Efficient Random Network Coding for Distributed Storage Systems

Ádám Visegrádi, Péter Kacsuk

Computer and Automation Research Institute, Hungarian Academy of Sciences, Hungary
{visegradi.adam,kacsuk.peter}@sztaki.mta.hu

Abstract. Making distributed storage systems reliable is an important challenge. Simple replication may cause severe storage overhead when individual components of the system are very unreliable. Using *erasure codes* is a promising solution for this problem, but it presents its own challenges; it makes the design of such a system very complex, and it presents the problem of reparation. *Network coding* has been suggested to be used in the communication in these networks to help reduce overhead. However, using random network coding *as*—not besides—erasure coding would be an even more promising field to investigate; such a system would have a simple design, need little or no centralization, and reparation of the system could be much simpler than it is in other erasure coding schemes.

The first step on this path is to investigate whether network coding can achieve such a performance that it is a feasible alternative to other erasure codes. This paper presents our experiences about the realization of random network coding based on the discrete logarithm of the finite field. We discuss possible performance optimizations for such a system, and provide performance measurement results focusing on data storage scenarios.

1 Introduction

In the era of Big Data, distributed storage systems are used to deal with the increasing volumes of data. Hadoop[5] and MapReduce[6] have proven that using cheap commodity hardware is a feasible alternative to expensive RAID storages. Even cheaper is to utilise volunteer or community storage, like BOINC[4], where storage space is donated to the system by people interested in it. These approaches use unreliable nodes as a basis for the distributed storage system. This unreliability can be handled by increasing *redundancy* in the system.

Consider the data to be a sequence of k blocks with a uniform size. The basic way to introduce redundancy is to *replicate* the blocks of the data. This approach has its drawbacks. First, the raw storage space required to store some data is the (integer) multiple of the size of the data; and this factor of redundancy has to be very high if the nodes' reliability is too low. Second, as failing nodes remove

replicas of a block of the data from the system, that block may become rare, impairing locality.

Erasure coding (EC) is an alternative to replication, which alleviates both problems of replication, at the cost of CPU time. Erasure coding algorithms—e.g. Reed–Solomon[11] or Fountain codes[10]—create $n > k$ coded blocks from the original data in a way that *any* $k' >= k$ coded blocks will be sufficient for reconstruction. Because *any* k' block is sufficient for reconstruction, much lower factor of redundancy is sufficient than in case of replication[14]. Also, in erasure coding, the factor of redundancy does not have to be an integer. Although this seems to be a trifle, but the difference between a required redundancy of 2.1 and $\lceil 2.1 \rceil = 3$ can be substantial when there are peta-bytes of data.

The problem with erasure coding is the design complexity of the system, and the overhead required for its reparation[12]. In both cases (replication and EC), when blocks go missing, they have to be complemented. In case of replication, only rare blocks have to be further replicated. In case of erasure coding, to complement the missing blocks, the whole data has to be reconstructed, so new coded blocks can be generated from it[12][8]. Dimakis et al. propose regenerating codes[7] to remedy this problem. Regenerating codes use network coding[3] to communicate encoded packets, which enables the reparation of redundancy without reconstructing the original data. However, if the data is very distributed and each node stores only one piece of a data object, this approach may not provide many benefits. This can happen in large volunteer systems, which are of particular interest to us.

A promising approach would be to use *random (linear) network coding*[9] (RNC) to store the data. Linear network coding (LNC) treats the blocks of data as vectors—and the data itself as a matrix—over a finite field $\mathbb{F}(2^w)$ ($w \geq 1$). Coding is performed by creating linear combinations of the original blocks, while decoding is done by solving the corresponding linear system. LNC has been well studied in the area of networking, as an alternative to routing; and random network coding has been proposed as a simple solution for finding suitable coefficients for linear combinations. RNC can be considered as an erasure coding method, as $n > k$ randomly encoded packets can be generated from the original file, of which any $k' >= k$ will be sufficient for reconstruction. Furthermore, it is stochastically optimal, and it converges to optimal with increasing field size[9].

