GF设计说明文档

1 需求分析

1.1 需求

1. 支持最大三块盘损坏 （实际实现了EC编码，通过配置支持任意冗余度）
2. 单核性能500MB/s （实际要远优于这个值，详情见测试）

1.2 分析

实现上有两种选择方案：范德蒙矩阵和柯西矩阵。原因是这两种矩阵能够保证**可逆**

1.2.1 vandermonde-Matrix

1. 编码复杂度: O(mn)
2. 解码复杂度: O(n^3)
3. 写性能受冗余度影响较大。
4. 矩阵形式

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ a_1^1 & a_2^1 & a_3^1 & \cdots & a_n^1 \\ a_1^2 & a_2^2 & a_3^2 & \cdots & a_n^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_1^{m-1} & a_2^{m-1} & a_3^{m-1} & \cdots & a_n^{m-1} \end{bmatrix}$$

5. ai互不相等且不为0
   `P = D_0 + D_1 + ... + D_n-2 + D_n-1`
   `Q = 2^n-1 * D_0 + 2^n-2 * D_1 + ... + 2^1 * D_n-2 + 2^0 * D_n-1`
   `R = 4^n-1 * D_0 + 4^n-2 * D_1 + ... + 4^1 * D_n-2 + 4^0 * D_n-1`

这里引用ZFS中关于该矩阵的描述：

We chose 1, 2, and 4 as our generators because 1 corresponds to the trival XOR operation, and 2 and 4 can be computed quickly and generate linearly-independent coefficients. (There are no additional coefficients that have this property which is why the uncorrected Plank method breaks down.)

1.2.2 cauchy-Matrix

1. 编码复杂度： O(n*m)
2. 解码复杂度： O(n^3)
3. 在伽瓦罗域中优化运算，可以将加法、乘法映射成位运算

4. 矩阵形式

$$\begin{bmatrix} \frac{1}{x_0+y_0} & \frac{1}{x_0+y_1} & \frac{1}{x_0+y_2} & \cdots & \frac{1}{x_0+y_n} \\ \frac{1}{x_1+y_0} & \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \cdots & \frac{1}{x_1+y_n} \\ \frac{1}{x_2+y_0} & \frac{1}{x_2+y_1} & \frac{1}{x_2+y_2} & \cdots & \frac{1}{x_2+y_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_m+y_0} & \frac{1}{x_m+y_1} & \frac{1}{x_m+y_2} & \cdots & \frac{1}{x_m+y_n} \end{bmatrix}$$

5. x和y都是伽瓦罗域中互不相等的元素，且集合XY交集为空
6. cauchy矩阵在伽瓦罗域中的优化方案:
   `1) 每个字节映射成w*w的位矩阵，将加法、乘法运算映射成位运算`
   `2) 每个size大小的数据块只需分解成size/w个packet，packet size满足机器字长整数倍即可`

2 设计实现

具体接口和参数说明，见源码，此处对逻辑以及设计原理进行说明！搭配源码看效果更好！！！

2.1 伽瓦罗域简介

伽瓦罗域（GF）也称有限域，由有限的元素和元素的操作组成，在域上的操作具有封闭性，比如实数域，但其上的元素有无限个。

2.1.1 本原多项式

[N次本原多项式]
[生成元]
对于GF(2^8)来说，2是其生成元，取8次本原多项式`p(x) = x^8+x^4+x^3+x^2+1`,则所有GF(2^8)元素皆可由该生成元和本原多项式确定。

2.1.2 伽瓦罗域运算

1. 加法：抑或操作 `c = a ^ b`
2. 乘法：多项式乘法，根据本原多项式，令`p(x) = 0`,可得 `x^8 = x^4+x^3+x^2+1`，`g(x) = g(x) mod p(x)`,如 `x^9 = x^8 * x = (x^4+x^3+x^2+1)*x`
3. 加法逆元是其本身
4. 乘法逆元：给定a和b，`a = g^n, b = g^m,` 若a b互为逆元，则`a*b = g^(n+m) = g^GFOrder = 1`（`GFOrder = 2^8 - 1 = 255`,因为0没有逆元，255为一周期）

2.1.3 运算优化

由于GF(2^8)共256个元素，通过定义域上的运算表格，可以用查表来减少运算的开销。
我们需要定义两个表：

```
指数表GF8Pow2[256]：下标为生成元次数，值为对应的数值
对数表GF8Log2[256]： GF8Log2[GF8Pow2[i]] = i
```

所以以上乘法运算和求逆（除法）运算可通过查表实现。

2.2 构造柯西矩阵(Gf_GenCauchyMatrixTable(...), GenCodeMatrix(...))

