WIRELESS COMMUNICATIONS AND MOBILE COMPUTING
Wirel. Commun. Mob. Comput. 2013; **13**:663–670
Published online 31 March 2011 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/wcm.1129

RESEARCH ARTICLE

A fast low-density parity-check code simulator based on compressed parity-check matrices

Shek F. Yau[1], Tan L. Wong[1], Francis C. M. Lau[1] and Yejun He[2]*

[1] Department of Electronic and Information Engineering, Hong Kong Polytechnic University, Hong Kong, China
[2] Department of Communication Engineering, College of Information Engineering, Shenzhen University, Guangdong, China

ABSTRACT

Low-density parity-check (LDPC) codes are very powerful error-correction codes with capabilities approaching the Shannon's limits. In evaluating the error performance of a LDPC code, the computer simulation time taken becomes a primary concern when tens of millions of noise-corrupted codewords are to be decoded, particularly for codes with very long lengths. In this paper, we propose modeling the parity-check matrix of a LDPC code with compressed parity-check matrices in the check-node domain (CND) and in the bit-node domain (BND), respectively. Based on the compressed parity-check matrices, we created two message matrices, one in the CND and another in the BND, and two domain conversion matrices, one from CND to BND and another from BND to CND. With the proposed message matrices, the data used in the iterative LDPC decoding algorithm can be closely packed and stored within a small memory size. Consequently, such data can be mostly stored in the cache memory, reducing the need for the central processing unit to access the random access memory and hence improving the simulation time significantly. Furthermore, the messages in one domain can be easily converted to another domain with the use of the conversion matrices, facilitating the central processing unit to access and update the messages. Copyright © 2011 John Wiley & Sons, Ltd.

KEYWORDS
compressed parity-check matrices; domain conversion matrices; LDPC codes; simulation time

***Correspondence**
Dr Yejun He, PhD, Department of Communication Engineering, College of Information Engineering, Shenzhen University, No. 3688, Nanhai Road, Nanshan District, Shenzhen City, Guangdong 518060, China.
E-mail: heyejun@ieee.org

1. INTRODUCTION

Low-density parity-check (LDPC) codes were first proposed by Gallager in 1962 [1]. They were considered as impractical because of their intensive computation requirement and lack of efficient computation facilities at that time. In 1996, the LDPC codes were re-discovered and were shown to possess Shannon limit-approaching performance [2]. Recently, derivatives of LDPC codes have been adopted in various standards such as IEEE 802.11n, IEEE 802.16e, and DVB-S2 [3,4].

A LDPC code can be represented by a parity-check matrix (PCM) **H** or a Tanner graph [5–7]. In the PCM, each row corresponds to a check node (CN), whereas each column corresponds to a bit node (BN) in the Tanner graph. We consider a binary LDPC code, implying that the elements of **H** are either 0 or 1. If an element in **H** is 1, the corresponding CN and BN in the Tanner graph will be connected; otherwise, those nodes are not connected. Figure 1 illustrates the relationship between a PCM and a Tanner graph.

In decoding LDPC codes, the message-passing algorithm is most commonly used because it has been proved to provide very good decoding capabilities [8–10]. During the decoding process, messages are passed from the bit nodes to the CNs along the connections, and vice versa, iteratively. Referring to Figure 1, in evaluating the message passing from CN0 to BN0, we need to make use of the BN6-to-CN0 message and the BN7-to-CN0 message. Similarly, when evaluating the message passing from BN4 to CN3, we have to consider the message sent from CN2 to BN4. In other words, we need to consider all connections of each BN or CN repeatedly in decoding a LDPC code. Moreover, all such connections are represented by the nonzero elements in the PCM **H**, and most of the elements in **H** are 0's because of the sparse nature of **H**. Instead of storing **H** entry by entry, a more efficient method should be used to model **H**.

$$\mathbf{H} = \begin{array}{c|cccccccc|l} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & \text{BN} \\ \hline & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \text{ CN} \\ & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 2 \\ & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 3 \end{array}$$

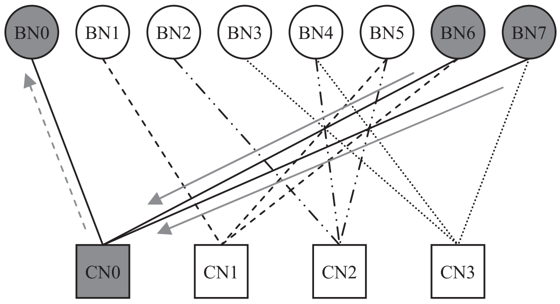

Figure 1. A parity-check matrix and its corresponding Tanner graph. BN, bit node; CN, check node.