As an erasure coding, RNC could solve the problems of replication; it would even perform better than traditional erasure coding schemes [2]. Also, the problem of reparation in a RNC system would become quite straight-forward: as stored blocks are random linear combinations (r.l.c.) of the original data blocks, a r.l.c. of the *coded* blocks will *also* be a r.l.c. of the *original* blocks. That is, reparation can be done by randomly selecting existing coded blocks from the system, and creating r.l.c.-s of them—no complete reconstruction is needed, nor the selection and location of specific sets of coded blocks. These properties would make r.l.c. a great candidate for coding data in distributed storage scenarios.

But linear coding is very CPU intensive. Would it be a feasible alternative to other coding schemes? There is little information on the practical implementa-

tion of r.l.c. in the literature, and no information on the possible optimizations there are. Also, most simulations and measurments focus only on corroborating the theoretical benefits of r.l.c. (e.g. [2], [7]). In [13], Wang et al. discuss a real implementation of r.l.c.; however, they use only the finite field $\mathbb{F}\left(2^8\right)$ and discuss no details on the possible optimizations.

We have implemented r.l.c.[1], and measured the raw coding speed we can achieve with it. In this paper we discuss the practical implementation, possible optimizations, and our performance measurement results of our implementation of network coding, with our conclusion being that, with the right choices, network coding can be a feasible alternative to existing erasure coding methods.

2 Efficient Realization of Network Coding

In this paper we are focusing on Big Data problems; that is, the handling of relatively large files (or data objects in general). In practice, really large files are broken up into smaller *segments*; for simple discussion we will use "file" as a denotation for this unit of storage.

Consider a file to be a sequence of k blocks; each block being $l := \frac{\text{file size}}{k}$ bytes long. Random network coding treats these blocks as vectors over a finite field $\mathbb{F}\left(2^w\right)$. For simplicity, we assume the file consists of k *whole* blocks; and for practicality, we assume that the word length, w, is a multiple of 8. Each word is of $u := \frac{w}{8}$ bytes, and each block is a vector of $\frac{l}{u}$ words.

2.1 Discrete Logarithm and Field Size

To study the possibilities of such a system, the first step is to implement finite field operations. Unfortunately, universal finite field implementations impose high overhead on encoding/decoding a file. This is because they rely on an internal representation of polynomials in the finite fields: each word of u bytes of the data—both original *and* coded—has to be converted to the internal representation, which requires extra memory and CPU time. Then, the field operations, whose complexity increases with the size of the field, are performed on these polynomials. And finally, the polynomials have to be converted back to binary words.

In our implementation, we used discrete logarithm tables to perform finite field multiplications. Using pre-calculated logarithm and power tables enables us to perform finite field operations on the words of the data directly. Without conversion, addition and subtraction becomes a simple bit-wise *xor* of the words of the data, while multiplication can be performed in *constant time* regardless of the field size. This approach requires $2(2^w u)$ bytes overhead: a power table and a logarithm table with the same size, each containing 2^w elements, each element being u bytes long. This means that using $\mathbb{F}\left(2^{16}\right)$ requires only 256kB constant extra memory. Unfortunately, $\mathbb{F}\left(2^{16}\right)$ is a practical limit, as $\mathbb{F}\left(2^{24}\right)$ and $\mathbb{F}\left(2^{32}\right)$ would need 96MB and 32GB respectively.

Using tables instead of generic libraries, the overall performance of encoding/decoding should *increase* with field size. As we are focusing on storage scenarios, we use relatively large blocks: this makes the matrix multiplication the dominant operation (as opposed to matrix inversion, which, in our case, is negligible). Coding—and decoding—is a multiplication of a $k \times k$ matrix (the coefficient matrix) with a $k \times \frac{l}{u}$ one (the file consisting of k blocks). Thus the multiplication is $\Theta(k^2 \frac{l}{u}) = \Theta(k^2 \frac{\text{filesize}/k}{u}) = \Theta(k \frac{\text{filesize}}{u})$ in complexity.

From this, it is clear that the multiplication complexity is inversely proportional to the word length, linearly proportional to the block count k, and linearly proportional to the file size. Practically, if we use constant time field operations like discrete logarithm, this means that using $\mathbb{F}\left(2^{16}\right)$ instead of $\mathbb{F}\left(2^{8}\right)$ should double the speed of the operation. Another consequence of using a larger field is the decreased probability of linearly dependent vectors. This reduces traffic and computation overhead caused by redundant transmission.

