# A PLATFORM FOR LDPC CODE DESIGN AND PERFORMANCE EVALUATION

**Esa Alghonaim**[1]

*King Saud University, Riyadh- 51178, Saudi Arabia*

**Aiman El-Maleh**[2]**, M. Adnan Landolsi**[3] **and Sadiq M. Sait**[4]

*King Fahd University for Petroleum & Minerals*
*Dhahran-31261, Saudi Arabia*

**الخلاصــة:**

تم - في هذه الورقة - عرض تصميم وتنفيذ جوانب نظام متقدم للمساعده في إنشاء وتقويم أداء شفرات تصحيح الأخطاء قليلة المتانة ، مبنية على اختيار التكافوء .

يقوم النظام المتقدم في هذه الدراسة بوظيفتين رئيسيتين هما:

1- المحاكاه المتوازية لحساب أداء شفرات تصحيح الأخطاء قليلة الكثافة بزمن قصير جدا مقارنة بأنظمة المحاكاة الموجودة.

2- فحص مناطق الضعف وشفرات تصحيح الأخطاء قليلة الكثافة عن طريق العرض البصري (GUI) لقيم الوحدات الداخلية للشفرة في أثناء عمليات فك الشفرات.

إن أدوات قياس شفرات تصحيح الأخطاء الموجودة حاليا تستغرق وقتاً طويلاً جدا في تقويم أداء شفرات التصميم المحدده ،LDPC، وبخاصة في مناطق SNR العالية ، وذلك بسبب كثرة العمليات الحسابية المطلوبه. ولقد تم تجاوز هذه المشكلة بتطوير نظام متواز لتوزيع الحسابات خلال نقاط المعالجة في شبكات TCP/IP.

وقد دلت نتائج الآختبار على أن نظام المحاكاة المقدم يتحجم بعدد نقاط المعالجة.

إن الميزة العملية الأخرى للنظام المقترح أنه لا يحتاج إلى معالجة خاصة بل يمكن الاستفادة من فترات التوقف في نطاق المعالجة في الشبكة والعمل بشفافية لأي مستخدم . وعلاوة على ذلك تم اختيار شبكة الشياطين من أجل الكشف والاستفادة من الشبكة حتى لو كان ذلك في حالة انقطاع، وتتمثل المهمه الرئيسية الثانية للمقترح المقدم في تحليل أداء شفرات LDPC وفحص المناطق الضعيفة في تصميم الشفرات LDPC . وللحصول على هذا الغرض تم اقتراح طريقة لعرض سلوك ال LDPC وعرضها بالروسومات في عمليات فك الشفرات ومن ثم استخدامها للكشف عن المناطق الضعيفة مثل الانهيار "المصائد الدائرية " التي تمنع جهاز فك الشفرات من التحسن المستمر في أداء كشف الخطاء.

Corresponding Authors:
[1] E-mail: essa@ksu.edu.sa
[2] E-mail: aimane@kfupm.edu.sa
[3] E-mail: andalusi@kfupm.edu.sa
[4] E-mail: sadiq@kfupm.edu.sa

**Paper Received December 18, 2008; Paper Revised August 1, 2009; Paper Accepted December 21, 2009**

**ABSTRACT**

In this paper, the design and implementation aspects of a novel platform for aiding in the construction and performance evaluation of Low Density Parity Check (LDPC) codes is presented. The proposed platform is capable of performing two major tasks: (1) parallel simulation for evaluating LDPC codes performance in a very short time compared to existing LDPC code simulation tools, and (2) displaying the internal state of LDPC decoder during decoding iterations using a graphical user interface (GUI) which allows the LDPC code designer to visually inspect the weak areas. The existing LDPC code simulation tools take a long time in evaluating the performance of a specific LDPC code design, particularly in high SNR regions. This is due to the large number of computations required. This problem is overcome by developing a parallel protocol to distribute the computations among processing nodes in a TCP/IP network. As indicated by experimental results, the proposed simulation platform is scalable with the number of processing nodes. Another practical advantage of the proposed system is that it does not need dedicated processors; rather, it can utilize idle times of processing nodes in a network and work transparently to any node user. Furthermore, network daemons are adopted in order to detect and utilize network nodes even if they are in the log-off state. The second major task of the proposed platform is to analyze LDPC code performance and inspect weak areas in a given LDPC code design. To meet this objective, a method is proposed to graphically display the behavior of the LDPC decoder through the decoding iterations. This is then exploited to detect weak constructions (such as failure "trapping loops") that prevent the decoder from continuously improving the bit error performance.

***Key words***: LDPC codes, parallel processing, simulation, iterative decoder, SPA

# A PLATFORM FOR LDPC CODE DESIGN AND PERFORMANCE EVALUATION

## 1. INTRODUCTION

Currently, the most common approach to the evaluation of the error performance of LDPC codes is through extensive, time-consuming simulations. This is mainly due to the difficulty of obtaining analytical bounds and other closed-form results on the bit and/or packet error probabilities of LDPC codes [12,13]. The problem of using simulation is the long time needed, especially for large codes and/or high SNR region. Cavus *et.al.* proposed a technique that used the importance sampling (IS) method for simulating the performance of LDPC codes in AWGN (Additive White Gaussian Noise) channel at high SNR values [14]. Their technique is based on finding and biasing special bit node combinations, called *trapping sets,* known to be among the dominant sources of error events in LDPC belief propagation iterative decoding [3]. As a result, decoder errors are encountered using fewer simulation runs. However, this technique suffers from a major drawback, which is the difficulty of precisely injecting the bit node combinations.

Similar to many other types of error-correcting codes, there is no complete theory or analytical techniques for the exact evaluation of the error performance of LDPC decoding, and many research works rely on extensive simulations to obtain such results. This is basically the motivation for the work presented in this paper, which aims at distributing LDPC code performance simulation across a cluster of parallel computing machines to speed up the simulation process.

In this paper, two main tools to aid in LDPC code design are introduced. First, a parallel simulation platform to aid LDPC code designers to efficiently evaluate the performance of different designs of LDPC codes is introduced. The tool can be effectively used during the LDPC code design phase. As an example, one may generate 1000 random LDPC codes (with some constraint, such as graph girth) and then evaluate the performance of each code to determine the code with the best performance. Secondly, we introduce a visual tool to aid in inspecting LDPC decoding algorithms. By means of visual concepts, the tool illustrates the flow of internal messages between different processing nodes. It also draws the frames in error in a semi-planar graph, so the LDPC code designer can identify the reason which prevented these frames from being corrected.

Some preliminary description of the proposed LDPC code design and simulation platform has been previously published in [15]. In this paper we present a more comprehensive description of this platform, supported by thorough illustrations and design examples which were not given in [15]. In particular, we also add a new discussion of the GUI-based design suite, which was developed in order to provide a user-friendly, fast, and interactive environment for researchers to perform efficient analysis and optimization of LDPC code designs. In this respect, the developed platform has been successfully used as a main design and evaluation tool in the work presented in [12], which focused on the analysis of the problem of "trapping sets" in LDPC Tanner graphs, and their impact on error performance. Indeed, with the aid of the sophisticated "capture & visualization" capabilities of this proposed platform, we were able to study special combinations prone to these detrimental error events, and it was subsequently possible to develop tailored algorithms that can mitigate the impairments of these "trapping sets" through learning and neutralization phases. Some of the examples and illustrations included in this paper are drawn from [12], but all the specific details pertaining to the architecture of the proposed design and simulation platform, its protocols, GUI visualization capabilities, and original features are exclusively described in this paper.

The rest of this paper is organized as follows. In Section 2, a brief background on LDPC codes is presented. In Section 3, we give an introduction on how to measure the performance of LDPC codes. The proposed parallel simulation algorithms are presented in Section 4. The visual techniques for inspecting LDPC codes are presented in Section 5. Section 6 gives experimental results and design examples for the parallel simulation algorithm and shows some snapshots illustrating LDPC visual code inspection and optimization algorithms. Finally, Section 7 gives a summary and concluding remarks for the paper.

