Research Article

Fast Parallel Implementation for Random Network Coding on Embedded Sensor Nodes

Seong-Min Choi,1 Kyogu Lee,2 and Joon-Sang Park1

1 Computer Engineering Department, Hongik University, Seoul 121-791, Republic of Korea
2 Department of Transdisciplinary Studies, Seoul National University, Seoul 151-742, Republic of Korea

Correspondence should be addressed to Joon-Sang Park; jsp@hongik.ac.kr

Received 30 August 2013; Accepted 4 January 2014; Published 18 February 2014

Academic Editor: Hwa-Young Jeong

Network coding is becoming essential part of network systems since it enhances system performance in various ways. To take full advantage of network coding, however, it is vital to guarantee low latency in the decoding process and thus parallelization of random network coding has drawn broad attention from the network coding community. In this paper, we investigate the problem of parallelizing random network coding for embedded sensor systems with multicore processors. Recently, general purpose graphics processing unit (GPGPU) technology has paved the way for parallelizing random network coding; however, it is not an option on embedded sensor nodes without GPUs and thus it is indispensable to leverage multicore processors which are becoming more common in embedded sensor nodes. We propose a novel random network coding parallelization technique that can fully exploit multicore processors. In our experiments, our parallel method exhibits over 150% throughput enhancement compared to existing state-of-the-art implementations on an embedded system.

1. Introduction

Network coding [1], especially random network coding [2], is known to enhance various performance metrics in computer networks. For example, random network coding can increase the multicast throughput [2] and reduce the file downloading delay in peer-to-peer (P2P) file sharing systems [3]. In P2P file sharing systems, a file is partitioned into multiple pieces each of which can be downloaded independently from many different peers. If a node downloads multiple pieces simultaneously from multiple peers, it dramatically reduces downloading delay but the node has to carefully select peers to communicate with and pieces to download since it has a big impact on the overall performance. When using random network coding, such piece selection problem is mitigated since the data are encoded into blocks such that all the blocks are equally important and thus the collecting node is only supposed to gather a specific number of equally important blocks from other peers. To take full advantage of random network coding, however, it is vital to guarantee low latency in the encoding/decoding process. When using random network coding, the data has to be encoded before being transferred at the sending node and the received data at a destination has to be decoded to recover the original data. The decoding process of random network coding is implemented as a variation of the Gaussian elimination. Its complexity, $O(n^3)$, where $n$ is the number of blocks comprising a file, is quite expensive especially when the file size is large. The time overhead spent for decoding would cancel out all the benefits of reduced transmission time in random network coding. Thus, it is critical to guarantee short decoding latency when implementing random network coding in practice.

A number of works have studied on reducing the decoding latency of random network coding. Parallelized decoding techniques for multicore processors have been proposed in [4, 5]. Also, it has been shown that parallel decoding using general purpose graphics processing unit (GPGPU) such as compute unified device architecture (CUDA) [6] can radically enhance the decoding speed of network coding [7–10]. However, on embedded systems such as sensor network nodes, it is usually not possible to leverage GPGPU technology since GPUs are not presented in such systems.
and thus full exploitation of CPU’s capability is the only way to improve system performance. In this paper, we focus on enhancing random network coding performance by taking full advantage of CPUs. Due to the proliferation of multicore architectures in these days, multicore CPUs and microcontrollers are becoming more common in embedded systems. Thus, we propose a novel random network coding parallelization technique for multicore embedded processors. Our parallelization technique further enhances the decoding speed of random network coding by over 150% compared to existing state-of-the-art parallel decoding schemes. The enhancement is achieved through maximizing concurrency of multiple threads in the decoding process of random network coding.

The rest of paper is organized as follows. Section 2 provides background information including related works. Section 3 details our novel parallel decoding method for random network coding. Section 4 shows the performance advantage of our proposed scheme, and finally we conclude this paper in Section 5.

2. Background

In this section, we give a brief introduction to random network coding technique and then discuss related works.