2.2 Data placement

We do not focus on the bottleneck using the disk, network, or any medium creates; we are interested in the bottleneck network coding creates. Therefore, all operations are performed in memory; files are loaded in memory before encoding takes place, and stored to disk after the operation. We measure only the time needed to perform the operation in memory, excluding media overhead.

Even though we exclude the overhead of using a storage or transfer medium, we still face this kind of problem on a smaller scale. In our case, because of large files and blocks, the dominant operation is the matrix multiplication. Because the operation is CPU intensive, it would be optimal to perform it as "close" to the processor as possible; however, these large files will not fit into higher level caches of the CPU. Furthermore, the naïve implementation of the matrix multiplication impedes cache locality. Therefore, we use simple matrix blocking to enhance the performance of multiplication. Also, we learnt that the order of the three nested loops of matrix multiplication is a significant factor in efficiency.

2.3 Details of Implementation

Our implementation is a C/C++ program, highly optimized both by hand and by the compiler (g++). The power and logarithm tables are represented as C arrays, calculated beforehand. Elements of the finite field are represented by raw words of data. This way, addition and subtraction in base-2 finite field can be performed by *xor*-ing the operands. Multiplication and division can be performed in constant time with lookups and integer operations, for example: $a \cdot b = pow\left[(log[a] + log[b]) \bmod (2^n - 1)\right]$.

When encoding, a $k \times k$ matrix is generated randomly—using the built-in C `rand()` function—with which the input file is multiplied. If the generated matrix cannot be inverted, it is noted and another matrix is generated; and so on. Please note, that we used the basic Gaussian elimination to invert the matrix.

Thus here, "cannot be inverted" does not necessarily mean that the matrix is singular, it can also mean that the rows should have been reordered. However, we will se that in $\mathbb{F}\left(2^{16}\right)$, naïve Gaussian elimination is adequate.

After generating a coding matrix, both the generated matrix and the result of the multiplication is saved to disk. When decoding, the previously generated matrix is inverted, and the result of the previous encoding is multiplied by it.

Manual optimization of the program code was necessary. We have gained marginal but noticeable performance increase with using local constant variables instead of `struct` dereferencing in the loop core. On the other hand, we could almost double the performance of the multiplication by reordering the nested loops.

We implemented both naïve and square-blocked matrix multiplication with adjustable block size to measure the effect of locality. This provided considerable performance gains over simple matrix multiplication.

We also measured the effects of parallelizing the calculations. The matrix is cut into pieces—workunits—that are processed by a thread pool with the specified number of threads. For non-blocked multiplication the workunits correspond to the rows of the matrix, for blocked multiplication, the pieces correspond to blocks.

3 Performance measurements

Our main interest was the raw performance of network coding, without the overhead of a medium (network, disk, etc.). Therefore, we loaded the (un)coded blocks in memory, measured the time to perform the multiplication, and then save the result back to disk. This way, the only overhead we experienced was that of the CPU–memory bus and cache latency.

We performed the measurements on our test machine, with an Intel i5-2410M dual core, hyperthreaded CPU. We also performed the measurements in our cloud infrastructure. The cloud-based measurements were slower, but they produced similar results, leading to the same conclusions. In this paper, the numbers produced on our test machine are shown.

3.1 Parameters

For a single measurement, the parameters listed in Table 1 can be specified. Some of the measurements were focused only on a subset of the ranges described.

Field size	q	The size of the field; $q \in \{2^8, 2^{16}\}$.
File size	fs	The size of the input file; $fs \in \{64\text{kB} \cdot 2^i \mid i \in [0..13]\}$. (That is, 64kB–512MB, exponentially.)
Block count	k	The number of blocks the file is split into; $k \in [2^3..2^7]$.
Mode		Encoding or decoding.
Thread count	$ncpus$	Number of threads to use; $ncpus \in [1..4]$.
Mul. block size	$mulbs$	The size of the square block used in blocked multiplication; $mulbs \in 2^0, ..., 2^6; mulbs \leq \frac{k}{2}$. If $mulbs = 1$, non-blocked multiplication is performed.

