YILDIRIM BEYAZIT UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

PERFORMANCE IMPROVEMENT METHODS FOR LAYERED DECODING OF LDPC CODES

by

Murat SEVER

July, 2015

ANKARA

PERFORMANCE IMPROVEMENT METHODS FOR LAYERED DECODING OF LDPC CODES

A Thesis Submitted to the

Graduate School of Natural and Applied Sciences of Yıldırım Beyazıt University

In Partial Fulfillment of the Requirements for the Master of Science in Electrical and Electronics Engineering, Department of Electrical and Electronics Engineering

by

Murat SEVER

July, 2015

ANKARA

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **"Performance Improvement Methods for Layered Decoding of LDPC Codes"** completed by **Murat SEVER** under supervision of **Asst. Prof. Dr. Enver ÇAVUŞ** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Enver ÇAVUŞ

____

Supervisor

Asst. Prof. Dr. Mehmet ÜNLÜ	Asst. Prof. Dr. M. Efe ÖZBEK
______________________________	______________________________
(Jury Member)	(Jury Member)

____

Prof. Dr. Fatih V. ÇELEBİ

Director

Graduate School of Natural and Applied Sciences

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor Dr. Enver Çavuş for his guidance in my study. From the very first lecture I took on channel coding to my last touches on this thesis, he was always helpful and accessible. He introduced me to the subject of LDPC and academic research. I feel myself lucky to have him as my advisor.

I also wish to thank to my family especially to my wife, my kids and to my parents. And special thanks to all my friends who help and support me. They never left me alone at this three-year journey.

Murat SEVER

July, 2015

CONTENTS

THESIS EXAMINATION RESULT FORM.. ii
ACKNOWLEDGEMENTS iii
CONTENTS iv
LIST OF ACRONYMS vi
LIST OF TABLES ix
LIST OF FIGURES x
LIST OF SYMBOLS xii
ABSTRACT xiii
ÖZET xiv
1 INTRODUCTION 1
1.1 Literature Review of Performance Improvement Methods 2
1.2 Objectives 4
1.3 Contributions 5
2 LDPC CODES 6
2.1 Linear Block Codes 7
2.2 Overview of LDPC Codes 10
2.2.1 Construction of LDPC Codes 11
2.2.2 DVB-S2 LDPC Codes 12
2.2.3 802.11n LDPC Codes 13
2.2.4 802.15.3c LDPC Codes 15
3 DECODING OF LDPC CODES 16
3.1 Sum-Product Algorithm 17
3.2 Min-Sum Algorithm 21
3.3 Normalized Min-Sum Algorithm 21
3.4 Dual-Scaling MSA (DS-MSA) 22
3.5 Self-Corrected Min-Sum Algorithm 22
3.6 Layered Decoding of LDPC Codes 23
3.7 Parallel Decoding of LDPC Codes 25
4 PROPOSED IMPROVEMENT METHODS 26
4.1 Fixed Order Layered Decoding of LDPC Codes (LOCK) 26
4.2 Dynamic Order Layered Decoding of LDPC Codes (ORD) 27
4.3 Satisfied Weight Order Layered Decoding of LDPC Codes (ORDE) ... 28

5 PERFORMANCE RESULTS OF PROPOSED METHODS 31
5.1 Simulation of Proposed Methods on PC 31
5.1.1 Simulation Results for 802.11n LDPC Codes 38
5.1.2 Results for 802.15.3c LDPC Codes 45
5.1.3 Results for DVB-S2 Codes 50
6 PARALLEL DECODING OF LDPC CODES ON REAL DSP HARDWARE 52
6.1 OpenMP Framework 52
6.2 Embedded DSP Platform 54
6.3 Implementation Details for Parallel Decoding of LDPC Codes on DSP 56
6.4 Results for Parallel Decoding 60
7 CONCLUSIONS AND FUTURE WORK 62
8 RESOURCES 64
9 ÖZGEÇMİŞ 66

LIST OF ACRONYMS

ARQ	Automatic Repeat Request
AWGN	Additive White Gaussian Noise
BER	Bit Error Rate
BN	Bit Node
BPSK	Binary Phase Shift Keying
CCS	Code Composer Studio
CN	Check Node
dB	Decibel
DSP	Digital Signal Processor
DVB	Digital Video Broadcasting
ECC	Error Correction Codes
EDMA	Enhanced Direct Memory Access
EVM	Evaluation Module
FEC	Forward Error Correction
GHz	Giga Hertz
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
IPC	Inter Processor Communication

IRA	Irregular Repeat Accumulate
JTAG	Joint Test Action Group
kB	Kilo Bytes
LAN	Local Area Network
LDPC	Low-Density Parity-Check
LINQ	Language Integrated Query
LLR	Log-Likelihood Ratio
MCN	Multicore Navigator
MCSDK	Multicore Software Development Kit
MPI	Message Passing Interface
MIT	Massachusetts Institute of Technology
MSA	Min-Sum Algorithm
MSMC	Multicore Shared Memory Controller
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
OS	Operating System
PLINQ	Parallel Language Integrated Query
RAM	Random Access Memory
RTOS	Real Time Operating System
RTSC	Real-Time Software Components

SoC	System on Chip
SNR	Signal to Noise Ratio
SPA	Sum-Product Algorithm
SRIO	Serial Rapid Input Output
TDMP	Turbo Decoding Message Passing
TI	Texas Instruments
TPMP	Two-Phase Message Passing
UIA	Unified Instrumentation Architecture
WER	Word Error Rate
WLAN	Wireless Local Area Network
WPAN	Wireless Personal Area Network

LIST OF TABLES

Table 2.1 DVB-S2 LDPC codes 12
Table 2.2 802.11n LDPC codes 14
Table 2.3 802.15.3c LDPC codes........ 15
Table 5.1 Alist file contents 32
Table 5.2 Array to keep column positions of check nodes 33
Table 5.3 Array to keep row positions of variable nodes 33
Table 5.4 Identification of edges........ 34
Table 5.5 Indexes for check nodes........ 34
Table 5.6 Indexes for variable nodes 35
Table 5.7 Selected LDPC codes from 802.11n standard 39
Table 5.8 Iteration savings obtained by SBS algorithm with (648, 540) LDPC code42
Table 5.9 Selected LDPC codes from 802.15.3c standard........ 45
Table 6.1 Execution times for an iteration and speedup versus number of cores...... 61

LIST OF FIGURES

Figure 2.1 Typical communication channel 6
Figure 2.2 Communication channel model 8
Figure 2.3 Systematic codeword 9
Figure 2.4 Structured LDPC code parity-check matrix example 12
Figure 2.5 Base matrix for 802.11n, 648-bit, rate: 5/6 15
Figure 3.1 Parity-check matrix showing nodes 16
Figure 3.2 Tanner graph representation 17
Figure 3.3 Bit-to-check message updating 19
Figure 3.4 Check-to-bit message updating 20
Figure 3.5 Reliability information update 20
Figure 4.1 Fixed order layered decoding example 27
Figure 4.2 Dynamic order layered decoding example 28
Figure 4.3 Example regular LDPC code 29
Figure 4.4 "Total unsatisfied check node" calculation 30
Figure 5.1 Encoder class diagram 35
Figure 5.2 Decoder class diagram 36
Figure 5.3 Decoding parallelism 37
Figure 5.4 Flow diagram of LDPC decoding process 38
Figure 5.5 Performance results for WLAN, short-length, low-rate code with 50 max iterations 40
Figure 5.6 Performance results for WLAN, short-length, low-rate code with 100 max iterations 40
Figure 5.7 Performance results for WLAN, short-length, high-rate code with 50 max iterations 41
Figure 5.8 Performance results for WLAN, short-length, high-rate code with 100 max iterations 41
Figure 5.9 Performance results for WLAN, long-length, low-rate code with 50 max iterations 43
Figure 5.10 Performance results for WLAN, long-length, low-rate code with 100 max iterations 43
Figure 5.11 Performance results for WLAN, long-length, high-rate code with 50 max iterations 44
Figure 5.12 Performance results for WLAN, long-length, high-rate code with 100 max iterations 45
Figure 5.13 Performance results for WPAN, short-length, low-rate code with 50 max iterations 46
Figure 5.14 Performance results for WPAN, short-length, low-rate code with 100 max iterations 47
Figure 5.15 Performance results for WPAN, short-length, high-rate code with 50 max iterations 48
Figure 5.16 Performance results for WPAN, short-length, high-rate code with 100 max iterations 48
Figure 5.17 Performance results for WPAN, long-length, high-rate code with 50 max iterations 49

Figure 5.18 Performance results for WPAN, long-length, high-rate code with 100 max iterations 50
Figure 5.19 Performance results for DVB-S2, normal-frame, high-rate code with 50 max iterations 51
Figure 5.20 Performance results for DVB-S2, normal-frame, high-rate code with 100 max iterations 51
Figure 6.1 OpenMP Fork/Join model 53
Figure 6.2 TMDSEVM6678 Target module 54
Figure 6.3 Reference DSP architecture 55
Figure 6.4 Application development with CCS 56
Figure 6.5 TI's OpenMP solution stack 56
Figure 6.6 Check node processing using horizontal striping of H matrix 58
Figure 6.7 Bit node processing using vertical striping of H matrix 59
Figure 6.8 Speedup using OpenMP 61

LIST OF SYMBOLS

λ_i	Channel a priori information
σ^2	Noise variance
BN	Bit node
C	Codeword, consisting of n bits
CN	Check node
G	Generator matrix
H	Parity-check matrix
H_{bm}	Base-matrix
H^T	Transpose of parity-check matrix
I_k	kxk Identity matrix
k	Information bit-length
n	Codeword bit-length
n_b	Number of layers
Q_{ij}	Extrinsic information passed from bit node i to check node j
R_c	Code Rate of a block code
R_{ji}	Extrinsic information from check node j to bit node i
w_c	Column weight
w_r	Row weight
Z	Submatrix block-size

PERFORMANCE IMPROVEMENT METHODS FOR LAYERED DECODING OF LDPC CODES

ABSTRACT

Low-density parity-check (LDPC) codes are a special type of linear block codes. Although they were originally invented by R. Gallager in 1960, they have captured the attention of scientific community since its rediscovery in 1996. Due to their excellent correction capability, they have been adopted in many communications and storage systems for forward error correction.

Gallager defined an iterative two-phase decoding algorithm referred to as "Belief Propagation" (BP). Much study has been conducted to improve the performance of LDPC decoding. Later, another scheduling, "Turbo Decoding Message Passing" (TDMP) is introduced as an alternative to his standard message passing algorithm. TDMP, also called layered decoding, has the advantage of converging faster than the standard BP because it uses more reliable information to update next set of values. Using layered decoding, it is possible to reduce the number of iterations by 50% without any performance degradation. In this thesis, we present several methods in order to improve layered decoding performance of LDPC codes. Methods proposed have been applied to several LDPC codes with different length and rate. According to our results, the biggest performance improvements are achieved when they are applied to small-length, high-rate codes. In addition to proposed methods, a simulation acceleration platform using OpenMP is also described where parallel decoding is implemented on a real multicore hardware platform, obtaining more than 6x speedup compared to single-core version.

Keywords: Low-density parity-check, error correction codes, belief propagation, layered decoding, performance improvement, DSP, parallelization, OpenMP

LDPC KODLARININ KATMANLI MİMARİDE ÇÖZÜMÜNDE PERFORMANS ARTTIRICI YÖNTEMLER

ÖZET

Düşük-yoğunluklu eşlik-denetim (LDPC) kodları doğrusal blok kodları arasındadır. İlk olarak 1960 yılında R. Gallager tarafından keşfedilmelerine rağmen uzun yıllar boyunca unutulmuş, 1996 yılında yapılan bir çalışma ile bilim dünyasının yeniden ilgisini çekmişlerdir. Yüksek hata başarımına sahip olmaları nedeni ile günümüzde modern iletişim ve depolama sistemlerinde hata düzeltici kodlar olarak yaygın bir biçimde kullanılmaktadırlar.

LDPC kod çözücülerin çalışma prensibi Gallager tarafından ortaya atılan kanı yayılımına (Belief Propagation) dayanmaktadır. LDPC kodlarının çözümünde performans iyileştirme maksadıyla çeşitli yöntemler bulunmuş ve pratikte de uygulanmaktadır. Bu yöntemlerden biri de kodların turbo çözümleme ile çözülmesidir. Önerilen yöntem ile kod çözümlemede daha güvenilir mesajların düğümler arası geçirilmesi sayesinde hızlı bir şekilde yakınsama sağlanmaktadır. Katmanlı mimaride çözümleme olarak da adlandırılan yöntem ile normal modda gereken yineleme sayısının yarısında aynı başarım seviyesine ulaşmak mümkündür. Tez çalışmamızda katmanlı mimaride LDPC çözümlemede performans arttırıcı yeni yöntemler sunulmuştur. Yeni yöntemler birçok değişik uzunlukta ve hızda LDPC kodları üzerinde denenmiştir. Yapılan benzetim çalışmaları sonucunda, önerilen yöntemler en iyi performans artırımını kısa ve hızlı LDPC kodları üzerinde sağladığı gözlenmiştir. Önerilen yöntemlerin seçilen LDPC kodları üzerinde denenmesine ilaveten, OpenMP kullanarak LDPC çözümleme işlemi paralelleştirilmiş, 8-çekirdekli gerçek DSP donanımı üzerinde yapılan testlerde tek çekirdeğe göre 6 kattan fazla hızlanma elde edildiği raporlanmıştır.

Anahtar sözcükler: Düşük-yoğunluklu eşlik-denetim, hata düzeltici kodlar, kanı yayılımı, katmanlı mimaride çözümleme, performans artırımı, DSP, paralelleştirme, OpenMP

1 INTRODUCTION

Low-Density Parity-Check (LDPC) codes have received widespread attention because of their excellent error correcting performance. They provide performance very close to the Shannon limit, which is why they are also called capacity-approaching codes. Invented by Gallager at MIT in 1960 [1], LDPC codes were forgotten for more than 30 years due to their computational complexity. After Berrou, et al published their research on Turbo codes in 1993 [2] researchers focused on alternative iterative decoding schemes and in 1996, LDPC codes were rediscovered by Mackay and Neal [3]. Today, LDPC codes are adopted in many modern communications systems for forward error correction such as DVB-S2 (satellite transmission of digital television), IEEE 802.11n (wireless LAN), IEEE 802.16 (WiMAX, broadband wireless internet), IEEE 802.15.3c (wireless personal area network), and IEEE 802.3an (10Gbit Ethernet).

As LDPC codes are adopted in many communication standards, improving performance or convergence rate of LDPC decoder with minimal additional complexity is a prime interest. One attractive performance improvement method is to modify the updating order (schedule) of passing the messages between bit and check nodes. In standard decoding algorithm, a flooding scheduling is employed, where all check nodes and subsequently all bit nodes are simultaneously updated in every iteration. Alternatively, non-simultaneous scheduling allows the updated messages to be used earlier, and hence achieves the same Bit Error Rate (BER) performance as flooding decoder with much less decoding iterations or improved performance for a given number of decoding iteration. In this thesis, we study performance improvement techniques for both simultaneous and non-simultaneous scheduling algorithms.

The rest of this thesis is organized as follows: In this section, after presenting a literature review for performance improvement methods of LDPC decoding algorithms, objectives and contribution of this thesis are provided. Section 2 gives an overview of linear block codes and provides basic knowledge on LDPC codes and parity-check matrix structures. Examples of LDPC codes which found their way into

industry standards are also given in Section 2. LDPC decoding algorithms are described in Section 3. Section 4 contains our proposed scheduling methods based on layered decoding. The simulation results, performance comparisons of proposed methods to existing ones will be presented in Section 5. Following the presentation of WER performance results, a simulation acceleration platform using OpenMP is described and obtained simulation speed gains are reported in Section 6. Lastly, conclusions will be presented in the last section, Section 7.

1.1 Literature Review of Performance Improvement Methods

As LDPC codes have found their way into many recent industry standards, there exists a high desire for low-complexity, and high throughput LDPC decoders. In order to attain this aim, many methods have been explored. Some of these methods try to simplify the decoding algorithm, while the others aim to achieve a high throughput by speeding up the convergence rate of the decoder. Among the decoding algorithms, Sum-product algorithm (SPA) performs the best but it has a very high computational complexity. There are several approaches to lower computational complexity of SPA while preserving the decoding performance as much as possible. The most popular approximation is the min-sum algorithm (MSA). MSA offers lower hardware complexity at the cost of performance degradation. There always exists tradeoff between complexity and performance. To minimize performance degradation of MSA, different methods are proposed. Normalized-MSA, Offset-MSA are the most popular ones. These methods use a scaling or an offset factor to minimize overestimation in common.

There are other methods proposed that use more than one scaling factor. [4] proposed two-way normalization to improve MSA performance. Since 0.25 and 0.50 are chosen as scaling factors, they are simple to implement simply as shift registers in hardware. Therefore, the technique can be applied to systems with minimal hardware complexity. Simulation results show better performance is achieved when applied to medium and short length codes.

[5] deploys two scaling factors to correct overestimation made by MSA. Proposed method provides extended waterfall region and has better performance. It is computationally-efficient algorithm which is well-suited for storage systems.

An Adaptive normalized MSA is proposed by [6] in which normalization factor is adaptively determined at each decoding iteration. The proposed method achieves better performance while keeping the complexity the complexity nearly the same.

[7] describes lazy scheduling approach in which only a subset of nodes get updated in an iteration resulting in reduced iteration cost. Partial update of nodes combined with serial scheduling reduces the decoding complexity significantly compared to standard belief propogation scheme while maintaining the same performance.

The standard message passing schedule is an iterative process which consists of two phases. It is also attractive due to its inherently highly parallelizable structure. However, one major drawback with decoding of LDPC codes using standard message passing schedule is its slow convergence. It is therefore essential to increase convergence speed of LDPC codes. A high-throughput memory-efficient decoder architecture for LDPC codes based on a novel turbo decoding was introduced in [8]. The main idea is to better utilize reliability messages by updating it more than once within an iteration. In [9] simulations show that as much as two times faster decoding speed can be achieved by applying turbo-scheduling algorithm. In [10] three optimized message passing, namely row message passing, column message passing and row-column message passing scheduling algorithms are introduced. In row message passing, bit-node and check-node probabilities are updated row by row. And it is shown by simulations that row message passing converges about two times faster than the standard message passing.

Another serial scheduling strategy based on variable node updating schedule is introduced as shuffled BP in [11]. The proposed scheme has about the same complexity as standard scheme but converge faster. When the maximum number of iterations has to be small, the proposed method offers better performance.

Original layered decoding schemes cannot be directly applied to non-layered codes whose column weights are greater than one within layers. [12] proposed parallel-layered belief propagation (PLBP) algorithm that can be used in "non-layered" codes. It achieves better error performance with 50% less iterations.

[13] proposes decoding based on turbo-decoding message-passing strategy. Various computation kernels, Normalized-MSA, Offset-MSA and Self-corrected MSA are compared in terms of implementation area, energy consumption and error-correcting performance. Studied kernels are applied to codes from IEEE 802.11n. Results show that Self-corrected MSA shows a better error correcting performance and energy efficiency per iteration.