## 2. LDPC CODES

LDPC codes are a class of linear block codes that use a sparse parity-check matrix $H$ [1,2]. An LDPC code defined by the parity check matrix $H$ represents the parity equations in a linear form, where any given codeword $u$ satisfies the set of parity equations such that $u \cdot H = 0$ . Each column in the matrix represents a codeword bit while each row represents a parity check equation.

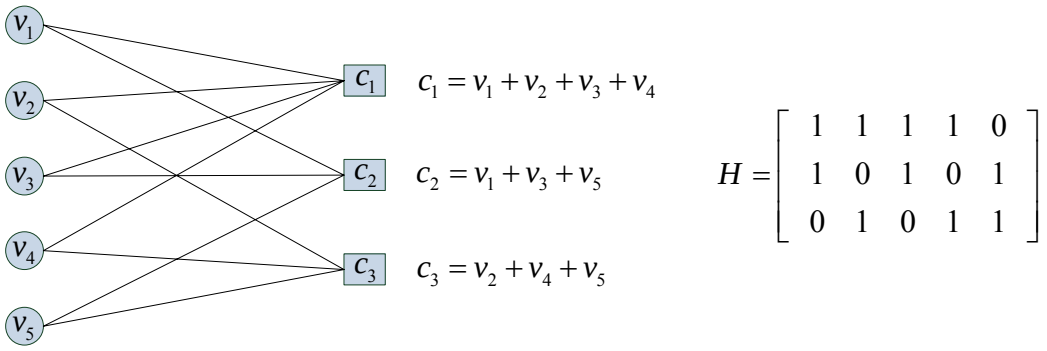


*Figure 1: The two representations of LDPC codes: graph form and matrix form*

LDPC codes can also be represented by bipartite graphs, usually called Tanner graphs, having two types of nodes: *variable bit nodes* and *check nodes*, interconnected by edges. An edge indicates that a given information bit appears in the parity check equation of the corresponding check bit, as shown in Figure 1. In the graph form, codewords are those vectors such that for all check nodes the sum of the neighboring positions among the message nodes is zero.

LDPC codes can be fully specified by their *H* matrix. The properties of an (*N*,*K*) LDPC code specified by an $M \times N$ *H* matrix can be summarized as follows:

- *N*: Block size, equals number of columns in the *H* matrix.
- *M*: Number of check equations, equals number of rows in the *H* matrix.
- *K*: Size of information bits, $K = N - M$ .
- *R*: Code rate, given by $R = \frac{K}{N} = 1 - \frac{M}{N}$, given that the $M$ rows in the *H* matrix are linearly independent.
- $d(c_j)$ : Check node degree, number of 1's in the $j^{th}$ row in the *H* matrix.
- $d(v_i)$ : Variable node degree, number of 1's in the $i^{th}$ column in the *H* matrix.
- *Regularity*: An LDPC code is said to be regular if $d(v_i) = p$ for $1 \le i \le N$ and $d(c_j) = q$ for $1 \le j \le M$ . In this case, the code is $(p, q)$ regular LDPC code. Otherwise, the code is considered irregular.
- *Code girth*: the minimum cycle length in the Tanner graph of the code.

Next, we give a brief description of the iterative message-passing belief propagation (BP) algorithm which is commonly used for decoding LDPC codes, and is shown to achieve optimum performance when the underlying code graph is cycle-free [1,2].

### 2.1. The Belief Propagation (BP) Algorithm

The BP algorithm is a message passing algorithm. The reason for the name is that, at each round of the algorithm, messages are passed from variable nodes to check nodes and from check nodes back to variable nodes. The messages from variable nodes to check nodes are computed based on the observed value of the variable node and some of the messages passed from the neighboring check nodes to that variable node [2].

The probability distribution for a binary random variable $u \in \{0,1\}$ is uniquely specified by the single parameter $p = \Pr[u = 1]$, since $\Pr[u = 0] = 1 - p$. Alternatively, the probability distribution is also uniquely specified by the logarithm of the ratio:

$$\lambda = \log \frac{\Pr[u = 1]}{\Pr[u = 0]}$$

To recover $p$ from $\lambda$, we observe that $e^{\lambda} = p/(1-p)$, and solving for $p$ yields $p = 1/(1+e^{-\lambda})$. The sign of $\lambda$ indicates the most likely value for $u$; $\lambda$ is positive when 1 is more likely than 0, and $\lambda$ is negative when 0 is more likely than 1. Moreover, the magnitude $|\lambda|$ is a measure of certainty. At one extreme, if $\lambda$ = 0, then 0 and 1 are equally likely. At the other extreme, if $\lambda = \infty$, then $u$ = 1 with probability 1, and $\lambda = -\infty$ implies $u$ = 0.

Before discussing the belief propagation algorithm, we discuss some terms and assumptions that will be used throughout the algorithm discussion [8]:

- $u_i$ : Code bit to be transmitted using Binary Phase Shift Keying (BPSK) modulation over an AWGN channel, $u_i \in \{0,1\}$, $1 \le i \le N$.
- $x_i$ : A transmitted channel symbol, $1 \le i \le N$, with a value given by:

$$x_i = \begin{cases} +s & when \quad u_i = 0 \\ -s & when \quad u_i = 1 \end{cases}$$, where $s$ is the BPSK signal strength.

- $y_i$ : A received channel symbol, $y_i = x_i + n_i$, where $n_i$ is zero-mean Additive White Gaussian Noise (AWGN) random variable with $\sigma^2$, $1 \le i \le N$.
- $\hat{u}_i$ : The estimated received bit value, $\hat{u}_i \in \{0,1\}$, $1 \le i \le N$. This is the output of the BP decoder.
- $R_j$ : For the $j^{th}$ row in an $H$ matrix, the set of column locations having 1's is given by $R_j = \{i : h_{ji} = 1\}$.
- $R_{j\backslash i}$ : The set $R_j$ excluding the location $i$ is by $R_{j\backslash i} = R_j - \{i\}$.
- $C_i$ : For the $i^{th}$ column in an $H$ matrix, the set of row locations having 1's is given by $C_i = \{j : h_{ji} = 1\}$.
- $C_{i\backslash j}$ : The set $C_i$ excluding the location $j$ is given by $C_{i\backslash j} = C_i - \{j\}$.
- $q_{ij}(b)$ : Message (extrinsic information) to be passed from variable node $v_i$ to check node $c_j$ regarding the probability of $u_i = b$, $b \in \{0,1\}$, as shown in Figure 2(a). It equals the probability that $u_i = b$ given extrinsic information from all check nodes, except node $c_j$.

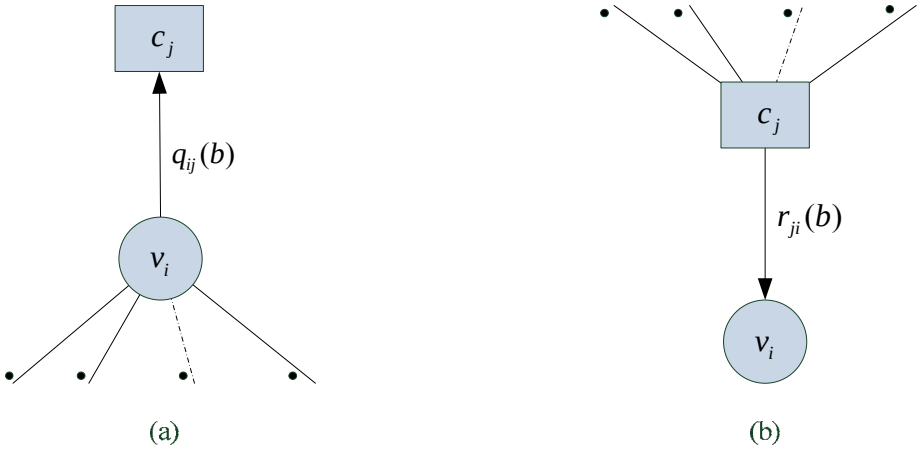


*Figure 2: (a) Variable-to-check message, (b) Check-to-variable message*

- $r_{ji}(b)$ : Message to be passed from check node $c_j$ to variable node $v_i$, which is the probability that the $j^{th}$ check equation is satisfied given that bit $u_i = b$ and the other bits have separable (independent) distribution given by $\{q_{ij'}\}_{j' \ne j}$, as shown in Figure 2(b).
- $Q_i(b)$ = the probability that $u_i = b$, $b \in \{0,1\}$.

