Parallelism-Aware Locally Repairable Code for Distributed Storage Systems

Jun Li
School of Computing and Information Sciences
Florida International University

Baochun Li
Department of Electrical and Computer Engineering
University of Toronto, Canada

***Abstract*—Distributed storage systems store a substantial amount of data in a large number of servers built with commodity hardware. In order to protect data against server failures, erasure coding has been deployed in many distributed storage systems because of its low storage overhead. In particular, since disk I/O is, in many cases, a bottleneck in the distributed storage system, locally repairable codes, have been proposed that incur low volumes of disk I/O when reconstructing missing data after server failures. However, since original data can only be read from specific servers, existing designs of locally repairable codes suffer from limited data parallelism. Besides, if the performance of servers is heterogeneous, slow servers may become the bottleneck when accessing data in parallel. In this paper, we propose *Galloper* codes, a novel family of locally repairable codes, that achieve low disk I/O during reconstruction and meanwhile extend data parallelism from specific servers to all servers. Moreover, the amount of original data in each server can be arbitrarily determined based on the performance of corresponding servers. We have implemented a prototype of Galloper codes on Apache Hadoop, and our experimental results have shown that Galloper codes can reduce the completion time of MapReduce jobs by up to 42.9%, with a comparable performance as existing locally repairable codes, in terms of disk I/O overhead, as well as encoding and reconstruction overhead.**

***Keywords*-distributed data storage, locally repairable code, pyramid code, parallelism, MapReduce**

I. Introduction

Distributed storage systems, such as Windows Azure Storage [3] and the Hadoop distributed file system (HDFS) [29], provide storage services with a substantial capacity by storing data over a large number of commodity servers. Due to the large number of such servers and their commodity nature, it is common to observe failures of servers in distributed storage systems [26], [22]. Therefore, it is critical for distributed storage systems to maintain data availability despite server failures.

Due to frequent server failures, distributed storage systems must store redundant data, such that data can be obtained from a subset of servers. In distributed storage systems, redundant data are usually generated by replicating data on multiple servers. For example, data are replicated three times in HDFS by default, and thus any two servers failures can be tolerated.

Comparing with replication, erasure coding can provide the same level of failure tolerance with much less storage overhead and thus it has been deployed in many practical distributed storage systems [11], [2], [32], [36], [5]. Taking Reed-Solomon codes — the most common choice — as an example, with a (k, r) Reed-Solomon code which takes k blocks of original data (*data blocks*) as input and computes r blocks of parity data (*parity blocks*) as output, we can get all original data from any k among the total $k + r$ blocks. Typically, such $k + r$ blocks are placed on different servers to maximize the tolerance to failures, and hence any r server failures can be tolerated. Therefore, we can tolerate any two server failures with either 3-way replication or a $(4, 2)$ Reed-Solomon code. However, the $(4, 2)$ Reed-Solomon code incurs only 1.5x storage overhead while 3x storage overhead is required by 3-way replication.

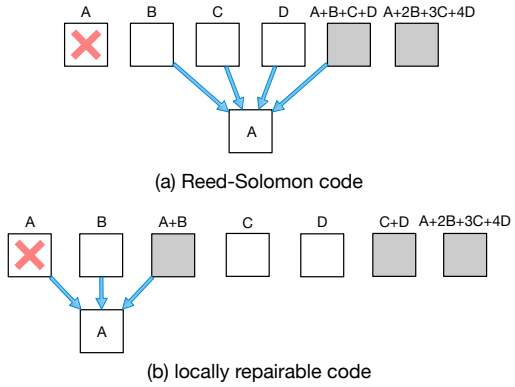

Figure 1. A comparison of disk I/O during reconstruction between a Reed-Solomon code and a locally repairable code, where we use white and gray blocks to present data blocks and parity blocks, respectively.

However, Reed-Solomon codes can incur high volumes of disk I/O when unavailable data need to be *reconstructed* after a server failure. As shown in Fig. 1a, with a Reed-Solomon code, when a block A becomes unavailable, we cannot directly reconstruct it by copying one of its copies as with replication, since there is no any copy of A among existing blocks. Instead, we need to download four other blocks from existing servers, even though there is only one block to reconstruct. Since disk I/O is the performance bottleneck in many cases, alternative designs of erasure codes, called *locally repairable codes* have been proposed such that a block can be reconstructed from a small number of other blocks. In Fig. 1b, we show an example of locally repairable

codes[1], that can tolerate the same number of failures as the Reed-Solomon code in Fig. 1a. Nevertheless, we can reconstruct the same block from just two blocks, saving 50% disk I/O by just adding two additional parity blocks.

Although locally repairable codes can save storage overhead as well as disk I/O overhead during reconstruction, it still limits data parallelism of data analytical jobs running on corresponding data as other conventional erasure codes [14]. Typically, a data analytical job takes advantage of data parallelism by running multiple tasks on different servers at the same time, and these tasks are usually scheduled to run on servers where the corresponding input data are stored. Therefore, comparing with replications where parallel tasks can run on all servers that store copies of the data, the parallelism of erasure coding cannot be extended to servers that store parity blocks unless additional network transfers are allowed. As shown in Fig. 2a, assume that we are running a MapReduce job over the data encoded by the locally repairable code in Fig. 1b. If the input data are stored in a distributed storage system, *e.g.*, in HDFS, the number of map tasks of a MapReduce job depends on the number of available data blocks, and such map tasks will be scheduled to run on the servers that store those blocks. However, as it is difficult to run general functions on parity data, we cannot directly run map tasks on parity blocks. Therefore, we can only run map tasks on data blocks no matter how many parity blocks we have.

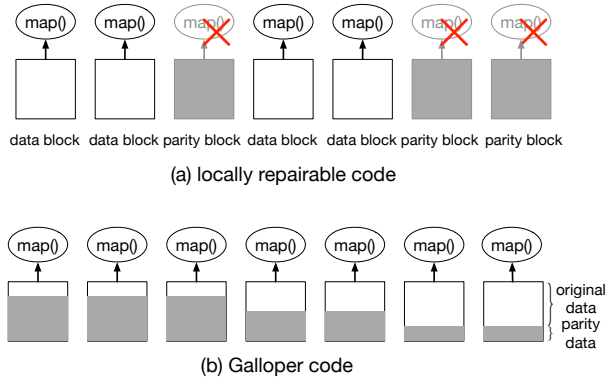

Figure 2. A comparison of data parallelism between locally repairable codes and Galloper codes, where we still use white and gray color to represent the original and parity data.

Existing work such as the Carousel code [14] has considered data parallelism by placing original data evenly on all servers. However, Carousel codes have two major drawbacks. First, a high volume of disk I/O will still be incurred during reconstruction with Carousel codes. Second, as multiple hardware configurations of servers exist in a typical data center [24], performance heterogeneity of servers in a distributed storage system is common, and thus evenly placing original data on all servers can lead to additional bottlenecks on slower servers.

In this paper, we propose *Galloper* codes, that can extend data parallelism from data blocks only to all blocks, while still maintaining the same properties of existing locally repairable codes during reconstruction. Different from conventional erasure codes, Galloper codes carefully embed original data into all blocks such that parallel tasks can be launched on any servers that store coded blocks. Moreover, we have also considered the performance heterogeneity of servers. In Fig. 2b, with Galloper codes, we can arbitrarily determine the amount of original data placed on all servers, based on the performance of the corresponding server.