While sequential scheduling improve the convergence speed, finding the best sequence of message updates is of prime interest. In [14] the informed dynamic scheduling is presented to find the next message to be updated. It uses the concept of residual belief propagation. Differences between the values of the messages before and after an update, called residuals, are used to dynamically update the schedule. Residuals are used as an ordering metric to update messages.

[15] combined the features of lazy schedule and node-wise residual belief propagation together to obtain efficient dynamic schedule for layered BP decoding of LDPC codes.

[16] proposes informed dynamic scheduling strategy which utilizes instability of variable node the residual of the variable-to-check message to locate the message to be updated first. Results show that it outperform other algorithms at the cost of increased complexity.

1.2 Objectives

Performance of error correcting codes has significant importance on overall performance of communications system. In order to improve the performance, there have been much research in the scientific community. Most of the study focuses on finding efficient ways to reduce complexity at the cost of small degradation in performance or to obtain better bit error rate by introducing new methods requiring more resources. There always exists some tradeoff between performance and complexity. Decoding of LDPC codes is an iterative process and includes a message-passing strategy, the most popular one being flooding schedule. Later several different message-passing schemes were introduced. Layered decoding algorithms have gained attention because of their faster convergence. In layered decoding,

finding the right ordering of message updates plays an important role in performance of the decoder. Ordering has an effect on decoding performance. In this thesis, we consider defining set of rules for ordering message updates in order to maximize decoding performance. We propose new techniques to improve performance of layered decoding of LDPC codes. We also present a simulation acceleration platform using OpenMP where parallel decoding is implemented on a real multicore hardware platform.

1.3 Contributions

The main contribution of this work is to show that further performance improvements can be achieved with minor changes in decoding algorithm. We have recalled message-passing algorithms for decoding of LDPC codes and proposed new methods based on turbo decoding message-passing algorithm. Because new methods are based on layered decoding, they have faster convergence speed than standard message-passing algorithms. Our layered decoding schemes can be implemented with minimal changes which are negligible. Only modification lies in the process order of layers. Therefore, they can be easily applied for layered decoding of LDPC codes without any increase in computational complexity. We have applied proposed methods to the codes in 802.11n, 802.15.3c and DVB-S2. According to our results, we manage to obtain better performance results compared to standard message-passing schemes.

One attractive solution to long simulation times of LDPC codes is to implement inherently parallel decoding algorithms using multicore platforms. In this thesis, we also present the first OpenMP parallel implementation of LDPC decoding algorithm on a multicore DSP architecture and report its performance. Parallelized Normalized Min-Sum decoding algorithm is implemented on 8-core Texas Instruments (TI) DSP using OpenMP framework. Performance results are obtained by Unified Instrumentation Architecture (UIA). Our results show that the parallelized decoding on 8-core TI DSP achieves more than 6x speedup compared to single-core version.

2 LDPC CODES

Error-correction plays an important role in digital communications and storage systems. As demand for data increases, higher data throughput is needed. More and more data must be delivered reliably over unreliable channels. Nowadays, another challenge in data communication is mobility. All of these reasons require more powerful channel coding methods to be utilized by communications systems. In Figure 2.1 a typical communication system is illustrated.

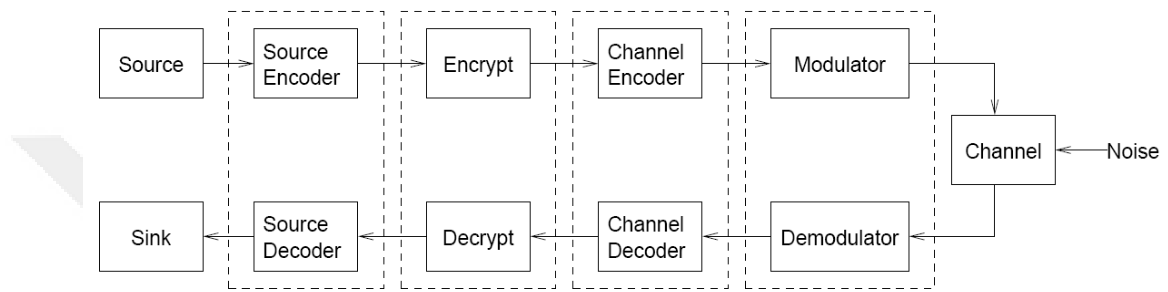

Figure 2.1 Typical communication channel

In this system, the source produces data to be transmitted. Information bits are taken from a source, which could be an audio, video or other data. Before channel encoding step, depending on the application, source encoding and encryption can be applied to information bits.

Channel coding aims to protect data against noise that exists during transmission or read/write to the storage media. People hardly realize errors that occur during transmission thanks to error correction techniques applied underneath. Without a suitable error control method, reliable data transmission would be impossible. Therefore, the need for a high throughput error correction control is a must for many applications areas.

Error correction is made possible by adding extra parity bits (redundancy) into data, which consists of information bits. Later, these redundant bits are used to detect or correct errors. Communication systems whether wireless or not deploy various error-correction schemes such as convolutional codes and block codes. Convolutional codes have memory and current data has dependency on the previous data sent. Block codes encode and decode data on a block-by-block basis, and there

is no data dependency between data blocks. They have higher error correcting performance and lower complexity. In this thesis, we will focus on Low-Density Parity-Check (LDPC) codes, which is a type of linear block codes.

The rest of this section is organized as follows. After a brief overview of Linear Block Codes is presented in Section 2.1, Section 2.2 gives an introduction on LDPC codes. Construction methods of LDPC parity check matrices are discussed in Section 2.2.1. In this thesis, different LDPC codes from three different standards are studied. LDPC codes defined in DVB-S2, IEEE 802.11n, and IEEE 802.15.3c standards are discussed in Section 2.2.2, Section 2.2.3 and Section 2.2.4, respectively.

2.1 Linear Block Codes

Block codes are used to detect and correct data errors introduced during transmission. While data passes through transmission medium, noise is introduced to data being sent. This leads to error in data when it is received. A typical communication channel is illustrated in Figure 2.2. An encoder adds redundancy to the message being sent. At the receiver, a decoder corrects errors introduced during transmission using redundancy in the received data. It is important to handle errors present in data received for systems to work properly. Therefore, every practical communication system incorporates error detection and correction mechanism. In a two-way communication systems, error detection may be sufficient because in case of error data can be requested for retransmission. In presence of error, receiver notifies the transmitter of the existence of errors. This strategy is called automatic repeat request (ARQ). But in one-way communication systems, there is no chance of requesting data for retransmission. System is required to recover data in error without retransmission. In this situation, error correction is needed at the receiver side in addition to error detection in order to improve communication quality. This strategy is known as forward error correction (FEC). All errors can be corrected without retransmission if error rate is under FEC capacity.

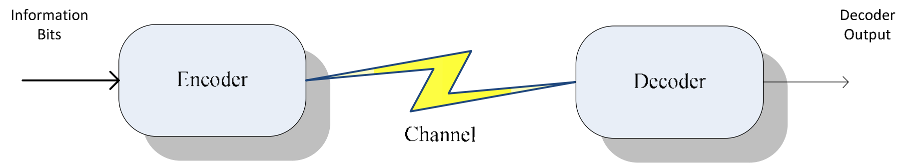

Figure 2.2 Communication channel model

A block code is represented by (*n, k*) where k corresponds to number of information bits and n corresponds to number of codeword bits. So, (*n, k*) code adds *n-k* parity bits into k-length original data block to obtain n-length codeword. These parity bits are used to correct data errors present in the received data. A block code is considered a linear block code if addition of any valid two codewords results in another valid codeword. Code rate is defined as ratio of length of information block (*k*) to codeword length (*n*). As we add more parity bits, code rate decreases. The higher code rate, the less transmission bandwidth needed. Code rate is defined by the following formula:

$$R_c = \frac{k}{n} \tag{2.1}$$

For uncoded systems, code rate is 1, i.e., no redundancy is added. The redundancy can be assumed as overhead because it consumes transmission resources like channel bandwidth or transmission power. Coding performance is inversely proportional to coding rate.

Construction of a codeword from information bits is done by encoder. If codeword is constructed such that parity bits appear after information bits, that code is called systematic code. An example of a systematic codeword is given in Figure 2.3. Basically, parity bits are computed then appended to the end of information block.

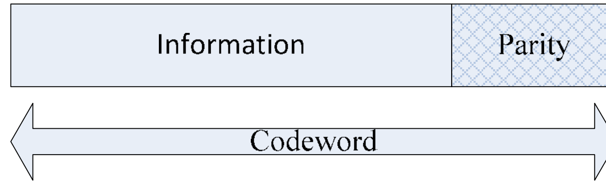

Figure 2.3 Systematic codeword

Encoding can be simply described as multiplication of two matrices, message matrix K and special matrix G. So, given information block row-vector *K*, codeword is constructed by

$$C = K \cdot G \tag{2.2}$$

where G is called generator matrix. For systematic codes, *G* is in the form of $[I_k|P]$. I_k is identity matrix with k rows and columns. Below is one example of generator matrix for $(7,4)$ code.

$$\boldsymbol{G} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} \end{bmatrix} \tag{2.3}$$

At the decoder side, another matrix called the parity check matrix is used to decode codeword. For systematic codes, it can be constructed from generator matrix *G* using:

$$\boldsymbol{H} = [\boldsymbol{P^T} |\ \boldsymbol{I_{n-k}}] \tag{2.4}$$

So, corresponding *H* matrix for systematic code represented by generator matrix in (2.3) is

$$\boldsymbol{H} = \begin{bmatrix} \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix} \tag{2.5}$$

Generator matrix and parity check matrix satisfy the following equation:

$$G \cdot H^T = 0 \tag{2.6}$$

H matrix is a *(n-k)xn* matrix. It contains (*n-k*) constraints. Each row corresponds to parity check equation. A valid codeword must satisfy:

$$C \cdot H^T = 0 \tag{2.7}$$

H performs (*n-k*) separate parity check operations on a received codeword. For example, parity check operations implied by *H* matrix given in (2.5) are:

$$c_0 \oplus c_2 \oplus c_4 = 0 \tag{2.8}$$

$$c_1 \oplus c_2 \oplus c_3 \oplus c_5 = 0 \tag{2.9}$$

$$c_0 \oplus c_1 \oplus c_6 = 0 \tag{2.10}$$

- The first parity equation checks bits 0, 2, and 4
- The second parity equation checks bits 1, 2, 3, and 5
- The third and last parity equation checks bits 0, 1, and 6

2.2 Overview of LDPC Codes

LDPC codes are a subclass of linear block codes, that have gained reputation as the most powerful channel coding technique. They are also called capacity-approaching codes because they can approach Shannon capacity limit which states maximum transmission rates over noisy channels with power and bandwidth constraints. In [17], an irregular LDPC code performs within 0.0045dB of the Shannon limit, making it the best performing code known so far. That is why LDPC codes have attracted much attention and have been employed by many recent communication standards. In addition, unlike other iterative codes, LDPC codes have a lower computational complexity with a suitable architecture for parallel decoding [18].

LDPC codes are linear block codes defined by a sparse parity-check matrix. Number of 1's in parity check matrix is low in density. Number of 1's in a row is called the row weight w_r and number of 1's in a column is called the column weight

w_c. In LDPC codes $w_r \ll n$ and $w_c \ll n-k$. An LDPC code said to be regular if w_r is constant for all rows and w_c is constant for all columns.

2.2.1 Construction of LDPC Codes

There are many methods for constructing LDPC codes. They can be classified into two categories: algebraic constructions and random constructions [19]. A simple algebraic code construction scheme is based on cyclically shifted identity matrices. A shifted identity matrix is obtained from identity matrix by shifting each row by a specific number. The parity-check matrix is in the form of

$$H = \begin{bmatrix} H_{11} & H_{12} & \cdots & H_{1n} \\ H_{21} & H_{22} & \cdots & H_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ H_{n-k,1} & H_{n-k,2} & \cdots & H_{n-k,n} \end{bmatrix} \quad \textbf{(2.11)}$$

where H_{ij} $(1 \le i \le n-k \; and \; 1 \le j \le n\,)$ is a submatrix with cyclically shifted identity matrix. Below is an example of 1-time right-shifted 4x4 identity matrix:

$$I_1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \textbf{(2.12)}$$

In random construction, LDPC codes are randomly chosen from code ensembles specified by left degree distribution, right degree distribution, and a block-length. There are also several random code construction schemes based on random graph lifts.

There also exist "structured codes" which are proposed to facilitate the hardware design of the decoder. Their parity-check matrices are constructed according to specific pattern in order to simplify implementation issues. Figure 2.4 shows parity-check matrix for a structured code. It consists of many non-overlapping rows that allow layered decoding. Each block or submatrix is either all-zero (null) or a cyclically shifted identity matrix. Location of submatrices with specific shift values are determined during code design. In Figure 2.4 there are $n_b$horizontal layers, where $n_b$is defined in (2.13).

$$n_b = \frac{(n-k)}{Z} \quad (2.13)$$

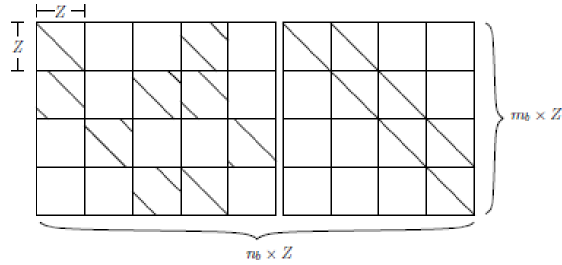

Figure 2.4 Structured LDPC code parity-check matrix example

In the following sections we will closely look at some structured codes defined by well-known industry standards.

2.2.2 DVB-S2 LDPC Codes

DVB-S2 is the first standard that adopts LDPC as FEC code. Two codeword lengths (short frame consisting of 16200 bits, long frame consisting of 64800 bits) are defined within DVB-S2 standard. Code properties of LDPC codes defined in DVB-S2 standard are given in the Table 2.1.

Table 2.1 DVB-S2 LDPC codes

Code Rate	Codeword Length	Information Length
1/5	16200	3240
1/3	16200	5400
2/5	16200	6480
4/9	16200	7200
3/5	16200	9720
2/3	16200	10800
11/15	16200	12150

7/9	16200	12960
37/45	16200	13320
8/9	16200	14400
1/4	64800	16200
1/3	64800	21600
2/5	64800	25920
1/2	64800	32400
3/5	64800	38880
2/3	64800	43200
3/4	64800	48600
4/5	64800	51840
5/6	64800	54000
8/9	64800	57600
9/10	64800	58320

Codes used in DVB-S2 are based on Irregular Repeat Accumulate (IRA) codes. Periodicity within the parity-check matrix reduces storage requirements. Codes are systematic codes, so parity bits are appended to the end of information bits. The LDPC encoder needs to create n-k parity bits from k information bits to construct a valid codeword of length n. Encoding procedure is defined within the standard of DVB-S2.

2.2.3 802.11n LDPC Codes

IEEE 802.11n defines 12 different LDPC codes with different code lengths and code rates. The supported code rates, codeword lengths, information length, and

submatrix sizes are given in Table 2.2. These codes are reused within 802.11ac standard which aims to increase the throughput of the 802.11n standard further [20].

Table 2.2 802.11n LDPC codes

Code Rate	Codeword Length	Information Length	Submatrix Size
1/2	648	324	27
2/3	648	432	27
3/4	648	486	27
5/6	648	540	27
1/2	1296	648	54
2/3	1296	864	54
3/4	1296	972	54
5/6	1296	1080	54
1/2	1944	972	81
2/3	1944	1296	81
3/4	1944	1458	81
5/6	1944	1620	81

Parity-check matrices of 802.11n LDPC codes are constructed by expanding submatrices from the base matrix. Figure 2.5 shows an example of such parity-check base matrix with a rate of 5/6. In this case, the block size is $Z=27$. The parity-check matrix is composed of all-zero submatrix or identity submatrix with different cyclic shifts. The positive numbers stand for the right cyclic shift value of the identity submatrix, and the “–1” denotes null submatrix.

$$H_{bm} = \begin{bmatrix} 17 & 13 & 8 & 21 & 9 & 3 & 18 & 12 & 10 & 0 & 4 & 15 & 19 & 2 & 5 & 10 & 26 & 19 & 13 & 13 & 1 & 0 & -1 & -1 \\ 3 & 12 & 11 & 14 & 11 & 25 & 5 & 18 & 0 & 9 & 2 & 26 & 26 & 10 & 24 & 7 & 14 & 20 & 4 & 2 & -1 & 0 & 0 & -1 \\ 22 & 16 & 4 & 3 & 10 & 21 & 12 & 5 & 21 & 14 & 19 & 5 & -1 & 8 & 5 & 18 & 11 & 5 & 5 & 15 & 0 & -1 & 0 & 0 \\ 7 & 7 & 14 & 14 & 4 & 16 & 16 & 24 & 24 & 10 & 1 & 7 & 15 & 6 & 10 & 26 & 8 & 18 & 21 & 14 & 1 & -1 & -1 & 0 \end{bmatrix}$$

Figure 2.5 Base matrix for 802.11n, 648-bit, rate: 5/6

2.2.4 802.15.3c LDPC Codes

802.15.3c is a standard by IEEE for high-rate wireless personal area networks (WPAN). The standard provides three physical layer modes for data rates exceeding 1Gb/s. It is the first standard in millimeter wave (mmWave) band since operating frequency of 60GHz (57-64 GHz) corresponds to 5 mm in wavelength.

One of the two FEC schemes specified in 802.15.3c standard is LDPC codes. There are five LDPC codes defined in the standard. Four of the LDPC codes have block length of 672, and the longer one has a block length of 1440. The LDPC encoder is systematic.

Similar to 802.11n codes, the parity-check matrices in 802.15.3c can be partitioned to submatrices. These submatrices are either null (all-zero) submatrices or cyclic-permutations of the identity matrix. Cyclic-permutation matrix is obtained by cyclically shifting the columns to the left by a specific amount. Properties of the codes defined in 802.15.3c can be found in Table 2.3.

Table 2.3 802.15.3c LDPC codes

Code Rate	Codeword Length	Information Length	Submatrix Size
1/2	672	336	21
3/4	672	504	21
5/8	672	420	21
7/8	672	588	21
14/15	1440	1344	96

3 DECODING OF LDPC CODES

In this section, a review of decoding algorithms of LDPC codes are given. Section 3.1 introduces Sum-Product decoding algorithm. Min-Sum algorithm, which is approximation to Sum-Product algorithm is given in Section 3.2. Afterwards, other decoding algorithms to improve performance of Min-Sum follow. Namely, Normalized Min-Sum algorithm, Dual-Scaling Min-Sum algorithm, Self-Corrected Min-Sum algorithm are given in Section 3.3, Section 3.4, and Section 3.5, respectively. Layered decoding of LDPC codes is discussed at the end of this section, Section 3.6.

LDPC decoding uses a message passing algorithm between two types of nodes, called check (parity) nodes and bit (variable) nodes. Every row in H matrix corresponds to a check node; every column corresponds to a bit node. Considering H matrix given in Figure 3.1, it has 7 bit nodes and 3 check nodes.

