Parallelized Progressive Network Coding With Hardware Acceleration

Hassan Shojania, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto
{*hassan, bli*}*@eecg.toronto.edu*

***Abstract*— The fundamental insight of network coding is that information to be transmitted from the source in a session can be *inferred*, or *decoded*, by the intended receivers, and does not have to be transmitted verbatim. It is a well known result that network coding may achieve better network throughput in certain multicast topologies; however, the practicality of network coding has been questioned, due to its high computational complexity. This paper represents the first attempt towards a high performance implementation of network coding. We first propose to implement progressive decoding with Gauss-Jordan elimination, such that blocks can be decoded as they are received. We then employ hardware acceleration with SSE2 and AltiVec SIMD vector instructions on x86 and PowerPC processors, respectively. We then use a careful threading design to take advantage of symmetric multiprocessor (SMP) systems and multi-core processors. The objective of this work is to explore the computational limits of network coding in off-the-shelf modern processors, and to provide a solid reference implementation to facilitate commercial deployment of network coding. Our high-performance implementation is packaged as a C++ class library, and runs in Linux, Mac OS X and Windows, in Intel, AMD and IBM PowerPC processor families. On a Dual dual-core PowerPC G5 2.5 GHz server, the coding bandwidth of our implementation is able to reach 43 MB/second with 64 blocks of 32 KB each, achieving speedup of 21 over the baseline implementation.**

***Index Terms*— Network coding, parallelization, random linear codes, hardware acceleration, SSE2, AltiVec.**

I. Introduction

First introduced by Ahlswede *et al.* [1] in information theory, *network coding* has received significant research attention in the networking community. The essence of network coding is to allow coding at intermediate nodes throughout the network topology between the source and the receivers, in multiple unicast or multicast sessions. The fundamental insight of network coding is that information to be transmitted from the source in a session can be *inferred*, or decoded, by the intended receivers, and does not have to be transmitted verbatim. It has also focused on the *coding* capabilities of intermediate nodes, in addition to forwarding and replicating incoming messages.

The pioneering work by Ahlswede *et al.* [1] and Koetter *et al.* [2] proves that, in a directed network with network coding support, a multicast rate is feasible if and only if it is feasible for a unicast from the sender to each receiver. Li *et al.* [3] has further proved that linear coding suffices in achieving the maximum rate. These results are significant in the sense that, with network coding, the cut-set capacity bounds of unicast flows from the source to each of the receivers can be achieved in a multicast session. In other words, network coding helps to alleviate competition among flows at the bottleneck, thus improving session throughput in general.

Recent work on network coding has gradually shifted its focus from a more theoretical point of view to a more practical setting. Intuitively, it may help to improve downloading times in large-scale peer-to-peer (P2P) content distribution. The current state-of-the-art of such P2P content distribution protocols is represented by *BitTorrent*, in which peers exchange blocks of data to serve missing blocks to each other. While existing work (such as the Avalanche project [4], [5]) has shown the effectiveness of network coding in P2P content distribution protocols, pessimistic results [6] also cast doubts on how much network coding may improve BitTorrent, citing its excessive computational overhead. For example, a dedicated server with dual 3.6 GHz Xeon processors can only achieve coding rates of around 270 KB/second for 256 blocks [6]. This is apparently not sufficient to saturate the upload bandwidth of high-bandwidth peers, such as those with dedicated connections of more than 100 Mbps.

On paper, it has been repeatedly shown that network coding can lead to more robust protocols with less overhead [7], and better utilization of the available bandwidth at any time. Further, network coding is capable of providing better quality of service (QoS) since improved session throughput, whether in unicast or multicast scenarios, and resilience against peer failures are both important QoS parameters. Unfortunately, to date, there has been no commercial applications or protocols that take advantage of the power of network coding. We believe that the main cause of this observation — and the main disadvantage of network coding — is the high computational complexity of *random linear codes* [3], especially as the number of blocks to code scales up. Since random linear codes are universally adopted in all practical network coding proposals, we believe that it is crucially important to design and implement random linear codes such that its real-world coding performance is maximized, on modern off-the-shelf processors. In addition, a high performance implementation of network coding is critical to determine whether network coding can offer any advantages over BitTorrent-like P2P protocols, given its high computational complexity.

The completion of this research was made possible thanks to the NSERC Strategic Grant and Bell Canada's support through its Bell University Laboratories R&D program.

To our knowledge, this paper represents the first attempt towards a high performance implementation of network coding. We first propose to implement progressive decoding with Gauss-Jordan elimination, such that blocks can be decoded progressively as they are received. We then employ hardware acceleration with SSE2 and AltiVec SIMD vector instructions on x86 and PowerPC processors, respectively. We finally use a careful threading design to take advantage of symmetric multiprocessor (SMP) systems and multi-core processors. The objective of this work is to explore the computational limits of network coding in off-the-shelf modern processors, and to provide a solid reference implementation to facilitate commercial deployment of network coding. Our high-performance implementation is packaged as a C++ class library, and runs in Linux, Mac OS X and Windows, in Intel, AMD and IBM PowerPC processor families. On a Dual dual-core PowerPC G5 2.5 GHz server, the coding bandwidth of our implementation is able to reach 43 MB/s with 64 blocks of 32 KB each.

The remainder of this paper is organized as follows. Sec. II discusses related work. Sec. III presents our design of using Gauss-Jordan elimination to achieve progressive decoding. Sec. IV presents our work on maximizing the coding performance of random linear codes, on modern off-the-shelf processors. Sec. V evaluates our high-performance implementation. Sec. VI concludes the paper with our final words.

II. Related Work

To practically implement the paradigm of network coding, one needs to address the challenges of computing *coding coefficients* to be used by each of the intermediate nodes in the session, so that the coded messages at the receivers are guaranteed to be decoded. This process is usually referred to as *code assignment*. Although deterministic code assignment algorithms have been proposed and shown to be polynomial time algorithms (*e.g.*, [8]), they require extensive exchanges of control messages, which may not be feasible in dynamic peer-to-peer networks. As an alternative, Ho *et al.* [9] has been the first to propose the concept of *randomized network coding*. With randomized network coding using random linear codes, an intermediate node transmits on each outgoing link a linear combination of incoming messages, specified by independently and randomly chosen *code coefficients* over some finite field.