We have implemented Galloper codes in C++ and developed a prototype with Apache Hadoop. Our experimental results show that Galloper codes can significantly improve the performance of data analytical jobs running on the coded data, while still maintaining desirable properties of locally repairable codes.

II. Related Work

In many distributed storage systems, Reed-Solomon codes have become a popular choice to generate redundant data and then tolerate potential failures [11], [2], [32], [36], [5]. In fact, Reed-Solomon codes achieve the optimal storage overhead to tolerate the same number of failures, as they are maximum distance separable (MDS) in coding theory [23]. However, (k, r) Reed-Solomon codes require k blocks to reconstruct just one block, incurring high volumes of network transfer and disk I/O. Therefore, various codes have been proposed to mitigate such overhead during reconstruction.

Dimakis *et al.* first proposed regenerating codes that optimize the network transfer during reconstruction [7]. Among the family of regenerating codes, there exists a tradeoff curve between storage overhead and network transfer during reconstruction. The two ends of such curve are termed as minimum-storage regenerating (MSR) codes and minimum-bandwidth regenerating (MBR) codes, respectively. Different from Reed-Solomon codes, regenerating codes require to contact more than k blocks to reconstruct an unavailable block, where each block only needs to offer a fraction of its data. Assuming that d existing blocks are required during reconstruction, $d \geq k$, and r failures need to be tolerated, Rashmi *et al.* have proposed constructions of MBR codes ($k \leq d < k+r$) and MSR codes ($2k-2 \leq d < k+r$) [21].

However, regenerating codes in general suffer from even higher disk I/O than Reed-Solomon codes. The reason is that even though a fraction of each block is required, such a fraction usually must be computed from the whole block. Therefore, at least k blocks must be read on the corresponding servers since $d \geq k$. Although there exist regenerating codes that are efficient in disk I/O, the constructions of such regenerating codes are usually limited to

[1]This example is a Pyramid code, a particular family of locally repairable code proposed by Huang *et al.* [12]. An improved construction of Pyramid codes has been implemented in Windows Azure Storage [11]

specific combinations of parameters [28], [27], [31], [33], or only some specific blocks can be reconstructed in this way [20].

As disk I/O is usually more scarce than network bandwidth in a distributed storage system, locally repairable codes have been proposed to minimize the disk I/O during reconstruction [30], [12], [26], [16], by allowing a block to be reconstructed from a small number of other blocks. The number of other blocks needed to reconstruct a block is termed as the *locality* of this block. If all data blocks have locality no more than l, such codes have *information locality* l. A code has *all-symbol* locality l if all blocks have locality no more than l. Gopalan *et al.* [10] established the bound of the minimum distance as a function of k, r, d and Papailiopoulos and Dimakis extended this bound by considering the size of blocks [17]. So far, there have been several designs of locally repairable codes that achieve the optimal distance with specific values of k, r, d [26], [17]. However, a general construction is still an open problem. In this paper, we propose Galloper codes that achieve the same locality as Pyramid codes [12], an existing family of local repairable codes that achieve information locality and have been deployed in Windows Azure Storage [11], and significantly improve data parallelism when running data analytical jobs on coded data.

The separation of data blocks and parity blocks makes it hard to expand data parallelism as we cannot run general functions on parity data unless the function is linear [13], [15]. Therefore, a technique called *striping* has been used in the code design such that original data can be spread into all blocks to extend data parallelism [34], [19], [4], [9], [8]. However, such designs work with MDS codes only and still suffer from high overhead during reconstruction. Li *et al.* have proposed Carousel codes to achieve high data parallelism and optimal network transfer during reconstruction simultaneously [14]. However, the Carousel code still requires high disk I/O during reconstruction as Reed-Solomon codes. The Galloper codes proposed in this paper focuses on low disk I/O during reconstruction and high data parallelism at the same time.

Meanwhile, it has been reported that the hardware of machines in a data center can have various configurations [24], and server heterogeneity can lead to system unavailability and errors [25]. Therefore, it is critical for data analytical jobs to consider the performance of heterogeneity of servers, by considering scheduling of jobs [35], [1], or through task configurations [6]. However, such designs typically do not consider how data are stored in distributed storage systems, and thus cannot work with distributed storage systems that deploy erasure coding. In this paper, Galloper codes can directly work with servers with heterogeneous performance, and thus can directly work with distributed analytical jobs or systems without change of code. We demonstrate that Galloper codes can be easily integrated into Hadoop and improve the performance of MapReduce jobs running on heterogeneous servers.

III. Preliminaries

A. Reed-Solomon Code

We assume that all data are stored in *blocks* of the same size, which is a common practice in distributed storage systems [29]. A (k, r) Reed-Solomon code encodes k data blocks into r parity blocks, and all the $k + r$ blocks will be distributed into $k + r$ different servers. In other words, assuming that the k original blocks contain M bytes, they can be represented as k vectors[2] of length M, *i.e.*, $F_1, \ldots, F_k$, and then the r parity blocks, $F_{k+1}, \ldots, F_{k+r}$, can be computed by a *generating matrix* G of size $(k+r)\times k$, *i.e.*,

$$\left[F_1^T \ \cdots \ F_{k+r}^T\right]^T = G \cdot \left[F_1^T \ \cdots \ F_k^T\right]^T.$$

In this case, the upper k rows of G constitute an identity matrix, and therefore the original data are embedded into the first k blocks of the results. If all original data are still embedded into the results after multiplying with the generating matrix, such erasure codes is known as *systematic*. Although there also exist non-systematic Reed-Solomon codes where all $k + r$ blocks are parity blocks, we focus on systematic codes in this paper as original data are essential to data parallelism.

To decode data from any k blocks, any k rows of the generating matrix G should be linearly independent. Assume that g_i is the i-th row of G. With any k blocks $F_{i_1}, \ldots, F_{i_k}$, where $\{i_1, \ldots, i_k\}$ is a k-subset of $\{1, \ldots, k + r\}$, the k original blocks can be decoded as

$$\left[F_1^T \ \cdots \ F_k^T\right]^T = \left(\left[g_{i_1}^T \ \cdots \ g_{i_k}^T\right]^T\right)^{-1} \cdot \left[F_{i_1}^T \ \cdots \ F_{i_k}^T\right]^T.$$

In other words, any k blocks among the total $k + r$ blocks can be decoded into the original data. However, when any block becomes unavailable, we still need to obtain k other blocks to reconstruct it, *i.e.*,

$$F_j = g_j \cdot \left(\left[g_{i_1}^T \ \cdots \ g_{i_k}^T\right]^T\right)^{-1} \cdot \left[F_{i_1}^T \ \cdots \ F_{i_k}^T\right]^T.$$

B. Pyramid Code

From Reed-Solomon codes we can construct Pyramid codes [12], a particular family of locally repairable codes. A (k, l, g) Pyramid code, where $l|k$, contains k data blocks and $l + g$ parity blocks, including l *local parity* blocks and g *global parity blocks*. The g global parity blocks are computed from the k data blocks with a (k, g) Reed-Solomon codes. Therefore, when $l = 0$, a Pyramid code becomes a Reed-Solomon code. For example, in Fig. 1a, the $(4, 2)$ Reed-Solomon code is a special case of the $(k = 4, l = 0, g = 2)$ Pyramid code.

[2]Each element in the vector is a symbol on a finite field. In this paper, we do not rely on any direct knowledge of finite filed and readers can simply consider its arithmetic as usual arithmetic unless mentioned otherwise.