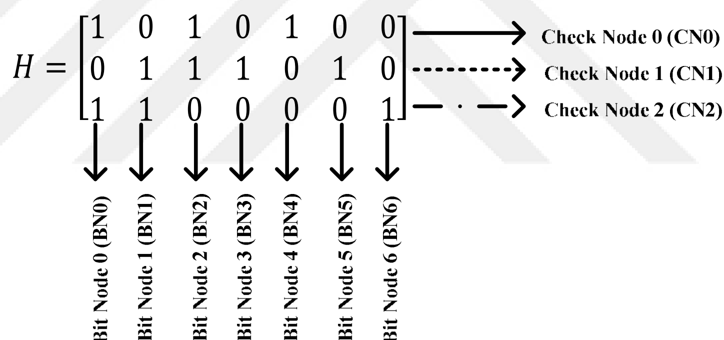

Figure 3.1 Parity-check matrix showing nodes

A visual representation of the parity check matrix assists in the understanding of decoding algorithms. Tanner graphs invented by R. Michael Tanner in 1981 [16], provide a graphical representation of linear block codes. A Tanner Graph is a bipartite graph with two types of nodes, check and bit nodes. The bit nodes are usually drawn as circles, and check nodes are usually drawn as squares. Edge between a bit node and check node indicates that corresponding bit node is involved in the parity check constraint. So there are edges as many as number of 1s in parity-check matrix. The parity check matrix illustrated in Figure 3.1 can be visualized using a Tanner Graph in Figure 3.2.

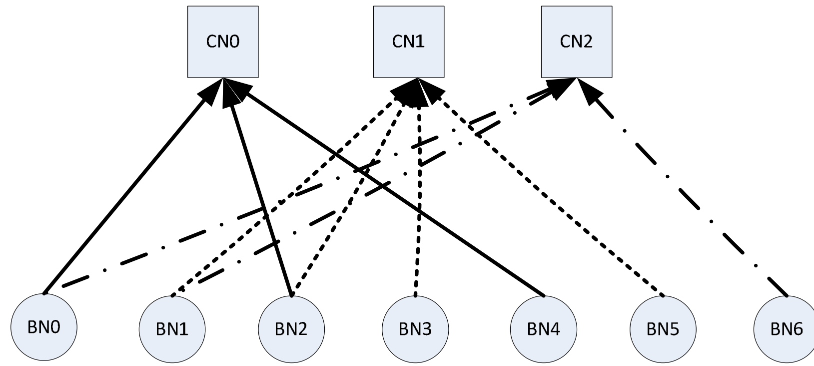

Figure 3.2 Tanner graph representation

Decoding of LDPC codes are done by iteratively passing messages between two sets of nodes represented in the Tanner graph. There are mainly two ways to accomplish decoding of LDPC codes. Hard-decision decoding uses bit-flip algorithm, whereas soft-decoding algorithm uses sum-product algorithm, which accepts soft values and uses these values to pass messages between nodes. Sum-Product algorithm, belief-propagation algorithm, or message-passing algorithm are the different names given to the same iterative decoding algorithm.

Difference between hard-decision and soft-decision lies in the values passed between nodes. The former propagates messages 0 or 1 while the latter propagates messages as soft values or probabilities of being 0 or 1.

There has been much research on decoding of LDPC codes to decrease computational complexity, to reduce energy consumption of the decoder, to lower number of iterations and obtain better error performance.

3.1 Sum-Product Algorithm

Sum-product algorithm (SPA) uses the concept of message passing between bit nodes and check nodes in an iterative manner. Soft values are propagated between nodes and as messages are passed successively, reliability information improves with the iteration count. The exchange of the soft probabilities is called message passing or belief propagation. Messages are passed along the edges between bit nodes and check nodes in the Tanner graph.

The SPA operates with probabilities. However working with probabilities has several drawbacks. SPA contains intensive numerical computation and involves

many multiplications. Multiplication is harder to implement compared to addition. So instead of using probability, log likelihood ratio (LLR) values can be used. This way, multiplication turns into sum operation, divisions simply become subtractions and decoding complexity is reduced by this way.

Suppose that an (n,k) LDPC code is defined, and Binary Phase Shift Keying (BPSK) modulation is used. BPSK maps a valid n-length codeword $c = \{c_0, c_1, \dots, c_{n-1}\}$ into a sequence $x = \{x_0, x_1, \dots, x_{n-1}\}$ according to $x_i = 1 - 2 \cdot c_i$ where 0 <= i <n. With BPSK, bit 0 is mapped to symbol +1 and bit 1 is mapped to symbol -1. After modulation, x is transmitted over Additive White Gaussian Noise (AWGN) channel. At the receiver, y= $\{y_0, y_1, \dots, y_{n-1}\}$ is observed. $y_i = x_i + n_i$ where n_i represents AWGN with zero mean and variance σ^2.

Iterative decoding refines log-likelihood ratio (LLR) of bits received which is defined as in (3.1)

$$LLR_i = \log\left(\frac{P(c_i = 0|y_i)}{P(c_i = 1|y_i)}\right) \quad \textbf{(3.1)}$$

The sign of the LLR value indicates bit value being 0 or 1, and its magnitude is linked to reliability of the belief.

Two-phase message-passing (TPMP) is the most common schedule for decoding of LDPC codes. It is also called flooding schedule. In flooding schedule the computations are executed in two phases. Messages from variable to check nodes in one phase; message updates from check nodes to their corresponding variable nodes happen in the second phase. Hard decision is made at the end of each iteration. SPA is carried out as follows:

Initialization Step-1:

In the initialization step extrinsic information from check node j to bit node i is set to 0, and LLR value for the bit node is set to channel's a priori value as in (3.3).

$$R_{ji} = 0 \tag{3.2}$$

$$\lambda_i = LLR_i = \frac{2y_i}{\sigma^2} \tag{3.3}$$

where R_{ji} is the extrinsic information from check node j to bit node i and λ_i is chanel a priori information.

Bit (Variable) Node Process Step-2:

In this step, bit node i combines messages from corresponding check nodes $CN_0,...CN_j,...CN_{w-1}$ with w being column weight for bit node i. Bit node computes the updated message to be sent to its neighbouring check nodes as follows as in (3.4)

$$Q_{ij} = LLR_i - R_{ji} \tag{3.4}$$

where Q_{ij} is the extrinsic information passed from bit node i to check node j. This is depicted in Figure 3.3.

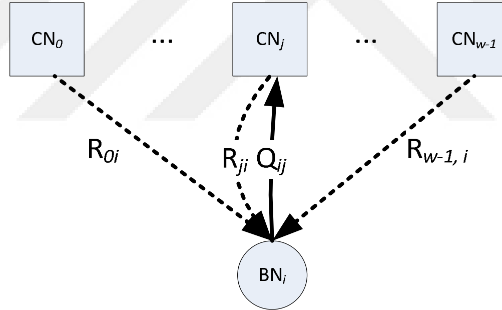

Figure 3.3 Bit-to-check message updating

Check Node Process Step-3:

In this step, check node j (CN_j) receives messages from neighbouring variable nodes, and propagates back the updated messages R_{ji} as:

$$R_{ji} = \left(\prod_{i' \in N(j) \setminus i} sgn(Q_{i'j}) \right) \Phi \left(\sum_{i' \in N(j) \setminus i} \Phi(|Q_{i'j}|) \right) \tag{3.5}$$

where $\Phi(x)$ is defined as

$$\boldsymbol{\Phi(x) = -\,log(tanh(x/2)) = log\left(\frac{e^x + 1}{e^x - 1}\right)} \tag{3.6}$$

and *N(j)* is the set of neighbour bit nodes connected to check node j. $i' \in N(j)\backslash i$ means all bit nodes connected to check node *j* except bit node *i*.

Figure 3.4 illustrates check-to-bit message updating.

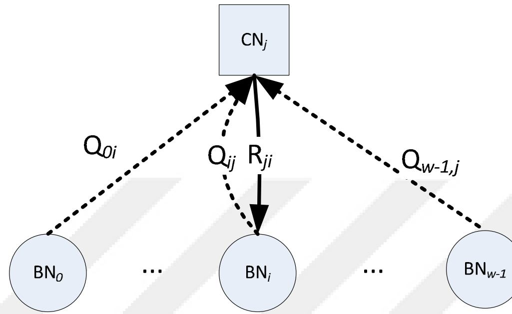

Figure 3.4 Check-to-bit message updating

Posterior reliability value, also referred to as soft output of the received bit is updated as in (3.7).

$$\boldsymbol{LLR_i = \lambda_i + \sum_{j \in N(i)} R_{ji}} \tag{3.7}$$

This process is illustrated in Figure 3.5.

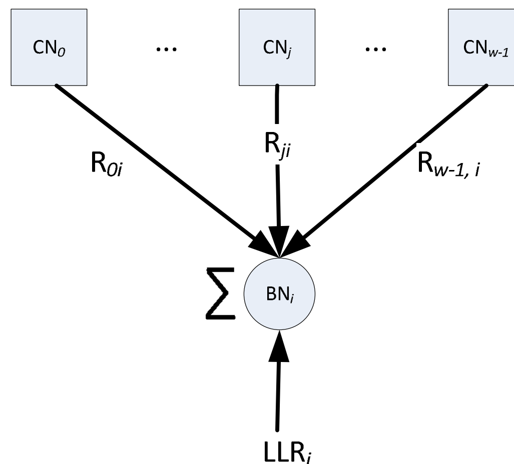

Figure 3.5 Reliability information update

Hard Decision Step-4:

At the end of each iteration hard decoding decision is made according to (3.8) and (3.9).

$$c_i = \begin{cases} 1 & if\ LLR_i < 0 \\ 0 & otherwise \end{cases} \tag{3.8}$$

$$c \cdot H^T = \begin{cases} all\ parity - checks\ satisfied\ if\ 0 \\ else\ not\ a\ valid\ codeword \end{cases} \tag{3.9}$$

Iterative procedure continues until decoded codeword satisfies all check node equations ($c \cdot H^T = 0\ condition$) or predefined number of iteration count is reached otherwise decoding goes back to step-2.

3.2 Min-Sum Algorithm

SPA is a computationally complex algorithm. There are several approximations to SPA. Min-Sum algorithm (MSA) approximates SPA with less operations but at the cost of error correcting performance.

Most of the complexity comes from check node processing. Nonlinear function $\Phi(x)$ given in (3.6) can be approximated such that check-to-bit message updating reduces to

$$R_{ji} = \min_{i' \in N(j) \setminus i} \left(|Q_{i'j}| \right) \prod_{i' \in N(j) \setminus i} sgn(Q_{i'j}) \tag{3.10}$$

This way decoder complexity is reduced at the cost of error performance. There are many correction methods proposed in the literature. In the following section we will look at one of the most popular correction methods, which has been used in performance tests performed in this work as well.

3.3 Normalized Min-Sum Algorithm

Several correction methods have been proposed to recover performance degradation in MSA approximation. Normalized-MSA improves decoding

performance by downscaling overestimated check-to-bit messages. NMSA introduces normalization factor α to improve decoding performance.

$$\boldsymbol{R_{ji}} = \alpha\left(\min_{i' \in N(j)\setminus i}\left(|\boldsymbol{Q_{i'j}}|\right) \prod_{i' \in N(j)\setminus i} sgn(\boldsymbol{Q_{i'j}})\right) \quad \textbf{(3.11)}$$

In (3.11) normalization constant α is usually chosen as 0.8. NMSA avoids biased estimation of check-to-bit messages.

3.4 Dual-Scaling MSA (DS-MSA)

In [5] Chang proposed a new decoding algorithm, Dual-Scaling Min-Sum Algorithm (DS-MSA) to compensate deficiencies of MSA approximation. There is a slight difference between N-MSA and DS-MSA methods. While N-MSA uses single scaling factor, DS-MSA uses two scaling factors. The difference can be summarized as follows:

$$R_{ji}{}^{tmp} = \min_{i' \in N(j)\setminus i}\left(|Q_{i'j}|\right) \prod_{i' \in N(j)\setminus i} sgn(Q_{i'j}) \quad \textbf{(3.12)}$$

First R_{ji} is calculated the same as in the MSA method given in (3.12). Based on value of Q_{ij} another comparison is made and R_{ji} is recalculated as given in (3.13)

$$R_{ji} = \begin{cases} \alpha_1 R_{ji}{}^{tmp} \; if \; R_{ji}{}^{tmp} \le Q_{ij} \\ \alpha_2 R_{ji}{}^{tmp} \; if \; R_{ji}{}^{tmp} > Q_{ij} \end{cases} \quad \textbf{(3.13)}$$

3.5 Self-Corrected Min-Sum Algorithm

In [21] Savin proposes a correction for overestimation of check-to-bit messages. Unlike Normalized-MSA where normalization factor is used, it modifies the bit node processing by erasing unreliable messages. Unreliable messages are detected whenever any variable node changes its message sign between consecutive iterations. Sign fluctuation leads to message cleaning.

In the Self-Corrected Min-Sum decoding, initialization, check node processing steps are the same as in Min-Sum algorithm. But variable node processing is modified as shown below:

$$Q_{ij}^{tmp} = LLR_i - R_{ji} \tag{3.14}$$

$$Q_{ij} = \begin{cases} Q_{ij}^{tmp}, & if\ sgn(Q_{ij}) == sgn(Q_{ij}^{tmp}) \\ 0, & else \end{cases} \tag{3.15}$$

In (3.14) new extrinsic information is calculated for the current iteration. This value is used for comparison with previous value that has been sent. If two messages have the same sign, then temporary value calculated in (3.14) is used as variable node message, otherwise variable node message is set to 0 as shown in (3.15).

3.6 Layered Decoding of LDPC Codes

The main decoding method for LDPC codes is the Belief Propagation (BP) algorithm. Common message-passing schedule used in the BP is two-phase message-passing (TPMP) schedule or flooding schedule. One problem with flooding schedule is its slow convergence. It is not efficient in terms of convergence. In order to achieve higher convergence speed, turbo decoding message-passing (TDMP) schedule or "layered decoding" is proposed [9]. In layered decoding parity-check matrix is divided into horizontal or vertical submatrices called layers. Reliability information updating occurs after each layer. In each sub-iteration, reliability values are updated and these intermediate updated messages are used for the next sub-iteration. This allows soft outputs of bit nodes to converge faster than flooding schedule.

There are two types of layered decoding schemes: Horizontal and vertical. In the horizontal layered decoding, also called serial-C schedule, parity-check matrix is divided into horizontal layers and check nodes in that layer are updated first, then whole neighbouring bit nodes are updated. In vertical layered decoding, also called serial-V schedule, parity-check matrix is divided into vertical layers and bit nodes in that layer are updated first, followed by update of corresponding check nodes in the layer. Both layered decoding schemes proceeds layer after layer. In this thesis, horizontal layered decoding scheme is used.

The key advantage of layered decoding is faster convergence speed achieved by reducing number of iterations by 50%. Another advantage of serial schedules is that they can be implemented with lower memory requirements.

Steps of layered decoding algorithm are summarized by the below pseudo code. The MS algorithm can be independently applied to layered decoding. By this way hardware implementation is greatly reduced. Initialization step is the same as flooding schedule. In this stage, log-likelihood ratios of bit nodes set to their initial values as shown in equations (3.16) and (3.17) where $\mathrm{y_i}$ denotes corresponding received value from the channel and σ^2 denotes channel's noise variance. Channel noise is assumed to be AWGN throughout the work.

Initialization-Step:

$$R_{ji} = 0 \tag{3.16}$$

$$Q_i = \frac{2y_i}{\sigma^2} \tag{3.17}$$

Process-Step:

for *layer k=0* **to** n_b **do**

for *j* ∈ *check nodes of layer* **do**

$$Q_{ij} = Q_i - R_{ji} \tag{3.18}$$

$$R_{ji} = \alpha\left(\min_{i' \in N(j)\backslash i}\left(|Q_{i'j}|\right)\prod_{i' \in N(j)\backslash i} sgn(Q_{i'j})\right) \tag{3.19}$$

$$Q_i = Q_{ij} + R_{ji} \tag{3.20}$$

Parity-check matrix is divided into n_b layers. Then, for all *i* (bit node) in the kth-layer of the rows, (3.18)-(3.20) are repeated for one layer after another. Steps (3.18)-(3.20) constitute a decoding sub-iteration. There are as many number of layers as sub-iterations. In equation (3.18) bit node messages are updated where Q_{ij} is the message from variable node *i* to check node *j*, Q_i is the a posteriori probability (APP) message of variable node i. In equation (3.19) check node messages are updated where R_{ji} is message from check node *j* to variable node *i*. Scale factor α is used to normalize overestimated check to bit messages. Lastly, APP values are updated as given in equation (3.20). Extrinsic information updated by processing of earlier layers are used as input for the processing of subsequent layers. Since updated

messages are used within an iteration, refined estimates spread faster among nodes, speeding up the convergence of the decoder.

Layered decoding method can be applied to decode any LDPC code. It accelerates convergence rate of the decoder by a factor of two while maintaining performance gain. In the following sections, we will propose new layered decoding schemes for decoding of LDPC codes. All proposed methods use the same concept outlined in this section for exchanging messages between check nodes and bit nodes. The difference lies in the process order of layers. In the proposed schemes, process order of layers is different but message updating rules remain the same.

3.7 Parallel Decoding of LDPC Codes

One important advantage of LDPC codes over Turbo codes is parallelism. Decoding of LDPC codes can be done in parallel using multicore architectures. In flooding schedule, all bit nodes and check nodes are processed together. Work is shared between cores by dividing parity check matrix into the horizontal and vertical non-overlapping strips for check to bit and bit to check processing, respectively. The parity-check matrix is divided evenly into equal strips in order to distribute load into cores equally. In multicore implementation of NMSA, each of selected number of cores concurrently processes messages sent form bit node to check node or vice-versa. We implement parallel decoding of LDPC codes on a real DSP hardware. The target hardware is Texas Instrument's TMDSEVM6678LE Evaluation Module. It contains 8 identical C6678 DSP cores. Parallelization of LDPC decoding is made possible by using OpenMP platform. More information about target hardware, OpenMP platform, implementation details and performance results are presented in Section 6.

4 PROPOSED IMPROVEMENT METHODS

Traditional layered decoding does not differentiate any layers to be processed first or last, layers are processed without a specific order. However, processing order of the layers has an impact on error performance of decoder. In this section, we propose different layered decoding methods where layers are processed in the order defined by our criterions. We define three methods to find the optimal process order of layers based on syndrome-check. Section 4.1 and Section 4.2 discusses the proposed fixed order and dynamic order scheduling of layered decoding of LDPC codes. A satisfaction weight order method is also introduced for layered decoding of LDPC codes. Since syndrome check is a part of the iterative decoding algorithm, our methods put a little computation burden on the existing methods.

4.1 Fixed Order Layered Decoding of LDPC Codes (LOCK)

In fixed order layered decoding scheme, layers are ordered for one time, and the order is kept the same throughout the iterative decoding process. The first iteration is an exception to this rule. In the first iteration, no specific order is imposed. Layers are processed sequentially from 0 to n_b. In the second iteration, layers are ordered according to the result of syndrome check made at the end of first iteration. For the purposes of ordering, a list that specifies the order of layers is generated. Satisfied check nodes are put at the top of the list, followed by unsatisfied nodes. This process is exemplified in Figure 4.1.