- $L(u_i) \equiv \log \frac{\Pr(x_i = +1 \mid y_i)}{\Pr(x_i = -1 \mid y_i)} = \log \frac{\Pr(u_i = 0 \mid y_i)}{\Pr(u_i = 1 \mid y_i)}$ , $L(u_i)$ is usually referred to as the intrinsic information for node $v_i$ .
- $L(r_{ji}) \equiv \log \frac{r_{ji}(0)}{r_{ji}(1)}$ and $L(q_{ij}) \equiv \log \frac{q_{ij}(0)}{q_{ij}(1)}$
- $L(Q_i) \equiv \log \frac{Q_i(0)}{Q_i(1)}$

The BP algorithm involves one initialization step and three iterative steps as shown below:

**Initialization step:** The variable nodes messages are initialized as follows:

$$L(q_{ij}) \equiv L(u_i) = 2y_i / \sigma^2 \tag{1}$$

where $\sigma^2$ is the variance of noise in the AWGN channel.

**Iterative steps:** The three iterative steps are as follows:

(**I**) Update check nodes as follows:

$$L(r_{ji}) = \left( \prod_{i' \in R_{j \setminus i}} \alpha_{i'j} \right) \times \phi \left( \sum_{i' \in R_{j \setminus i}} \phi(\beta_{i'j}) \right) \tag{2}$$

where $\alpha_{ij} = sign(L(q_{ij}))$ , $\beta_{ij} = \left| L(q_{ij}) \right|$, and $\phi(x) = -\log(\tanh(x/2)) = \log \frac{e^x + 1}{e^x - 1}$

(**II**) Update variable nodes as follows:

$$L(q_{ij}) = L(u_i) + \sum_{j' \in C_{i \setminus j}} L(r_{j'i}) \tag{3}$$

(**III**) Compute estimated variable nodes as follows:

$$L(Q_i) = L(u_i) + \sum_{j \in C_i} L(r_{ji}) \tag{4}$$

Based on $L(Q_i)$ , the estimated value of the received bit ( $\hat{u}_i$ ) is given by:

$$\hat{u}_i = \begin{cases} 1 & if \quad L(Q_i) < 0 \\ 0 & else \end{cases} \tag{5}$$

**Stopping Criteria:** During LDPC decoding, the iterative steps I to III are repeated until one of the following two events occurs:

(1) The estimated vector ( $\hat{u} = (\hat{u}_1, ..., \hat{u}_n)$ ) satisfies the check equations, i.e., $\hat{u} \cdot H = 0$ .

(2) Maximum number of decoding iterations is reached.

## 3. LDPC PERFORMANCE SIMULATION

The block diagram of LDPC simulation over an AWGN (Additive White Gaussian Noise) channel is shown in Figure 3. The figure illustrates one simulation run, which involves the following steps: generating an information block, encoding it, sending it through an AWGN channel, decoding the received block, comparing it with the original transmitted information, and finally, updating simulation statistical counters. Statistical counters mainly include: total transmitted blocks, number of blocks in error, and total bits in error. First, *K* random information bits are generated and then encoded into an LDPC block of length N. Each bit $u_i \in \{0,1\}$ in the code block is modulated into a BPSK symbol $x_i = \{ -s , +s\}$ based on the value of $u_i$. The signal $x_i$ is then passed to an AWGN channel which adds noise to it to produce the received signal $y_i$ as follows: $y_i = x_i + n_i$, where $n_i$ is the AWGN sample. Note that instead of generating a random information block and encoding it into a block code, it is sufficient to generate an all zeros codeword ($u_i = 0 , 0 \le i \le N$). In this case, the encoding step is not needed and the simulation process will be faster. The last step is comparing the decoded received information with the originally transmitted information and then updating the simulation statistical counters accordingly.

To conclude this section, we give a review for the modeling of BPSK signal transmission over an AWGN channel. Assume $u_i \in \{0,1\}$ is the bit to be transmitted, then, the transmitted BPSK signal $x_i$ corresponding to $u_i$ is given by:

$$x_i = \begin{cases} -s & if \quad u_i = 1 \\ +s & if \quad u_i = 0 \end{cases} \tag{6}$$

The signal strength $s$ depends on the code rate (*R*) and the signal to noise ratio $E_b / N_0$ ) and is given by:

$$s = \sqrt{2 \times R \times \left( E_b / N_0 \right)} \tag{7}$$

The reason for including the LDPC code rate *R* (which is given by *K/N*) in Equation (2) is to make a fair comparison between different codes of different rates. This is because at lower LDPC code rates, the receiver will be allowed to accumulate the channel output for a longer time, and thus, the amount of noise (relative to signal) will decrease as a result of averaging [7].

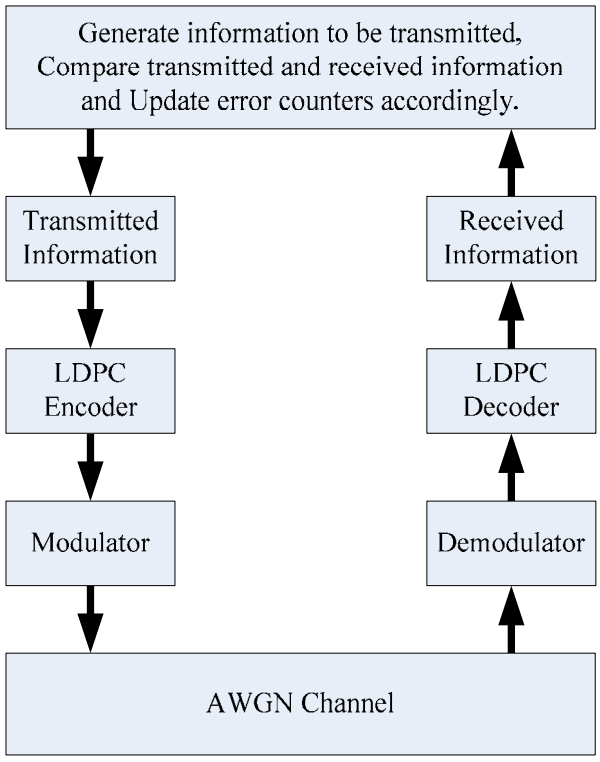


*Figure 3: Block diagram of a conventional LDPC performance evaluation simulator*

## 4. PARALLEL SIMULATION

The proposed parallel simulation platform consists of two main components: (1) Simulation controller, and (2) Processing nodes, as shown in Figure 4. The task of the simulation controller is to control the operation of processing nodes. The simulation controller sends simulation parameters to processing nodes, instructs them to start simulation, and collects statistical results from them.

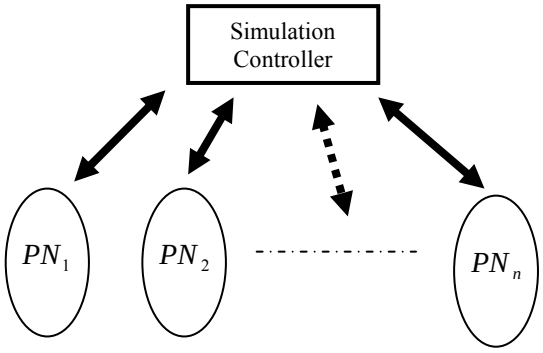


*Figure 4: Components of the proposed parallel LDPC simulation platform*

On the other hand, processing nodes receive simulation requests and parameters from the simulation controller, perform the LDPC simulation (as in Figure 3), and send simulation results to the simulation controller upon receiving a request from it.