2.2.1 定义

对于GF(2^w)的伽瓦罗域，构造`m × n`的柯西矩阵M，要求`n + m ≤ 2w`。令`X = {x1 ,...,xm }`，`Y = {y1 ,...,yn }`。`xi`和`yj`两两各不相等，且`X ∩ Y = ∅`，那么`Mij = 1/(xi +yj )`。

2.2.2 编码原则

假定系统硬盘同时失效的可能性: `1块 > 2块 > 3块 ... >> n块`，事实上这个假定也是符合常识的，我们有理由设计raid等级的时候，优先注重低等级的运算开销。同时这么做有**两个支持前提**:

```
1 在设计编解码逻辑的时候，编码和解码的核心逻辑是相同的。
2 基于柯西矩阵EC，能够实现有效的解码逻辑，即高等级的raid，可以根据实际失效硬盘数，
进行低等级的恢复。如raid6在失效一块盘的时候，可以只选择一个冗余恢复失效的这块盘，而不需要两块冗余同时进行计算。
```

2.2.3 最优柯西矩阵(FindBestBitMat(...))

基于编码设计原则，我们有理由找到一个最稀疏矩阵M。衡量方法是将GF(2^8)中所有元素扩展成位矩阵（见下文），求对应最稀疏位矩阵的元素的逆，将其作为最优元素`yi`。然后遍历剩下元素x与Y的和式`xi + yj`，统计其位矩阵中最稀疏元素，将其逆作为`xi`，生成向量X，则可以构造出最优柯西矩阵。**这是我们提高编码速度的一个优化**

2.3 矩阵求逆(GenInvertMatrix(...))