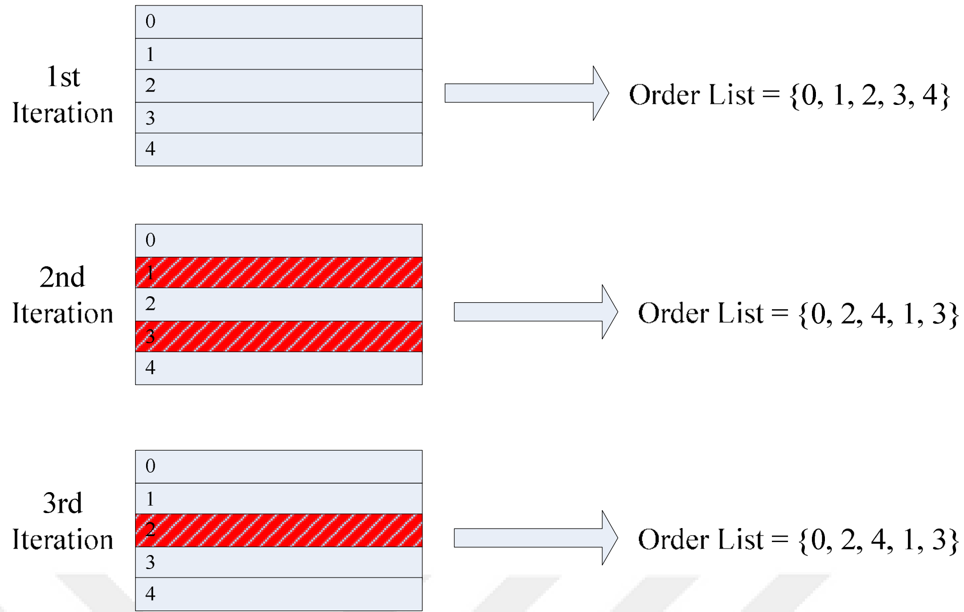

Figure 4.1 Fixed order layered decoding example

In Figure 4.1, an LDPC code with 5 layers is assumed. In fixed order layered decoding scheme, layers are processed sequentially in the first iteration as illustrated at the top of Figure 4.1. After the completion of syndrome check at the end of the first iteration, check nodes 1 and 3 become unsatisfied. Now, the processing order of layers is updated so that satisfied check nodes appear at the top of the list and unsatisfied check nodes are moved to the end of the list. This processing order list is preserved until the end of decoding. As seen in

Figure 4.1, check node 2 becomes unsatisfied at the end of second iteration but order list is not modified. Processing is done at the fixed order specified in the second iteration.

4.2 Dynamic Order Layered Decoding of LDPC Codes (ORD)

Like fixed order layered decoding scheme, dynamic order layered decoding method uses syndrome values to decide the processing order of layers. Layers are sequentially processed in the first iteration as in the previous method. But at the end of each iteration, the order list is dynamically updated. Although, the dynamic ordering requires more computation than its fixed ordering technique, our results show that dynamic order method has better error performance. In Figure 4.2, dynamic order method is exemplified using the same example given in Section 4.1.

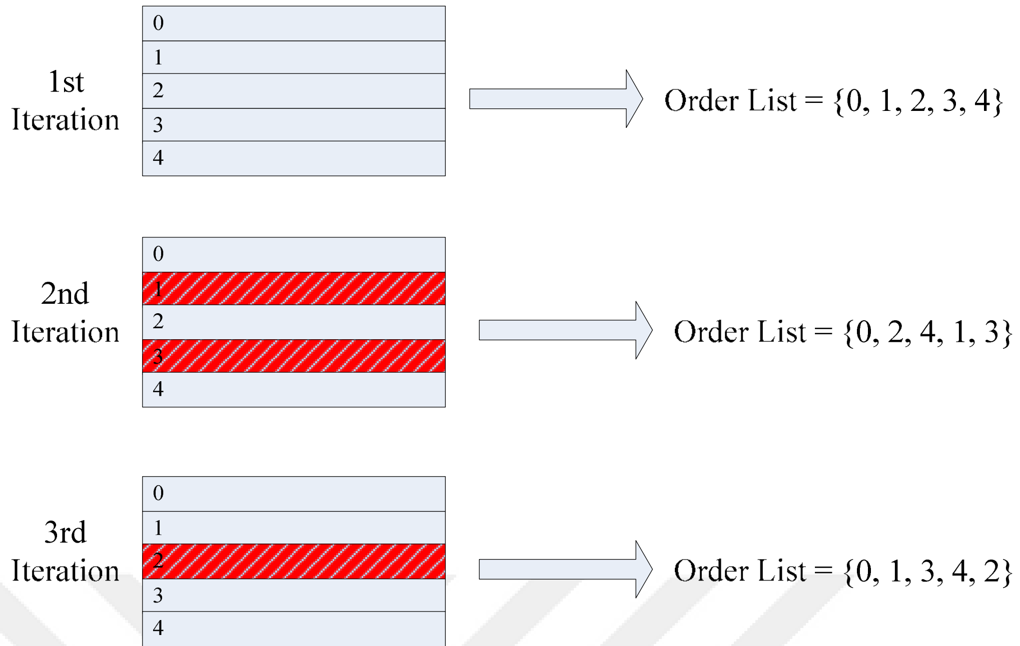

Figure 4.2 Dynamic order layered decoding example

As can be seen in Figure 4.2, the order list is updated at the end of each iteration to ensure that satisfied checks are processed first. At the end of second iteration only check node 2 is unsatisfied, and other check nodes are satisfied. Order list is updated based on these syndrome values and therefore, check node 2 is relocated at the end of the order list.

4.3 Satisfied Weight Order Layered Decoding of LDPC Codes (ORDE)

In the previous sections we have discussed new methods for layered decoding of LDPC codes with two different ordering. Process order of layers is decided based on syndrome values of check nodes. The processing ordering rule was simple: satisfied check nodes are processed first, whereas unsatisfied check nodes are processed last. Only distinction made among check nodes is check node being satisfied or unsatisfied. In satisfied weight order decoding method, we further evaluate check nodes based on their "total check node satisfaction". This concept can be explained more clearly with the help of an example given in Figure 4.3.

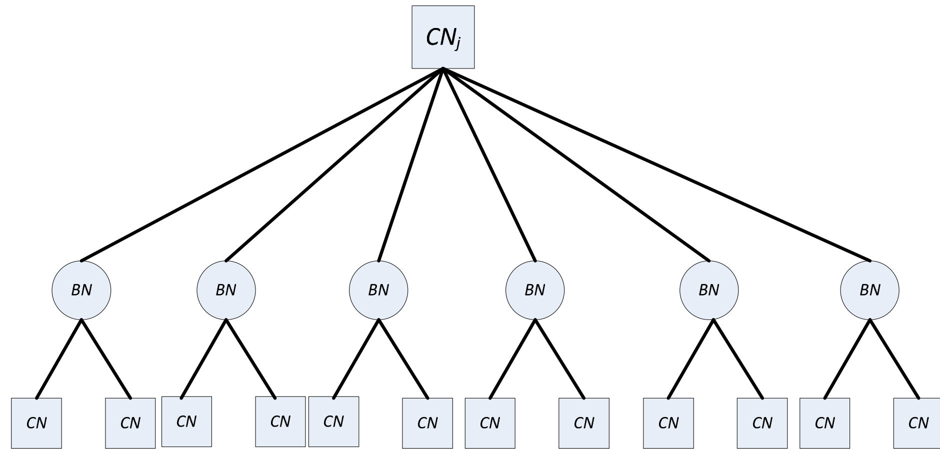

Figure 4.3 Example regular LDPC code

In Figure 4.3, a regular LDPC code with a constant row weight of 6 and constant column weight of 3 is assumed. Check node, CN_j that we will rate has connections to 6 bit nodes. Each bit node has connection to 3 check nodes including the check node that we will evaluate. In satisfaction weight order method, we first calculate total unsatisfied neighbouring check nodes of the check node CN_j. For example, assume that check nodes in red of

Figure 4.4 are unsatisfied. So in this case, total unsatisfied neighbouring check nodes for the check node CN_j is 3. In satisfaction weight order method, every check node is evaluated based on this criteria and, check node with the highest total unsatisfied rating is processed last. As satisfaction weight order method is dynamic, the process order of layers is re-evaluated at the end of every iteration based on the their satisfaction weight values.

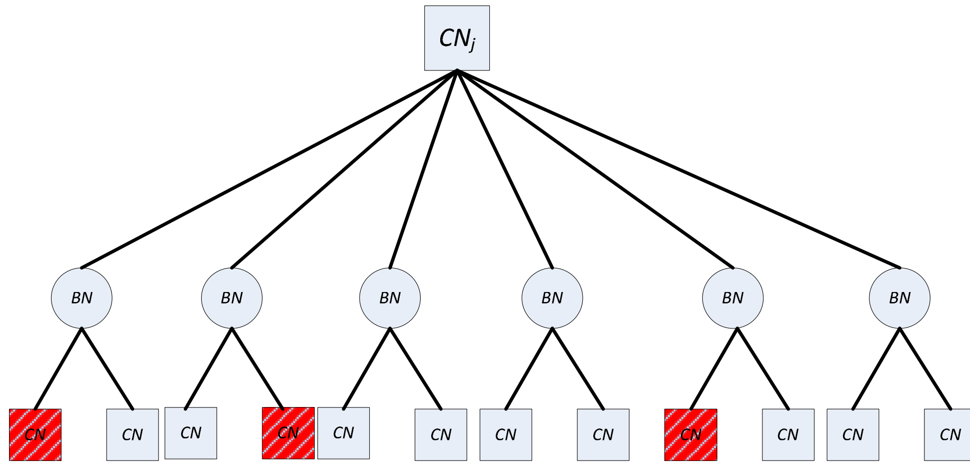

Figure 4.4 "Total unsatisfied check node" calculation

In the next section we apply proposed methods to selected codes from well-known standards and compare their performance results. In the results given in the next section, LOCK, ORD and ORDE simply refer to fixed order, dynamic order and weight order layered decoding methods defined in this section, respectively.

5 PERFORMANCE RESULTS OF PROPOSED METHODS

In this section, we apply standard layered decoding method as well as the proposed methods of Section 4 on different LDPC codes defined in WLAN, WPAN and DVB-S2 standards and compare performance results of investigated methods. Both short and long length codes with different rates have been selected for performance evaluations.

It is assumed that encoded codewords are modulated with BPSK modulation and transmitted over an AWGN channel. In the layered decoder, two different maximum number of iterations, 50 and 100, are used. Aforementioned layered decoding methods have been applied to some well-known LDPC codes. We have made performance comparisons among many methods for layered decoding of LDPC codes. In the simulations presented in this thesis, we have used sequential check-node update scheme for layered decoding as presented in [10]. The simulation results for Word Error Rate (WER) are summarized in this section. We use the following shorthands to distinguish between different methods. NMSA, LOCK, ORD and ORDE stand for the layered min-sum algorithm using the proposed scheduling method with normalized MSA, fixed order, dynamic order and satisfaction weight order methods, respectively.

5.1 Simulation of Proposed Methods on PC

In order to evaluate and compare performance of decoding algorithms, a simulation application is developed with C# in Visual Studio. The followings are key-features implemented and incorporated into the simulation software:

- Alist file parser to create parity-check matrix
- Sparse-matrix implementation for efficient use of memory
- Encoder for LDPC codes from 802.11n, 802.15.3c, DVB-S2
- BPSK modulation
- AWGN noise generation and channel simulation
- LDPC decoder implementation for flooding and layered schedules
- Parallel decoding of LDPC codes
- Generation of Matlab scripts to plot performance graphs

As an example, consider the LDPC code defined by the following parity-check matrix:

$$H_{9x12} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.1)$$

Alist file is a format for defining low density parity-check matrices. Alist file for the H matrix given in (5.1) is given in Table 5.1.

Table 5.1 Alist file contents

Line Number	***Content***	***Description***
1	12 9	N M
2	3 2	max_row max_col
3	3 3 3 2 2 2 3 3 3	Row weights
4	2 2 2 2 2 2 2 2 2 2 2 2	Column weights
5	1 5 12	Row positions (M=9 lines)
6	2 6 10	"
7	3 4 11	"
8	3 7	"
9	1 8	"
10	2 9	"
11	4 8 10	"
12	5 9 11	"
13	6 7 12	"
14	1 5	Column positions (N=12 lines)
15	2 6	"
16	3 4	"
17	3 7	"
18	1 8	"
19	2 9	"
20	4 9	"
21	5 7	"
22	6 8	"
23	2 7	"
24	3 8	"
25	1 9	"

Since H matrix is sparse, it is not effective to use two-dimensional arrays for use in decoder implementation. Instead jagged arrays are utilized. A jagged array is an array whose elements are arrays with different sizes. They can store efficiently many rows of varying lengths. In the application, we create jagged arrays to represent parity-check matrix in memory. $N(j)$ is set of variable nodes for check node j. To keep column positions of a check node we have the following jagged array. It has 9 arrays in total. Each array corresponds to the lines 5-13 in Table 5.1. Note that not all arrays have the same size.

Table 5.2 Array to keep column positions of check nodes

Check Node	Column Positions		
0	0	4	11
1	1	5	9
2	2	3	10
3	2	6	
4	0	7	
5	1	8	
6	3	7	9
7	4	8	10
8	5	6	11

Similarly, $M(i)$ represents set of check nodes for variable node i. The jagged array structure in Table 5.3 keeps row positions of variable nodes. It looks like a two-dimensional array because it is structured, i.e., all the arrays are of the same length.

Table 5.3 Array to keep row positions of variable nodes

Bit Node	Row Positions	
0	0	4
1	1	5
2	2	3
3	2	6
4	0	7
5	1	8
6	3	8
7	4	6
8	5	7
9	1	6
10	2	7
11	0	8

Extrinsic messages sent between nodes are stored in one-dimensional arrays. In decoder implementation, we need to give a unique id to each node participating in the parity-check matrix. In Table 5.4, this labeling process is exemplified. Each "1" representing an edge in the parity-check matrix is assigned a unique id.

Table 5.4 Identification of edges

	0	1	2	3	4	5	6	7	8	9	10	11
0	0				8							22
1		2				10				18		
2			4	6							20	
3			5				12					
4	1							14				
5		3							16			
6				7				15		19		
7					9				17		21	
8						11	13					23

In decoding algorithm, we need to access index of the node for variable node and check node processing. In check node processing, in order to compute check-to-bit messages we horizontally scan check nodes. Whereas in variable node processing we vertically scan parity-check matrix to calculate check-to-variable messages. The following index tables given in Table 5.5, Table 5.6 are used for these needs.

Table 5.5 Indexes for check nodes

Check Node	Indexes of Neighbour Bit Nodes		
0	0	8	22
1	2	10	18
2	4	6	20
3	5	12	
4	1	14	
5	3	16	
6	7	15	19
7	9	17	21
8	11	13	23

Table 5.6 Indexes for variable nodes

Bit Node	Indexes of Neighbour Check Nodes	
0	0	1
1	2	3
2	4	5
3	6	7
4	8	9
5	10	11
6	12	13
7	14	15
8	16	17
9	18	19
10	20	21
11	22	23

In simulations, we randomly generate information bits and encode it using an encoder. Encoder base class named "LDPCEncoder" implements all-zero codeword generation. Other classes "DVBS2Encoder", "WPANEncoder" and "WLANEncoder" classes are derived from the encoder base class as shown in Figure 5.1.

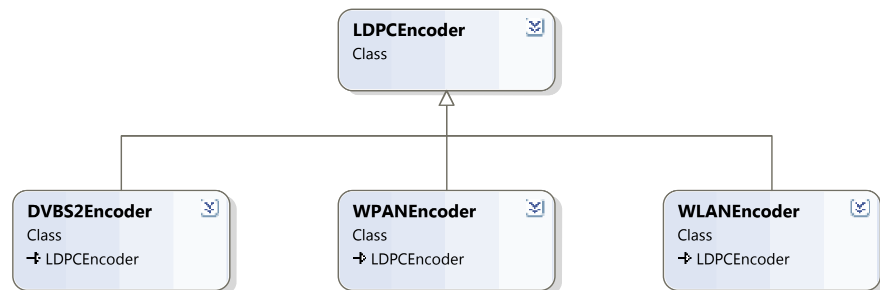

Figure 5.1 Encoder class diagram

Several decoding algorithms are implemented within the application. Object-oriented architecture is utilized to implement all decoders. One base class is implemented. All other decoders derive from this base class. If a decoder differs from base class in variable node processing or check node processing, it overrides methods inherited from the base class. Class diagram of the decoders can be seen in Figure 5.2.

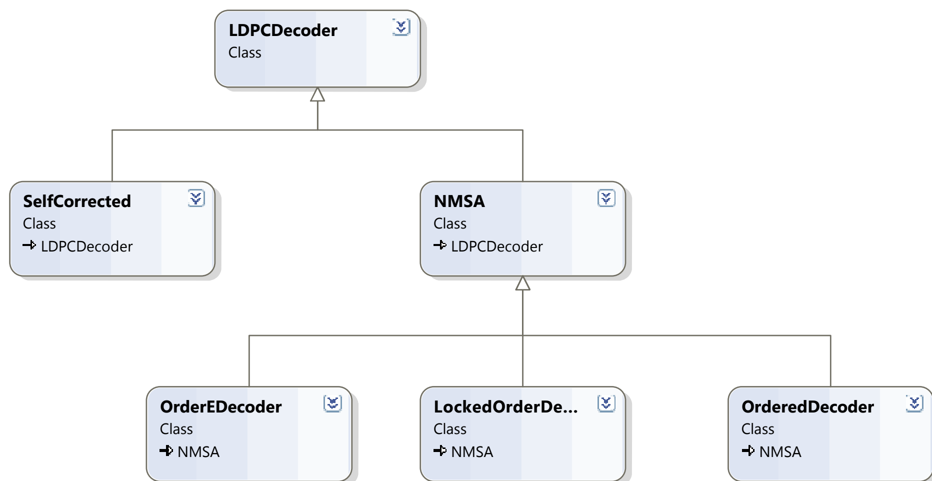

Figure 5.2 Decoder class diagram

In the application, parallelism where possible is added to speed up simulation. The LDPC decoding algorithm is an iterative process containing a lot of computations and constitutes the vast majority of the time spent in performance simulations. Therefore, master thread spawns a team of threads as many as different decoders. Each thread conducts a specific decoding algorithm in parallel. This is shown in Figure 5.3 as "Parallel Region". In Figure 5.3, four different decoding algorithm run in parallel to minimize the required simulation time.