In computer simulations, the linked-list method has been widely used to model the PCM **H**, or equivalently the structure of the Tanner graph, in the decoding process. A link can be considered as a pointer pointing to another object, and a linked list consists of objects embedded with these links. For the given PCM **H**, the linked list is constructed as follows. Referring to Figure 2 , each of the non-zero entries in **H** (a connection in the Tanner graph) is first represented by an object. Further, the corresponding BN-to-CN and CN-to-BN messages are stored in the object. Two objects are linked if they are next to each other in the same row or in the same column, and such link information is also stored in each of the linked objects. Thus, each object in the linked list contains four direction pointers (links) and two double-precision floating-point values (BN-to-CN and CN-to-BN messages). For a 64-bit computer system, each direction pointer is represented by an 8-byte word (also referred to as a qword), and each double-precision floating-point value requires 8 bytes for storage. Consequently, each object requires a memory of 48 bytes for storage. The total memory requirement for storing all the objects will be $48N_{\text{edge}}$ bytes where N_{edge} denotes the number of edges in the Tanner graph. Moreover, starting from one object, we can make use of the link information to access an adjacent object. Note that the link information in the objects of the linked list can be readily re-created for different PCMs **H** without affecting other parts of the simulation program.

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

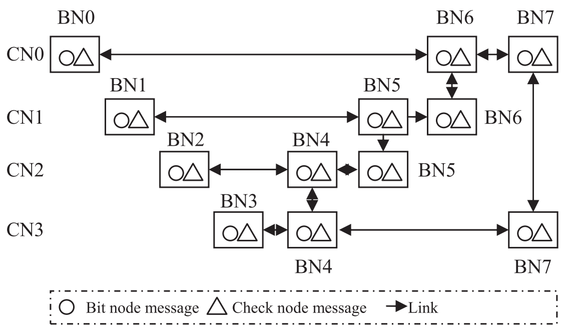

Figure 2. A linked-list approach to modeling a parity-check matrix. BN, bit node; CN, check node.

In today's computer architecture, a central processing unit (CPU) can access a built-in small-size L1 cache memory with little latency. It can further access a nearby L2 cache via a local bus with a little bit more latency. When the CPU fetches an instruction or some data, it will attempt to load the instruction or data from the L1 and L2 caches first. If the CPU fails to find the target instruction/data in the cache, a "cache miss" occurs, and the CPU will access the L3 cache[†] or even the random access memory (RAM), which requires a much longer latency compared with accessing the L1 and L2 caches. Thus, to optimize the speed of a computer program, one should minimize the CPU's need to access data in the L3 cache/RAM. However, the aforementioned linked-list approach is not optimized for memory access. In particular, if the decoding algorithm needs to fetch some data several rows down in the matrix, it is likely that such data are not readily stored in the cache and have to be fetched from the L3 cache/RAM. Consequently, the CPU has to spend much time fetching data from the L3 cache/RAM frequently.

In this paper, we propose compressing a PCM into two matrices in the check-node domain (CND) and in the bit-node domain (BND), respectively. Because the compressed matrices have much smaller sizes compared with the original PCM, the matrices and their associated data (BN-to-CN and CN-to-BN messages) can be readily stored within the capacity of the cache memory. Thus, during the iterative decoding process, it will be more likely that the computation of the updated messages can be completed based on the data in the cache memory alone. Consequently, the need for the CPU to access the L3 cache/RAM can be significantly reduced. Our whole idea is therefore to keep more data in the cache closer to the CPU.

Furthermore, it is known that the use of pointers can significantly improve the performance for repetitive operations in a computer program. Because the decoding of the LDPC codes involve repetitive operations, the use of pointers should help in improving the performance of the simulation program. Here, we will also apply the "pointer" approach to convert the updated messages associated with compressed matrices from the CND to the BND, and vice versa. The aim is to further improve the simulation time. Our simulation results show that compared with the linked-list approach, the proposed approach consumes a

[†]Note that there is no L3 cache in some computer architectures.

much shorter simulation time in decoding the LDPC codes. Moreover, the time reduction increases with the size of the LDPC-code length.

