PARALLEL COMPUTING PLATFORM FOR EVALUATING LDPC CODES PERFORMANCE

Esa Alghonaim, Aiman El-Maleh and Adnan Al-Andalusi

King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia

ABSTRACT

This paper presents a novel approach for the design and implementation of a simulation platform for evaluating LDPC codes performance. The existing LDPC code simulation tools consume very long time in evaluating the performance of a specific code design. This is due to the intensive number of required computations. 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 to run it; rather, it can utilize idle times of processing nodes in a network and work transparent to a node user. Furthermore, network daemons are used to utilize network nodes even if they are in the log-off state.

Index Terms— LDPC codes, parallel processing, simulation, iterative decoder, SPA.

1. INTRODUCTION

Forward Error Correcting (FEC) codes are an essential component of modern state-of-the-art digital communication and storage systems. Indeed, in many of the recently developed standards, FEC codes play a crucial role for improving the error performance capability of digital transmission over noisy interference-impaired communication channels.

The leading family of FEC codes are widely considered to be Low Density Parity Check codes (LDPCs) [1], as they demonstrate performance very close to the information-theoretic bounds predicted by Shannon theory, while at the same time having the distinct advantage of low-complexity, near-optimal iterative decoding. Unfortunately, there is as yet no theory for evaluating the performance of a given LDPC code. Currently, the only way to evaluate LDPC codes performance is through simulation. The problem of using simulation is the long time needed, especially for large codes.

In this paper, we introduce a parallel simulation platform to aid LDPC code designers to evaluate the performance of different designs of LDPC codes. The tool can also be used during the LDPC code design phase. As an example, one may generate 1000 random LDPC codes (with some constraint, such as cycling structure) and then evaluate the performance of each code to determine the code with the best performance.

The remainder of this paper is organized as follows: In Section 2, conventional LDPC code simulator is introduced. In Section 3, we give a brief review of the sum-product algorithm. In Section 4, the proposed simulation platform is presented. Experimental results are given in Section 5. Section 6 concludes the paper.

2. LDPC CODE PERFORMANCE SIMULATION

The block diagram of LDPC simulation over an AWGN (additive White Gaussian Noise) channel is shown in Figure 1. The path of one simulation iteration is illustrated, 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.

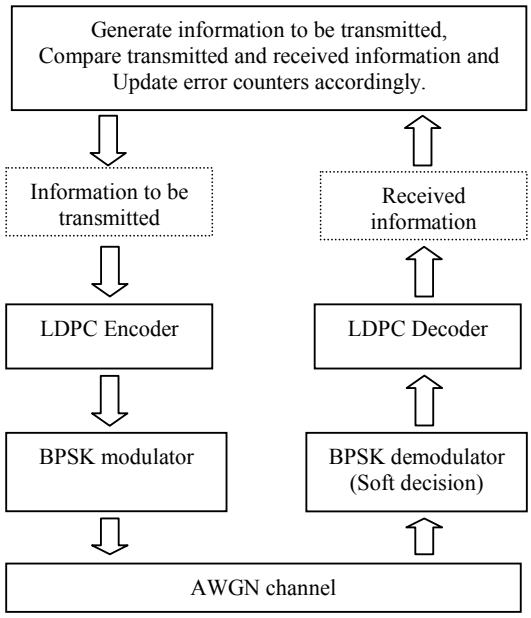

Figure 1 Block diagram of a conventional LDPC performance evaluation simulator.

First, K random information bits are generated and then encoded into an LDPC block of length N. Each bit $c_i \in \{0,1\}$ in the code block is modulated into a BPSK (Binary Phase Shift Keying) symbol $x_i = \{ -s , +s\}$ based on the value of c_i, where s is the BPSK signal strength. 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 additive value. Note that instead of generating a random information block and encoding it into a block code, it is enough to generate an all zeros block code ($c_i = 0, 0 \le i \le N$). 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 $c_i \in \{0,1\}$ is the bit to be transmitted, then, the transmitted BPSK signal x_i corresponding to c_i is given by:

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

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

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

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 [5].

3. THE SUM-PRODUCT ALGORITHM

LDPC codes are a class of linear block codes that use a sparse, random-like parity-check matrix [1]. LDPC codes can also be represented by bi-partite factor graphs having two types of nodes: *variable bit nodes* and *check nodes*, interconnected by edges whenever a given information bit appears in the parity check equation of the corresponding check bit. The iterative sum-product algorithm (SPA) can be used for decoding LDPC codes, and is shown to achieve optimum performance when the underlying code graph is cycle-free In the following, a brief description of this algorithm is given based on the notation in [4]. The SPA algorithm is also called Belief Propagation (BP) algorithm.

Assume a binary (N,K) LDPC code is described by a sparse parity check matrix of size $M \times N$, where M is the number of parity-checks corresponding to the parity-check nodes in a bipartite graph, and N is the number of variable nodes corresponding to the encoded symbols.