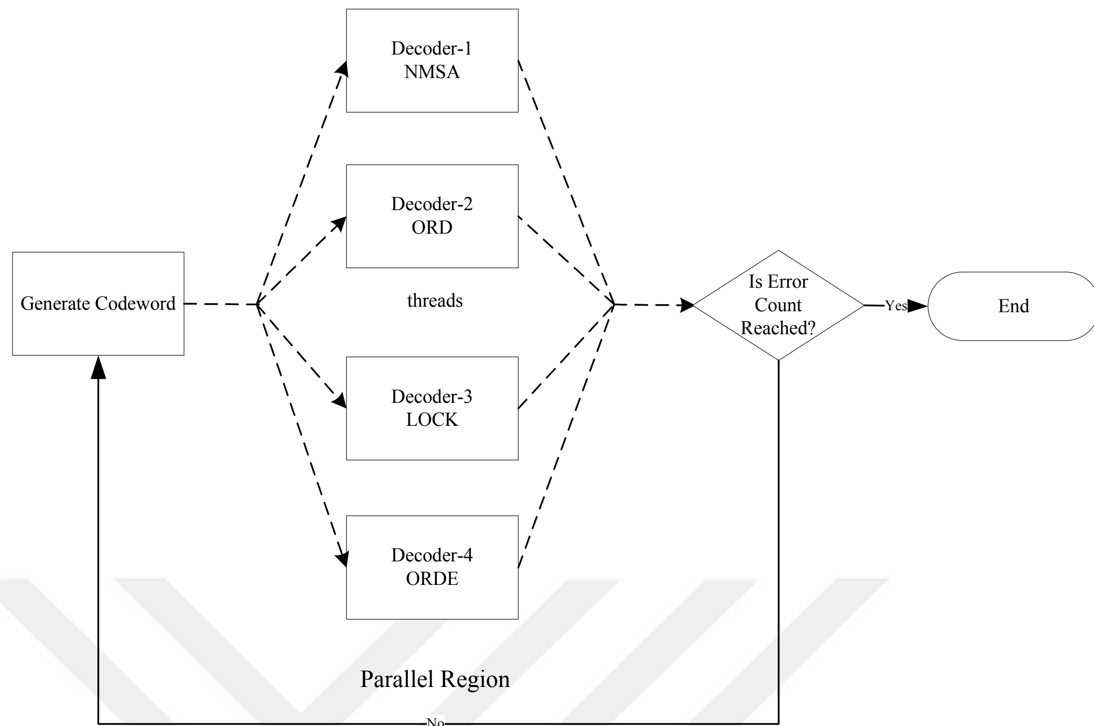

Figure 5.3 Decoding parallelism

Corresponding C# code excerpt for the above decoding loop is given below. The code gives an example of task parallelism. Task parallelism is often the simplest abstraction to use when a routine can be decomposed into independent separate tasks. The goal is to maximize processor utilization. If the methods execute sequentially, the total execution time would be the sum of the duration of each method. However, when a group of independent tasks start executing in parallel, total time spent is the elapsed time of the longest task. In the simulation software, codeword generated is passed as an input to all decoders, and decoding occurs in parallel. After decoding finishes, the results are collected and stored. Application makes extensive use of "Language Integrated Queries". LINQ is a general purpose query language to operate over collection of objects. Its parallel version "Parallel LINQ" (PINQ) is used to execute operations in parallel.

```
while (true) {
        GenerateCodeword();
        // parallel decoding
        Decoders.AsParallel().ForAll(x=>x.Decode(LLRInput, Codeword));
        for (int i = 0; i < Decoders.Count; i++){
                for (int j = 0; j < MaxIterationCounts.Length; j++){
                        WordErrors[point][i][j] += Decoders[i].WordError[j];
                        BitErrors[point][i][j] += Decoders[i].BitError[j];
                        Iterations[point][i][j] += Decoders[i].Iteration[j];
                 }
```

```
        }
        if (WordErrors[point].All(x=>x.All(y => y>=ErrorCountLimits[point])))
                break;
}
```

Blocks shown in the parallel region in Figure 5.3 are actually decoding blocks which carry out LDPC decoding steps given in Section 3. Figure 5.4 gives the flow diagram of LDPC decoding. As seen in Figure 5.4, LDPC decoding is an iterative process.

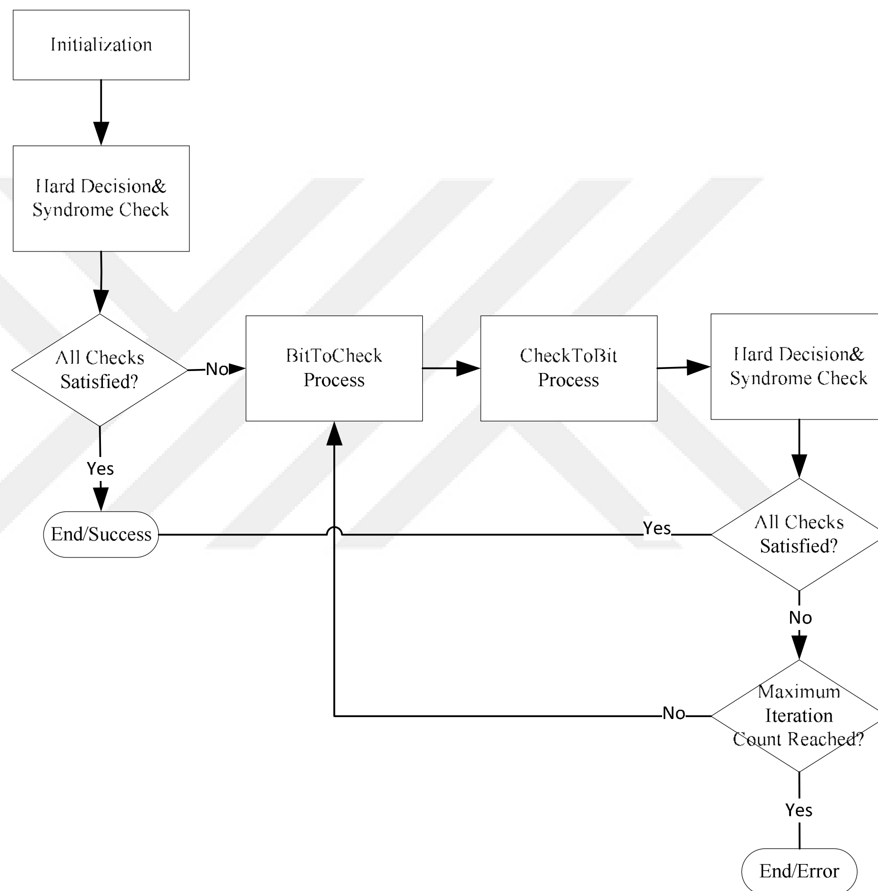

Figure 5.4 Flow diagram of LDPC decoding process

5.1.1 Simulation Results for 802.11n LDPC Codes

Proposed ordering methods and standard scheduling layered decoding methods which are discussed in Section 4 have been applied to the LDPC codes defined in IEEE 802.11n standards. Codes with two different codelengths (short&long) and

codes with two different rates (low&high) for each codeword length are chosen. Table 5.7 presents the selected LDPC codes from 802.11n standard.

Table 5.7 Selected LDPC codes from 802.11n standard

	Low-rate (1/2)	**High-rate (5/6)**
Short-length	(648, 324)	(648, 540)
Long-length	(1944, 972)	(1944, 1620)

Studied layered decoding schemes are applied to selected LDPC codes of 802.11n standards and the following WER performance results are obtained. WER results for a rate 1/2 with 648 codeword-length is given in given in Figure 5.5 and Figure 5.6, assuming 50 and 100 maximum iterations, respectively. As shown in Figure 5.5 and Figure 5.6, ORD (dynamic ordering) and ORDE (satisfaction weight ordering) methods display better performance for low SNR range, up to 1.9dB. After this point, performance improvement is not observed anymore.

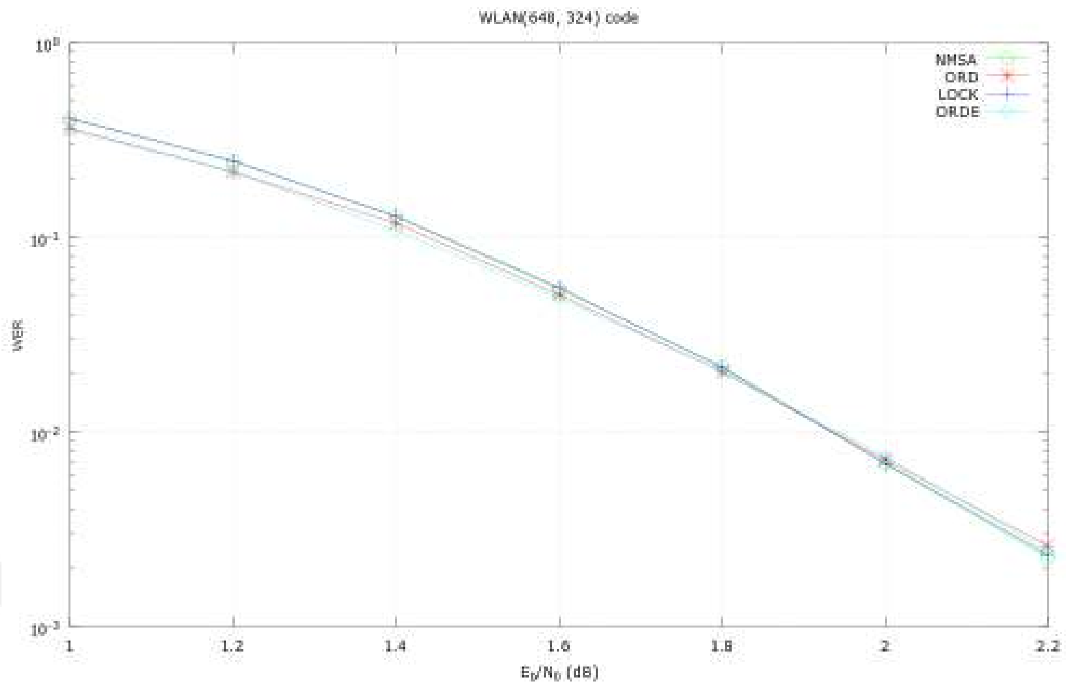

Figure 5.5 Performance results for WLAN, short-length, low-rate code with 50 max iterations

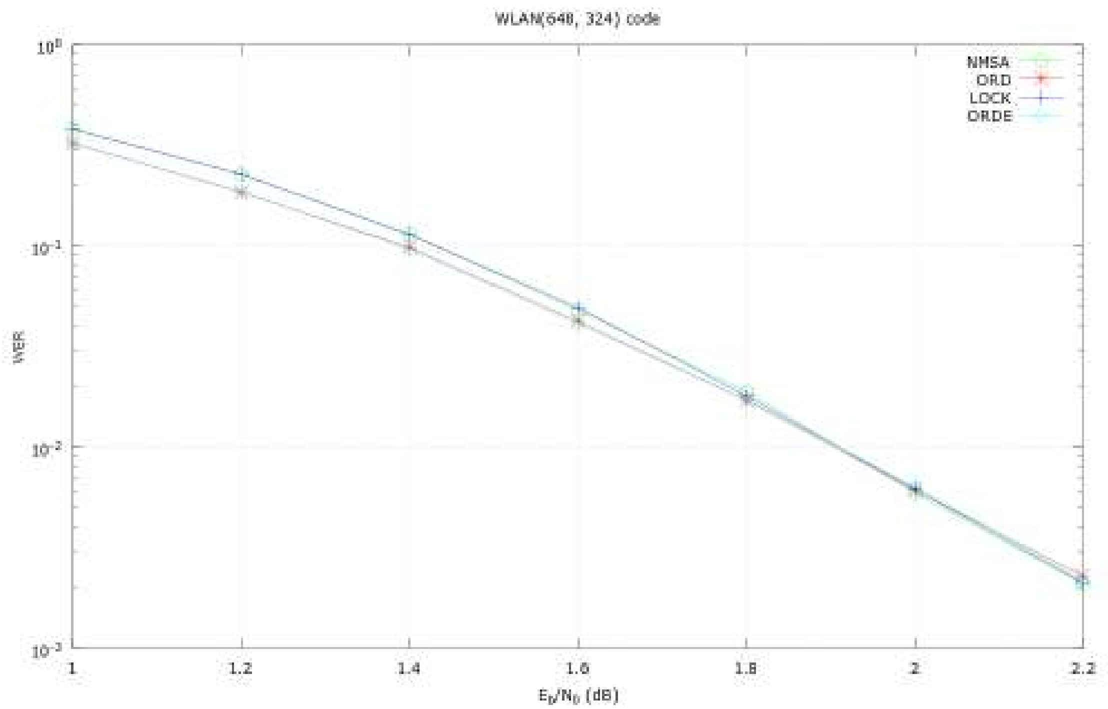

Figure 5.6 Performance results for WLAN, short-length, low-rate code with 100 max iterations

Performance results for a higher rate (5/6) with the same codeword length, 648, are given in Figure 5.7 and Figure 5.8, assuming 50 and 100 maximum iterations, respectively. Results show an improvement in performance in full SNR range.

Comparing Figure 5.7 and Figure 5.8, it is clear that with higher maximum number of iterations, the more performance gain is achieved.

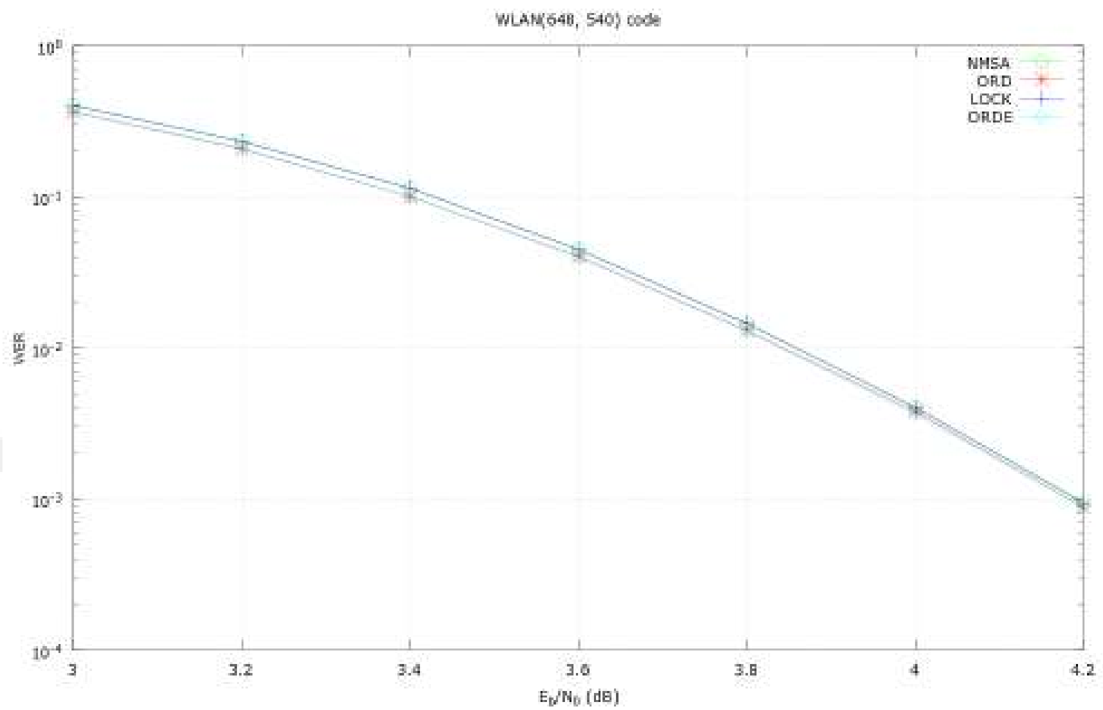

Figure 5.7 Performance results for WLAN, short-length, high-rate code with 50 max iterations

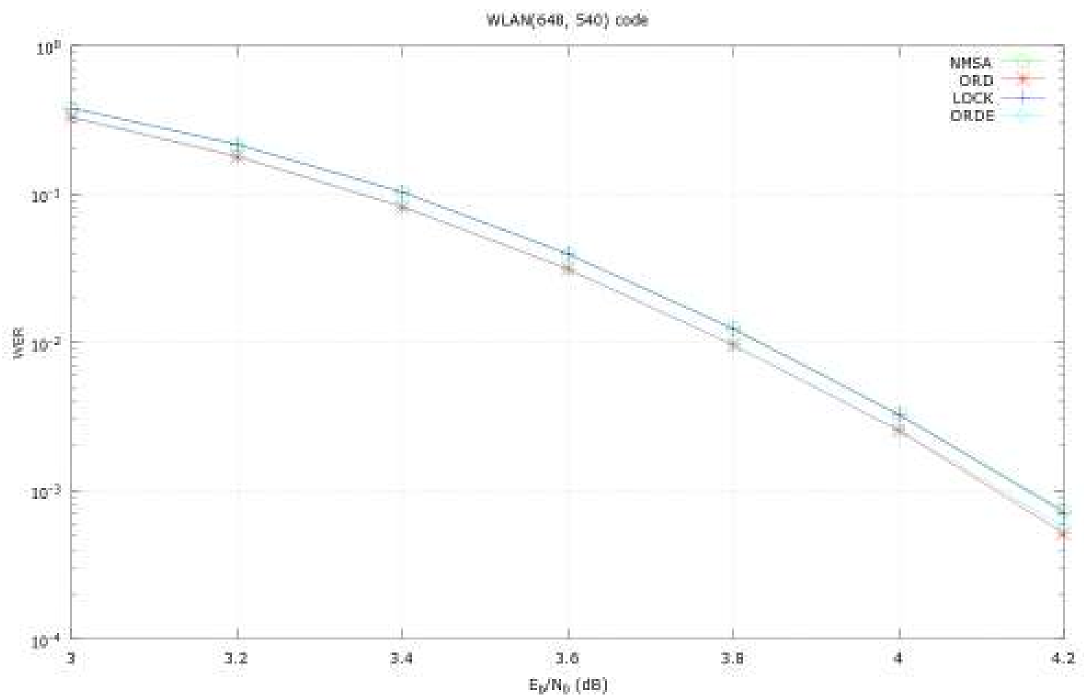

Figure 5.8 Performance results for WLAN, short-length, high-rate code with 100 max iterations

Table 5.8 presents maximum iteration values of NMSA and ORD (dynamic ordering) algorithms with (648, 540) LDPC code of IEEE 802.11n standard. The results clearly show that compared to NMSA, ORD algorithm achieves the same WER with much less iteration counts. For a WER of 10^{-2}, the ORD algorithm is able to converge twice as fast with only 48 iterations while NMSA requires 97 iterations to reach the same WER. As the WER gets lower, convergence speed improvement of ORD algorithm diminishes, but it is still able to achieve a significant 28% reduction in the iteration count at a WER of 10^{-4}. Consequently, it is clear that prioritizing the satisfied check nodes in processing order over unsatisfied checks leads to considerable convergence speed increase for decoding of LDPC codes.

Table 5.8 Iteration savings obtained by SBS algorithm with (648, 540) LDPC code

SNR (dB)	WER	NMSA Max. Iter.	ORD Max. Iter.	%
4.4	1.5x10-4	98	71	28%
4.2	7.2x10-4	97	69	29%
4.0	3.4x10-3	97	58	40%
3.8	1.2x10-2	99	57	42%
3.6	4.0x10-2	97	48	51%

Next, we apply the proposed ordering schemes for layered decoding to two other LDPC codes with 1944 bit-length from IEEE 802.11n standard. WER performance results of the code with rate 1/2 are given in Figure 5.9 and Figure 5.10, assuming maximum iterations counts of 50 and 100, respectively. ORD and ORDE (satisfaction weight ordering) methods perform slightly better for low SNR range. As SNR gets higher, improvement diminishes for ORD and ORDE methods. LOCK (fixed ordering) method gives nearly the same performance as NMSA. This characteristics is similar to what we obtain with short-length, low-rate code.

