Implementation of LDPC Decoder using AHIR Tool Chain

Submitted in partial fulfillment of the requirements

of the degree of

Master of Technology

by

Anurag Gupta

(*Roll no.* 153070050)

Supervisor:

Prof. Madhav P. Desai

Department of Electrical Engineering

Indian Institute of Technology Bombay

2017

Dedicated to my parents.

Dissertation Approval

This dissertation entitled **"Implementation of LDPC Decoder using AHIR Tool Chain "**, submitted by Anurag Gupta(Roll No. 153070050), is approved for the award of degree of Master of Technology in Electrical Engineering.

Examiner 1

____

Examiner 2

____

Supervisor

____

Chairman

____

Date: ...... June 2017

Place: ______________________

Declaration of Authorship

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Signature: ....

Anurag Gupta
153070050

Date: ...... June 2017

Abstract

The thesis investigates the performance of the Low-density parity-check (LDPC) codes for their use in storage systems using AHIR tool chain. AHIR is a open source tool- used for high level synthesis (HLS), developed at IIT Bombay.

Some decoding algorithms can achieve error-correction performance very close to the Shannon limit by using properties of low density parity check matrices. We have implemented Bit Flipping algorithm, Sum Product algorithm and Min Sum algorithm in C to decode the code block. The algorithms are written in C so that the description can be converted into hardware via high level synthesis (HLS) tool. The performance of Sum Product algorithm is very close to Shannon limit but implemented hardware is very complex and costly. Generally, the performance of Min Sum decode algorithm is relatively lesser than Sum Product algorithm and implemented hardware cost is also relatively lesser. Thus, we chose to implement Min Sum decode algorithm on a field-programmable gate array (FPGA) plateform. The implemented hardware is a serial min sum decoder. To parallelize the hardware we have performed partitioning of the parity check matrix. The results of partitioning different low density parity check matrix have promised a good level of parallelism in hardware. Thus, Min Sum decoding algorithm was modified to make decoding more efficient. This will result in a partial parallel decoder. The modified Min Sum algorithm incorporates parallel decoding with less complex hardware.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The error performance of the low density parity check codes approach to Shannon limit, which make low density parity check code the best known codes. An active research is going on to implement the low density parity check decoder for storage devices to achieve the bit error rate of the order of 10^{-16}. Thus, to make a low density parity check decoder with a optimum cost and performance trade off is a challenging task.

1.2 Organisation of the Report

In chapter 1, we have discussed the basics of error correction in a communication channel. In chapter 2, we have discussed how to generate different parity check matrices. In chapter 3, we have discussed different decoding algorithms in detail. In chapter 4, we have shown the results of the C level implementation of Min Sum decoding algorithm. In chapter 5, we have discussed the concept and procedure to partition a parity check matrix. Level of parallelism is taken as a metric and results are plotted and discussed. In chapter 6, we have deployed partitioning in the matrix and modified the decoding algorithm to parallelise the hardware. In chapter 7, the conversion of the algorithms from C to VHDL is discussed along with results of the implementation of the algorithm on FPGA.

1.3 Basics of Error Correction

The goal of communication is to transmit a message and receive it correctly even after noisy transmission through the channel. This is achieved by introducing redundancy in

the message at the transmitter side, called as encoding of message. The encoded message is called codeword. Then codeword is then transmitted through the noisy channel, which alters the codeword. By some error correcting algorithm the message is extracted back at receiver side, called decoding of the codeword. Thus, the error free transmission takes place by applying error correcting codes in communication system.
We have a k bit long message. To encode it we introduce m redundant bits (called parity bits) to form an $n(= m + k)$ bit long codeword. This category of codes are called (n,k) block codes.

1.3.1 Parity Check Matrix

The codeword must satisfy a group of conditions to ensure error free transmission or to indicate the error has taken place. If error occurs, the error can be corrected by applying some algorithm on those group of conditions . The group of conditions are called parity check equations.The matrix form of the condition is called parity check matrix.
Example: If a code block $y = [c_1\, c_2\, c_3\, c_4\, c_5\, c_6]$ has to satisfy following parity check equations.

$$c_1 \oplus c_2 \oplus c_4 = 0 \tag{1.1}$$

$$c_2 \oplus c_3 \oplus c_5 = 0 \tag{1.2}$$

$$c_1 \oplus c_2 \oplus c_3 \oplus c_6 = 0 \tag{1.3}$$

Then it's parity check matrix is as follows:

$$H = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 \end{bmatrix} \tag{1.4}$$

s.t. $Hy^T = 0$.

1.3.2 Encoding of Message

We can rewrite the above equations (1),(2),(3) as:

$$c_4 = c_1 \oplus c_2 \tag{1.5}$$

$$c_5 = c_2 \oplus c_3 \tag{1.6}$$

$$c_6 = c_1 \oplus c_2 \oplus c_3 \tag{1.7}$$

We can find parity check bits c_4, c_5, c_6 by message bits c_1, c_2, c_3. Thus we can encode message bits to find codeword.

Encoding is preferably done in matrix form, by manipulating parity check matrix to find a generator matrix. If parity check matrix can be written in the form $H = [A, I_{n-k}]$, where A is an $(n-k)$xk binary matrix and I_{n-k} is the identity matrix of order $(n-k)$. The generator matrix is then $G = [I_k, A^T]$. s.t $GH^T = 0$.

$$[c_1 c_2 ... c_6] = [c_1 c_2 c_3] \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \tag{1.8}$$

Thus, if m is message block containing message bits $[c_1 c_2 c_3]$, then codeword can be generated as $y = mG$.

1.3.3 Transmission of the Code Block

The code block is then transferred through a noisy communication channel that corrupts the code block. According to the behaviour of the noise that is added to the code block, various models of communication channels exists.

Binary Symmetric Channel

As the name suggests the channel is binary, it has two input symbols and two output symbols. The channel is symmetric because the probability of receiving 0 when 1 was send is same as probability of receiving 1 when 0 was sent. The error probability is also called cross over probability. The transition probability diagram is shown in figure 1.1 The cross over probability is denoted as p.

$$p(y = 0|x = 1) = p(y = 1|x = 0) = p \tag{1.9}$$

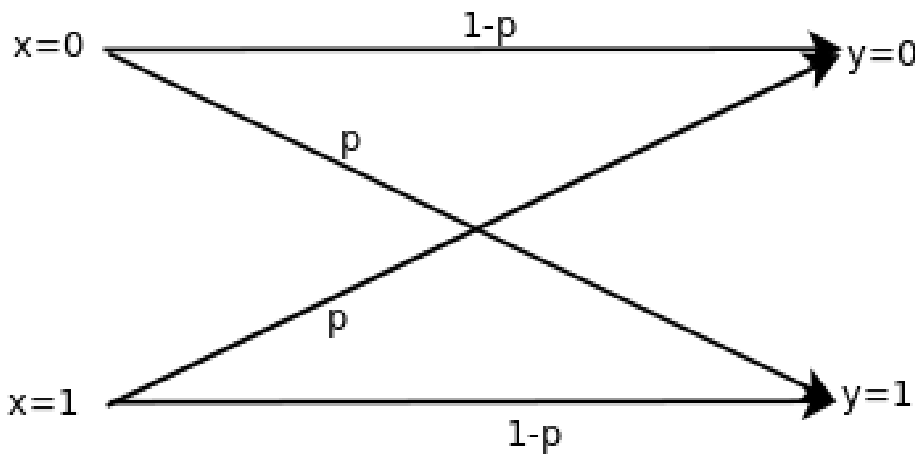

Figure 1.1: Binary Symmetric Channel

$$p(y = 1|x = 1) = p(y = 0|x = 0) = 1 - p \tag{1.10}$$

AWGN (A White Gaussian Noise Channel)

If a signal in a digital system is represented a continuous random variable as,

$$Y = X + Z \tag{1.11}$$

where X is the digital information carrier and Z is the noise component then we can define a Gaussian channel that has input X and output Y, with noise as a Gaussian random variable Z.
The pdf of a Gaussian random variable is expressed as follows:

$$f_y(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \frac{-(x-m)^2}{2\sigma^2} \tag{1.12}$$

where m is mean and sigma is standard deviation of the random variable Z.

1.3.4 Error Detection & Correction

If the codeword gets corrupted in the transmission then all the parity check equations will not get satisfied. Thus, we will get $Hy^T \neq 0$. The non-zero vector is called syndrome. That shows that received message is corrupted. But there is a certain limit to the number of bits upto that error can be detected and corrected. The limit is represented in term of hamming distance. Hamming distance between two codes is number of flipped bits between them. If d_{min} is minimum distance between codes then maximum number of bits upto which error can be correctly detected is $d_{min} - 1$ and maximum number of bits upto which error can be corrected is (t) = $[\frac{d_{min}-1}{2}]$ where [] denotes greatest integer function. More the number of redundant bits, more is the hamming distance. Thus, more error bits can be detected and corrected but the code rate is reduced. The correction is done by directly taking the received vector and comparing it to all the codewords and correcting it to the codeword having minimum distance to it. This is called maximum likelihood decoding. But if n is larger then this task becomes complex. LDPC maximum likelihood decoding, bit flipping decoding are other decoding schemes which reduce complexity of this task.

Chapter 2

Generation of different LDPC matrices

The Bit Error Rate of decoded code word depends upon properties of parity check matrix. A good parity check matrix should be sparse and it should have good girth.

Basic parameters to decide the formation of parity check matrix are as follows .

- **Random parity check matrix vs systematic parity check matrix:** Parity check matrix that has a specific method of filling $1s$ in matrix is called systematic parity check matrix, else if the position of $1s$ is random, the matrix is called random parity check matrix.

