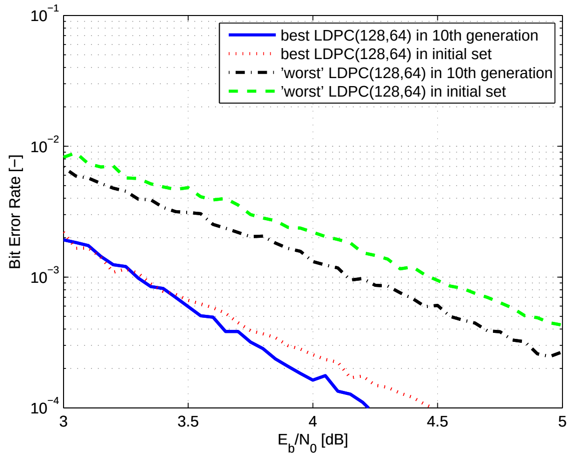
(c) Fitness values for all 76 simulations



(d) Box plot E_b/N_0 for $BER = 10^{-4}$



(e) Detail of BER values



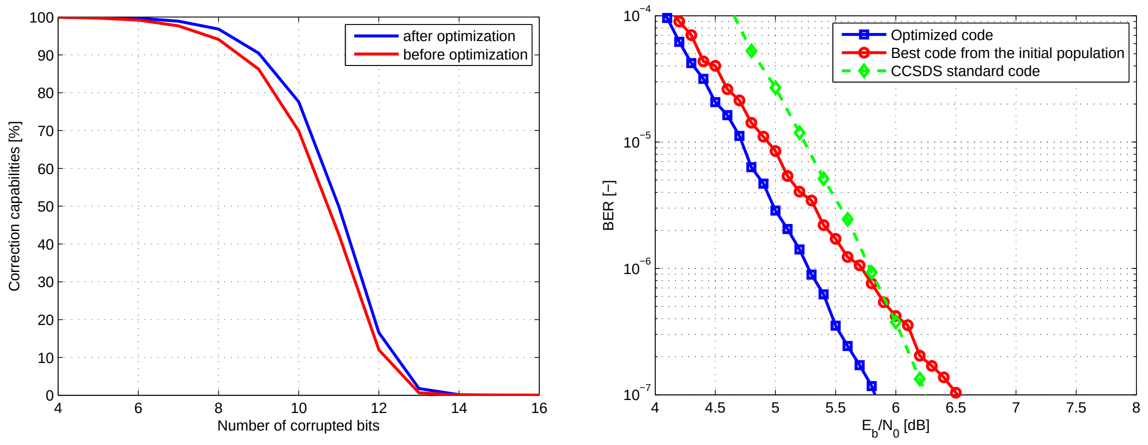
(f) Detail of the fitness function

Figure 4-12: An optimization task using Tanner graph mutations

4.3.2 Long run optimization of LDPC (128,64) code

In this section, the application of the proposed genetic optimization methods is presented. The LDPC code of length (128,64) was chosen for the optimization task. The chosen code was optimized to provide the best performance at 10 iterations. The fitness function used Eq. 4.6 on page 53 with parameters $err_{\min} = 1$ and $err_{\max} = 20$.

Codes before the optimization and after the optimization are compared in Fig. 4-13a and Fig 4-13b in terms of correcting capabilities. The evolution of the fitness function is shown in Fig. 4-13 and the best code achieved is listed in Table 4.2.



(a) Correction capabilites. (b) Bit error rate simulation.

Figure 4-13: The optimization task run on (128,64) LDPC code to provide the best performance at 10 decoding iterations. The initial population were generated with the use of Algorithm 5.

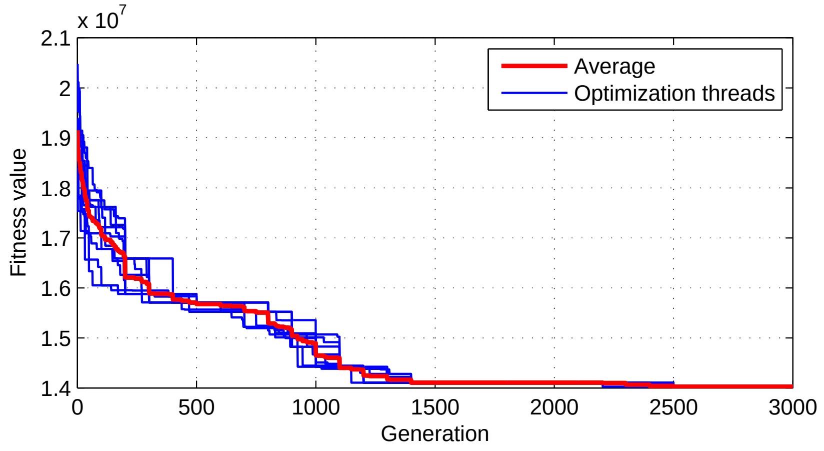


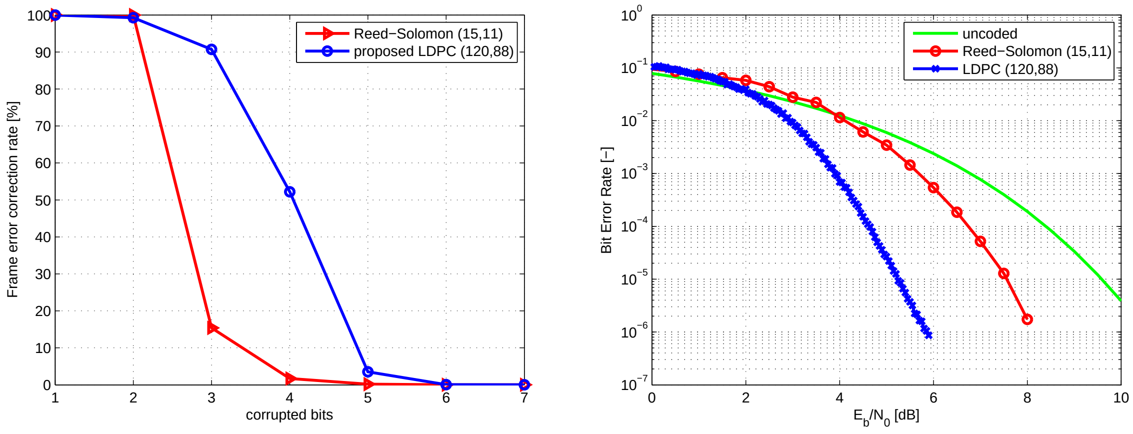
Figure 4-14: Evolution of fitnes value of the optimization task LDPC (128,64), 10 decoding iterations).

Table 4.2: The optimized (128,64) code. The parity-check matrix is listed in the hexadecimal format.

row	**H** matrix
1-2	00002800040010000000041000000000 000001000000001000100008080010000
3-4	00020008000000008100080000000000 00008000008000000000000400006021
5-6	00000008040040400024000000000000 0040000A000000000040001000080000
7-8	80000000000810009000000280800000 30000000020000200000000040000000
9-10	04010041000000001200000010080000 00000400820000008200000000400000
11-12	00002400200000000800000010000000 00000400408000020040020000020000
13-14	00000000002000000004401000004200 00800000000004044004000440000000
15-16	00200028000804000000000002000000 00400100101000000001000000001000
17-18	20000400000100000000080000000040 40080000400000000002000080040288
19-20	01000000000000CA0000400000000000 00000200020248000000000004000000
21-22	01200200000000000000100000000001 60000000000204000000000480000000
23-24	00020800800000000000080400040000 00048000000101000000000000800002
25-26	40000040000000000410400000000014 00000000000000000000000044008003800
27-28	000A0010002000804000000000000000 008000000040010000000010010004000
29-30	00100000000000404010000000000400 00000000000600800000010080200400
31-32	000010000000000040050000020000020 04200000050000000000000000040400
33-34	00001000020000004002000200012000 100000000000000000020004000800200
35-36	00000000000000080180000008000020 00004008100002020000042000000000
37-38	80005020000000100000000000200040 00010002020000010000150000000000
39-40	28002000010000001000002000000801 001000000000020000100080000180000
41-42	0000000000000A000A00000600100000 00840080000040000400200000040000
43-44	41404000200000000000000000000100 00000000800040840000002000000800
45-46	020080000000000021008100008000000 00000005000000100000000000440080
47-48	00000000000000401001000060000002 02000000400820000201010000000000
49-50	00080000010002000001400100200000 0A000000040080000040004000000000
51-52	00000000000080000080001105004040 00000820000000002200000008000200
53-54	02080000004100200010000200000000 00000050180100000000000080000000
55-56	000000001020000000200800000800000 080280000000000002020000080008000
57-58	000000100000000000020001020000009 00002080080000180000000000000600
59-60	00008004000000000200002020080000 00400000800000000400100081400000
61-62	000000000020001000080000C0100000 00000811000000200000000000020020
63-64	00000000000200000000000200000805 00000101000006800000000000008000

4.3.3 Comparison of the optimized LDPC code with the RS code

The irregular LDPC(120,88) was designed with the use of genetic optimization. The length of this code was chosen according to solutions used in CERN for coding GBT packets [42]. The GBT packet is splitted into to parts and each part is coded by Reed-Solomon (15,11). The presented solution assumes that the whole packet is encoded as one codeword. Results are shown in Fig. 4-15.



(a) Correction performance against the number of corrupted bits

(b) Bit Error Rate simulation of LDPC and RS codes

Figure 4-15: Comparison of the genetically designed LDPC (120,88) with the RS (15,11) code.

4.3.4 Evaluation of parallelization

The performance of the accelerated LDPC code genetic construction was tested by the optimization experiment performed on LDPC (64,56) code. The code construction algorithm proposed in Section 4.1 had generated 50 different codes, which were included in the initial population. The optimization using mutations was held until a code with a capability of correcting all single bit errors using the Sum-Product (SP) decoder [70] was reached. The number of required generations is measured for different number of optimization threads and different sharing periods.

Each consequently reproduced generation contains 7 codes. The algorithm preserves the best codes in the whole generation process with the use of elitism, which avoids the loss of the best code during the optimization process. The two best codes from a previous generation are included in a new generation, while two worst codes from a new generation are dropped.

Table 4.3 shows the number of generations required for finding a code with 100% correction capabilities of single bit errors for particular numbers of sharing periods and optimizing threads. Tables 4.4, 4.5 show the minimum and median values of the required generations for the condition defined above. As can be seen, the optimal sharing period K was found to be 100. The fitness values were calculated as the number of uncorrectable single bit errors. An illustrative graph for $L = 4$ is shown in Fig. 4-16 and the comparison of the values for K =10,100,1000,10000 is shown in Fig. 4-17.

Table 4.3: Mean required number of generations for achieving a 100% correction capability of single bit errors.

Sharing period	10	100	1000	10000
2 threads	25942	26003	24125[a]	31330[a]
4 threads	**15830**	19889	21358	34002
8 threads	22526	**9028**	12275	21060
16 threads	21003	**3852**	5403	18314
No sharing	67598[b]			

[a] One optimization fails in convergence
[b] Two optimizations fails in convegence

Table 4.4: Minimum required number of generations for achieving a 100% correction capability of single bit errors.

Sharing period	10	100	1000	10000
2 threads	10039	2675	7428	1096
4 threads	8712	**3382**	9200	4600
8 threads	2460	**3102**	9525	11954
16 threads	4502	**2274**	1386	1822
No sharing	1887			

Table 4.5: Median required number of generations for achieving a 100% correction capability of single bit errors.

Sharing period	10	100	1000	10000
2 threads	20842	21870	34002	31330
4 threads	14058	**13991**	22502	30614
8 threads	14777	**9137**	11002	20002
16 threads	21047	**3608**	4911	19544
No sharing	8670			

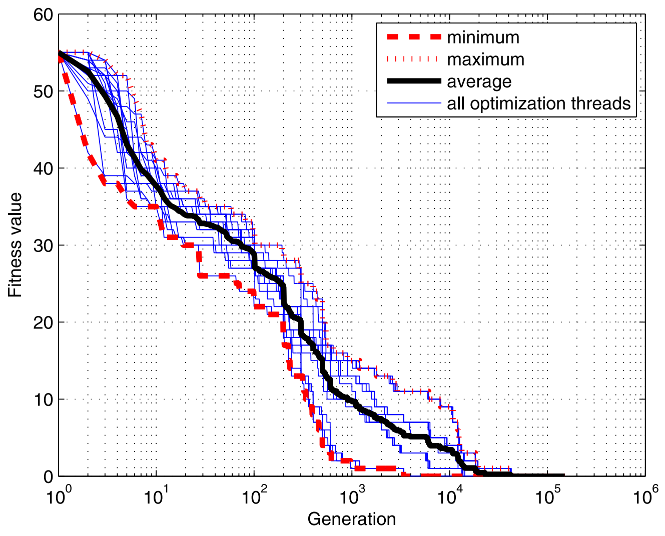


Figure 4-16: Fitness values for $L = 4$ and $K = 100$.

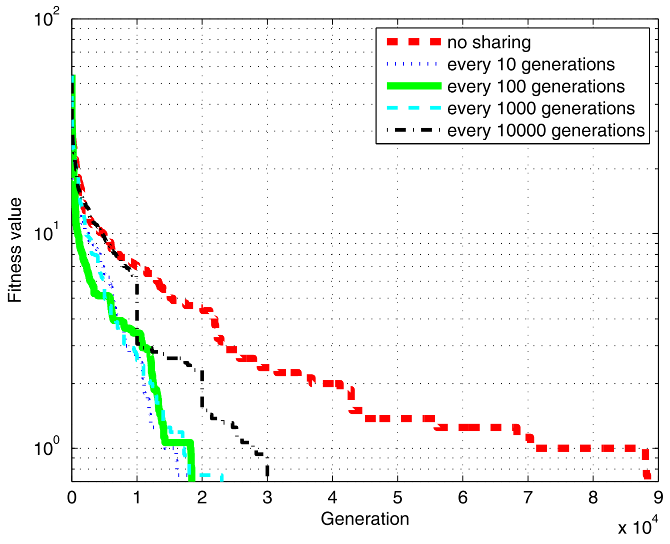


Figure 4-17: Fitness values for $L = 4$ and different sharing periods.

Chapter 5

Mapping LDPC decoder onto parallel architectures

In this chapter, a general parallel approach of an iterative Low Density Parity Check (LDPC) decoder is proposed. The presented parallel approach can be implemented in platforms allowing massive parallel computing, such as Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and computer data storages. The proposed approach supports decoding of any irregular LDPC code, and the maximum node degree is not limited. Benchmarks of the LDPC decoder implemented using Open Computing Language (OpenCL) [90] and Compute Unified Device Architecture (CUDA) [89] frameworks are discussed and the performance comparison is presented.

Inspiring by various comparisons between the OpenCL and CUDA applications from different fields of research, e. g. [3], [6], [23], we have developed parallel algorithms for LDPC decoding using OpenCL and CUDA. Several contributions published so far deal with a general comparison of OpenCL and CUDA [18] and with fitting the LDPC decoder on GPU platform [79], [20], [82], [68], [5], [83], [26]. [69], [66]. However, the decoders are mostly limited for applications with some families of LDPC codes or bounded with the maximum node degree in the associated Tanner graph. The proposed parallel approach is suitable for decoding any irregular LDPC code without the bound in terms of the maximum node degree.

The SP algorithm works as an iterative process of message passing between the two sets of nodes (variable and check) in the Tanner graph. Although the number of operations needed to be performed grows with the number of edges in the graph, the algorithm can be accelerated when deployed on massive parallel architectures. Moreover, the potential acceleration achieved by the parallelization of calculations grows with the number of edges in the graph, because more values can be calculated simultaneously. This can lead to interesting applications for long block length codes providing excellent error correcting capabilities.

In recent years, there has been an increasing interest in implementing LDPC decoders in a wide variety of hardware architectures, including GPU. Several contributions deal with fitting the decoder on parallel architectures with the use OpenCL or CUDA frameworks and discuss the benchmarks [79], [20], [82], [68], [5], [83], [26], [69], [66]. However, work reviewed so far deal mostly with some families of LDPC codes and the application of parallel decoders is limited. The proposed approach divides calculations into a scalable number of threads. Each thread performs the calculation of the value outgoing through the edge, which is associated with the thread itself (edge-level parallelization). The approach was chosen because of its suitability for any irregular LDPC matrices, scalability for any code block lengths and deployablity on many hardware architectures. It is also convenient for derived algorithms for LDPC decoding, such as Min-Sum (MS) or adaptive MS [77]. In the previous work dealing with the parallel LDPC decoding, the calculations were mostly divided on the level of rows and columns of the parity-check matrices, which is the main difference from the proposed approach.

5.1 A general parallelization of LDPC decoders

In this section, the approach of the edge-level parallelization used for the LDPC decoder is described.

Considering the LDPC code and associated Tanner graph, the following arrays are defined as address iterators for the parallel message passing algorithm (described in Algorithms 9, 10 and 11):

- a sorted m-tuple of variable nodes $\mathbf{v} = (v_0, \ldots, v_m)$ starting with the lowest index and associated m-tuple of check nodes $\mathbf{c} = (c_0, \ldots, c_m)$, such $i, j : \mathbf{H}_{i,j} = 1$ and $i \in [0, n-k), j \in [0, n)$; then, (c_i, v_j) unequivocally defines an edge in the Tanner graph; n is the number of variable nodes and $(n-k)$ is the number of check nodes, k is the number of information bits, and m is the number of edges in the Tanner graph
- m-tuple of edges $\mathbf{e} = (e_0, \ldots e_m) = (0, 1, 2, ..., |\mathbf{c}|)$,
- m-tuple of connected edges $\mathbf{t} = (t_0, \ldots, t_m)$ with a variable node v_k, where, $t_k = |\{v_k\}|$ for $k \in \{0, 1, \ldots, m-1\}$, and $v_k \in \mathbf{v}$
- m-tuple of starting positions $\mathbf{s} = (s_0, \ldots, s_m)$ for iterating in order to calculate the value passed through the edge e_k; $s_k = \arg\min_\mathrm{k}(v_k : v_k \in \mathbf{v})$
- m-tuple $\mathbf{u} = (u_0, \ldots, u_m)$ of relative positions of the e_k associated with the connected node v_k; $u_k = k - |(v_q) : q < k, v_q \neq v_k|$

The arrays defined above are used as address iterators for calculations of messages outgoing from variable nodes to check nodes (the first half of the iteration). The arrays are also shown in the illustrative example. Supposing the code (14,7), the parity-check matrix and arrays derived by the principle described above are shown in Appendix A.

The first half of the iteration of the LDPC decoding process calculates the values passed from the variable nodes to the check nodes. With the use of the array iterators we can perform such calculations without any complicated operations with array indices. The pseudo code is shown in Algorithm 9. The local index of the thread (according to the OpenCL terminology) is denoted as *lid* and the number of synchronized threads working in parallel is denoted as *lgsize*. Because all threads performing the calculations have to be synchronized after they finish writing to the memory and the number of synchronizable threads is strictly limited (e. g. 1024), the calculations are divided in several steps (pages) if necessary. This is when the number of edges is greater than the *lgsize* variable. An illustrative example for 12 synchronizable threads is shown in Fig. A-2.

The arrays for the second half on the iteration can be derived similarly. Keeping of the unique edge identifier (c_i, v_j) and associated edge index e_k, the arrays $\mathbf{c}, \mathbf{v}, \mathbf{e}$ are sorted starting with the lowest check node index and other arrays are derived considering the messages outgoing from the check nodes. Such arrays are then denoted as $\overline{\mathbf{e}}, \overline{\mathbf{c}}, \overline{\mathbf{v}}, \overline{\mathbf{t}}, \overline{\mathbf{s}}, \overline{\mathbf{u}}$) in the following descriptions. As a demonstrative example, the arrays for the second half of the iteration are shown in Table B.2.

The algorithm performing the second half of the iteration processes the arrays described above. Its pseudo code is shown in Algorithm 9. After finishing the second half of the iteration we can continue with the next iteration. The whole decoding principle remains the same, as described in Algorithm 9.