As $l|k$, we also compute each local parity block from every k/l data blocks, with a $(k/l, 1)$ Reed-Solomon code. Fig. 1b shows an example of such a construction with $k = 4, l = 2, g = 1$. Therefore, all data blocks and local parity blocks can be reconstructed from k/l other blocks. For example, from B and $A + B$ we can reconstruct A directly. Only the global parity block needs to contact four other blocks.

It can be proved that a (k, l, g) Pyramid code can tolerate any $g+1$ failures, if $l \geq 0$. It is also possible to tolerate more than $g+1$ failures but not all combinations of such failures. For example, in Fig. 1b, if A, B, and $A + 2B + 3C + 4D$ are all unavailable, the rest blocks can not be decoded. Comparing with Reed-Solomon codes that can tolerate any failures of the same number, the introduction of local parity blocks in a Pyramid code brings slightly more storage overhead but also makes it possible to save the number of blocks required to reconstruct a block on a failed server. In this paper, we propose Galloper codes that can reconstruct the same block by visiting the same blocks as Pyramid codes, while the original data can be placed into all blocks instead of just data blocks.

C. Symbol Remapping

In this paper, we use the technique called *symbol remapping*, which has been used in the constructions of some erasure codes [21], [20], [14]. Symbol remapping changes the basis of the generating matrix of a erasure code and then make a new code that is linearly equivalent as the original one and maintains its original properties.

We use Carousel codes [14] as an example to explain how symbol remapping works, which "moves" original data from data blocks to all blocks. Since each block will need to contain both original data and parity data, it is necessary to expand the original generating matrix G at first, such that

$$G_g = \begin{bmatrix} g_{1,1}I_{k+r} & \cdots & g_{1,k}I_{k+r} \\ \vdots & \ddots & \vdots \\ g_{k+r,1}I_{k+r} & \cdots & g_{k+r,k}I_{k+r} \end{bmatrix},$$

where $g_{i,j}$ corresponds to an element in G. Then we can rewrite the encoding process by splitting each block into $k + r$ stripes, *i.e.*,

$$\begin{aligned}\begin{bmatrix} F_{1,1}^T & \cdots & F_{1,k+r}^T & \cdots & F_{k+r,1}^T & \cdots & F_{k+r,k+r}^T \end{bmatrix}^T = \\ G_g \cdot \begin{bmatrix} F_{1,1}^T & \cdots & F_{1,k+r}^T & \cdots & F_{k,1}^T & \cdots & F_{k,k+r}^T \end{bmatrix}^T,\end{aligned}$$

where $F_{i,1}, \ldots, F_{i,k+r}$ are the $k + r$ stripes equally split from the block F_i. A Carousel code then chooses k stripes from each block which can be written as

$$\begin{aligned}\begin{bmatrix} F_{1,i_{1,1}}^T & \cdots & F_{1,i_{1,k}}^T & \cdots & F_{k+r,i_{k+r,1}}^T & \cdots & F_{k+r,i_{k+r,k}}^T \end{bmatrix}^T = \\ G_{g0} \cdot \begin{bmatrix} F_{1,1}^T & \cdots & F_{1,k+r}^T & \cdots & F_{k,1}^T & \cdots & F_{k,k+r}^T \end{bmatrix}^T,\end{aligned}$$

where $\{i_{j,l} | l = 1, \ldots, k\}$ is a k-subset of $\{1, \ldots, k + r\}$. By carefully choosing the k-subsets for all j, G_{g0} can be a non-singular matrix and then we can rewrite all stripes as a linear combination of the $k(k + r)$ chosen stripes. Since G_{g0} can be used as a new basis of the generating matrix, after a linear combination by using G_{g0} as the basis, the original Reed-Solomon code can be converted to a Carousel code that maintains the original properties of Reed-Solomon codes:

$$\begin{aligned}\begin{bmatrix} F_{1,1}^T & \cdots & F_{1,k+r}^T & \cdots & F_{k+r,1}^T & \cdots & F_{k+r,k+r}^T \end{bmatrix}^T = G_g G_{g0}^{-1} \cdot \\ \begin{bmatrix} F_{1,i_{1,1}}^T & \cdots & F_{1,i_{1,k}}^T & \cdots & F_{k+r,i_{k+r,1}}^T & \cdots & F_{k+r,i_{k+r,k}}^T \end{bmatrix}^T,\end{aligned}$$

and $G_g G_{g0}^{-1}$ becomes the generating matrix of Carousel codes. It can be proved that there will be k stripes that are the same as the original data and in this way the original data are evenly placed into all blocks [14].

D. Possible Methods and Challenges

As mentioned above, Carousel codes incur the same volumes of disk I/O during reconstruction as Reed-Solomon codes, since it is linearly equivalent to the original Reed-Solomon codes. Second, Carousel codes cannot work well with heterogeneous servers, as original data are evenly placed in each block. Therefore, the objectives of Galloper codes in this paper is to provide data parallelism corresponding to the performance of each server, while still incurring low disk I/O during reconstruction as Pyramid codes. However, the method used in Carousel codes cannot be directly applied to achieve such objectives. The reason is that to maintain the reconstruction properties, the new code needs to be linearly equivalent to a Pyramid code. However, we cannot expand an existing Pyramid code and find stripes from all blocks that can be decoded into all original data, because different from Reed-Solomon codes, we cannot decode the original data from *any* k blocks. Therefore, a new method is needed to maintain the existing properties of Pyramid codes and be applicable to servers with heterogeneous performance.

Another possible method, which has been used in RAID, is to cyclically rotate the placement of stripes among blocks. For example, in RAID-5 [18] which deploys a $(4, 1)$ Reed-Solomon code, there are five blocks where each block contain five stripes. If we place the five blocks in a row, the five stripes in the five blocks will also be placed in five rows. For the stripe in the same row, they are computed from the $(4, 1)$ Reed-Solomon code, containing four data stripe and one parity stripe. The parity stripes in the five rows are cyclically rotated such that one block contains only one parity stripe. Therefore, each block contains four data stripes and still maintain the failure tolerance of the $(4, 1)$ Reed-Solomon codes. This method can work for some simple cases like Reed-Solomon codes and may be extended for heterogeneous servers. However, if it is applied

with Pyramid codes, it will break the original properties of Pyramid codes. For example, when we need to reconstruct one block, we will have to visit different blocks as each stripe in a block cannot be reconstructed from any other stripes but from some specific stripes, and at different rows such stripes can be placed in different blocks. This is particularly undesirable for archival data where servers with no access are typically put into sleep. While Pyramid codes originally only need to wake up a small number of servers, simply rotating stripes may need to wake up more servers or even all servers in the worst case. It also increases the complexity of decoding for the same reason. In this paper, the method that we propose for Galloper codes will conquer this problem and maintain the original properties of Pyramid codes in terms of locality and failure tolerance.

IV. Galloper Codes: A Special Case

We now present the construction of Galloper codes, with the objective of extending data parallelism from only the data blocks to all blocks, while maintaining the existing failure tolerance and locality of Pyramid codes. Therefore, Galloper codes are still associated with three parameters: k, l, g. Moreover, a (k, l, g) Galloper code have the same failure tolerance and locality as a (k, l, g) Pyramid code.