Before discussing each of the two components in detail, we give a quick overview of the proposed LDPC simulation platform. First, a user sets simulation parameters (H-matrix, SNR point, decoding iterations …) and then starts the simulation. Upon starting, the simulation controller communicates with each processing node and sends simulation parameters to it. When a processing node receives simulation parameters, it starts LDPC code simulation independent of other processing nodes. The simulation controller periodically sends a results request message to each processing node. When a processing node receives a results request message, it sends its current results to the simulation controller and then initializes simulation counters.

### 4.1. Simulation Controller

The simulation controller is the central part in the proposed parallel simulation platform. It controls the operation of processing nodes. The simulation controller builds a look-up table, one entry for each reachable processing element. A processing node entry is used to keep track of its current state, such as: node address, statistical counters, H-matrix signature, node activity, decoding algorithm version, *etc.*

The simulation controller algorithm is divided into three stages:

**Stage 1**: Locating processing nodes: In this phase, the simulation controller sends a request message to all connected processing nodes to start simulation. Each active processing node responds by sending an acknowledgement to the simulation controller indicating its address and its decoding algorithm version number. When the simulation controller receives an acknowledgement from a processing node, it performs two tasks: (1) Adding a new record to the look-up table to keep track of the processing node, (2) Checking the version of the processing node decoding algorithm. If it is not up-to-date, the simulation controller sends a new version to the processing node using Simple File Transfer Protocol (SFTP). At the end of this stage, the simulation controller has a look-up table for all active processing nodes with an up-to-date decoding algorithm.

**Stage 2**: Sending simulation parameters: In this phase, the simulation controller sends LDPC simulation parameters to each processing node in its look-up table. The simulation parameters are listed in **Table 1**. When a processing node receives simulation parameters, it immediately starts simulation.

**Table 1. Description for the Simulation Parameters**

| Parameter | Description |
|---|---|
| Simulation ID | A unique number assigned for the current running simulation, to distinguish it from previous simulation sessions. |
| H-matrix | The locations of the ones entries in the H-matrix. |
| SNR | The SNR point in which simulation is requested to run. |
| Maximum decoding iterations | The number of maximum decoding iterations. |
| Decoding algorithm type | A number indicating which decoding algorithm type to run. Decoding algorithm types include: standard BP, Min-Sum algorithm, Bit Flipping, … *etc.* |
| Precision type | A flag indicating whether the precision used in decoder messages is fixed point or floating point. |
| Channel type | A number indicating the type of the channel used in simulation. Channel type may be: AWGN, Binary Erasure (BEC) or Binary Symmetric (BSC) channel. |

**Stage 3**: Partial results collection: In this phase, the simulation controller periodically sends results request messages to processing nodes. A processing element responds to this message by sending its results to the simulation controller and then initializing its statistical counters. Upon receiving a result message from a processing node, the simulation controller updates its aggregate statistical counters. The simulation controller periodically collects partial results from all processing nodes ($n$) in a time period of $T$ seconds. This means that each $t = T/n$ seconds, the simulation controller sends a results request message to a processing node, as shown in Figure 5: Simulation results request distribution over time. The simulation controller continues on this phase until the desired number of simulated blocks is reached. Then, it sends *stop simulation* messages to all processing nodes.

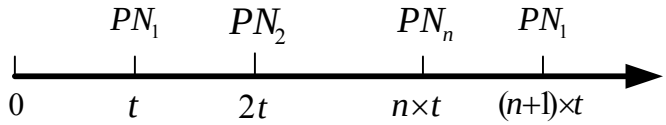


*Figure 5: Simulation results request distribution over time*

### 4.2. Processing Nodes

The function of processing nodes is to run LDPC performance simulation and send results to the simulation controller. Initially, a processing node receives a simulation request from the simulation controller, performs simulation, and then sends results to the simulation controller upon receiving a results request message from it.
The protocol of a processing node is shown in Figure 6, and is summarized as follows:

1. Wait until a start simulation request message is received from the simulation controller.
2. Receive simulation parameters (H-matrix, value of SNR, maximum decoding iterations, *etc.*,) from the simulation controller.
3. Initialize statistical counters (as mentioned before, statistical counters include: total transmitted blocks, number of blocks in error and total bits in error).
4. Perform transmission and decoding simulation for one block and update simulation counters accordingly.
5. If a 'results request' message is received from the simulation controller, then go to Step 7; otherwise go to Step 4.
6. If a stop simulation message is received from the simulation controller, then go to Step 1.
7. Send current statistical counters to the simulation controller, then go to Step 3.

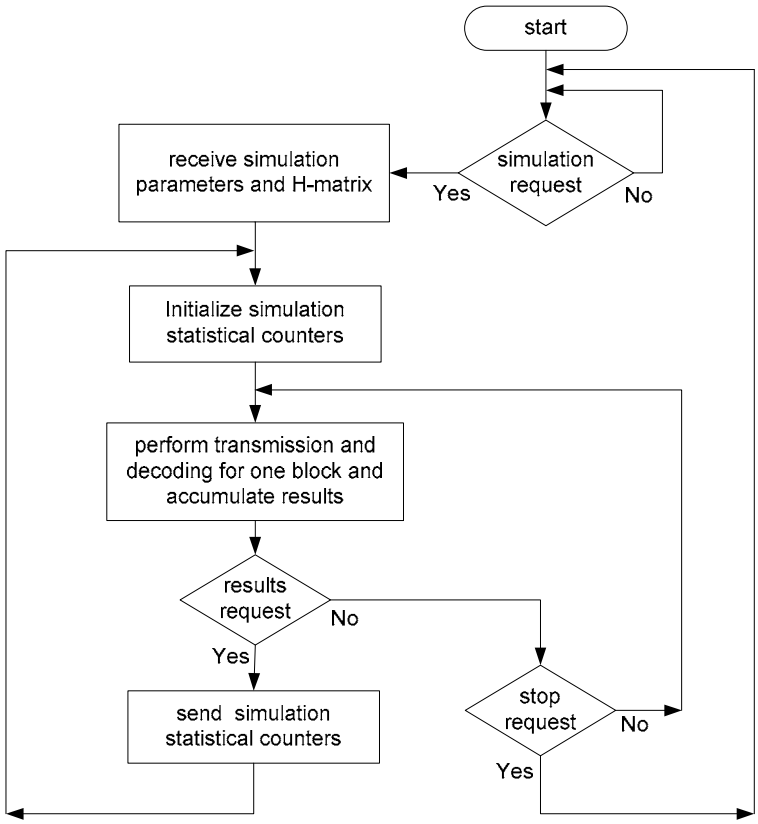


*Figure 6: Processing node protocol*

In the following, we discuss two issues related to processing nodes: fault tolerance and portability. The design of the simulation controller algorithm and the processing node protocol indicate that the parallel simulation platform is immune to processing nodes failure. The platform will keep running as long as the simulation controller is active and at least one processing node is active. This is because the results collected from processing nodes are independent of each other. For example, assume that the platform initially starts with 100 processing nodes. During the parallel simulation running, the simulation controller will distribute 100 simulation results requests every $T$ seconds (see Figure 5). If some processing nodes failed, the simulation controller still requests results from them; however, they will not respond. Since the simulation controller results request is not a blocking process, the simulation controller will continue collecting results from other active processing nodes. The only problem with a processing node failure is that the parallel simulation speed will decrease. However, this problem cannot be avoided. The parallel simulation platform is implemented using Windows OS. However, the processing nodes can run under different operating systems as long as they support TCP/IP protocols.