Since the landmark paper on randomized network coding by Ho *et al.*, there has been a gradual shift in research focus in the area of network coding, from purely theoretical studies to more practical studies on applying network coding in a practical setting. Such a shift of focus has been marked by Wu *et al.* [10], in which the authors have concluded that randomized network coding can be designed to be "robust to random packet loss, delay, as well as any changes in network topology and capacity." The highly visible *Avalanche* project by Microsoft Research [4] has further proposed that randomized network coding can be used for bulk content distribution, in competition with *BitTorrent*, one of the most practical P2P content distribution protocols. The follow-up work in Avalanche has sought to demonstrate the feasibility of network coding with a real-world implementation in C# [5], [11]. The work has concluded that "network coding incurs little overhead, both in terms of CPU and I/O, and it results in smooth and fast downloads."

In Wang *et al.* [6], the computational complexity of random linear codes has received dedicated attention. Unfortunately, the conclusion was pessimistic, in that network coding may not improve downloading times as compared to protocols without coding, due to its high computational overhead. Theoretically, the computational complexity of random linear codes has been well known: it has been a driving force towards the development of more efficient codes in content distribution applications, including traditional Reed-solomon (RS) codes, *fountain codes* [12], and more recently, *chunked codes* [13].

While fountain codes are much less computationally intensive as compared to random linear codes, they suffer from their own drawbacks: (1) Coded blocks cannot be recoded without complete decoding, which defeats the original nature of network coding; (2) there exists some bandwidth overhead (about 5% with 10,000 blocks, and over 50% with 100 blocks); and (3) the decoding process cannot be progressively performed while receiving coded blocks, which leads to very bursty CPU usage when the final blocks are decoded. Alternatively, while Reed-Solomon (RS) codes may be also be used to reduce coding complexity, it also suffers from the lack of progressive decoding, and its significantly smaller coded message space makes it difficult for multiple independent encoders to code a shared data source, to be sent to a single receiver.

To summarize, while there is no doubt that more efficient codes exist, they may not be suitable for randomized network coding in a practical setting. In contrast, random linear codes are simple, effective, and can be recoded without affecting the guarantee to decode. We believe that our work on a high-performance parallelized implementation of random linear codes may help academics and practitioners to realize the full potential of randomized network coding in a real-world setting.

III. Random Linear Codes: A Progressive Implementation

With random linear codes, data to be disseminated is divided into n *blocks* $[b_1, b_2, \ldots, b_n]$, where each block b_i has a fixed number of bytes k (referred to as the block size). To code a new coded block x_j in network coding, a network node first independently and randomly chooses a set of coding coefficients $[c_{j1}, c_{j2}, \cdots, c_{jn}]$ in GF(2^8) Galois field [14], one for each received block (or each original block on the data source). It then produces one coded block x_j of k bytes:

$$x_j = \sum_{i=1}^{n} c_{ji} \cdot b_i \tag{1}$$

Since each coded block is a linear combination of the original blocks, it can be uniquely identified by the set of coefficients that appeared in the linear combination.

A peer decodes as soon as it has received n linearly independent coded blocks $\mathbf{x} = [x_1, x_2, \ldots, x_n]$. It first forms a $n \times n$ matrix $\mathbf{C}$, using the coefficients of each block b_i. Each row in $\mathbf{C}$ corresponds to the coefficients of one coded block. It then recovers the original blocks $\mathbf{b} = [b_1, b_2, \ldots, b_n]$ as:

$$\mathbf{b} = \mathbf{C}^{-1}\mathbf{x}^T \tag{2}$$

In this equation, it first needs to compute the inverse of $\mathbf{C}$, using Gaussian elimination. It then needs to multiply $\mathbf{C}^{-1}$ and $\mathbf{x}^T$, which takes $n^2 \cdot k$ multiplications of two bytes in GF(2^8). The inversion of $\mathbf{C}$ is only possible when its rows are linearly independent, *i.e.,* $\mathbf{C}$ is full rank.

We are now ready to show a baseline implementation of random linear codes, which includes the implementation of GF(2^8) operations, as well as progressive decoding using Gauss-Jordan elimination.

GF(2^8) operations are routinely used in random linear codes within tight loops. Since addition in GF(2^8) is simply an XOR operation [14], it is important to optimize the implementation of multiplication on GF(2^8). Our baseline implementation takes advantage of the widely-used fast GF multiplication through logarithm and exponential tables similar to the traditional multiplication of large numbers [15]. Fig. 1 shows a C++ function to multiply using three table references where log and exp reflect GF(2^8) logarithmic and exponential tables, each having 256 entries. Such a baseline implementation requires three memory reads and one addition for each multiplication.

```
byte gf256::baseline_gf_multiply(byte x, byte y)
{
    if (x == 0 || y == 0)
        return 0;
    return exp[log[x] + log[y]];
}
```

Fig. 1. Table-based multiplication in GF(2^8): our baseline implementation.

We note that a network node does not have to wait for all n linearly independent coded blocks before decoding a segment. In fact, it can start to decode as soon as the first coded block is received, and then *progressively* decodes each of the new coded blocks, as they are received over the network. In this process, the decoding time overlaps with the time required to receive the original block, and thus hidden from the tally of overhead caused by encoding and decoding times. This is an attractive feature that is uniquely available with random linear codes using dense code matrices. Although progressive decoding of later messages becomes increasingly more complex, the decoding complexity is much better balanced than fountain codes, where the bulk of the decoding process is performed after receiving the final coded blocks [12].

We use *Gauss-Jordan elimination* [16] to implement such a progressive decoding process, rather than the more traditional Gaussian elimination. Gauss-Jordan elimination is a variant of Gaussian elimination, that transforms a matrix to its *reduced row-echelon form* (RREF), in which each row contains only zeros until the first nonzero element, which must be 1. The benefit of the reduced row-echelon form is that, once the matrix is reduced to an identity matrix, the result vector on the right of the equation constitutes the solution, without any additional needs of decoding. Since we operate in GF(2^8), the usual numerical instability caused by Gauss-Jordan elimination does not affect our decoding process.