A. System Model and Example

Given a (k, l, g) Galloper code, there will be $k + l + g$ blocks in total, including k data blocks, l local parity blocks, and g global parity blocks. Among the total $k + g + l$ blocks, Galloper codes allow to associate each block with a weight that corresponds to the performance of its server. For example, we can use the throughput of sequential disk read as the performance measurement of the server, or the CPU processing throughput if CPU is the bottleneck for some data analytical jobs. We use p_i, $i = 1, \ldots, k + l + g$, to denote the performance measurement of the server. We then use w_i as the weight of each block. Since the original data in the input are in k blocks, and our objective is to assign such k blocks of original data into $k + l + g$ blocks corresponding to the performance of their servers, the weight will be used to indicate the ratio of original data in the corresponding block. Therefore, the ideal value of w_i should be $\frac{kp_i}{\sum_{i=1}^{k+l+g} p_i}$. However, there will be some constraints on the weights. For example, we cannot accommodate more than one block of original block in one block, *i.e.*, $w_i \leq 1$, even if the performance of some server is much higher than the others. We will elaborate on how to calculate the weight of each block without violating such constraints in Sec. IV-C, while still maintaining that servers with higher performance should store blocks with more original data.

The code construction of Galloper codes starts from a special case in this section where $l = 0$ and then be extended to the general case in Sec. V. In this case, the corresponding Pyramid code has no local parity blocks, and is equivalent to a Reed-Solomon code. We use symbol remapping to spread original data from data blocks to (global) parity blocks and assign different amount of original data in different blocks based on the performance of the server.

We now demonstrate a toy example of the code construction in Fig. 3, where $k = 4, g = 1, l = 0$. In this example, we have five blocks, in which the first four have a weight of $\frac{6}{7}$ and the last one has a weight of $\frac{4}{7}$. Therefore, eventually $\frac{6}{7}$ of the first four blocks contain the original data, so does the $\frac{4}{7}$ of the last block. In the construction of Galloper codes, each block will be divided into multiple stripes. In this example, each block contains seven stripes. The first four blocks contain six stripes of original data and the last block contains four stripes of original data. Compared with a $(4, 1)$ Reed-Solomon code, this Galloper code achieves the same failure tolerance, *i.e.,* all original data can be decoded from any four blocks. We will explain how each parity stripes are calculated in Sec. IV-B.

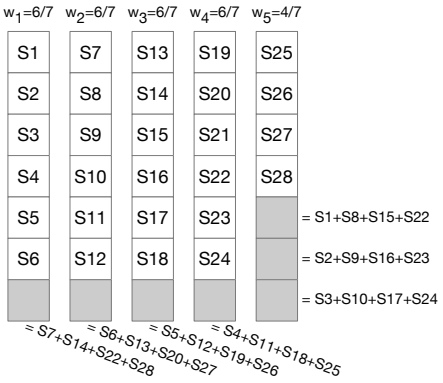

Figure 3. An illustration of Galloper codes with $k = 4, l = 0, g = 1$, where white stripes contain original data and gray stripes contains parity data.

B. Code Construction

Now we describe how to construct Galloper codes with heterogeneous block weights and $l = 0$. As there are no local parity blocks, all parity blocks mentioned in this section will directly imply global parity blocks. Given k and g, we construct the corresponding Galloper code from a (k, g) Reed-Solomon code. As shown in Fig. 4, the (k, g) Reed-Solomon code will compute g parity blocks from k data blocks, where $k = 4, g = 1$. In this case, the $(4, 1)$ Reed-Solomon code is equivalent to an XOR code, *i.e.*, the parity block is computed as an (XOR) sum of four data blocks.

As mentioned above, we first divide each block into N stripes. Since eventually in a block with weight w_i, $w_i N$ stripes will contain original data, one way to choose N is the lowest common multiple of fractions of all weights. In Fig. 4, we thus choose $N = 7$. After striping, we can see that all stripes that contain parity data are still encoded by the (k, r) Reed-Solomon code from the stripes containing

original data in the same row. In other words, any stripe can be "reconstructed" by four other stripes in the same row. For example, we have $s25 = s4 + s11 + s18 + (s4 + s11 + s18 + s25)$ at the fourth row in Fig. 4[3].

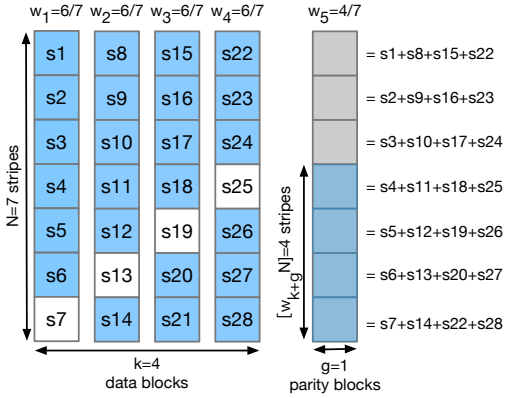

Figure 4. The change of the basis in the construction of Galloper codes with $k = 4, l = 0, g = 1$, where the highlighted stripes (in the blue color) denote the chosen kN stripes of the new basis, $N = 7$.

For convenience, we term the stripes that contain original data as data stripes, and those that contain parity data as parity stripes. In the original Reed-Solomon code, all original data are placed in the k data blocks, and all parity stripes are linear combinations of data stripes. Therefore, the set of $s1 - s28$ in Fig. 4 is a basis. The idea to convert the Reed-Solomon code into a Galloper code is to find another set of kN stripes as a new basis. After changing the basis, we will get a linearly equivalent code with the original data placed in the corresponding stripes of the new basis. Therefore, $w_i N$ stripes should be selected from the i-th block into the new basis.

We now show how to choose the $w_i N$ stripes in each block, $i = 1, \ldots, k + g$. As shown in Fig. 4, we choose the first stripe in the first block and go down sequentially to choose all the $w_1 N$ stripes. Starting from the row below the last stripe in the first block, we then sequentially choose $w_2 N$ stripes from the second block, and so on. If the stripe in the last row is chosen, we will start from the first row again when choosing the next stripe. Eventually we will choose $\sum_{i=1}^{g+k} w_i N = kN$ stripes, enough to form a basis.

In order to prove that the kN chosen stripes can be a basis, we need to show that they are linearly independent, *i.e.,* all original data can be solved from such kN stripes. Because we always choose stripes sequentially from top to bottom, by choosing the total kN stripes, we have gone through from the first row to the last row for k times. In other words, in each row there are k stripes chosen. Therefore, each data stripe that is not chosen can be decoded from the k chosen stripes in each row.

As we have proved that the chosen kN stripes can be a basis, we can perform symbol remapping by changing the basis such that the chosen stripes become data stripes. Finally, to maximize the chance of sequential data access, we can rotate the N stripes in each block upwards such that the chosen stripes stay at the top of the block. If we use $S1 - S28$ as the new labels to replace the original labels of the chosen stripes in the same sequence, we can see that the Galloper code in Fig. 3 is the code in Fig. 4 after the change of basis and the rotation of stripes.

C. Weight Assignment

Now we discuss how the weight of a block should be determined. As mentioned above, w_i should correspond to the performance of the server that stores the block. However, as the weight w_i of a block should be within $[0, 1]$. Therefore, if the performance of a server is too much higher than the rest of the servers, we should "limit" the performance of that server by d_i, $0 \le d_i \le p_i$. In other words, we assume that the actual performance of each server is $p_i - d_i$, and then the value of w_i should be $\frac{k(p_i - d_i)}{\sum_{i=1}^{k+g}(p_i - d_i)}$.

In order to maximize the overall performance, we first calculate the completion time of running identical parallel tasks on each block. As w_i is the ratio of original data in a block, then the completion time to process this block will be proportional to $\frac{w_i}{p_i - d_i} = \frac{k}{\sum_{i=1}^{k+g}(p_i - d_i)}$. Therefore, to minimize the completion time, we should maximize $\sum_{i=1}^{k+g}(p_i - d_i)$, *i.e.,* minimize $\sum_{i=1}^{n} d_i$.