Table 1: Parameters of a single measurement.

3.2 Results

Field size The size of the field is known to affect the reliability of the system, as increasing field size reduces the probability of vectors being linearly dependent[9]. On the other hand, as described in Section 2.1, if field operations take constant time, increasing the field size will increase the performance of the matrix multiplication.

In total, 12,445 matrices were generated during the experiments discussed here. When a matrix is generated, it is checked if it is *acceptable*, that is, if it can be inverted with simple Gaussian elimination; if not, it is thrown away, and a new one is generated. Of all the matrices, 6648 were generated for $\mathbb{F}\left(2^8\right)$ and 5797 for $\mathbb{F}\left(2^{16}\right)$. More matrices were generated for $\mathbb{F}\left(2^8\right)$, as the chance of generating an unacceptable matrix was higher in this case: in $\mathbb{F}\left(2^8\right)$, 1218 matrices were unacceptable, in $\mathbb{F}\left(2^{16}\right)$ only 7. Using the built-in C random generator **81.6787% of the matrices in $\mathbb{F}\left(2^8\right)$ were acceptable, while in $\mathbb{F}\left(2^{16}\right)$, 99.8792%**.

There is also a performance gain when using $\mathbb{F}\left(2^{16}\right)$ (Figure 1). In theory, this gain should be 2, as the operational complexity of the matrix multiplication is inversely proportional to the word size. However, the actual gain was less than this, because the discrete logarithm table is much bigger (256kB versus the 512 bytes for $\mathbb{F}\left(2^8\right)$), which impedes cache locality. The performance gain was usually between 1 and 2, its median varied between 1.467 and 1.221 depending on the number of blocks (k).

Matrix multiplication We considered matrix blocking to optimize cache locality. Our Intel CPU has two physical cores, two memory channels and two 32kB L1 caches, one for each core. Theoretically, if we use $\mathbb{F}\left(2^{16}\right)$ and 64×64 blocks, a single iteration would use $64 \times 64 \times 2 \times 3$ bytes = 24kB (block size $\times$ element size $\times$ (left-hand, right-hand and result matrices)), which 24kB can fit in the L1 cache, providing superior performance compared to naïve multiplication. In practice, this is close to be true, but results fluctuate marginally, because the logarithm and power tables have to be accessed too.

We experienced that using blocking reduces operation time greatly; however, increasing the block size to from 4 to 64 provides only marginal performance gains (Figure 2).

Another optimization was to reorder nested iterations of matrix multiplication. For the multiplication, we need three indices, i, j and k. These indices iterate over the matrix in the dimensions shown on Figure 3. For matrix multiplication, one can iterate through the target matrix with i and j, and in the innermost loop create scalar products with k. As addition is commutative, the iterations can be reordered. By using nested loops iterating over i, k and then j, we doubled the speed of the multiplication (Fig. 4).

Number of threads As matrix multiplication is trivially parallelizable, it was expected, that using multiple physical cores with independent memory chan-

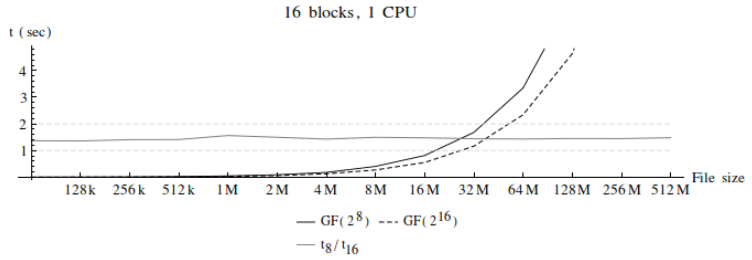


Fig. 1. Performance using $\mathbb{F}\left(2^8\right)$ compared to $\mathbb{F}\left(2^{16}\right)$. The results show that the operational time is linearly proportional to the file size, while in case of $\mathbb{F}\left(2^{16}\right)$, the constant factor is smaller. The ratio of the two results is shown as the grey horizontal graph; the levels of 1 and 2 are shown for reference. The domain spans up to 512M, but the plot is trimmed for visibility.