- **Regular parity check matrix vs irregular parity check matrix:** Regular parity check matrix has constant number of $1s$ in row and columns, whereas irregular parity check matrix has variable number of $1s$ in its rows and columns.
  If w_c is number of $1s$ in a column and w_r is number of $1s$ in a row then in a m$\times$ n regular parity check matrix

$$m * (w_r) = n * (w_c) \tag{2.1}$$

  If we take the fraction of columns of weight i by v_i and the fraction of rows of weight i by h_i then in a irregular parity check matrix

$$m * \sum_i (h_i * i) = n * \sum_i (v_i * i) \tag{2.2}$$

2.1 Gallager Parity Check Matrix

Gallager proposed parity check matrix is regular in nature[1]. A regular matrix has constant number of non-zero entries in a row and a column. These are represented as (n,w_c,w_r) codes. where,
w_c = Number of 1's in a column
w_r = Number of 1's in a row
n = Block length.
Method of construction:
o Divide rows in w_c sets with (m / w_c) rows in each set.
o All rows of first set of rows contain w_r consecutive once ordered from left to right.
o Every other set of rows is random column permutation of first set of rows.
Example of Gallager Matrix:
(n , w_c , w_r) = (12 , 3 , 4) ; m = 9

$$H = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ - & - & - & - & - & - & - & - & - & - & - & - \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ - & - & - & - & - & - & - & - & - & - & - & - \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

2.2 Quasi-Cyclic (QC) parity Check Matrix

QC matrix can be constructed by Sridhara Fuja Tanner (SFT)[3] method is discussed. Performance of quasi-cyclic codes is better for smaller block length, comparable for moderate block length with respect to random-regular codes.
Method of Construction of (j,k)-regular QC-LDPC code:

- Construct two sequences $\{s_1, s_2, ..., s_{j-1}\}$ and $\{t_1, t_2, ..., t_{k-1}\}$, whose elements are randomly selected from GF(p), where p is prime and p>2 , $s_i \neq s_x$ & $t_i \neq t_x$ if i

$\neq$x.

- Now, form a preliminary matrix Y with the elements of GF(p) as follows:

$$E = \begin{bmatrix} e_{0,0} & e_{0,1} & \cdots & e_{0,k-1} \\ e_{1,0} & e_{0,1} & \cdots & e_{0,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ e_{j-1,0} & e_{j-1,1} & \cdots & e_{j-1,k-1} \end{bmatrix} \tag{2.3}$$

where (i,j)th element of E is calculated by following quadratic congruential equation for a fix parameter $\kappa\epsilon\{1, 2, ..., p-1\}$ and $\nu_i, \nu_j\epsilon\{1, 2, ..., p-1\}$:

$$e_{i,j} = [\kappa(s_i + t_j)^2 + \nu_i + \nu_j] \tag{2.4}$$

- So the parity check matrix H is represented by jXk array of circulant permutation of identity matrix.

$$H = \begin{bmatrix} I(e_{0,0}) & I(e_{0,1}) & \cdots & I(e_{0,k-1}) \\ I(e_{1,0}) & I(e_{0,1}) & \cdots & I(e_{0,k-1}) \\ \vdots & \vdots & \ddots & \vdots \\ I(e_{j-1,0}) & I(e_{j-1,1}) & \cdots & I(e_{j-1,k-1}) \end{bmatrix}$$

Where I(x) is p×p identity matrix with row cyclically shifted right by x position.

2.3 MacKay Neal Parity Check Matrix

Mckey Neal proposed regular (n,w_c,w_r) construction of codes using random distribution of non-zero entries[2]. These codes have better performance for large block length compared to other codes. Method of construction:

- Start from the first column. Place w_c 1s in the column randomly and track number of 1s in a row.
- Repeat the process for other columns. Break only if at any point number of 1's in the row becomes greater than w_r.
- If break occurs then go back to some columns and repeat algorithm till all columns get filled.

Example:

n = 12, m = 9

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

MacKay Neal construction can be adapted to avoid cycles of length 4, called 4-cycles. Method to avoid 4-cycles[7]:

- First, generate a preliminary parity check matrix.
- Put 1s to parity check matrix in rows that don't have any 1s in them or that have only one 1. The places where 1s are to be added in those rows are selected randomly.
- Choose odd number of 1s to put in a column. If preliminary parity check matrix constructed has an even number of 1s in each column, problem may occur that will cause the rows to sum up to zero, and hence at least one check will become redundant.
- To remove situation that has a pair of columns both have all 1s in a particular pair of rows, that makes cycles of length four in graph, one of the 1s involved is moved randomly within its column.

Eliminating the cycles improves the girth of the graph and thus improves the convergence and make the iterative decoding faster.

Chapter 3

Decoding Algorithms

3.1 Message Passing Decoding

The algorithms used to decode LDPC codes are iterative in nature. In every iteration some information has to be passed through the edge of the bipartite graph (Tanner graph), representing corresponding parity check matrix. Thus these type of iterative algorithm are generally termed as message passing decoding[9].

Two type of message passing decoding algorithms are discussed. Bit flipping algorithm takes hard decision at the received information and then principally uses majority whereas belief propagation algorithm uses soft information or the probabilistic approach.

LDPC codes are often represented in graphical form by a Tanner graph. The Tanner graph consists of two sets of vertices. n vertices for the codeword bits (called bit nodes), and m vertices for the parity-check equations (called check nodes). An edge joins a bit node to a check node if that bit is included in the corresponding parity-check equation. Number of edges in the Tanner graph is equal to the number of ones in the parity-check matrix. Notion of Tanner graph was given by Tanner

Example:

$$H = \left[\begin{array}{c|cccccccc} & B1 & B2 & B3 & B4 & B5 & B6 & B7 & B8 \\ \hline c1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ c2 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ c3 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ c4 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

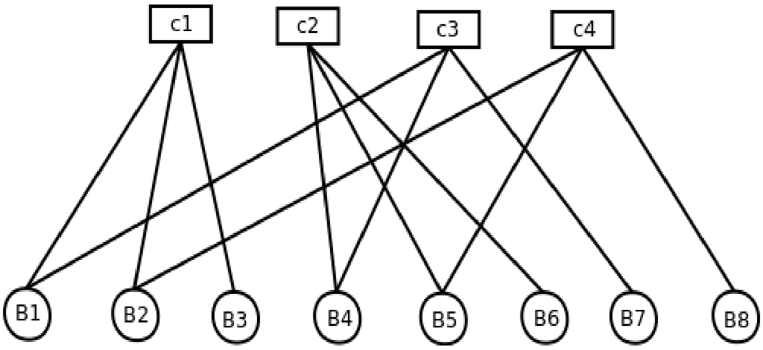

Figure 3.1: Tanner graph

3.1.1 Bit Flipping Decoding

A hard decision is made by the detector for the received bits before passing them to decoder. Then bits are placed on the bit nodes and passed through the edges of the Tanner graph. The check node determines that its parity-check equation is satisfied if the modulo-2 sum of the incoming bit values is zero else they pass the flipped value to the corresponding bit. If the majority of the reception by a bit node are different from its present value then bit node changes (flips) its current value. This process is repeated until all of the parity-check equations are satisfied, or the decoder gives up.[9]
As LDPC matrix is sparse it is unlikely to have same set of checks for a single bit. Still if several checks applied to single bit are incorrect then it is likely to be flipped.
Example:
If valid codeword for a Tanner graph is c = [0 0 1 0 1 1] (transmitted code);
Received code word is y = [1 0 1 0 1 1] (detected code);
The decoding algorithm proceeds as following:

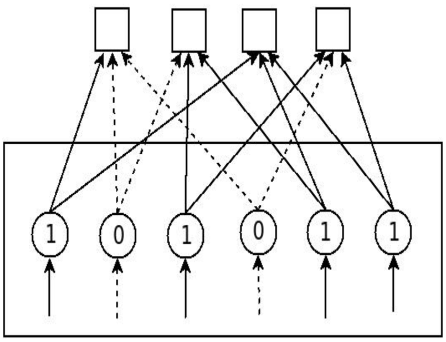

Figure 3.2: Initializing bit nodes

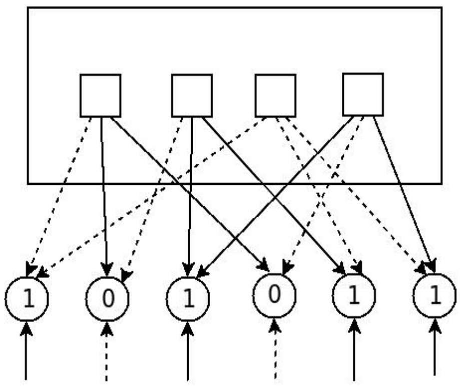

Figure 3.3: Performing computation

Step 1: The received code bits are assigned to bit nodes in the corresponding Tanner graph, as shown in figure 3.2.

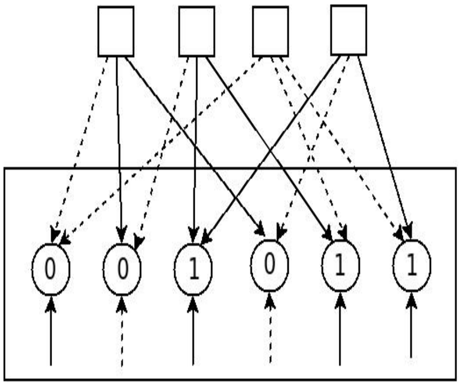

Figure 3.4: Updating bit node values

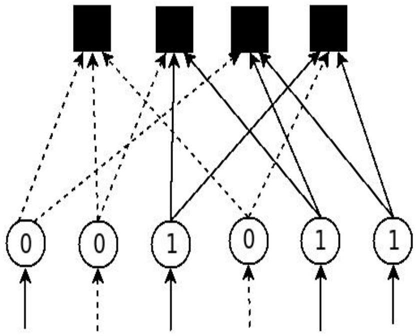

Figure 3.5: Testing for completion

Step 2: Modulo-2 sum is performed at the check nodes. And every bit node gets estimate by all the check nodes connected to it. Every check node send estimate to each bit node connected to it, by computing the result of modulo-2 sum of all the bit nodes connected to it, except the node to which estimate is being send. The computation is illustrated in fig 3.3

Step 3: According to the estimates send from check nodes, a bit node take decision by majority. Thus, either is retains its value or flips it, as shown in fig 3.4.

Step 4: After the iteration, the parity check conditions are checked to satisfy the code word. If code word satisfies all the conditions then decoding stops, else we have to iterate again.