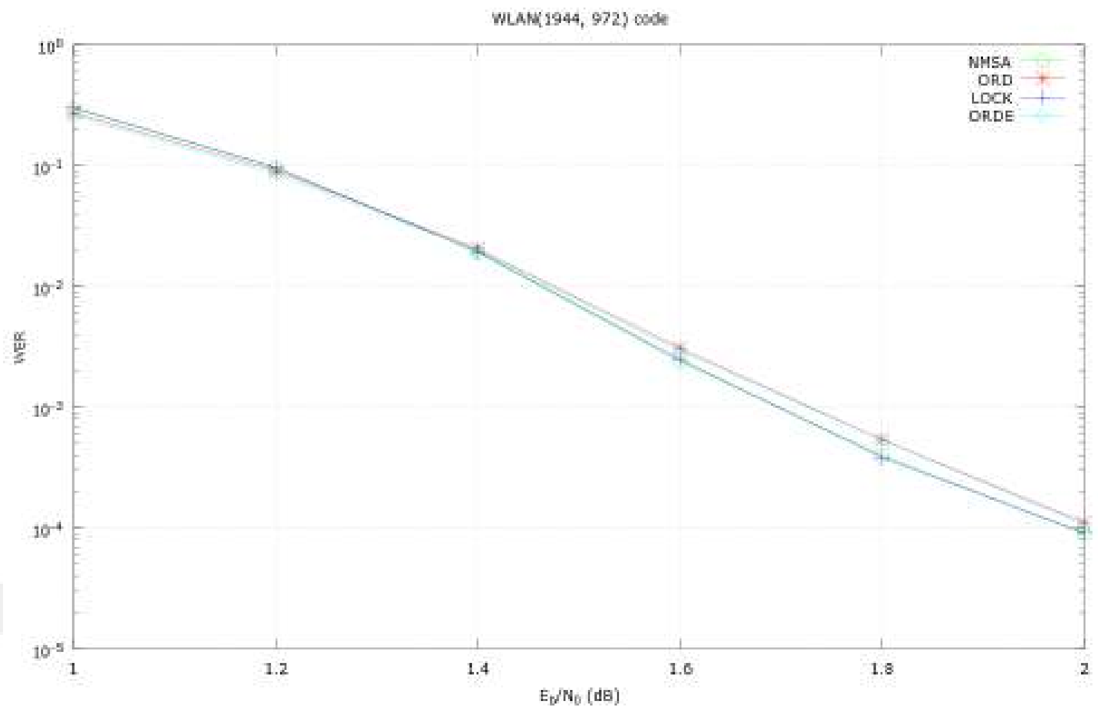

Figure 5.9 Performance results for WLAN, long-length, low-rate code with 50 max iterations

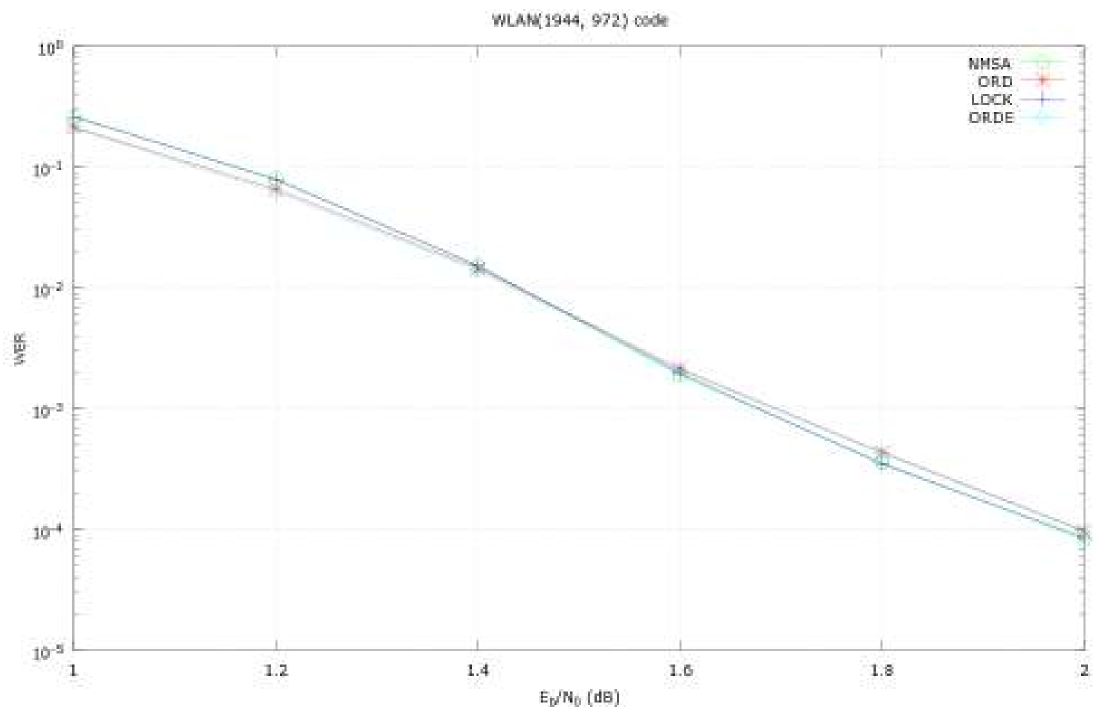

Figure 5.10 Performance results for WLAN, long-length, low-rate code with 100 max iterations

Figure 5.11 and Figure 5.12 presents the WER results for higher rate code with the same codeword-length of 1944, with maximum iteration counts of 50 and 100, respectively.

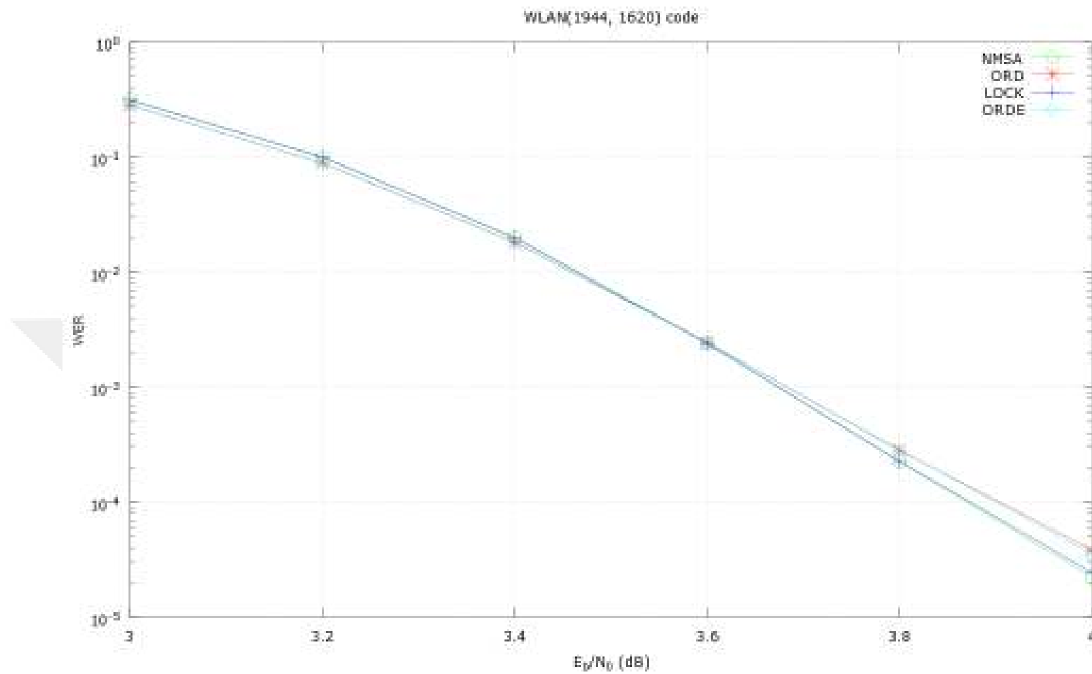

Figure 5.11 Performance results for WLAN, long-length, high-rate code with 50 max iterations

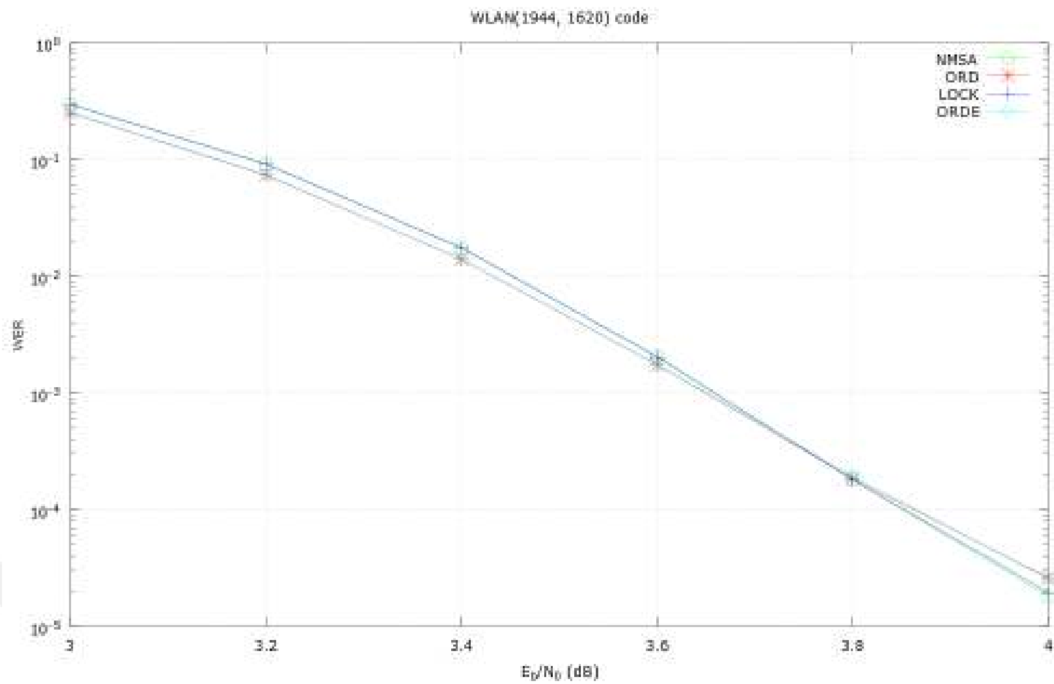

Figure 5.12 Performance results for WLAN, long-length, high-rate code with 100 max iterations

5.1.2 Results for 802.15.3c LDPC Codes

In this section, we apply proposed decoding methods to the LDPC codes defined in IEEE 802.15.3c (WPAN) standard. As in Section 5.1.1, LDPC codes of different lengths and rates are chosen for evaluation. The selected codes are given in Table 5.9.

Table 5.9 Selected LDPC codes from 802.15.3c standard

	Low-rate (1/2)	**High-rate (7/8)**	**High-rate (14/15)**
Short-length	(672, 336)	(672, 588)	-
Long-length	-	-	(1440, 1344)

WER results for 672-bit, rate 1/2 LDPC code are given in Figure 5.13 and Figure 5.14.

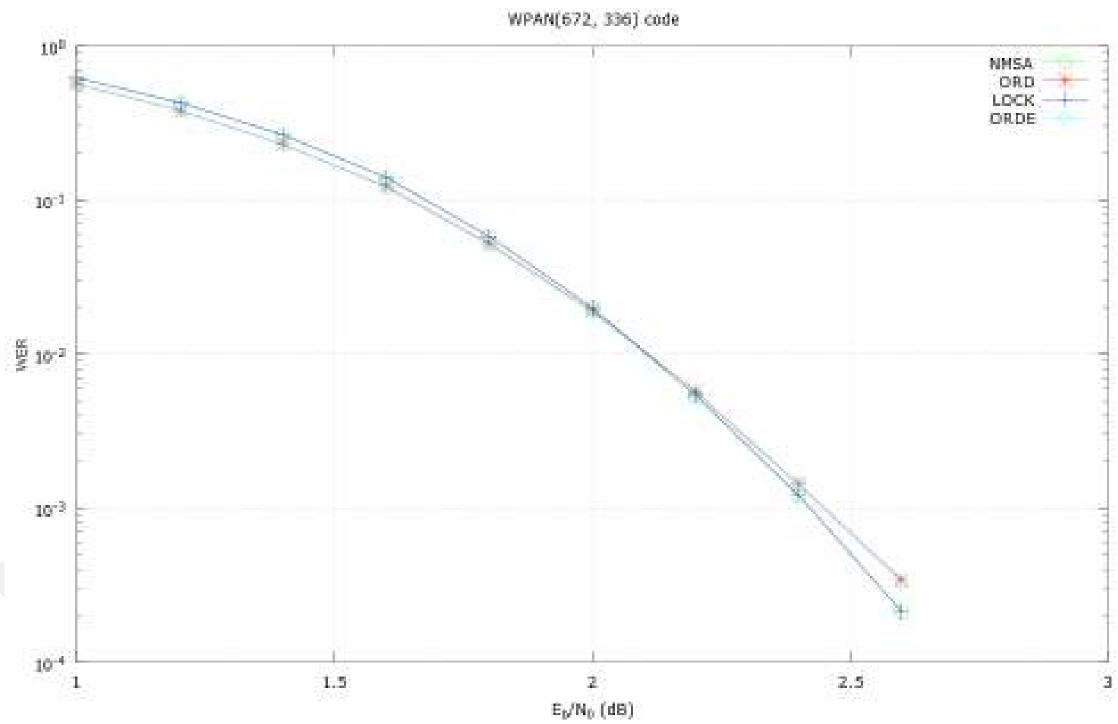

Figure 5.13 Performance results for WPAN, short-length, low-rate code with 50 max iterations

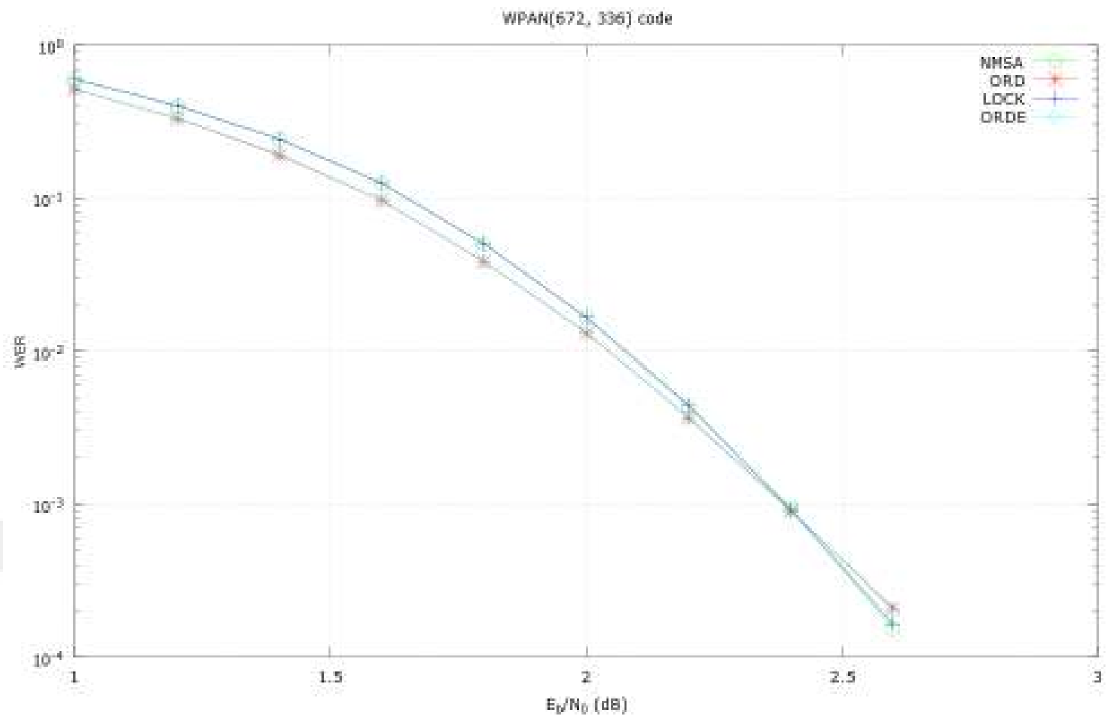

Figure 5.14 Performance results for WPAN, short-length, low-rate code with 100 max iterations

Performance results for short-length and high-rate WPAN codes are given in Figure 5.15 and Figure 5.16. Figures clearly show that ORD and ORDE methods provide performance improvement over NMSA and LOCK methods. This is in accordance with results obtained from short-length, high-rate codes from WLAN.