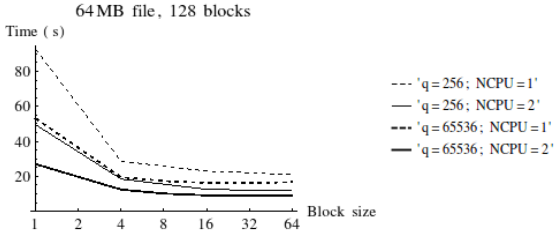
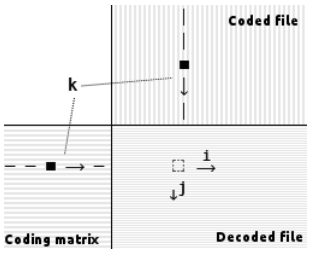


Fig. 2. Performance differences using different block sizes for matrix multiplication. For block size = 1, naïve multiplication was used.




```
1 fill_with_zeroes(decoded_file);
2 for i = 1 to blocklength do
3 |   for j = 1 to blockcount do
4 |   |   for k = 1 to blockcount do
5 |   |   |   decoded_file[i, j]+=coding_matrix[j, k] * coded_file[k, i];
6 |   |   end
7 |   end
8 end
```

Fig. 3. Illustration of the indices used for matrix multiplication. The algorithm shown is the "original" algorithm. In the "reordered" algorithm, lines 3 and 4 are exchanged.

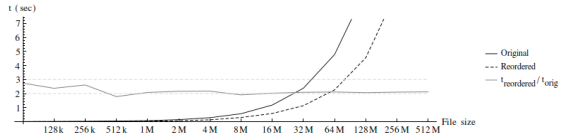


Fig. 4. Performance using original (i, j, k) and reordered (i, k, j) iterations. The results show that using the reordered iteration, the resulting multiplication can be performed at least twice as fast. The domain spans up to 512M, but the plot is trimmed for visibility.

nels the speed of the computation can be increased. On the other hand, using more concurrent threads than there were physical cores did not increase the performance. Using two separate cores nearly doubled the performance of the multiplication (Fig. 5).

Overall Performance Using the best optimization methods we identified, we measured the overall performance of coding for different file sizes and values for k. In the following measurements, we used $\mathbb{F}\left(2^{16}\right)$, 2 CPUs, and reordered, blocked matrix multiplication.

As expected, the time to encode or decode a file takes the same amount of time, as these are practically the same operation with different names for the operands.

Our experiments show that the time needed to perform the multiplication increases linearly with the file size (Fig. 6) and linearly with k (Fig. 7). This accords with our expectations described in Section 2.1. This conclusion holds true regardless of the number of threads we used, or the field size—changing these parameters only changed the resulting numbers, but not the correlations described here.

In terms of *throughput*, as the operation time increases linearly with file size, the throughput will be *constant for a given value of* k, while it will drop linearly as k is increased. Our results are detailed in Table 2.

These results show that certain parameter sets can achieve considerable performance, and may be used for suitable data-intensive applications. For example, consider 128MB chunks cut into 16 blocks. Decoding a chunk would take 4.55 seconds. After the initial delay, if the data is processed as a stream, a bandwidth of about 49.3 MB/s can be achieved. As the next chunk can be lodaded from a medium while the previous is being decoded, assuming a medium fast enough, its overhead will only appear in the initial delay, and the decoding bandwidth can be sustained. In the best case above, 87 MB/s of bandwidth could be achieved.

4 Conclusion and Future Work

In this paper we have presented our experiences about the implementation and performance measurements of randomized linear coding. We described several approaches to improve the performance of such a system. From these results, we conclude that, with the right choices, using randomized coding over the finite field $\mathbb{F}\left(2^{16}\right)$ can be a feasible alternative to erasure coding in distributed storage systems.

Learning this, we are now confident that using randomized linear coding instead of classical erasure coding is an area worth investigating. In our future research, we will study the theoretical possibilities of this approach, and will try to further improve the technical methods of realization.