## 5. LDPC DECODER VISUAL ANALYSIS

The second task of the proposed platform is to graphically display the state of LDPC decoders during each decoding iteration. Instead of using numeric values to represent the messages between variable and check nodes, we use colors to represent the type (positive or negative) and strength of messages. We assume that a frame input to the decoder is an all-zeros word modulated using BPSK and then transmitted over an AWGN channel. The received signal is given by: $y_i = s + n_i$, where $n_i$ is the AWGN additive value for the $i^{th}$ bit, $s$ is the BPSK signal strength, and $0 \le i \le N$.

To analyze an LDPC code using the proposed tool, the code is first simulated using the developed parallel simulation platform. The parallel simulator collects the frames that could not be corrected (failed frames) by the LDPC decoder and stores them in a file. The visual inspection tool reads the failed frames from the input file and processes them one after another. For each failed frame, the tool reads the channel received values, initializes the LDPC decoder, and starts decoding iterations step by step. The user can select the decoding algorithm among several available decoding algorithms, such as standard BP algorithm, Min-Sum, *etc.*

The display of the internal decoding process involves two steps: (1) variable and check nodes placement, and (2) evaluating color levels for the connections (messages) between the nodes. The following sections illustrate each of the two steps in detail.

### 5.1. Variable and Check Nodes Placement

The variable and check nodes are placed in such a way as to provide a clear view for the LDPC designer for easier inspection of LDPC decoding iterations. Displaying the state of *all* variable and check nodes messages is not practical due to the large number of information to be displayed, especially when the code size is large. Instead of displaying information for all nodes, only messages related to variable nodes in error and check nodes connected to them are displayed. Variable nodes in error are called *failed* variable nodes. It is found that LDPC decoder failures are due to trapping sets [3–5]. Therefore, the proposed placement algorithm is designed to be compatible with trapping sets concepts. Before presenting the proposed placement algorithm, we give a brief overview of trapping sets.

#### *5.1.1. Trapping sets*

Trapping sets represent subgraphs in the Tanner graph of LDPC code that exhibit a strong influence on the height and point of onset of the error-floor [3–6,9]. In this sub-section, we define trapping sets.

**Definition:** A (*z*, *w*) trapping set *T* is a set of *z* variable nodes, for which the subgraph of the *z* variable nodes and the check nodes that are directly connected to them contains exactly *w* odd-degree check nodes [9].

Figure 7 shows an example of (3,2) trapping set $T = \{v_2, v_6, v_9\}$. Two check nodes ($c_4$ and $c_2$) are connected to an odd number of variable nodes in *T*.

The exact relationship between the frame error probability and the distribution of trapping sets is still not well understood. It is conjectured [3] that the frame error rate (*FER*) of a code can be represented as

$$FER \approx \sum_{z,w} \sum_{T(z,w)} P\{\xi_{T(z,w)}\} \tag{8}$$

where $\xi_{T(z,w)}$ denotes the set of decoder inputs that lead to a decoding failure on the (*z*,*w*) trapping set *T*; clearly, this set varies with the channel parameters. Although, at this point, the probabilities $P\{\xi_{T(z,w)}\}$ cannot be characterized analytically, there exist numerical methods for estimating their values [3].

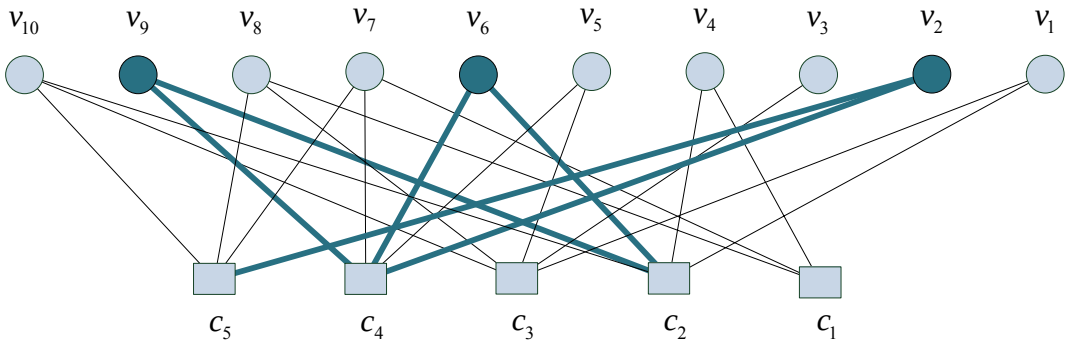


*Figure 7: An example of a (3,2) trapping set* $T = \{v_2, v_6, v_9\}$

#### *5.1.2. The proposed placement algorithm*

The circular shape is found to be the most appropriate shape in which variable and check nodes are displayed. Using the circular shape, failed variable nodes are placed on a circle, called the placement circle; while check nodes connecting the failed variable nodes are placed inside and outside the placement circle depending on the number of failed variable nodes they are connected to. Check nodes that have a single connection to failed variable nodes are placed outside the placement circle; while check nodes connected to more than one failed variable node are placed inside the placement circle. An example of variable and check nodes placement is shown in Figure 8. This example shows the placement of five failed variable nodes and the check nodes connected to them. Only three check nodes have a single connection to the failed variable nodes; these check nodes are placed outside the placement circle; while the remaining check nodes are placed inside the placement circle.

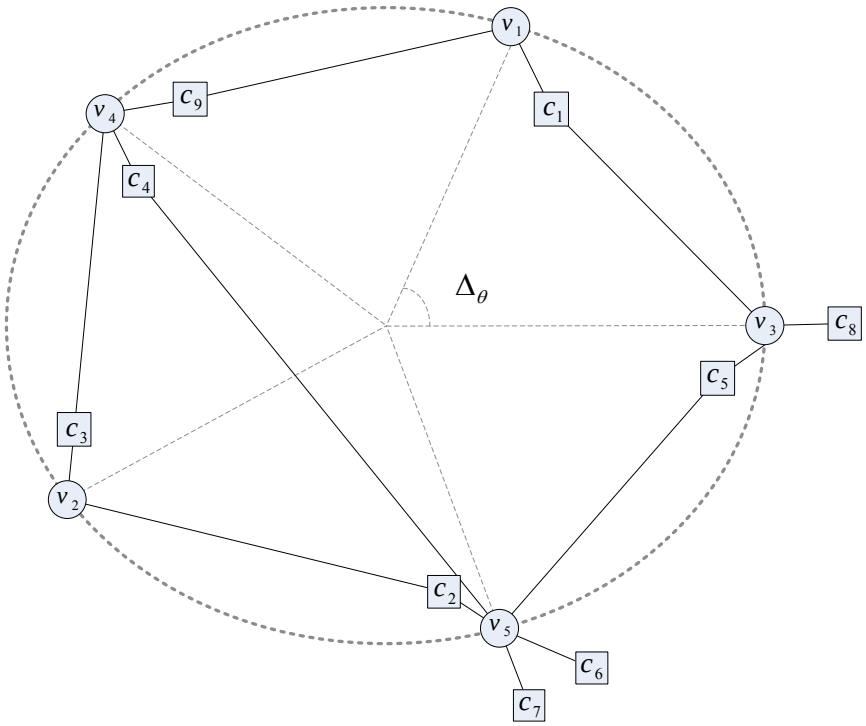


*Figure 8: An example of variable and check nodes placement*

Algorithm 1 shows the variable and check nodes placement algorithm. The example in Figure 8 is used to illustrate the algorithm. Initially, the set of failed variable nodes and the set of check nodes connected to them are identified as $V = \{v_1, v_2, v_3, v_4, v_5\}$ and $C = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9\}$. Since $|V|$=5, the variable nodes are placed on the placement circle starting from angle $\theta = 0^{\circ}$ with an increment of $\Delta_\theta = \frac{360^{\circ}}{5} = 72^{\circ}$. The first variable node is picked randomly from $V$, which is $v_3$, and is placed at angle $\theta = 0^{\circ}$. The check nodes connected to $v_3$ are $\{c_1, c_5, c_8\}$. Only $c_8$ is connected to only a single node from $V$. Therefore, $T = \{c_8\}$ and $O = \{c_1, c_5\}$. $c_8$ is placed outside the placement circle, while $\{c_1, c_5\}$ are placed inside the placement circle. Next, the algorithm determines the next variable node place. First, the algorithm determines that set of unplaced variable nodes that are a distance of two from $v_3$. A variable node $v_i$ is at a distance of two from $v_j$ if $v_i$ and $v_j$ have a common check node. Therefore, $S = \{v_1, v_5\}$. Since $S \neq \varnothing$, a variable node is picked randomly from $S$, which is $v_1$ in this example. Then, the while loop is repeated for the placement of variable nodes $v_1$, $v_4$, $v_2$, and $v_5$.