2.1. Random Network Coding. To transfer a set of data such as a single file using random network coding, the source node generates a set of coded packets from the original file and transmits them towards destination nodes. To this end, the data at the source is first divided into a number of blocks. We use \( p_k \) to denote \( k \)th block. A coded packet \( c_i \) is a linear combination of the original blocks. That is, \( c_i = \sum_{k=1}^{n} e_{ik} p_k \), where \( n \) is the number of blocks and the coefficient \( e_{ik} \) is a certain element randomly chosen in a certain Galois field \( GF \) [2]. The coded packet \( c_i \) is transmitted towards destination nodes along with the coefficient vector \( e_i = [e_{i1}, ..., e_{in}] \) describing how the coded packet is generated and stored in the header [11]. The concatenation of a coded packet and its coefficient vector is referred to as transfer unit. On reception of transfer units, intermediate nodes on the path to a destination generate a linear combination of received coded packets and send them to downstream nodes. For a destination to be able to decode a set of received coded packets and send them to downstream nodes. For a destination to be able to decode a set of received coded packets and recover the original file, it needs to collect at least \( n \) coded packets with coefficient vectors linearly independent to each other. Let us say that a destination has collected \( n \) coded packets, \( c_1, ..., c_n \), and let \( E^T = [e_1^T, ..., e_n^T] \), \( C^T = [c_1^T, ..., c_n^T] \), and \( P^T = [p_1^T, ..., p_n^T] \), where superscript \( T \) denotes the transpose operation. As the relationships among \( C, E, \) and \( P \) can be expressed as \( C = EP \), the destination can recover the original file \( P \) by multiplying the inverse of \( E \) to \( C \) assuming that \( E \) is invertible; that is, the linear independence among all coefficient vectors \( e_{ik} \)'s is preserved. In random network coding, to calculate \( P = E^{-1}C \), a variant of the Gaussian Elimination (GE) called progressive decoding is widely used. The conventional GE requires collecting \( n \) transfer units and having the \( n \times n \) coefficient matrix before starting the process. However, in most random network coding scenarios, waiting for the entire matrix to be formed is not an optimal solution. Packets are delivered one by one and the time gap between the arrivals of the first transfer unit and the last one can be very large. Thus, instead of waiting for all the transfer units to arrive, partial decoding can be performed on reception of each transfer unit and hence the name of progressive decoding [4]. A simple description of progressive decoding borrowed from [4] is presented in Table 1. Note that received transfer units containing coefficient vectors and coded data blocks should be organized as an augmented matrix in which a transfer unit constitutes a row such that the progressive decoding/GE can be run on the matrix. On each transfer unit's arrival, a new row is inserted in the augmented matrix and then the procedures from Stage A to Stage E are applied to the augmented matrix in order to obtain the reduced row-echelon form of the augmented matrix at the end.

2.2. Related Work. Network coding is generally attributed to Ahlswede et al. [1], who showed the utility of the network coding for multicast. The work of Ahlswede et al. was followed by other works by Koetter and Médard [12] that showed that codes with a simple, linear structure were sufficient to achieve the capacity of multicast connections in wireline networks. This result was augmented by Ho et al. [2], who showed that a random construction of the linear codes, that is, random network coding, was sufficient. Chou et al. [11] proposed a practical way of implementing random network coding: network codes are carried along with packets. Gkantsidis and Rodriguez [3] have shown that random network coding is beneficial in large scale P2P systems. Random network coding has been known as helpful technique for smooth, fast downloads, efficient server utilization [13], and mobile P2P system [14]. Park et al. showed improvements in the reliability of ad hoc network systems [15].

To solve the high decoding latency problem in random network coding, several approaches have been proposed. In [16], a variant of random network coding called pipeline network coding (PNC) has been proposed. PNC reduces encoding/decoding delay by using a special form of encoding matrix. In [17], a new coding scheme with a lower computational complexity compared to the conventional random network coding has been proposed. Shojania and Li [4].