The organization of the paper is as follows. Section 2 describes the details of the proposed approach. In Section 3, we present our simulation results, and Section 4 provides the conclusions.

2. PROPOSED APPROACH

2.1. Compressed parity-check matrices

Given a file (with .pcm extension in our program) storing a PCM, we will first compress it into matrices in the CND and in the BND, respectively. Moreover, we will take the maximum row weight and the maximum column weight into consideration during the CND compression and the BND compression, respectively. Referring to the PCM in Figure 2, the number of rows (CNs) is 4, and the number of columns (BNs) equals 8. Because the maximum row weight equals 3, the compressed PCM (CPCM) in the CND (CPCM-CND) will be of size 4×3 (Figure 3). Similarly, the maximum column weight equals 2, and hence, the CPCM in the BND (CPCM-BND) will be of size 8×2 (Figure 4).

Referring to Figure 3, in the construction of the CPCM-CND, the (i,k)th entry equals $j-1$ if the (i,j)th entry in the original PCM is the kth non-zero entry in the ith row. For example, the (3,2)th entry in the CPCM-CND equals 4 because the (3,5)th entry in the original PCM is the second non-zero entry in the third row. Similarly, if the (i,j)th entry in the original PCM is the lth non-zero entry in the jth column, the (j,l)th entry in the CPCM-BND equals $i-1$, as shown in Figure 4. An example is the (6,2)th entry of the CPCM-BND. It has a value of 2 because the (3,6)th entry in the original PCM is the second non-zero entry in the sixth column. Finally, any unfilled entries in the CPCM-CND and CPCM-BND will be given a value of -1. The -1 value indicates a "no more connection" status and is used to prevent the computer program from accessing an incorrect memory address for data. It can be seen that in our examples, there are entries with value -1 in the CPCM-BND but not in the CPCM-CND.

Note that in our example, the CPCMs are not very much smaller compared with the original PCM. In practice, the

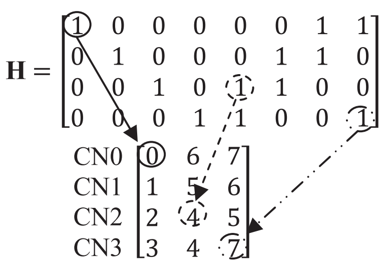

Figure 3. Formation of the compressed parity-check matrix in the check-node domain (CPCM-CND).

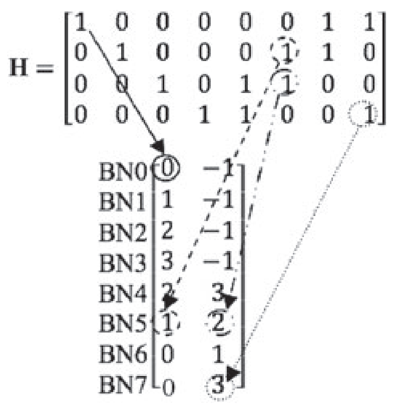

Figure 4. Formation of the compressed parity-check matrix in the bit-node domain (CPCM-BND).

PCM is of a much larger size and is very sparse. Thus, the CPCMs will be much smaller compared with the original PCM. Consequently, the CPCMs can be mainly stored in the L1/L2 cache memory, and the probability of a cache miss will be small. As the CPU does not need to access the L3 cache/RAM frequently, much simulation time can be saved.

2.2. Message matrices

Two message matrices (MMs), one in the CND and another in the BND, will be created to store the messages of all CNs and BNs. The MM in the CND (MM-CND) will be of the same size as the CPCM-CND, whereas the MM in the BND (MM-BND) will be of the same size as the CPCM-BND.