For example, the address iterators for the LDPC (14,7) code are listed in Table A.1 and Table B.2. Both tables are particularly useful for understanding the principle and checking the correctness of the implementation. To keep the consistency and for tutorial purposes, both tables are associated with the LDPC (14,7) code given by the parity-check matrix from Fig. A-2.

5.2 OpenCL and CUDA implementation

In current signal and data processing systems, there is an unambiguous trend to use parallel architectures to increase the processing speed, which plays a crucial role in real time applications and determines a deployability of computationally complex algorithms in hardware. Hardware devices supporting massively parallel processing algorithms generally include Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs).

In this work, the CUDA and the OpenCL frameworks are used for GPU computations. The OpenCL is an open standard for parallel programming using the different computational devices, such as CPU, GPU, or FPGA. It provides a programming

Algorithm 9 Parallel message passing

```
1: procedure ITERATE TO CHECK NODES
   ▷ Half on an iteration
   Input: r – incoming values e, s, t, u
   Output: q
2:     for ($p = 0$; $p <$ totaledges; $p+ = lgsize$) do
3:         for $i = s_{lid+p}$ to $s_{lid+p} + t_{lid+p} - 1$ do
4:             if $i = u_{lid+p} + s_{lid+p}$ then    continue
5:             end if
6:             $value$ = perform calculations                          ▷ Algorithm 1
7:         end for
8:         $index = e_{lid+p}$
9:         $q_{index} = value$
10:    end for
11: end procedure

12: procedure ITERATE TO VARIABLE NODES
   ▷ Half on an iteration
   Input: q – incoming values $\overline{\mathbf{e}}$, $\overline{\mathbf{s}}$, $\overline{\mathbf{t}}$, $\overline{\mathbf{u}}$
   Output: r
13:    for ($p = 0$; $p <$ totaledges; $p+ = lgsize$) do
14:        for $i = \overline{s_{lid+p}}$ to $\overline{s_{lid+p}} + \overline{t_{lid+p}} - 1$ do
15:            if $i = \overline{u_{lid+p}} + \overline{s_{lid+p}}$ then    continue
16:            end if
17:            $value$ = perform calculations                          ▷ Algorithm 1
18:        end for
19:        $index = \overline{e_{lid+p}}$
20:        $r_{index} = value$
21:    end for
22: end procedure
```

language based on the C99 standard. Unlike OpenCL, CUDA is only for NVIDIA devices starting from G80 series (so called CUDA-enabled GPUs). CUDA gives a possibility to write programs based on the C/C++ and Fortran languages.

When implementing an algorithm on GPU platform using OpenCL or CUDA frameworks, two main issues have to be considered:

- size of the local memory (OpenCL) or shared memory (CUDA),
- size of the working group (OpenCL) or block size (CUDA).

Algorithm 10 Parallel calculation of the estimation

```
1: procedure CALCULATE ESTIMATION
   ▷ Parallel approach
   Input: $\mathbf{r}$ – incoming values $\mathbf{s}$, $\mathbf{t}$, $\mathbf{v}$
   Output: $\widehat{\mathbf{c}}$
2:     for ($p = 0$; $p <$ totaledges; $p+ = lgsize$) do
3:         $Q(1) = r_{lid+p}$
4:         $Q(0) = 1 - r_{lid+p}$
5:         for $i = s_{lid+p}$ to $s_{lid+p} + t_{lid+p} - 1$ do
6:             $Q(1) = Q(1) r_{i+p}$
7:             $Q(0) = Q(0)(1 - r_{i+p})$
8:         end for
9:         $index = v_{lid+p}$
10:        if $Q(1) > Q(0)$ then $\widehat{c}_{index} = 1$
11:        else $\widehat{c}_{index} = 0$
12:        end if
13:        $index = v_{lid+p}$
14:        $q_{index} = value$
15:    end for
16: end procedure
```

GPU devices offer several types of the allocable memory, which differ in their speed and their size. The memory type used to store variables is specified in the source code by the prefix according to the OpenCL or CUDA syntax rules. Generally, the largest allocable size, typically in gigabytes for current devices, is located in the global memory. However, the global memory is also the slowest one. A higher speed is provided by the local memory, but the size is typically only in kilobytes. Exceeding the limited size of the local memory usually leads to incorrect results without any warnings in the compilation report.

Another crucial issue related to an algorithm implementation in GPU devices is the working group size. Although the GPU can run thousands of threads in parallel, these threads are not synchronized among each other in terms of writing to the memory. The threads are split up into working groups and they can be synchronized only among other threads at the same working group. The size of the working groups is limited (typically 1024).

Both frameworks processes two types of code

Algorithm 11 Parallel calculation of the syndrome

```
1: procedure CALCULATE SYNDROME
   ▷ Parallel approach
   Input: $\widehat{\mathbf{c}}$ – codeword estimation, $\overline{\mathbf{s}}$, $\overline{\mathbf{t}}$, $\overline{\mathbf{c}}$, $\overline{\mathbf{v}}$
   Output: $\mathbf{z}$ – syndrome $\widehat{\mathbf{c}}\mathbf{H}^\top$
2:     for ($p = 0$; $p <$ totaledges; $p+ = lgsize$) do
3:         $value = 0$
4:         for $i = \overline{s_{lid+p}}$ to $\overline{s_{lid+p}} + \overline{t_{lid+p}} - 1$ do
5:             $index = \overline{v_{lid+p}}$
6:             $value \,\hat{}= \widehat{c}_{index}$
7:         end for
8:         $index = \overline{c_{lid+p}}$
9:         $z_{index} = value$
10:    end for
11: end procedure
```

- host (runtime), running serially on CPU
- kernel (device), running parallely on GPU

The kernel is executed by the host. In CUDA, the kernel execution is more straightforward compared to OpenCL. Both codes execute the kernel *berSimulate* in 100 working groups (blocks) with 512 threads per one working group. Because the kernel function has to be considered as a function running in parallel, each thread has its own unique identifier - the combination of global ID and local ID in OpenCL or the combination of thread ID and block ID in CUDA, which can be recalculated vice versa. Types used for code definition and passing messages are pointed in Listing A.1.

Some main differencies between the OpenCL and CUDA syntax rules are shown in Table A.3, which can be used when moving the source code from one framework to another one.

5.3 Experimental evaluation

Developed algorithms for LDPC decoding were run on NVIDIA Tesla K40 (Atlas) and Intel Xeon E5-2695v2 platforms [91, 92]. The NVIDIA device contains 2880 CUDA cores and runs at 745 MHz. The peak performance for double precision computations with floating point is 1.43 TFLOPS. The clock frequency of the Intel Xeon CPU is 2.4 GHz. All measurements include the time required for random generation, realized by the Xorshift+ algorithm and the Box-Muller transform.

Benchmarks were performed through the calculation of the Bit Error Rate at $E_b/N_0 = 2\text{dB}$ for a code given by the NASA CCSDS standard [93] and its protographically expanded derivations [59], [19]. Based on the results obtained from NVIDIA Tesla K80, we got slightly better performance with the use of the CUDA framework, as shown in Fig. 5-1. Compared to the CPU implementation run on Intel Xeon, the acceleration grows with the size of working groups and the number of decoders running in parallel to the limit of the device, as illustrated in Fig. 5-2, 5-3. GPU become very effective for longer block length codes, as also shown in Table 5.1. The ratio between CPU (C++ compiler with O3 optimization) and GPU was 25 for code of 262144 bits.

To keep the generality, no simplifications in the decoding algorithm were applied and the experimental evaluation was performed with the use of the global memory. For further acceleration, several tasks can be considered, i. e. usage of the local memory, variables with a lower precision, look-up tables, or modifications of the algorithm for certain families of LDPC codes. For example, by moving the part of variables in the local (shared) memory, the decoder works approximately 40% faster in our experience. However, it is not possible to decode longer codewords because of the size limitations (240 kB of the local memory per working group). Another possibility for greater optimization could be the parallelization of less computationally intensive functions. After applying parrallel algorithms for passing messages, calculating the syndrome and the estimation, the most serial time-consuming operation is checking syndrome for all zero equality (approximately 34% of the decoding function in our experience).

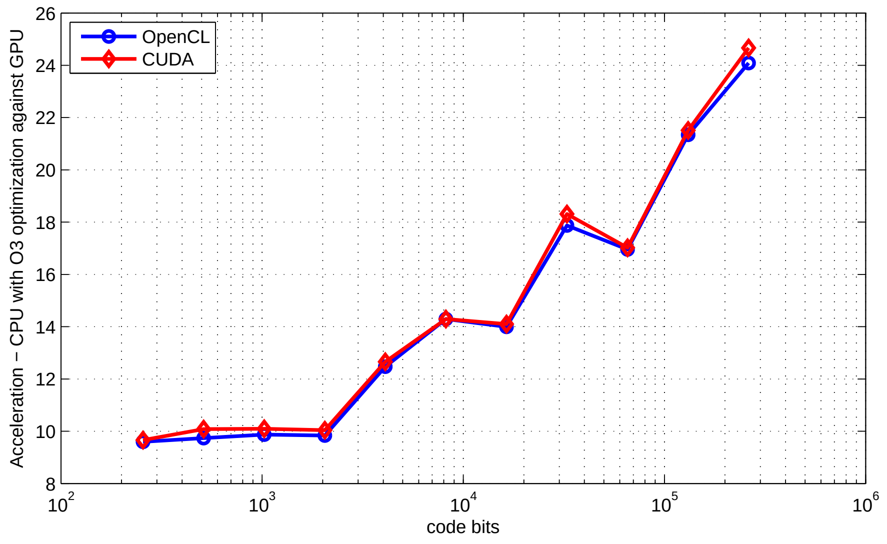


Figure 5-1: The acceleration dependence on the length of the code. Comparison for OpenCL and CUDA frameworks (local group of 512 threads and 100 decoders working in parallel) against CPU implementation using C++ compiler with O3 optimization. Time was mesuared for 10000 decoded codewords at E_b/N_0 =2dB.

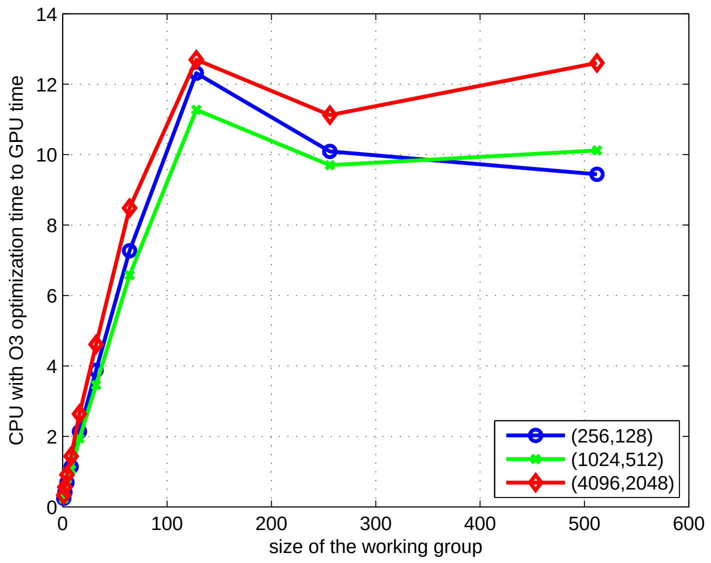


Figure 5-2: The acceleration dependence on the block (working group) for 100 decoders running in parallel.

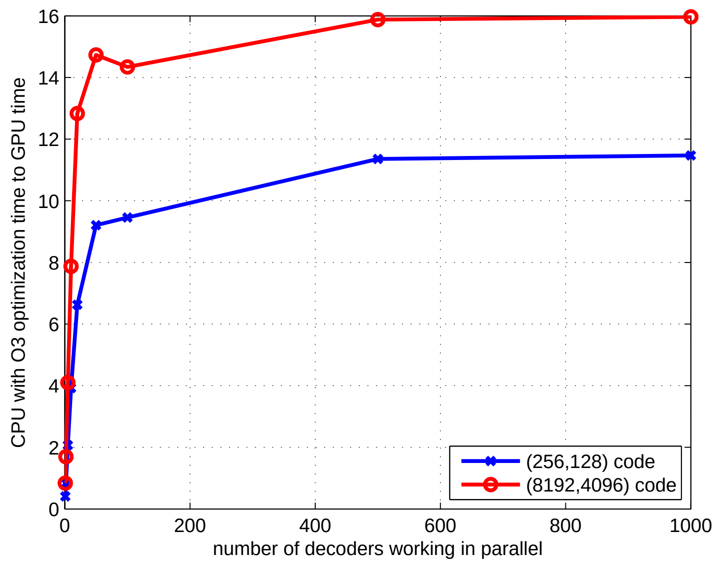


Figure 5-3: Acceleration dependence on the number of decoders working in parallel when the size of the working group is 512.

Table 5.1: Comparison for OpenCL and CUDA framework (local group of 512 threads and 100 decoders working in parallel) against the CPU implementation using C++ compiler with O3 optimization. Time was mesuared for 10000 decoded codewords at E_b/N_0 =2dB.

code	**edges**	**OpenCL**	**CUDA**	**C++**	**C++ with O3 opt.**
(256,128)	1024	0.32 s	0.32 s	24.24 s	3.11 s
(512,256)	2048	0.64 s	0.61 s	26.98 s	6.24 s
(1024,512)	4096	1.26 s	1.24 s	99.59 s	12.52 s
(2048,1024)	8192	2.56 s	2.51 s	105.56 s	25.27 s
(4096,2048)	16384	5.54 s	5.46 s	415.35 s	69.17 s
(8192,4096)	32768	12.08 s	12.08 s	545.74 s	172.67 s
(16384,8192)	65536	26.27 s	26.08 s	1717.25 s	367.75 s
(32768,16384)	131072	57.40 s	56.02 s	2893.91 s	1025.9 s
(65536,32768)	242144	117.31 s	116.86 s	8572.08 s	1989.26 s
(131072,65536)	524288	244.36 s	242.43 s	14082.71 s	5215.11 s
(262144,131072)	1048576	510.06 s	498.16 s	35104.28 s	12287.61 s

Chapter 6

Improving performance of LDPC decoders

The performance of decoding algorithms is usually simulated on the Additive White Gaussian Noise (AWGN) channel. The simulation results provide the dependence of a Bit Error Rate (BER) on a Signal-To-Noise (SNR) ratio. SNR is often recalculated to E_b/N_0.

The convergence of BP algorithms is guaranteed solely in graphs with a tree structure. The occurrence of cycles in the Tanner graph causes the convergence failure, which has a negative impact on the decoding performance. Short cycles generally cause the worse convergence. However, there are methods that show how the convergence of the BP can be improved [4], [31], [32], [72], [75], [77], [78]. Two innovative methods are described in the following sections.

6.1 Belief Propagation Based on Estimation Backtracking

The first proposed method, the estimation BackTracking (BT), holds all estimations calculated progressively in each iteration. The estimations are saved in the array or in the stack and backtracked if the forward decoding is not successful. This algorithm is performed only if the traditional BP decoding fails. When a maximum number of iterations is reached in the BP decoding and the codeword estimation after this does not meet all parity checks, the proposed backtracking algorithm is performed. Improved decoding algorithms are denoted as SP-BT for Sum-Product with the estimation BackTracking, MS-BT for Min-Sum with the estimation Back-Tracking, and BF-BT for Bit-Flipping with estimation BackTracking. The proposed algorithm improves the Bit Error Rate (BER) of the LDPC decoders, which is shown in the following section.

The backtracking process searches for the bits which meet the parity-check condition. The estimations c_{it} are backtracked starting with the last estimation. If some parity-check conditions in c_{it} are met, relevant bits in c_{BT} are replaced by the bits from c_{it} and the syndrome is checked. The whole algorithm for soft and hard decoding is described in Algorithm 12 and 13.

Algorithm 12 Improved soft-decision BP algorithm

```
 1: procedure DECODE

    Input: $\mathbf{y}$, $p$, ITERATIONS
    Output: $\widehat{\mathbf{c}}$
 2:     Initialize: $p_j(a) = P(c_j = a|y_j), it \leftarrow 0$
 3:     Send to check nodes: $q_{j\to i} = \log(p_j(0)/p_j(1))$
 4:     Calculate $r_{i\to j}$ values and send back to variable nodes
    ▷ Different formulas for different algorithms
 5:     if $\widehat{\mathbf{c}}\mathbf{H}^\top = \mathbf{0}$ then
 6:         return $\widehat{\mathbf{c}}_{it}$                    ▷ termination, successful
 7:     end if
 8:     if $it$ = ITERATIONS then
    ▷ See Algorithm 14
 9:     end if
10:     Calculate $q_{j\to i}$ values and send to check nodes
    ▷ Different formulas for different algorithms
11:     $it \leftarrow it + 1$
12:     goto(4)
13: end procedure
```

The implementation of the Estimation Backtracking algorithm adds a non-negligible complexity in the BP decoder. In particular, the algorithm requires logical operations

Algorithm 13 Improved Bit-Flipping algorithm

```
1: procedure DECODE

    Input: $\mathbf{y}$, ITERATIONS
    Output: $\widehat{\mathbf{c}}$
2:     Initialize: $b_j = y_j, it \leftarrow 0$
3:     Send to check nodes: $q_{j \to i} = b_j$
4:     Calculate $r_{i \to j} =$ xor of all $q_{j \to i}$
   ▷ Calculate $r_{i \to j}$ based on the exclusive OR of all incoming $q_{j \to i}$ values and send
   back to variable nodes
5:     $r_j = \sum r_{i \to j}$
   ▷ Calculate the sum $r_j$ of all incoming messages to the each variable node of index
   $j$,
6:     Flip $b_j$, $j \in \text{argmax}(r_0, r_1, .., r_{n-1})$
7:     $\widehat{\mathbf{c}}_{it} = (b_j)$, $j = 0, 1, ..., n-1$
8:     if $\widehat{\mathbf{c}}\mathbf{H}^\top = \mathbf{0}$ then
9:         return $\widehat{\mathbf{c}}_{it}$                    ▷ termination, successful
10:    end if
11:    if $it =$ ITERATIONS then
   ▷ See Algorithm 14
12:    end if
13:    Calculate $q_{j \to i}$ values and send to check nodes
14:    $it \leftarrow it + 1$
15:    goto(4)
16: end procedure
```

for performing the algorithm and memory for saving the node indices. It can be also bypassed when a low decoding time is crucial. In the next sections, a description of the proposed algorithm is provided, which is followed by the complexity analysis and a simulation of the decoding performance.

6.1.1 Algorithm description

Let $\mathcal{M}_j$ be the vector of check nodes connected with j-th variable node and $\mathcal{N}_i$ the vector of variable nodes connected with the i-th check node. Then

$$\mathcal{M}_j = \{i\} \Leftrightarrow \mathbf{H}_{i,j} = 1 \tag{6.1}$$

$$\mathcal{N}_i = \{j\} \Leftrightarrow \mathbf{H}_{i,j} = 1 \tag{6.2}$$

where $\mathbf{H}$ is the parity-check matrix of the LDPC code. The estimation of a codeword after it-th iteration is denoted as $\widehat{\mathbf{c}}_{it}$, where it is 1, 2, ..., ITERATIONS, and ITERATIONS is the actual number of iterations performed by the BP decoding algorithm. The estimation of a codeword produced by the proposed algorithm is denoted as $\widehat{\mathbf{c}}_{BT}$. The proposed estimation BackTracking is described in Algorithm 14 in more detail. At the initialization, the last estimation of a codeword is copied into $\widehat{\mathbf{c}}_{BT}$ and then, the estimations $\widehat{\mathbf{c}}_{it}$ are iterated starting with the last vector. In each iteration, the bits of $\widehat{\mathbf{c}}_{it}$ met the parity-check condition are copied into the vector $\widehat{\mathbf{c}}_{BT}$, where they replace the old values in the same positions. This is performed through two for cycles using vectors $\mathcal{M}_j$ and $\mathcal{N}_i$, as can be seen in Algorithm 14. If all bits of $\widehat{\mathbf{c}}_{BT}$ meet the parity-check condition ($\widehat{\mathbf{c}}_{it}\mathbf{H}^\top = 0$), the algorithm is terminated and the vector $\widehat{\mathbf{c}}_{BT}$ is returned as the codeword estimation. Otherwise, the algorithm continues to the previous $\widehat{\mathbf{c}}_{it}$.