| Table 1: Tasks in progressive decoding [4]. |
|------------------|----------------------------------------------------------|
| Stage | Task description |
|-------|----------------------------------------------------------|
| A | Using the former coefficients rows, reduce the leading coefficients in the new coefficient row to 0 |
| B | Find the first nonzero coefficient in the new coefficient row |
| C | Check for linear independence with existing coefficient rows |
| D | Reduce the leading nonzero entry of the new row to 1, such that results in row-echelon form |
| E | Reduce the coefficient matrix to the reduced row-echelon form |
suggested a parallelized decoding technique for multicore CPUs with SIMD (single instruction multiple data) instructions, for example, Intel’s SSE. The decoding process in [4] is based on the well-known Gauss-Jordan elimination algorithm but it can progressively decode data on arrival of each partial data block. This distinctive feature, progressive decoding, reduces overall decoding latency when the arrivals of partial data blocks to be decoded span over a long period of time. Note that, however, the original data only can be recovered and decoding process can be finished at the arrival of the last data even though such progressive decoding is used. Conventional parallelized Gauss-Jordan elimination algorithms such as parallel adaptive Gauss-Jordan algorithm [18] and other related algorithms such as parallel matrix inversion [19] and parallel LU decomposition [20] require the entire data before starting the decoding process and thus will incur additional decoding latency on the receiver compared to the progressive decoding. Park et al. also have proposed efficient parallelized progressive network coding algorithm with dynamic partitioning algorithms for multicore CPUs [5]. Shojania et al. in [7] have proposed GPGPU based parallelized progressive decoding algorithm and in [8] parallel multisegment decoding algorithm for buffered data. In these GPGPU based parallelized schemes, hundreds of GPU threads can simultaneously encode and decode data blocks and thus can easily outperform multicore CPU based implementations. Chu et al. showed improved encoding throughput on GPU with an aid of CPU [9]; however, the decoding methodology used in [9] is not a progressive one as in [4].

3. Role Division Progressive Decoding: A Novel Parallelization Technique for Random Network Coding

In this section, we introduce our novel parallelization scheme for random network coding on embedded multicore processors. We start with reviewing current state-of-the-art parallelization techniques for random network coding on multicore processors.

Current state-of-the-art parallelization techniques [4, 5] for random network coding following the progressive approach divide each row in the augmented matrix into a number of equal-size chunks and allocate them to a number of threads. These “homogenous threads” perform the same decoding operation on different chunks in parallel. (Note that the same technique is used for the parallelization leveraging GPGPU [7–10]. Also note that some parallelization schemes, for example, [9], are not following the progressive approach.) The size of each partition in a row can be the same for every row [4] or dynamically chosen based on the row number [5] in the matrix. In [4], a static partitioning method called PPNC (parallelized progressive network coding) was suggested. As depicted in Figure 1, the way PPNC works is to partition the encoding vector matrix and the data matrix vertically into equal-size units and allocate each of them to each concurrent thread such that each unit can be decoded concurrently.

DVP (dynamic vertical partitioning) [5] was suggested in order to complement PPNC. The problem of PPNC is that it does not consider the load balancing issues among multiple threads/cores and thus the unbalanced workload hinders full exploitation of parallelization opportunities especially when the matrix size is large. Addressing this problem, DVP aims to balance the workload distribution among multiple threads via dynamic partitioning. As exemplified in Figure 2, the (vertical) partition size of each workload is dynamically chosen based on the number of rows to be processed. That is, the partition size reduces as the number of rows to be processed increases (or as the decoding process progresses). (Figure 2 is reproduction of Figure 8 in [5].)

The main problem of previous approaches described above is that there exist dependences among the threads. That is, all the threads must progress synchronously to each stage of the progressive decoding (Table 1). Although each thread is working on its own data chunk, the threads are not fully independent in the fact that synchronization among all the threads is inevitable at the end or at the beginning of each stage. Therefore, threads that have finished earlier must wait for others to be finished, wasting time while waiting. One solution to this dependence issue is to intentionally install redundancy in calculation. That is, the coefficient matrix is copied to each thread and let each thread run the progressive GE independently while decoding its part. Indeed, the redundancy in this approach, running the progressive GE on multiple copies of the coefficient matrix, limits the overall speed-up achieved via parallelization.