We now can determine the actual performance of each server by solving the following linear programming problem.

$$\begin{aligned} \text{min.} \quad & \sum_{i=1}^{k+g} d_i \\ \text{s.t.} \quad & k(p_i - d_i) \le \sum_{i=1}^{k+g}(p_i - d_i),\ i = 1, \ldots, k+g, \\ & 0 \le d_i \le p_i,\ i = 1, \ldots, k+g. \end{aligned}$$

Then we can determine the weight of each block by letting $w_i = \frac{(p_i - d_i)k}{\sum_{i=1}^{k+g}(p_i - d_i)}$. In practice, since we choose N as the lowest common multiple of the fractions of all weights, we need to make w_i a rational number, and thus we can round up $p_i - d_i$ such that $w_i = \frac{\lceil p_i - d_i \rceil k}{\sum_{i=1}^{k+g}(\lceil p_i - d_i \rceil)}$.

V. General Construction

A. Adding Local Parity Blocks

We now discuss the general construction of Galloper codes with $l > 0$. Notice that with Galloper codes, the original data are no longer stored solely in "data blocks"; instead, they are stored in all blocks. Therefore, the global and parity blocks from the original Pyramid codes do not hold their

[3]Notice that the arithmetic operation is performed on a finite field, and in practice we use a finite field of size 2^q, $q \ge 1$. We have $A + A = 0$ on such a finite field.

original meaning any more. However, for convenience we still keep such terms to describe the blocks that correspond to the data/local parity/global parity blocks in the original Pyramid codes, as they will still be reconstructed by visiting the same blocks in Galloper codes as in Pyramid codes.

The major difference made by the l local parity blocks is that the original data cannot be decoded from *any* k blocks. Therefore, we cannot move original data into local parity blocks by still sequentially choosing stripes in local parity blocks, as in some rows we cannot use the chosen stripes to reconstruct other stripes and the chosen stripes do not become a basis. Therefore, we take two steps to construct a general (k, l, g) Galloper code. We first construct a $(k, 0, g)$ Galloper code, and then continue to move the original data to local parity blocks.

Since the original data will be moved to local parity blocks in the second step, in the first step the original data that should have been stored in local parity blocks should be kept in the data blocks. Therefore, we need to adjust the weight of data blocks such that enough data can be left in data blocks for local parity blocks.

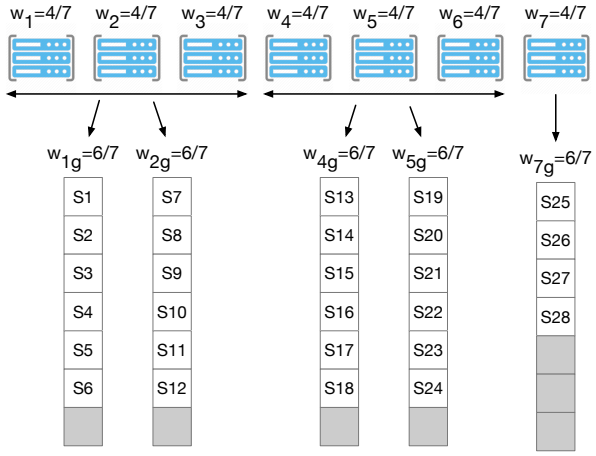

Figure 5. The assignment of weight in the construction of a $(4, 2, 1)$ Galloper code.

In the first step, we construct a $(k, l = 0, g)$ Galloper code. Since each local parity block is computed from k/l data blocks, the weight of such k/l data blocks should be $\frac{l}{k}$ of the total weights of such k/l data blocks and the local parity block. Assume we are constructing a $(k = 4, l = 2, g = 1)$ Galloper code and all blocks have the same weight, *i.e.*, $w_i = \frac{4}{7}$. In Fig. 5, we have seven servers to host blocks of the same weight $\frac{4}{7}$. Therefore, in the first step the weight of all data blocks will be $\left(\frac{4}{7} + \frac{4}{7} + \frac{4}{7}\right) \times \frac{2}{4} = \frac{6}{7}$. In order to differentiate the weight used in different steps, we use w_{ig} to represent the weight of blocks in the first step.

With the weight w_{ig} we can construct a $(k, 0, g)$ Galloper code containing k data blocks and g global parity blocks only, as shown in Fig. 5. Because in each group of k/l data blocks which will be used to compute a local parity block, their weight will be the same, we know that the number of data stripes in such data blocks will also be the same.

Now we can start the second step, which adds local parity blocks. Without loss of generality, we only consider one local parity block which is encoded from data blocks $1, \ldots, k/l$. Other local parity blocks can be constructed similarly. We have already known that after the first step, a data block will contain $w_{ig}N$ data stripes. As the k/l data blocks that we consider here have the same weight, we use w_g to represent their weights.

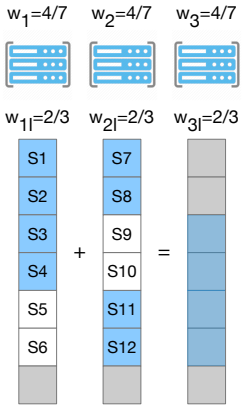

Figure 6. An illustration of adding a local parity block in the construction of a $(4, 2, 1)$ Galloper code.

We first use a $(k/l, 1)$ Reed-Solomon code to compute the local parity block directly, and then change the basis as in Sec. IV-B to assign the original data into this block. This time, the weight of each block will be assigned as $w_{il} = \frac{w_i}{w_g \times \frac{k}{l}} \times \frac{k}{l} = \frac{w_i}{w_g}$, such that the weight is still proportional to the performance of the corresponding server and the sum of weights equals $\frac{k}{l}$. Before changing the basis, stripes will be chosen only among the first $w_g N$ stripes in each block since data stripes only appear in the first $w_g N$ rows in data blocks. Therefore, in block i, $\frac{w_i}{w_g} \times w_g N = w_i N$ stripes will be chosen and eventually there will be $w_i N$ data stripes after the second change of basis.

We show an example of the second step in Fig. 6, which is a continuation of Fig. 5. We can see that since all servers have the same performance measurement, the original data which are now in the first blocks should be equally spread into the three blocks now. Therefore, the weight of the three blocks in this step is $\frac{2}{3}$. As there are only six original stripes in the first two blocks, we sequentially choose four stripes in each of the three blocks and then perform symbol remapping. Notice that the only parity stripes in the first two blocks are also subject to the change of the basis.

In this case, the only changes in the basis happen in the group of the first $k/l+1$ blocks, all other original stripes in the other blocks remain the same. Therefore, as long as we can prove that the original data stripes can be decoded from

the chosen stripes, the chosen stripes (and the data stripes in other blocks) can be a new basis. From Fig. 6 we can see that overall there will be $\sum_{i=1}^{k/l+1} w_i N = \frac{kN}{l}$ chosen. Since we choose stripes from block to block sequentially, it means that in each row $\frac{k}{l}$ stripes are chosen. As originally there are k/l data blocks, all other stripes can be reconstructed by the chosen stripes in the same row. Therefore, we can get a new basis in this way.

We change the basis in all groups of k/l data blocks and will eventually get the Galloper code in which the amount of original data inside each block is proportional to the performance measurement of the corresponding server. At last, we still rotate stripes upwards to make the original stripes appear at the top of each block.