6.1.2 Memory requirements

The proposed algorithm requires all codeword estimations to be stored in the memory (stack). These vectors are being read during backtracking process starting with the

Algorithm 14 Estimation backtracking

```
1: procedure ESTIMATION BACKTRACKING

   Input: $\widehat{\mathbf{c}}_{it}$, $it = 1, 2, ..., \text{ITERATIONS}, \mathcal{M}, \mathcal{N}$
   Output: $\widehat{\mathbf{c}}_{BT}$
2:     Initialize: $\widehat{\mathbf{c}}_{BT} = \widehat{\mathbf{c}}_{\text{ITERATIONS}}$
3:     for $it = \text{ITERATIONS} - 1$ downto 0 do
4:         for $k = 0$ to $n - 1$ do                                  ▷ all variable nodes
5:             parity $\leftarrow 0$
6:             for all $i \in \mathcal{M}_k$ do
7:                 bit $\leftarrow 0$
8:                 for all $j \in \mathcal{N}_i$ do
9:                     bit $\leftarrow$ bit xor $\widehat{\mathbf{c}}_{it}(j)$
10:                end for
11:                bit $\leftarrow$ parity xor bit
12:            end for
13:            if parity $= 0$ then
14:                $\widehat{\mathbf{c}}_{BT}$(k) $\leftarrow \widehat{\mathbf{c}}_{it}(k)$
15:            end if
16:        end for
17:    end for
18:    return $\widehat{\mathbf{c}}_{BT}$
19: end procedure
```

last codeword estimation. The number of bits needed to be stored is:

$$n \times \text{ITERATIONS} \tag{6.3}$$

where n is the number of columns of $\mathbf{H}$. The Estimation Backtracking algorithm also requires the vectors $\mathcal{M}_j$ and $\mathcal{N}_i$ to be stored. Supposing the average variable degree d_v, the average check degree d_c, the number of integers needed to be stored is:

$$n \times d_v + (n-k) \times d_c \tag{6.4}$$

where $(n-k)$ is the number of rows of $\mathbf{H}$. However, information provided by vectors $\mathcal{M}_j$ and $\mathcal{N}_i$ can be shared with the BP decoder. Thus, the memory requirements highly depend on the architecture used for the implementation of the decoder.

6.1.3 Complexity requirements

The BT algorithm also requires a non-negligible number of logical operations to be performed. The operations are ORs and eXclusive ORs (XORs). The maximum number of ORs performed during the estimation backtracking is given by the formula below:

$$n \times d_v \times \text{ITERATIONS} \tag{6.5}$$

and the maximum number of XORs performed during the BT algorithm is the following:

$$n \times d_c \times d_v \times \text{ITERATIONS} \tag{6.6}$$

These values are reached if the BT algorithm is terminated as unsuccessful or if it succeeds in the last iteration.

6.1.4 Comprehensive AWGN simulations

The performance of the proposed decoding algorithms was tested on Mackay's widely known code (504,252) [36], irregular code (128,64) [10], irregular code (256,128) provided by CCSDS NASA standard [93]. The dependence of the Bit Error Rate on the Signal-to-Noise Ratio was measured while the Additive White Gaussian Noise (AWGN) channel was used for the transmission. Comprehensive simulations were running for 150 iterations. All Bit Error Rates are calculated from 400 codewords for which the iterative decoding algorithm failed at particular Signal-to-Noise Ratios. The Bit Error Rates simulated on LDPC (128,64), CCSDS (256,128) and MacKay's (502,252) code with the use of SP of are shown in Fig. 6-1, 6-2, 6-3. The MS-BT and BF-BT algorithms are demonstrated in Fig. 6-4, 6-5. Results confirm an improvement of the Bit Error Rate calculated for particular E_b/N_0 ratio. The simulated BER can be decreased several times according to the used code. The largest improvement from tested codes can be seen in the simulation of MacKay's code.

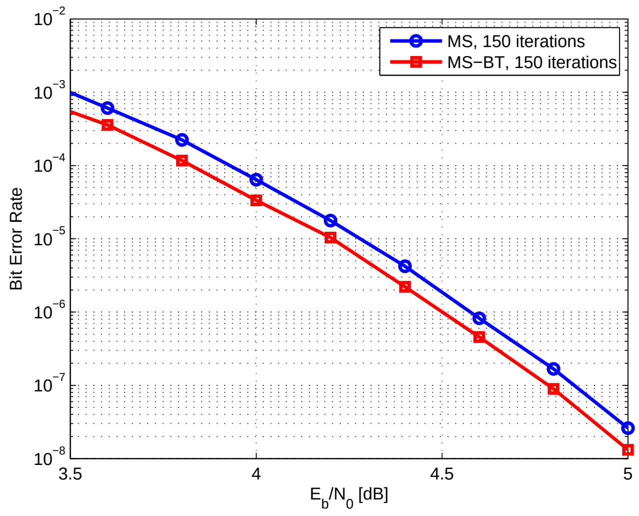


Figure 6-1: Bit error rate performance simulated on NASA CCSDS (256,128) standard.

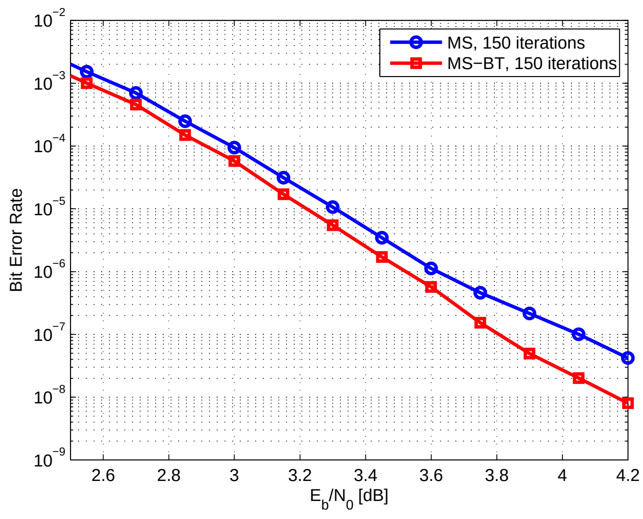


Figure 6-2: Bit error rate performance simulated on MacKay's (504,252) code.

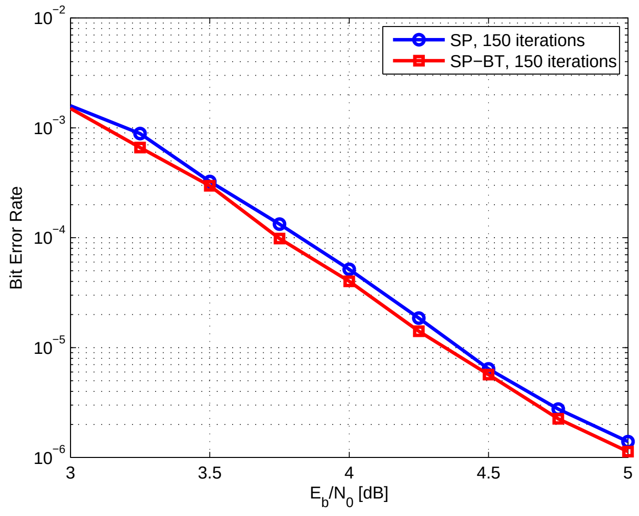


Figure 6-3: Bit error rate simulated on our irregular LDPC (128,64) code. The original SP algorithm and the improved SP-BT are compared.

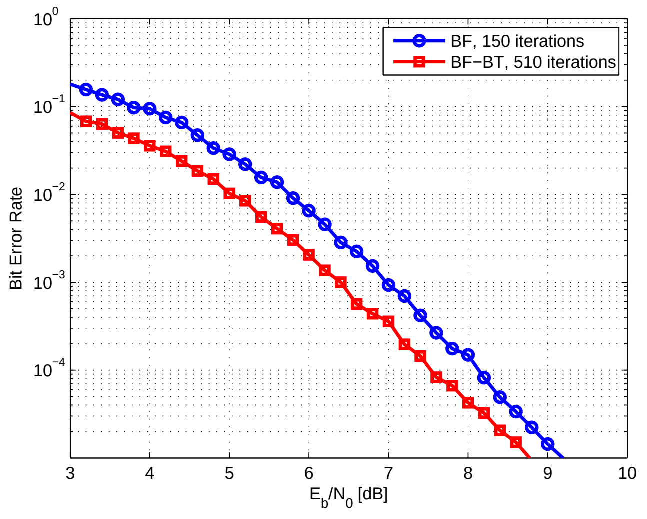


Figure 6-4: Bit error rate simulated on our irregular LDPC (128,64) code. The original Bit-Flipping algorithm and the improved Bit-Flipping algorithm with backtracking of estimations are compared.

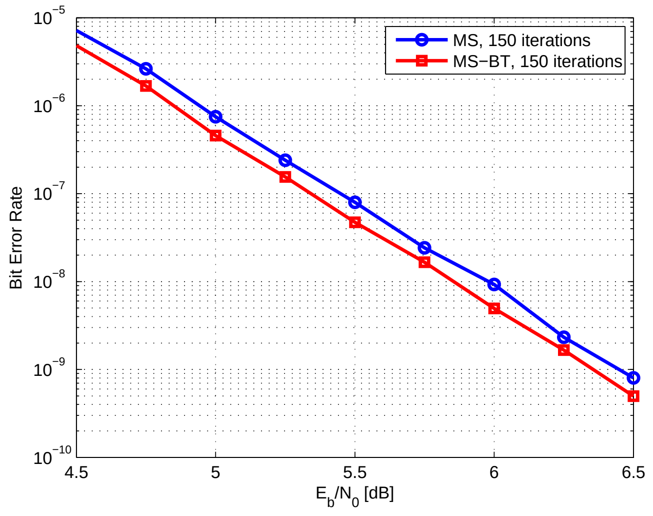


Figure 6-5: Bit error rate simulated on our irregular LDPC (128,64) code. The original Min-Sum algorithm and improved Min-Sum algorithm with backtracking of estimations are compared.

6.2 Mutational LDPC decoding

The second proposed method is based on the following principle. Considering the LDPC code $\mathcal{C}$ defined by the optimized parity-check matrix $\mathbf{H}$ and the unique generator matrix $\mathbf{G}$, such $\mathbf{G}\mathbf{H}^\top = \mathbf{0}$. Because the parity-check matrix is not unique, the decoder for the same code can run using different $\mathbf{H}$ matrices, which belong to the same code space. The convergence properties of the decoders are different then. Although the LDPC code is usually defined by the optimized $\mathbf{H}$ parity matrix for decoding under the Belief Propagation algorithm and decoding with the use of other parity-check matrices provides worse correcting performance, it can be shown that the information from several decoders can be combined together for achieving better performance.

All codewords of the code $\mathcal{C}$ satisfies the condition $\mathbf{c}\mathbf{H}^\top = \mathbf{0}$, where $\mathbf{H}$ consists of $(n-k)$ row vectors, denoted as $\mathbf{h_i}$. Therefore, all matrices which belong to the same code space are given by all possible summations of row vectors among each other.

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}_1 \\ \mathbf{h}_2 \\ \vdots \\ \mathbf{h}_{(n-k)} \end{bmatrix} \sim \begin{bmatrix} \mathbf{h}_1 \oplus \left(\bigoplus_{i_1 \in I_1} \mathbf{h}_{i_1}\right) \\ \mathbf{h}_2 \oplus \left(\bigoplus_{i_2 \in I_2} \mathbf{h}_{i_2}\right) \\ \vdots \\ \mathbf{h}_{(n-k)} \oplus \left(\bigoplus_{i_{(n-k)} \in I_{(n-k)}} \mathbf{h}_{i_{(n-k)}}\right) \end{bmatrix}, \tag{6.7}$$

where $I_s \subseteq \{1, 2, \ldots, (n-k)\}$ for $s \in \{1, 2, \ldots, (n-k)\}$.

Supposing a slight mutation of the parity-check matrix performed according to Algorithm 15, the overall estimaton of the codeword is calculated by combining information of several decoders. There are two algorithms for performing the combination considered:

- voting (majority),
- calculating the global estimation based on probabilities (described in the following sections)

The methodologies for combining the information based of probabilities are described in the following paragraphs. These methodologies provide better performance than the majority.

Algorithm 15 Mutation of H matrix

```
1: procedure MUTATE
     Input: $\mathbf{H}$, M                          ▷ M is the number of mutations
     Output: $\mathbf{H}^m$                          ▷ a mutated matrix
2:     for $k = 0$ to $\mathrm{M} - 1$ do
3:         $r_1 = random, r_2 = random$
4:         $\mathbf{H}(r_1) = \mathbf{H}(r_1) \oplus \mathbf{H}(r_2)$
     ▷ $\mathbf{H}(r_1), \mathbf{H}(r_2)$ are rows of the matrix $\mathbf{H}$
5:     end for
6: end procedure
```

6.2.1 Principle of Mutational LDPC decoding

The principle of the Mutational LDPC (MLDPC) decoding, utilizing several decoders working in parallel, is introduced in the following paragraphs. We define the probabilities $p_j^{m,s}(a) = p^{m,s}(c_j = a|y_j)$ as the amount of belief that the sent j-th symbol is equal to a after s-th iteration of m-th decoder, where $a \in \{0, 1\}$. For $s = 0$, the probabilities are the initial probabilities $p_j^{m,0}(a) = p_j^0(a)$. After the initialization, their values are sent to check nodes as q_{ij} messages:

$$q_{ij}^{m,s}(a) = p_j^{m,0}(a), \tag{6.8}$$

where m is the mutation index, i is the check node index, j is the variable node index, and s is the iteration index ($s = 0$ here).

The values, indicating how check nodes are satisfied with $p_j^{m,0}$ after s-th iteration, are calculated as follows:

$$r_{ij}^{m,s}(0) = \frac{1}{2} + \frac{1}{2} \prod_{j' \in \mathcal{N}_i^m \setminus j} \left(1 - q_{ij'}^{m,s}(1)\right) \tag{6.9}$$

$$r_{ij}^{m,s}(1) = 1 - r_{ij}^{m,s}(0) \tag{6.10}$$

where $\mathcal{N}_i^m = \{j\}$ for all $H_{ij}^m = 1$, and $\mathbf{H}^m$ is a matrix produced by slight mutations according to the principle given in Eq. 6.7 and Algorithm 15 .

Then, the probabilities of beliefs are updated:

$$p_j^{m,\,s}(a) = K_j^{m,\,s} p_j^{m,\,0}(a) \prod_{i \in \mathcal{M}_j^m} r_{ij}^{m,\,s}(a) \tag{6.11}$$

where $K_j^{m,\,s}$ is the normalization constant to satisfy $p_j^{m,\,s}(0)+p_j^{m,\,s}(1) = 1$, and $\mathcal{M}_j^m = \{i\}$ for all $H_{ij}^m = 1$. Considering $p_j^{m,\,s}$ as independent events, we can calculate the overal probabilities of beliefs:

$$p_j^s(a) = K_j^s \prod_m p_j^{m,\,s}(a), \tag{6.12}$$

where $K_j^s\ :\ p_j^s(0) + p_j^s(1) = 1.$ and update the codeword estimation:

$$\widehat{c}_j^s = \begin{cases} 0 & p_j^s(0) > p_j^s(1) \\ 1 & \text{otherwise} \end{cases} \tag{6.13}$$

If $\widehat{\mathbf{c}}^s \mathbf{H}^\top = \mathbf{0}$, decoding is terminated as successful. If not, it continues with the following calculations:

$$q_{ij}^{m,\,s+1}(a) = K_{ij}^{m,\,s+1} p_j^{m,\,0}(a) \prod_{i' \in \mathcal{M}_j^m \backslash i} r_{i'j}^{m,\,s}(a) \tag{6.14}$$

where $K_{ij}^{m,\,s+1}\ :\ q_{ij}^{m,\,s+1}(0)+q_{ij}^{m,\,s+1}(1) = 1$, and moves to Eq. (6.9) and (6.10) with the incremented iteration index s. If decoding is not successful after a given maximum number of iterations, it is terminated. We denote the method decribed above as MLDPC decoding. The state-of-the art LDPC decoding is shown in Fig. 6-6 and the proposed MLDPC decoding is depicted in Fig. 6-7

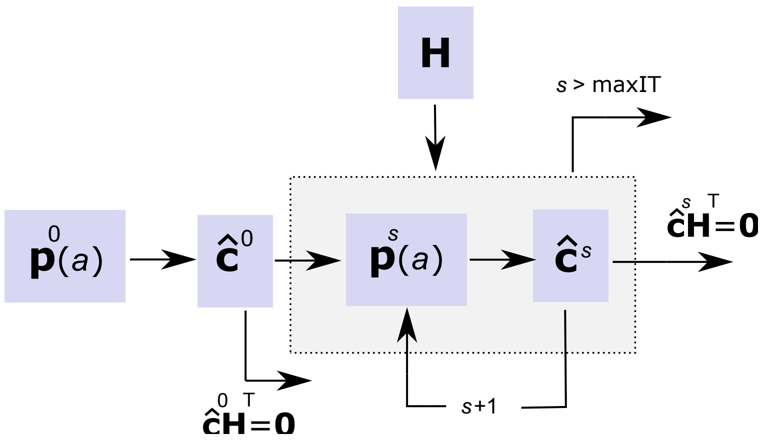


Figure 6-6: State-of-the art LDPC decoding scheme.

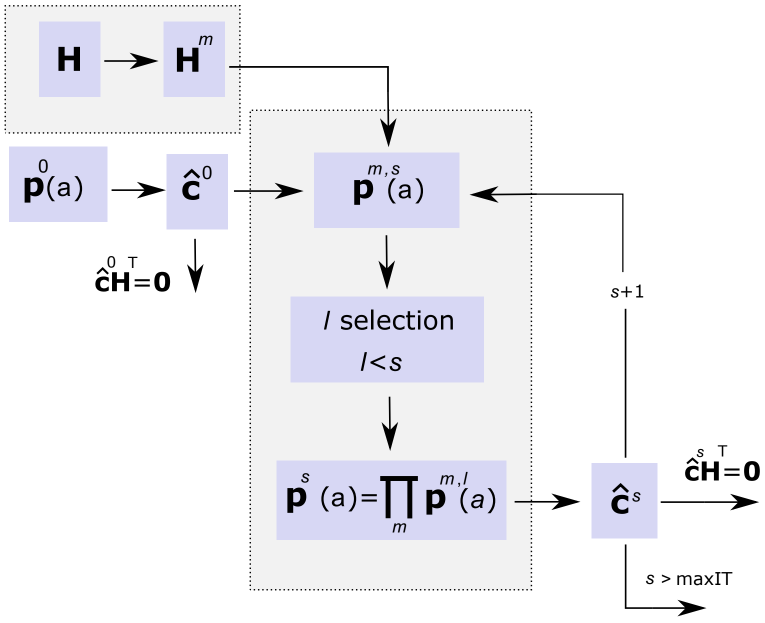


Figure 6-7: The proposed MLDPC scheme.