**Algorithm 1**: Variable and check nodes placement algorithm.

*Inputs*: LDPC code of size (*N*,*K*),

$L(r_{ji})$ : Message to be passed from check node $c_j$ to variable node $v_i$,

$L(Q_i)$ : The estimated value for variable node $v_i$,

$(x, y)$ : The coordinates of the placement circle origin,

$r$ : The radius of the placement circle.

*Output*: Placement of failed variable nodes and the check nodes connected to them.

$V = \{\varnothing\}$ // Set of failed variable nodes (variable nodes to be placed)

$C = \{\varnothing\}$ // Set of check nodes to be placed

For $i = 1$ to $N$ do

if $L(Q_i) \le 0$ then

$V = V + \{v_i\}$

$C = C$ + set of check nodes connected to $v_i$

end for

$\Delta_\theta = \dfrac{360^o}{|V|}$ // divide the circle into angles

$v_i$ = variable node picked randomly from $V$

$\theta = 0^o$ ; $U = V$

While $U \neq \varnothing$ do

$U = U - \{v_i\}$

// computing the place to display $v_i$

$x_i = r \times \cos(\theta) + x$ ; $y_i = r \times \sin(\theta) + y$

Place variable node $v_i$ at $(x_i, y_i)$

// placing check nodes connected to $v_i$

$T = \{\varnothing\}$ // set of check nodes with single connection to $V$

$O = \{\varnothing\}$ // set of check nodes with multiple connections to $V$

For all check nodes $c_j \in C$ connected to $v_i$ and not yet drawn do

$k$ = number of variable nodes in $V$ connected to $c_j$

if $k$ =1 then $T = T + \{c_j\}$ Else $O = O + \{c_j\}$

end for

Place check nodes in $T$ outside the placement circle

Place check nodes in $O$ inside the placement circle

// select next failed variable node to be placed

$S$ = set of variable nodes in $U$ that are at a distance of two from $v_i$

If $S \neq \varnothing$ then $v_i$ = variable node picked randomly from $S$

else if $U \neq \varnothing$ then $v_i$ = variable node picked randomly from $U$

$\theta = \theta + \Delta_\theta$ // next angle

end while

### 5.2. Decoder Messages Display

Displaying decoder messages using a coloring scheme is more suitable than numerical values, especially if the quantity of messages to be displayed is large. The decoder messages to be displayed are: (1) received signals (from AWGN channel), (2) variable node estimation value, and (3) messages from check nodes to variable nodes. Recall that the original transmitted codeword is an all-zeros codeword modulated using BPSK over an AWGN channel. Therefore, correct messages are those having positive values. In general, a message can be in one of two states: correct or in error. Messages which have positive values are correct signals, while messages with negative values are

in error. The green and red colors have been chosen to represent correct messages and messages in error, respectively.

Both green and red colors can take a value from 1 to $L$, where $L$ is the number of color levels. Therefore, the total number of different color levels for representing a message value is $2L+1$, after adding the zero value signal which is displayed in a black color. For a given message value $t$, the color level is obtained in two steps.

First, the message is clipped by $T$ (selected by the designer) as follows:

$$\bar{t} = \begin{cases} t & if \quad |t| < T \\ +T & if \quad t \geq T \\ -T & otherwise \end{cases} \tag{9}$$

Second, the color value $l \in \{-L,..,0,..,+L\}$ is computed as follows:

$$l = \left\lfloor \frac{\bar{t}}{T} \times L + 0.5 \right\rfloor \tag{10}$$

Figure 9 shows different color levels for representing a message with number of levels $L = 8$ and truncation value $T = 4.0$.

Computer monitors use an RGB system for graphical display. Each pixel (dot) has three attributes: red, green and blue. Various pixel colors are obtained by varying the amount of red, green and blue attributes of the pixel. Usually, one byte is assigned for each of the three colors: red, green and blue. If the values of the three bytes are set to zero, the resultant color is black. When the value of the green byte is increased from zero to 1,2 … up to 255 (fixing red and blue bytes values at zero), resultant color changes from black to green. As the value of the green byte increases, the green color becomes shinier, until it reaches to the shiniest green when the green byte value reaches to 255. Similarly, different levels of red color can be obtained by changing the red byte and fixing the green and blue bytes at zero.

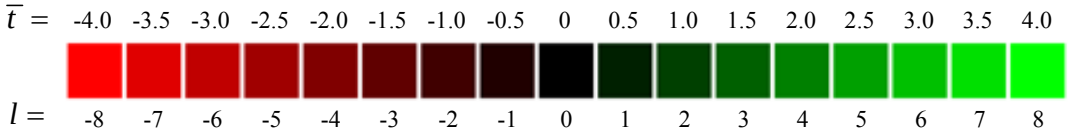


*Figure 9: Color levels for message representation, L=8 and T=4*

Using red and green colors with different levels of brightness, messages can be displayed as shown in the example of Figure 9. An LDPC designer can read messages simply as follows: (1) brightness represents the message reliability; shiner display indicates more reliable message, and (2) message color represents the type of the message; green color indicates correct (positive) message while red color indicates incorrect (negative) message.

Figure 10: Example illustrates messages color assignments shows an example of decoder messages coloring and Table 2 illustrates the numerical values of the messages and the way in which the colors of the messages are evaluated.

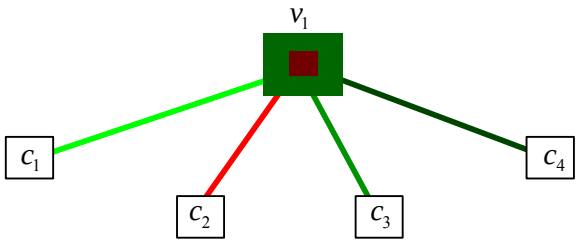


*Figure 10: Example illustrates messages color assignments*

Note that the shape of the variable node $v_1$ in the example of

Figure 10: Example illustrates messages color assignments is divided into two rectangles: outer and inner rectangles. The inner rectangle represents the received (intrinsic) message for $v_1$, while the outer rectangle represents the estimated decoding value for $v_1$.

Finally, the check nodes are displayed in two colors, depending on whether their corresponding check equations are satisfied or not. Satisfied check nodes are displayed in blue while unsatisfied check nodes are displayed in yellow.

**Table 2. Message Values and Their Associative Color Levels for**

| Message | Value ($t$) | $\bar{t}$ | $l$ | Color |
|---|---|---|---|---|
| Message from $c_1$ to $v_1$ | 5.2 | 4.0 | 255 | Green |
| Message from $c_2$ to $v_1$ | -4.1 | -4 | -255 | Red |
| Message from $c_3$ to $v_1$ | 2.3 | 2.3 | 147 | Green |
| Message from $c_4$ to $v_1$ | 1.1 | 1.1 | 70 | Green |
| Received Message (intrinsic) for $v_1$ | -1.8 | -1.8 | 115 | Red |
| Estimated value for $v_1$ | 1.6 | 1.6 | 102 | Green |

## 6. RESULTS

In this section, results from the proposed parallel simulation platform are shown. In addition, snapshots from the proposed GUI for analyzing LDPC decoders are depicted. A college network is used to implement the proposed parallel simulation platform with a maximum of 130 processing nodes. The message transfer between the simulation controller and processing nodes has been implemented using TCP/IP and UDP/IP. Indy 8.0 under Delphi 6 is used to run these network protocols.

