Founsure 1.0: An Erasure Code Library with Efficient Repair and Update Features

Şuayb Ş. Arslan
MEF University, Maslak, Istanbul, Turkey

Abstract

Founsure is an open-source software library, distributed under LGPLv3 license and implements a multi-dimensional graph-based erasure coding entirely based on fast exclusive OR (XOR) logic. Its implementation utilizes compiler optimizations and the multi-threaded implementation to generate the right assembly code for the given multi-core CPU architectures with vector processing capabilities. Founsure (version 1.0) supports a variety of features that shall find interesting applications in modern data storage as well as communication and computer network systems which are becoming hungry in terms of network bandwidth, computational resources and average consumed power. In particular, Founsure library provides a three dimensional design space that consists of computation complexity, coding overhead and data/node repair bandwidth to meet different requirements of modern distributed data storage and processing systems in which the data needs to be protected against device, hardware and node failures. Unique features of Founsure include encoding, decoding, repairs/rebuilds and updates while the data and computation can be distributed across the network nodes.

Keywords: Distributed Storage, Erasure Coding, Fountain Coding, Single Instruction Multiple Data (SIMD), Erasure coding, openMP, Reliability.

1. Motivation and significance

Erasure coding is used for fault tolerance and providing required reliability to ensure high availability of data in distributed data storage and processing systems [1]. Reed-Solomon (RS) codes are the conventional option for constructing erasure codes that provide coding overhead–optimal design and thereby use the storage space as efficiently as information theoretically possible [2]. As the modern data storage systems evolved to possess different requirements, the set of constraints on the design of erasure codes has dramatically changed. For instance, previous research on erasure codes such as RS codes mostly focused on optimizing the coding overhead i.e., minimization of storage space consumption for a given target data reliability [2, 3]. Moreover, some of the most popular designs considered pure XOR operations to both provide required reliability and efficient computation [3]. More recently, locally repairable codes have attracted attention due to their efficient utilization of network resources and eventually achieve better overall data reliability [3] at the expense of suboptimal coding overhead. Besides the proprietary implementations of advanced erasure coding algorithms, many open source implementations based on different mathematical operations are available online [6, 7]. Main objective of these open source software has been to provide to the community the overhead optimal and fast/efficient erasure coding libraries without any consideration of the harnessed network and scarce computational resources as well as data/node repair efficiency in a distributed setting.

Founsure 1.0 utilizes pure binary operations on a three-dimensional bipartite graph to construct a multi-functional erasure code. The design space includes computation complexity, coding overhead and repair bandwidth as different dimensions of optimization. If one prefers
to have an overhead optimal design with very good complexity performance, then it might be advisable to use the well known Jerasure 2.0 [7] erasure coding library by Dr. Plank and Dr. Greenan, which is now fully supported by RedHat Ceph community [8]. Unfortunately, this library and the likes (zfec [9]) do not provide a sufficient code structure to address modern problems of distributed storage systems such as degraded reads, data repair/rebuild, consumed bandwidth, data security, etc. On the other hand, the main objective and purpose of developing Founsure 1.0 software library is to provide different operating points in three dimensional design space based on the requirements of the storage applications through a set of parameter configurations. Therefore, Founsure can be shown to demonstrate huge potential for distinct storage applications through smart/guided configuration steps. For instance, Founsure 1.0 has a configured structure that enabled its successful adaptation to a baseline deduplication engine within an archival scenario [10].

The encoding process of Founsure 1.0 begins with a two-dimensional conventional bipartite graph which leads to a non-systematic low density generator matrix (LDGM) code. A version of this class of codes with appropriate input distributions are generally known as fountain codes [12, 13]. The degree distribution of Founsure’s LDGM code is specially selected to meet a good trade-off operating point between computation complexity, coding overhead and repair bandwidth. Current version (1.0) supports Robust Soliton Distribution (RSD) [13] (if one prefers good coding overhead design) as well as all possible finite max-degree distributions (if one prefers different operating points) including the one in [14] by default. However, we note that the referenced distributions are optimized for minimum coding overhead criterion only.

One of the building pillars of Founsure is genuine symbol check relationships. The terminology of check symbols are quite common in Low Density Parity Check (LDPC) code community. What check symbols do is that they provide a mathematical relationship between a subset of data symbols to check a certain condition, usually a simple binary sum or equivalent Galois field operation. The check idea is also quite beneficial for local data repairs/rebuilds [15]. So on top of the two-dimensional bipartite graph of Founsure, the encoding engine generates check nodes for “data-only” (referred hereafter as check #1), “data & coding” (referred as check #2) and “coding-only” (referred as check #3) chunks/symbols. As can be imagined, these check nodes (mathematical relationships) can be added to the two-dimensional bipartite graph to give it a three dimensional look. This new representation shall be used to provide advanced decoding, repair and update features of Founsure. Throughout the document, nodes typically contain multiple chunks and chunks typically contain multiple symbols.

Founsure uses Belief Propagation (BP) [11] (a.k.a. message passing) algorithm to resolve or decode the user data, to repair the encoded data or update the encoded data. Sticking to BP as a design criterion is to ensure low-complexity decoding process and allow fast/efficient operation. Version 1.0 also supports register-level parallelism through compiler optimizations as well as multi-threading using the open standard openMP primitives with its encode, decode, repair and update functions. Multi-threading feature, once properly configured and used, allows parallel processing capability and allow fast processing time for shared-memory architectures by utilizing parallel hardware resources at the operating system’s thread-level. By reducing the processing time, Founsure ensures quick responses to the common read/write requests of any generic distributed storage system.

With the current release v1.0, the user data does not appear at the output in pure/plain format. In other words, one cannot read off data from the encoder output without any further processing. Therefore, using Founsure 1.0 encoding, one can think of the user data to be encrypted automatically. Note that we use pseudo-random number generators (based on linear congruential generator) and seed (integers of long type) to generate graph connections of the underlying bipartite graph. So without the seed number (the keys in an encryption context), there is no way to recover original user data because the graph cannot be generated reliably. Therefore, Founsure 1.0 software package also provides a user configurable light weight built-
in encryption feature. Compared to systematic codes in which we can read off data without any additional processing, Founsure requires more work but provides in turn data security in addition to inherent data protection. As long as decoding is made fast using various techniques we discuss in this study, non-systematic codes should not be an overall performance bottleneck for the rest of the system.

This paper shall briefly describe the set of functionalities provided with Founsure 1.0 library and the details of the software architecture as well as the advanced features. The source code, a comprehensive user guide, few test results and related documentation is available also from github and the web link [http://www.suaybarslan/founsure.html](http://www.suaybarslan/founsure.html).

## 2. Software description and architecture

### 2.1. Software Functionalities

Founsure 1.0 has the following three executable main components that achieve four important functionalities.

- **founsureEnc**: Encoder engine that generates $s$ number of data chunks (to be stored in $s$ different failure domains) under a local Coding directory and a metadata file that include information about the file and the coding parameters including the seed information.

- **founsureDec**: Decoder engine that requires a local Coding directory with enough number of files, a valid file name and an associated metadata file to run multiple Belief Propagation (BP) passes in order to decode the user data.