6.2.2 Entropy based algorithm (MLDPCe)

Using the iteration index l, explained below, we modify formula (6.12) to get the Overall Probability Vector (OPV),

$$p_j^s(a) = K_j^s \prod_m p^{m,l^m}(a) \tag{6.15}$$

We define the probability that the received bit was incorrect after s-th iteration as follows:

$$w_j^{m,\,s} = \begin{cases} p_j^{m,\,s}(1) & p_j^0 < 0.5 \\ 1 - p_j^{m,\,s}(1) & p_j^0 \geq 0.5 \end{cases} \tag{6.16}$$

After normalization:

$$\overline{w_j}^{m,\,s} = \frac{w_j^{m,\,s}}{\sum_{j'=1}^{n} w_{j'}^{m,\,s}} \tag{6.17}$$

Then, the entropy S of m-th decoder after s iterations is defined as follows:

$$S^m(s) = -\sum_j \overline{w_j}^{m,\,s} \log \overline{w_j}^{m,\,s} \tag{6.18}$$

Considering the entropy difference given by

$$\Delta S^m(t) = S^m(t) - S^m(t-1) \tag{6.19}$$

we define the iteration index l^m as:

$$l^m : \text{argmin } \left(\delta\left(\text{sgn}(\Delta S(t)) - 1\right) + 1\right) |\Delta S| \tag{6.20}$$

where $t \leq s$ and δ is the delta function. Then, the index l^m corresponds to the iterations index, where the entropy difference is minimal and not positive. This algorithm is said to be Algorithm MLDPCe.

6.2.3 Metric based algorithm (MLDPCr)

We suggest to define a radius as an approximate distance of the codeword from the received vector. Considering the initial probaiblity vector $\mathbf{p}^0$ and the initial estimation $\widehat{\mathbf{c}}^0$, the radius is given by the metric:

$$R = \sum_j |p_j^0 - \widehat{c}_j^0| \tag{6.21}$$

Considering the distance $d(\widehat{c}^0, \widehat{c}^s)$, given as the number of different bits in two input binary vectors, we define the iteration index l for MLDPCr as follows:

$$l^m = \underset{t}{\operatorname{argmin}} \left| R - d(\widehat{c}^0, \widehat{c}^t) \right| \tag{6.22}$$

Algorithm 16 Message passing in MLDPC algorithm

```
1: procedure VALUES TO CHECK NODES
   Input: p, r
   Output: q
   ▷ M is the number of mutational decoders,
   $\mathcal{M}$ and $\mathcal{N}$ are defined in Section II
2:     for all $m \in [0, \mathrm{M}-1]$ do
3:         for all $j \in [0, |\mathcal{M}|)$ do
4:             for all $i \in [0, |\mathcal{N}|)$ do
5:                 $q_{ij}^m(a) = p_j^0(a)$
6:                 for all $i' \in \mathcal{M}_j \setminus i$ do
7:                     $q_{i'j}^m(a) = q_{ij}^m(a) r_{i'j}^{m,s}(a)$
8:                 end for
9:             end for
10:        end for
11:    end for
12: end procedure
13: procedure VALUES TO VARIABLE NODES
   Input: q
   Output: r
14:    for all $m \in [0, \mathrm{M}-1]$ do
15:        for all $j \in [0, |\mathcal{M}|)$ do
16:            for all $i \in [0, |\mathcal{N}|)$ do
17:                $r_{ij}^m(0) = 1$, $r_{ij}^m(1) = 1$
18:                for all $j' \in \mathcal{N}_i \setminus j$ do
19:                    $r_{ij}^m(0) = r_{ij}^m(0)(1 - 2q_{ij'}^m(1))$
20:                    $r_{ij}^m(1) = r_{ij}^m(1)(1 - 2q_{ij'}^m(0))$
21:                end for
22:                $r_{ij}^m(0) = 1/2 + 1/2 r_{ij}^m(0)$
23:                $r_{ij}^m(1) = 1 - r_{ij}^m(1)$
24:            end for
25:        end for
26:    end for
27: end procedure
```

The performance of the proposed MLDPC decoding was compared using several widely known LDPC codes, generated according to WiMAX [94] and CCSDS [93]

Algorithm 17 Modified soft-decision decoding

```
1: procedure DecodeLDPC                                    ▷ SP algorithm
   Input: $\mathbf{y}$ – output from a demodulator
   $\max_{\mathrm{IT}}$ – maximum number of iterations
   Output: $\widehat{\mathbf{c}}$
2:     $\mathbf{q} = \mathbf{p}$
3:     $\mathbf{r}$ =Values to Variable Nodes($\mathbf{q}$)
4:     $\widehat{\mathbf{c}}$ =Calculate Estimation($\mathbf{r}$)
5:     if $\widehat{\mathbf{c}}\mathbf{H}^{\top} = \mathbf{0}$ then    return $\widehat{\mathbf{c}}$
6:     end if
7:     for $s \in (1, \max_{\mathrm{IT}})$ do
8:         $\mathbf{q}^s$ =Values to Check Nodes($\mathbf{r}^s$)
9:         $\mathbf{r}^s$ =Values to Variable Nodes($\mathbf{q}^s$)
10:        $\mathbf{p}^s$ =Update Probabilities($\mathbf{p^0}, \mathbf{r^s}$)
11:        $\mathbf{l}^s$ =Selection($\mathbf{p^0}, \mathbf{p^1}, ..., \mathbf{p^s}$)
   ▷ Selection according to
   the criterea (MLDPC, MLDPCe, MLDPCr)
12:        $\widehat{\mathbf{c}}$ =Calculate Estimation($\mathbf{p}, \mathbf{l^s}$)
13:        if $\widehat{\mathbf{c}}\mathbf{H}^{\top} = \mathbf{0}$ then    return $\widehat{\mathbf{c}}$
14:        end if
15:    end for
16: end procedure
```

Algorithm 18 Function for updating probabilities

```
1: procedure Update Probabilities
   Input: $\mathbf{p^0}$, $\mathbf{r}$
   Output: $\mathbf{p}$
2:     for all $m \in [0, k)$ do
3:         for all $j \in [\mathbf{0}, |\mathcal{M}|)$ do
4:             $p_j^m(a) = p_j^0(a)$
5:             for all $i \in \mathcal{M}_j$ do
6:                 $p_j^m(a) = p_j^m(a) r_{ij}^m(a)$
7:             end for
8:         end for
9:     end for
10: end procedure
```

standards. Performance results are shown in Fig. 6-8, 6-10, 6-11, and Table 6.1 respectively. The most significant improvement can be seen in the error floor region and for the MLDPCe algorithm. A comparison for different codes is shown in Table 6.1, where the MLDPCe algorithm also provides the lowest Bit Error Rate. A

Algorithm 19 Function for calculating the estimation

```
1: procedure CALCULATE ESTIMATION
     Input: $\mathbf{p}$ – all probabilities
   $\mathbf{l^s}$ – indices from $m$-th decoder, where $m \in 1...M$
     Output: $\widehat{\mathbf{c}}$
2:     for all $j \in [\mathbf{0}, |\mathcal{M}|)$ do
3:         $Q_j(a) = p_j^0(a)$
4:         for all $m \in [0, \mathrm{M})$ do
5:             $l = \mathbf{l}(m)$
6:             $Q_j(a) = Q_j(a) p_j^{m,l}$
7:         end for
8:         if $Q_j(a) > Q_j(a)$ then    $\widehat{c}_j = 0$
9:         else  $\widehat{c}_j = 1$
10:        end if
11:    end for
12: end procedure
```

demonstrative example of the entropy evolution is depicted in Fig. 6-12, 6-13.

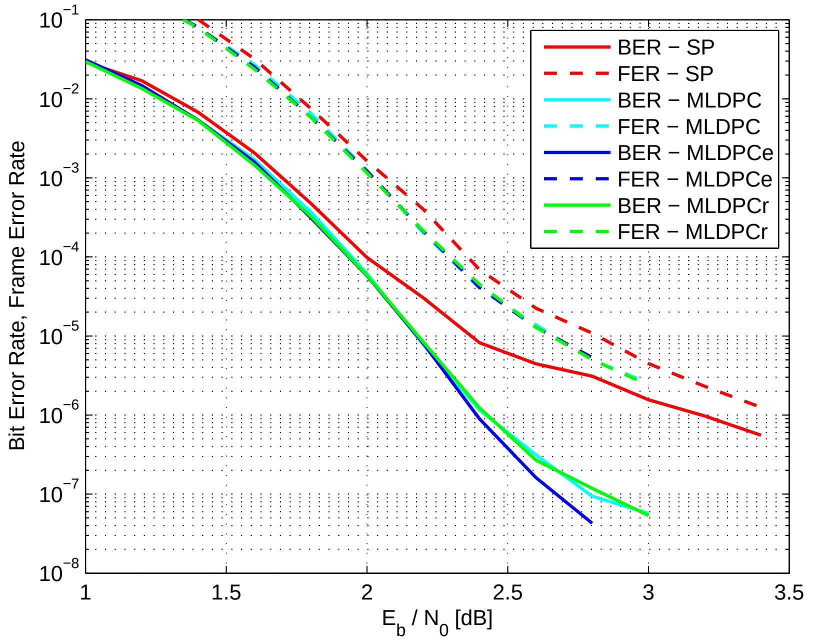


Figure 6-8: Simulation on WiMAX 1056 code, 4 additional decoders were used.

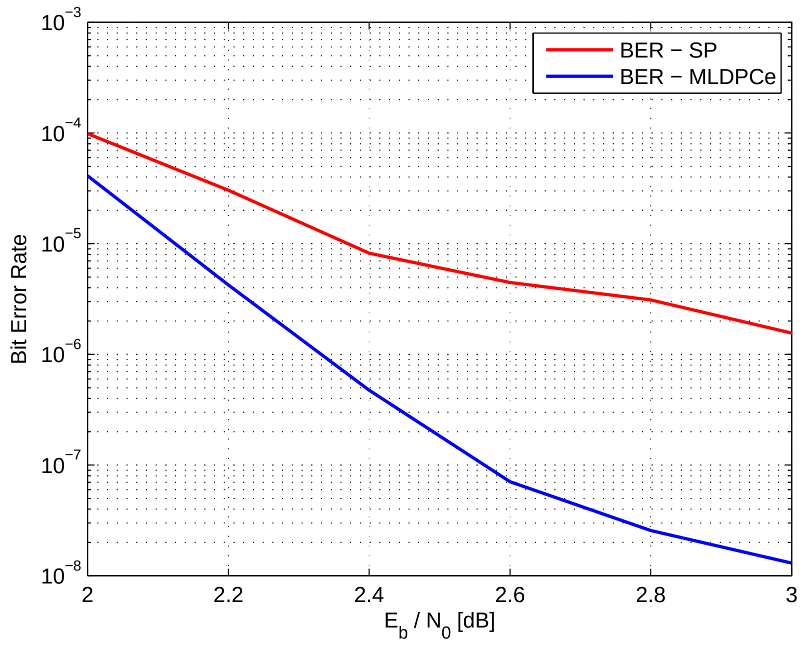


Figure 6-9: Simulation on WiMAX 1056 code, 19 additional decoders were used.

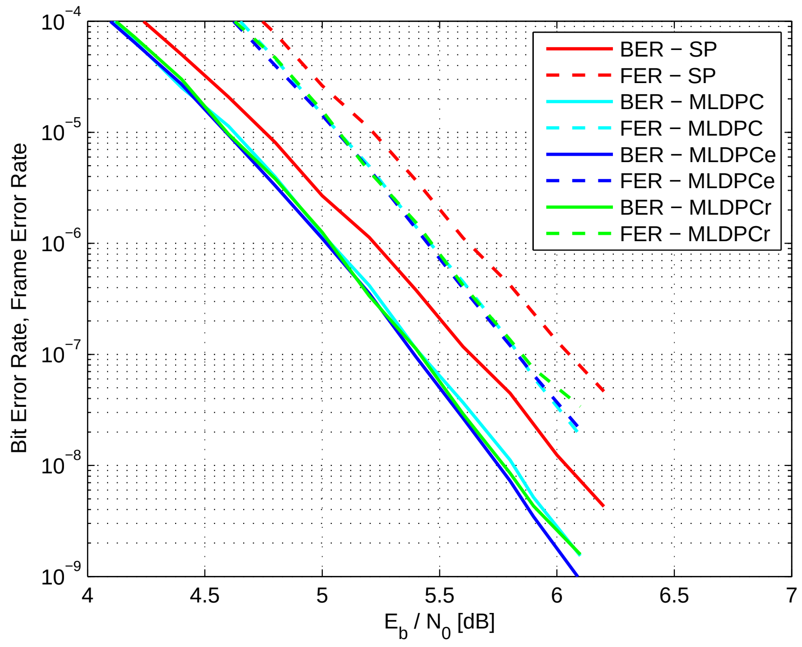


Figure 6-10: Simulation on CCSDS 128 code, 4 additional decoders were used.

6.2.4 Memory and complexity requirements

The algorithm requires several number of decoders working independently with mutated matrices and it produces one codeword estimation based on all decoders. There-

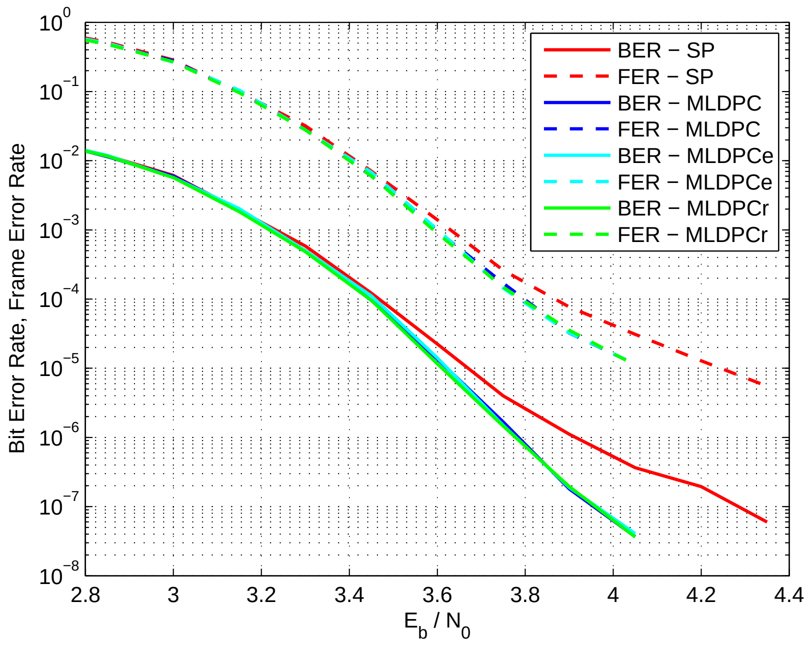


Figure 6-11: Simulation on WiMAX 2304 code, 4 additional decoders were used.

fore, the complexity increases linearly with the number of mutants. Using entropy and radius adds a non-neglible calculations, but keeps the lineariy.

Memory requirements depends on the decoders implementation and grows with the number of mutants linearly. Using entropy or radius requires the best value, the best iteration index, and associated probability vector to be stored.

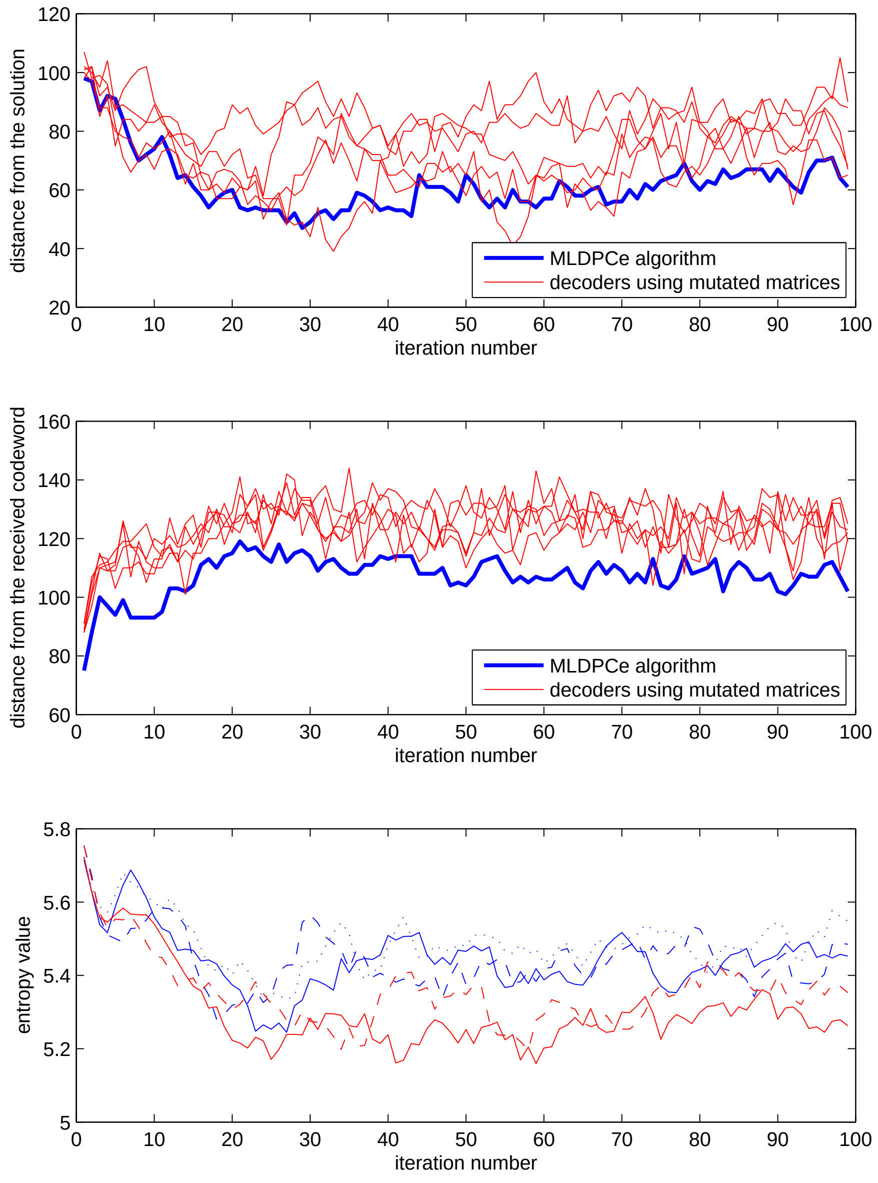


Figure 6-12: The Evolution of entropy values and distance values when the decoder fails in the convergency.

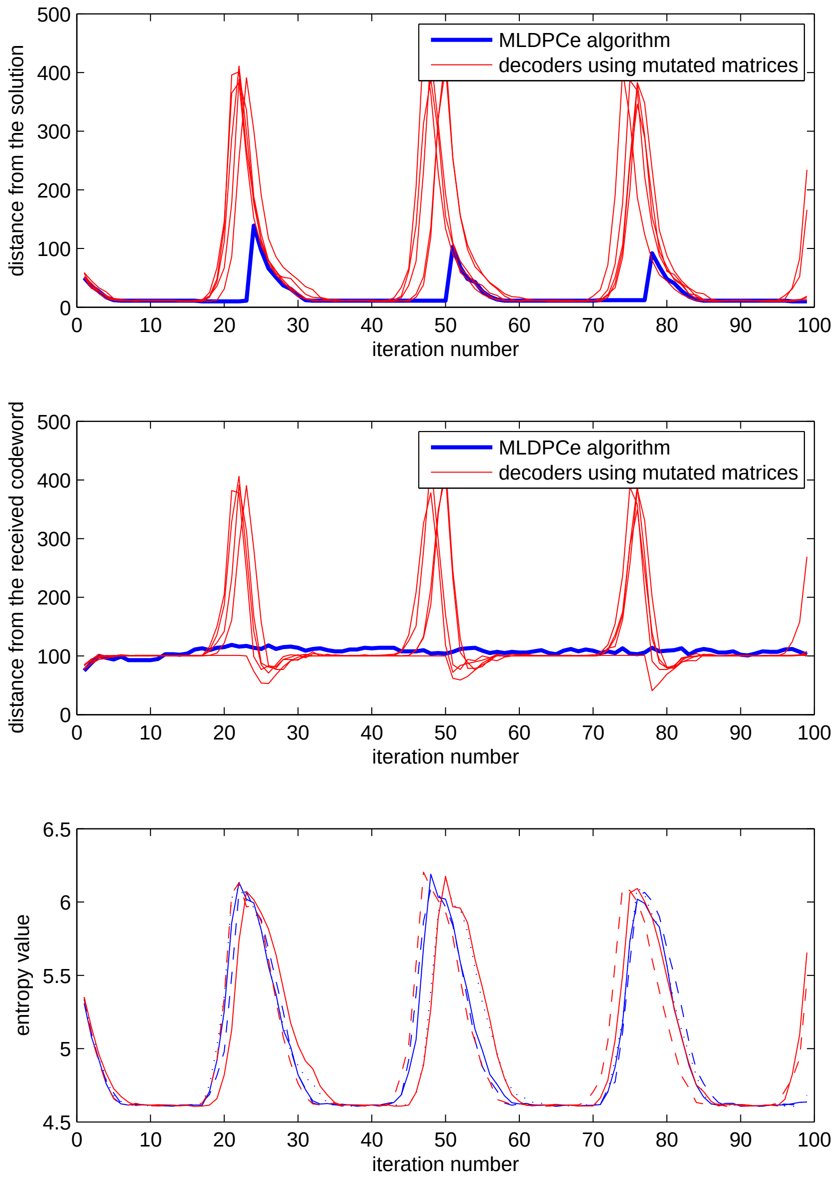


Figure 6-13: The Evolution of entropy values and distance values when the decoder oscillates.