2.3. Domain conversion matrices

During the iterative decoding process, we need to convert the updated messages from one domain to another, that is, from CND to BND and vice versa. Here, we apply the pointer approach to perform such conversions. Based on Figures 3 and 4, the CND-to-BND conversion matrix and the BND-to-CND conversion matrix are readily obtained as in Figure 5. In fact, the CND-to-BND conversion matrix is formed by adding an extra entity to each of the entry in the CPCM-CND, whereas the BND-to-CND conversion matrix is formed by adding an extra entity to each of the entry in the CPCM-BND. For example, the (4,3)th entry in the CPCM-CND equals 7, implying that it corresponds to BN7, that is, the eighth row of the CPCM-BND. However, there are more than one entries in that row. Thus, we need to add an extra entity 1 (shown in parentheses in the CND-to-BND conversion matrix) to the (4,3)th entry in the CPCM-CND in order to form the CND-to-BND conversion matrix, indicating that this entry is pointing to the (8,1+1)th entry, that is, the (8,2)th entry of the

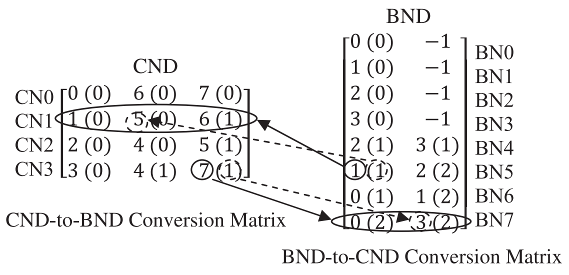

Figure 5. Formation of the CND-to-BND conversion matrix and the BND-to-CND conversion matrix. BND, bit-node domain; CND, check-node domain.

CPCM-BND. Similarly, the (6,1)th entry of the CPCM-BND equals 1. A simple study shows that this entry should be given an offset value of 1 in the BND-to-CND conversion matrix, showing that this entry is pointing to the (2,2)th entry of the CPCM-CND.

Note that all entries in the conversion matrices are given as offset values (i.e., with reference to the first column and the first row), and with the pointer approach, the CPU can locate the target data much faster.

2.4. Memory requirement

Because 8 bytes are needed to store a double-precision floating-point message, $8N_{\text{edge}}$ bytes are required to store each of the two MMs—one in the CND and another in the BND. Furthermore, we use 8 bytes to store an element (the address offset of the memory) in each domain conversion matrix. Each conversion matrix (CND-to-BND conversion matrix or BND-to-CND conversion matrix) consumes a memory of $8N_{\text{edge}}$ bytes. The total memory usage for storing the two MMs and the two domain conversion matrices is therefore equal to $32N_{\text{edge}}$ bytes.

2.5. Simulation program

The simulation program [11] begins by reading the parity check matrix file with .pcm extension to form the CPCM-CND, CPCM-BND, CND-to-BND conversion matrix, and BND-to-CND conversion matrix. Then, a codeword corrupted by additive white Gaussian noise is generated. Based on the corrupted signal vector, the initial probability ratio for each bit node is evaluated. Using the PCM shown in Figures 1 to 4 as an example, we denote the probability ratios for different bit nodes by symbols A to H, as shown in Table I. Based on these initial messages, the MM in the BND (MM-BND) is formed, as shown in Figure 6.

Table I. Initial probability ratios of the bit nodes.

Bit-node number	0	1	2	3	4	5	6	7
Initial probability ratio	A	B	C	D	E	F	G	H

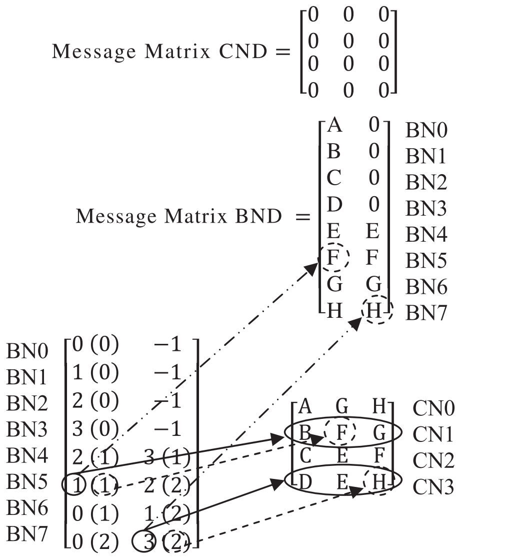

Figure 6. Initialization step and conversion of messages from bit-node domain (BND) to check-node domain (CND).