- **founsureRep**: Repair engine that also requires a Coding directory with enough number of files and
  - fixes/repairs one or more data chunks should they have been erased, corrupted or flagged as unavailable.
  - generates extra coding chunks should a code update has been requested. System update is triggered if data reliability is decreased/degraded over time or increased due to equipment replacements.

These functions are used to execute encoding, decoding, repair and update operations. There are also utility functions of Founsure 1.0 which are provided to help system admins to make correct design choices on degree distributions, required reliability, desired complexity and storage space efficiency. We also use utility functions to trigger update functionality as will be demonstrated later. One of the distinctive features of utility functions is that they do not directly process user data, however they are used to help main functions to modify and process the user data properly. Version 1.0 currently supports two utility functions as listed below.

- **simDisk**: This function can be used to exhaust all possible combinations of disk failures for a given set of coding parameters. In other words, this function checks whether the provided coding parameters are sufficient to achieve a user defined reliability goal. Therefore, running this function can help us design target-policy erasure codes by configuring degree distributions for achieving various system-level goals besides reliability.

- **genChecks**: This utility function is crucial for two different important functionalities: (1) fast/efficient repair/rebuild of data and (2) seamless on-the-fly update. For the repair process, it generates two types of checks: Check #2 and Check #3 and writes them to a `<testfile>_check.data` file in a format described within this document. In case of update, it modifies the metadata file as well as `<testfile>_check.data` file so that the coding chunks can be updated by running `founsureRep` function.
Next, we provide the details of Founsure 1.0 encoding, decoding, repair and update operations, particularly the implementation details of FounsureEnc, FounsureDec and FounsureRep functions. For theoretical details, we refer the reader to the appropriate reference documents such as [16].

2.2. Implementation details of Encoding/Decoding Operations

In graph terminology, nodes (sometimes referred as equations) are represented as graph vertices and node relationships are represented by edges of the graph. There are three types of nodes in a 3-D bipartite graph; data nodes, coding nodes and check nodes. The coding nodes represent a set of linear combinations of data nodes generated through a predetermined mathematical function such as Exclusive Or (XOR) logic operation. Check nodes represent all the local sets of data and coding nodes for which a certain mathematical relationship is satisfied such as even or odd parity. A simple mathematical function that is used by Founsure 1.0 is XOR operation that utilizes multiple data blocks and generate a single block of information as a result of XOR logic operation. We use $f$ flag to indicate the file name, $k$ to indicate the total number data nodes/sysmbols where $b$ of these are the original user data nodes/symbols, $n$ to indicate the total number of coding nodes/symbols and $t$ to indicate the number of bytes to store per node/symbol.

In FounsureEnc function, data file with $\text{filesize}$ bytes is partitioned into multiple $b \times t$ bytes and each partition is encoded independently as shown in Figure 1. As of version 1.0, partition coupling is not supported between distinct partitions i.e., partitions are processed independent of eachother. This technique is currently under investigation and might have interesting performance improvements to our design/implementation in analogy to spatially-coupled LDPC codes [17]. However, coupling may have different effects in case of partial disk failures and may lead to non-uniform decoding performance across partitions.

If $\text{filesize}$ is not a multiple of $b \times t$ bytes, then we use zero padding to make $\text{filesize}$ a multiple. FounsureEnc also checks whether $s|n$. If not, the least largest $n$ is selected automatically to satisfy $s|n$. Such a necessary correction enables us to store exact same amount of information bytes across different failure domains. This is particularly important if the underlying storage mediums share the same physical durability against failures.
Encoding proceeds as follows. First, a memory space of worth \((k+n)t\) bytes is allocated as the buffer and the buffer contents are initialized to zero. Next, check \# 1 equations are generated by an efficient array LDPC encoding. The choice of array LDPC as the precode is to enable efficient encoding operation and fast processing. As a result of this operation, an extra \(k - b\) chunks are created to make up a total of \(k\) chunks of data. The precoding process is shown as (1) in Figure 1. Later, a total of \(n\) coding chunks are generated from the whole set of \(k\) data chunks based on a LDGM base code with a configured "FiniteDist" degree and pseudo-random selection distributions. This process is shown as (2) in Figure 1. Finally, \(n\) coding chunks are distributed (striped) equally across distinct output files for allocation on \(s\) number of drives. We repeat this process for each data partition in a looped subprocess and append coding chunks at the end of the corresponding output files. For a given <filename>.ext file, we use <filename>_.disk0..0i.ext to refer to the \(i\)th output file. The number of zeros that appear in the name of output files is controlled by the "parameter.h" variable DISK_INDEX_STRING_LEN.

In Founsure 1.0 implementation, we have distinct object definitions for encoding, decoding, and repair operations. These objects have the trailer "*Obj" in common and include the same set of parameters in their object fields. For instance, both encoding and/or decoding functions accept EncoderObj and/or DecoderObj constructs as inputs. Similarly, \(b\) and \(k\) variables can be accessed using the standard way EncoderObj.sizesb and EncoderObj.sizek.

Each encoding/decoding object is associated with an initial seed value (EncoderObj.seed, default value is 1389488782 which is observed to give good recovery performance) from which other seed values as well as the local sets of data chunks are pseudo-randomly created. Each coding chunk within EncoderObj and DecoderObj has their own unique ID. These IDs are used to identify the coding chunks which might be erased. The seed value is used by the pseudorandom generator to produce a sequence of integers which form the basis of coding chunk degree number assignment as well as the selected data chunks for coding chunk computations. These numbers are stored as part of the object and can be regenerated using the same initial seed number followed by the regular recurrence relationship. Let us assume we have \(s\) number of output files (failure domains), then we use the default value EncoderObj.seed + \(i\) as the seed of the \(i\)th output file with \(0 \leq i < s\).

Each check node \(c\) is associated with a degree number \(c_d\) (chosen according to an appropriate degree distribution \(\Omega(x) = \sum_i \Omega_i \) where \(\Omega_i\) is the probability of choosing degree \(i\)) and \(c_d\) data node neighbors are selected to be involved in final symbol computation. The degree distribution \(\Omega(x)\) is typically selected to minimize the coding overhead. For instance the following degree distribution is proposed for Raptor codes:

\[
\Omega(x) = 0.007969x + 0.49357x^2 + 0.16622x^3 + 0.072646x^4 + 0.082558x^5 \\
+0.056058x^8 + 0.037229x^9 + 0.05559x^{19} + 0.025023x^{64} + 0.003135x^{65}
\]
where $\Omega_1 = .007969$, $\Omega_2 = .49357$, $\Omega_3 = 0.16622$, $\Omega_4 = .072646$, $\Omega_5 = .082558$, $\Omega_8 = .056058$, $\Omega_9 = .037229$, $\Omega_{19} = .05559$, $\Omega_{64} = .025023$, $\Omega_{65} = .003135$.

However, Founsure does not necessarily minimize overhead. It may optimize overhead, repair bandwidth and complexity at the same time. We recommend to choose degree distributions that will give us a good trade-off point between these three objectives. A systematic optimization procedure to achieve a desired operation point is the subject of further investigation. Although there is no optimal point for all applications, Founsure is designed to be highly configurable to fit in different requirements and sensitivities of modern storage ecosystems.