3.1.2 Belief Propagation Decoding (Sum-Product Decoding)

It is soft decision algorithm. Bit-flipping decoding accepts an hard decision on the received bits as input, whereas the sum-product algorithm accepts the probability of each received bit as input. The input bit probabilities are called the a priori probabilities. The bit probabilities returned by the decoder are called the a posteriori probabilities. For sum-product decoding these probabilities are expressed as log-likelihood ratios (LLR).

$$L(x) = log\frac{p(x=0)}{p(x=1)} = log\frac{1-p(x=1)}{p(x=1)} \tag{3.1}$$

If p(x = 0) > p(x = 1) then L(x) is positive.

Log Likelihood Ratios represent probability as a single value rather than individual probability of being zero and one. LLR based representation has benefit when probabilities need to be multiplied LLR need only be added, reducing the implementation complexity.[9] The goal is to achieve maximum a posteriori probability for each bit. The extra information about bit i received from the parity-check j is called extrinsic information for bit i denoted by $E_{j,i}$. The probability ($P_{j,i}^{ext}$) that check j is satisfied when ith bit is one is equal to the probability of having odd number of 1s in the check j other than bit i.

$$P_{j,i}^{ext} = \frac{1}{2} - \frac{1}{2} \prod_{i' \in B_j,\ i' \neq i} (1 - 2P_{i'}^{int}) \tag{3.2}$$

$P_{i'}^{int}$ is a priori probability of ith bit to be 1. Thus,

$$E_{(j,i)} = LLR(P_{j,i}^{ext}) = log\left(\frac{1 - P_{j,i}^{ext}}{P_{j,i}^{ext}}\right) \tag{3.3}$$

$$E_{(j,i)} = log\left(\frac{\frac{1}{2} + \frac{1}{2}\prod_{i' \in B_j\ \ i' \neq i}(1 - 2P_{i'}^{int})}{\frac{1}{2} - \frac{1}{2}\prod_{i' \in B_j\ \ i' \neq i}(1 - 2P_{i'}^{int})}\right) \tag{3.4}$$

Using relationship:

$$tanh\left(\frac{1}{2}log\left(\frac{1-p}{p}\right)\right) = 1 - 2p \tag{3.5}$$

$$E_{(j,i)} = log\left(\frac{\frac{1}{2} + \frac{1}{2}\prod_{i' \in B_j\ \ i' \neq i} tanh(M_{j,i'}/2)}{\frac{1}{2} - \frac{1}{2}\prod_{i' \in B_j\ \ i' \neq i} tanh(M_{j,i'}/2)}\right) \tag{3.6}$$

where

$$M_{(j,i')} = LLR(P_{j,i'}^{int}) = log\left(\frac{1 - P_{j,i'}^{int}}{P_{j,i'}^{int}}\right) \tag{3.7}$$

Alternatively, using the relationship

$$2tan^{-1}(p) = log\left(\frac{1+p}{1-p}\right) \tag{3.8}$$

Thus extrinsic information from check j to bit i is:

$$E_{(j,i)} = 2tan^{-1}\left(\prod_{i' \in B_j,\ i' \neq i} tanh(M_{j,i'}/2)\right) \tag{3.9}$$

Total LLR passed to bit i is

$$L_i = LLR(P_i^{int}) = r_i + \sum_{j \in A_i} E_{j,i} \tag{3.10}$$

where r_i is input a priori for bit i. But, message sent again from bit nodes to check nodes should avoid the information which checks already have. Thus, $M_{j,i}$ is not exactly the extrinsic information, it excludes the message generated by the same check node.

$$M_{j,i} = \sum_{j' \in A_i j' \neq j} E_{j,i} + r_i \tag{3.11}$$

.

After every iteration hard decision is made on the LLR post priori. If code satisfies $Hc^T = 0$ then decoding stops else $M_{j,i}$ is found and next iteration is performed.

3.1.3 Min Sum Decoding

The min sum decode algorithm is simplification in the sum product algorithm.
For BPSK modulation transmitted $0's$ are represented as $-1s$ and transmitted $1s$ are represented as $1s$.
The probability that bit 1 is received

$$f_y(y|f = -1) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \frac{-(y+1)^2}{2\sigma^2} \tag{3.12}$$

The probability that bit 0 is received

$$f_y(y|f = 1) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp \frac{-(y-1)^2}{2\sigma^2} \tag{3.13}$$

Thus getting LLR as:

$$LLR = \log \frac{f_y(y|f = -1)}{f_y(y|f = 1)} = \frac{-2y}{\sigma^2} \tag{3.14}$$

A priori information on the bit node side is expressed in term of LLR as:

$$aPriori[I] = -4 * C[I] * R * \frac{Eb}{No} \tag{3.15}$$

where C[I] = i^{th} code block, R = code rate and $\frac{Eb}{No}$ = signal to noise power ratio

Messages are the information propagating from bit nodes to check nodes. These are initialized to a priori of their respective bit node.

$$message[I][J] = aPriori[I] \tag{3.16}$$

Extrinsic information of a bit node is calculated min sum of all the messages connected to that particular check node.

$$|E_{(j,i)}| = Min_{i' \in B_j \ \ i' \neq i} |M_{j,i'}| \tag{3.17}$$

$$sign(E_{(j,i)}) = \prod_{i' \in B_j \ \ i' \neq i} sign(M_{j,i'}) \tag{3.18}$$

A posteriori probabilities are the output bit probabilities. These are used to modify the code block after every iteration.

$$aPosteriori[I] = \sum_{j \in A_i} E_{j,i} + aPriori[I] \tag{3.19}$$

Then hard decision is taken on the a posteriori information, that represent the decoded code block. If decoded code block satisfies $c * H^T = 0$, then decoding stops. Else messages are updated and transmitted back to start the next iteration of decoding.

$$message_{(j,i)} = aPosteriori[i] - E_{(j,i)} \tag{3.20}$$

Chapter 4

C Level Implementation & Verification

We have implemented *bit flipping algorithm*, *sum product algorithm* and *min sum algorithm* in C.
The decoding algorithms are so implemented that it can be used for any arbitrary rate and parity check matrix. The software takes parity check matrix file, code block file as input arguments and after decoding it writes decoded block in a file. The software can also compute time to decode and accuracy of decoding according to additional input argument given.

After developing the software for basic LDPC decoding algorithms in C we decided to convert min sum decode algorithm from C level description to hardware. This is done by Ahir tool chain, developed at IIT-Bomabay under the supervision of Prof. Madhav P. Desai. Firstly, we concentrated on testing algorithmic accuracy of min sum decode algorithm. The reason to verify the algorithm so regressively in C level is that it gives us confidence to proceed for the hardware design.

4.1 Min Sum Decode Algorithm

The key specification of the algorithm and notations on the tables are as follows :

- Min sum algorithm is implemented for Gaussian channel.
- Quasi-cyclic matrix of block size(n=) 4K, 8K and 12K are formed using Sridhara-Fuja-Tanner algorithm.

- Five different code rates(R=) 0.75, 0.80, 0.85, 0.90 and 0.95 are taken.
- Raw input bit error rate(BER(IN)) is between 10^{-2} to 10^{-3}, converted in form of Eb/No(db) to express input SNR in db.
- BER(OUT) : Output block error rate.
- CDB : Number of correctly decoded blocks.
- Itr : Average number of iterations per block.

4.1.1 Results on Quasi Cyclic Matrix

Block Error Rate

We have sent code blocks continuously and noted when the first code block gets incorrectly decoded. Maximum number of sent blocks is 1 million. If all 1 million blocks gets correctly decoded then $'-'$ is shown in table.

Table 4.1: Table for block error estimates

n≃	BER(In)≃	R=0.75	R=0.80	R=0.85	R=0.9	R=0.95
4K	1.0×10^{-2}	-	-	2.677×10^3	1.7819×10^4	1
	0.5×10^{-3}	-	-	2.4944×10^4	$1.65511x10^5$	1.79×10^2
	1.0×10^{-3}	-	-	5.47550×10^5	4.89654×10^5	3.328×10^3
8K	1.0×10^{-2}	-	-	2.3817×10^4	1.16847×10^5	1
	0.5×10^{-3}	-	-	6.9491×10^4	1.72263×10^5	1.001×10^3
	1.0×10^{-3}	-	-	9.16505×10^5	6.28939×10^5	9.338×10^3
12K	1.0×10^{-2}	-	-	9.705×10^3	5.37754×10^5	1
	0.5×10^{-3}	-	-	5.6400×10^4	-	1.318×10^3
	1.0×10^{-3}	-	-	-	-	1.6920×10^4

Error threshold and number of iterations required

We have tabulated results for the test case when we send 100 code blocks and noted BER and number of iterations to decode. The testing focus to get error threshold of the algorithm. Error threshold is the maximum input bit error rate that can be corrected by the decoder. We performed testing by varying input BER between 10^{-2} to 10^{-3}. To get error threshold we check for the input BER at which no block gets correctly decoded when sending 100 code blocks are sent.