We have known that by changing the basis the original linear dependency of vectors can be maintained. As each parity stripe is a linear combination of data stripes, it can be regarded as a scalar product of an coefficient vector and a vector of original stripes. Therefore, the original dependency in the Pyramid code can be maintained, such that the first $k+l$ blocks can be reconstructed with the other k/l blocks and the last g blocks can be reconstructed from other k blocks. Therefore, a (k, l, g) Galloper code can tolerate $g+1$ failures with $\frac{k+l+g}{k}$ times storage overhead, just like the original Pyramid code.

B. Weight Assignment

Comparing with the special case of $l = 0$, the general construction of Galloper codes introduces additional constraints on the weights of blocks. First, the weight of each block in the first step, *i.e.*, w_{ig}, should satisfy the condition that $0 \leq w_{ig} \leq 1$. Moreover, in the second step, to compute local parity blocks, the weights in each group of $k/l + 1$ blocks should also satisfy a similar condition, such that $0 \leq w_{il} \leq 1$. Given the original performance p_i of each server, such constraints can be translated to the following linear programming problem to properly lower the performance of the overqualified servers:

$$
\begin{aligned}
\text{min.} \quad & \sum_{i=1}^{k+g+l} d_i \\
\text{s.t.} \quad & l \sum_{i=j(k/l+1)+1}^{(j+1)(k/l+1)} (p_i - d_i) \leq \sum_{i=1}^{n} (p_i - d_i), \\
& \qquad\qquad\qquad\qquad\qquad\qquad j = 0, \ldots, l-1, \\
& (k/l)(p_i - d_i) \leq \sum_{i=j(k/l+1)+1}^{(j+1)(k/l+1)} (p_i - d_i), \\
& \forall i \in [j(k/l+1)+1, (j+1)(k/l+1)], j = 0, \ldots, l-1, \\
& k(p_i - d_i) \leq \sum_{i=1}^{k+g+l} (p_i - d_i), \; i = 1, \ldots, k+g+l, \\
& 0 \leq d_i \leq p_i, \qquad\qquad\qquad i = 1, \ldots, k+g+l.
\end{aligned}
$$

Similar to Sec. IV, we can round up $p_i - d_i$ to make w_i a rational number.

VI. Implementation

We have implemented Galloper codes in C++, with all coding operations performed as vector/matrix multiplications on a finite field. The size of the finite field we choose is 2^8, which corresponds to one byte. The size of the finite filed are sufficient for most values of k, l, g in practice, as long as $k + l + g < 2^8$. For larger values of k, l, g, we can also increase of the size of the finite field. In this way, data containing M bytes can be regarded as a vector of size M symbols on the finite field. To encode a file of size kM bytes, we divide this file equally into k blocks and further divide each block equally into N stripes. All the kN stripes have the same size, and we use row vector s_i, $i = 1, \ldots, kN$ to denote them. Therefore, the original data can be represented as a matrix of size $kN \times \frac{M}{N}$, *i.e.*, $\begin{bmatrix} F_{1,1}^T & \cdots & F_{1,N}^T & \cdots & F_{k,1}^T & \cdots & F_{k,N}^T \end{bmatrix}^T$.

As mentioned in Sec. III-A, the encoding of linear erasure codes, including Reed-Solomon codes, Pyramid codes, as well as Galloper codes, can be implemented as a product of a generating matrix G and the matrix of the original data, *i.e.*, $G \cdot \begin{bmatrix} F_{1,1}^T & \cdots & F_{1,N}^T & \cdots & F_{k,1}^T & \cdots & F_{k,N}^T \end{bmatrix}^T$. To get the generating matrix of a Galloper code, we start from a (k, g) Reed-Solomon code with a $(k + g) \times k$ generating matrix, and then expand this generating matrix into a $(k+g)N \times kN$ matrix G_g, by replace each element with the product of this element and a $N \times N$ identity matrix, in the same way as Carousel codes in Sec. III-C. We then perform the change of the basis by selecting a submatrix of G_g as G_{g0}. We know that each row in G_g corresponds to one stripe after encoding. Thus, we choose the rows in G_g that correspond to the stripes chosen in Sec. IV and calculate $G_g G_{g0}^{-1}$ as the generating matrix of the $(k, 0, g)$ Galloper code.

If $l > 0$, we then add lN rows into $G_g G_{g0}^{-1}$ that correspond to the l local parity blocks. To calculate the lN rows, we first get a generating matrix of a $(k/l, 1)$ Reed-Solomon code, and expand it by N times into G_{g1}. We rewrite $G_g G_{g0}^{-1}$ as $[\hat{G}_1^T \cdots \hat{G}_l^T \quad \hat{G}_{l+1}^T \cdots \hat{G}_{l+g}^T]^T$ where $\hat{G}_j$ corresponds to each of the k/l "data" blocks, $j = 1, \ldots, l$ and $\hat{G}_{l+1}, \ldots, \hat{G}_{l+g}$ correspond to the last g "global parity" blocks. We then use $G_{g1}\hat{G}_j$ to replace $\hat{G}_j$, $j = 1, \ldots, l$, and get a new matrix $\hat{G}$. Finally, we perform symbol remapping in every $k/l + 1$ blocks as described in Sec. V, still by choosing a submatrix and then multiply $\hat{G}$ with the inverse of such submatrices. Then we can get the generating matrix of the (k, l, g) Galloper code.

In our implementation, we use Intel's storage acceleration library (ISA-L) to implement the finite field operations. We have also implemented a prototype on Apache Hadoop 2.7.2 that deploys Galloper codes. In addition, we implement Reed-Solomon codes and Pyramid codes for compar-

ison purposes. In particular, we have implemented a new `FileInputFormat` class in Hadoop so that Hadoop can know the boundary between the original data and parity data in each block by the parameters of Galloper codes, and only read original data from each block when running Hadoop jobs.

VII. Evaluation

In this section, we evaluate the performance of Galloper codes by comparing with Reed-Solomon codes and Pyramid codes. We start from the performance of coding operations, including encoding, decoding, and reconstruction. We then measure the performance of running data analytical jobs on the coded data. In general, the evaluation results show that Galloper codes achieve similar performance during most coding operations as existing Pyramid codes, but significantly improve the performance of running data analytical jobs on the coded data, on both homogeneous and heterogeneous servers.

A. Performance of Encoding, Decoding, and Reconstruction

To evaluate the performance of encoding, decoding, and reconstruction, we run the corresponding operations on Amazon EC2 instances of type `c4.4xlarge`, with 16 CPU cores and 30 GB memory. In the evaluation, we compare three kinds of erasure codes, including Reed-Solomon codes, Pyramid code, and Galloper codes. With each code, we encode a file into k blocks and guarantee that any 2 failures can be tolerated. In other words, $r = 2$ for Reed-Solomon codes and $g = 1$ for Pyramid codes and Galloper codes. We change the value of k between 4 and 12 and adjust the size of the file accordingly such that each block always contains the same amount of data after encoding with different values of k. In addition, we set $l = 2$ for Pyramid codes and Galloper codes. We run all operations repetitively for 20 times and show the means as the results.

In Fig. 7a, we show the performance of encoding with Reed-Solomon codes, Pyramid codes, and Galloper codes. The size of each block after encoding is 45 MB with different values of k. Therefore, we can see that the completion time increases with k as the total amount of data to encode also increases with k. Comparing with Reed-Solomon codes, Pyramid codes and Galloper codes both require more time to encode data, because they have one more block in the output and higher complexity than Reed-Solomon codes. However, we observe that Galloper codes maintain very similar encoding time as the original Pyramid code, for all values of k in Fig. 7a. Therefore, we believe that even though the construction of Galloper codes is based on existing Pyramid codes, we still achieve a comparable complexity.