The performance of the proposed simulation platform is evaluated by running it on a progressive edge growth (PEG) LDPC code [10] of size 1024bits, ½ rate, at SNR = 2.5dB, 128 decoding iterations and 10,000,000 transmitted blocks. Figure indicates simulation time as a function of the number of processing nodes which are varied from 2 to 128. Each time the number of processing nodes is doubled the corresponding simulation time is recorded.

Results indicate that simulation task speed up increases almost linearly as the number of processing nodes increases. By doubling the number of processing nodes, simulation time becomes about half. Simulation speedup is not exactly linear for the following two reasons: (1) the network is heterogenous, that is different network nodes have different processing capabilities, and (2) CPU time of network nodes is divided between network users (in our case, students working in labs) and simulation nodes (which have lower priority).

Recall that the simulation controller algorithm is divided into three phases: locating nodes, sending parameters, and collecting results. As the number of nodes increase, the time needed for locating nodes and sending parameters will increase linearly. However, this overhead time is very small compared to the processing time needed for the simulation. Another effect related to the scalability of the simulation platform is the update of the decoding algorithm code in the processing nodes. Each time the decoding algorithm code is to be updated, the simulation controller sends the new version to each processing node. This results in a linear increase of the code updating time as a function of number of processing nodes. In our implementation, the update is done only by the simulation controller. However, a suggested improvement is to allow each processing node to participate in the decoding algorithm update. In this way the, the update process can be performed using a tree structure and the update time will be a logarithmic function of the number of processing nodes.

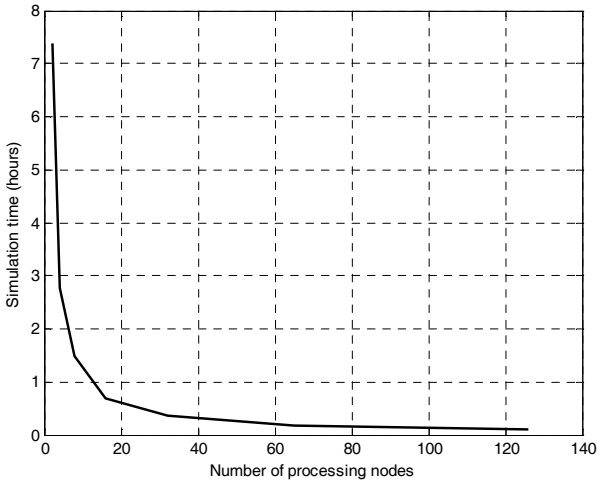


*Figure 11: Simulation time vs number of processing nodes*

Table 3 shows the simulation time for different decoding parameters and codes. For each simulation session, the number of generated blocks is 10,000,000 blocks. The first two results in the table (Sessions 1 and 2) indicate the effect of increasing the maximum decoding iterations on the simulation time. It is clear that doubling the maximum decoding iterations from 64 to 128 has a small impact on the simulation time. The reason is that the majority of generated blocks get corrected after several iterations. During decoding, very few blocks need more than 64 iterations. Sessions 1 and 3 indicate that the simulation time increases as the channel becomes worse. This is due to the increase of the average number of iterations per received block. However, most simulation programs use the count of incorrectly decoded blocks as a stopping criterion instead of using the total number of generated blocks. In this case, as the channel becomes worse (lower SNR), the number of incorrectly decoded blocks will increase and, hence, the simulation will stop earlier. Sessions 2 and 4 indicate the effect of changing the nodes degrees on the simulation time. Simulation time increases as the nodes degrees increase. This is because the decoding algorithm is based on the update of messages between the nodes. Finally, Sessions 2 and 5 indicate the effect of the computation precision type on the simulation time.

Next, some snapshots from the implementation of the proposed decoder visual analysis platform are presented. Figure shows a snapshot from the developed platform while decoding a (50,25) LDPC code.

**Table 3. Effect of Decoding Parameters on Simulation Time**

| Session | Code Size | Degree | SNR | Precision | Decoding iterations | Simulation Time |
|---|---|---|---|---|---|---|
| 1 | (2000,1000) | (3,6) | 2.0 dB | Floating point | 128 | 3 days and 3 hours |
| 2 | (2000,1000) | (3,6) | 2.0 dB | Floating point | 64 | 2 days and 22 hours |
| 3 | (2000,1000) | (3,6) | 1.0 dB | Floating point | 64 | 4 days and 7 hours |
| 4 | (2000,1000) | (4,8) | 2.0 dB | Floating point | 64 | 6 days and 15 hours |
| 5 | (2000,1000) | (3,6) | 2.0 dB | Fixed point | 64 | 13 hours |

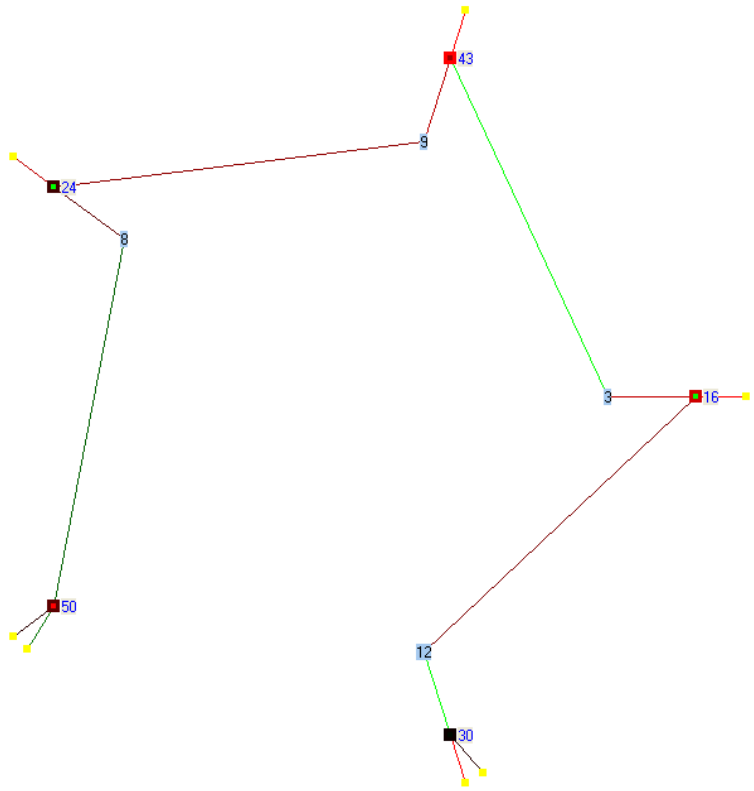


*Figure 12: Example for an LDPC decoding iteration display of a (50,25) code*

There are 5 failed variable nodes $T_1 = \{v_{16}, v_{24}, v_{30}, v_{43}, v_{50}\}$ and 11 check nodes in the example of Figure . Among the 11 check nodes, there are 7 check nodes that have a single connection to the failed variable nodes; therefore, they are drawn outside the placement circle. The remaining 4 check nodes are drawn inside the placement circle. Note

that the placement circle is not drawn here. It can be observed that there are 7 unsatisfied check nodes, displayed in yellow, which are connected to an odd number of variable nodes from $T_1$.

The example of Figure 13:

demonstrates one of the benefits of the developed decoding display platform. In this example, a frame in error (frame that could not be corrected by BP LDPC decoder) for an LDPC code of size (1024,512) is used as an input to the display platform for analysis. The decoding iterations are displayed starting from 1 and up to the maximum iterations (64 in this example).