As each new coded block x_j is received, its coefficients (carried within x_j) are added to the coefficient matrix $\mathbf{C}$. A pass of Gauss-Jordan elimination is performed on this matrix, with identical operations performed on the coded

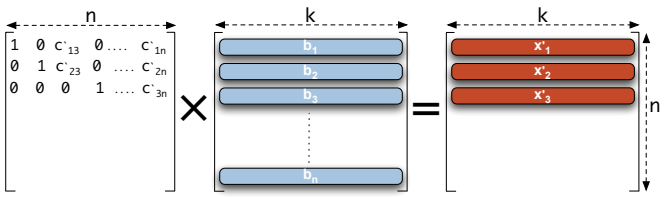

Fig. 2. State of Gauss-Jordan elimination in progressive decoding.

blocks, such that they become partially decoded blocks x'_j. After all n coded blocks are received, these partially decoded blocks become the original blocks. In addition, if a network node receives a coded block that is linearly dependent with existing blocks that have been received already, the Gauss-Jordan elimination process will lead to a row of all zeros, in which case this coded block can be immediately discarded, and there are no explicit linear dependence checks required.

Our implementation of the progressive decoding process is shown in Fig. 2, at the moment that the third coded message has been progressively decoded. The first three rows have already gone through iterations of Gauss-Jordan elimination and are now in RREF.

Our baseline implementation of progressive decoding is summarized in Fig. 3, marked by the percentage of execution times in each stage for a typical ($n = 256$, $k = 1024$) experiment over 100 runs. The stages of **A** and **E** are the most time-consuming portions of Gauss-Jordan elimination, as they loop through all existing matrix rows involving up to $n-1$ row operations each. Stage **D** takes at most one full row operation. The overall decoding complexity is n^2 row operations.

The overall encoding complexity is n^2 row operations as well, since each encode operation requires n row operations, leading up to n^2 for generating n coded blocks. However, the n^2 row operations of the decode process is performed on both coefficient rows of dimension n and coded block rows of dimension k. At encoding, the row operation are performed only on the original blocks of dimension k. As a result, decoding is generally more computationally complex than encoding.

```
Stage A Reduce leading coefficients in the new
        coefficient row to 0. [50.05%]
Stage B Find the leading non-zero coefficient in
        the new coefficient row. [0.05%]
Stage C Check for linear independence
        with existing coefficient rows. [0.00001%]
Stage D Reduce the leading non-zero entry of
        the new row to 1, such that the result is
        in REF. [0.38%]
Stage E Reduce the coefficient matrix to the
        reduced row-echelon form (RREF). [49.5%]
```

Fig. 3. Progressive decoding: our baseline implementation.

IV. Parallelized Network Coding with Hardware Acceleration

Random linear coding suffers from two major performance bottlenecks. First, multiplication in GF(2^8) is a costly operation. Second, the multiplication and addition operations are performed in tight loops over rows of coefficients and coded blocks, each of n and k bytes respectively. Each row operation is performed through a series of byte-length GF(2^8) operations because GF(2^8) multiplication is not easily scalable

to a higher granularity than the byte level. The following experiment reflects the importance of addressing these performance bottlenecks.

A sample encode and decode using our baseline implementation of random linear coding takes 9.89 and 11.91 seconds, respectively, for $n = 256$ blocks of $k = 1024$ bytes, on an iMac with 1.83 GHz Intel Core Duo processor. If the table-based multiplication in Fig. 1 is replaced with a simple $x + y$ addition, the execution time of encode and decode will be reduced to 7.12 and 9.53 seconds, a significant reduction of 39% and 25%, respectively. In the second test, we assume the same $x + y$ addition is now performed in parallel for every 16 neighboring elements, *i.e.*, row operations in 16-byte granularity. The same encode and decode operations now execute in only 0.191 and 0.242 seconds respectively which is an extra 37 and 39 times reduction in execution time!

Noting the above observations, we attempt to address both bottlenecks through hardware acceleration and parallelization, which complement each other to radically improve the coding performance.

A. Hardware acceleration with SIMD instruction sets

One way to increase the granularity of row operations is to perform GF(2^8) multiplication in wider chunks without necessarily going to GF(2^{16}) domain. All row multiplications of random linear coding require multiplying a single one-byte factor into all byte elements of a row, whether a coefficients row or a data block row. One can multiply a factor by two bytes of a row at once by building logarithm and exponential tables of $64K$ entries (to address 2^{16} elements). Similarly, the chunk size can be increased to three bytes by employing tables of $16M$ entries and so on. Obviously the tables grow quickly and become difficult to hold in memory. Also, the cache misses increase quickly and offset any gain achieved through widening the multiplication domain.

As an alternative, we propose to revisit the basics by performing the multiplication on-the-fly using a loop-based approach in Rijndael's finite field [17][14], rather than using traditional `log/exp` tables. Although the basic loop-based multiplication takes longer to perform, it lends itself better to a parallel implementation that takes advantage of vector instructions in order to operate on wider chunks of elements from a matrix row at the same time. The loop-based equivalent of the table-based multiplication in Fig. 1 is shown in Fig. 4, which resembles a regular hand multiplication by looking into lower bit of x and adding y at each iteration. At each iteration, y is shifted to the left to reflect moving to the next bit of x. However, loop-based GF(2^8) multiplication also requires a division by an *irreducible polynomial* [14] (governed by the cyclic nature of the finite field) at the end of multiplication. This division at the end can be emulated by subtracting the irreducible polynomial whenever the shift of y is about to overflow. In our implementation, the irreducible polynomial of $x^8 + x^4 + x^3 + x^2 + 1$ is used. Note that subtraction and addition are both equivalent to the *xor* operation. The loop takes at most 8 iterations to complete.