We also compare the time of decoding operations in Fig. 7b. In this experiment, we decode the original data from k blocks with Reed-Solomon codes, Pyramid code, and Galloper codes. With Reed-Solomon codes and Pyramid codes, we remove one data block and decode the original data from $k - 1$ data blocks and one parity block. With Galloper codes, on the other hand, there is no such a block where all data are original. For a fair comparison, we still remove the same block and use the same set of blocks to decode the original data as Reed-Solomon codes and Pyramid code. Different from encoding, this time we observe that the decoding time of Galloper codes is higher than Reed-Solomon codes and Pyramid codes. This results can be expected because with Galloper codes, there are more parity data in k blocks, while with Reed-Solomon codes and Pyramid codes only one block contains original data. We can expect a lower completion time if we can visit all the rest blocks to compute the original data.

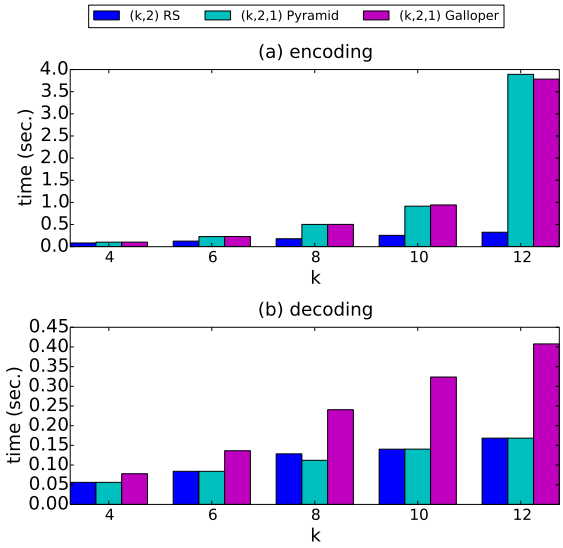

Figure 7. Comparisons of the completion time of encoding/decoding a file for various values of k with a $(k, 2)$ Reed-Solomon code, a $(k, 2, 1)$ Pyramid code, and a $(k, 2, 1)$ Galloper code.

To evaluate the performance of reconstruction, we still remove one block and measure the completion time and the amount of data read from other existing blocks during reconstruction. We encode data with a $(4, 2)$ Reed-Solomon code, a $(4, 2, 1)$ Pyramid code and a $(4, 2, 1)$ Galloper code and each block after encoding is still 45 MB. In this experiment, we remove each of the six blocks (Reed-Solomon code) or seven blocks (Pyramid code and Galloper code) and reconstruct this block. Still, the first $k = 4$ blocks with the Reed-Solomon code are data blocks. Similarly, with the Pyramid code and Galloper code, the first $k + l = 4 + 2 = 6$ blocks are data blocks or local parity blocks. Therefore, we can expect lower overhead to reconstruct the first six blocks than Reed-Solomon codes, which has been reflected in Fig. 8. In Fig. 8a, we can see that to reconstruct the first six blocks, the Pyramid code and the Galloper code require less time than the Reed-Solomon code. The reason

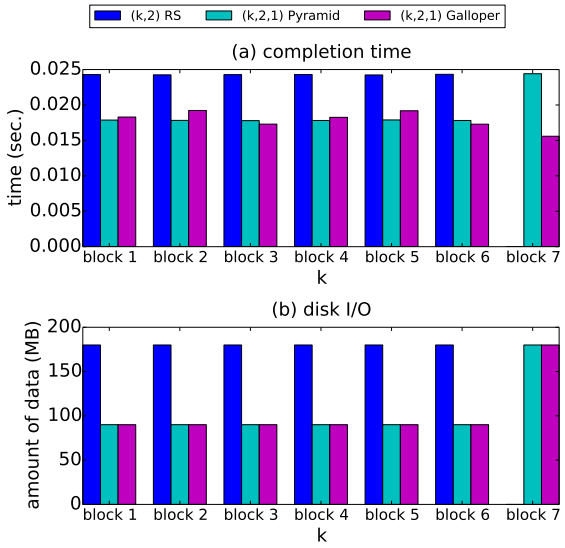

Figure 8. Comparisons of the completion time of reconstructing a block and the corresponding disk I/O for various values of k with a $(k, 2)$ Reed-Solomon code, a $(k, 2, 1)$ Pyramid code, and a $(k, 2, 1)$ Galloper code.

is shown in Fig. 8b that the Pyramid code and the Galloper code incur less disk I/O, in terms of the amount of other blocks read to reconstruct a block, on data blocks and local parity blocks than the Reed-Solomon code. Once again, we can see that the Galloper code incurs similar or even lower overhead during reconstruction than existing Pyramid codes. Since the original Pyramid codes achieve information locality only, Galloper codes can only achieve low disk I/O in the corresponding blocks as well. Therefore, we suggest placing the global parity blocks on servers with lower performance, such that less original data will be placed in such blocks. We will study how to achieve all-symbol locality in our future work.

B. Performance of Running Hadoop Jobs

To evaluate the performance of running data analytical jobs over Galloper codes, we run two representative Hadoop benchmarks, `terasort` and `wordcount`, on 30 Amazon EC2 instances of type `r3.large` with 2 CPU cores and 15 GB memory. The original data are encoded with Pyramid codes or Galloper codes with $k = 4, l = 2, g = 1$, and each of the seven coded blocks contains 450 MB. We run each benchmarks repetitively for 20 times and show the average results in Fig. 9.

We can see that with Galloper codes, the average completion time of the map tasks in the two benchmarks can both be significantly saved, by 31.5% and 40.1%, respectively. This is because with Galloper codes, $\frac{4}{7}$ of blocks are original data, leading to a saving of completion time by at most 42.9%. Although the completion time of reduce tasks is not affected so significantly, the overall completion time can still be saved by 30.4% and 36.4%, respectively.

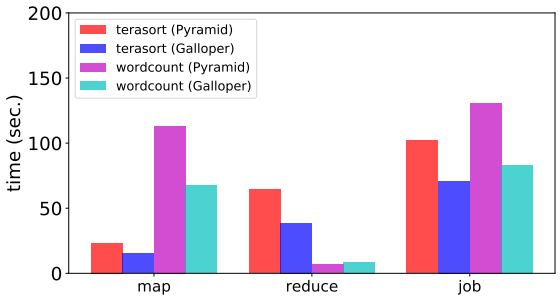

Figure 9. Comparison of Hadoop jobs running on data encoded with Pyramid codes and Galloper codes, where $k = 4, l = 2, g = 1$.

Finally, we evaluate the performance of running Hadoop jobs on heterogeneous servers. We still use the same servers, but limit the CPU usage of some servers to 40% of their original performance. We then encode data using Galloper codes with weights calculated according to the performance of servers. We show the average completion time of map tasks of running a `wordcount` job on different servers in Fig. 10. We can see that compared to the previous Galloper codes constructed for homogeneous servers, this time the completion time on the two types of servers becomes very similar, since the amount of original data are distributed according to the performance of corresponding servers. Therefore, the overall completion time can also be further saved by 32.6%.

