Network Coding — State of the Art

Florian Stamer, Jonas Andre*, Stephan Guenther*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: florian.stamer@tum.de, andre@in.tum.de, guenther@tum.de

***Abstract*—Network Coding (NC) [1] confers to intermediate nodes of a network the ability to combine packets via code. Instead of the traditional store-and-forward mechanisms of routing this new paradigm of store-code-forward mechanism has the potential to increase throughput, robustness against network loss, and security.**

In this paper we give an overview of recent advancements in network coding. We present an implementation of a Random Linear Network Coding (RLNC) data plane in P4 as introduced in [2]. Furthermore we focus on two optimization approaches for RLNC, with one being a novel Online Directed Acyclic Graph (DAG) algorithm [3] that tries to improve the decoding process and the other being an optimization to the encoding process of RLNC through the use of processor specific SIMD vector extensions [4]. By comparing the benefits of using the online DAG algorithm or SIMD vector extensions, we conclude the latter to be more practical since the online DAG algorithm is quite complex and only provides slightly better performance compared to already existing offline DAG algorithms.

***Index Terms*—Network Coding, Random Linear Network Coding, Directed Acyclic Graph, AVX, SIMD**

1. Introduction

In traditional networks packets are forwarded through store-and-forward mechanisms, but some networks can profit from combining packets to improve throughput. Ahlswede et al. [1] proposed the idea of combining packets via code and creating a store-code-forward mechanism called Network Coding (NC). This way inner nodes of a network can freely combine packets and provide the benefit of improved throughput [5], robustness against network loss [6], and security [7].

In Figure 1 we give an example of a butterfly network to illustrate how network coding can outperform traditional routing. The two source nodes (*server A/B*) transmit information *A* and *B*, respectively. Both must be received by the destination nodes (*PC 1/2*). With traditional routing only information *A* or *B* can be sent between the two switches at a time, thus both destination nodes do not receive all information at the same time. With network coding the information can be combined by a simple operation (*XOR*) and reconstructed at the destination nodes, in this case with antoher *XOR* operation.

The purpose of this paper is to give an overview of the current state of network coding and advances that have been made. We structure the paper as follows:

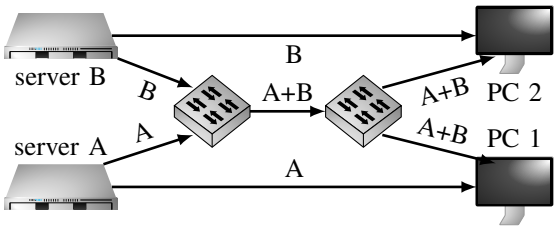

Figure 1: Butterfly Network

We first give a short summary of the network coding basics. We proceed to take a look at a P4 implementation of a Random Linear Network Coding (RLNC) data plane in Section 2. In Section 3 the focus lies on optimizing the RLNC decoding process through the usage of an Online Directed Acyclic Graph (DAG) algorithm. Another potential optimization to the performance of RLNC are Single Instruction Multiple Data (SIMD) vector extensions of certain processors. We take a deeper look at this idea in Section 4. We provide a comparison of the online DAG and SIMD vector extension approaches in Section 5 and give our conclusion as well as thoughts on future work in Section 6.

Basics of Network Coding. Network Coding introduces the ability to combine packets via code. In the case of Linear Network Coding (LNC) an encoded packet is created by linear combinations of N packets. In general the group of packets is referred to as a *generation* and the number of packets N is called the *generation size*. These linear combinations can be expressed as a matrix-vector multiplication over a given finite extension field. A finite field is a Galois field of the form $GF(2^n)$. For (random) linear network coding the finite extension fields $GF(2^1)$, $GF(2^2)$, $GF(2^4)$ and $GF(2^8)$ are of particular interest as mentioned in [4]. For convenience of notation we use F_q with $q = 2^n$ instead of $GF(2^n)$.