With such a loop-based implementation of multiplication, we are ready to take advantage of SIMD (single-instruction, multiple data) instruction sets, which are available on all modern commodity processors, including Intel, AMD, and IBM PowerPC families. These SIMD instruction sets allow a single operation — such as floating point/integer arithmetic and logical operations — be performed on multiple data in a parallel fashion. IBM's implementation of such instruction set is known as AltiVec and supported on all POWER family of processors. Intel's x86 vector instruction set is called SSE (Streaming SIMD Extensions) which has matured since its SSE2 variant introduced in the Pentium 4 family. AMD has also added SSE2 support since its Opteron and Athlon64 products. Both AltiVec and SSE2 employ 128-bit (16 bytes wide) registers and allow parallel integer operations on each register as 16 byte-long (or 8 short, 4 regular, 2 long) integers [18], [19].

```
byte gf256::loop_gf_multiply(byte x, byte y)
{
    byte result = 0;
    bool overflowing;

    while (x != 0) {
        if ((x & 1) != 0)
            result = result ^ y;
        overflowing = y & 0x80;
        y = y << 1;
          // irreducible poly: x^8+x^4+x^3+x^2+1
        if (overflowing == true)
            y = y ^ 0x1d;
        x = x >> 1;
    }
    return result;
}
```

Fig. 4. Loop-based multiplication in GF(2^8).

Let us now observe how SIMD instructions may improve the performance of row operations in random linear coding. The encoder performs a series of row operations solely on the incoming (or original) blocks, b_j. In the decoder, the bulk of Gauss-Jordan elimination also requires series of row operations on both coefficient and coded block rows as suggested by stages **A** and **E** in Fig. 3, which collectively consume more than 99% of the execution time. In each row operation, a single byte factor is multiplied by a full row and the result is *xor*-ed to another row. Noting the nature of such row operations, the loop-based multiplication in Fig. 4 opens up an opportunity for a vector implementation of the row operation, by GF(2^8)-multiplying the factor into 16 adjacent elements of a row, and then *xor*-ing the 16-byte result into another row at once. As a result, 16 elements of a row is now processed with one execution of the loop-based multiplication[1]. Further SIMD-based optimizations were also applied to the other stages in Gauss-Jordan elimination.

A challenging implementation detail worth noting is related to the alignment of memory allocations in the accelerated implementation. For performance reasons, many SSE2 and AltiVec instructions either require or prefer to have their memory arguments 16-byte aligned. Mac OS X guarantees the heap memory allocations to be 16-byte aligned. On Linux and Windows, we had to use special OS-specific memory

[1]Applying the irreducible polynomial to individual elements of a 16-byte chunk has to be conditioned on the value of each element. We omit implementation details due to space constraints, but it can be handled via a few SIMD instructions.

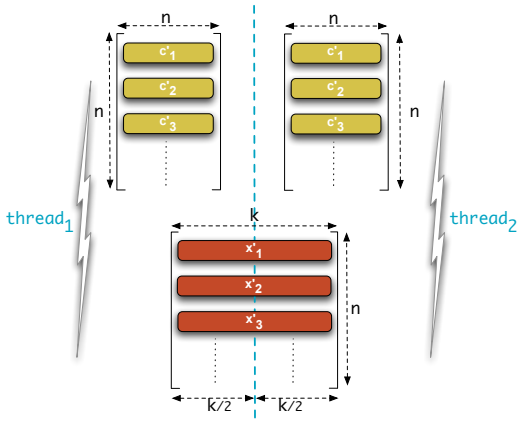

Fig. 5. Parallelized decoding: partitioning the decoding of coded blocks.

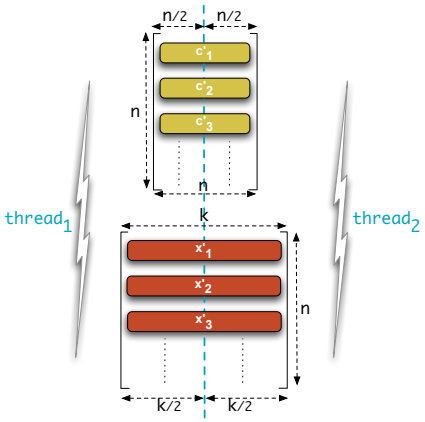

Fig. 6. Parallelized decoding: partitioning both coefficients and coded blocks.

allocation APIs for this purpose.

With our new accelerated implementation of random linear coding, we repeat the same test scenario described earlier. The accelerated implementation takes 1.78 and 2.20 seconds for encoding and decoding processes, respectively, reflecting speedups of 556% and 541% over the baseline table-based GF(2^8)-multiplication! The speedup is less than the ideal 1600%, due to the obvious usual overhead preventing a linear speedup. Such a speedup via the use of SIMD instruction sets would not be possible without switching to the loop-based multiplication. So far, our work leads to a fully accelerated and cross-platform implementation of randomized network coding, on Intel, AMD and PowerPC processors, and across Windows, Linux and Mac OS X systems.

B. Parallelized network coding

Since modern commodity processors are routinely multi-core processors, we naturally wish to further improve our accelerated implementation by increasing the level of parallelization, such that all processing cores may be fully utilized. A multi-threaded implementation would naturally take advantage of additional processors, and its performance may benefit significantly through workload partitioning. That acknowledged, parallelized network coding with more than one thread per processor may actually affect performance negatively due to threading overhead, as random linear coding is computationally intensive (CPU bound), without I/O intervals in between.

We first recall that the encoding process generates a new linear combination of incoming blocks, weighted according to random coefficients. This calculation can be partitioned among several threads, each working on a partition of all original blocks to generate the corresponding partition of the coded block. All threads start with the same sequence of random coefficients $[c_{j1}c_{j2}...c_{jn}]$, either through already prepared coefficients, or generating the coefficients on their own from a shared seed.

Next, the decoding process can similarly divide each coded block into partitions and assign each, of width $k/\text{cpu\_count}$, to a different thread. Every thread retrieves the coefficient sequence on its own and maintains the full coefficient matrix of width n. As shown in Fig. 5 for two processors, each thread operates on its private copy of the coefficient matrix, and partitions the coded block without any need to communicate with other threads. Such partitioning also improves cache performance.

Ideally, all threads start their new task at the same time and finish around the same time, since they process equal amounts of data. However, the encoding or decoding process is not complete until all threads have completed their tasks. This implies that one of the threads should serve as the coordinating thread, which synchronizes the task assignment and collection to and from other worker threads.