Addressing these problems, we propose a novel parallelization technique for random network coding called
role division progressive decoding (RDPD). RDPD aims at minimizing dependence among concurrent threads, that is, minimizing needs for synchronization among threads and/or redundant calculations, in the progressive decoding by using nonhomogeneous threads. In RDPD, two different types of threads, supervisor and worker, work on two independent matrices, the encoding vector matrix and the coded data block matrix; the supervisor thread runs the progressive GE on the encoding vector matrix and the worker threads work on the data matrix, that is, the collection of coded data blocks, governed by the supervisor thread. While conducting the progressive GE, the supervisor thread produces and stores into a work-queue a sequence of work orders each of which informs the worker threads how to manipulate rows in the data matrix. The sequence of work orders in fact is a decomposition of the progressive GE on a specific matrix into a set of atomic row operations. The content of a work order includes the type of operation, the multiplicand row (dividend row), the result row, and the multiplier (divisor). There are two types of operations: “subtraction after multiplication” and “division”. The first type of operations, subtraction after multiplication, defines the following calculation: the results row is updated by subtracting the multiplicand row multiplied by the multiplier from the result row. The second type of operations, division, is dividing the dividend row by the divisor and store it in the result row.

Figure 3 exemplifies the work orders generated by the supervisor thread when the third transfer unit arrives. In Stage A, it is required to subtract the first row and the second row multiplied by the first and second elements of the third row, respectively, from the third row. The supervisor thread produces two subtraction after multiplication work orders that have 3 and 8 as the multipliers, respectively. Stage D must generate a division work order that requires dividing the third row by 13 to make the leading value to 1. In general, the bigger the file size is, the more the row operations are required and work orders are produced. After Stage E, first and second rows must contain zero elements except the leading 1 so two subtraction after multiplication work orders subtracting the third row multiplied by 9 and 3 from the first
// On arrival of each transfer unit, run the following code as the supervisor thread
// Assume (i + 1)th transfer unit has been arrived
// (k + 1)th element of (j + 1)th encoding vector is stored in en_vector[j][k]
// G is the number of blocks comprising a file

Stage A
for j = 0 to i - 1 // enqueue subtraction after multiplication work orders
    Make_Work(j, i, en_vector[j][i], SUB_AFTER_MUL);
// first param is multiplicand row number, second is result row, third is weight
for j = 0 to i - 1 // manipulating coefficient vector matrix
    weight = en_vector[i][j];
    en_vector[i][j] = 0;
for k = j to G - 1
    temp = GaloisField_Multiplication(en_vector[j][k], weight);
    en_vector[i][k] ^= temp; // XORing for Galois field subtraction

Stage B and C
// Check for linear independence and find the first non-zero element
ptr = Find_First_Nonzero(en_vector[i]);

Stage D
Make_Work(i, i, en_vector[i][ptr], DIVISION); // enqueue a division work order
// first param is dividend row number, second is result row, third is weight
for k = ptr to G - 1
    en_vector[i][k] = GaloisField_Division(en_vector[i][k], weight);

Stage E
for j = 0 to i - 1 // enqueue subtraction after multiplication work orders
    Make_Work(i, j, en_vector[i][ptr], SUB_AFTER_MUL);
for j = 0 to i - 1 // manipulating coefficient vector matrix
    weight = en_vector[j][ptr];
    en_vector[j][ptr] = 0;
for k = ptr to G - 1
    en_vector[j][k] ^= GaloisField_Multiplication(en_vector[i][k], weight);

Algorithm 1: Pseudocode of supervisor thread.

and second rows, respectively, are produced. The work orders are inserted in the work-queue sequentially as they appear and the worker threads process them in the order in which they are enqueued. Pseudocode of the supervisor thread is presented in Algorithm 1.

Worker threads perform actual decoding of coded data blocks. Each worker thread performs calculations on a specific chuck of coded data blocks following the sequence of work orders produced by the supervisor thread. As shown in Figure 4, a coded data block (or each row in the coded data matrix) is divided into a set of nonoverlapping chucks of the same size each of which is allocated to a worker thread independently working on the specific region. To retain independency among threads, each thread maintains its own work-queue pointer pointing the first one of the uncompleted work orders to be processed by the thread in the work-queue such that the thread can be run on its own pace. Figure 5 illustrates a snapshot of the work-queue to which the work orders numbered 8 and 9 are about to be inserted.