Using the BND-to-CND conversion matrix, the entries in the MM-BND are copied appropriately to entries in the MM in the CND (MM-CND), which contains all zeros initially, as illustrated in Figure 6. Table II shows the memory allocation of the MM-CND. Note that the entries in the MM-CND are located consecutively such that the CPU can access and update them more efficiently. During the first half of the iteration, the messages of the CNs are updated using the standard sum–product algorithm [8] and are shown as symbols W to Z in Table II. Using the CND-to-BND conversion matrix, the entries in the MM-CND are then used to update the entries in the MM-BND, as illustrated in Figure 7. Table III shows the consecutive memory allocation of the MM-BND. As mentioned earlier, the aim is to allow the CPU to access and update them more efficiently. During the second half of the iteration, the messages of the bit nodes are updated using the standard sum–product algorithm and are shown as symbols I to P in Tables III and IV.

At the end of the iteration, all bit-node symbols will be estimated using hard decisions. If the decoded codeword is a valid one, the iteration will stop. Otherwise, another iteration will start by using the updated bit-node messages. If a valid codeword is not found within a specified number of iterations, say 50 iterations, the iteration will also stop, and the decoding is declared as a failure. Then, the next corrupted codeword will be generated and decoded until a termination condition is satisfied; for example, 10 000 corrupted codewords have been decoded.

3. RESULTS AND DISCUSSIONS

To demonstrate the performance of the proposed method, a LDPC decoder has been simulated under the × 64 environment shown in Table V. Five different LDPC codes, as

Table II. Memory allocation of the message matrix in the check-node domain (MM-CND).

	CN0			CN1			CN2			CN3		
MAO	0	1	2	3	4	5	6	7	8	9	10	11
BN no.	0	6	7	1	5	6	2	4	5	3	4	7
Value	A	G	H	B	F	G	C	E	F	D	E	H
After 1st-half iteration		W			X			Y			Z	

BN, bit node; CN, check node; MAO, message address offset.

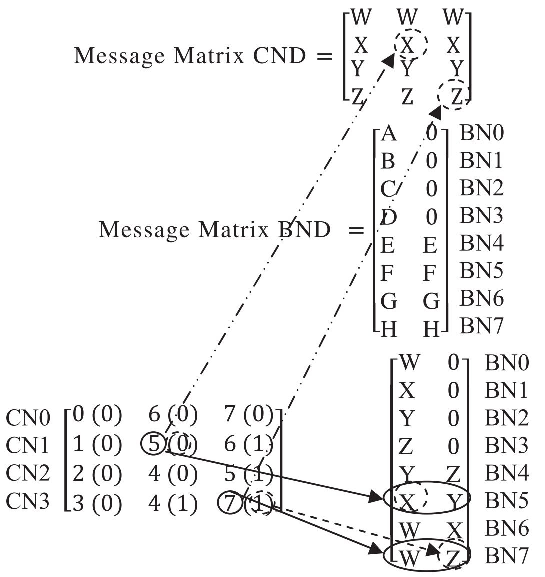

Figure 7. Conversion of messages from check-node domain (CND) to bit-node domain (BND).

Table IV. Updated messages of the bit-node messages after the first iteration.

Bit-node number	0	1	2	3	4	5	6	7
Updated probability ratio	I	J	K	L	M	N	O	P

Table V. Configurations of the computer hardware and software.

CPU	Core i7 920
Clock speed	2.67 GHz
L1 cache for each core	32 KB (data) and 32 KB (instruction)
L2 cache for each core	256 KB
L3 cache shared among cores	8 MB
RAM	6 GB
Memory speed	1.6 GHz
Programming language	C
OS	Windows 7 Enterprise 64 bits

CPU, central processing unit; OS, operating system; RAM, random access memory.

shown in Table VI, have been used in our simulations. They all have a code rate of 0.5, whereas the code-length ranges from 1008 to 10^6. Moreover, they are constructed with the progressive-edge-growth algorithm [12]. The maximum number of iterations taken to decode each codeword equals 200. If a valid codeword is not found at the end of 200 iterations, a decoding failure (block error) will be declared.

In Table VII, we show the simulated block error rates (BLERs) using the linked-list approach and our proposed CPCM approach. It can be seen that both approaches give the same BLERs. For each signal-to-noise ratio (SNR), the simulation continues until 100 block errors are found. For Codes I to IV, SNRs of 0.5 dB to 0.9 dB in steps of 0.1 dB are used, whereas for Code V, SNRs of 0.5 dB to 0.7 dB in steps of 0.1 dB are used. For Code V having a SNR of 0.8 dB, no block errors have been detected over 1000 sent codewords. However, the time taken is already too long, so we decide to stop the simulations after decoding the 1000 codewords.