With only coded blocks partitioned, the achieved speedup of the decoding process is limited to the parallel portion of the overall task (which is equal to $k/(k+n)$ since each row operation is performed on both coefficient and coded block rows). If the block size k is much larger than number of blocks n, such a partitioning scheme performs close to perfection. But in a typical real-world scenario of $n = 256$ and $k = 1024$, the coefficient row operations cost 20% of the total row operations.

To further improve the speedup, we propose to extend parallelized decoding to include the coefficient matrix. Fig. 6 shows an ideal task partitioning for two threads. There exist a few challenges towards this goal, however. At stage **A** of Gauss-Jordan elimination in Fig. 3, all threads need to have knowledge of the full row of coefficients associated with the last received coded block, *i.e.*, partial knowledge is not sufficient. Searching for the first non-zero element at stage **B** is a more problematic issue, requiring each thread to pass the result of its local search to the coordinating thread, and wait for a response on the global result. Stage **E** needs to retrieve the coefficient of all previous rows that are right above the first non-zero element of the current row. Obviously, such coefficients can belong to any other partition and is not locally owned.

To solve the issues of stages **A** and **E**, each thread needs to keep some redundant data, and to access a global list updated by the coordinating thread. Unfortunately, stage **B** requires explicit synchronization between threads. To address this challenge and to reduce cache coherency updates, we have carefully designed an appropriate synchronization scheme that assigns each thread its own cache-aligned data structure. Each thread's local structure is set by the thread, and read only by the coordinating thread. The coordinating thread's response is set through a similar structure, and read by all other threads.

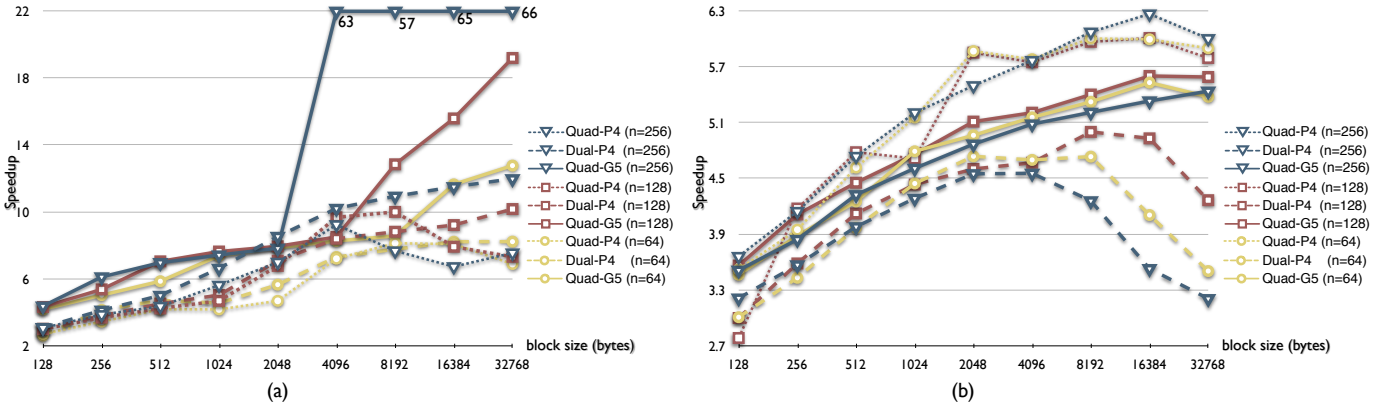

Fig. 7. Speedup of single-threaded SIMD acceleration for **(a)** encoding and **(b)** decoding processes, over the baseline implementation.

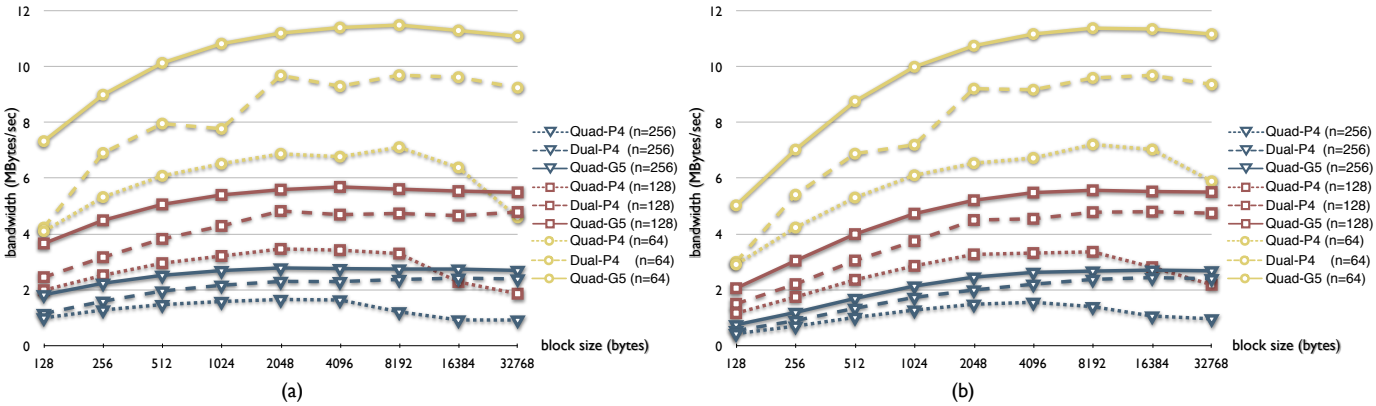

Fig. 8. Coding bandwidth of single-threaded SIMD accelerated **(a)** encoding and **(b)** decoding.

V. Performance Evaluation

In cross-platform C++, we have implemented and fine-tuned our parallelized progressive network coding with hardware acceleration. Our implementation is packaged as a library that can be statically or dynamically linked, runs well in Windows, Linux and Mac OS X, and on Intel, AMD and PowerPC processors. Our original objective of implementing a high-performance network coding engine is to explore its computational limits in modern processors. In this section, we evaluate the performance of our implementation with respect to its coding bandwidth in megabytes per second, as well as the speedup when compared to our baseline implementation. Both implementations operate in GF(2^8). Most our experimental results reflect the average of 100 runs.