Table 4.2: Table for error threshold estimate at code rate = 0.75

n	BER(In)	Eb/No(db)	BER(OUT)	CDB	Itr
4112	3.3×10^{-2}	3.5	0	100	5
	1.0×10^{-2}	5.5	0	100	2
	4.8×10^{-3}	6.5	0	100	1
	1.8×10^{-3}	7.5	0	100	1
8180	2.9×10^{-2}	3.8	0	100	5
	1.0×10^{-2}	5.5	0	100	2
	4.8×10^{-3}	6.5	0	100	1
	1.8×10^{-3}	7.5	0	100	1
12304	3.4×10^{-2}	3.4	0	100	6
	1.0×10^{-2}	5.5	0	100	2
	4.8×10^{-3}	6.5	0	100	1
	1.8×10^{-3}	7.5	0	100	1

Table 4.3: Table for error threshold estimate at code rate = 0.95

n	BER(IN)	Eb/No(db)	BER	CDB	Itr(/25)
4260	1.1×10^{-2}	4.5	6.8×10^{-3}	23	-
	4.7×10^{-3}	5.5	0	100	3
	1.6×10^{-3}	6.5	0	100	2
8220	1.1×10^{-2}	4.5	7.5×10^{-3}	8	-
	4.7×10^{-3}	5.5	2.8×10^{-7}	99	-
	1.6×10^{-3}	6.5	0	100	2
12660	1.1×10^{-2}	4.5	7.2×10^{-3}	4	-
	4.7×10^{-3}	5.5	0	100	4
	1.6×10^{-3}	6.5	0	100	2

Table 4.4: Table for error threshold estimate at code rate = 0.80

n	BER(In)	Eb/No(db)	BER	CDB	Itr(/25)
4075	2.2×10^{-2}	4.1	0	100	4
	0.8×10^{-2}	5.5	0	100	2
	3.8×10^{-3}	6.5	0	100	1
	1.5×10^{-3}	7.5	0	100	1
8275	2.4×10^{-2}	3.9	0	100	5
	0.8×10^{-2}	5.5	0	100	2
	3.8×10^{-3}	6.5	0	100	1
	1.5×10^{-3}	7.5	0	100	1
12275	2.5×10^{-2}	3.8	0	100	6
	0.8×10^{-2}	5.5	0	100	2
	3.8×10^{-3}	6.5	0	100	1
	1.5×10^{-3}	7.5	0	100	1

Table 4.5: Table for error threshold estimate at code rate = 0.85

n	BER(IN)	Eb/No(db)	BER	CDB	Itr(/25)
4220	1.9×10^{-2}	4	0	100	6
	1.0×10^{-2}	5	0	100	3
	4.5×10^{-3}	6	0	100	2
	1.7×10^{-3}	7	0	100	1
8180	1.9×10^{-2}	4	0	100	6
	1.0×10^{-2}	5	0	100	3
	4.5×10^{-3}	6	0	100	2
	1.7×10^{-3}	7	0	100	1
12260	1.9×10^{-2}	4	0	100	6
	1.0×10^{-2}	5	0	100	3
	4.5×10^{-3}	6	0	100	2
	1.7×10^{-3}	7	0	100	1

Table 4.6: Table for error threshold estimate at code rate = 0.90

n	BER(IN)	Eb/No(db)	BER	CDB	Itr(/25)
4120	1.1×10^{-2}	4.7	0	100	4
	0.9×10^{-2}	5	0	100	3
	3.6×10^{-3}	6	0	100	2
	1.3×10^{-3}	7	0	100	1
8440	1.2×10^{-2}	4.5	0	100	5
	0.9×10^{-2}	5	0	100	3
	3.6×10^{-3}	6	0	100	2
	1.3×10^{-3}	7	0	100	1
12280	1.2×10^{-2}	4.5	0	100	5
	0.9×10^{-2}	5	0	100	3
	3.6×10^{-3}	6	0	100	2
	1.3×10^{-3}	7	0	100	1

Table 4.7: Table for error threshold estimate at code rate = 0.95

n	BER(IN)	Eb/No(db)	BER	CDB	Itr(/25)
4260	1.1×10^{-2}	4.5	6.8×10^{-3}	23	-
	4.7×10^{-3}	5.5	0	100	3
	1.6×10^{-3}	6.5	0	100	2
8220	1.1×10^{-2}	4.5	7.5×10^{-3}	8	-
	4.7×10^{-3}	5.5	2.8×10^{-7}	99	-
	1.6×10^{-3}	6.5	0	100	2
12660	1.1×10^{-2}	4.5	7.2×10^{-3}	4	-
	4.7×10^{-3}	5.5	0	100	4
	1.6×10^{-3}	6.5	0	100	2

4.1.2 Results on Random Matrices

Block Error Rate

We have sent code blocks continuously and noted when the first code block gets incorrectly decoded. Maximum number of sent blocks is 1 million. If all 1 million blocks get correctly decoded then $'-'$ is shown in table.

Table 4.8: Table for block error estimates

n≃	BER(In)≃	R=0.75	R=0.8	R=0.85	R=0.9	R=0.95
4K	1.0×10^{-2}	$1.2799{\times}10^4$	$2.0754{\times}10^4$	$3.39{\times}10^2$	$1.259{\times}10^3$	NA
	0.5×10^{-3}	$5.53727{\times}10^5$	$1.72781{\times}10^5$	$6.6700{\times}10^4$	$1.65511{\times}10^5$	NA
	1.0×10^{-3}	-	$6.24436{\times}10^5$	$3.45503{\times}10^5$	$1.19008{\times}10^5$	NA
8K	1.0×10^{-2}	$1.92476{\times}10^5$	$8.3898{\times}10^4$	$5.193{\times}10^3$	$5.947{\times}10^3$	NA
	0.5×10^{-3}	$3.21027{\times}10^5$	$4.6092{\times}10^4$	$3.7952{\times}10^4$	$1.1389{\times}10^4$	NA
	1.0×10^{-3}	-	-	-	-	NA
12K	1.0×10^{-2}	$2.20022{\times}10^5$	$1.57371{\times}10^5$	$1.2894{\times}10^4$	$1.2626{\times}10^4$	1.1
	0.5×10^{-3}	$2.17452{\times}10^5$	$9.0158{\times}10^4$	$1.56487{\times}10^5$	$5.4866{\times}10^4$	$1.034{\times}10^4$
	1.0×10^{-3}	-	-	-	-	$1.4759{\times}10^4$

For rate = 0.95 & n = 4K,8k - we were not able to generate cycle-4 free parity check matrix.

Chapter 5

Partitioning of Tanner Graph

5.1 Introduction

The literature survey about hardware implementation of the LDPC decoder concludes three different implementation strategies.The serial decoder is the simplest decoder. Hardware cost and complexity is very less. It consists of single check node, single bit node and memory. The bits are first passed one by one from bit side to check side and generated checks are stored in memory, then checks are passed from check side to bit side one at a time to update the bit nodes. The main drawback is this implementation is too slow [4]. The second approach is implementing a fully parallel algorithm. This is a direct implemention of the Tanner graph in hardware [5]. This increases the decoding speed, but hardware cost and implementation complexity becomes too high. Another approach is midway between serial and parallel, called partial parallel implementation. This implementation is more flexible as trade off between speed and cost can be done. In partial parallel implementation bit nodes and check nodes are divided into several partitions. All these partitions works in parallel, but within one partition the information is transferred serially. Larger the number of partitions, faster the decoder as it acts more like parallel decoder. Fewer the partitions, simpler the decoder as it acts more like serial implementation of the decoder. Thus, we can trade off between speed and cost. Further, a reconfigurable interconnection network has to be designed to change the connection from bit to check side for different iterations in partial parallel decoder. The multistage interconnection network provides cost efficient parallel processing. Non-blocking and rearrangable networks are preferred. In blocking network connection is not always possible, whereas non-blocking network always provides a path between input to output. A rearrangable network can always provide a path, but path has to be rearranged. Lee and Ryu have used Benes network in their partial parallel LDPC decoder [5]. Benes network is a rearrangable network.

5.2 Partitioning

To use the partial parallel approach we propose to iteratively decompose the Tanner graph into sets of bits and sets of checks. In one iteration one set of bits is transferred towards one set of checks. It will be preferred that maximum number of the computational bits required for a particular check set are in same bit set so that computation can be completed for as many check nodes as possible. If a check set requires bits from many bit sets, then the number of iterations required to complete the computation of that set will be equal to the number of bit sets required to perform all the computation. Thus, if many check sets require bits from a large number of bit sets, then number of iterations increases and decoding time will become worse. Thus, an efficient partitioning of bit and check side of the Tanner graph is required.
The objective is to partition the bipartite graph corresponding to parity check matrix into two subsets in both bit side and check side such that most of the edges of a set of bit side relates to a particular set of check side. We call these edges as 'Through edges'. The remaining edges of the same set on the bit side that goes to different set in check side are called 'Transverse edges'. The average ratio of total number of through edges to total number of edges is the performance index for parallelization.
Suppose we have bipartite graph G, bits are B = { b_1 , b_2 ,b_3 ,...,b_n } and checks are C={ c_1,c_2 ,c_3,...,c_m } and the edge between bit node b_i and check node c_j is represented as ei, j. Now we have to partition G into four subsets B_1 ,B_2 ,C_1 and C_2 such that

- Size of setB_1 and set B_2 should be nearly equal as well as size ofC_1 $andC_2$ should be nearly equal.
- There should be a very small number of edges between B_1-$C_2$and B_2-C_t1so that the number of cross edges are minimised.

5.3 Procedure

To approach this we initially find a weighted check matrix from the parity check matrix. The weighted check matrix is an incidence matrix for the check node graph. A check node graph is a weighted graph. The weight between $checknode_i$ and $checknode_j$ is the

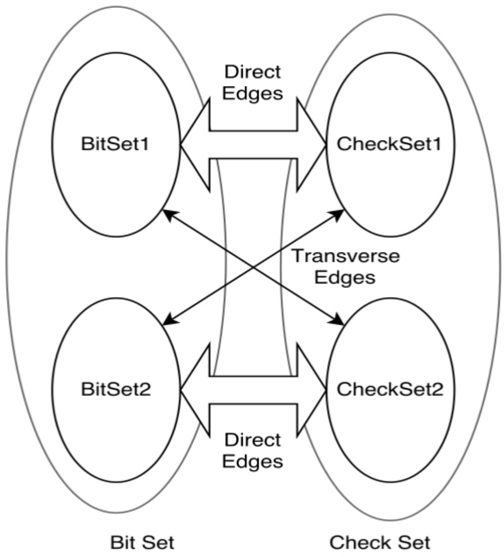

Figure 5.1: Partitioning a bipartite graph

number of common bit nodes performing check on both $checknode_i$ and $checknode_j$. After forming the weighted check graph we partitioned it into two equal parts using METIS [6]. After partitioning the check set into two equal sets with minimum weight cut, we need to partition the bit set. To partition the bit set we take into account the number of checks associated with a bit. If number of checks associated with a bit are more in first check set then it goes to first check set else it goes to second check set. As all matrices are sparse, the number of bits divided in two sets comes out nearly equal. To make them exactly equal we force some bits to go to other set after partitioning. Then the average ratio of parallelized edges to total edges and transverse edges to total edges is calculated as a performance index of parallelization of the decoder. Figure 5.2 shows the block diagram for the process of partitioning the bipartite graph corresponding to a low density parity check matrix.