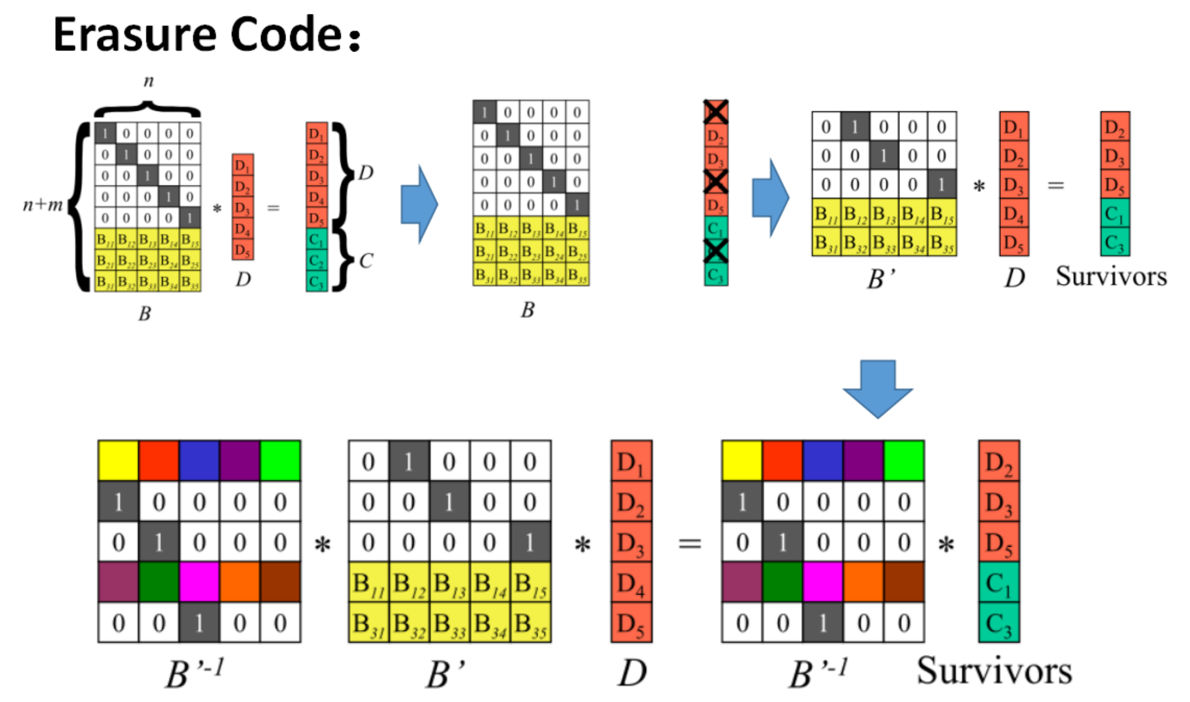
2.3.1 求逆算法

高斯消元法适用于各种域矩阵求逆

2.3.2 算法优化（原创）

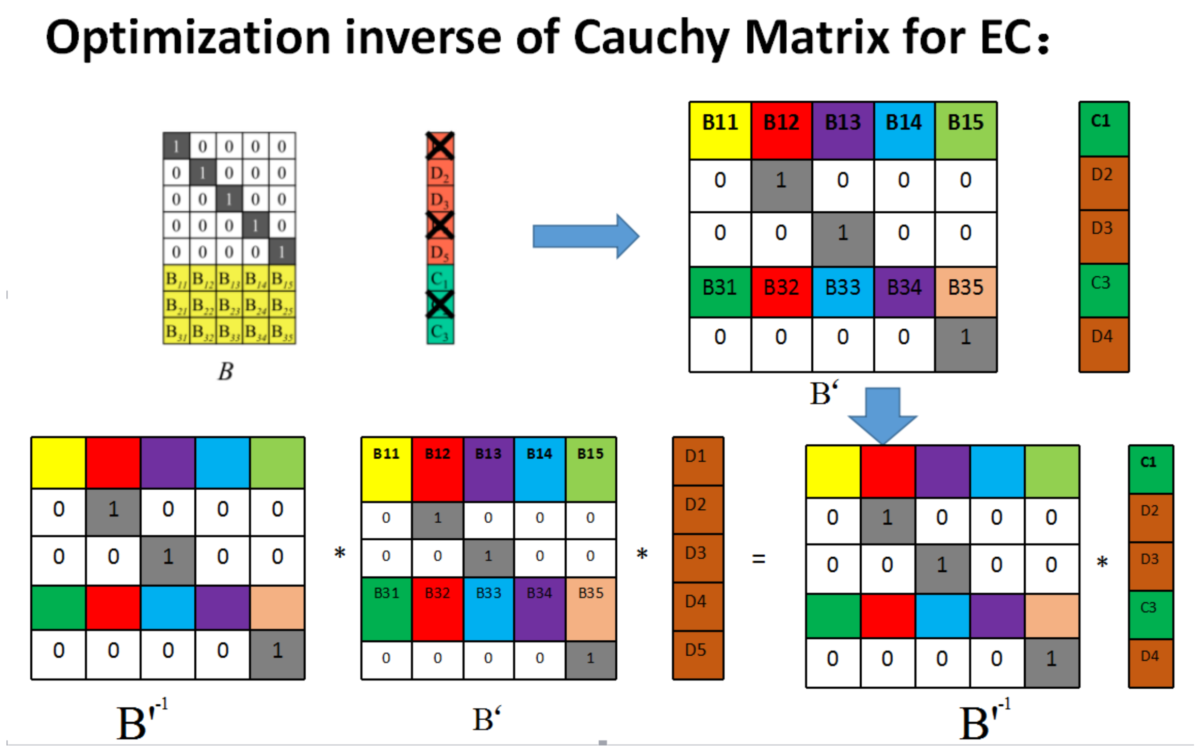
在有限域中，因为加法乘法的运算特性以及EC码的编码原理，发现可以对求逆进行优化，将复杂度由`O(n^3)`降为`O(en^2),(n为编码数据节点数，e为实际损坏节点数)`