We evaluate our implementation in three hardware platforms: **(1)** a Quad CPU Intel Pentium 4 Xeon 2.8 GHz server (16 GB RAM, 512 KB L2 cache on each CPU and 2 MB shared L3 cache, Linux kernel 2.6.17); **(2)** a Dual CPU Intel Pentium 4 Xeon 3.6 GHz server (2 GB RAM, 2 MB L2 cache on each CPU, Linux kernel 2.6.13); and **(3)** a Dual dual-core Power Mac G5 server with two PowerPC G5 2.5 GHz dual-core processors (4 GB RAM, 1 MB L2 cache on each CPU, Mac OS X 10.4.8). The Intel servers use SSE2 SIMD acceleration, while the Quad Power Mac G5 uses AltiVec.

A. Hardware acceleration with SIMD instruction sets

We first evaluate our single-threaded implementation of network coding, accelerated with SIMD instruction sets. We evaluate 128 bytes to 32 KB per block, with 64, 128, and 256 blocks. When compared to our baseline implementation without acceleration, the speedups of accelerated encoding and decoding are shown in Fig. 7. The coding bandwidth, in terms of MB per second, is shown in Fig. 8.

We have observed from Fig. 7 that the encoding process achieves much higher speedups than the decoding process, especially at larger block sizes. If we refer to the actual coding bandwidth in Fig. 8, however, we may observe that though the encoding bandwidth is up to 2 MB/s higher than decoding with smaller block sizes, the gap becomes much closer as the block size increases. In addition, the coding bandwidth of both encoding and decoding eventually saturate around similar block sizes of 2–8 KB. The decoding bandwidth graph resembles a delayed form of the encoding graph. In what follows, we attempt to decipher the phenomenon that we have observed.

Although both encoding and decoding processes require n^2 row operations, there exist some differences. The encoding process for each coded block always requires n row operations, performed in a single loop. Its n^2 row operations are performed only on the incoming blocks of width k. In contrast, progressive decoding on coded blocks that are received earlier requires less row operations that on those received later, and accesses a smaller number of rows. Overall, decoding requires n^2 row operations on the coded blocks of width k, plus coefficient rows of width n. In general, encoding is more "regular" than decoding.

We believe that the more impressive encoding speedup is due to the decreasing performance of the baseline encoding

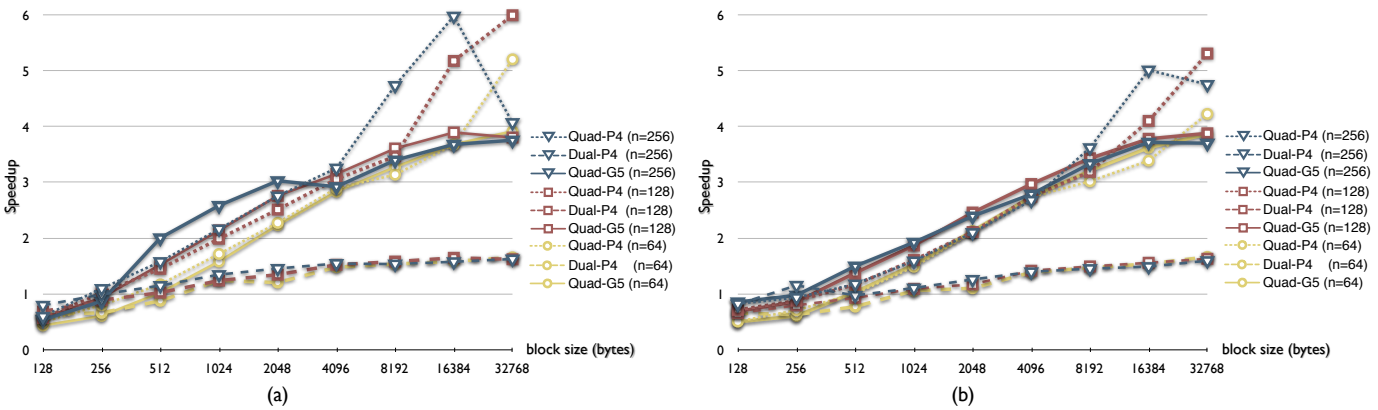

Fig. 9. Speedup results of multi-threaded SIMD acceleration for **(a)** encode and **(b)** decode over the single-threaded accelerated scheme of Fig. 7.

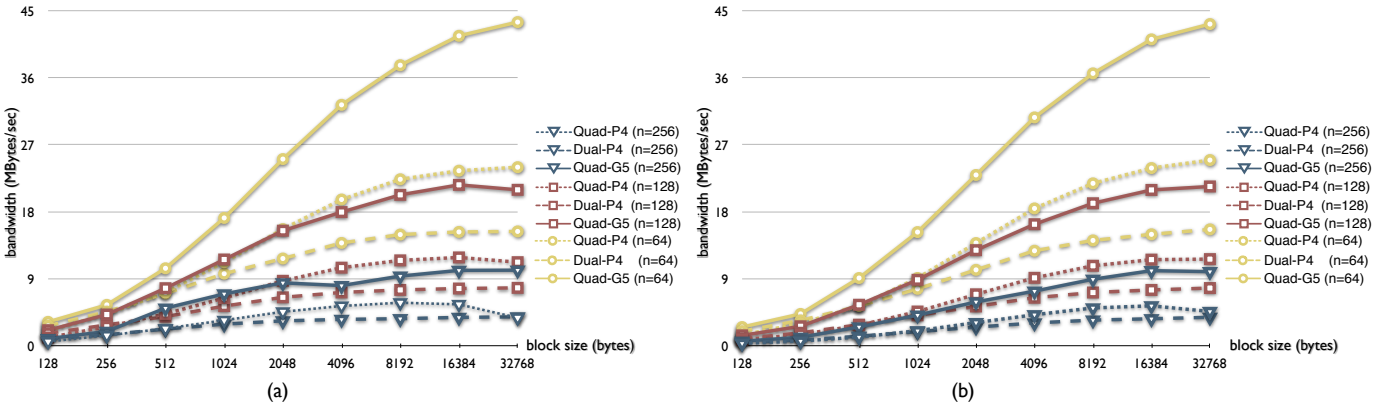

Fig. 10. Coding bandwidth performance of multi-threaded SIMD acceleration for **(a)** encode and **(b)** decode processes.

implementation, caused by the L2 cache. As the block size increases, the caching performance of memory operations is penalized, leading to lower coding performance. The accelerated encoding, however, continues to maintain its gain by compensating for the decreasing performance of caching. With respect to decoding, the extra decoding workload on the coefficient matrix becomes less significant as $n/(n+k)$ decreases. Further, the decreasing performance of caching is of lesser importance for decoding, since the decoding data set is initially small, and grows gradually as new coded blocks are received. If the cache is not sufficiently large to store the entire data set, such a gradual increase leads to less cache thrashing than encoding, which traverses the entire data set right from the very first coded block.