- **Generation of Parity Check Matrix**
- **Generation of Check Node Incident Matrix**
  The weight between two check nodes is the number of common bit nodes performing exor operation on both check nodes. Taking this into account, we followed following algorithm to convert a parity check matrix to a check node incidence matrix.
- **Bisection of Check Node Graph**

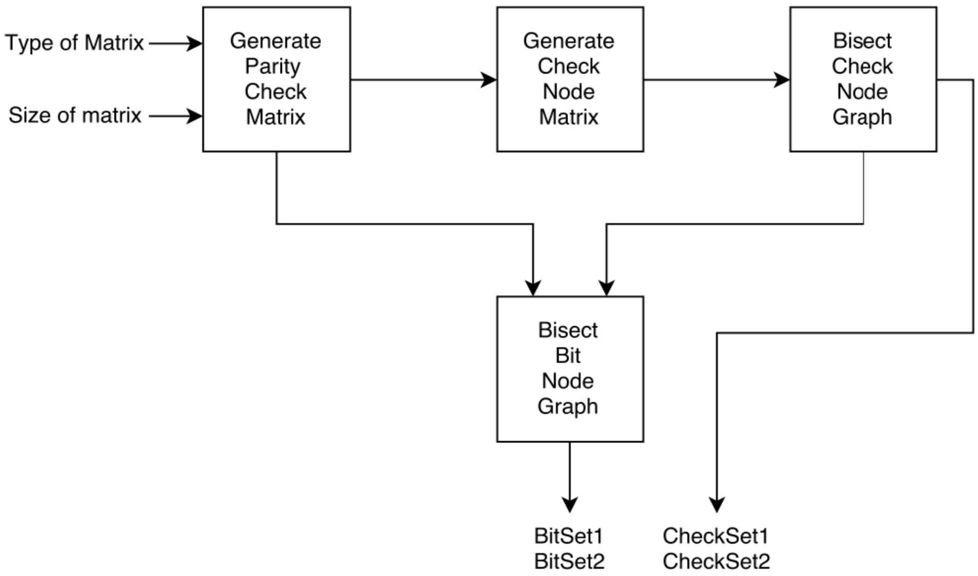

Figure 5.2: Block Diagram of process flow

Partitioning of check node graph is done using METIS [6]. "METIS is a set of serial programs for partitioning graphs, partitioning finite element meshes, and producing fill reducing ordering for sparse matrices. The algorithms implemented in METIS are based on the multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning schemes"[6]. First, the incidence matrix is converted into a corresponding graph format that can be given as input to METIS. At output we get two partitions of equal size with the minimum weight cut.

- **Bisection of Bit Node Graph**

  After bisection of the check set we get two sets C_1 and C_2 . Now, we have to divide a partition of bit set into B_1 and B_2 . A bit node corresponds to set B_1 if the number of edges between that bit node and set C_1 are more than the number of edges between bit node and set C_2 . As the check set is bisected by keeping in mind that the more the number of checks relating to a bit are in same set and the matrix is sparse thus we get nearly equal bits in set B_1 and set B_2. As this bisection has to be further iterated for partitioning, we force the bisection to be equal. Forcing the bisection to be equal increases the transverse edges.

5.4 Results of partitioning applied to matrices

Simulation for partitioning are performed in MATLAB environment. Figure 5.3 shows the performance index for Gallager matrix when we vary the order of the parity check matrix from 1,000 to 10,000. The performance index comes 0.78 for Gallager matrix. Similarly, by varying the order of the parity check matrix from 1,000 to 10,000 we get performance index of MacKay Neal matrix as 0.88, depicted in Figure 5.4. Quasi Cyclic matrix can be formed only for a special set of numbers. In simulation, the order of matrix taken are 155, 305, 905 and 11555, and performance index is above 0.88 in all cases, as depicted in Figure 5.5.

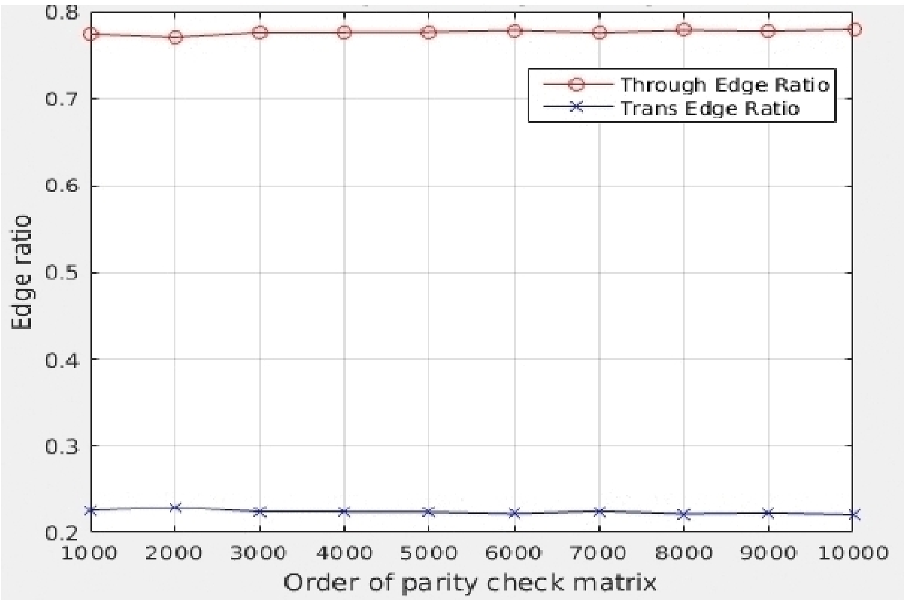

Figure 5.3: Performance of Gallager matrix

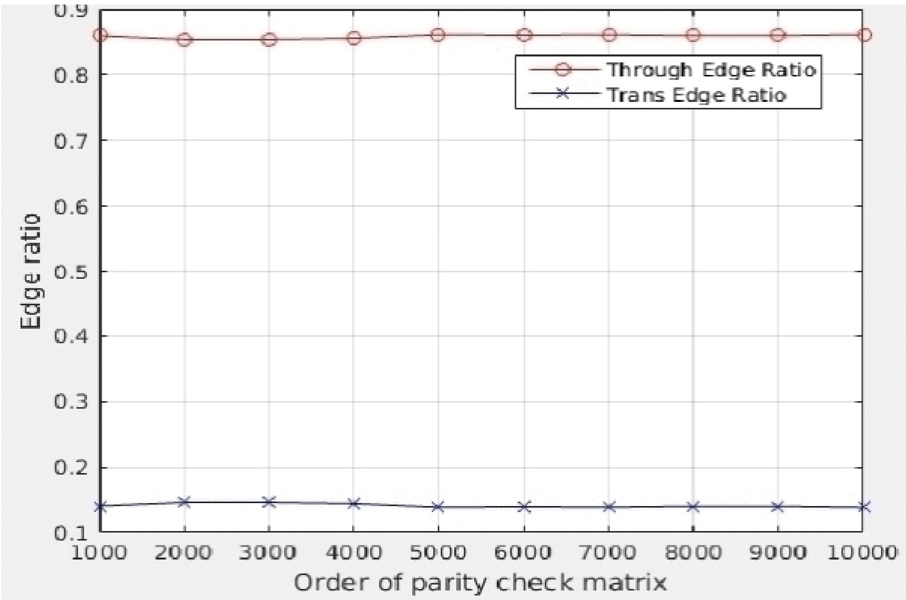

Figure 5.4: Performance of MacKay Neal matrix

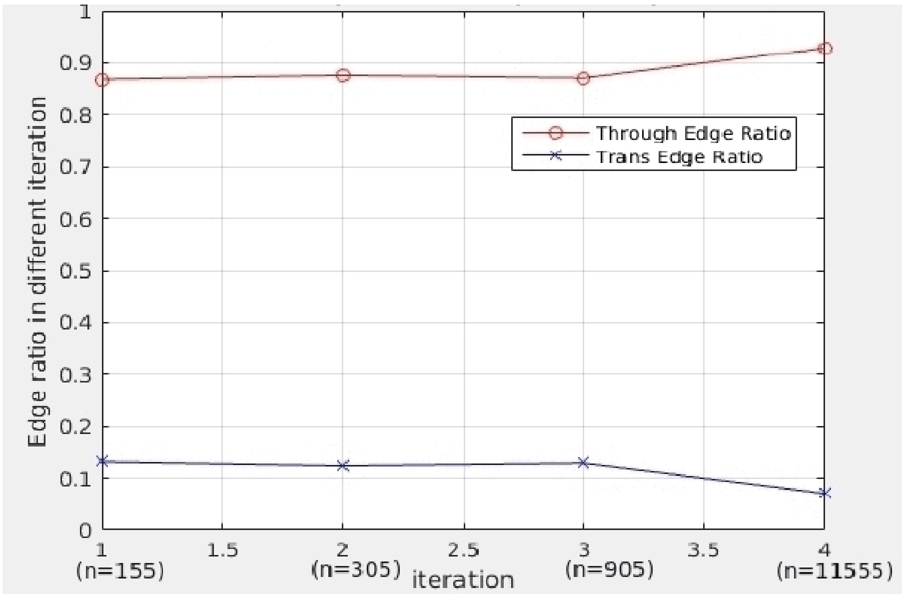

Figure 5.5: Performance of quasi-cyclic matrix

Chapter 6

Modification in Min Sum Algorithm using partitioning

The idea is to modify the min sum algorithm such that we can deploy it in a partitioned matrix iteratively to decode a code block. To state the modification we first explain min sum algorithm by a example and then we show the modification in it:

6.1 Example explaining min sum decode

The parity check matrix chosen to decode the code block is as following.

$$\left[\begin{array}{c|cccccccc} & c1 & c2 & c3 & c4 & c5 & c6 & c7 & c8 \\ \hline r1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ r2 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ r3 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ r4 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

The Tanner graph corresponding to above parity check matrix is depicted in figure 6.1.