Table 6.1: Simulated Bit Error Rate values. There were 4 additional mutational decoders used for simulations. $R = k/n$ is the code rate. Similarly, 4 additional decoding attemps were used for dithered algorithm [32]

WiMAX 1056, $R = 0.5$

$E_b/N_0 = 2.8$ dB	BER	FER
SP	3.11e-6	1.09e-5
SP+dithered	3.88e-7	8.58e-6
MLDPC	9.39e-8	4.99e-6
MLDPCe	4.29e-8	5.35e-6
MLDPCr	1.19e-7	5.02e-6
MS	7.27e-7	1.44e-5
MS+dithered	6.85e-7	8.30e-6

WiMAX 2304, $R = 0.83$

$E_b/N_0 = 4.05$ dB	BER	FER
SP	3.65e-6	3.10e-5
SP+dithered	4.13e-7	3.07e-5
MLDPC	4.08e-8	1.15e-5
MLDPCe	3.68e-8	1.14e-5
MLDPCr	3.63e-7	1.11e-5
MS	5.17e-7	3.54e-5
MS + dithered	8.24e-7	2.72e-5

CCSDS 128, $R = 0.5$

$E_b/N_0 = 5.8$ dB	BER	FER
SP	4.48e-8	4.20e-7
SP+dithered	4.43e-8	4.24e-7
MLDPC	1.12e-8	1.31e-7
MLDPCe	7.25e-9	1.20e-7
MLDPCr	8.51e-9	1.35e-7
MS	6.74e-6	7.21e-7
MS+dithered	1.91e-8	1.46e-7

Chapter 7

Conclusions

This thesis was focused on several issues related to Low Density Parity-Check (LDPC) codes and proposed several novel methodologies in the topics of code construction algorithms, decoder architectures and improving performance of state-the-art decoders. The main research contribution can be summarized as follows:

- Novel code construction techniques, especially genetic optimization algorithms.
- A method for mapping a decoder onto parallel architectures.
- Two unique algorithms for improving performance of state-of-the art LDPC decoders.

LDPC code construction techniques, introduced in this work, involve the algorithms generating Tanner graphs of controlled girth, applications of genetic optimization algorithms, and applications of coarse grained parallelism for genetic algorithms to accelerate their convergence.

Very interesting results were pointed out for different code lengths and different redundancies using the construction method proposed. The algorithm of the control girth has been shown to be a very useful method for constructing a wide variety of LDPC codes. LDPC codes of block lengths in the range from 64 to 2048 bits and redundancies in the range from 10% to 75% were constructed using this algorithm. For all tested lengths and redundancies, the algorithm provides a powerful construction method.

The proposed genetic optimization algorithms use the mutation and crossover operators, various fitness functions, utilization of elitism to keep the best solutions, and coarse grained parallelization for accelerating the convergence. These algorithms have been applied to the optimization of short blocks length codes for different optimization parameters. The interesting result was shown in the optimization of (128,64) code to provide the best performance using 10 decoding iterations. The code gain between the codes before and after the optimization was 0.4 dB at the Bit Error Rate (BER) value of 10^{-7}. Compared to the NASA CCSDS standard, the optimized code achieved the value E_B/N_0 better by 0.15 dB at $BER = 10^{-7}$. The LDPC (120,88) code, also shown in the illustrative optimization task, was compared against the RS code using the same length, where the constructed LDPC code provided a gain greater than 2 dB.

Another important contribution of this work is a parallel implementation of the decoder for any irregular LDPC code. The parallel architecture is particularly useful for hardware implementations using GPUs and FPGAs. Associated benchmarks were performed on GPU platform using OpenCL and CUDA frameworks. Using CCSDS code and its protographically expanded derivations, the GPU implementation were running up to 25 times faster compared to the serial implementation.

A significant novelty is offered in the methods introduced for improving performance and lowering BER of LDPC decoders. Two main methods were introduced; the Belief Propagation based on the estimation BackTracking and the Mutational LDPC decoding (MLDPC). The Mutational LDPC decoding further utilizes the information entropy (denoted as MLDPCe) and so called radius (denoted as MLDPCr) to control the convergence. For tested codes and given input parameters, the MLDPCe algorithm achieved 100 times lower BER compared to the state-of-the art algorithms when using the WiMAX code.

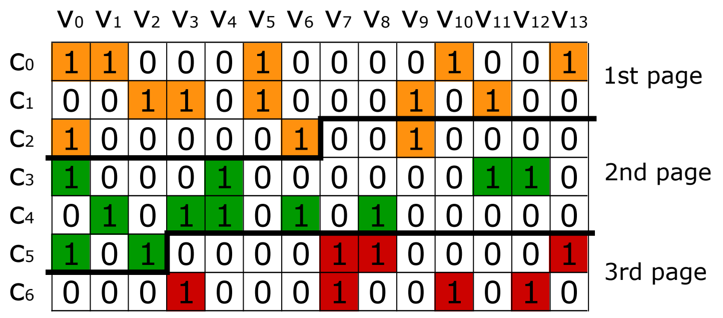
The possible research work can continue towards unique applications in hardware architectures, using neural networks for decoders, optimization of long block length codes, applications with nonbinary codes, and other related issues.

Appendix A

On GPU implementation of LDPC decoder

In this chapter, the principle of edge-level parallelization, used for GPU decoding, is shown in an example. All variables in the example are consitent with the terminology introduced in Chapter 5. The illustrated example is supported by consistent figures associated with the same LDPC (14,7) code.

Considering the code given by the parity-check matrix (Fig. A-1) and associated Tanner graph, the following arrays are defined as address iterators for the parallel message passing algorithm (described in Algorithms 9, 10 and 11). All arrays are listed in Table A.1 and B.2. Both tables are particularly useful for understanding the principle and checking the correctness of the implementation. To keep the consistency and for tutorial purposes, both tables are associated with the LDPC (14,7) code given by the parity-check matrix from Fig. A-2.

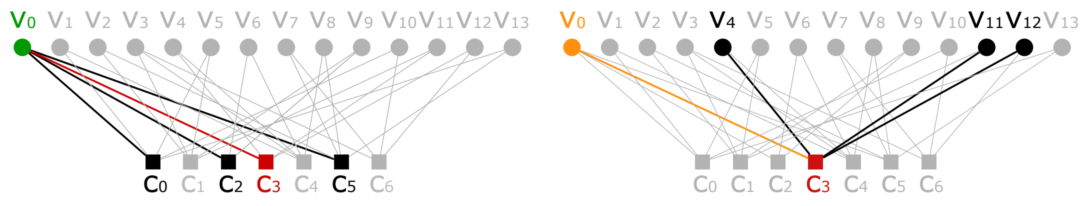


	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}
c_0	1	1	0	0	0	1	0	0	0	0	1	0	0	1
c_1	0	0	1	1	0	1	0	0	0	1	0	1	0	0
c_2	1	0	0	0	0	0	1	0	0	1	0	0	0	0
c_3	1	0	0	0	1	0	0	0	0	0	0	1	1	0
c_4	0	1	0	1	1	0	1	0	1	0	0	0	0	0
c_5	1	0	1	0	0	0	0	1	1	0	0	0	0	1
c_6	0	0	0	1	0	0	0	1	0	0	1	0	1	0

1st page
2nd page
3rd page

Figure A-1: Parity-check matrix divided into pages

Figure A-2: Parity-check matrix and the principle of the parallelization


V0 V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13
C0 C1 C2 C3 C4 C5 C6
V0 V1 V2 V3 V4 V5 V6 V7 V8 V9 V10 V11 V12 V13
C0 C1 C2 C3 C4 C5 C6

(a) The first half of the iteration - from variable nodes to check nodes. Values used for the calculation of the message between v_0 and c_3 are highlighted.

(b) The second half of the iteration - from check nodes to variable nodes. Values used for the calculation of the message between c_3 and v_0 are highlighted.

Figure A-3: Tanner graph of the LDPC (14,7) code.

Table A.1: Addresses used for message calculation outgoing from variable nodes.

array	**values**																														
e	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
v	0	0	0	0	1	1	2	2	3	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10	11	11	12	12	13	13
c	5	3	2	0	4	0	5	1	6	4	1	4	3	1	0	4	2	6	5	5	4	2	1	6	0	3	1	6	3	5	0
t	4	4	4	4	2	2	2	2	3	3	3	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
s	0	0	0	0	4	4	6	6	8	8	8	11	11	13	13	15	15	17	17	19	19	21	21	23	23	25	25	27	27	29	29
u	0	1	2	3	0	1	0	1	0	1	2	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Table A.2: Addresses used for message calculation outgoing from check nodes.

array	**values**																														
$\overline{\mathbf{e}}$	3	5	14	24	30	7	10	13	22	26	2	16	21	1	12	25	28	4	9	11	15	20	0	6	18	19	29	8	17	23	27
$\overline{\mathbf{v}}$	0	1	5	10	13	2	3	5	9	11	0	6	9	0	4	11	12	1	3	4	6	8	0	2	7	8	13	3	7	10	12
$\overline{\mathbf{c}}$	0	0	0	0	0	1	1	1	1	1	2	2	2	3	3	3	3	4	4	4	4	4	5	5	5	5	5	6	6	6	6
$\overline{\mathbf{t}}$	5	5	5	5	5	5	5	5	5	5	3	3	3	4	4	4	4	5	5	5	5	5	5	5	5	5	5	4	4	4	4
$\overline{\mathbf{s}}$	0	0	0	0	0	5	5	5	5	5	10	10	10	13	13	13	13	17	17	17	17	17	22	22	22	22	22	27	27	27	27
$\overline{\mathbf{u}}$	0	1	2	3	4	0	1	2	3	4	0	1	2	0	1	2	3	0	1	2	3	4	0	1	2	3	4	0	1	2	3

Listing A.1: Types

```
typedef struct Edge{
   int index; // e array
   int vn;    // v array
   int cn;    // c array
   int edgesConnectedToNode;  // t array
   int absoluteStartIndex;    // s array
   int relativeIndexFromNode; // u array
} Edge;

typedef struct EdgeData{
   double passedValue;
} EdgeData;

typedef struct CodeInfo{
   int totalEdges;  // number of edges
   int varNodes;    // number of variable nodes
   int checkNodes;  // number of check nodes
} CodeInfo;
```

Table A.3: Comparison of chosen OpenCL and CUDA syntax rules

command	**OpenCL**	**CUDA**
thread synchronization	`barrier(CLK_GLOBAL_MEM_FENCE);`	`__syncthreads();`
kernel prefix	`__kernel`	`__global__`
local memory prefix	`__local`	`__shared__`
get local ID	`int lid = get_local_id(0);`	`int lid = threadIdx.x;`
get global ID	`int gid = get_global_id(0);`	`int gid = blockIdx.x * blockDim.x+ threadIdx.x;`

Appendix B

Fully parallel LDPC decoder implementation in FPGA

We introduce a scalable architecture for decoding irregular LDPC codes. The architecture allows easy deployability in Field Programmable Gate Arrays (FPGAs) and it supports a wide variety of LDPC codes. The main limitation is the size of the FPGA circuit. The architecture is described in the following section.

The aim is to design the architecture for decoding a wide variety of irregular LDPC codes. The codes are generated with the of a framework, which produce the Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) file. This file is than added to the source files of the decoder, which remains constant (Fig. B-1).

Suppose

$$cn = (C_0, C_1, ..., C_{N\_CN}) \tag{B.1}$$

where $C_k = k, k, ..., k$ $|C_k| = |M_k|$ and N_CN is the number of check nodes.

Tuple $(cn(i), vn(i))$, $i = 1, 2, ..., N\_EDGES$ defines edges in the Tanner graph Consider the tuple icn of check node input indices as the following:

$$icn = (D_{c_0}, D_{c_1,}, ..., D_{|cn|}) \tag{B.2}$$

$$D_0 = 0, D_1 = 0, 1, ..., D_k = 0, 1, 2, ..., |N_k| \tag{B.3}$$

The tuple of edges $(cn(i), vn(i)$ sorted ascending by cn and the transformation:

$$S \,:\, (cn(i), vn(i)) \to (cn'(i), vn'(i)) \tag{B.4}$$

where $(cn'(i), vn\text{'}(i))$ is sorted ascending by vn'.

The tuple of variable input indices ivn'=$(E_0, E_1, ..., E_k)$

$$E_0 = 0, E_1 = 0, 1, ..., E_k = 0, 1, 2, ..., |M_k| \tag{B.5}$$

A transformation

$$S^{-1} \,:\, (cn'(i), vn'(i)) \to (cn(i), vn(i)) \tag{B.6}$$

The set of variable node indices ivn related to icn is then given as:

$$S^{-1} \,:\, ivn' \to ivn \tag{B.7}$$

Lengths of the tuples cn, vn, icn, ivn are equal to the number of edges in the Tanner graph.

$$|cn| = |vn| = |icn| = |ivn| = K \tag{B.8}$$

In paragraphs below, we define vectors used for the description of connections and the storage. Vectors $entityIndex1$, $entityIndex2$, $inputIndex1$, $inputIndex2$, $edgeIndex1$ and $edgeIndex2$ are used as indices for connections between entities. The vector $EDGE\_DATA$ represents memory elements for outgoing and incoming messages.

The vectors of indices are filled by the method described in Algorithm 20. The architecture of connections is depicted in Fig. B-2 and B-3. A simplified state diagram of the decoding algorithms is shown in Fig. B-4.

Algorithm 20 Filling indices

```
1:  procedure FILLING INDICES

        Input: cn, vn, icn, ivn
        Output: entityIndex1, 2, edgeIndex1, 2, inputIndex1, 2, $ctr1$, $ctr2$
2:      ctr1 = 0, ctr2 = 0
3:      for $i = 0$ to $K - 1$ do
4:          for $j = 0$ to $K - 1$ do
5:              if $cn(i) = cn(j)$ and $(i! = j)$ then
6:                  $entityIndex1(ctr1) = i$
7:                  $inputIndex1(ctr1) = icn(j)$
8:                  $edgeIndex1(ctr1) = j$
9:                  $ctr1 + +$
10:             end if
11:             if $vn(i) = vn(j)$ and $(i! = j)$ then
12:                 $entityIndex2(ctr2) = i$
13:                 $inputIndex2(ctr2) = ivn(j)$
14:                 $edgeIndex2(ctr2) = j$
15:                 $ctr2 + +$
16:             end if
17:         end for
18:     end for
19: end procedure
```

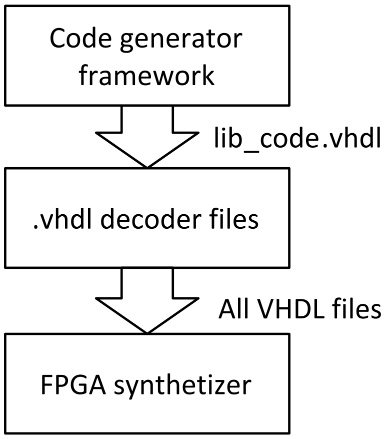


Figure B-1: Flowchart of the decoder synthesis.

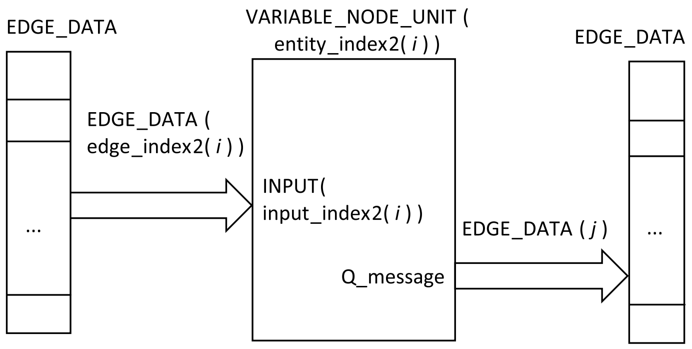

EDGE_DATA
VARIABLE_NODE_UNIT (entity_index2(i))
EDGE_DATA
EDGE_DATA (edge_index2(i))
INPUT(input_index2(i))
EDGE_DATA (j)
Q_message
...
...

Figure B-2: Variable node unit in the relation of message passing, where i is in $0...ctr1-1$, $ctr1$ is the value after running Algorithm 20 and j is in $0...K-1$

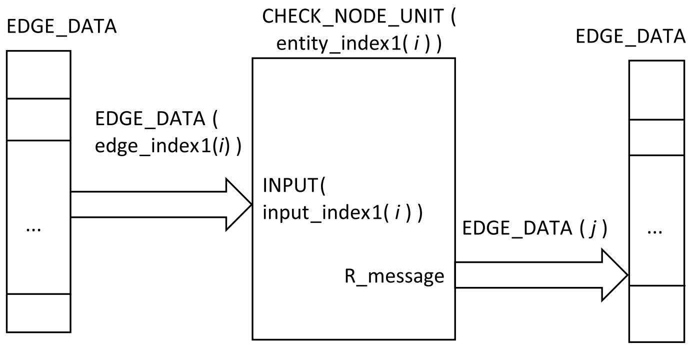

EDGE_DATA
CHECK_NODE_UNIT (entity_index1(i))
EDGE_DATA
EDGE_DATA (edge_index1(i))
INPUT(input_index1(i))
EDGE_DATA (j)
R_message
...
...

Figure B-3: Check node unit in the relation of message passing, where i is $0...ctr2-1$, $ctr2$ is the value after running Algorithm 20 and j is $0...K-1$

The functionality of the decoder was tested by the simulations of the decoder including generated codes by our framework. Syntheses of decoders for particular codes were performed. The clock frequencies and circuit utilizations is shown in Table B.1. The circuits were chosen because of their easy availability in development

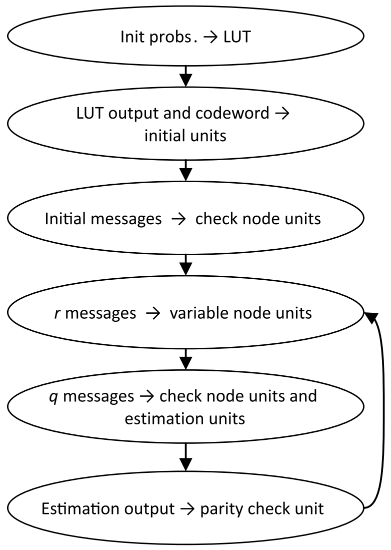

Init probs. → LUT
LUT output and codeword → initial units
Initial messages → check node units
r messages → variable node units
q messages → check node units and estimation units
Estimation output → parity check unit

Figure B-4: Simplified state diagram of the decoding algorithm.

kits [11].

Table B.1: Synthesis results. The number of required ALMs is compared for different code lengths.