Not surprisingly, at a fixed block size, both encoding and decoding achieve a higher coding bandwidth with a smaller number of blocks, since they both require n^2 row operations. The dual-CPU Intel server consistently performs better than the quad-CPU Intel server in this experiment, due to its larger L2 cache and higher CPU clock speed. However, the PowerPC G5 system attains a high performance margin over the Intel servers, apparently due to its higher performance SIMD implementation. It is impressive to observe that both encoding and decoding bandwidth approach 10 MB/s on the dual-CPU Intel server and even surpass 11.4 MB/s on the G5 system, with 64 blocks. The effect of the 512 KB L2 cache of the Quad-CPU Intel server is clearly visible in Fig. 8-(a). The encoding bandwidth declines when the working set overflows the cache capacity after approximately ($k = 2$ KB, $n = 256$), ($k = 4$ KB, $n = 128$), and ($k = 8$ KB, $n = 64$) points.

At a fixed number of blocks, we have also observed that the coding bandwidth increases until a saturation point as higher block sizes are used, a typical behavior often observed in parallelized computation in response to the increase of problem sizes. This is mainly due to hardware factors such as better performance of the memory fetcher, instruction cache and branch predictor units. The coding bandwidth is affected as the working set no longer fits in the cache and becomes memory-bound.

B. *Parallelized network coding*

We now repeat the same experiments to evaluate parallelized network coding with one thread per CPU. We first start with partitioning the decoding of coded blocks only (Fig. 5). In Fig. 9, we present the speedup of using multi-threaded parallelization over a single thread with SIMD acceleration. In all test cases, the Quad-CPU Intel server outperforms the dual-CPU Intel server, showing the obvious advantage of multi-threading. An interesting observation is the super-linear speedup on the Quad-CPU Intel server. This is a classic example of how the aggregate cache of multiple processors can improve the performance of a memory-intensive computation task by partitioning the per-processor working data set.

Fig. 10 shows the coding bandwidth. When encoding 256 blocks, we observe from Fig. 9 that the Quad-Intel server achieves a speedup of 6, which is almost 4 times higher than the speedup of the Dual-Intel server. However, Fig. 10 shows that the Quad-Intel server is only 1.5 times faster than the dual-

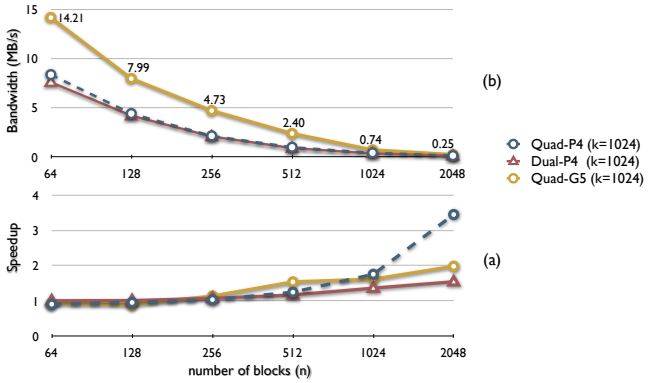

Fig. 11. Decoding with full partitioning for $k = 1024$: **(a)** speedup over decoding with partial partitioning; **(b)** decoding bandwidth.

Intel server in terms of the absolute coding bandwidth. This implies that a dramatic speedup of multi-threading does not translate to equally significant bandwidth gain. This reflects the importance of analyzing speedup and coding bandwidth together at all times.

The Quad-G5 server achieves impressive coding rates by reaching near linear speedup at large block sizes. However, the bulk of these high coding rates stems from the original performance gain through SIMD acceleration shown in Fig. 8. We have also observed that, the encoding bandwidth of the Quad-Intel server at 256 blocks peaks at $k = 8$ KB and decreases afterward. This is exactly the point that the overall data set grows to 2 MB, and overflows the 4 processors' aggregate cache ($4 \cdot 512$ KB L2 cache). The decrease is sharper for $k = 32$ KB as the encoding data set grows to 8 MB, which surpasses even the 2 MB L3 cache. Note that decoding is more tolerant of the increased block size, due to its gradually increasing working sets.

Finally, we study the advantages of full partitioning (Fig. 6), by also partitioning the decoding of the coefficient matrix. Although the decoding of the coefficient matrix is no longer performed redundantly by all threads, the extra synchronization required leads to additional overhead. Fig. 11 shows its resulting speedup over partial partitioning along with decoding bandwidth. Unlike previous experiments, the block size is now fixed at $k = 1024$ bytes, and we increase the number of blocks. This is designed to emphasize the advantage of applying threading to the coefficient matrix when $n/(n+k)$ does not diminish quickly with increasing k. By increasing $n/(n+k)$, we obviously expect to see higher gains due to the partitioning of coefficient rows with width n. At $n = 2048$ and $k = 1024$ bytes, for example, the coefficient matrix of 4 MB will become larger than the coded block matrix of 2 MB. As a result, its partitioning improves the cache performance besides improving the parallelism and obviously would lead to a high speedup.

The experiment with 512 blocks is more interesting, because the data set completely fits into the cache and the achieved speedup is solely due to full partitioning. At ($n = 512$, $k = 1024$), each processor of the Quad-Intel server will operate on 128-byte coefficient rows and 256-byte coded block rows. This reduces the aggregate row size to $128 + 256$ bytes from $512 + 256$ bytes of partial partitioning, effectively reducing the aggregate row into half. Of course, we only gain a speedup of 1.26, rather than the ideal 2, because of the extra threading overhead and synchronization of full partitioning.