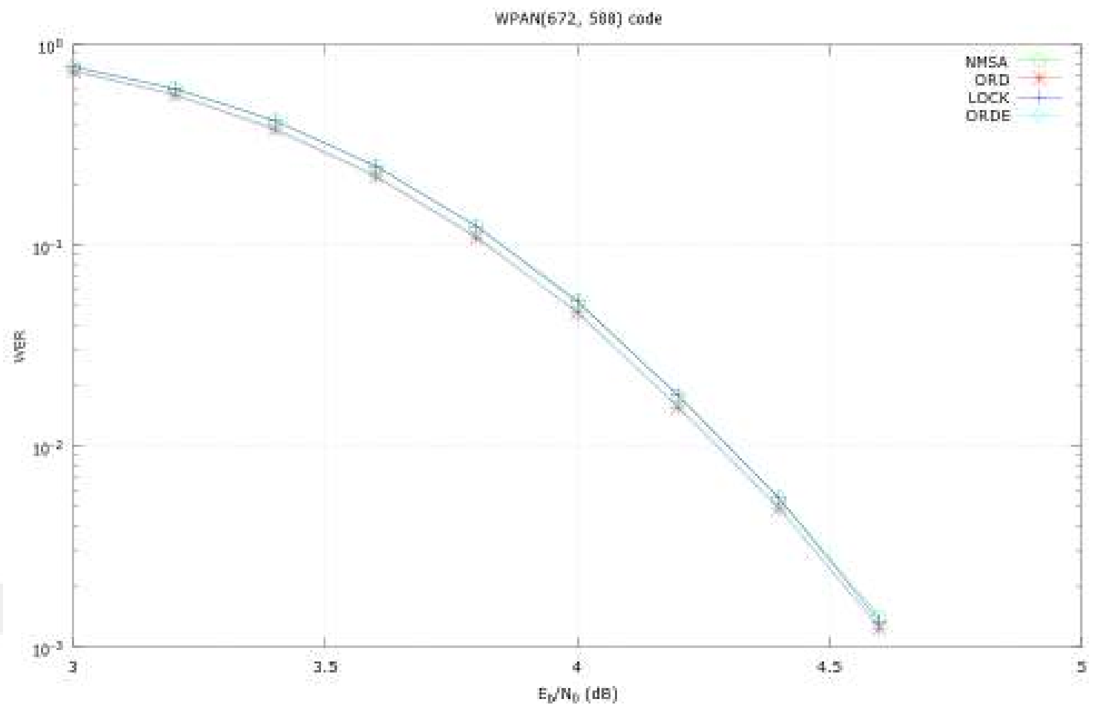

Figure 5.15 Performance results for WPAN, short-length, high-rate code with 50 max iterations

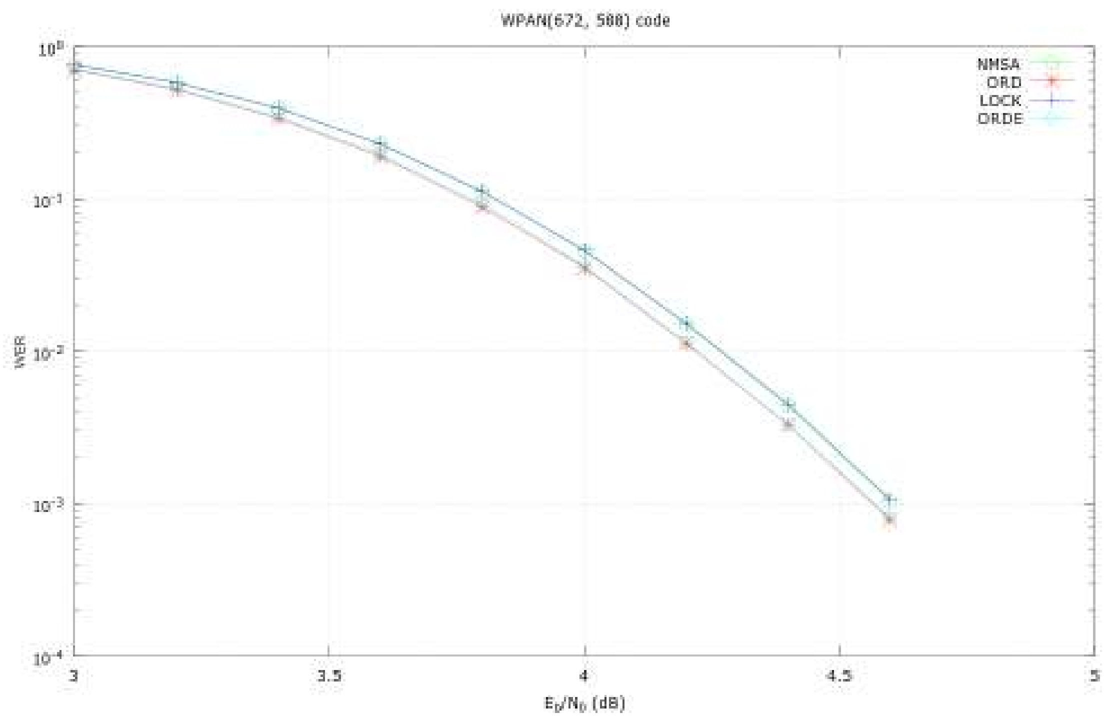

Figure 5.16 Performance results for WPAN, short-length, high-rate code with 100 max iterations

802.15.3c standard defines only one LDPC code with longer length. This is 1440 bit codeword-length, high-rate code. Figure 5.17 and Figure 5.18 show performance results of proposed methods against NMSA for maximum iteration counts of 50 and 100, respectively. Results are similar to the ones that are obtained from short-length, high-rate code, i.e. (672, 588) code.

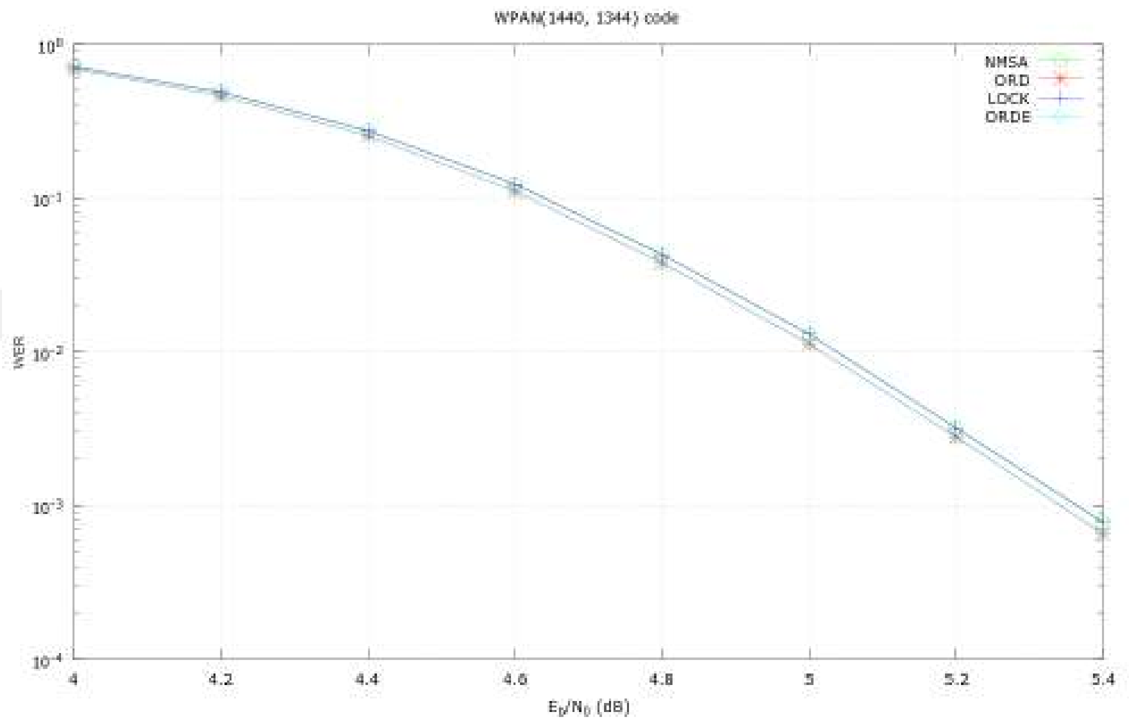

Figure 5.17 Performance results for WPAN, long-length, high-rate code with 50 max iterations

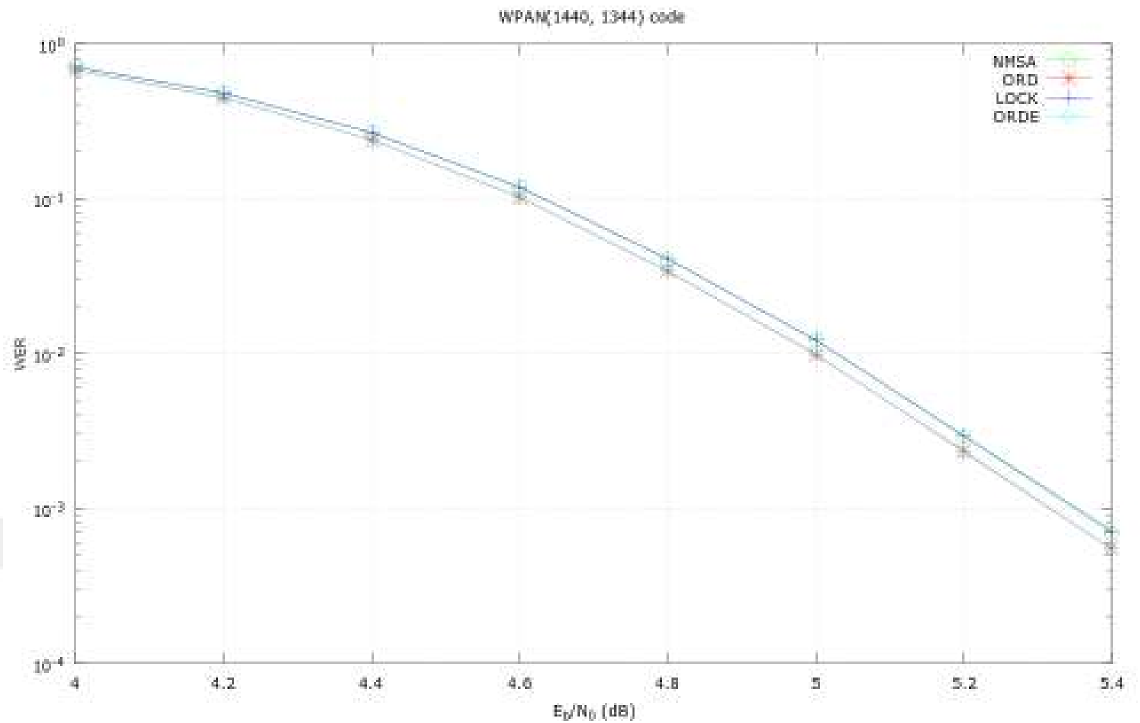

Figure 5.18 Performance results for WPAN, long-length, high-rate code with 100 max iterations

5.1.3 Results for DVB-S2 Codes

In this section, we have applied proposed ordering layered decoding methods to the codes from DVB-S2 standard. In simulations, codes with 16200 frame size are used. Figure 5.19 and Figure 5.20 present the WER results. As results show there is not much performance improvement achieved using proposed methods.

In Section 6, a simulation acceleration platform using OpenMP is described, parallel decoding is implemented on real hardware platform and obtained simulation speed gains are reported.

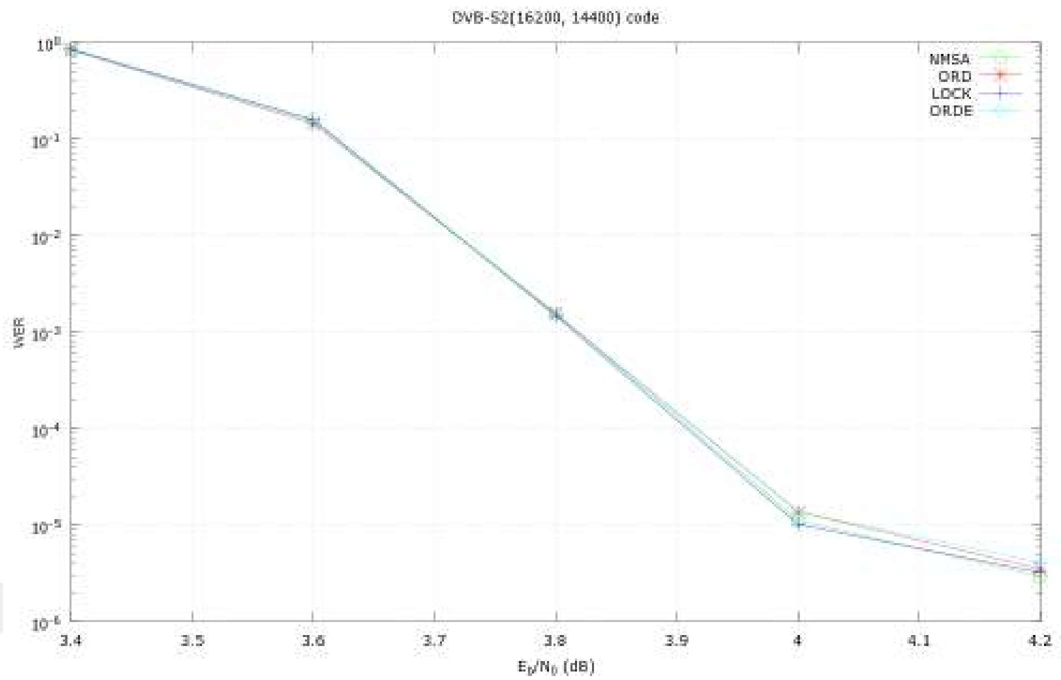

Figure 5.19 Performance results for DVB-S2, normal-frame, high-rate code with 50 max iterations

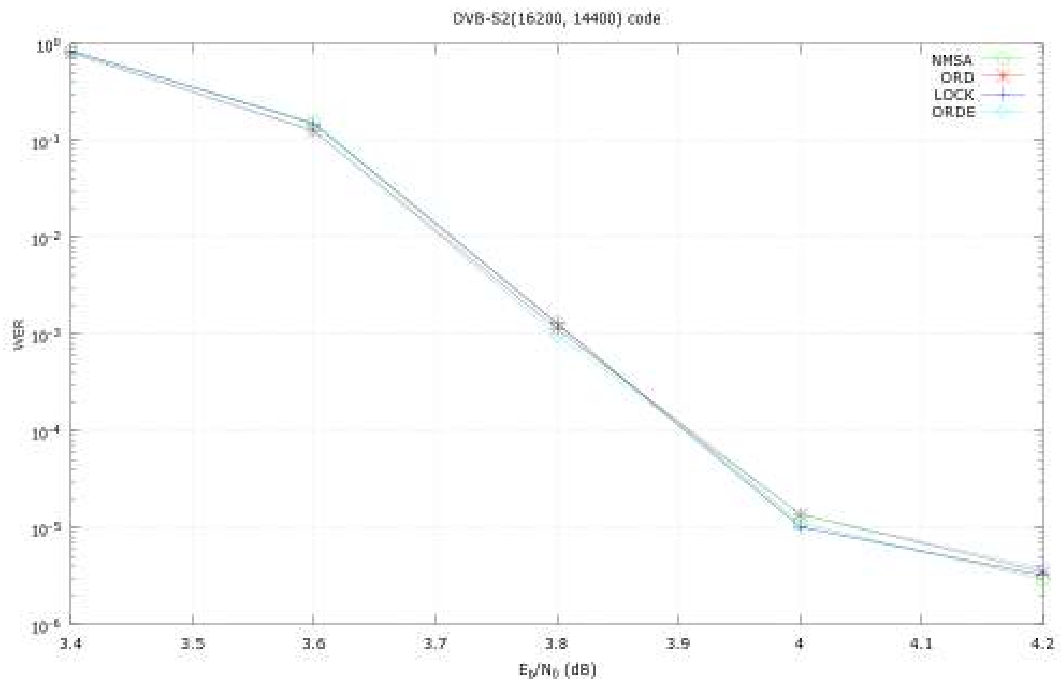

Figure 5.20 Performance results for DVB-S2, normal-frame, high-rate code with 100 max iterations

6 PARALLEL DECODING OF LDPC CODES ON REAL DSP HARDWARE

Despite widespread usage of LDPC codes, the highly complicated iterative decoding process leads to very long simulation times when verifying decoding performance under different decoding parameters. One attractive solution to long simulation times of LDPC codes is to implement inherently parallel decoding algorithms using multicore platforms. In this section, we present the first OpenMP parallel implementation of LDPC decoding algorithm on a multicore DSP architecture and report its performance. Parallelized Normalized Min-Sum decoding algorithm is implemented on 8-core Texas Instruments (TI) DSP using OpenMP framework. Performance results are obtained by Unified Instrumentation Architecture (UIA).

Multicore architectures considerably enhance data processing speeds, as multiple cores can handle many operations simultaneously. An important aspect of multicore approach is the proper migration of software development to the multicore environment. Parallelizing code to run over multicore platform requires some thread library and compiler support. Texas Instruments (TI) has recently introduced an OpenMP support into their development environment, which enables easy porting of single-core applications to multicore platform.

A brief information about OpenMP is presented in Section 6.1. Embedded platform used and implementation details for parallel decoding of LDPC codes are provided in Section 6.2, and Section 6.3, respectively. Finally, speedup achieved by parallelization is reported in Section 6.4.

6.1 OpenMP Framework

With multicore trend, parallelizing existing code previously written for running on a single core is one of the most challenging jobs for software developers. Several parallel programming models have been proposed so far, such as Message-Passing Interface (MPI), thread libraries (Pthreads), Open Computing Language (OpenCL) and Open Multi-Processing (OpenMP) to address multicore software development challenge. MPI is generally used for distributed memory architectures whereas

OpenCL is an open standard to write parallel applications for heterogeneous platforms. In contrast, OpenMP is a framework designed for shared memory multiprocessing. As LDPC decoding algorithm extensively uses shared memory for message passing between bit and check nodes, OpenMP framework is more suitable to parallelize the decoding algorithm of LDPC codes.

OpenMP is not a new programming language. It introduces some notation that provides straightforward port of existing sequential codes by simple use of compiler directives. The directives tell the compiler which part of the code executes in parallel and how to distribute them among the parallel threads. The compiler considers and interprets directives when application is enabled to use OpenMP. The program starts as a single thread of execution as any sequential program does. This thread is called initial thread. A team of threads is forked at the start of parallel region and joined at the end [23]. The start of parallel region is indicated by *#pragma omp parallel* directive. This fork-join process of OpenMP is illustrated in Figure 6.1.

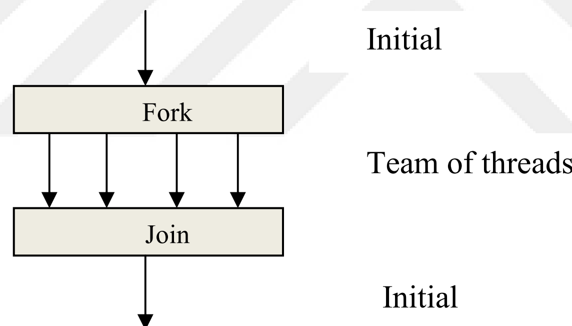

Figure 6.1 OpenMP Fork/Join model

Workload can be shared by all cores available using appropriate OpenMP work sharing constructs. The most common work sharing approach is to distribute the work in a for-loop among the threads in a team. However, not all loops can be suitable for this type of work sharing. Data dependencies may prevent loop parallelization. It is the user's responsibility to decide loop parallelization is applicable or not. OpenMP allows the programmer to control the number of threads that execute a parallel region. If it takes T_1 to execute an application on a single core and T_P to execute the same code on P cores then, parallel speedup can be defined as the ratio:

$$Speedup = \frac{T_1}{T_P} \tag{6.1}$$

If speedup increases with an increasing number of cores, the system is said to be scalable.

6.2 Embedded DSP Platform

In this work, performance improvement achieved by parallelization is evaluated on TI's Keystone based module, TMDSEVM6678LE, shown in Figure 6.2. The module features TMS320C6678 System-on-chip (SoC) which includes identical 8 TMS320C66x (C66x) DSP cores providing both fixed- and floating-point capability. TI's KeyStone family is a well-known low power DSP architecture, which consumes only 10W at 1GHz clock frequency. The KeyStone architecture is designed with tiered multicore memory architecture, allowing for full processing entitlement across all cores while executing concurrently [24]. It contains two levels of memory: separate 32kB local program and data memories exist at level 1 (L1P and L1D). There exists 512kB L2 memory separate to each core. The L1 and L2 memories can be configured to be used as cache, RAM or part RAM/part cache. The L1 local memories are configured as RAM entirely. L2 memory is used as local memory to store thread private variables.

Figure 6.2 TMDSEVM6678 Target module

The C6678 also integrates 4096kB internal memory (usually referred as MSMC memory) shared among cores. The MSMC inside SoC allows the cores to dynamically share the internal and external memories for both code and data. TI's OpenMP implementation configures MSMC in shared level 2 mode (SL2). In this mode, SL2 RAM is cacheable only within the local L1P and L1D caches. However, since cache coherency is not performed upon this state, shared variables concurrently

accessed and modified by different threads running on different cores must be placed into a non-cached shared memory segment. In order to allow this, OpenMP runtime creates a non-cached alias for entire MSMC memory range [25]. Shared variables are placed into non-cached portion of MSMC memory. Code and constants are put into cached part of the memory. This approach makes parallelization even faster. It is important not to overlap these two segments. The compiler translates OpenMP into multi-threaded code with calls to a custom runtime library built on top of SYS/BIOS, a lightweight native real-time operating system (RTOS), and inter-processor communication (IPC) protocols [25].