In this example, the worker thread with ID 3 has to pick the work order number 2 (w2) from the work-queue and then forward its pointer to the next work order, w3. Also notice that the pointers are pointing different work orders. This means that each worker thread is working on its assignment on its own pace. Owing to this work-queue based interthread communication scheme, RDPD can minimize dependence among threads and maximize concurrency in the parallel progressive decoding. The supervisor thread needs not wait for worker threads and can work on newly arrived packets as soon as they arrive and worker threads need not wait until other workers progress to the same stage.

4. Performance Evaluation
In this section, we show RDPD’s performance advantage, up to 2.5x enhancement compared to existing schemes, via
experiments on an embedded sensor node. For the comparison purpose, we have implemented RDPD as well as the existing state-of-the-art parallel methods proposed in [4, 5] and performed experiments. Hereafter, we denote the scheme proposed in [4] as SP and the scheme proposed in [5] as DVP. We use a small form factor embedded system with Intel Atom D525 (dual-core, 1.6 GHz) as the experimental embedded sensor node platform. Also we use an Intel i7-960 CPU (quad-core, 3.2 GHz) based PC to see the performance behavior of RDPD on general-purpose computing platforms. Both systems are equipped with 4 G RAM and run Ubuntu 12.04. In the experiments, the data or file being decoded is divided into blocks and we refer to the number of blocks comprising a file as generation size. The total file size varies with the
generation size and block size used in the experiments. For example, if the generation size is 2048 and the block size is 16384 bytes, then the total data size is $2048 \times 16384 = 32$ Mbytes. The metrics used to compare the three schemes are decoding throughput (Mbytes/sec) which is calculated as the total size of data being decoded divided by the time spent for decoding. We use a look-up table based Galois field multiplication/division method.

Figure 6 shows the throughputs of the three schemes with the generation size varying from 16 to 512 and the block size varying from 1024 to 16384 bytes on the Atom-based sensor platform. In the figure, x-axis and y-axis represent the block size (in bytes) and throughput (in MB/sec), respectively. As we can notice in Figure 6(a), RDPD exhibits over 2.5 times higher throughput than SP and DVP when the generation size is 16 and the block size is 1024 bytes. Regardless of the generation size, RDPD shows at least 90% throughput enhancement over SP and DVP when block size is 1024. The performance advantage of RDPD compared to SP and DVP becomes less dominant as the generation size and block size get larger. With big generation sizes (e.g., 256 and 512) and big block sizes (e.g., 8 K and 16 K bytes), RDPD shows around 40% enhancement over SP and DVP. The performance degradation of SP and DVP compared to RDPD comes from the fact that they either perform redundant calculation or waste time while synchronizing among threads.

Figure 7 depicts the throughputs of the three schemes and also a sequential implementation of random network coding (denoted as SQ) on the i7 based platform with the generation size varying from 16 to 512 and the block size varying from 1024 to 8192 bytes. In the figure, RDPD shows around 1.7 times higher throughput than SP and DVP when the generation size is 16 and the block size is 1024 bytes. Compared to the sequential version, RDPD achieves
speed-up of 3.31 whereas DVP/SP only shows speed-up of 1.9, meaning that quad-core CPUs are underutilized with DVP/SP when the generation size is 16 and the block size is 1024 bytes. Similar to the Atom-based sensor platform, the performance advantage of RDPD compared to SP and DVP becomes less dominant on the i7-based PC as either the generation size or block size gets larger. When the generation size is 64 and the block size is 8192, RDPD shows around 15% enhancement over SP and DVP; however, the speed-up ratio of RDPD over the sequential implementation (SQ) is 3.62, meaning that the i7 quad-core CPU is utilized in nearly full capacity.

5. Conclusion

In this paper, we have investigated the problem of parallelizing random network coding for embedded sensor systems with multicore processors. Recently, GPGPU technology has paved the way for parallelizing random network coding but it is not an option on embedded sensor nodes since GPUs are not presented in such systems. We propose a novel random network coding parallelization scheme called role division progressive decoding (RDPD) that can fully exploit multicore processors. RDPD shows over 150% throughput enhancement compared to existing state-of-the-art implementations on an embedded sensor platform with Intel Atom processor and around 70% throughput enhancement on a high-end desktop processor, Intel i7. Future work includes investigating effects of the hyperthreading feature of i7 processors on the performance of parallel random network coding.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication this paper.

Acknowledgment

This work was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning (2013005876).