We run FounsureDec when we want to collect a subset of output data files and recover the input data file. Decoder is based on belief propagation algorithm a summary of which is provided in Algorithm 1. BP function admits DecoderObj, indexes of erasures $E \subset N = \{1,2,\ldots,n\}$, the generator matrix of the base code $B \in \mathbb{F}_2^{k \times n}$ and a maximum number of iterations $maxit$.

In Algorithm 1 we use $b_{i,:}$ to refer to the $i$th row of $B$ and $b_{:,i}$ to refer to the $i$th column of $B$. Additionally, $b_{A,B}$ refers to a matrix whose rows and columns are given by the rows and columns of $B$ indexed by the sets $A$ and $B$. The decoder utilizes the information contained in metadata file to generate (prepare) the contents of DecoderObj, particularly the underlying coding graph. It works in a similar fashion to FounsureEnc i.e., it reads the striped coding chunks, loads the buffer, and runs BP algorithm at most twice (once for the outer graph code and if need be, additional one for the inner Array LDPC precode) and recovers the $bt$ bytes at each turn. Finally, these bytes are written to decoded/recovered data file by calling standard kernel I/O commands.

To summarize the software architecture of Founsure’s encoding and decoding schemes, we provide Fig. 3 to illustrate the order of software building blocks that take place to encode/decode the user data. The repair operation’s architecture resembles a lot to this figure and hence omitted to save space. As can be seen, we used different colors to differentiate the encoding and decoding as some of these software blocks are uniquely used and some are commonly used by the encoder and decoder processes.

Having all computation based on pseudorandomly selected chunks and carrying out these computations solely in terms of simple XOR logic has the cost of making the code non-optimal in terms of overhead (tough it might be near-optimal through choosing appropriate degree distributions). If $n$ coding symbols are distributed over $s$ drives and when one of the drives fail, a subset of coding symbols are lost. In order to find what fraction of $f$-failure combinations can be tolerated for a given degree distribution, we provide a utility function $\text{simDisk}$ that exhausts all possible combinations of failures to report which failures cases are tolerable by the code and which are not. Such a utility function is extremely useful for determining the reliability of the data protected by Founsure library.

---

**Algorithm 1** Belief Propagation (BP) Algorithm (*Skeleton*)

1: procedure BP($B, \mathcal{E}, \text{maxit}$)  
   \> Inputs: $B \in \mathbb{F}_2^{k \times n}$, $\mathcal{E} \subset N$: The set of erasures.
2:  $\mathcal{N} = \{1,\ldots,n\}$  
   \> Initialize indexes.
3:  $\mathcal{K} = \{1,\ldots,k\}$  
   \> Initialize indexes.
4:  $C \leftarrow b_{\mathcal{N}\setminus\mathcal{E}}$  
   \> Find survival matrix.
5:  $F \leftarrow \mathcal{K}$  
   \> Initialization: $F$ holds the unrecovered indexes.
6:  while $F \neq \emptyset$ and $i < \text{maxit}$ do
7:      for $g = 1,\ldots,|\mathcal{N}\setminus\mathcal{E}|$ do
8:          $c_{\mathcal{K}\setminus F,:} \leftarrow 0$  
   \> Zero-out rows for decoded symbols.
9:              if $\text{weight}(c_{:,g}) = 1$ then
10:                 $F \leftarrow F - \{f : c_{f,g} = 1 \text{ for } f \in F\}$  
   \> Find degree-1 coding symbols.
11:          return $F$  
   \> Return unrecovered indexes.

---

![Image](image-url)
2.3. Check Equations and The Data Repair Process

We have three types of check nodes as mentioned before. We provide the details of such checking process in this subsection.

**Checks #1**: These check equations are defined by the precode of the Founsure (for version 1.0, we selected an Array LPDC code family [21] for efficient processing as given in Algorithm 2). Based on the selection of good precodes, the mathematical and coding parameter selections etc., the graph connections are automatically determined. Founsure 1.0 includes a precode support based on a binary array LDPC code and future releases shall include external precode support which can be provided by the user using a preformatted input file. Please see precoding subsection to find more information about the construction of these check equations.

**Checks #2**: These check equations are generated as given in Algorithm 4. One of the special features of these checks is that only one neighbor is selected from the data nodes and the rest of the neighbors of the check node are from the coding nodes. This special feature can be used to partially decode the input data without running the complete decoder and reconstruct the unnecessary parts of the input data. An application of this could be securely stored multimedia source in which the Region of Interest (RoI) can be directly reconstructed using this type of check equations.

**Checks #3**: These check equations are generated as given in Algorithm 4. These checks form the local groups based on the coding nodes. These checks are primarily used to repair the permanently erased, long-time unavailable or unresponsive coding nodes in case of hardware, software and network failures.

2.4. Precoding Process - Generation of Check #1 equations

A \((b, k, n)\) Founsure code takes \(b\) data symbols (a total of \(bt\) bytes) and initially generates \(k - b\) Check #1 parity symbols based on the binary array LDPC encoding [21]. This special choice of array LDPC codes enable efficient encoding operation (linear with blocklength) and improves the complexity performance of the overall Founsure code.
Algorithm 2 Array LDPC Checks (Check \# 1)

1: procedure checkone(\(b, k\))
2: \(L^{(1)} \leftarrow 0_{k \times k - b}\)
3: \(p \leftarrow\) largest_prime_factor\((k)\)
4: \(k' \leftarrow k/p\)
5: \(j' \leftarrow k' - b/p\)
6: for \(j = 0; j < j'; j + +\) do
7: for \(i = 0; i < p; i + +\) do
8: for \(m = 1; m < k' - j' + 1; m + +\) do
9: \(l_{i+jp,m-1}^{(1)} \leftarrow k'p - (j' - j + m - 1)p - (m(j' - j - 1) - i - 1 + p) \pmod{p} - 1\)
10: return \(L^{(1)}\)

The procedure outlined in Algorithm 2 uses a generic function largest_prime_factor(.) which chooses the largest prime factor of the argument. The rate of the array LDPC is defined as \(r_{LDPC} = b/k\). The user can choose any \(k, n\) and \(r_{LDPC}\) and hence we can calculate the appropriate \(b = \lfloor kr_{LDPC} \rfloor\). Let \(p =\) largest_prime_factor\((k)\), we may not be able to get the quantity \(\lfloor kr_{LDPC} \rfloor/p\) equal to an integer. We can definitely use floor function to get an estimate of \(j'\). However, the array LDPC code performance is heavily dependent on \(k'\) and \(j'\) values and there is no array LDPC code for all \((k, r_{LDPC})\) pairs. If \((j', k')\) pair are small, the code performance is observed to be pretty bad. For this reason, we provide an algorithm that reasonably chooses a good performing array LDPC code and satisfies (within some error margin) the user provided parameters \(k, n\) and \(r_{LDPC}\) at the same time. One can see the chosen parameters by the library through adding "-v" flag at the end of Founsure main functions.

Let us define the following system parameters before we formally provide the algorithm that determines the closest good performing array LDPC code for the user provided parameters \(k, n\) and \(r_{LDPC}\). These system parameters with their default values are defined in "parameter.h" file and can be modified.

- DIFF_TH: Allowed error threshold between the estimated and user provided \(b\) values.
- RRATE_TH: Allowed error threshold between the estimated and user provided precode rate.
- RED_BYTE_TH: Allowed redundant zero bytes to be appended at the end of the file for parameter consistency.
- RAND_WIN_MAX: Random number search window maximum value.
- RAND_WIN_MIN: Random number search window minimum value.