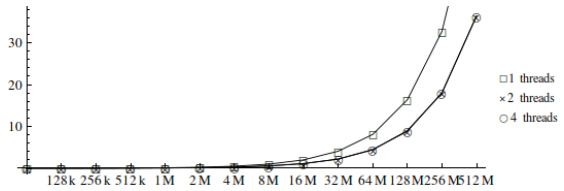


Fig. 5. Performance using different number of threads on two physical cores. The results show that the performance of the multiplication can be increased by using multiple physical cores; while using more threads than there are cores will not increase performance.

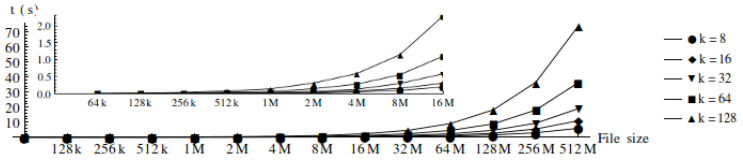


Fig. 6. Performance using files of different sizes, with different values for k. The interval between 64kB and 16MB is magnified for visibility. The results show that the time complexity of the multiplication is linearly proportional to k.

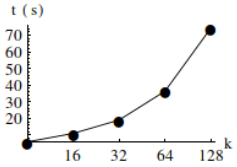


Fig. 7. Performance using different values for k with fixed file size (64MB). The results show that the time complexity of the multiplication is linearly proportional to k.

	File size													
k	64kB	128kB	256kB	512kB	1MB	2MB	4MB	8MB	16MB	32MB	64MB	128MB	256MB	512MB
8	0.0009	0.0016	0.0028	0.0076	0.0108	0.0214	0.0490	0.0986	0.1946	0.3780	0.7273	1.4550	2.9411	5.8764
16	0.0016	0.0035	0.0066	0.0107	0.0277	0.0365	0.0811	0.1566	0.3072	0.6480	1.2729	2.5983	5.4194	10.9605
32	0.0023	0.0055	0.0099	0.0256	0.0346	0.0663	0.1304	0.2976	0.5828	1.1390	2.2639	4.5539	9.3095	19.0175
64	0.0043	0.0086	0.0203	0.0439	0.0655	0.1459	0.2780	0.5567	1.1023	2.2089	4.4876	8.8185	18.0695	36.3531
128	0.0134	0.0182	0.0362	0.0810	0.1414	0.3100	0.5904	1.1449	2.2859	4.5113	9.0622	18.1723	36.5472	74.2251

Table 2. Numerical performance results for each (file size, k) pair. Each value is the time, in *seconds*, a matrix multiplication was performed.

Bibliography

[1] Random Network Coding Library. https://github.com/avisegradi/rnc-lib. Accessed: 2013-09-10.

[2] S. Acedanski, S. Deb, M. Médard, and R. Koetter. How good is random linear coding based distributed networked storage. In *Workshop on Network Coding, Theory and Applications*, 2005.

[3] R. Ahlswede, N. Cai, S. Li, and R. Yeung. Network information flow. *Information Theory, IEEE Transactions on*, 46(4):1204–1216, 2000.

[4] D. Anderson. BOINC: a system for public-resource computing and storage. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4 – 10, Nov. 2004.

[5] D. Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11:21, 2007.

[6] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *Information Theory, IEEE Transactions on*, 56(9):4539–4551, 2010.

[8] A. Dimakis, K. Ramchandran, Y. Wu, and C. Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3):476–489, 2011.

[9] T. Ho, M. Médard, R. Koetter, D. Karger, M. Effros, J. Shi, and B. Leong. A random linear network coding approach to multicast. *Information Theory, IEEE Transactions on*, 52(10):4413–4430, 2006.

[10] D. MacKay. Fountain codes. In *Communications, IEE Proceedings-*, volume 152, page 1062–1068, 2005.

[11] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.

[12] R. Rodrigues and B. Liskov. High availability in DHTs: erasure coding vs. replication. *Peer-to-Peer Systems IV*, page 226–239, 2005.

[13] M. Wang and B. Li. How practical is network coding? In *Quality of Service, 2006. IWQoS 2006. 14th IEEE International Workshop on*, page 274–278, 2006.

[14] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. *Peer-to-Peer Systems*, page 328–337, 2002.