STRATIX V 5SGXEA7N2F40C2N	**CLK**	**Circuit utilization**
$LDPC(32,16)$	65 MHz	9,683
$LDPC(64,32)$	103 MHz	29,152
$LDPC(128,64)$	102 MHz	57,950
$LDPC(256,128)$	102 MHz	120,730 of 234,720 ALMs

Table B.2: Addresses used for message calculation outgoing from check nodes.

array	values
cn	0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 4, 4, 4, 4, 4, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7, 7, 7, 8, 8, 8, 8, 9, 9, 9, 9, 10, 10, 10, 11, 11, 11, 11, 12, 12, 12, 12, 13, 13, 13, 14, 14, 14, 14, 15, 15, 15
vn	24, 23, 21, 17, 16, 14, 5, 29, 30, 31, 22, 18, 19, 27, 30, 3, 0, 28, 2, 20, 26, 25, 29, 15, 22, 6, 8, 15, 10, 11, 18, 5, 7, 4, 11, 19, 3, 1, 7, 28, 6, 8, 13, 17, 12, 1, 12, 26, 10, 2, 8, 23, 0, 8, 9, 27, 4, 8, 31, 11, 20, 25, 9, 11, 16, 13
icn	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 0, 1, 2, 3, 4, 0, 1, 2, 0, 1, 2, 3, 4, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 0, 1, 2, 3, 0, 1, 2
ivn	0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, 10, 11, 11, 11, 12, 12, 12, 13, 13, 13, 14, 14, 14, 15, 15, 15, 16, 16, 16, 17, 17, 17, 18, 19, 20, 20, 20, 20, 21, 21, 21, 21, 22, 22, 22, 22, 23, 23, 23, 23, 24, 24, 24, 24, 25, 25, 26, 26, 27, 27, 28, 28, 28, 28, 29, 29, 29, 29, 30, 30, 30, 30, 31, 31, 31, 31, 32, 32, 32, 32, 33, 33, 33, 34, 34, 34, 35, 35, 35, 36, 36, 36, 37, 37, 37, 38, 38, 38, 39, 39, 39, 40, 40, 40, 41, 41, 41, 42, 42, 42, 43, 43, 43, 44, 44, 44, 45, 45, 46, 46, 47, 47, 48, 48, 48, 49, 49, 49, 50, 50, 50, 51, 51, 51, 52, 52, 52, 53, 53, 53, 54, 54, 54, 55, 55, 55, 56, 56, 57, 57, 58, 58, 59, 59, 59, 60, 60, 60, 61, 61, 61, 62, 62, 62, 63, 63, 64, 64, 65, 65
entityIndex1	0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, 10, 11, 11, 11, 12, 12, 12, 13, 13, 13, 14, 14, 14, 15, 15, 15, 16, 16, 16, 17, 17, 17, 18, 19, 20, 20, 20, 20, 21, 21, 21, 21, 22, 22, 22, 22, 23, 23, 23, 23, 24, 24, 24, 24, 25, 25, 26, 26, 27, 27, 28, 28, 28, 28, 29, 29, 29, 29, 30, 30, 30, 30, 31, 31, 31, 31, 32, 32, 32, 32, 33, 33, 33, 34, 34, 34, 35, 35, 35, 36, 36, 36, 37, 37, 37, 38, 38, 38, 39, 39, 39, 40, 40, 40, 41, 41, 41, 42, 42, 42, 43, 43, 43, 44, 44, 44, 45, 45, 46, 46, 47, 47, 48, 48, 48, 49, 49, 49, 50, 50, 50, 51, 51, 51, 52, 52, 52, 53, 53, 53, 54, 54, 54, 55, 55, 55, 56, 56, 57, 57, 58, 58, 59, 59, 59, 60, 60, 60, 61, 61, 61, 62, 62, 62, 63, 63, 64, 64, 65, 65
inputIndex1	1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 0, 2, 3, 0, 1, 3, 0, 1, 2, 1, 2, 3, 0, 2, 3, 0, 1, 3, 0, 1, 2, 1, 0, 1, 2, 3, 4, 0, 2, 3, 4, 0, 1, 3, 4, 0, 1, 2, 4, 0, 1, 2, 3, 1, 2, 0, 2, 0, 1, 1, 2, 3, 4, 0, 2, 3, 4, 0, 1, 3, 4, 0, 1, 2, 4, 0, 1, 2, 3, 1, 2, 3, 0, 2, 3, 0, 1, 3, 0, 1, 2, 1, 2, 3, 0, 2, 3, 0, 1, 3, 0, 1, 2, 1, 2, 3, 0, 2, 3, 0, 1, 3, 0, 1, 2, 1, 2, 0, 2, 0, 1, 1, 2, 3, 0, 2, 3, 0, 1, 3, 0, 1, 2, 1, 2, 3, 0, 2, 3, 0, 1, 3, 0, 1, 2, 1, 2, 0, 2, 0, 1, 1, 2, 3, 0, 2, 3, 0, 1, 3, 0, 1, 2, 1, 2, 0, 2, 0, 1
edgeIndex1	1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 10, 12, 13, 10, 11, 13, 10, 11, 12, 15, 16, 17, 14, 16, 17, 14, 15, 17, 14, 15, 16, 19, 18, 21, 22, 23, 24, 20, 22, 23, 24, 20, 21, 23, 24, 20, 21, 22, 24, 20, 21, 22, 23, 26, 27, 25, 27, 25, 26, 29, 30, 31, 32, 28, 30, 31, 32, 28, 29, 31, 32, 28, 29, 30, 32, 28, 29, 30, 31, 34, 35, 36, 33, 35, 36, 33, 34, 36, 33, 34, 35, 38, 39, 40, 37, 39, 40, 37, 38, 40, 37, 38, 39, 42, 43, 44, 41, 43, 44, 41, 42, 44, 41, 42, 43, 46, 47, 45, 47, 45, 46, 49, 50, 51, 48, 50, 51, 48, 49, 51, 48, 49, 50, 53, 54, 55, 52, 54, 55, 52, 53, 55, 52, 53, 54, 57, 58, 56, 58, 56, 57, 60, 61, 62, 59, 61, 62, 59, 60, 62, 59, 60, 61, 64, 65, 63, 65, 63, 64
entityIndex2	1, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 26, 26, 26, 27, 28, 29, 29, 29, 30, 31, 32, 33, 34, 34, 34, 35, 36, 37, 38, 39, 40, 41, 41, 41, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 50, 50, 50, 51, 52, 53, 53, 53, 53, 54, 55, 56, 57, 57, 57, 57, 58, 59, 59, 59, 60, 61, 62, 63, 63, 63, 64, 65
inputIndex2	0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 3, 2, 1, 0, 1, 0, 2, 1, 0, 1, 1, 0, 0, 3, 1, 0, 1, 1, 0, 1, 1, 1, 4, 2, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 4, 3, 1, 0, 1, 1, 4, 3, 2, 0, 0, 1, 1, 4, 3, 2, 1, 1, 3, 2, 0, 1, 1, 1, 3, 2, 1, 1, 1
edgeIndex2	51, 43, 64, 31, 22, 14, 58, 24, 30, 35, 55, 8, 36, 52, 39, 49, 60, 47, 61, 7, 27, 10, 40, 41, 50, 53, 57, 23, 48, 34, 59, 63, 11, 6, 38, 56, 29, 59, 63, 12, 15, 45, 32, 17, 25, 26, 50, 53, 57, 65, 3, 46, 37, 44, 20, 28, 18, 26, 41, 53, 57, 1, 16, 26, 41, 50, 57, 62, 13, 33, 26, 41, 50, 53, 9, 29, 34, 63, 19, 21, 54, 29, 34, 59, 4, 42

Appendix C

Burst correctability of optimized codes

In this Chapter, results for multiple-burst error capabilities are presented.

The burst error pattern, used as an optimization parameter for fitness function, is given by $E_P = (x_1, x_2, ..., x_k)$, where x_i are lengths of error bursts in the codeword and $|E_P|$ is the number of burst errors (as proposed in Section 4.2). The total number of corrupted bits is then

$$\sum_{i=1}^{k} E_P(i). \tag{C.1}$$

This fitness function provides deterministic evaluation and can be calculated in a reasonable period of time.

Tables C.1 -C.4 show the results of the best optimized codes. The optimization tasks were held for 100 generations and the sharing period was 10 generations. The correction performance for double burst error correction measured for different codes and different burst lengths are given in Tables C.1-C.4.

Table C.1: Correcting capabilities measured for ultra short block lengths

variable nodes	check nodes	R	burst lengths		
			(1,1)	(2,1)	(3,1)
64	16	0.25	71.5	13.6	8.5
			(3,2)	(4,2)	(4,3)
64	32	0.5	90.7	59.3	33.4
			(5,4)	(6,5)	(7,6)
64	48	0.75	98.4	87.3	48.4

Table C.2: Correcting capabilities measured by the ratio of frames which are able to be corrected by the decoder

variable nodes	check nodes	R	burst lengths									
			(1,1)	(2,1)	(3,1)	(3,2)	(4,2)	(4,3)	(5,3)	(5,4)	(6,4)	(6,5)
128	32	0.25	98.8	90.1	55.8							
128	40	0.3125		98.1	91.7	66.7						
128	48	0.375		99.6	98.2	95.3						
128	56	0.375				98.8	96.8	93.4	81.2	52.4		
128	64	0.5				100	99.8	99.2	96.2	89.7	77.5	61.9
			(5,3)	(5,4)	(6,4)	(6,5)	(7,5)	(7,6)	(8,6)	(8,7)	(9,7)	(9,8)
128	72	0.5625	99.7	99.4	98.0	93.9	84.5	71.0				
128	80	0.625			100.0	99.5	98.6	94.8	87.3	74.7	55.1	40.0
			(7,5)	(7,6)	(8,6)	(8,7)	(9,7)	(9,8)	(10,8)	(10,9)	(11,9)	(11,10)
128	88	0.6875	100.0	99.6	98.2	97.1	93.8	89.6	78.0	60.2		
128	96	0.75		100	100	99.9	99.8	99.0	98.2	95.1	88.9	80.1

Table C.3: Correcting capabilities measured by the ratio of frames which are able to be corrected by the decoder

variable nodes	check nodes	R	burst lengths							
			(1,1)	(2,1)	(3,1)	(3,2)	(4,2)	(4,3)	(5,3)	(5,4)
256	56	0.21875								
256	64	0.25	99.7	97.9	93.6	84.5	59.8	29.4	6.3	
			(4,2)	(4,3)	(5,3)	(5,4)	(6,4)	(6,5)	(7,5)	(7,6)
256	80	0.3125	96.6	93.3	88.6	75.6	51.5	23.4	10.7	
256	96	0.375		99.8	98.4	95.9	89.9	86.6	74.5	49.9
			(5,4)	(6,4)	(6,5)	(7,5)	(7,6)	(8,6)	(8,7)	(9,7)
256	112	0.4375	99.8	99.7	99.5	97.2	94.3	91.6	85.2	73.3
			(6,5)	(7,5)	(7,6)	(8,6)	(8,7)	(9,7)	(9,8)	(10,8)
256	128	0.5	100	100	99.8	99.6	98.9	97.3	95.8	92.8
256	144	0.5625					100	100	99.8	99.5
			(10,8)	(10,9)	(11,9)	(11,10)	(12,10)	(12,11)	(13,11)	(13,12)
256	160	0.625	100	99.9	99.9	99.7	99.4	99.1		

Table C.4: Correcting capabilities measured by the ratio of frames which are able to be corrected by the decoder

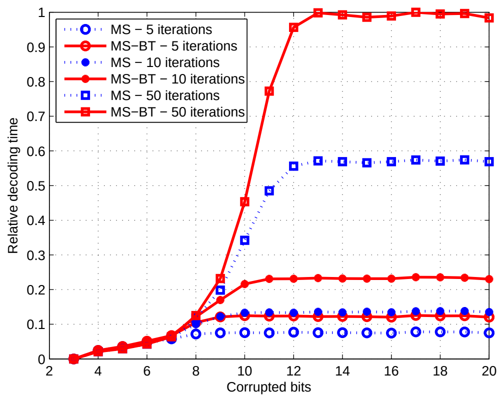
variable nodes	check nodes	R	burst lengths								
			(2,1)	(3,1)	(3,2)	(4,2)	(4,3)	(5,3)	(5,4)	(6,4)	(6,5)
512	128	0.25	100	100	99.3	99.4	98.3	95.5	95.6	90.1	85.5
512	160	0.3125			99.9	99.7	99.4				
512	192	0.375									
			(9,7)								
512	256	0.5	100	99.9	98.2						
			(2,1)	(3,1)	(3,2)	(4,2)	(4,3)	(5,3)	(5,4)	(6,4)	(6,5)
1024	128	0.125		90.4							
1024	256	0.25			99.7	99.6	99.1				
1024	320	0.325		99.9							

Appendix D

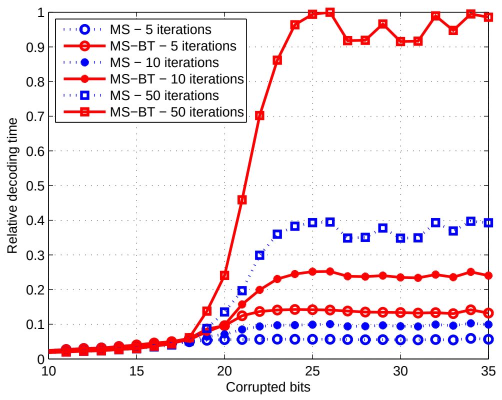
On Belief Propagation based on the Estimation Backtracking

Decoding time measured on CPU

The time complexity of the decoder was measured on the CPU implementation. The time is compared relatively for the MS decoding and MS-BT decoding. Results measured for (128,64) code, (256,128) code and (504,252) code can be seen in figures below. All results are normalized to relative values.



(a) Relative decoding time vs the number of corrupted bits in a codeword measured on the LDPC(128,64) code.



(b) Relative decoding time vs the number of corrupted bits in a codeword measured on the LDPC(504,252) code.

Figure D-1: Relative decoding time

Hardware implementation

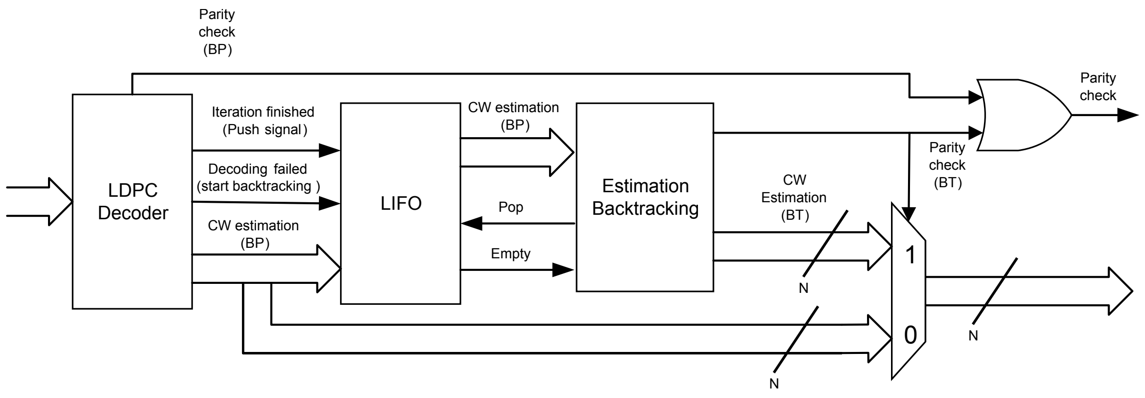


Figure D-2: Hardware architecture outline.

In this paragraph, we provide an outline of the hardware architecture, which is shown in Fig. D-2. The architecture consists of several parts, including the BP LDPC decoder, Last-In-First-Out (LIFO) memory and a block devoted for the implementation of the proposed backtracking algorithm. The multiplexer, located at the output, switches between the BP decoder output and the output outgoing from the block performing the estimation backtracking. Switching is controlled by the parity-check signal, which is set to 1 when all bits in the estimation meet the parity-check conditions given by the matrix $\mathbf{H}$. A syndrome of the codeword estimation is equal to the zero vector in such case. In practical implementations, the architecture can be pipelined in order to increase the throughput. Then, the estimation backtracking block is calculating the estimation of the codeword received before the codeword being processed by the decoder at the same time. Furthermore, when the throughput is crucial, the estimation backtracking block can be bypassed by the multiplexer at the output. The behavior of the architecture is then simplified and only the Estimation Backtracking is not performed in such cases.

Error correction capabilities

The error correcting capabilities were simulated on MacKay's widely known code (504,252) [36], and irregular code (128,64) [10], irregular code (256,128) provided by according to the CCSDS standard [93]. The error correcting capabilities are compared to the number of corrupted bits and the number of iterations performed by the Min-Sum (MS) decoder for particular codes respectively. This can be seen in Tables D.1, D.2, D.3. It should be highlighted that the number of iterations needed to be performed by the decoder in order to achieve the same Bit Error Rate is significantly decreased in several cases by using the proposed MS-BT algorithm. For (256,128) code, the MS-BT algorithm with the use of 5 iterations outperformed the traditional MS decoder with the use of 10 iterations in all tested cases (Table II). Furthermore, the MS-BT using 5 iterations outperformed the MS algorithm using 50 iterations for codewords with 7-11 corrupted bits. Similarly, for MacKay's (504,252) code, the MS-BT at 10 iterations outperformed the MS algorithm at 50 iterations in all tested cases. Although this behavior was not observed for our short-length (128,64) code, the MS-BT algorithm improved the error correcting capability significantly, with the maximum improvement of 10% for certain number of corrupted bits.

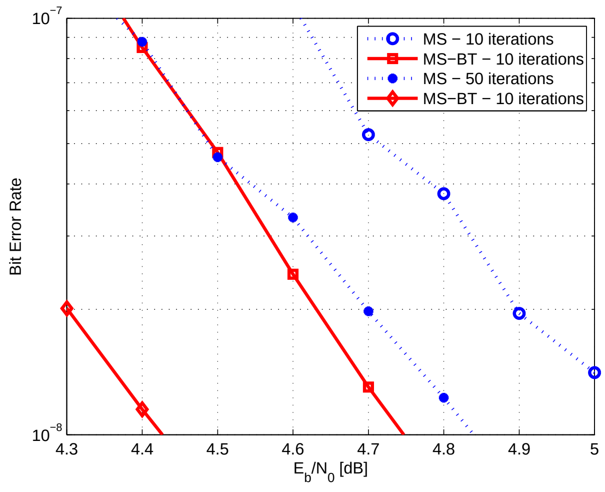


Figure D-3: Bit error rate simulated on irregular MacKay's (504,252) code compared for different number of iterations. The values are calculated from 300 codewords for which the iterative decoding algorithm failed at particular Signal-to-Noise Ratios.

Table D.1: Error correcting capability measured on our (128,64) LDPC code.

		error correcting capability (%)	
n-bit error	iterations	MS	MS-BT
4	5	99.25	99.76
5	5	94.14	96.93
6	5	80.75	89.12
7	5	36.36	46.36
8	5	11.75	16.67
9	5	1.83	3.17
10	5	0.10	0.32
11	5	0.03	0.04
4	10	99.90	99.94
5	10	99.11	99.47
6	10	94.97	97.28
7	10	76.00	81.01
8	10	48.88	55.96
9	10	17.53	21.97
10	10	2.34	3.07
11	10	0.19	0.27
4	50	99.99	99.998
5	50	99.56	99.97
6	50	99.66	99.74
7	50	98.03	98.34
8	50	90.73	92.83
9	50	68.85	74.34
10	50	32.12	34.51
11	50	6.93	7.73

Table D.2: Error correcting capability measured on NASA CCSDS (256,128) LDPC code.

		error correcting capability (%)	
n-bit error	iterations	MS	MS-BT
6	5	99.34	99.52
7	5	93.78	95.08
8	5	71.50	75.29
9	5	37.57	**44.88**
10	5	9.19	**11.09**
11	5	1.21	**1.66**
6	10	99.50	99.51
7	10	94.60	94.99
8	10	74.38	75.30
9	10	**38.07**	**42.13**
10	10	**10.51**	**11.23**
11	10	**1.50**	**1.59**
6	50	99.52	99.53
7	50	94.79	95.25
8	50	74.65	75.61
9	50	**38.33**	43.33
10	50	**10.49**	10.83
11	50	**1.48**	1.69

Table D.3: Error correcting capability measured on MacKay's (504,252) LDPC code.

		error correcting capability (%)	
n-bit error	iterations	MS	MS-BT
16	5	93.22	97.80
17	5	84.26	94.02
18	5	55.19	76.78
19	5	44.63	68.51
20	5	16.05	36.17
21	5	5.10	16.83
16	10	98.32	99.54
17	10	95.22	**98.46**
18	10	87.44	**92.65**
19	10	73.36	**86.01**
20	10	47.74	**57.46**
21	10	26.07	*35.89*
16	50	98.36	99.58
17	50	**95.5**	98.47
18	50	**88.23**	92.72
19	50	**74.58**	86.08
20	50	**50.85**	57.66
21	50	*27.90*	36.77

Appendix E

MLDPC BER simulations

In this Section, BER simulations of MLDPC decoding are presented for selected codes and parameters. Values are compared for differend number of iterations and different number of corrupted bits.