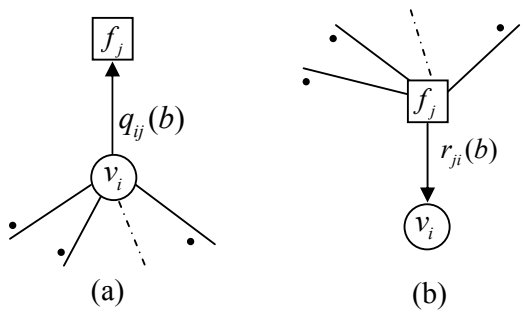

Figure 2 (a) Variable-to-check message, (b) Check-to-variable message.

Before discussing the SPA algorithm, we introduce some terms that will be used throughout the discussion of the SPA algorithm [4]:

- For the j^{th} row in an H matrix, the set of column locations of the 1's is given by $R_j = \{i : h_{ji} = 1\}$. The set of column locations of the 1's, excluding location i is given by $R_{j \setminus i} = \{i' : h_{ji'} = 1\} \setminus \{i\}$.
- For the i^{th} column in an H matrix, the set of row locations of the 1's is given by $c_i = \{j : h_{ji} = 1\}$. The set of row locations of the 1's, excluding the location j is given by $c_{i \setminus j} = \{j' : h_{j'i} = 1\} \setminus \{j\}$
- $q_{ij}(b)$: Message (extrinsic information) to be passed from variable node v_i to check node f_j regarding the probability that $c_i = b, b \in \{0,1\}$, as shown in Figure 2(a). It equals the probability that $c_i = b$ given extrinsic information from all check nodes, except node f_j.
- $r_{ji}(b)$: Message to be passed from check node f_j to variable node v_i, which is the probability that the j^{th} check equation is satisfied given bit $c_i = b$ 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 $c_i = b$, $b \in \{0,1\}$
- $L(c_i) \equiv \log \dfrac{\Pr(x_i = +1 \mid y_i)}{\Pr(x_i = -1 \mid y_i)} = \log \dfrac{\Pr(c_i = 0 \mid y_i)}{\Pr(c_i = 1 \mid y_i)}$
- $L(r_{ji}) \equiv \log \dfrac{r_{ji}(0)}{r_{ji}(1)}$ and $L(q_{ij}) \equiv \log \dfrac{q_{ij}(0)}{q_{ij}(1)}$
- $L(Q_i) \equiv \log \dfrac{Q_i(0)}{Q_i(1)}$

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

Initialization step: Set the initial value of each variable node signal as follows: $L(q_{ij}) \equiv L(c_i) = 2y_i / \sigma^2$, where σ^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:

$$L(r_{ji}) = \left(\prod_{i' \in R_{j \setminus i}} \alpha_{i'j} \right) \times \phi \left(\sum_{i' \in R_{j \setminus i}} \phi(\beta_{i'j}) \right) \quad (3)$$

Where $\alpha_{i'j} = sign(L(q_{ij}))$, $\beta_{ij} = \left| L(q_{ij}) \right|$

$$\phi(x) = -\log(\tanh(x/2)) = \log \frac{e^x + 1}{e^x - 1}$$

(II) Update variable nodes as follows:

$$L(q_{ij}) = L(c_i) + \sum_{j' \in C_{i \setminus j}} L(r_{j'i}) \quad (4)$$

(III) Compute estimated variable nodes as follows:

$$L(Q_i) = L(c_i) + \sum_{j \in C_i} L(r_{ji}) \quad (5)$$

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

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

During LDPC decoding, the iterative steps I to III are repeated until the following event occurs: $\hat{\bar{c}} \cdot H^T = 0$ OR maximum iterations is reached.

4. THE PROPOSED PARALLEL SIMULATION

The proposed parallel simulation platform consists of two main components: (1) Simulation controller, and (2) Processing nodes, as shown in Figure 3. The task of simulation controller is to control the operation of processing nodes. Simulation controller sends simulation parameters to processing nodes, instruct them to start simulation and collect statistical results from them. In the other hand, processing nodes receive simulation requests and parameters from simulation controller, perform the LDPC simulation (as in figure 1) and send simulation results to simulation controller upon receiving a request from it.

Before discussing each of the two components in details, we give a quick view for the proposed LDPC simulation platform. First, a user sets simulation parameters (H-matrix, SNR point, decoding iterations …) and then starts the simulation. Upon starting, 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. Simulation controller periodically sends a results request message for each processing node. When a processing node receives a results request message, it sends its pending results to simulation controller and then initializes simulation counters.

4.1. Simulation Controller

Simulation controller is the central part in the proposed parallel simulation platform. It controls the operation of processing nodes. 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, decoding algorithm version, etc.

The simulation controller algorithm is divided into three stages:

Stage 1: Locating processing nodes: In this phase, simulation controller sends a start simulation request

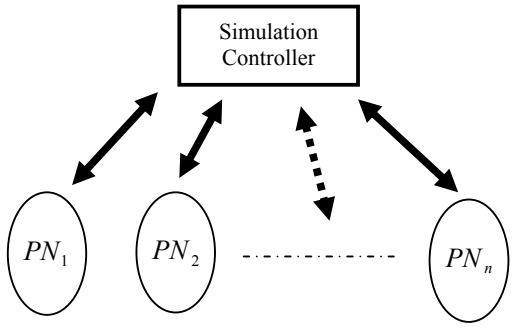

Figure 3. Components of the proposed parallel LDPC simulation platform.

message to all connected processing nodes. Each active processing node responds by sending an acknowledgement to simulation controller indicating its address and its decoding algorithm version number. When 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, simulation controller sends a new version to the processing node using File Transfer Protocol (FTP). At the end of this stage, simulation controller has a look-up table for all active and up to date processing nodes.