References

[1] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung, “Network information flow,” IEEE Transactions on Information Theory, vol. 46, no. 4, pp. 1204–1216, 2000.
[2] T. Ho, M. Médard, R. Koetter et al., “A random linear network coding approach to multicast,” IEEE Transactions on Information Theory, vol. 52, no. 10, pp. 4413–4430, 2006.
[3] C. Gkantsidis and P. R. Rodriguez, “Network coding for large scale content distribution,” in Proceedings of the IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE INFOCOM ’05), pp. 2235–2245, Miami, Fla, USA, March 2005.
[4] H. Shojania and B. Li, “Parallelized progressive network coding with hardware acceleration,” in Proceedings of the 15th IEEE International Workshop on Quality of Service (IWQoS ’07), pp. 47–55, June 2007.
[5] K. Park, J.-S. Park, and W. W. Ro, “On improving parallelized network coding with dynamic partitioning,” IEEE Transactions on Parallel and Distributed Systems, vol. 21, no. 11, pp. 1547–1560, 2010.
[6] NVIDIA, CUDA Toolkit, http://www.nvidia.com/content/cuda/cuda-toolkit.html.
[7] H. Shojania, B. Li, and X. Wang, “Nuclei: GPU-accelerated many-core network coding,” in Proceedings of the 28th Conference on Computer Communications (IEEE INFOCOM ’09), pp. 459–467, April 2009.
[8] H. Shojania and B. Li, “Pushing the envelope: extreme network coding on the GPU,” in Proceedings of the 29th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS ’09), pp. 490–499, June 2009.
[9] X. Chu, K. Zhao, and M. Wang, “Accelerating network coding on many-core GPUs and multi-core CPUs,” Journal of Communications, vol. 4, no. 11, pp. 902–909, 2009.
[10] S. Lee and W. W. Ro, “Accelerated network coding with dynamic stream decomposition on graphics processing unit,” Computer Journal, vol. 55, no. 1, pp. 21–34, 2012.
[11] P. A. Chou, Y. Wu, and K. Jain, “Practical network coding,” in Proceedings of Allerton Conference on Communication, Control, and Computing, 2003.
[12] R. Koetter and M. Médard, “An algebraic approach to network coding,” IEEE/ACM Transactions on Networking, vol. 11, no. 5, pp. 782–795, 2003.
[13] C. Gkantsidis, J. Miller, and P. Rodríguez, “Comprehensive view of a live network coding P2P system,” in Proceedings of the 6th ACM SIGCOMM on Internet Measurement Conference (IMC ’06), pp. 177–188, October 2006.
[14] U. Lee, J.-S. Park, S.-H. Lee, W. W. Ro, G. Pau, and M. Gerla, “Efficient peer-to-peer file sharing using network coding in MANET,” Journal of Communications and Networks, vol. 10, no. 4, pp. 422–429, 2008.
[15] J.-S. Park, M. Gerla, D. S. Lun, Y. Yi, and M. Médard, “CodeCast: a network-coding-based ad hoc multicast protocol,” IEEE Wireless Communications, vol. 13, no. 5, pp. 76–81, 2006.
[16] C.-C. Chen, C. Chen, S. Y. Oh, J.-S. Park, M. Gerla, and M. Y. Sanadidi, “ComboCoding: combined intra-/inter-flow network coding for TCP over disruptive MANETs,” Journal of Advanced Research, vol. 2, no. 3, pp. 241–252, 2011.
[17] P. Maymounkov, N. J. A. Harvey, and D. S. Lun, “Methods for efficient network coding,” in Proceedings of the 44th Annual Allerton Conference on Communication, Control, and Computing, 2006.
[18] N. Melab, E.-G. Talbi, and S. Petiton, “Parallel adaptive Gauss-Jordan algorithm,” Journal of Supercomputing, vol. 17, no. 2, pp. 167–185, 2000.
[19] L. Csanky, “Fast parallel matrix inversion algorithms,” in Proceedings of IEEE Symposium on Foundations of Computer Science, 1975.
[20] R. H. Bisseling and J. G. van de Vorst, “Parallel LU decomposition on a transputer network,” in Proceedings of the Shell Conference on Parallel Computing, 1989.