Table E.1: Error correcting capability measured o MacKay's (504,252) LDPC code using MLDPC.

		error correcting capability (%)		
n-bit error	iterations	SP	MLDPC	MLDPCe
30	5	59.81	47.19	45.50
31	5	49.33	35.68	34.73
38	50	81.08	84.29	84.05
39	50	73.2	76.89	77.34

Table E.2: CCSDS 256, 4 additional decoders.

		error correcting capability (%)			
n-bit error	iterations	SP	MLDPC	MLDPCe	MLDPCe(19 dec)
10	5	99.74	99.70	99.61	99.69
15	5	82.63	69.16	69.02	67.00
16	5	67.18	43.43	44.06	37.91
17	5	44.22	16.13	16.35	12.41
18	5	19.15	2.80	2.79	2.15
19	5	4.66	0.24	0.27	0.21
20	5	0.57	0.03	0.03	0.01
10	50	99.99	99.98	99.98	100.00
15	50	98.51	98.58	99.07	99.67
17	50	90.91	91.13	93.13	96.61
18	50	79.92	79.53	82.70	89.70
19	50	59.34	59.07	63.14	72.23
20	50	32.30	32.27	34.42	43.79
21	50	10.89	10.97	11.86	16.20
22	50	2.10	2.08	2.32	3.34
10	100	100.00	100.00	100.00	100.00
15	100	98.91	99.47	99.36	99.78
18	100	81.55	85.71	86.06	91.06
19	100	60.81	67.01	67.12	75.01
20	100	33.53	39.05	38.80	45.52
21	100	11.83	14.47	14.72	17.92
22	100	2.32	3.06	3.21	3.51

Table E.3: LDPC (128, 64), 4 additional decoders.

		error correcting capability (%)			
n-bit error	iterations	SP	MLDPC	MLDPCe	MLDPCe(19 dec)
1	5	100.00	100.00	100.00	100.00
3	5	100.00	100.00	100.00	99.99
7	5	99.70	99.52	99.33	99.29
10	5	86.03	76.74	73.35	74.56
11	5	67.27	43.36	41.49	41.44
12	5	36.00	8.72	8.60	9.12
1	50	100.00	100.00	100.00	100.00
3	50	100.00	100.00	100.00	100.00
5	50	99.95	99.96	99.97	99.99
7	50	99.28	99.44	99.52	99.66
10	50	78.05	79.86	79.74	85.86
11	50	49.50	51.04	49.91	56.85
12	50	15.17	14.98	15.12	16.88
13	50	0.99	0.81	0.96	1.00
14	50	0.02	0.00	0.01	0.01
1	100	100.00	100.00	100.00	100.00
3	100	100.00	100.00	100.00	100.00
7	100	99.45	99.62	99.54	99.76
10	100	78.72	80.91	80.28	86.88
11	100	50.17	52.01	52.10	59.68
12	100	16.10	15.86	15.18	19.84
13	100	1.29	1.16	1.01	1.36
14	100	0.03	0.01	0.02	0.01

Appendix F

Feature based classification

The most time consuming operation in genetic optimization tasks is the calculation of fitness values, which can take prohibitively long time. Therefore, a feature based classification of Tanner graphs can be cosidered as an interesting approach to deal with this issue.

The main *features* of a Tanner graph can be:

- average cycle length,
- average degree of check nodes,
- variance of degree nodes,
- minimum degree of all nodes.

Two independent experiments were performed in order to test this approach. Experiment A was held for 50 generations, whereas Experiment B was held for 100 generations. The required Signal-to-Noisel Ratio for the Bit Error Rate of 10^{-3} was used as an evaluation function. The lower value of this function means better correction performance.

As can be seen in Fig. F-3a, the optimization algorithm converges to better codes and the best codes are located at the bounded area of check node degrees. Other figures depict variance of check node degrees and the 3D plot shows the average degree and the degree variance.

The charts in Fig. F-3b indicates the interval where the best codes in terms of the simulation used are located (the concentrated ares). This fact can be utilized to speed up an optimization algorithm. If a code is located outside of the concentrated area, the performance of the code is considered as inefficient with high probability and such a code is then dropped during the optimization. Only codes located inside the bounded interval of average degrees are passed to the time consuming simulator. Using this principle, we can classify the codes in two groups.

Such a classificator needs to be prepared before doployment. Thereby, the algorithm working with the classificator is separated into two steps sequentially performed:

- learning,
- deployment of the classificator

Table F.1 shows a comparison of times required for the formal description and the time consuming simulation. The time required for the calculation of mean degrees of check nodes is the fastest. The ratio between times required for the comprehensive simulation and the calculation of mean degrees is depicted in Table F.2. As can be seen in figures below, this feature may be used to distinguish if a code is potentionally efficient or inefficient.

- codes potentially good, which will be passed to the simulator,
- codes potentially bad, which will be dropped.

Table F.1: Comparison of times required for formal description and comprehensive simulation.

required time	**mean cycle length**	**mean node degree**	E_b/N_0 **at** $\mathbf{10^{-3}}$
average	3.0126 ms	0.00015418 ms	11828 ms
minimum	1.137 ms	0.000135 ms	2407 ms
maximum	5.047 ms	0.000193 ms	17709 ms
median	3.112 ms	0.000155 ms	12545 ms

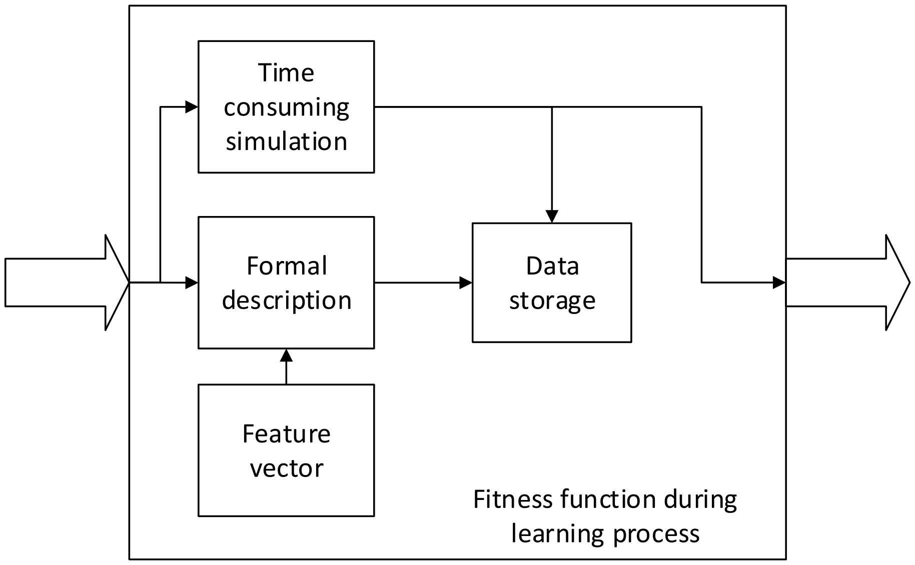

Time consuming simulation
Formal description
Data storage
Feature vector
Fitness function during learning process

Figure F-1: The algorithm of learning a feature based classificator.

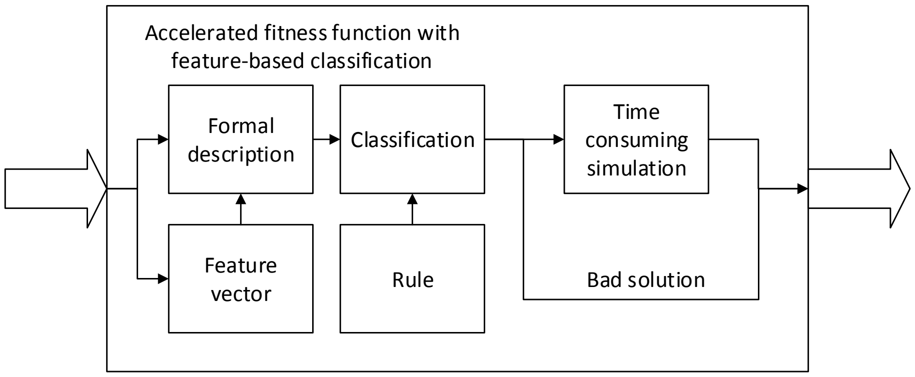
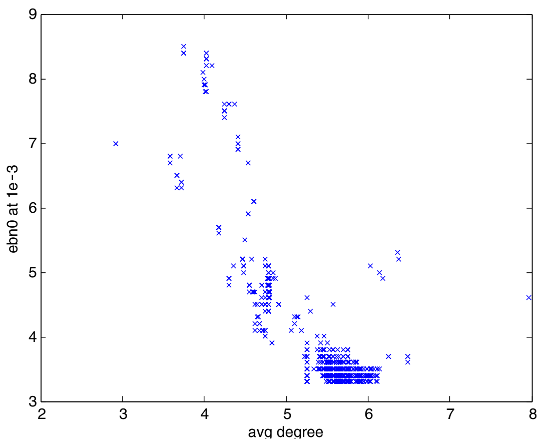

Accelerated fitness function with feature-based classification
Formal description
Classification
Time consuming simulation
Feature vector
Rule
Bad solution

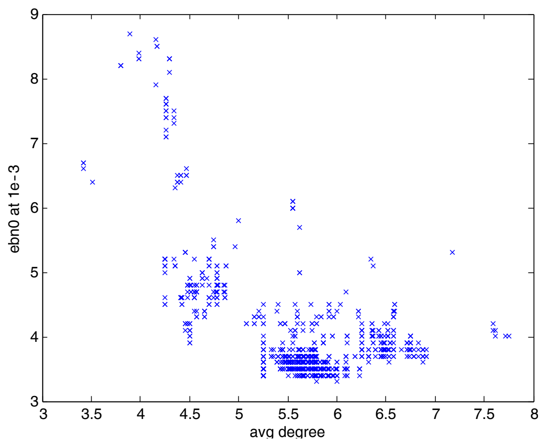
Figure F-2: The algorithm for accelerating the evaluation function.

Table F.2: Comparison of times required for formal description and comprehensive simulation.

Ratio	**Simulation / Formal description**
mean	7.67E+07
minimum	1.78E+07
maximum	9.18E+07
median	8.09E+07

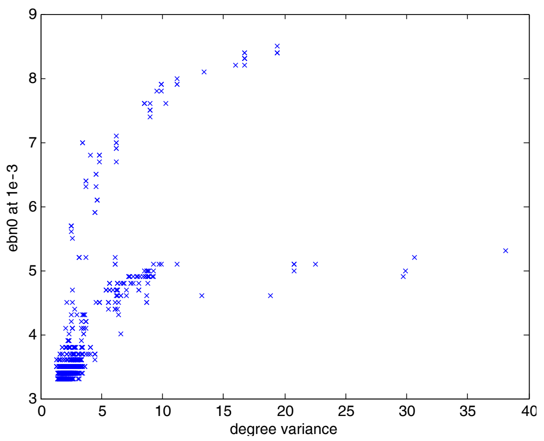


(a) Experiment A.

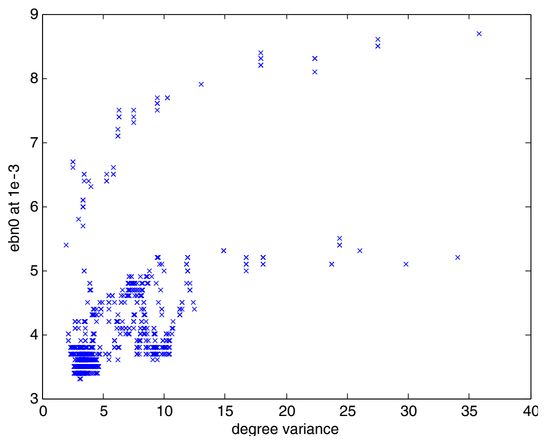


(b) Experiment B.

Figure F-3: Required Signal to Noise Ratio against average variable node degrees.

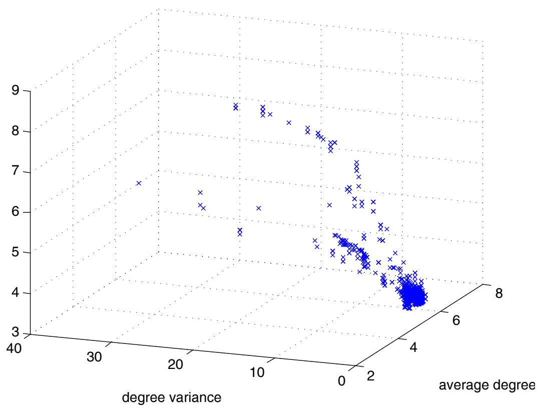


(a) Experiment A.

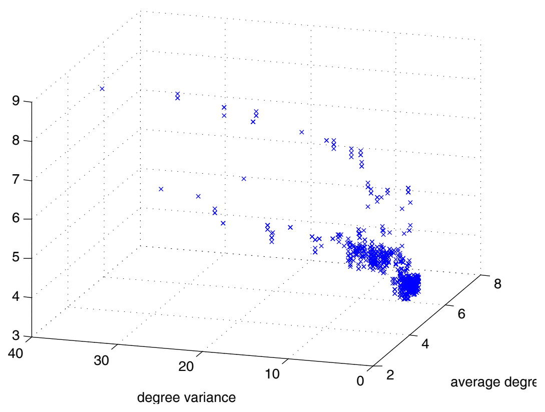


(b) Experiment B.

Figure F-4: Required Signal to Noise Ratio against average variable node degrees.



(a) Experiment A.



(b) Experiment B.

Figure F-5: Required Signal to Noise Ratio against features.

References

[1] C. Anton, L. Ionescu, I. Tutanescu, A. Mazare, G. Serban, *Error detection and correction using LDPC in parallel Hopfield networks*, Electrical and Electronics Engineering (ISEEE), 2013 4th International Symposium on , vol., no., pp.1,4, 11-13 Oct. 2013

[2] E. Arikan, *Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels*, in IEEE Transactions on Information Theory, vol. 55, no. 7, pp. 3051-3073, July 2009.

[3] J. P. Arun, M. Mishra and S. V. Subramaniam, *Parallel implementation of MOPSO on GPU using OpenCL and CUDA*, 2011 18th International Conference on High Performance Computing, Bangalore, 2011, pp. 1-10.

[4] C. A. Aslam, Y. L. Guan and K. Cai, *Improving the Belief-Propagation Convergence of Irregular LDPC Codes Using Column-Weight Based Scheduling,* in IEEE Communications Letters, vol. 19, no. 8, pp. 1283-1286, Aug. 2015.

[5] M. Beermann, E. Monr, L. Schmalen and P. Vary, *High speed decoding of non-binary irregular LDPC codes using GPUs*, SiPS 2013 Proceedings, Taipei City, 2013, pp. 36-41.

[6] G. Bernab, G. D. Guerrero and J. Fernndez, *CUDA and OpenCL implementations of 3D Fast Wavelet Transform*, Circuits and Systems (LASCAS), 2012 IEEE Third Latin American Symposium on, Playa del Carmen, 2012, pp. 1-4.

[7] R. Bose, D. Ray-Chaudhuri, *On a Class of Error-Correcting Binary Codes.* Inf. and Control, vol. 3, pp. 68-79, 1960.

[8] N. Bonello, S. Chen, L. Hanzo, *Low-Density Parity-Check Codes and Their Rateless Relatives*, Communications Surveys & Tutorials, IEEE , vol.13, no.1, pp.3,26, First Quarter 2011.

[9] G. J. Byers, F. Takawira, *Fourier transform decoding of non-binary LDPC codes.* In Proceedings Southern African Telecommunication Networks and Applications Conference, 2004.

[10] J. Broulim, V. Georgiev, *LDPC error correction code utilization*, Telecommunications Forum (TELFOR), 2012 20th , vol., no., pp.1048,1051, 20-22 Nov. 2012.

[11] J. Broulim, V. Georgiev, J. Moldaschl and L. Palocko, *LDPC code optimization based on Tanner graph mutations*, 2013 21st Telecommunications Forum Telfor (TELFOR), Belgrade, 2013, pp. 389-392.

[12] J. Broulim, P. Broulim, J. Moldaschl, V. Georgiev and R. Salom, *Fully parallel FPGA decoder for irregular LDPC codes*, 2015 23rd Telecommunications Forum Telfor (TELFOR), Belgrade, 2015, pp. 309-312.

[13] Sae-Young Chung, G. D. Forney Jr., T. J. Richardson, R. Urbanke, *On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit*, Communications Letters, IEEE , vol.5, no.2, pp.58,60, Feb 2001.

[14] M. C. Davey, D. MacKay, *Low-density parity check codes over GF(q)*, Communications Letters, IEEE , vol.2, no.6, pp.165,167, June 1998.

[15] M. C. Davey, D. MacKay, *Low density parity check codes over GF(q)*, Information Theory Workshop, 1998 , vol., no., pp.70,71, 22-26 Jun 1998.

[16] M. C. Davey, *Error-correction using Low-Density Parity-Check Codes.* Ph.D. dissertation, University of Cambridge, 1999.

[17] P. Elias, *Coding for Noisy Channels.* IRE Conv. Rept. Pt. 4, pp. 37-47, 1955.

[18] J. Fang, A. L. Varbanescu and H. Sips, *A Comprehensive Performance Comparison of CUDA and OpenCL*, 2011 International Conference on Parallel Processing, Taipei City, 2011, pp. 216-225.

[19] Y. Fang, G. Bi, Y. L. Guan and F. C. M. Lau, *A Survey on Protograph LDPC Codes and Their Applications*, in IEEE Communications Surveys & Tutorials, vol. 17, no. 4, pp. 1989-2016, Fourthquarter 2015.

[20] G. Falcao, V. Silva, L. Sousa and J. Andrade, *Portable LDPC Decoding on Multicores Using OpenCL [Applications Corner]*, IEEE Signal Processing Magazine, vol. 29, no. 4, pp. 81-109, July 2012.

[21] M.P.C. Fossorier, *Quasicyclic low-density parity-check codes from circulant permutation matrices*, Information Theory, IEEE Transactions on , vol.50, no.8, pp.1788,1793, Aug. 2004.

[22] R. Hamming, *Error detecting and error correcting codes.* Bell Syst. Technical Journal. vol. 29, pp. 41-56, 1950.

[23] C. Heinemann, S. S. Chaduvu, A. Byerly and A. Uskov, *OpenCL and CUDA software implementations of encryption/decryption algorithms for IPsec VPNs*, 2016 IEEE International Conference on Electro Information Technology (EIT), Grand Forks, ND, 2016, pp. 0765-0770.

[24] R. G. Gallager, *Low Density Parity Check Codes*, Transactions of the IRE Professional Group on Information Theory, Vol. IT-8, January 1962, pp. 2l-28.

[25] M. J. E. Golay. *Notes on Digital Coding*, Proc. IRE 37: 657, 1949.

[26] Joo-Yul Park and Ki-Seok Chung, *Parallel LDPC decoding using CUDA and OpenMP*. Park and Chung EURASIP Journal on Wireless Communications and Networking, 2011.

[27] A. R. Karami, M. A. Attari, H. Tavakoli, H., *Multi Layer Perceptron Neural Networks Decoder for LDPC Codes*," Wireless Communications, Networking and Mobile Computing, 2009. WiCom '09. 5th International Conference on , vol., no., pp.1,4, 24-26 Sept. 2009.