Table VIII compares the time consumed per iteration for the linked-list approach and our CPCM approach. The results show that the proposed CPCM approach takes only 24.2% to 64.4% of the time required by the linked-list approach for each iteration. In other words, the new approach can improve the simulation time significantly, with a reduction of over 35% when the code length is 1008 to a reduction of more than 75% when the code-length is 10^6. The improvement of the proposed approach over the linked-list approach can be analyzed as follows.

Table III. Memory allocation of the message matrix in the bit-node domain (MM-BND).

	BN0		BN1		BN2		BN3		BN4		BN5		BN6		BN7	
MAO	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
CN no.	0	−1	1	−1	2	−1	3	−1	2	3	1	2	0	1	0	3
Value	W	0	X	0	Y	0	Z	0	Y	Z	X	Y	W	X	W	Z
After 2nd-half iteration	I		J		K		L		M		N		O		P	

BN, bit node; CN, check-node; MAO, message address offset.

Table VI. Types of rate—0.5 low-density parity-check codes used in the simulations and the total memory storage requirement.

Code	Type	Size	No. of edges N_{edge}	Total memory requirement $= 48N_{\text{edge}}$ bytes (linked-list method)	Total memory requirement $= 32N_{\text{edge}}$ bytes (proposed method)
I	Irregular	1008×504	3716	174 KB	116 KB
II	Irregular	2304×1152	7200	337 KB	225 KB
III	Irregular	$10,008 \times 5004$	37,214	1744 KB	1163 KB
IV	Regular	$10^5 \times 5 \cdot 10^4$	$3 \cdot 10^5$	14 MB	9.4 MB
V	Irregular	$10^6 \times 5 \cdot 10^5$	3,305,703	155 MB	103 MB

Table VII. Simulated block error rate (BLER) of the low-density parity-check codes using the linked-list approach and the compressed parity-check matrix (CPCM) approach. Both the linked-list approach and the CPCM approach produce the same BLERs.

Code	SNR = 0.5 dB	SNR = 0.6 dB	SNR = 0.7 dB	SNR = 0.8 dB	SNR = 0.9 dB
I	0.877	0.725	0.685	0.592	0.435
II	0.962	0.943	0.877	0.690	0.546
III	0.980	0.901	0.667	0.329	0.106
IV	1.000	1.000	1.000	1.000	1.000
V	1.000	1.000	0.901	No errors found	NA

NA, not applicable; SNR, signal-to-noise ratio.

Table VIII. Average time consumed per iteration in μs for the LDPC codes. Results in bracket show the time taken by the CPCMs approach when normalized by the time taken by the Linked-list approach.

Code	Linked-list approach (microseconds)	CPCMs Approach (microseconds)
I	188	121(64.4%)
II	375	229(61.1%)
III	2248	1191(53.0%)
IV	30,440	9548(31.4%)
V	439,110	106,095(24.2%)

Table VI shows the total memory requirement under the linked-list approach and the proposed CPCM approach for the five given codes. In all cases, the proposed method requires a smaller amount of memory, which implies a smaller probability of a "cache miss". We then consider Codes I, II, and III. Table VI shows that the total memory required to store the objects in the linked-list approach ranges from 174 KB to 1744 KB, whereas the total memory for the proposed CPCM approach ranges from 116 KB to 1163 KB. Because L2 cache has a size of only 256 KB, it will not be able to store all the instructions and data to be used by the CPU. Yet, in these cases, the objects/data can be readily stored in L3 cache, which possesses an 8 MB capacity. Whenever an instruction or data needed by the CPU is not available in L1 and L2 cache, a "cache miss" occurs.[‡] Then, the L2 cache has to fetch instructions/data from the L3 cache/RAM. Suppose the CPU requires some data, which are stored in the L3 cache. According to the specifications of Core i7, L2 cache will then collect a total of 64 bytes of data from L3 cache including the required data. As each object in the linked list has a size of 48 bytes, the L2 cache will be able to load one object from the L3 cache at one time. However, using the proposed CPCM approach, the L2 cache will be able to load eight consecutive double-precision floating-point messages from the L3 cache every time because each message in the MMs is of size 8 bytes. Because these consecutive messages will be used by the CPU in the following cycles, it is guaranteed that "cache miss" will not occur in the next few steps. Consequently, the proposed CPCM approach will reduce the occurrences of "cache miss" significantly. As each "cache miss" is followed by a number of clock cycles spent to fetch the data from the L3 cache, the proposed CPCM approach reduces the simulation time by reducing the overall number of cache misses. Furthermore, as the total memory used to store the objects in the linked-list approach increases from 174 KB (for Code I) to 337 KB (for Code II) and to 1744 KB (for Code III), cache misses will occur more frequently. Table VIII indicates that using the CPCM approach saves an increasing proportion of time for Code I to Code II and to Code III because (i) the memory requirement for the CPCM approach is smaller and (ii) the CPCM approach will reduce the occurrences of cache miss by fetching eight consecutive double-precision floating-point messages from L3 cache at each time.