![Figure 4: Founsure graph code is three dimensional bipartite graph with three sets of check equations.](image)
Algorithm 3 Adjust Parameters with Precode (APP)

1: procedure app($r_{LDPC}$, filesize, b, t)
2: \[ b' \leftarrow b, b \leftarrow 0, k \leftarrow 1, k' \leftarrow 1e9, j' \leftarrow 1e9, p \leftarrow 2, blocks \leftarrow 1, tries \leftarrow 0, iter \leftarrow 0 \]
3: while \(|b - b'| > \text{DIFF}_\text{TH} \lor |b/k - r_{LDPC}| > \text{RRATE}_\text{TH} \lor k' > p \lor j' > p \) do
4: \[ k \leftarrow \left\lfloor \frac{b'}{r_{LDPC}} \right\rfloor + \text{rand()} \mod (\text{RAND}_\text{WIN}_\text{MAX} + \text{RAND}_\text{WIN}_\text{MIN}) - \text{RAND}_\text{WIN}_\text{MIN} \]
5: \[ p \leftarrow \text{largest prime factor}(k) \]
6: \[ k' \leftarrow k/p, j' \leftarrow k' - b/p \]
7: if \( j' < \text{ARRAY}_\text{MIN}_\text{JJ} \) then
8: \[ j' \leftarrow 2 \]
9: \[ b \leftarrow (k' - j')p \]
10: if \( b > 0 \) then
11: \[ blocks \leftarrow \text{filesize} / tb + 1 \]
12: \[ tries + + \]
13: if \( tries > \text{TRIES}_\text{TH} \) then
14: \[ \text{DIFF}_\text{TH} \leftarrow \text{DIFF}_\text{TH} + \text{DELTA}_\text{DIFF}_\text{TH} \]
15: \[ \text{RRATE}_\text{TH} \leftarrow \text{RRATE}_\text{TH} + \text{DELTA}_\text{RRATE}_\text{TH} \]
16: \[ tries \leftarrow 0 \]
17: \[ iter + + \]
18: if \( iter > 1e7 \) then
19: \[ \text{print error and exit.} \]
20: \[ blocks \leftarrow \text{filesize} / (tb) + 1 \]
21: \[ \text{redundantzeros} \leftarrow bt \times blocks - \text{filesize} \]
22: return \( b, k, \text{redundantzeros} \)

- \text{ARRAY}_\text{MIN}_\text{JJ}: Minimum Array LDPC "$j'n" parameter value.
- \text{ARRAY}_\text{MIN}_\text{KK}: Minimum Array LDPC "$k'n" parameter value.
- \text{TRIES}_\text{TH}: Threshold on the number of tries before incrementing \( \text{DIFF}_\text{TH} \) and \( \text{RRATE}_\text{TH} \).
- \text{DELTA}_\text{DIFF}_\text{TH}: Step size increment for \( \text{DIFF}_\text{TH} \)
- \text{DELTA}_\text{RRATE}_\text{TH}: Step size increment for \( \text{RRATE}_\text{TH} \).

Next, we provide the algorithm that returns the estimated values of \( b, k \) and redundant number of zeros (\( \text{redundantzeros} \)) that need to be appended to the input user data. The algorithm admits four inputs, namely \( r_{LDPC}, \text{filesize}, b \) and \( t \). Initial values of system parameters shall be set by "parameter.h" file and are changed locally within the function implementing Algorithm 3.

2.5. Generating Information for Efficient Repair

To be able to efficiently repair lost data, we need to extract repair information from the underlying graphical content of the generated Founsure code. We observe that Check #3 nodes are the most suitable node type for the repair process since it directly establishes a relationship between the coded symbols. It is not hard to see having more of these type of nodes (created independently or dependently) gives alternative ways of repairing a given node in case of different combinations of node failures happening in the communication network. In other words, the more of these type of checks we find, the more potential we have for the regeneration of the lost coded chunks. With regard to this observation, we can use two
techniques to increase the number of check #3 type information based on check #1 and check #2 equations:

- First method is as follows. We identify the coded nodes with degree one (say we have \( M \) of those nodes). Identify their data node neighbors. There are \( M \) check #2 equations that connect these data nodes with the coding nodes. Since the corresponding \( M \) coding nodes carry the same information, we can use these check #2 type check equations as additional check #3 check equations. Note that since check #2 and check #3 equations are derived from the same Founsure base graph, this technique is likely to generate already existent local recovery groups or dependent local recovery groups (i.e., local groups that can be derived from existent local groups) for coding nodes.

- Second method is as follows. Note that check #1 is user defined although subject to a predefined structure. This defines local recovery groups over the data nodes. Since each data node is linked to local recovery groups of coded nodes through check #2, we can use this relationship to derive check #3 type check equations. For example, suppose we have the following check #1 local recovery group defined for data nodes \( D_0, D_1 \) and \( D_2 \): (\( D_0, D_1, D_2 \)). Also, suppose that we have the following check #2 equations:

\[
\begin{align*}
\Rightarrow D_0 &= C_0 \oplus C_1 \oplus C_2 & (1) \\
\Rightarrow D_1 &= C_1 \oplus C_{12} \oplus C_{17} \oplus C_{20} \oplus C_{99} & (2) \\
\Rightarrow D_2 &= C_0 \oplus C_{21} \oplus C_{99} & (3)
\end{align*}
\]

Thus, we can find a check #3 type equation given by \( (C_2, C_{12}, C_{17}, C_{20}, C_{21}) \) by observing the following equivalence,

\[
C_2 \oplus C_{12} \oplus C_{17} \oplus C_{20} \oplus C_{21} = D_0 \oplus D_1 \oplus D_2
\]

Note that since this technique uses check #1 equations, it is likely to generate distinct check #3 local recovery groups and help improve repair performance dramatically. These additional check #3 local recovery groups (for instance the operation of Equation (4)) are efficiently computed by the set union function \( \text{setXOR} \) given in "encoder.c" file.

2.6. Algorithm for Jointly Generating Group #2 and Group #3 Check Equations

We propose a heuristic algorithm to generate Group #2 and Group #3 check equations at the same time for efficiency. This algorithm uses XOR operation (\( \oplus \)) to sparsify the generator matrix \( B \). If \( B \) is full rank, i.e., \( \text{rank}(B) = k \) then the algorithm is guaranteed to converge successfully. This is due to elementary matrix row operations shall generate \( n - k \) zero columns for a full rank \( B \) and hence the algorithm will leave the main while loop and generating a modified \( B \) with column weights \( k \) ones and \( n - k \) zeros. In mathematical terms at the end, we should be able to find a permutation matrix \( P \) such that \( BP = [I_{k \times k} \mid 0_{k \times n - k}] \). Also, \( \mathbf{L}^{(2,3)} = \mathbf{L}^{(2,3)} P \) shall hold all the local recovery sets i.e., Check #2 equations in the first \( k \) columns and Check #3 equations in the last \( n - k \) columns. We can express different types of checks as the union of all Check #2 and Check #3 equations as given by

\[
\bigcup_{i<k}(D_i, \{C_s : \tilde{x}^{(2,3)}_{s,i} \neq 0\}) \cup \bigcup_{i\geq k}(\{C_s : \tilde{x}^{(2,3)}_{s,i} \neq 0\})
\]