Since a larger number of blocks dramatically affects the coding bandwidth, it is natural to use the smallest number of blocks possible in real-world network coding. This implies that in most real-world applications based on network coding, parallelized network coding with full partitioning may not gain more than roughly 10% to 15% of coding bandwidth over partial partitioning. Nevertheless, we have still observed that encoding and decoding bandwidth reaches 20.3 and 19.2 MB/s at 128 blocks of 8 KB each with our Quad PowerPC G5 server, and 43.5 and 43.3 MB/s at 64 blocks of 32 KB each. If we use 16 blocks of 32 KB each, they are even able to reach 155.8 and 156.2 MB/s![2] As another way to show our results, Fig. 12 conveniently illustrates the coding bandwidth for 128 blocks of 4 KB each across different hardware and OS platforms. No matter how one sees them, these represent impressive coding performance, thanks to our parallelized and accelerated implementation of network coding.

Platform setup					Coding rate (MB/s)	
System	OS	SIMD type	# of threads	L2 Cache	Encoding	Decoding
Quad PowerPC G5 2.5 GHz	Mac OS X	AltiVec	4	1 MB	18.01	16.38
Quad P4 Xeon 2.8 GHz	Linux	SSE2	4	512 KB	10.54	9.17
Dual Opteron (AMD) 2.4 GHz	Linux	SSE2	2	1 MB	9.56	8.75
Dual P4 Xeon 3.6 GHz	Linux	SSE2	2	2 MB	7.21	6.50
iMac Intel Core Duo 1.83 GHz	Mac OS X	SSE2	2	2 MB (shared)	5.81	5.60
Intel Core Duo 1.66 GHz	Windows XP	SSE2	2	2 MB (shared)	4.62	4.43

Fig. 12. Platform comparison of coding performance at (n = 128, k = 4 KB).

VI. Conclusions

This paper represents the first attempt towards a high-performance implementation of randomized network coding. The objective of this research is to explore the computational limits of random linear coding in modern processors. We propose to use Gauss-Jordan elimination to perform progressive decoding, such that the decoding time may overlap with the time required for data transmission. Our implementation is now complete with hardware acceleration with SIMD instruction sets available on modern commodity processors, as well as multi-threading to take advantage of symmetric multiprocessors to parallelize computation tasks. With a wide variety of working sets in our coding tests, significant speedup and coding bandwidth have been achieved. We are now confident to claim that, as long as we code fewer than 128 blocks, the computational complexity of randomized network coding may not become a performance bottleneck even on dedicated servers with more than 100 Mbps connections. In peer-to-peer applications with typical DSL bandwidth, we believe the CPU usage is minimal. The task of coding more than 128 blocks still remains to be an interesting challenge.

[2]To establish a context, I/O bandwidth of SATA disk drives is usually between 30 and 60 MB/s.

REFERENCES

[1] R. Ahlswede, N. Cai, S. R. Li, and R. W. Yeung, "Network Information Flow," *IEEE Trans. on Information Theory*, vol. 46, July 2000.

[2] R. Koetter and M. Medard, "An Algebraic Approach to Network Coding," *IEEE/ACM Transactions on Networking*, vol. 11, no. 5, pp. 782–795, October 2003.

[3] S. Y. R. Li, R. W. Yeung, and N. Cai, "Linear Network Coding," *IEEE Transactions on Information Theory*, vol. 49, pp. 371, 2003.

[4] C. Gkantsidis and P. Rodriguez, "Network Coding for Large Scale Content Distribution," in *Proc. of IEEE INFOCOM 2005*, March 2005.

[5] C. Gkantsidis, J. Miller, and P. Rodriguez, "Anatomy of a P2P Content Distribution System with Network Coding," in *Proc. of the 5th International Workshop on Peer-to-Peer Systems (IPTPS 2006)*, 2006.

[6] M. Wang and B. Li, "How Practical is Network Coding?," in *Proc. of the 14th Intl. Workshop on Quality of Service (IWQoS 2006)*, 2006, pp. 274–278.

[7] C. Fragouli, J.-Y. Le Boudec, and J. Widmer, "Network Coding: An Instant Primer," *ACM SIGCOMM Computer Communication Review*, vol. 36, January 2006.

[8] P. Sanders, S. Egner, and L. Tolhuizen, "Polynomial Time Algorithm for Network Information Flow," in *Proc. of the 15th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2003)*, June 2003.

[9] T. Ho, R. Koetter, M. Medard, D. Karger, and M. Effros, "The Benefits of Coding over Routing in a Randomized Setting," in *Proc. of International Symposium on Information Theory (ISIT 2003)*, 2003.

[10] P. Chou, Y. Wu, and K. Jain, "Practical Network Coding," in *Proc. of Allerton Conference on Communication, Control, and Computing*, October 2003.

[11] C. Gkantsidis, J. Miller, and P. Rodriguez, "Comprehensive View of a Live Network Coding P2P System," in *Internet Measurement Conference (IMC 2006)*, 2006.

[12] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman, "Efficient Erasure Correcting Codes," *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 569–584, February 2001.

[13] P. Maymounkov, N. Harvey, and D. Lun, "Methods for Efficient Network Coding," in *Proc. of 44th Annual Allerton Conference on Communication, Control, and Computing*, September 2006.

[14] Ron Roth, *Introduction to Coding Theory*, Cambridge University Press, first edition, 2006.

[15] Neal Wagner, *The Laws of Cryptography with Java Code*, http://www.cs.utsa.edu/ wagner/lawsbookcolor/laws.pdf, 2003.

[16] Carl Meyer, *Matrix Analysis and Applied Linear Algebra*, SIAM, 2001.

[17] Wikipedia, "Finite field," http://en.wikipedia.org/wiki/Finite_field.

[18] Freescale Semiconductor, *AltiVec Technology Programming Interface Manual*, June 1999.

[19] Intel Corporation, *IA-32 Intel® Architecture IA-32 Intel Architecture Optimization Reference Manual*, April 2006.