Finally, we consider Codes IV and V, which require a total memory of 14 MB and 155 MB, respectively, in the linked-list approach. As the memory exceeds the capacity of L3 cache, the objects/data will have to be stored in the RAM. Same as that discussed earlier and compared with the linked-list approach, the CPCM approach will allow more consecutive double-precision floating-point

[‡]A larger L1 and L2 cache will reduce the occurrence of a "cache miss", and the simulation time will be shorter.

messages to be loaded from the RAM to the L2/L3 cache at each time. In addition, the time taken to load data from RAM is longer compared with loading data from L3 cache. Thus, we can observe an even higher reduction in simulation time by using the CPCM approach when the memory used to store the objects in the linked-list approach increases from 1744 KB (for Code III) to 14 MB (for Code IV) and to 155 MB (for Code V).

4. CONCLUSIONS

This paper proposes a new approach based on CPCM to simulate a LDPC decoder. The method can significantly reduce the need of the CPU to access L3 cache or the main memory, and hence, a large amount of time can be saved during the simulations. Results show that compared with the commonly used linked-list approach, the new approach provides substantial speed-up improvements, ranging from 35% to 75% reduction in computation time. Moreover, the speed-up improvement becomes more remarkable as the code-length increases. The simulation program is readily downloaded from [11] and is free to use.

Note that in the case of field-programmable gate array (FPGA)/ application-specific integrated circuit (ASIC) implementation of a LDPC decoder, the memories will not be categorized; that is, there will be no L1/L2/L3 cache and RAM, and nothing similar to "cache miss" will occur. Moreover, the memory requirement is much more stringent than in computer simulations. Thus, more modifications to the proposed CPCM approach have to be done before it can be implemented in FPGA/ASIC. For example, we have to define the exact memory location where each message will be retrieved/stored in the FPGA/ASIC implementation. In addition to memory, other considerations include complexity (the number of logic gates), throughput, error performance, and latency will have to be taken into consideration when implementing a LDPC decoder with FPGA/ASIC.

ACKNOWLEDGEMENTS

The work described in this paper was partially supported by a grant from the Research Grants Council of the Hong Kong SAR, China (project no. PolyU 521809) and by the National Natural Science Foundation of China (grant no. 60972037).

REFERENCES

1. Gallager R. Low-density parity-check codes. *IEEE Transactions on Information Theory* 1962; **8**(1): 21–28.
2. MacKay DJC, Neal RM. Near Shannon limit performance of low density parity check codes. *Electronics Letters* 1996; **32**(18): 1645–1646.
3. IEEE. IEEE standard for local metropolitan area networks. Part 16: Air interface for fixed and mobile broadband wireless access systems. Amendment 2: physical and medium access control layers for combined fixed and mobile operation in licensed bands. In *IEEE Standard 802.16e-2005*. IEEE Press: New York, NY, 2006.
4. IEEE. Draft IEEE standard for local metropolitan networks-specific requirements. Part 11: wireless LAN medium access control (MAC), and physical layer (PHY) specifications: enhancements for higher throughput. In *IEEE P802.11n/D1.0*. IEEE Press: New York, NY, 2006.
5. Ryan WE. An introduction to LDPC codes. In *Coding and Signal Processing for Magnetic Recording Systems*, Vasic B, Kurtas EM (eds). CRC Press, 2005, Chapter 36.
6. Shokrollahi A. LDPC codes: an introduction. In *Coding, Cryptography and Combinatorics*, Feng K, Niederreiter H, Xing C (eds). Birkhäuser, Boston, 2004; 85–112.
7. Richardson TJ, Urbanke RL. The capacity of low-density parity-check codes under message-passing decoding. *IEEE Transactions on Information Theory* 2001; **47**(2): 599–618.
8. Richardson T, Urbanke R. *Modern Coding Theory*. Cambridge University Press: Cambridge, 2008.
9. Zheng X, Lau FCM, Tse CK, He Y, Hau SF. Application of complex-network theories to the design of short-length LDPC codes. *IET Communications* 2009; **3**(10): 1569–1577.
10. Zheng X, Lau FCM, Tse CK. Constructing short-length irregular LDPC codes with low error floor. *IEEE Transactions on Communications* Oct 2010; **58**(10): 2823–2834.
11. Yau SF, Wong TL, Lau FCM. *Fast simulation program for LDPC codes*, 2010. Available at http://francis.eie.polyu.edu.hk/.
12. Hu XY, Eleftheriou E, Arnold DM. Regular and irregular progressive edge-growth Tanner graphs. *IEEE Transactions on Information Theory* 2005; **51**(1): 386–398.