The code block received at the receiver is represented by C. The SNR at the input of receiver is represented as $\frac{E_b}{No}$. The code is represented as R.
Step 1: We calculate the a priori probabilities at the receiver end. A priori probability of a particular code bit is the log likelyhood ratio of the code bit.

$$aPriori[I] = -4 * C[I] * R * \frac{Eb}{No}$$

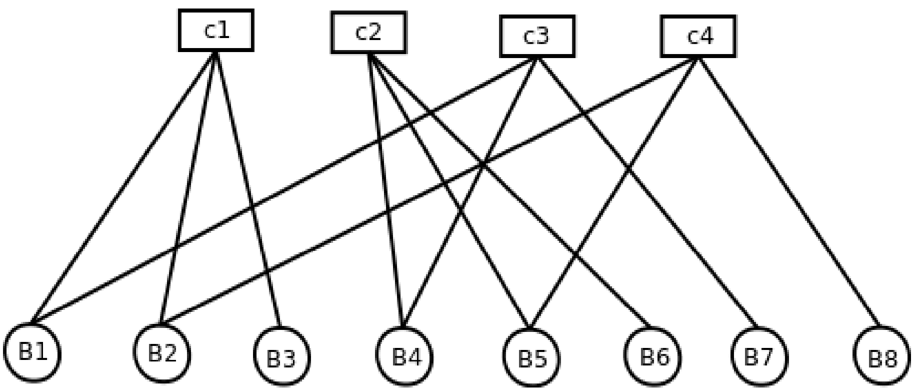

Figure 6.1: Tanner Graph

Assuming we get the a priori probability = { -3.2 , 2.8 , -3.6 , 2.8 , 2 , -6 , - 9.6 , -4.8 }

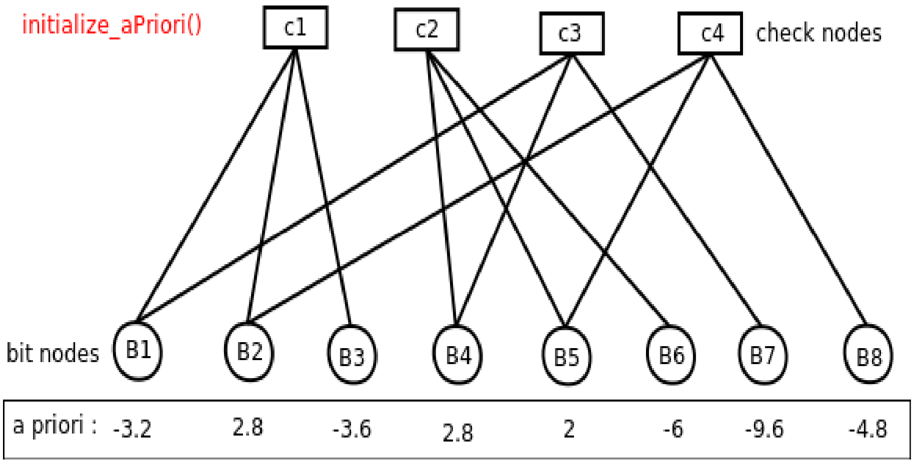

Figure 6.2: Initializing a priori probabilities

Step 2: We calculate messages transferred from bit nodes to check nodes through the edges of Tanner graph as depicted in figure 6.3

$$message[I][J] = aPriori[I]$$

Step 3: We calculate extrinsic information at the check nodes and transfer the information back to the bit nodes through edges as depicted in figure 6.4

$$|E_{(j,i)}| = Min_{i' \in B_j \ i' \neq i} |M_{j,i'}|$$

$$sign(E_{(j,i)}) = \prod_{i' \in B_j \ i' \neq i} sign(M_{j,i'})$$

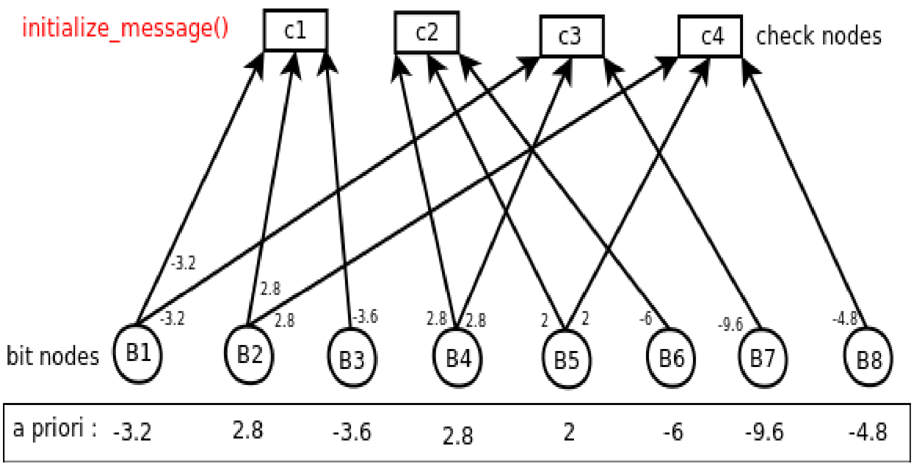

Figure 6.3: Initializing messages

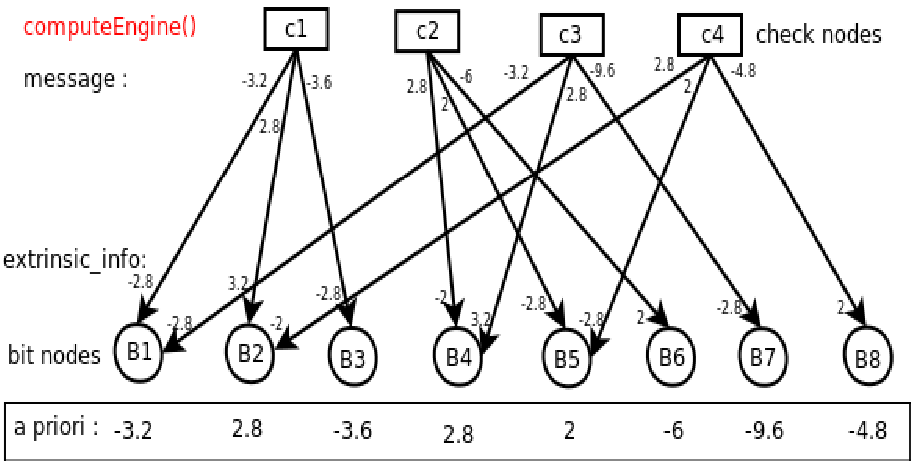

Figure 6.4: Computing extrinsic information

Step 4: After computation of extrinsic informations we calculate a posteriori probabilities at each bit nodes as depicted in figure 6.5

Step 5: A posteriori probabilities are the estimate of the code bits. Thus we take hard decision on the a posteriori probabilities. And this new estimate of code block is compared to the present code block. If both are same then decoding stops.

Step 6: Else, we modify the code bits and messages and start the next iteration to decode the code block. The messages are updated as shown is figure 6.6

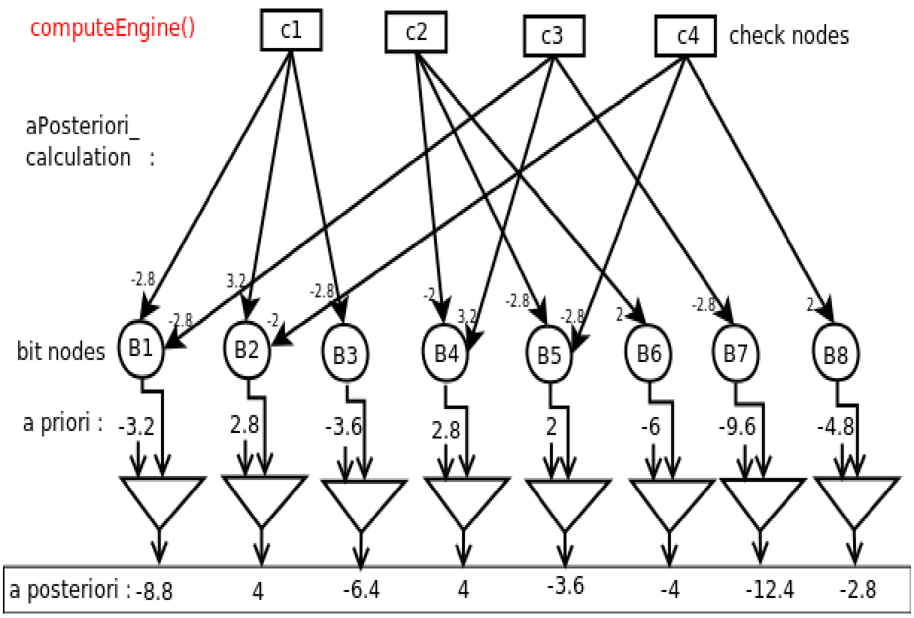

Figure 6.5: Calculating a posteriori probabilities

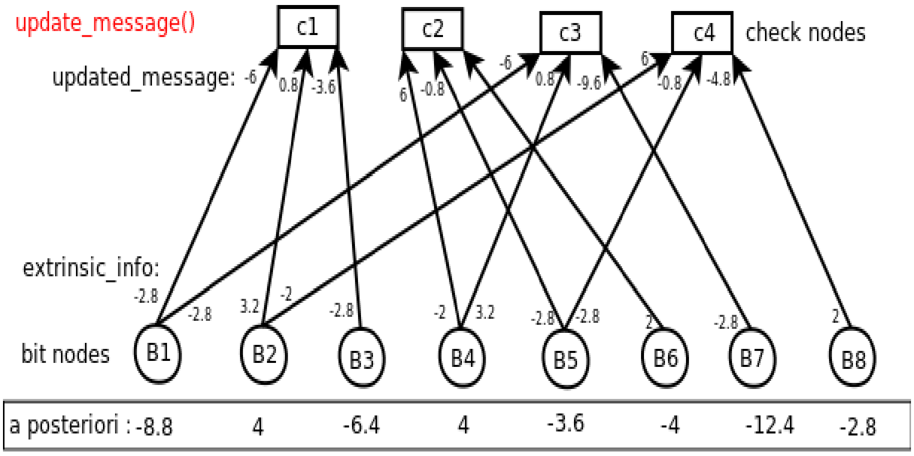

Figure 6.6: Updating the messages

6.2 Example explaining modifications on min sum decode algorithm for a partitioned matrix

As we have partitioned the matrix the parity check matrix looks as follows.