The following mathematical description of the encoding process is based on [4]. A packet with M symbols, each of size n, can be written as vector $\boldsymbol{a} = [a_1, a_2, \ldots, a_M]^T$ with the symbols $a_i \in F_q$. This gives for a generation of N packets the matrix

$$\boldsymbol{A} = [\boldsymbol{a}_1 \ldots \boldsymbol{a}_N] = \begin{bmatrix} a_{11} & \ldots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{M1} & \ldots & a_{MN} \end{bmatrix} \in F_q^{M \times N}. \quad (1)$$

An encoded packet $\boldsymbol{b}$ is generated by multiplication with

doi: 10.2313/NET-2021-05-1_13

an encoding vector $\boldsymbol{c} \in F_q^N$.

$$\boldsymbol{b} = \boldsymbol{A}\boldsymbol{c} = \sum_{i=1}^{N} c_i \boldsymbol{a}_i. \quad (2)$$

The components of $\boldsymbol{c}$ are chosen from the finite extension field F_q, either independently and identically distributed for RLNC or otherwise by some deterministic algorithm. In the given extension fields the addition operation is always a bit-wise XOR operation while the multiplication is a more complex modulo operation given a reduction polynomial specific to the field.

The decoding process is more complex and uses algorithms such as *Gauss-Jordan elemination* or *LU decomposition* on the recieved encoded packets.

2. An RLNC Implementation in P4

A recent implementation of a random linear network coding data plane in P4 has been proposed by Gonçalves et al. in [2]. By using RLNC, the network needs to be able to transmit additional information such as the encoding vectors. Thus a new packet format is designed to cope with this new way of packet processing. Since they use RLNC for their approach, the coefficients are chosen independently and uniformly at random from the finite field F_q. This provides the bonus of decentralizing code generation computation, but has the drawback that enough linearly independent *coded packets* are needed to decode the original message. We elaborate more on the packet format, the P4 program, and different methods of finite field multiplication in the following subsections.

2.1. Packet Format

The packet format of the generation-based RLNC protocol uses an inner and an outer header. Both are carried over Ethernet frames. The inner header contains the symbols and the coding vectors, if present, and carries information about the packet length as well as the type of packet. The types of packets are either *coded* or *uncoded*. The outer header holds information about meta parameters, such as the generation id, generation, finite field and symbol size.

2.2. RLNC P4 Program

The P4 PISA-like switch architecture buffers packets of different generations, which are limited in number per buffer. A generation stays buffered until enough packets of this generation are collected and the coding process starts. New linear combinations of the packets are transmitted until receiving an acknowledgment. Afterwards the buffer is flushed and starts accumulating packets of a new generation. The outer header of the newly coded packets stays the same while the inner header, i.e., the symbols and coding vectors, are readjusted.

2.3. Finite Field Multiplication

For the finite field arithmetic module the authors have featured two multiplication techniques. One is a compute-intensive method, based on simple shift and add operations, that results in an iterative algorithm which operates bit by bit. The other algorithm is based on pre-computed lookup tables, containing values for the log and $antilog$ of the elements in the finite extension field F_q.

3. Parallelization of the Decoding Process

While random linear network coding improves throughput, robustness against network loss, and security, it suffers from decoding delay since enough linearly independent packets have to arrive at the node before the decoding process can commence. This can be improved by using a *progressive* RLNC decoder that can partially decode a generation before all packets arrive. Based on progressive RLNC decoding Wunderlich et al. [3] proposes a novel strategy using directed acyclic graph scheduling. By arranging matrix block operations in a DAG manner, multiple operations are worked on in parallel by different threads. The novelty of this approach lies in it being an *Online* DAG algorithm, thus constructing the graph on the fly, instead of pre-computing, and making optimal use of progressive RLNC decoding. In the following subsections we give an explanation about the difference between non-progressive and progressive RLNC decoders, how the matrix block operations are defined and how the online DAG scheduling works, based on the information provided in [3].

3.1. Non-Progressive vs Progressive RLNC Decoder

There exist two categories of RLNC decoders, the non-progressive and progressive decoders. The classic non-progressive decoder expects all information to be present before starting the decoding process. An example is the generation-based RLNC approach as presented in [2], which we elaborate on in Section 2. Such a decoder can make use of matrix inversion algorithms other than Gauss-Jordan elimination, such as LU inversion.

A progressive RLNC decoder on the other hand can partially decode the data that has already been gathered and does not need to wait for the arrival of all data. While new encoded packets and coding vectors can be fed into the decoder as they are received. When considering conventional full-vector RLNC code, the decoded packets can only be released after the last packet of the generation is decoded. An optimization for such a progressive RLNC decoder is the use of low-delay codes, like sliding window codes [8, 9] or systematic generation based codes [10]. This way the decoder can already release any fully decoded information to upper levels without receiving all coded packets of a generation.