AUTHORS' BIOGRAPHIES

Shek F. Yau obtained his BEng (Hons) degree in Electronic and Information Engineering from Hong Kong Polytechnic University, Hong Kong in 2010. He is now working in the industry.

Tan L. Wong obtained his BEng (Hons) degree in Electronic and Information Engineering from Hong Kong Polytechnic University, Hong Kong in 2010. He is now working in the industry.

Francis C. M. Lau received the BEng (Hons) degree with first class honors in Electrical and Electronic Engineering and the PhD degree from King's College London and University of London, UK, in 1989 and 1993, respectively.

He is a professor and associate head at the Department of Electronic and Information Engineering, The Hong Kong Polytechnic University, Hong Kong. He has published over 200 papers, including over 80 journal papers and over 120 conference papers. He is also the co-author of *Chaos-based Digital Communication Systems* (Heidelberg: Springer-Verlag, 2003) and *Digital Communications with Chaos: Multiple Access Techniques and Performance Evaluation* (Oxford: Elsevier, 2007). He is a co-holder of two granted US patents and one pending US patent. His main research interests include channel coding, applications of complex-network theories, cooperative networks, wireless sensor networks, chaos-based digital communications, and wireless communications.

He served as an associate editor for *IEEE Transactions on Circuits and Systems II* in 2004–2005 and *IEEE Transactions on Circuits and Systems I* in 2006–2007. He was also an associate editor of *Dynamics of Continuous, Discrete and Impulsive Systems, Series B* from 2004 to 2007 and was a co-guest editor of *Circuits, Systems and Signal Processing* for the special issue "Applications of Chaos in Communications" in 2005. He is currently serving as a guest associate editor of *International Journal of Bifurcation and Chaos*.

Yejun He received his PhD degree in Information and Communication Engineering from Huazhong University of Science and Technology (HUST) in 2005, MS degree in Communication and Information System from Wuhan University of Technology (WHUT) in 2002, and his BS degree from Huazhong University of Science and Technology in 1994. From September 2005 to March 2006, he was a research associate at the Department of Electronic and Information Engineering, The Hong Kong Polytechnic University. From April 2006 to March 2007, he was a research associate at the Department of Electronic Engineering, Faculty of Engineering, The Chinese University of Hong Kong.

Dr. He is currently an Associate Professor at Shenzhen University, China. His research interests include channel coding and modulation, multiple-input multiple-output orthogonal frequency division multiplexing wireless communication, space-time processing, smart antennas, and so on. Dr. He is a senior member of IEEE, a senior member of China Institute of Communications. He is also serving/served as a reviewer/technical program committee member/session chair for various journals and conferences, including *IEEE Transactions on Vehicular Technology, IEEE Communications Letters, International Journal of Communication Systems*, IEEE VTC (Springs of 2008 and 2009), IEEE ICCCAS (2007, 2008, and 2009), WRI CMC2009, ICST ChinaCom (2009 and 2010), APCC (2009 and 2010), ACM IWCMC2010, IEEE WiMob2009, and IEEE CSE2010. He served as the organizing committee vice chair of the 2010 International Conference on Communications and Mobile Computing (CMC2010) and an editor of the CMC2010 Proceedings. He is acting as the publicity chair of the 2011 International Conference on Communications and Mobile Computing (CMC2011).