It is found that the LDPC decoder is stuck and does not change its state starting from iteration number 8 and up to the maximum iteration, and gives the graph displayed in Figure 13:

It can be observed that the failed variable nodes (their outer square color is red) forms a (7,1) trapping set *T*. The elements of the trapping set are $T_2 = \{v_{21}, v_{70}, v_{186}, v_{258}, v_{569}, v_{678}, v_{932}\}$. There is only one check node (with yellow color) that is connected to an odd degree of variable nodes in $T_2$. Another observation is that some nodes may be a subset of a trapping set although their intrinsic (received) value is correct. The variable node $v_{258}$ is an example of this case. Even though the intrinsic (received) value of $v_{258}$ is correct with high reliability, as can be observed from the shiny green color of its drawing, its decoded value is in error due to the incorrect (red) messages from check nodes $c_{166}$ and $c_{219}$.

It can be observed that the graph of the trapping set $T_2$ consists of an interconnection of small length cycles. For example, a cycle of size 6 is formed between the variable nodes set $\{v_{21}, v_{70}, v_{258}\}$ and three of the check nodes connected to them. Similarly, a cycle of size 6 is formed between the variable nodes set $\{v_{186}, v_{569}, v_{678}\}$ and three of the check nodes connected to them. From this analysis, to improve error correction performance, one may break this trapping set by deleting some connections, one or two, in the trapping set graph and reconnect them elsewhere in the LDPC code graph.

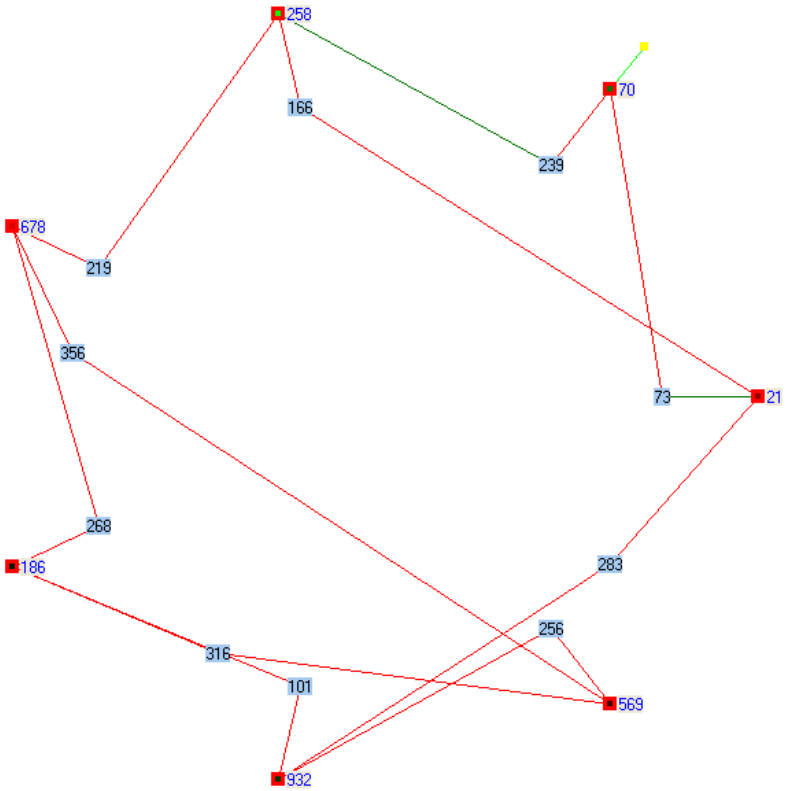


*Figure 13: Example for an LDPC decoding iteration display for a (1024,512) code*

## 7. CONCLUSIONS

This paper introduced a proposal and implementation for LDPC codes performance evaluation and design platform. This platform performs two main tasks: first, it performs parallel simulation to evaluate an LDPC code performance; second, it graphically displays the state of LDPC decoder during decoding iterations, which allows LDPC code designer to investigate the weakness of a given LDPC code.

The proposed parallel simulation platform achieves almost linear speed-up as a function of the number of processing elements used. Simulation time for evaluating the performance of a PEG LDPC code of size 1024 bits at SNR=2.5 and 10,000,000 blocks is reduced from 14.5 hours using a single node to less than 7 minutes using 126 network nodes.

The proposed simulation platform is cost effective as it does not need a dedicated network and is based on using existing networks, such as college networks. Simulation processes running in network nodes are transparent from network users and they are assigned low priorities so they have minor effects on the performance of a network user's jobs. Furthermore, a mechanism is proposed to allow simulation processes to run only at night, and they are stopped during network users' working hours.

The second task of the platform is to allow for a visual inspection of the message transfer of LDPC decoders. The proposed visual decoding consists of two steps. First, variable nodes in error and check nodes connected to them are placed on a graph. The placement of the nodes is made so that it becomes easy to distinguish trapping sets in the graph. Second, decoder messages are displayed using colors rather than using numerical values. This makes it easier for quickly identifying the variable nodes and messages in error.

## REFERENCES


[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 Matrices", *IEEE Trans. Inform. Theory*, **45**(1999), pp. 399–431.

[3] T. Richardson, "Error Floors of LDPC Codes", in *Proc. 41st Annual Allerton Conf: On Communication, Computing and Control, Monticello*, Oct. 2003.

[4] E. Cavus and B. Daneshrad, "A Performance Improvement and Error Floor Avoidance Technique for Belief Propagation Decoding of LDPC Codes", in *IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications*, **4**(2005), pp. 2386 –2390.

[5] T. Tian, C. Jones, J. D. Villasenor, and R. D. Wesel, "Construction of Irregular LDPC Codes With Low Error Floors", in *Proc. IEEE International Conference on Communications*, **5**(2003), pp. 3125–3129.

[6] O. Milenkovic, E. Soljanin, and P. Whiting, "Trapping Sets in Irregular LDPC Code Ensembles", in *Proc. IEEE Int'l Conf. on Communications, ICC 2006*, pp. 1101–1106.

[7] N. Radford, Software Package for LDPC Codes, www.cs.utoronto.ca/~radford/ldpc.software.html.

[8] W. Ryan, "A Low-Density Parity-Check Code Tutorial, Part II - The Iterative Decoder", Technical Report, ECE Dept., University of Arizona, April 2002.

[9] S. Landner and O. Milenkovic, "Algorithmic and Combinatorial Analysis of Trapping Sets in Structured LDPC Codes", in *International Conference on Wireless Networks, 2005, Communications and Mobile Computing*, **1**, pp. 630–635.

[10] X.-Y. Hu, E. Eleftheriou, and D.-M. Arnold, "Progressive Edge-Growth Tanner Graphs", in *Proc. IEEE GLOBECOM 2001, San Antonio, TX*, Nov. 2001, pp. 995–1001.

[11] Condor Software Package , http://www.cs.wisc.edu/condor/.

[12] E. Al-Ghonaim, A. El-Maleh, and M.A. Landolsi, "New Technique for Improving the Performance of LDPC Codes in the Presence of Trapping Sets", *EURASIP Journal on Wireless Communications and Networking*, Article ID 362897, DOI:10.1155/2008/362897, June 2008.

[13] S. Lin and D. J. Costello, Jr., "Error Control Coding", Pearson Prentice Hall, 2004.

[14] E. Cavus, C. L. Haymes, and B. Daneshrad, "An IS Simulation Technique for Very Low BER Performance Evaluation of LDPC Codes", in *Proc. IEEE International Conference on Communications* 2006. ICC '06. **3**, pp. 1095–1100.

[15] E. Alghonaim, A. El-Maleh, and M. A. Landolsi, "Parallel Computing Platform for Evaluating LDPC Codes Performance", in *Proc. IEEE Int'l Conf. on Signal Processing and Communications ICSPC'07*, pp. 157–160, November 2007.