Stage 2: Sending simulation parameters: In this phase, simulation controller sends LDPC simulation parameters to each processing node in its look-up table. The simulation parameters include: H-matrix, SNR value, maximum LDPC decoding iterations, decoding algorithm type (floating point or fixed point), etc. When a processing node receives simulation parameters, it immediately starts simulation.

Stage 3: Partial results collection: In this phase, simulation controller periodically sends results request messages to processing nodes. A processing element responds to this message by sending its results to simulation controller and then initializing its statistical counters. Upon receiving results message from a processing node, simulation controller updates its aggregate statistical counters. 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, simulation controller sends a results request message to a processing node, as shown in Figure 4. 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.

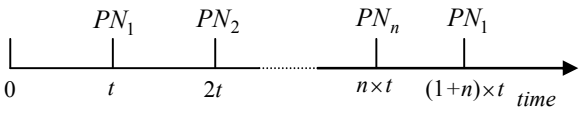

Figure 4 Simulation results request distribution over time.

4.2. Processing Nodes

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

1. Wait until a start simulation request message is received from simulation controller.
2. Receive simulation parameters (H-matrix, value of SNR, maximum decoding iterations,…) from simulation controller.
3. Initialize statistical counters.
4. Perform transmission and decoding simulation for one block and update simulation counters accordingly.
5. If a results request message is received from simulation controller, then go to step 7, otherwise go to step-4.
6. If a stop simulation message is received from simulation controller, then go to step 1.
7. Send pending statistical counters to simulation controller, then go to step 3.

5. EXPERIMENTAL RESULTS

Our college network is used to implement the proposed parallel simulation platform with a maximum of 126 processing nodes. The messages between simulation controller and processing nodes have 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 an LDPC code of size 1024bits, ½ rate, at SNR = 2.5dB, 128 decoding iterations and 10,000,000 transmitted blocks. Figure 5 indicates simulation time as a function of the number of processing nodes. We vary processing nodes from 2 nodes up to 126 nodes, each time we almost double number of processing nodes.

Results indicate that simulation time decreases almost linearly as number of processing nodes increases. By doubling the number of processing nodes, simulation time becomes about half. Simulation time decrease is not exactly linear because of the following two reasons: (1) different network nodes have different processing capabilities, (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).

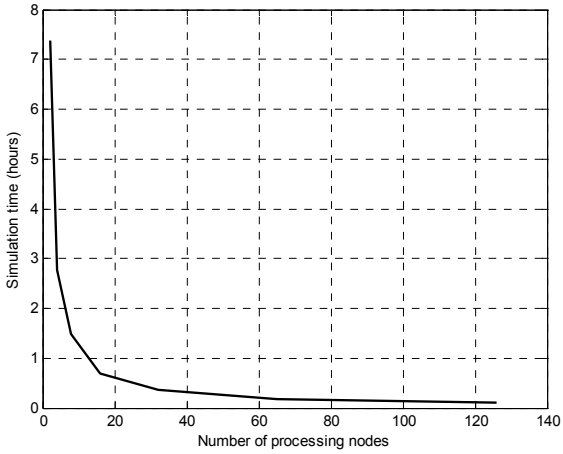

Figure 5 Simulation time vs. number of processing nodes.

6. CONCLUSION

In this work, we have proposed a parallel computing simulation platform for efficiently evaluation LDPC code performance. It achieves almost linear speed-up as a function of the number of processing elements used. Simulation time for evaluating the performance of 1024 bits LDPC code at SNR=2.5 and 10,000,000 blocks is reduced from 14.5 hours using a single node to only 6 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 do not affect performance of a network user's jobs. The proposed platform can be efficiently used in development of LDPC codes with high performance.

ACKNOWLEDGMENT

The authors thank King Fahd University of Petroleum & Minerals for support of this work under project no. EE/DENSITY/387.

REFERENCES

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

[2] Dong-U Lee , "Reconfigurable Hardware for Function Evaluation and LDPC Coding ", *PhD Thesis - Department of Computing Imperial College London United Kingdom* dong.lee@ic.ac.uk, *July 2003*

[3] Frederic GUILLOUD , "Generic Architecture for LDPC codes decoding", *PhD Thesis – July 2004.*

[4] William Ryan, "A Low-Density Parity-Check Code Tutorial, Part II - The Iterative Decoder", ECE dept. The University of Arizona, April 2002.

[5] Radford Neal , Software for Low Density Parity Check codes, www.cs.utoronto.ca/~radford/ldpc.software.html

[6] S.-Y. Chung, J. G. D. Forney, T. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Commun. Lett.*, vol. 5, pp. 58–60, Feb. 2001.