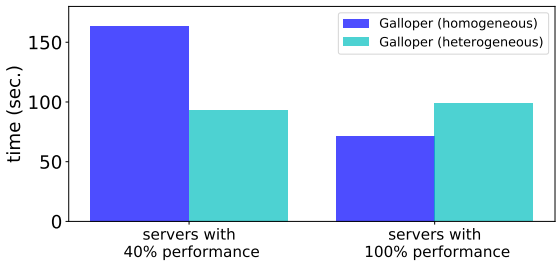

Figure 10. Comparison of Hadoop jobs running on data encoded with Galloper codes with homogeneous and heterogeneous weights, with $k = 4, l = 2, g = 1$.

VIII. Conclusion

Locally repairable codes achieve desirable properties for distributed storage systems, including high failure tolerance and low disk I/O during data reconstruction. However, the parallelism of running data analytical jobs over data encoded by locally repairable codes is limited to specific servers. In this paper, we propose *Galloper* codes, a new family of locally repairable codes that maintains the same properties of failure tolerance and disk I/O as Pyramid codes, an existing family of locally repairable codes, and even significantly extend data parallelism from specific servers to all servers. Moreover, Galloper codes can adapt to servers with heterogeneous performance to avoid stragglers.

REFERENCES

[1] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Tarazu: Optimizing Mapreduce on Heterogeneous Clusters," in *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, 2012, pp. 61–74.

[2] J. Arnold, "Erasure Codes with OpenStack Swift –Digging Deeper," https://swiftstack.com/blog/2013/07/17/erasure-codes-with-openstack-swift-digging-deeper/, July 2013.

[3] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. ul Haq, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," in *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[4] Y. Cassuto and J. Bruck, "Cyclic Lowest Density MDS Array Codes," *IEEE Transactions on Information Theory*, vol. 55, no. 4, pp. 1721–1729, 2009.

[5] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips, "Giza: Erasure Coding Objects Across Global Data Centers," in *Proc. USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, 2017, pp. 539–551.

[6] D. Cheng, J. Rao, Y. Guo, and X. Zhou, "Improving MapReduce Performance in Heterogeneous Environments With Adaptive Task Tuning," in *Proc. 15th ACM International Middleware Conference*, 2014, pp. 97–108.

[7] A. G. Dimakis, P. B. Godfrey, M. J. W. Y. Wu, and K. Ramchandran, "Network Coding for Distributed Storage System," *IEEE Trans. Inform. Theory*, vol. 56, no. 9, pp. 4539–4551, 2010.

[8] Y. Fu and J. Shu, "D-Code: An Efficient RAID-6 Code to Optimize I/O Loads and Read Performance," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2015, pp. 603–612.

[9] Y. Fu, J. Shu, and X. Luo, "A Stack-Based Single Disk Failure Recovery Scheme for Erasure Coded Storage Systems," in *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. IEEE, 2014, pp. 136–145.

[10] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the Locality of Codeword Symbols," *IEEE Transactions on Information Theory*, vol. 58, no. 11, pp. 6925–6934, 2012.

[11] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure Coding in Windows Azure Storage," in *Proc. USENIX Annual Technical Conference (USENIX ATC)*, 2012.

[12] C. Huang, M. Chen, and J. Li, "Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems," *ACM Trans. Storage*, vol. 9, no. 1, pp. 3:1–3:28, March 2013.

[13] J. Li and B. Li, "Zebra: Demand-Aware Erasure Coding for Distributed Storage Systems," in *Proc. IEEE/ACM International Symposium on Quality of Service (IWQoS)*, 2016.

[14] ——, "On Data Parallelism of Erasure Coding in Distributed Storage Systems," in *Proc. IEEE International Conference on Distributed Computing Systems (ICDCS 2017)*, 2017.

[15] S. Li, M. A. Maddah-Ali, and A. S. Avestimehr, "Coded MapReduce," in *Proc. Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, Sept. 2015, pp. 964–971.

[16] D. S. Papailiopoulos, J. Luo, A. G. Dimakis, C. Huang, and J. Li, "Simple Regenerating Codes: Network Coding for Cloud Storage," *Proc. IEEE INFOCOM*, pp. 2801–2805, 2012.

[17] D. S. Papailiopoulos and A. G. Dimakis, "Locally Repairable Codes," *IEEE Transactions on Information Theory*, vol. 60, no. 10, pp. 5843–5855, 2014.

[18] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *SIGMOD Rec.*, vol. 17, no. 3, pp. 109–116, jun 1988.

[19] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage," in *Proc. 7th USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[20] K. V. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran, "Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-Bandwidth," in *Proc. USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[21] K. V. Rashmi, N. B. Shah, and P. V. Kumar, "Optimal Exact-Regenerating Codes for Distributed Storage at the MSR and MBR Points via a Product-Matrix Construction," *IEEE Trans. Inform. Theory*, vol. 57, no. 8, pp. 5227–5239, 2011.

[22] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A Solution to the Network Challenges of Data Recovery in Erasure-Coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster," in *Proc. USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[23] I. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[24] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis," in *Proc. ACM SoCC*, 2012.

[25] R. K. Sahoo, M. S. Squillante, A. Sivasubramaniam, and Y. Zhang, "Failure Data Analysis of a Large-Scale Heterogeneous Server Environment," in *Proc. IEEE International Conference on Dependable Systems and Networks*, 2004, pp. 772–781.

[26] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing Elephants: Novel Erasure Codes for Big Data," in *Proc. VLDB Endowment*, 2013.

[27] N. B. Shah, K. V. Rashmi, P. Vijay Kumar, and K. Ramchandran, "Distributed Storage Codes With Repair-By-Transfer and Nonachievability of Interior Points on the Storage-Bandwidth Tradeoff," *IEEE Trans. on Inform. Theory,*, vol. 58, no. 3, pp. 1837–1852, 2012.

[28] K. Shum and Y. Hu, "Functional-Repair-By-Transfer Regenerating Codes," *Proc. IEEE International Symposium on Information Theory (ISIT)*, 2012.

[29] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proc. 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[30] I. Tamo and A. Barg, "A Family of Optimal Locally Recoverable Codes," *IEEE Transactions on Information Theory*, vol. 60, no. 8, pp. 4661–4676, 2014.

[31] I. Tamo, Z. Wang, and J. Bruck, "Zigzag Codes: MDS Array Codes With Optimal Rebuilding," *IEEE Transactions on Information Theory*, vol. 59, no. 3, pp. 1597–1616, 2013.

[32] W. Wang and H. Kuang, "Saving Capacity With HDFS RAID," https://code.facebook.com/posts/536638663113101/saving-capacity-with-hdfs-raid/, 2014.

[33] Y. Wang, X. Yin, and X. Wang, "MDR Codes: A New Class of RAID-6 Codes With Optimal Rebuilding and Encoding," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 5, pp. 1008–1018, May 2014.

[34] L. Xu and J. Bruck, "X-Code: MDS Array Codes With Optimal Encoding," *IEEE Transactions on Information Theory*, vol. 45, no. 1, pp. 272–276, 1999.

[35] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce Performance in Heterogeneous Environments." in *Proc. USENIX OSDI*, vol. 8, no. 4, 2008, p. 7.

[36] Z. Zhang, A. Wang, K. Zheng, U. M. G., and V. B, "Introduction to HDFS Erasure Coding in Apache Hadoop," *Cloudera Engineering Blog*, Sept. 2015, https://blog.cloudera.com/blog/2015/09/introduction-to-hdfs-erasure-coding-in-apache-hadoop/.