(5)

where \( \tilde{x}^{(2,3)}_{s,i} \) denote \( s \)th row and \( i \)th column entry of \( \mathbf{L}^{(2,3)} \). In Algorithm 4 we provide the details of the algorithm using a pseudocode. We use a simple function \( \text{zero_columns}(\cdot) \) that finds the number of nonzero columns of the matrix in the argument. Since \( B \) is typically sparse, we use sparse representation of matrices in our Founsure 1.0 implementation for efficient memory utilization and order the local recovery groups based on their cardinality i.e., the set with smallest cardinality comes first. Such an arrangement helps reduce the repair/update complexity since the repair function processes local groups in sequential order.
Algorithm 4 Check Generation Algorithm (CGA)

1: procedure CGA(B) > Input matrix $B \in \mathbb{F}_2^{k \times n}$
2: $L^{(2,3)} \leftarrow I_{n \times n}$ > Identity matrix
3: while zero_columns(B) < $n - k$ do > Number of zero-columns is checked.
4: $\text{temp} \leftarrow 0$
5: for $j=0; j<n; j++$ do
6: for $i=0; i<n; i++$ do
7: if $j \neq i$ and $2w(b_{.,j}) > w(b_{.,i})$ then
8: if $w(b_{.,j} \oplus b_{.,i}) < w(b_{.,j})$ then
9: $b_{.,j} \leftarrow b_{.,j} \oplus b_{.,i}$
10: $l^{(2,3)}_{.,j} \leftarrow l^{(2,3)}_{.,j} \oplus l^{(2,3)}_{.,i}$
11: $\text{temp} \leftarrow 1$
12: if $\text{temp} = 0$ then
13: break;
14: return $B, L^{(2,3)}$ > B: Check type, $L^{(2,3)}$: Local sets.

2.7. Management of Check Equations for Group #2 and Group #3 and the Generation of <filename>_check.data File for Efficient Repair/Update Process

Check #1 equations are determined through a binary array LDPC code as explained before. The number of equations are user defined through a selection of precode rate and selected based on the reliability imposed by the application. The graph connections are deterministic and given by the constraints of the array code.

Unlike Check #1, Check #2 and #3 are determined pseudo randomly by the Founsure base code. Based on the generator matrix of the code $B$, Algorithm 4 is run to determine $n$ equations. If the algorithm converges, then we should have $k$ equations for Check #2 type and $n - k$ equations for Check #3 type. The algorithm produces a correct set of local equations (sets) but does not guarantee those equations to be independent. In generating those equations, we do not employ any matrix inversions (which is quite costly for large size matrices) to find check equations and hence we trade off the efficiency by performance. The function that generates check #2 and #3 local recovery groups is the utility function genChecks.

The function genChecks assumes that a metadata file is already generated by a previous run of the encoder founsureEnc. Hence genChecks generates check groups and modifies the meta_data file (appends the size of check data in terms of sizeof(int) bytes at the end of the metadata file if “-m” flag parameter is True). The check information is stored in another binary file called <filename>_check.data. This file stores an integer array with a specific format. The reason for introducing a format is to use bulk read/write capabilities of fread and fwrite C library functions which will make kernel’s I/O performance acceptable.

The proposed format in this study is pretty straightforward and can be improved. We use flag bits to differentiate between the two distinct check equations. Thus, the integer value of the first sizeof(int) bytes in <filename>_check.data is either 0 or 1.

- If it is 1 (Group #2), then the next integer value (next sizeof(int) bytes) gives the data symbol index which is involved within a local recovery group whose degree is given by the following integer (next sizeof(int) bytes). This degree also indicates the next “degree” number, i.e., the number of integers to be read as part of one local recovery group for the coded symbols.
- If it is 0 (Group #3), then the next integer value (next sizeof(int) bytes) gives the degree number i.e., the number of integers to be read as part of one local recovery group for coding symbols.
The nice thing about Algorithm 4 is that if it converges, then all of the data symbols are covered exactly by one particular Group #2 local recovery group. Let us provide an example to illustrate the working principle and suppose that we have the following integer array stored in `<filename>_check.data`:

0 4 13 56 17 66 1 19 2 11 13 0 2 39 88 ...

If we decode this integer array, we will be able to say that the first local set is of type Group #3 and this set has four elements. In other words, 13th, 56th, 17th and 66th coding symbols form a local recovery group i.e., their binary sum should produce all-zero content. The next local set belongs to Group #2 and the associated data symbol index is 19. This data symbol along with 11th and 13th coded symbols (two coding symbols) form a local recovery group. This way we can decode the whole integer array stored in `<filename>_check.data`. If the algorithm converges, there should be $k$ leading 1’s and $n-k$ leading 0’s in the integer array not necessarily written in sequential order. Note that the total number of integers contained in the array is given by

$$N = \sum_{c=0}^{n-1} L_c + k + 2n$$  \hspace{1cm} (6)

where $L_c$ is the total number of elements in local Group #2 (excluding the data symbols) and Group #3 indexed by c. We also note that even if the algorithm does not converge, the maximum memory occupancy possible is $N \times \text{sizeof(int)}$ bytes, so it is enough to allocate the size of memory given by Equation (6) for the file without encountering a segmentation fault.

Fig. 5 summarizes how different functions of Founsure interact with each other, what other metadata is generated/used and what type of read/write permissions are granted to each of these functions for the proper operation of Founsure library.

2.8. Reading/Formatting the contents of `<filename>_check.data` file

When the repair process is initiated, memory allocation as well as repair object (`RepairObj`) preparation begins. The main repair engine shall look for `<filename>_check.data` under `/Coding` directory. If it finds one and if the metadata is appropriately formatted (after a successful format check), it will read-in the metadata and format the check # 2 and check
3 equations for the preparation of RepairObj. A bulk read kernel call is performed and all the content is transferred to memory (inside the buffer pointed by content2read). Since Founsure’s decoding, repair and update operations are solely based on BP algorithm, it sequentially searches only one unknown over the available local sets in a loop. In order to reduce the computation and bandwidth, it is essential that repair/decode process use small size check # 3 equations first so that we do not have to run though the end of the loop to complete the overall repair process. Founsure implementation extracts check # 3 equations from the buffer (content2read) using the standard qsort(.) function and then fills in the appropriate fields of RepairObj. The ordering can be enabled or disabled for check # 2 and # 3 equations using parameters ORDER_CHECK_2 and ORDER_CHECK_3 in "parameter.h" file.

2.9. Update Process

An update process is about making the existing Founsure code stronger or weaker by either generating more redundancy (in case of increased failures or wear-out) or taking away unwanted redundancy (in case of using more reliable devices for storing information). If we would like to make the existing code weaker, it is quite easy. We just modify the metadata accordingly and erase the redundancy manually. Founsure does not automatically erase files and leave it to upper layer software management. So for the rest of this section, we particularly mean making the code stronger when we update the code structure.

The desirable features of a generic update process can be listed as follows.

- An update process should minimize the modification of existent data generated by the encoding operation.
- An update process should generate extra redundancy consistent with the existent data with minimum processing effort.
- An update process should have minimum limitation on the extent of extra redundancy that can be generated.
- An update process should not violate the rules set by the encoding and decoding processes.