Keystone architecture provides an external memory access to DDR memory and many industry standard peripherals such as PCIe, Serial Rapid I/O (SRIO), Enhanced Direct Memory Access (EDMA). Multicore Navigator (MCN) allows data exchange among cores and peripherals. Figure 6.3 shows the general functional block diagram of the multicore processor used in this work.

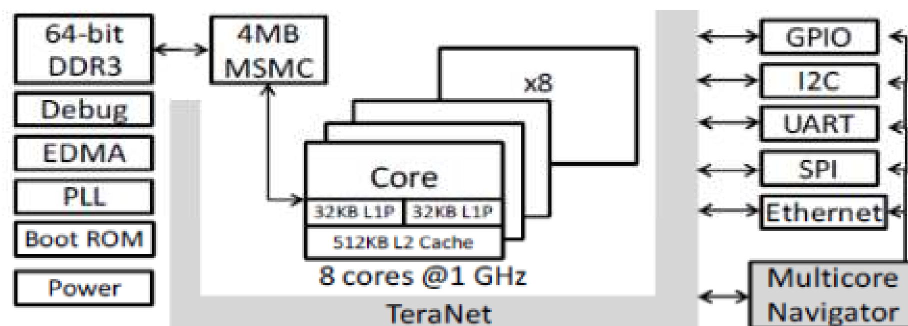

Figure 6.3 Reference DSP architecture

There is no operating system (OS) running at the processor level, instead each core executes its own instance of Real-Time Operating System (RTOS), called SYS/BIOS. It is a lightweight native RTOS provided by TI. A C/C++ compiler, debugger is provided in the development environment. The compiler supports OpenMP 3.0 to allow easy porting of existing single core code to multicore platform. The compiler translates OpenMP into multi-threaded code with calls to a custom runtime library built on top of SYS/BIOS and inter-processor communication (IPC) protocols [25]. TI also provides Code Composer Studio (CCS); an Eclipse based Integrated Development Environment (IDE) for code development. Figure 6.4 illustrates code development process for target system.

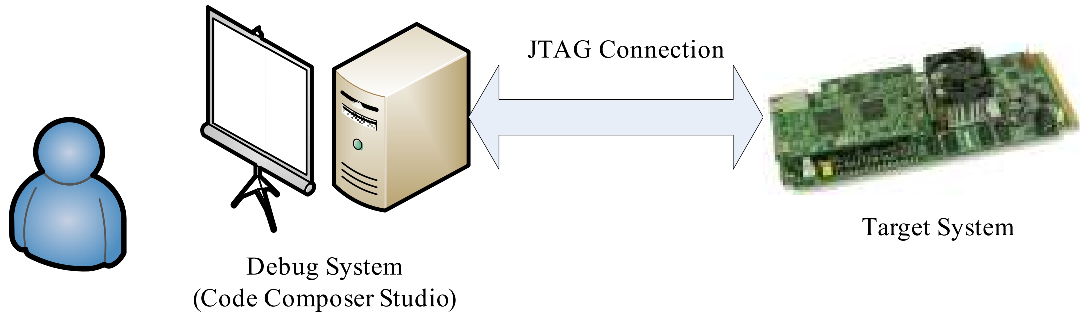

Figure 6.4 Application development with CCS

The Figure 6.5 shows the TI's OpenMP solution stack. Currently, OpenMP is supported on TI DSPs only for SYS/BIOS operating system. All OpenMP programs must be linked with the OMP run-time library [26].

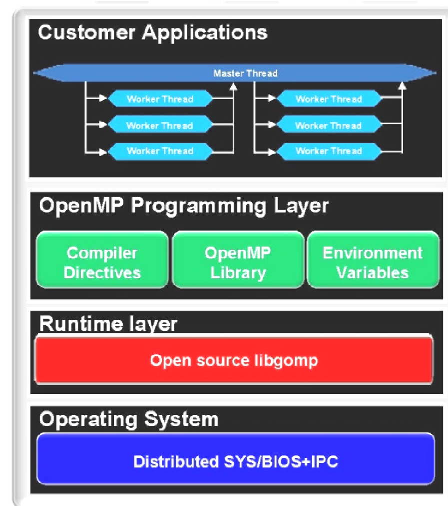

Figure 6.5 TI's OpenMP solution stack

6.3 Implementation Details for Parallel Decoding of LDPC Codes on DSP

In this work, a single core version of LDPC decoding algorithm is modified to run on TI's KeyStone multicore DSP platform, where an OpenMP-based multi-threading scheme is used to parallelize code. In this section, the key aspects of the implementation are described.

LDPC decoding is coded entirely in C using the Code Composer Studio™ Integrated Development Environment (IDE) v5.5, MCSDK v2.1.2.6 and SYS/BIOS v6.35.4.50. UIA v1.3.1.08 is utilized to instrument code and to provide benchmark statistics. OpenMP 3.0 is supported in evaluation platform. The project is built with

the maximum compiler optimization level. To verify improvement by multicore processing, we use 1 to 8 cores run the NMSA decoding algorithm simultaneously.

There are many options to improve the execution speed of code on DSP. With proper use of compiler directives and better memory placement and alignment better execution times can be achieved. Different data sections can be placed into specific region in memory by using appropriate directives. RTSC project provides a configuration file where necessary information is given to the linker on how to bind memory sections to the memory segments defined by a platform file. Code segment from configuration file shown below instructs the linker to map user defined section ".L2vars" into memory segment "L2SRAM":

```
// load const vars section into L2SRAM
Program.sectMap[".L2vars"] = new Program.SectionSpec();
Program.sectMap[".L2vars"].loadSegment = "L2SRAM";
```

Pragma directives tell the compiler how to treat a certain function or a section of code. In our implementation, compiler is guided to allocate memory from desired section. We allocate constant data on L2 memory for speed consideration:

```
#pragma DATA_SECTION (WeightRow, ".L2vars")
#pragma DATA_ALIGN ( WeightRow, 64 );
const unsigned char WeightRow[M] = {...}
```

The first step in creating an OpenMP program is to identify the parallelism it contains. Considering LDPC decoding, this is an easy task since LDPC decoding is inherently parallel. The parallel implementation using the eight cores on the TMS320C6678 DSP platform can be achieved by letting each core process a different portion of the parity check matrix. Work is shared between cores by dividing parity check matrix into the horizontal and vertical non-overlapping strips for check to bit and bit to check processing, respectively. The parity-check matrix is divided evenly into equal strips in order to distribute load into cores equally. In multicore implementation of NMSA, each of selected number of cores concurrently processes messages sent form bit node to check node or vice-versa. Bit to check processing, check to bit processing and syndrome check code blocks are parallelized using OpenMP directive *#pragma omp parallel for* specifying variable scope information by means of *private* and *shared* annotations. OpenMP runtime allows

different threads to run across different cores. OpenMP *#pragma omp parallel for* language extension offers easy parallelization of sequential code. Figure 6.6 and Figure 6.7 illustrate how the parity-check matrix is divided into horizontal and vertical strips and allocated among 4 cores DSP cores, respectively.

1	0	1	0	0	1	0	0	**Core#0**
0	1	0	0	1	0	0	0	
0	0	0	1	0	0	1	1	**Core#1**
1	0	1	0	0	1	0	0	
0	0	0	0	1	0	1	0	**Core#2**
1	0	0	1	0	0	0	0	
0	1	0	0	0	1	1	0	**Core#3**
0	0	1	0	0	0	0	1	

Figure 6.6 Check node processing using horizontal striping of H matrix

Core#0		Core#1		Core#2		Core#3	
1	0	1	0	0	1	0	0
0	1	0	0	1	0	0	0
0	0	0	1	0	0	1	1
1	0	1	0	0	1	0	0
0	0	0	0	1	0	1	0
1	0	0	1	0	0	0	0
0	1	0	0	0	1	1	0
0	0	1	0	0	0	0	1

Figure 6.7 Bit node processing using vertical striping of H matrix

We chose to store extrinsic messages in the non-cached shared memory region while constants are stored in the cached region to access them through L1D cache memory. This way each core has access to up-to-date data to process and writes updated messages back to their relevant places without any additional synchronization mechanism. Below code segment shows example work sharing loop construct for making hard decision given below. The work is shared between cores through vertical striping.

```
static void MakeHardDecision()
{
    int i;
    // make hard decision here
    #pragma omp parallel for
    for (i = 0; i < N; i++)
    {
        if (SumAll[i] > 0)
            CodewordHat[i] = 0;
        else
            CodewordHat[i] = 1;
    }
}
```

Decoding steps-2 and-3, given in Section 3.1 are repeated until maximum number of iterations is reached. We instrument code with log events in order to measure time spent in each step:

```
while(currentIter < MAX_ITER_COUNT)
{
  currentIter++;
  // start a new iteration
  Log_write2(UIABenchmark_start, (xdc_IArg)"iteration time, using %d cores", NumberOfCores);

  // start c2b
  Log_write2(UIABenchmark_start, (xdc_IArg)"c2b time, using %d cores", NumberOfCores);
  CheckToBitProcess();
  Log_write2(UIABenchmark_stop, (xdc_IArg)"c2b time, using %d cores", NumberOfCores);

  // start b2c
  Log_write2(UIABenchmark_start, (xdc_IArg)"b2c time, using %d cores", NumberOfCores);
  BitToCheckProcess();
  Log_write2(UIABenchmark_stop, (xdc_IArg)"b2c time, using %d cores", NumberOfCores);

  Log_write2(UIABenchmark_stop, (xdc_IArg)"iteration time, using %d cores", NumberOfCores);
}
```

6.4 Results for Parallel Decoding

After all considerations mentioned in the previous sections are taken into account in porting sequential decoding algorithm into multicore platform, we performed experimental analysis using TDMSEVM6678LE Evaluation Module (EVM), C6678 processor running at 1GHz. XDS560V2 emulator was used to program and debug code running on the EVM. Benchmark results are acquired while code is running on the target through JTAG connection. In our experiments, we applied NMSA to the codes from DVB-S2. Selected LDPC code has a codeword length of 16200 bits and rate of 8/9.

Figure 6.8 shows speedups achieved for each type of processes. Figure 6.8 also reveals that check node processing has a better scalability compared to bit node processing.

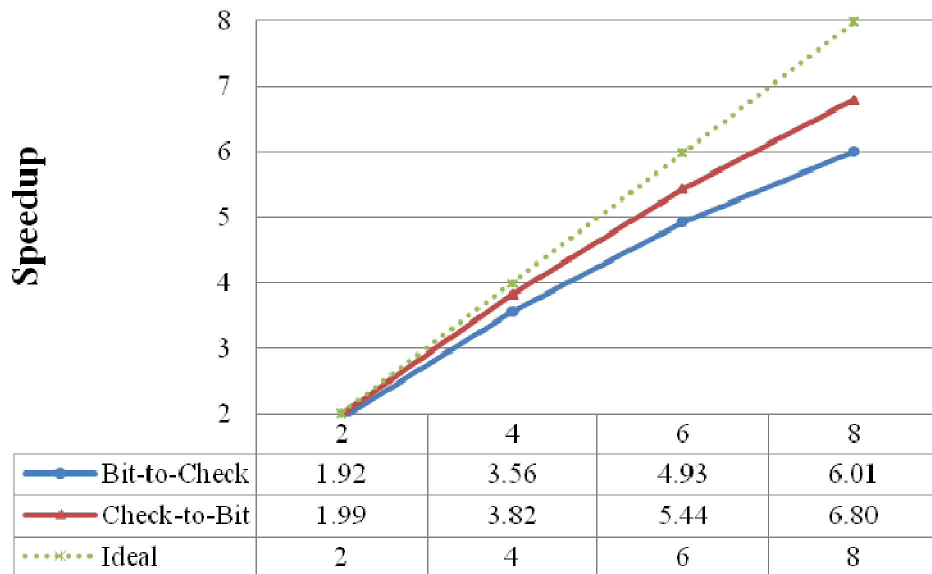

Figure 6.8 Speedup using OpenMP

Because LDPC decoding is an iterative process, time spent at each iteration is an important parameter. Table 6.1 shows the execution times of each iteration and achieved speedup with number of cores. It can be seen that 8-core implementation achieves a 6.4x speedup. The results also demonstrate scalability of LDPC decoding algorithm successfully on multicore DSPs. However, the throughputs achieved are far from those required for real-time execution.

Table 6.1 Execution times for an iteration and speedup versus number of cores

	Number of cores				
	1	***2***	***4***	***6***	***8***
Execution time (us)	6015	3081	1636	1167	947
Speedup		1.95	3.68	5.15	6.35

These results prove that inherently parallel decoding algorithm of LDPC codes can be easily parallelized using OpenMP, which provides flexible programming model to create parallel code for shared memory architectures, hiding internal synchronization details from the programmer.

7 CONCLUSIONS AND FUTURE WORK

In this thesis, three different ordering layered decoding schemes are presented to improve decoding performance of LDPC codes and a simulation acceleration platform using OpenMP is studied. The implementation of the proposed methods can be realized with small modifications, therefore the computational overhead involved is small. The proposed algorithms can achieve a good performance and a high convergence rate. We have simulated complete communication system in software. We assume BPSK transmission over AWGN channel. Our analysis results show that the maximum performance gain is observed for short length and high rate LDPC codes. ORD and ORDE methods when applied to (648, 540) code defined in 802.11n and (672, 588) code in 802.15.3c achieve the best performance. LOCK method perform very near to NMSA by changing process order of layers. For long length codes, proposed methods provide better performance gain in low SNR region, when the SNR is high performance gain is lost. Performance gain always improves as maximum iteration number is incremented. At large number of iterations ordered methods outperform traditional algorithms. So our results also confirm better performance is obtained with large maximum iteration number of 100 when compared to 50. No significant gain is obtained from DVB-S2 codes.

We have also presented parallel implementation of LDPC decoding using OpenMP. It can be leveraged to take advantage of the multicore DSP by parallelizing algorithms quickly. Our implementation has been tested on a multicore embedded DSP platform, TI's C6667 DSP. Multicore processing using OpenMP reduces processing times considerably. Our performance evaluation shows scalable speedup as more cores are included in multicore processing. When 8 cores are run simultaneously, the speedup reaches 6.4. Further speedup can be achieved by taking advantage of DSP architecture which itself provides several levels of parallelism like compiler specific pragmas or intrinsic SIMD instructions.

In future, proposed methods can be applied to other types of codes. Since simulation takes significant amount of time, methods to shorten simulation time can be searched. We have implemented parallel decoding of DVB-S2 code in flooding schedule. Other structured LDPC codes can be decoded in both layered and parallel manner so that parallelism can be exploited to accelerate simulation further.

8 RESOURCES

[1] R. G. Gallager, *Low-density parity-check codes*, *IRE Trans. Inf. Theory,* Vols. vol. IT-8, pp. 21-28, Jan 1962.

[2] A. G. a. P. T. C. Berrou, *Near Shannon limit error-correcting coding and decoding: Turbo codes*, 1993.

[3] D. MacKay & R. Neal, *Near shannon limit performance of low density parity check codes*, Electron. Lett., vol. 32, no. 18, p. 1645, Aug 1996.

[4] W. Ullah, *Two-way normalization of min-sum decoding algorithm for medium and short length LDPC codes*, in Networking and Mobile Computing (WiCOM), 2011.

[5] B.-Y. Chang, M. Ivkovic & L. Dolecek, *Computationally-Efficient Iterative Decoding for Storage System Design: Min-Sum Refined*, 2011.

[6] L. Fan, *Adaptive Normalized Min-Sum Algorithm for LDPC*, in IEEE, 2013.

[7] D. Levin, E. Sharon & S. Litsyn, *Lazy scheduling for LDPC decoding*, Communications Letters, pp. 70-72, 2007.

[8] M. Mansour & N. Shanbhag, *High-throughput LDPC decoders*, *I*EEE Tran. Very Large Scale Integration Systems, vol. 11, pp. 976-996, 2003.

[9] D. E. Hocevar, *A reduced complexity decoder architecture via layered decoding of LDPC codes*, in IEEE Workshop on Signal Processing Systems (SIPS), 2004.

[10] P. Radosavljevic, A. de Baynast & J. Cavallaro, *Optimized Message Passing Schedules for LDPC Decoding*, in Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, 2005.

[11] J. Z. a. M. P. C. Fossorier, *Shuffled Iterative Decoding*, Transaction Letters, vol. 53, no. 2, pp. 209-213, 2005.

[12] K. Guo, *A Parallel-Layered Belief-Propagation Decoder for Non-layered LDPC Codes*, Journal of Communications, 2010.

[13] E. Amador, *Very Large Scale Integration (VLSI-SoC)*, in Energy Efficiency of SISO Algorithms for Turbo-Decoding Message-Passing LDPC Decoders, 2009.

[14] A. Vila Casado, M. Griot & R. Wesel, *Informed Dynamic Scheduling for Belief-Propagation Decoding of LDPC Codes*, in IEEE International Conference on Communications, 2007.

[15] G. Han and X. Liu, *An efficient dynamic schedule for layered belief-propagation decoding of LDPC codes*, Communications Letters, pp. 950-952, 2009.

[16] X. L. Yi Gong, *Effective Informed Dynamic Scheduling for Belief Propagation Decoding of LDPC Codes*, Transactions Papers, vol. 59, no. 10, October 2011.

[17] S.-Y. Chung, G. D. Formey, T. J. Richardson & R. Urbanke, *On the design of low-density parity-check codes within 0.0045 dB of the Shannon limits*, IEEE Comm. Let., vol. 5, no. 2, pp. 58-60, 2001.

[18] Y. Jiang, *A Practical Guide to Error-control Coding Using Matlab*, Boston: Artech House, 2010.

[19] X. Ma, in *Selected Topics in Information and Coding Theory*, 2010, pp. 471-505.

[20] *Creonic IP Cores& System Solutions*, [Online]. Available: http://www.ldpc-

decoder.com. [Accessed 3 February 2015].

[21] R. M. Tanner, *A recursive approach to low complexity codes*, IEEE Trans. Information Theory, 1981.

[22] V. Savin, *Self-Corrected Min-Sum decoding of LDPC codes*, in IEEE International Symposium on Information Theory, 2008.

[23] C. Chapman, G. Jost and R. Van Der Pas, *Using OpenMP*, The MIT Press, 2008.

[24] C. Hu and D. Bell, *KeyStone Memory Architecture*, Texas Instruments, 2010.

[25] *OMP (OpenMP Runtime for SYS/BIOS) Users Guide*, Texas Instruments.

[26] BIOS MCSDK 2.0 User Guide, [Online]. Available: http://processors.wiki.ti.com/index.php/BIOS_MCSDK_2.0_User_Guide.

9 ÖZGEÇMİŞ

Adı Soyadı : Murat SEVER

Doğum Yeri : Kilis

Doğum Tarihi: 27.11.1976

Lisans : ODTÜ, Mühendislik Fakültesi, Elektrik-Elektronik Mühendisliği Bölümü (1998)

E-posta : murat-sever@live.com