[28] N. F. Kiyani, J. H. Weber, *Analysis of random regular LDPC codes on Rayleigh fading channels*. Proceedings of the twenty-seventh symposium on information theory in the Benelux. WIC, 2006. p. 69-76.

[29] L. Kocarev, Z. Tasev Z., A. Vardy, *Improving turbo codes by control of transient chaos in turbo-decoding algorithms*, Electronics Letters, vol. 38, pp. 1184-1186, 2002.

[30] Y. Kou, S. Lin and M. P. C. Fossorier, *Low-density parity-check codes based on finite geometries: a rediscovery and new results*, in IEEE Transactions on Information Theory, vol. 47, no. 7, pp. 2711-2736, Nov 2001.

[31] A.D. Kumar, A. Dukkipati, *A two stage selective averaging LDPC decoding,* in Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on , vol., no., pp.2866-2870, 1-6 July 2012.

[32] F. Leduc-Primeau, S. Hemati, S. Mannor and W. J. Gross, *Dithered Belief Propagation Decoding*, in IEEE Transactions on Communications, vol. 60, no. 8, pp. 2042-2047, August 2012.

[33] Wang Lin, Xiao Juan and Guanrong Chen, *Density evolution method and threshold decision for irregular LDPC codes*, 2004 International Conference on Communications, Circuits and Systems (IEEE Cat. No.04EX914), Chengdu, 2004, pp. 25-28 Vol.1.

[34] G. Liva, M. Chiani, *Protograph LDPC Codes Design Based on EXIT Analysis*, Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE , vol., no., pp.3250,3254, 26-30 Nov. 2007

[35] M. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. Spielman, *Improved low-density parity-check codes using irregular graphs and belief propagation*, in Proc. 1998 Int. Symp. Information Theory, p. 117.

[36] D. J. MacKay. *Encyclopedia of sparse graph codes.* [Online]. Available: http://www.inference.phy.cam.ac.uk/mackay/codes/data.html

[37] D. J. C. MacKay, R. M. Neal, *Good Codes based on Very Sparse Matrices. Cryptography and Coding.* 5th IMA Conference, number 1025 in Lecture Notes in Computer Science. 1995.

[38] D.J.C. MacKay, S.T. Wilson, M.C. Davey, *Comparison of constructions of irregular Gallager codes*, Communications, IEEE Transactions on , vol.47, no.10, pp.1449,1454, Oct 1999.

[39] D.J.C. MacKay, *Good error-correcting codes based on very sparse matrices*, Information Theory, IEEE Transactions on , vol.45, no.2, pp.399,431, Mar 1999.

[40] N. Mobini, *New Iterative Decoding Algorithms for Low-Density Parity-Check (LDPC) Codes.* Otawa-Carleton Institute for Electrical and Computer Engineering. 2011.

[41] T. K. Moon, *Error Correction Coding : Mathematical methods and algorithms.* John Wiley & Sons, Inc., 2005, 756 s. ISBN 0-471-64800-0.

[42] P. Moreira, A. Marchioro, K. Kloukinas, *The GBT, a Proposed Architecture for Multi-Gb/s Data Transmission in High Energy Physics.* CERN

[43] J. M. F. Moura, Jin Lu and Haotian Zhang, *Structured low-density parity-check codes*, in IEEE Signal Processing Magazine, vol. 21, no. 1, pp. 42-55, Jan. 2004.

[44] D. Muller, *Application of Boolean Switching Algebra to Switching Circuit Design.* IEEE Trans. on Computers, vol. 3, pp. 6-12, Sept. 1954.

[45] N. Obata, Jian Yung-Yih, K. Kasai, H. D. Pfister, *Spatially-coupled multi-edge type LDPC codes with bounded degrees that achieve capacity on the BEC under BP decoding*, Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on , vol., no., pp.2433,2437, 7-12 July 2013

[46] C.E. Shannon, *A mathematical theory of communication.* Bell Sys. Technical Journal. vol. 27, pp. 379423, 623656, July, October, 1948.

[47] T. Ott, R. Stoop, *The neurodynamics of belief propagation on binary markov random fields.* Advances in Neural Information Processing Systems 19, Cambridge, MA: MIT Press. pp. 10571064. 2007.

[48] Prange E, *Cyclic Error-Correcting Codes in Two Symbols.* Air Force Cambridge Research Center, Cambridge, MA, Tech. Rep. TN- 57-103, Sept. 1957.

[49] I. Reed. *A class of Multiple-Error Correcting Codes and a Decoding Scheme.* IEEE Trans. Information Theory, vol. 4, pp. 38-49, Sept 1954.

[50] T. J. Richardson, M.A. Shokrollahi, R. L. Urbanke, *Design of capacity-approaching irregular low-density parity-check codes*, Information Theory, IEEE Transactions on , vol.47, no.2, pp.619,637, Feb 2001.

[51] T. J. Richardson and R. L. Urbanke, *The capacity of low-density parity-check codes under message-passing decoding*, in IEEE Transactions on Information Theory, vol. 47, no. 2, pp. 599-618, Feb 2001.

[52] T. J. Richardson, R. L. Urbanke, *Multi-Edge LDPC Codes.* DRAFT, 2004.

[53] I. Reed, G. Solomon. *Polynomial Codes over Certain Finite Field.* J. Soc. Indust. Appl. Math. vol. 8 pp. 300-304, 1960.

[54] D. A. Spielman, *Finding good LDPC codes*, 36th Annual Allerton Conference on Communication, Control, and Computing, 1998.

[55] R. Tanese. *Distributed Genetic Algorithms.* Proceedings of the Third International Conference on Genetic Algorithms, pages 434-439. Morgan Kaufmann, 1989.

[56] R. M. Tanner, *A recursive approach to low complexity codes*, Information Theory, IEEE Transactions on , vol.27, no.5, pp.533,547, Sep 1981.

[57] S. S. Tehrani, S. Mannor, W. J. Gross, *Fully Parallel Stochastic LDPC Decoders*, Signal Processing, IEEE Transactions on , vol.56, no.11, pp.5692,5703, Nov. 2008.

[58] Tao Tian, C. Jones, J. D. Villasenor and R. D. Wesel, *Construction of irregular LDPC codes with low error floors*, Communications, 2003. ICC '03. IEEE International Conference on, 2003, pp. 3125-3129 vol.5.

[59] J. Thorpe, *Low-Density Parity-Check (LDPC) Codes Constructed from Protographs*, IPN Progress Report 42-154, 2003.

[60] J. Thorpe, K. Andrews, S. Dolinar, *Methodologies for designing LDPC codes using protographs and circulants*, Information Theory, 2004. ISIT 2004. Proceedings. International Symposium on , vol., no., pp.238,, 27 June-2 July 2004.

[61] Tao Tian, C. R. Jones, J. D. Villasenor, R. D. Wesel, *Selective avoidance of cycles in irregular LDPC code construction*, Communications, IEEE Transactions on , vol.52, no.8, pp.1242,1247, Aug. 2004.

[62] C. Berrou, A. Glavieux, P. Thitimajshima, *Near Shannon limit error-correcting coding and decoding: Turbo-codes (1).* Communications, 1993. ICC '93 Geneva. Technical Program, Conference Record, IEEE International Conference on , vol.2, no., pp.1064,1070 vol.2, 23-26 May 1993.

[63] A. G. D. Uchoa, C. Healy, R. C. de Lamare, R. D. Souza, *LDPC codes based on Progressive Edge Growth techniques for block fading channels*, Wireless Communication Systems (ISWCS), 2011 8th International Symposium on , vol., no., pp.392,396, 6-9 Nov. 2011.

[64] B. Vasic, *Combinatorial constructions of low-density parity check codes for iterative decoding*, Information Theory, 2002. Proceedings. 2002 IEEE International Symposium on , vol., no., pp.312,, 2002

[65] B. Vasic and O. Milenkovic, *Combinatorial constructions of low-density parity-check codes for iterative decoding*, in IEEE Transactions on Information Theory, vol. 50, no. 6, pp. 1156-1176, June 2004.

[66] G. Wang, M. Wu, B. Yin and J. R. Cavallaro, *High throughput low latency LDPC decoding on GPU for SDR systems*, Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE, Austin, TX, 2013, pp. 1258-1261.

[67] Z. Wang, Z. Cui, *Low-Complexity High-Speed Decoder Design for Quasi-Cyclic LDPC Codes*, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , vol.15, no.1, pp.104,114, Jan. 2007.

[68] S. Wang, S. Cheng and Q. Wu, *A parallel decoding algorithm of LDPC codes using CUDA*, 2008 42nd Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, 2008, pp. 171-175.

[69] X. Wen et al., *A high throughput LDPC decoder using a mid-range GPU*, 2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Florence, 2014, pp. 7515-7519.

[70] N. Wiberg, Codes and Decoding on General Graphs. PhD thesis, Dept. of Electrical Engineering, Lionkoing, Sweden, 1996. Lionkoing studies in Science and Technologz. Dissertation No. 440.

[71] Darrell Whitley, Soraya Rana, Robert B. Heckendorn. *Island model genetic algorithms and linearly separable problems.* Evolutionary Computing, pp. 109-125, 1997.

[72] Z. Wu, K. Su, L. Guo, *A modified Min Sum decoding algorithm based on LMMSE for LDPC codes*, in AEU - International Journal of Electronics and Communications, vol. 68, i. 10, October 2014.

[73] Xiao-Yu Hu, E. Eleftheriou, D.M. Arnold, *Progressive edge-growth Tanner graphs*, Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE , vol.2, no., pp.995,1001 vol.2, 2001

[74] Xiao-Yu Hu, E. Eleftheriou, D.M. Arnold, *Regular and irregular progressive edge-growth tanner graphs*, Information Theory, IEEE Transactions on , vol.51, no.1, pp.386,398, Jan. 2005.

[75] M.R. Yazdani, S. Hemati, A.H. Banihashemi, *Improving belief propagation on graphs with cycles,* in Communications Letters, IEEE , vol.8, no.1, pp.57-59, Jan. 2004, doi: 10.1109/LCOMM.2003.822499

[76] J. S. Yedidia, W. T. Freeman, Y. Weiss, *Generalized belief propagation*, in Advances Neural Inf. Proc. Syst. (NIPS), pp. 689695, MIT Press, 2001.

[77] X. Wu, Y. Song, M. Jiang and C. Zhao, *Adaptive-Normalized/Offset Min-Sum Algorithm*, in IEEE Communications Letters, vol. 14, no. 7, pp. 667-669, July 2010.

[78] Z. Wu, K. Su, L. Guo, *A modified Min Sum decoding algorithm based on LMMSE for LDPC codes*, in AEU - International Journal of Electronics and Communications, vol. 68, i. 10, October 2014.

[79] Yue Zhao, Xu Chen, Chiu-Wing Sham, Wai M. Tam, and Francis C.M. Lau *Efficient Decoding of QC-LDPC Codes Using GPUs*, Algorithms and Architectures for Parallel Processing. 2011

[80] X. Zhang; F. Cai, *Efficient Partial-Parallel Decoder Architecture for Quasi-Cyclic Nonbinary LDPC Codes*, Circuits and Systems I: Regular Papers, IEEE Transactions on , vol.58, no.2, pp.402,414, Feb. 2011.

[81] Haotian Zhang and J. M. F. Moura, *Large-girth LDPC codes based on graphical models*, 2003 4th IEEE Workshop on Signal Processing Advances in Wireless Communications - SPAWC 2003 (IEEE Cat. No.03EX689), 2003, pp. 100-104.

[82] Y. Zhao and F. C. M. Lau, *Implementation of Decoders for LDPC Block Codes and LDPC Convolutional Codes Based on GPUs*, IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 3, pp. 663-672, March 2014.

[83] Yue Zhao, Xu Chen, Chiu-Wing Sham, Wai M. Tam, and Francis C.M. Lau, *Efficient Decoding of QC-LDPC Codes Using GPUs.* 11th International Conference, ICA3PP, Melbourne, Australia, October 24-26, 2011, Proceedings, Part I

[84] Xia Zheng, F. C. M. Lau, C. K. Tse, *Constructing Short-Length Irregular LDPC Codes with Low Error Floor*, Communications, IEEE Transactions on , vol.58, no.10, pp.2823,2834, October 2010.

[85] V. V. Zyablov, M. S. Pinsker, *Estimation of the error-correction complexity for Gallager low-density codes*, Problems of Inform. Transmission, vol. 11, no. 1, pp. 2326, Jan.March 1975

[86] ETSI EN 302 755 European standard (DVB-T2) [Online]. Available: http://www.etsi.org/. Accessed: [25-Nov-2016].

[87] IEEE 802 LAN/MAN Standards Committee materials, [Online]. Available: http://www.ieee802.org/. Accessed: [25-Nov-2016].

[88] ETSI EN 302 307 European standard (DVB-S2), [Online]. Available: http://www.etsi.org/. Accessed: [25-Nov-2016].

[89] NVIDIA Corporation, *Cuda Runtime API*, Reference manual, 2015 [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_Runtime_API.pdf. Accessed: [28-Jan-2018].

[90] Khronos OpenCL Working Group, *The OpenCL Specification*, 2011 [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf. Accessed: [28-Jan-2018].

[91] Ryan Smith, *NVIDIA Launches Tesla K80, GK210 GPU*. AnandTech (November 17, 2014), Available: http://www.anandtech.com/tag/gpus. Accessed: [02-Jun-2016]

[92] *Whitepaper of NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110/210*. Available: http://www.nvidia.com/object/gpu-architecture.html. Accessed: [02-Jun-2016]

[93] CCSDS, *Short Block Length LDPC Codes for TC Synchronization and Channel Coding*. Washington, DC, USA, Apr. 2015, CCSDS 231.1-O-1.

[94] IEEE Standard for Local and Metropolitan Area Networks Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems Amendment 2: Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands and Corrigendum 1,in *IEEE Std 802.16e-2005 and IEEE Std 802.16-2004/Cor 1-2005 (Amendment and Corrigendum to IEEE Std 802.16-2004) , vol., no., pp.0_1-822, 2006.*

List of author's publications

[1] J. Broulim, V. Georgiev, *LDPC error correction code utilization*, 20th Telecommunications Forum TELFOR 2012. Blehrad: IEEE, 2012. s. 1048-1051. ISBN: 978-1-4673-2984-2.

[2] M. Holik, V. Kraus, V. Georgiev, J. Broulim, *The interface for the pixelated particle detector with the capability of the spectroscopy function and the coincidence measurement operation*, 21st Telecommunications Forum (TELFOR) Proceedings of Papers. IEEE, 2013. s. 557-560. ISBN: 978-1-4799-1419-7.

[3] J. Broulim, V. Georgiev, J. Moldaschl, L. Palocko, *LDPC code optimization based on Tanner graph mutations*, 21st Telecommunications Forum (TELFOR) Proceedings of Papers. Blehrad: IEEE, 2013. s. 389-392. ISBN: 978-1-4799-1419-7.

[4] L. Palocko, J. Broulim, J. Moldaschl, *Decoder with the Dynamic CMOS Matrix*, 21st Telecommunications Forum TELFOR 2013. Blehrad: IEEE, 2013. s. 612-615. ISBN: 978-1-4799-1419-7.

[5] P. Broulim, J. Broulim, V. Georgiev, J. Moldaschl, *Very high resolution time measurement in FPGA*, 22nd Telecommunications Forum TELFOR 2014 Proceedings of Papers. Blehrad: IEEE, 2014. s. 745-748. ISBN: 978-1-4799-6190-0.

[6] J. Moldaschl, J. Broulim, L. Palocko, *Principle of Power Factor Corrector with Critical Conduction Mode*, 2014 International Conference on Applied Electronics. Pilsen: University of West Bohemia, 2014. s. 217-220. ISBN: 978-80-261-0276-2 , ISSN: 1803-7232.

[7] J. Moldaschl, J. Broulim, L. Palocko, *Boost Power Factor Topology with Average Current Control*, 2014 International Conference on Applied Electronics. Pilsen: University of West Bohemia, 2014. s. 213-216. ISBN: 978-80-261-0276-2 , ISSN: 1803-7232.

[8] R. Salom, J. Broulim, *LDPC (512,480) genetic design as alternative to CRC in implementation of AODV routing protocol stack.*, Proceedings of Papers : 2015 23rd Telecommunications Forum (TELFOR 2015). Piscataway: IEEE, 2015. s. 643-645. ISBN: 978-1-5090-0055-5.

[9] J. Broulim, P. Broulim, J. Moldaschl, V. Georgiev, R. Salom, *Fully parallel FPGA decoder for irregular LDPC codes*, Proceedings of Papers : 2015 23rd Telecommunications Forum (TELFOR 2015). Piscataway: IEEE, 2015. s. 309-312. ISBN: 978-1-5090-0055-5.

[10] P. Broulim, J. Bartovsky, J. Broulim, P. Burian, V. Georgiev, M. Holik, V. Kraus, A. Krutina, J. Moldaschl, V. Pavlicek, S. Pospisil, J. Vlasek, *Compact device for detecting single event effects in semiconductor components*, Proceedings of Papers : 2015 23rd Telecommunications Forum (TELFOR 2015). Piscataway: IEEE, 2015. s. 639-642. ISBN: 978-1-5090-0055-5.

[11] J. Zich, J. Broulim, *Wireless unit for environmental monitoring*, 2016 24th Telecommunications Forum (TELFOR 2016) : Proceedings of Papers. Piscataway: IEEE, 2016. s. 643-646. ISBN: 978-1-5090-4086-5.

[12] J. Broulim, V. Georgiev, N. Boulgouris, *On fast exhaustive search of the minimum distance of linear block codes*, 8th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT 2016) : proceedings. Piscataway: IEEE, 2016. s. 342-345. ISBN: 978-1-4673-8818-4 , ISSN: 2157-0221.

[13] J. Broulim, S. Davarzani, V. Georgiev, J. Zich, *Genetic optimization of a short block length LDPC code accelerated by distributed algorithms*, 2016 24th Telecommunications Forum (TELFOR 2016) : Proceedings of Papers. Piscataway: IEEE, 2016. s. 250-253. ISBN: 978-1-5090-4086-5.

[14] J. Moldaschl, J. Broulim, *An impact of the boost diode selection on the overall efficiency of active power factor correctors*, International Conference on Applied Electronics (AE 2016) : proceedings. Piscataway: IEEE, 2016. s. 187-190. ISBN: 978-80-261-0601-2 , ISSN: 1803-7232.

[15] J. Broulim, J. Moldaschl, V. Georgiev, *A programmable voltage source with high speed current feedback protection*, In International Conference on Applied Electronics (AE 2016) : proceedings. Piscataway: IEEE, 2016. s. 31-34. ISBN: 978-80-261-0601-2 , ISSN: 1803-7232.

[16] J. Zich, J. Broulim, M. Holik, *Smart single-phase battery storage system*, 25th Telecommunications Forum (TELFOR) : Proceedings of Papers. Piscataway: IEEE, 2017. s. 589-592. ISBN: 978-1-5386-3073-0.

[17] J. Broulim, V. Georgiev, M. Holik, J. Zich, *Improved belief propagation based on the estimation backtracking*, 25th Telecommunications Forum (TELFOR) : Proceedings of Papers. Piscataway: IEEE, 2017. s. 262-265. ISBN: 978-1-5386-3073-0.

[18] M. Holik, J. Broulim, V. Georgiev, Y. Mora Sierra, *Enhanced timepix3 chipboard for operation in vacuum and back-side-pulse spectroscopy*, 25th Telecommunications Forum (TELFOR) : Proceedings of Papers. Piscataway: IEEE, 2017. s. 593-596. ISBN: 978-1-5386-3073-0.

[19] In preparation: *OpenCL/CUDA algorithms for parallel decoding of any irregular LDPC code using GPU*

[20] In preparation: *Mutational LDPC decoding using Information Entropy*

[21] In preparation: *A synchronization and data acquisition system for silicon detectors*

[22] In preparation: *j-Pix : A multiplatform acquisition package for Timepix 3*