Founsure’s update mechanism poses no modification changes to the existent data generated by the encoding process. In that sense, its update process functions as ideal as possible. Founsure update process is tightly related to repair process. This is mainly because updating a code is nothing but repairing the needed blocks of information to help increase the reliability of data. We call genChecks to update the current code using the flag ‘-e’. There must be a valid metadata associated with the code at the time genChecks is called. Code update process will rewrite n, the number of bytes used for the integer array due to check #2 and check #3 equations and update <filename>_check.data. Hence the repair process uses the metadata (the rule set) generated by the previous runs of encoder/decoder pair. These series of modifications do not make any changes to the existent data/coding chunks. In order to trigger/sync changes with data, we finally need to call founsureRep function with the appropriate file name. Since the existent data is only read and we use minimum cardinality local recovery groups while updating, the processing effort is minimized. We finally note that since Founsure is based on fountain-like codes, there is no practical limit to the number of coding symbols that can be generated. As can be seen, the update functionality of Founsure is designed and implemented in observation of desirable features listed above.

3. Advanced Features

Although many advanced features of Founsure are described in previous sections, we have two more important implementation-specific advanced features that make Founsure’s performance stand out.
3.1. Shared Memory Parallelism

In shared memory multiprocessor architectures, threads can be used to implement parallelism. The shared-memory standard openMP, is high level and portable interface which makes it easier to use multi-threading capability and obtain satisfactory performance improvements. Many erasure coding libraries such as Jerasure 2.0 [7] has encoding/decoding engines which comprise independent “for” loop iterations and hence possess huge potential for multi-threaded processing. Multi-threaded implementations of Jerasure 2.0 are studied in [19] and [20].

As can be seen in Fig. 3 for founsureEnc, the software consists of three stages executed in a loop. However, two of these stages, namely reading the data in to the EncoderObj and DecoderObj that stay on memory and writing the object contents to the persistent storage devices require kernel I/O calls. As a result of these calls, the performance will be inhibited by the throughput performance of the underlying storage devices I/O bandwidth. Thus, our focus is essentially the second stage, namely the pure encoding and decoding in which the data traverses only between CPU caches and the main memory.

Founsure 1.0 utilizes shared memory approach standard openMP library directives to help use multiple threads to handle the workload of encoding, decoding, repair and update operations in parallel. However, in order to use openMP directives effectively, we needed to implement encoding/decoding operations differently. We use ‘-m’ flag to set the number of threads in the main functions of Founsure. This parameter can independently be assigned but we provide recommendations for picking out the right number of threads for each main function because selecting the wrong number could result in degraded performance. For instance, we recommend it to be equal to the number of failure domains (disks for instance) for Founsure. This parameter can independently be assigned but we provide recommendations for picking out the right number of threads for each main function because selecting the wrong number could result in degraded performance. For instance, we recommend it to be equal to the number of failure domains (disks for instance) for founsureEnc so that each data block is generated by a different thread. Considering each output file gets written to a different disk or storage node, we can maximize the overall throughput of the system if ‘-m’ and ‘-s’ flags are set to the same number provided that the underlying CPU architecture supports that many concurrent threads.

In founsureDec, remember that we use Algorithm 1 to resolve the user data in an iterative manner. Suppose that for a maximum of convergent \( t_m \) iterations, we decode a set of data symbols (also known as the ripple size) \( G_i \) at the \( i \)th iteration for \( i \in 0, 1, \ldots, t_m \). However, we note that a decoded data symbol \( s \in G_i \) might be using another symbol \( h \in G_i \) while decoding, which results in intra-iteration decoding dependency. If we let \( j \)-th iteration to use only decoded symbols in \( \bigcup_{i=0}^{j-1} G_i \), this would lead to another decoded set sequence \( G_0, G_1, \ldots, G_{t_m} \) where \( G_0 = G_0 \) and \( t_m > t_m \). Note that upon convergence, we should have \( \bigcup_{i=0}^{t_m} G_i = \bigcup_{i=0}^{t_m} G_i \). Although this new delayed BP will converge late compared to original version with single thread, this observation is not necessarily true with multi-threads as \( G_i s \) can be computed by multiple threads because data symbols in \( G_i \) are decoded completely independent of each other and decoding process for each only share data for read operations eliminating potential race conditions. We note that for a given \( k \), if we increase the block length \( n \) we would need less number of iterations i.e., smaller \( t_m \) and \( t_m \) with larger ripple sizes in each iteration. Finally, we note that we have many calls of Algorithm 1 for decoding independent partitions of the user data. Using shared memory approach, we use multi-threading to compute \( G_i \) in parallel in each iteration. Thus, the larger is ripple size, the better would become the performance of our implementation. It is recommended to use more threads as the number of coding blocks \( n \) increase. We also recommend to test the best number of threads for a given \( n \) to find the optimal value because this number is heavily dependent on the degree distribution. Finally the multi-threaded implementation of repair/update operations is similar to decoding since in both cases, we resolve the repaired/updated data using BP algorithm. For founsureRep function, the recommended number of threads equal to either the number of repaired coding blocks or the number of extra coding blocks generated out of an update operation.

We have two functions that takes advantage of the multi-core multi-threaded systems and carries out the main operations of Founsure in parallel. These functions are EncodeCompute-
Algorithm 5 Decoding Path Generation (DPG) Algorithm

1: procedure DPG($B, E, \text{maxit}$) \hfill $\triangleright$ Inputs: $B \in \mathbb{F}_2^{k \times n}, E \subset N$: The set of erasures.
2: $N = \{1, \ldots, n\}$ \hfill $\triangleright$ Initialize indexes.
3: $K = \{1, \ldots, k\}$ \hfill $\triangleright$ Initialize indexes.
4: $i \leftarrow 0, DP \leftarrow \emptyset$ \hfill $\triangleright$ Initialization: $i$ holds the unrecovered indexes.
5: $C \leftarrow b_{\cap N \setminus E}$ \hfill $\triangleright$ Find survival matrix.
6: $F \leftarrow K$ \hfill $\triangleright$ Initialization: $F$ holds the unrecovered indexes.
7: while $F \neq \emptyset$ and $i < \text{maxit}$ do
8: \hspace{1em} $c_{K \setminus F} \leftarrow 0$ \hfill $\triangleright$ Zero-out rows for decoded symbols.
9: \hspace{1em} $\mathcal{G}_i \leftarrow \{g : \text{weight}(c_{\cdot g}) = 1 \text{ for } g = 1, \ldots, |N \setminus E|\}$ \hfill $\triangleright$ Find degree-1 coding symbols.
10: \hspace{1em} $DP \leftarrow DP \cup \{(f, g) : \text{For each } f \in F, g \in \mathcal{G}_i \text{ s.t. } c_{f,g} = 1 \text{ and } g_d \text{ is minimum.}\}$ \hfill $\triangleright$ Update $F$
11: \hspace{1em} $F \leftarrow \{f : \text{weight}(c_{f, \mathcal{G}_i}) = 0 \text{ for } f \in F\}$ \hfill $\triangleright$ Update $F$
12: \hspace{1em} $i \leftarrow i + 1$
13: return $F$ \hfill $\triangleright$ Return unrecovered indexes.