A hybrid scheme aims to combine the strengths of both non-progressive and progressive RLNC decoders. By performing sub-generation, more than one encoded packet, but less than the normal generation size, based progressive RLNC decoding [11].

3.2. Matrix Block Operations

A given matrix, of coding coefficients or encoded packets, gets split into blocks of size $b \times b$, where $b \leq N \wedge N/b \in \mathbb{N}$ and $16 \leq N \leq 1024$ is the generation

doi: 10.2313/NET-2021-05-1_13

size, e.g. a matrix of dimension 16×16 is split into four blocks of size 4×4. Each block gets separately processed by the three phases of the Gauss Jordan elimination with the use of helper matrices, hence the name *Matrix Block Operations*.

Given the number of symbols per packet M, the generation size N and the finite extension field F_q, let $\boldsymbol{C} \in F_q^{N \times N}$ be the coding coefficient matrix and $\boldsymbol{D} \in F_q^{N \times M}$ be the data matrix. Both are initially padded with zeros. $\boldsymbol{C'} \in F_q^{b \times N}$ and $\boldsymbol{D'} \in F_q^{b \times M}$ describe the encoded coding coefficient matrix and data matrix respectively, for the sub-generation of size b. In the following the 'row of a block *X*' is used to reference every block left and right of *X* spanning over the same rows and respectively 'column' for the blocks above and below of *X*. For a more in depth explanation and helpful figures we refer the reader to [3].

Forward Elimination. When a new sub-generation of encoded packets arrives, i.e. $\boldsymbol{C'}$ and $\boldsymbol{D'}$, the blocks in $\boldsymbol{C}$ containing the pivots on the diagonal are used to fill the corresponding blocks in $\boldsymbol{C'}$ with zeros. The row operations for a block are recorded in a helper matrix and applied to the other blocks in $\boldsymbol{C'}$ and to the blocks of $\boldsymbol{D'}$. Gaussian elimination is used on the first block in $\boldsymbol{C'}$ containing non zero values. The row operations are once more recorded in a helper matrix and applied to the rest of $\boldsymbol{C'}$ and $\boldsymbol{D'}$.

Backward Substitution. Continuing in this phase $\boldsymbol{C'}$ contains blocks of zeros followed by a block with the pivots on the diagonal. This block is used to fill the corresponding block *X* in $\boldsymbol{C}$ with zeros. The row operations are once again recorded in a helper matrix and applied to the row of *X* in $\boldsymbol{C}$ and $\boldsymbol{D}$. This process is repeated for every non zero block in the column of X.

Row Swapping. After the backward substitution concludes, both $\boldsymbol{C'}$ and $\boldsymbol{D'}$ are moved to the corresponding row in $\boldsymbol{C}$ and $\boldsymbol{D}$. If $\boldsymbol{C}$ is still missing pivots on the diagonal the algorithm starts once more after collecting enough new packets for a sub-generation. This is repeated until $\boldsymbol{C}$ is an identity matrix.

3.3. Online DAG Algorithm

New block operations are added on the fly to the Directed Acyclic Graph, instead of collecting all operations and constructing the entire DAG a priori (offline). This way the algorithm can take advantage of the properties of a progressive RLNC decoder. The iterative RLNC program is executed by a main thread. Block operations are added to the online DAG as new *task descriptions*, each summarizing the read and write dependencies regarding the other task descriptions. The main thread can then delegate tasks to a later time or another worker thread. The worker threads check independently for task descriptions in the DAG which have all of their dependencies resolved, pick and execute them.

Task Descriptions. These are objects in the DAG that hold information about the type of operation they represent, pointers to memory where matrices are stored, parameters describing said matrices (e.g. size) and more. Every object also has an access queue that keeps track of the sub tasks, that need to be concluded beforehand.