$$H = \left[\begin{array}{c|c} H11 & H12 \\ \hline H21 & H22 \end{array}\right]$$

$$\left[\begin{array}{c|cccccccc} & c1 & c2 & c3 & c4 & c5 & c6 & c7 & c8 \\ \hline r1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ r2 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ r3 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ r4 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{array}\right] \Rightarrow \left[\begin{array}{c|cccc|cccc} & c4 & c6 & c7 & c1 & c2 & c3 & c8 & c5 \\ \hline r2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ r3 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ \hline r1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ r4 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{array}\right]$$

Step 1: Assuming we get the a priori probabilities = { 5.59 , -11.99 , -19.19 , -6.39 , 5.59 , -7.1 , - 9.59 , 3.99 }. The a priori initialization is shown in figure 6.7

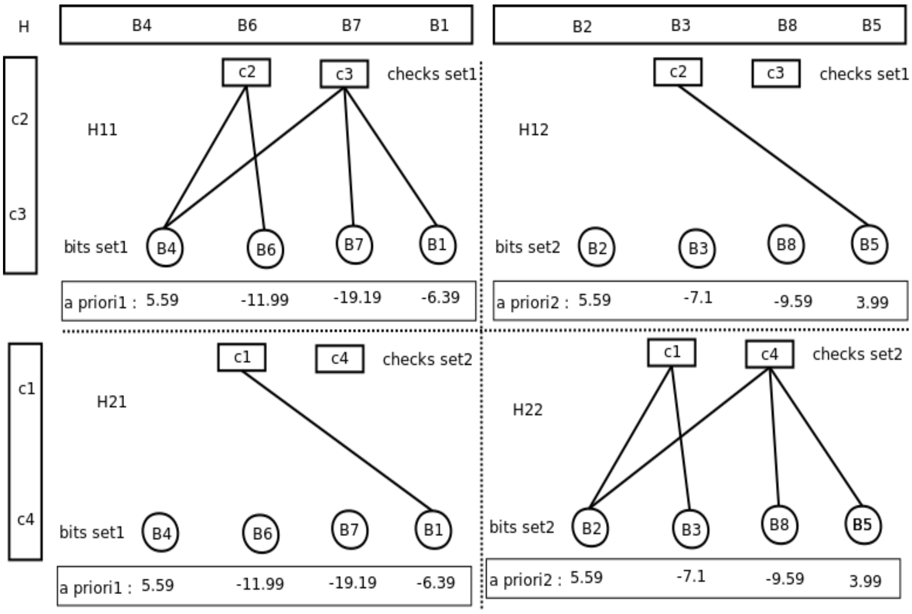

Figure 6.7: Initializing a priori probabilities on partitioned Tanner graphs

Step 2: The message initialization is done similarly and is depicted in figure 6.8

Step 3: The computation of extrinsic information is broken into two parts. First we calculate partial extrinsic information at each matrix separately. At the same time we compute the transverse information required to calculate complete information from one check matrix to other check matrix, as depicted in figure 6.9

Step 4: Then extrinsic information computation is completed by the transverse information as depicted in figure 6.10

Step 5: A posteriori probabilities are calculated similarly and the following steps of the algorithms remain the same.

6.3 Modified Min Sum Algorithm

By partitioning the matrix we can start computation in each matrix in parallel. This gives the hopes to parallelize the system and reduce the overall computation time. But

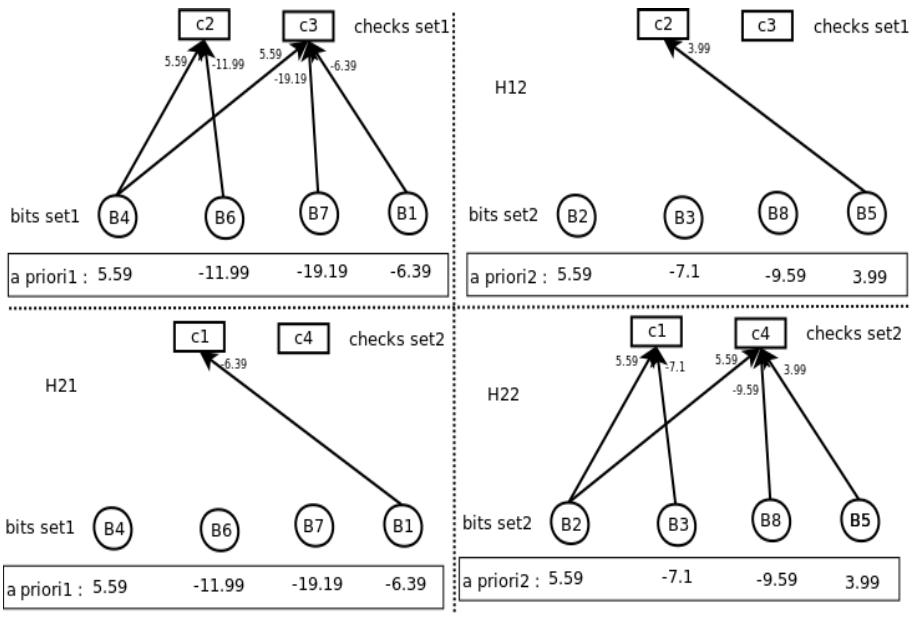

Figure 6.8: Initializing messages on partitioned Tanner graphs

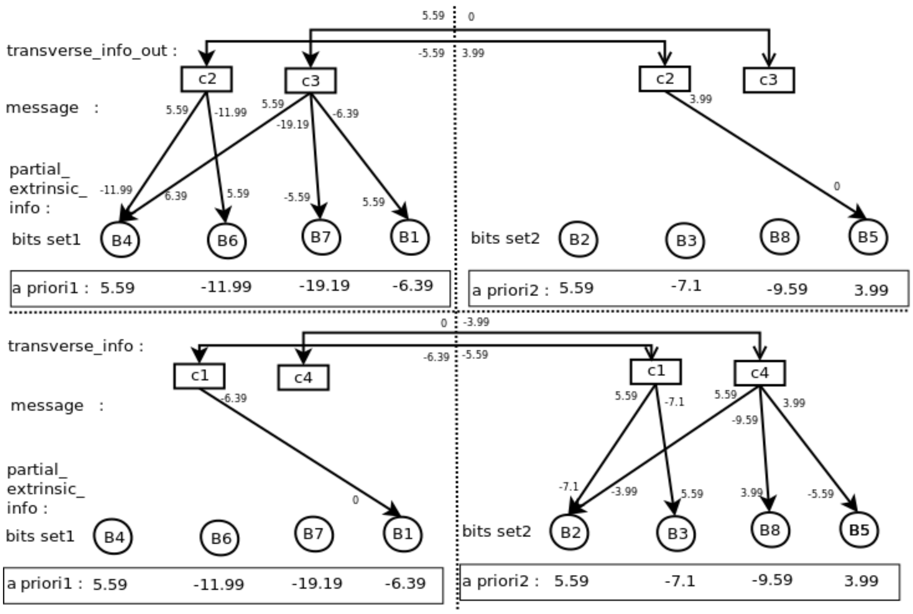

Figure 6.9: Computing partial extrinsic information & Transverse information

the problem is that the computation on the partitioned matrix is not totally independent of the other matrices. Thus, the goal is to reduce the overall dependency of the computation in one partition of the matrix with the other partition of the matrix. The results after partitioning are shown in chapter 6 .The results are promising to get the desired behaviour.

The modification is essentially breaking the computation of extrinsic informations into multiple parts. The example explained that for a two way partition we have to break the

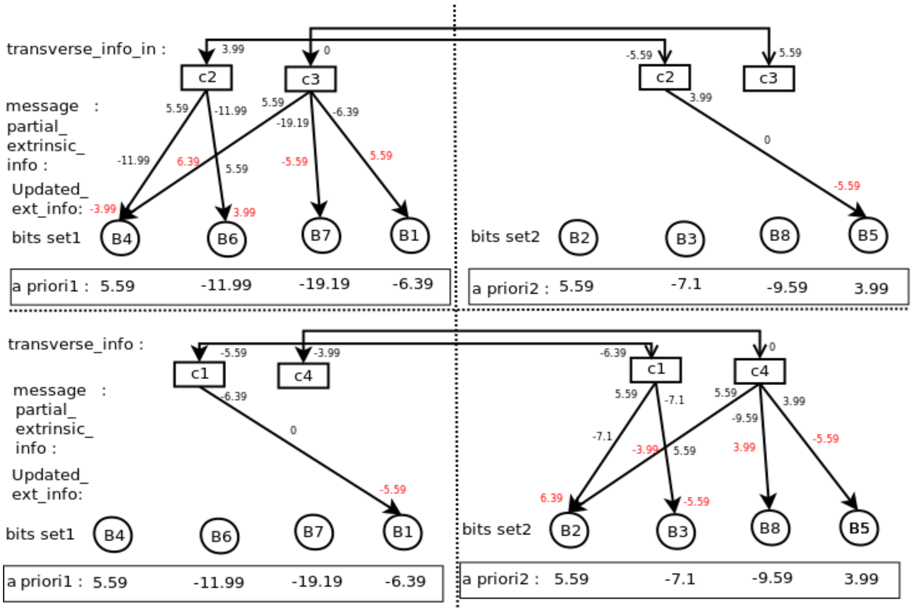

Figure 6.10: Computing extrinsic information using Transverse information

computation in two stages. Similarly for a n-way partition we can break the computation in n stages. The trade off lies between number of stages and the number of computation engines.

For a two way partitioned matrix, the extrinsic information calculation is broken into two stages as following.