Fast\_mt for performing the data encoding and generating output files simultaneously and runBP\_mt which runs the BP algorithm as parallel as possible. In our revised BP implementation, we remove the intra-iteration decoding dependency (see also Algorithm 5) by

- allowing BP to proceed using only the decoded symbols in $\bigcup_{i=0}^{j-1} \mathcal{G}_i$ in $j$th iteration or decoding step,
- allowing only one particular (lowest-degree) coding symbol to decode each source symbol in $\mathcal{G}_j$,

where the latter eliminates the possibility of race conditions (double writes by different threads) and optimizes the complexity performance by reducing the number of XOR operations. Note that as the number of failures increase, the number of coded symbols decrease and hence finding lowest degree coding symbol will not usually end up with much performance improvement. We finally note that, since different threads deal with different level of workloads, we use dynamic scheduling of threads in openMP.

3.2. Optimal Decoding Path Generation

In this section, we assume that degree and selection distributions of Founsure is determined by different requirements of the system and $\mathcal{D}P$ represents the set of source and coding symbol pairs in which the coding symbol is used to decode the paired up source symbol. Thus, in the case of convergence of BP, we should expect $|\mathcal{D}P| = 2k$. The elements of $\mathcal{D}P$ is found according to Algorithm 5. A careful look at the algorithm reveals that the following line does the local optimization of finding the lowest-degree coding symbol that decodes a specific source symbol.

$$\mathcal{D}P \leftarrow \mathcal{D}P \cup \{(f, g) : \text{For each } f \in F, g \in \mathcal{G}_i \text{ s.t. } c_{f,g} = 1 \text{ and } g_d \text{ is minimum.}\} \quad (7)$$

One another note about Algorithm 5 is that by keeping unrecovered symbols in $F$, we do not allow the same source symbol to be decoded more than once. This leads to suboptimality but helps us with multi-threaded implementation since it will save us from dealing with race conditions which would otherwise be handled with time consuming locks. Also comparing it with Algorithm 1, we can observe that symbol decodings are done one iteration at a time and hence symbols that are decoded at a given iteration do not help with other symbols that could have potentially be decoded within the same iteration. This approach is adapted to help with shared memory implementation of the previous subsection.
4. Illustrative Examples

Here are the set of commands to use encoding, decoding, repair and update features of Founsure 1.0. Note that Founsure 1.0 comes with man pages or you can always use “-h” flag command for immediate help when you call Founsure functions.

The following command will encode a test file `testfile.txt` with \( k = 500 \) data chunks with each chunk occupying \( t = 512 \) bytes. The encoder generates \( n = 1000 \) coding chunks using \( d = \text{FiniteDist} \) degree distribution and \( p = \text{ArrayLDPC} \) precoding. Finally, generated chunks are striped/written to \( s = 10 \) distinct files for default disk/drive allocation under `/Coding` directory\(^1\). The flag “-v” is used to output parameter information used during the encoding operation. Founsure encoder also generates a metadata file with critical coding parameters which will later be useful for decoding, repair and update operations. Without an appropriate metadata, Founsure cannot operate on files.

```bash
founsureEnc -f testfile.txt -k 500 -n 1000 -t 512 -d 'FiniteDist' -p 'ArrayLDPC' -s 10 -v
```

Now, let us erase one of the coding chunks and run Founsure decoder. The decoder shall generate a decoded file `test_file_decoded.txt` under `/Coding` directory. You can use “diff” command to compare this file with the original.

```bash
rm -rf Coding/testfile_disk0007.txt
founsureDec -f testfile.txt -v
```

One of the things we notice about `founsureDec` function is that it does not recover the lost drive data `Coding/testfile_disk0007.txt`, because this function is responsible only for the original data recovery process. In storage systems however, we need to recover lost data to keep the system data reliability at an acceptable level. In Founsure 1.0, it is extremely easy to initiate repair (current version only supports exact repair at the moment) process by running the following command.

```bash
founsureRep -f testfile.txt -v
```

This would trigger the conventional repair operation and first shall decode the entire data and then re-run partial encoding to generate the lost chunks. `founsureRep` outputs the pure computation speed as well as the bandwidth consumed due to repair. If you observe carefully, conventional repair is heavily time and bandwidth consuming operation. In fact, due to non-optimal overhead, the number of bytes that need to be transferred for the conventional repair is little larger than the size of the original user file. Founsure 1.0 supports fast and efficient repair as well. In order to use this feature, one needs to modify the metadata file and create an extra helping data/file called `testfile_check.data` which shall contain information for fast repair. Details can be found later in the document. In order to make these changes, we primarily run `genChecks` function. Finally, we can re-run the repair function as before and you will realize from the comments printed out that the function will be able to recognize that there is available information for fast/efficient repair and will run that process instead of switching to conventional repair. You should be able to observe the reduced bandwidth consumed by the repair operation.

```bash
genChecks -f testfile.txt -m 1 -v
founsureRep -f testfile.txt -v
```

We can also use `genChecks` to trigger ‘update’ functionality. For example, let us assume

\(^1\)This directory naming is conventional and maintained for the legacy of Jerasure erasure code library.
that the system reliability is degraded due to drive wear and we want to generate extra two drive-worth information in addition to already generated 10 drive information. We use `-e' flag to modify metadata file as well as testfile_check.data for the update operation. This shall change the code and all its related parameters. However in order to apply it to encoded data, we shall use founsureRep to generate new coding chunks and output files. Alternatively, you can erase drive info as well by supplying negative values for `-e' flag. In this case, you do not need to call founsureRep because there is nothing to generate. You can simply erase corresponding drive chunks after you scale the system down.

```bash
genChecks -f testfile -m 1 -v -e 2
founsureRep -f testfile.txt -v
```

5. Numerical Results and Impact

Our tests are run on a server system the details of which are given in Table 1. To be able to draw a summary of library performance, we provide Table 2 for a quantification of Encoding/Decoding speed. In our test, we used a 64MiB file and encoded data are spread across 10 disks equally. We have used multi-threading support and set `-m' parameter to 12. While decoding, we have removed 2, 3 and 4 disk worth of information before running decoder. The `-t' parameter is judiciously chosen in powers of two to enable fast operation. No exhaustive optimization is carried out. As can be seen, with almost a halve code rate, we could achieve super fast encoding and decoding speeds with the current implementation.

While complexity performance of the library is attractive, we can also show that it is also bandwidth friendly when the data is repaired. We considered the case $k = 10246$ and $n = 21800$ with a 100MiB = 104,857,600 bytes file while all the rest of the parameters are the same as before. If we consider double disk failures, the conventional repair method requires us to

| Property    | Name: Intel Xeon CPU E5-2620 |
|-------------|------------------------------|
| Socket      | 2                            |
| Core        | 12                           |
| Threads     | 24                           |
| Arc.        | IA64                         |
| L1 cache    | 32KiB                        |
| L2 cache    | 256KiB                       |
| L3 cache    | 15360KiB                     |
| Main memory | $\approx 8$GiB              |

| $k$ | $n$ | rate | $t$ | Encoder Performance (MB/sec) | Failure # | Decoder Performance (MB/sec) |
|-----|-----|------|-----|-------------------------------|-----------|-------------------------------|
| 1036| 2180| 0.4752| 1024| 2722                          | 2         | 3291                          |
|     |     |      |     |                               | 3         | 2852                          |
|     |     |      |     |                               | 4         | 2340                          |
| 10246| 21800| 0.47 | 512 | 1691                          | 2         | 2668                          |
|     |     |      |     |                               | 3         | 2496                          |
|     |     |      |     |                               | 4         | 2356                          |
transfer 108,267,520 bytes of data for the repair to be successful. This is a little over 100MiB as expected due to overhead suboptimality of Founsure code. On the otherhand, if we use the improved repair scheme suggested in this study, we can achieve a maximum of 65,952,320 bytes of transfer for successfully recovery, which is almost 2× more efficient use of bandwidth over that of the conventional method. Note again that no optimization is performed in terms of degree distribution $\Omega(x)$ and advanced graph partitioning to minimize transfer size.