4. Encoding Process Optimization Using SIMD Vector Extensions

Contrary to optimizing the decoding process of (random) linear network coding, Günther et al. [4] try to improve the encoding process. In particular they implement and evaluate algorithms for finite field multiplication using processor specific vector instructions. For this study they implement two algorithms using the new family of instruction sets AVX512 in their finite field library *libmoepgf* [12]. AVX512 is a family of extensions and the two subsets AVX512-F (foundation) and AVX512-BW (byte and word) are the focus for the presented algorithms. While the byte-wise operations of AVX512-BW set of instructions are only supported by Intels Skylake-X and Ice Lake processors as of the time [4] was written, the AVX512-F extension will be supported by any processor that supports AVX512.

As we present in Section 1, the encoding process (2) expects the vectors $\boldsymbol{a}_i$ to be multiplied by the constant values c_i and finally accumulated into $\boldsymbol{b}$, this is commonly know as *multiply and add (madd)*. One such algorithm using vector instructions has been proposed by Plank et al. [13] and is called *shuffle* algorithm. This algorithm requires a shuffle instruction to swap words in vector registers. Hence the byte-wise operations of the AVX512-BW instruction set are necessary. Another algorithm introduced by Günther et al. in [12] is called *imul*. Contrary to the shuffle algorithm, it does not need any special instructions, but its complexity linearly depends on the word size. For the implementation this algorithm relies on the AVX512-F instructions set. Both algorithms are implemented for the finite expansion fields F_2, F_4, F_{16} and F_{256}.

Shuffle Algorithm . This algorithm expects the accumulator array $\boldsymbol{b}$, the source packet vector $\boldsymbol{a}_i$ and the coefficient c_i to calculate $\boldsymbol{b} := \boldsymbol{b} + \boldsymbol{a}_i \cdot c_i$. The shuffle algorithm needs certain constants, such as lookup tables and bit masks, different for each finite field F_q. Those are preloaded into the register variables. After handling the trivial cases for $c_i \in \{0, 1\}$, either no operation or a simple XOR, additional temporary registers are preloaded and the necessary madd operations using the shuffle instruction are performed.

Imul Algorithm. Similar to the previous algorithm, the imul algorithm expects the accumulator array $\boldsymbol{b}$, the source packet vector $\boldsymbol{a}_i$ and coefficient c_i. This algorithm also preloads lookup tables and after caching the trivial cases, sets up the temporary registers. One holds bit masks to isolate the coefficients of $\boldsymbol{a}_i$ and the other the powers of the constant c_i. These again are different depending on the finite field F_q. Afterward polynomial multiplication is performed inside a loop.

5. Evaluation and Comparison

In this Section we give a compact overview of the results and evaluations of both optimization approaches

doi: 10.2313/NET-2021-05-1_13

and put the results into perspective. It is important to mention, that the presented algorithms (online DAG vs shuffle/imul) try to optimize different processes (decoding vs encoding) and have been tested on different hardware.

Online DAG. The algorithm has been tested on both the ODROID-XU-3 and ODROID-XU+E, each equipped with four Cortex-A15 (big) cores and four Cortex-A7 (LITTLE) cores. The Cortex-A15 are clocked at 2.0GHz in XU-3 and 1.6GHz in XU+E, while the Cortex-A7 are clocked at 1.4GHz and 1.2GHz respectively [3].

While keeping the finite field (F_{256}) the same throughout the tests, a multitude of combinations for different values of the other parameters such as generation size, symbol size, number of threads and more are examined. This way optimal parameterization for high throughput and low delay are collected. For the benchmark the online DAG algorithm is compared to its offline DAG counter part and a state-of-the-art progressive *coefficient matrix duplication* (CD) approach [14]. The online DAG approach performs in general similar to the conventional offline method, while resulting in slightly better performance for smaller generations sizes. This is expected to be the result of the computational complexity, that increases with growing generation size. This approach performs also better then the CD approach for small symbol size, while falling short when the symbol size is especially large.

AVX512 Instruction Set Extensions. The shuffle and imul algorithms are tested on a multitude of processors, but in this overview we only mention the ones that support AVX512. These are the Intel Xeon Gold 6130 (clocked at 3.7GHz), Silver 4116 (clocked at 3.0GHz) and D-2166NT (clocked at 3.0GHz) [4].

For the tests only a single core is used and the generation size is kept at 16, while different finite fields ($F_2, F_4, F_{16}, F_{256}$) get analyzed. The AVX512 implementations perform in general better then the AVX2 implementations, but when the packet size reaches the size of the L2 cache the throughput drops regardless of extension used. Even for the commonly used finite field F_{256} an average of roughly $30Gbit/s$ of throughput can be achieved.