非优化矩阵



对于矩阵`B'`求逆，需对整个矩阵进行高斯消元，代价为`O(n^3)`

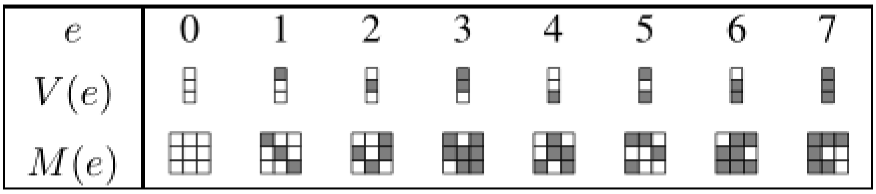
优化矩阵



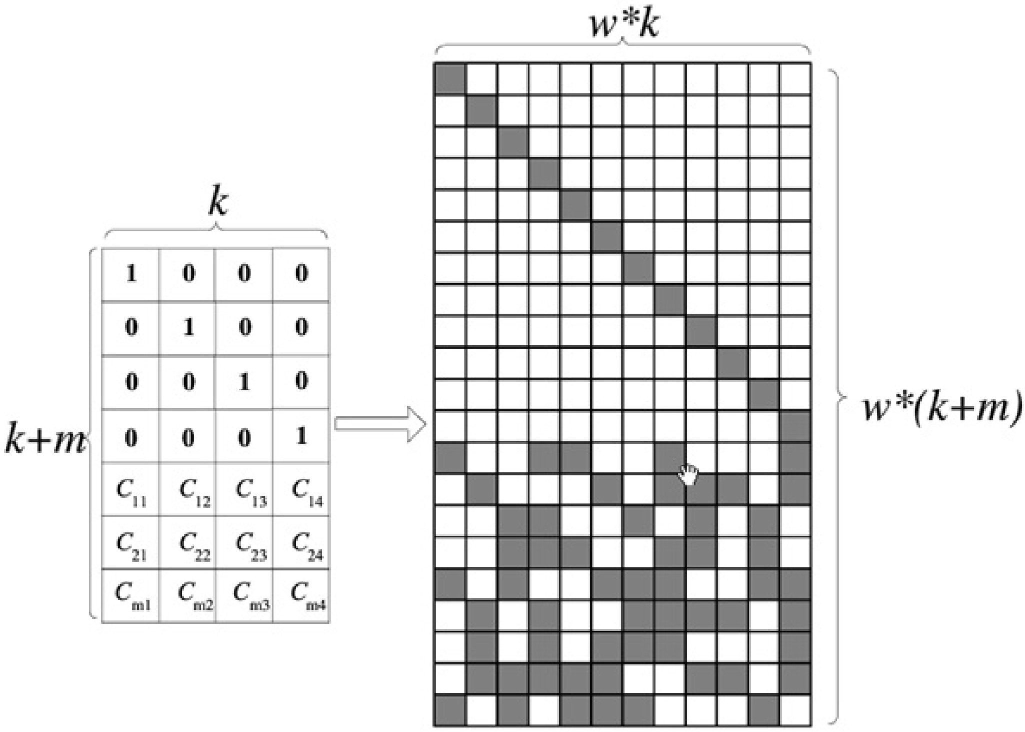
经有序组织的编码矩阵`B'`，由于有限域的运算特点，只需对相应erasure行进行列主元化解，需要一个`e * e`的矩阵`tmat`存储相对应erasure位置的值，即图中的`{{B11,B14}, {B31,B34}}`。以及逆矩阵`invmat = {{1, B12,B13,0,B15}, {0, B32,B33,1,B35}}`,求逆过程根据tmat状态，对其进行高斯消元，并对invmat进行同样操作。这样就将整个B'矩阵的求逆操作化简为invmat的求逆操作，最终得到的invmat即为所求。**这也是我们提高解码速度的一个优化，虽然对于小规模n(编码数据长度)不是很明显**

2.4 矩阵位扩展(ToBitMatrix(...))