Stage 1: Partial extrinsic informations are calculated as for every matrix as following :

$$|Ep_{(j,i)}| = Min_{i' \in B_j \ i' \neq i} |M_{j,i'}| \tag{6.1}$$

$$sign(Ep_{(j,i)}) = \prod_{i' \in B_j \ i' \neq i} sign(M_{j,i'}) \tag{6.2}$$

And the transverse corrections are calculated as following :

Transverse corrections is the information transferred between computation engines that shares common check nodes. Thus, these are computed for each check node.

$$|T_{(j)}| = Min_{i' \in B_j} |M_{(j,i')}| \tag{6.3}$$

$$sign(T_{(j)}) = \prod_{i' \in B_j} sign(M_{(j,i')}) \tag{6.4}$$

Stage 2: By using transverse information we calculate the net extrinsic information.

$$|E_{(j,i)}| = Min|Ep_{(j,i')}|, |T_{(j)}| \tag{6.5}$$

$$sign(E_{(j,i)}) = \prod sign(Ep_{(j,i')}, sign(T_{(j)}) \tag{6.6}$$

Just keep in mind extrinsic information of a particular check node in a particular matrix depends upon partial extrinsic information of the same check node calculated in stage 1 by the same matrix, but transverse information of the matrix that shares the common check node.
As explained in the above example transverse information is transferred from $H_{(1,1)}$ to $H_{(1,2)}$ and vice versa. Similarly from $H_{(2,1)}$ to $H_{(2,2)}$ and vice versa to compute the overall extrinsic information.

Thus if we generalise the concept for n-way partitioned matrix we just need to calculate the partial extrinsic information in stage 1. And then in subsequent stages we have to take in account the transverse information computed by same check node in all other n-1 partitioned matrices, that shares common check node. Thus we need a total number of n stages for a n-way partitioned matrix.

Chapter 7

Hardware Implementation of the algorithms

The algorithms are written in Aa language. The Aa description is then converted into vhdl using AHIR-tool-chain.[8]

7.1 Results

The designs are characterised by two parameters. First parameter is number of clock cycles required to complete the decoding process or the time required to complete one iteration of decoding. Second parameter is the hardware required to implement the design. By this way we can compare the designs and opt a better trade off between cost verses performance.

Figure 7.1 shows the time taken to decode for all the implemented designs for various order of matrices. The matrices used are randomly generated parity check matrices. The code rate is taken as half. Figure 7.1 concludes that partitioned min sum decoder can decode approximately at half the time as compared to min sum decoder.
A floating point operation is very costly. Thus to reduce the cost of hardware we can also use single floating point operator. Its a trade off between cost verses performance. Figure 7.1 shows that, if for both decoders we use shared floating point operator then the time to decode increases.

Figure 7.2 shows the variation of time to decode as the function of order of matrix. This concludes that the time to decode increases linearly with the order of matrix.

The hardware to implement the designs can be compared using Table 7.1

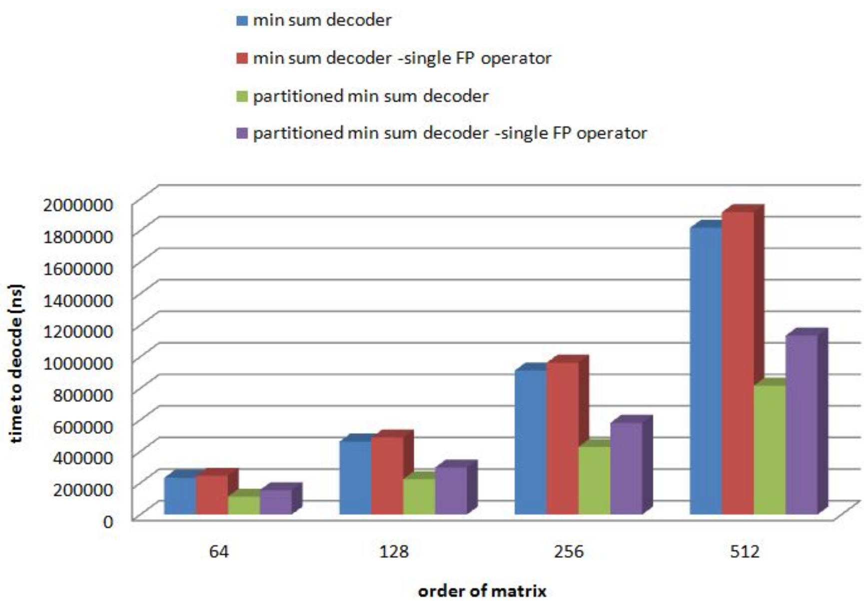

Figure 7.1: Comparison of time taken to decode

Table 7.1: Comparison of hardware generated after implementing the designs

	min sum decoder	min sum decoder (single FP unit)	partitioned min sum decoder	partitioned min sum decoder(single FP unit)
FF	18,076	19,034	49,854	55,988
LUT	19,502	20,621	51,929	60,296
Memory LUT	6	3	23	2
I/O	128	128	128	128
BRAM	56	56	80	80
BUFG	1	1	1	1

Figure 7.3 shows the time to decode the code block as a function of order of partitioning. The plot shows that for n-way partitioning of parity check matrix the time to decode reduce by a factor of n. The matrix used are random and have rate of half.

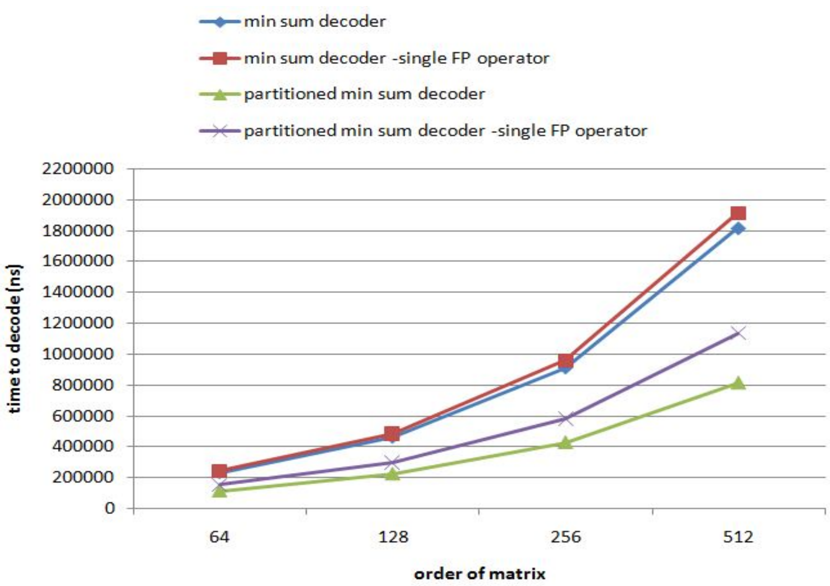

Figure 7.2: Timing trends as the function of order of matrix

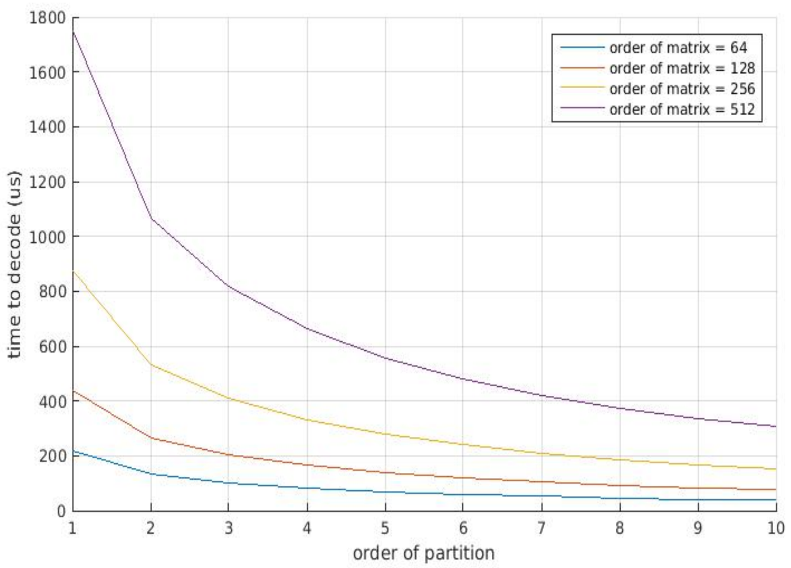

Figure 7.3: Timing trends as the function of order of partition

Chapter 8

Conclusion & Future Work

Partitioning can help to improve the performance of the min sum decoding algorithm by a factor of n for a n-way partition. Still, the trade off between cost to performance exist as increasing the partitions costs almost 2.5xn times per n-way partitioning. Another way to reduce cost is to decrease floating point units but again time to decode suffers.

The extension of the work can have different quantization levels of the floating point values and check for the trade off between accuracy of operation and error correcting threshold. As the precision of floating point value is decreased the convergence of algorithm also suffers. Thus, to get a optimum precision that gives a desired error correction can be explored as the future work.

Bibliography

[1] R. G. Gallager, Low-Density Parity-Check Codes. Cambridge, MA: MIT Press, 1963.

[2] D. J. C. MacKay, "Good error-correcting codes based on very sparse ma- trices," IEEE Trans. Inform. Theory, vol. 45, no. 2, pp. 399-431, March 1999.

[3] C. M. Huang, J. F. Huang and C. C. Yang, "Construction of quasi-cyclic LDPC codes from quadratic congruences," in IEEE Communications Letters, vol. 12, no. 4, pp. 313-315, April 2008.

[4] Muhammad Awais and Carlo Condo, "Flexible LDPC Decoder Architec- tures," VLSI Design, vol. 2012, Article ID 730835, 16 pages, 2012.

[5] Jong-Yeol and H.-J. Ryu, "A 1-gb/s flexible ldpc decoder supporting multiple code rates and block lengths," Consumer Electronics, IEEE Transactions on, vol. 54, pp. 417-424, May 2008.

[6] http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/manual.pdf

[7] http://www.cs.utoronto.ca/ radford/ldpc.software.html

[8] https://github.com/madhavPdesai/ahir/

[9] http://sigpromu.org/sarah/SJohnsonLDPCintro.pdf

[10] Arijit Mondal "Design of a min-sum LDPC decoder for error correction",IISC-banglore, 2014

Acknowledgements

I would like to express my gratitude to my guide Prof. Madhav P. Desai, IIT Bombay, whose invaluable guidance, constant encouragement and motivation has inspired me a lot.

Signature: ....

Anurag Gupta
153070050

Date: ...... June 2017