Encoding vs Decoding Optimization. We compare and evaluate both optimizations based on their performance gain compared to existing methods, as well as their implementation complexity. The online DAG algorithm is a complex approach, needing support for matrix block operations and the DAG scheduling with custom task descriptions. It provides only slight performance improvements compared to the common offline DAG algorithms, with throughput measured in MiB/s. Contrary the AVX512 based shuffle and imul algorithm already have library implementations and perform better compared to the older AVX2 extension, with throughput measured in $Gbit/s$. The difference in the units of measurement are most likely linked to the higher computational complexity of the decoding process or the different hardware used. To improve the performance of an RLNC implementation, the optimization of the encoding process through the use of vector extensions should be prioritized.

6. Conclusion and future work

We provide an overview of current advances in network coding, with the focus on two approaches to improve and optimize random linear network coding. One uses a progressive RLNC online directed acyclic graph based algorithm to parallelize the decoding process. The other provides the shuffle and imul algorithm, which make use of AVX512 vector extensions to speed up the finite field multiplication of the encoding process. In Section 5 we provide a summary of the evaluations of both approaches and came to the conclusion, that the use of processor specific vector extensions yield better results with less complex algorithms. Thus the optimization of the encoding process is more appealing.

Different implementations of network coding protocols or other areas of network coding, such as network security, can be of interest and be the focus of future works.

References

[1] R. Ahlswede, N. Cai, S. . R. Li, and R. W. Yeung, "Network information flow," *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, 2000.

[2] D. Goncalves, S. Signorello, F. M. V. Ramos, and M. Medard, "Random linear network coding on programmable switches," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2019*, 2019.

[3] S. Wunderlich, F. H. P. Fitzek, and M. Reisslein, "Progressive Multicore RLNC Decoding with Online DAG Scheduling," *IEEE Access*, vol. 7, pp. 161 184–161 200, 2019.

[4] S. M. Günther, N. Appel, and G. Carle, "Galois Field Arithmetics for Linear Network Coding using AVX512 instruction set extensions," 2019.

[5] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Medard, and J. Crowcroft, "XORs in the air: Practical wireless network coding," *IEEE/ACM Transactions on Networking*, vol. 16, no. 3, pp. 497–510, 2008.

[6] D. S. Lun, M. Médard, R. Koetter, and M. Effros, "On coding for reliable communication over packet networks," *Physical Communication*, vol. 1, no. 1, pp. 3–20, 2008.

[7] L. Lima, M. Médard, and J. Barros, "Random linear network coding: A free cipher?" in *IEEE International Symposium on Information Theory - Proceedings*, 2007, pp. 546–550.

[8] S. Wunderlich, F. Gabriel, S. Pandi, F. H. P. Fitzek, and M. Reisslein, "Caterpillar RLNC (CRLNC): A Practical Finite Sliding Window RLNC Approach," *IEEE Access*, vol. 5, pp. 20 183–20 197, 2017.

[9] F. Gabriel, S. Wunderlich, S. Pandi, F. H. P. Fitzek, and M. Reisslein, "Caterpillar RLNC With Feedback (CRLNC-FB): Reducing Delay in Selective Repeat ARQ Through Coding," *IEEE Access*, vol. 6, pp. 44 787–44 802, 2018.

[10] D. E. Lucani, M. Médard, and M. Stojanovic, "Systematic network coding for time-division duplexing," pp. 2403–2407, 2010.

[11] M. Kim, K. Park, and W. W. Ro, "Benefits of using parallelized non-progressive network coding," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 293–305, 2013.

[12] S. M. Günther, M. Riemensberger, and W. Utschick, "Efficient GF arithmetic for linear network coding using hardware SIMD extensions," in *2014 International Symposium on Network Coding (NetCod)*, 2014.

[13] J. S. Plank, K. M. Greenan, and E. L. Miller, "Screaming fast galois field arithmetic using intel simd extensions," *USENIX Conference on File and Storage Technologies*, vol. 11, 2013.

[14] H. Shin and J.-S. Park, "Optimizing random network coding for multimedia content distribution over smartphones," *Multimedia Tools and Applications*, vol. 76, 10 2017.