To best of my knowledge, Founsure is the most flexible erasure coding library that is open source and can be configured based on the requirements of the application. With the current software architecture, many more functionalities can be integrated such as partial user data construction and advanced error detection for failure localization. In addition, as numerical results suggest even if many more optimizations are possible to make the performance better, the current version’s performance in terms of computation complexity and repair bandwidth still stand out. Founsure has highly parallel architecture and lends itself to parallel programming. Unlike Founsure, existing research is mostly focused on overhead optimal designs using the parallel hardware. Originally, Founsure is developed for data storage systems, it can simply be adapted to packet switched networks in which the underlying channel is erasure channel or sporadic erasure channels [22]. With advanced features such as error detection, multi-threaded implementations, advanced decoding, Founsure can further be used for error correction which would open up more fields of its application such as image reconstruction and data protection over noisy communication channels. Consequently, we believe that Founsure could be a strong candidate to be used for any system that secures data protection and recovery in one way or another.

6. Conclusions

We developed an erasure coding library that can be used to solve the tradeoff between computation complexity, coding overhead and repair bandwidth. Founsure can be thought as the base software on which we can put lots of different features that will make it more application-centric and configurable for future generation reliable system requirements.

Acknowledgements

I would like to thank THE SCIENTIFIC AND TECHNOLOGICAL RESEARCH COUNCIL OF TURKEY (TUBITAK) and Quantum Corporation who provided the support for maturing ideas, as well as hardware platforms for extensive testing.

References

References

[1] B. Nisbet, “FAS Storage Systems: Laying the Foundation for Application Availability,” Network Appliance white paper: http://www.netapp.com/us/library/analyst-reports/ar1056.html February, 2008.

[2] I. S. Reed and G. Solomon, “Polynomial codes over certain finite fields,” Journal of the Society for Industrial and Applied Mathematics, 8, 1960, pp. 300-304.

[3] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby and D. Zuckerman, “An XOR-Based Erasure-Resilient Coding Scheme,” Technical Report TR-95-048, International Computer Science Institute, August, 1995.

[4] M. Blaum, J. Brady, J. Bruck and J. Menon, “EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures,” IEEE Transactions on Computing, 44(2), February, 1995, pp. 192-202.
[5] M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, “Xoring elephants: Novel erasure codes for big data”, Proceedings of the VLDB Endowment (PVLDB), 6(5), 2013.

[6] A. Partow, “Schifra Reed-Solomon ECC Library,” Open source code distribution: http://www.schifra.com/downloads.html, 2000-2007.

[7] J. S. Plank, S. Simmerman, and C. D. Schuman, “Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2,” University of Tennessee, Tech. Rep. CS-08-627, August 2008.

[8] CEPH software-defined storage software. Available online: http://www.ceph.com.

[9] Z. Wilcox-O’Hearn, “Zfec 1.4.0,” Open source code distribution: Available online: http://pypi.python.org/pypi/zfec, 2008.

[10] S. S. Arslan, T. Goker and R. Wideman, “A Joint Dedupe-Fountain Coded Archival Storage,” IEEE ICC’17, Paris, France, May 2017.

[11] J. Pearl, “Reverend Bayes on inference engines: A distributed hierarchical approach.” AAAI-82: Pittsburgh, PA. Second National Conference on Artificial Intelligence, Menlo Park, California: AAAI Press. pp. 133–136. 1982.

[12] D. J. C. MacKay, “Fountain codes,” in IEE Proc. Commun., vol. 152, pp. 1062–1068, Dec. 9, 2005.

[13] M. Luby, “LT-codes,” in Proc. 43rd Annu. IEEE Symp. Foundations of Computer Science (FOCS), Vancouver, BC, Canada, Nov. 2002, pp. 271–280.

[14] A. Shokrollahi, “Raptor codes,” IEEE Trans. on Information Theory, June 2006.

[15] D. Lee, H. Park and J. Moon, “Reducing Repair-Bandwidth Using Codes Based on Factor Graphs”, IEEE International Conference on Communications, Kuala Lumpur, Malaysia, 2016

[16] S. S. Arslan “Incremental redundancy, fountain codes and advanced topics,” 2014. Available online: arXiv:1402.6016.

[17] A. J. Felstrom and K. S. Zigangirov, “Time-varying periodic convolutional codes with low-density parity-check matrix,” IEEE Trans. Inf. Theory, vol. 45, no. 6, pp. 2181–2190, Sep. 1999.

[18] J. L. Fan, “Array codes as low-density parity-check codes,” in Proc. 2nd International Symposium on Turbo Codes and Related Topics, Brest, France, Sept. 2000, pp. 543-546.

[19] S. S. Arslan, “Implementation of multi-threaded erasure coding under multi-processing environments”, 24th Signal Processing and Communication Application Conference (SIU) 2016.

[20] S. S. Arslan, H. Le, J. Landman, and T. Goker, “OpenMP and POSIX thread implementation of Jerasure 2.0,” IEEE BlackseaCom’17, Istanbul, Turkey, 2017.

[21] E. Eleftheriou and S. Olcer, “Low-density parity-check codes for digital subscriber lines,” in Proc. IEEE Int. Conf. Communications (ICC 2002), vol. 3, 2002, pp. 1752–1757.

[22] G. Liva, E. Paolini, B. Matuz, M. Chiani, “A Decoding Algorithm for LDPC Codes Over Erasure Channels with Sporadic Errors,” Allerton Conf. on Commun. Cont. and Comput., 2010.
| Nr. | Code metadata description                                      | Please fill in this column                                      |
|-----|----------------------------------------------------------------|----------------------------------------------------------------|
| C1  | Current code version                                           | v10                                                             |
| C2  | Permanent link to code/repository used for this code version   | For example: [https://github.com/suaybarslan/founsure](https://github.com/suaybarslan/founsure) |
| C3  | Legal Code License                                             | LGPLv3                                                           |
| C4  | Code versioning system used                                    | git                                                             |
| C5  | Software code languages, tools, and services used              | C, Python, OpenMP, etc.                                         |
| C6  | Compilation requirements, operating environments & dependencies| ANSI C89 and C++98; Unix, Linux, MacOS, Windows                  |
| C7  | If available Link to developer documentation/manual           | For example: [http://www.suaybarslan.com/founsure.html](http://www.suaybarslan.com/founsure.html) |
| C8  | Support email for questions                                    | arslans@mef.edu.tr                                               |

Table 3: Code metadata (mandatory)