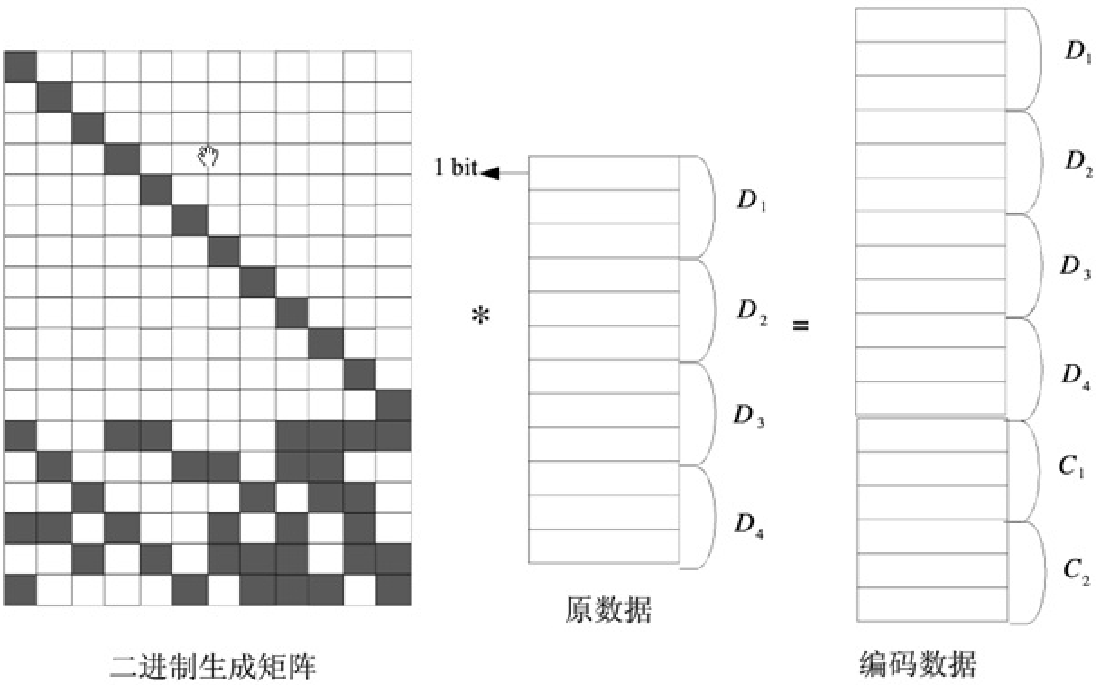
`GF(2^w)`中的元素可以被表示由位组成的`1*w`列向量`V(e)`，该列向量值等于该元素的二进制表示（高位在下）。也可以表示成`w*w`的位矩阵`M(e)`，第i列向量等于`V(e*2^(i-1)), (i>=1)`。比如`GF(2^3)`中：



那么编码矩阵就可以扩展成完整的位矩阵



那么原先的大数据块就可以分解成w个小块，计算有效块的和即可。如：



图示中展示的数据块D1是由3个bits组成的数据，实际应用中，我们可以根据计算机提供的最大运算类型进行组成，比如avx2指令，支持256bits数据的运算，那么我们的每一个数据块Di可以由`w*32`个bytes大小的数据组成。**这是我们又一个提高编码速度的关键。**

2.5 稀疏位矩阵转逻辑表示（BitMatToLMatrix(...)）

域中的元素扩展为位矩阵后，矩阵由于经过我们的挑选，基本是稀疏矩阵，如果每次运算都要遍历所有元素，无疑浪费了很多不必要的时间。我对位矩阵做了一个优化，按序存储有效位的下标值。比如`v = {0,1,1,0,0,1}`，我们需要的是`v' = {3,1,2,5,0,0，0}`,第一个元素记录有效位的个数，接下来记录实际有效位的下标值，比原矩阵多一个存储单元。这样我们很容易用较小的时间代价就得到计算的值`c = d1 + d2 + d5('+'表示亦或)`，但是空间上多了一点点开销，这个是值得的。
当然在实现上存在两种实现逻辑，一种是行优先，另一种是列优先，在实现测试之后，发现列优先的方案能够有更好的cache hit率,这样编码的实现逻辑会稍复杂，但依然采用了后者，。**这是我们另一个重要的优化。**

2.6 编解码

对于EC来说，编码和解码逻辑是一致的，都是对已知的数据节点进行编码，生成所需的编码节点。
编码：用户数据 --> 校验数据
解码：有效的用户数据 + 校验数据 --> 丢失的用户数据
约定：数据buff即`*codep[]`按照`k user nodes + m coding nodes`方式组织，前k个是输入数据，需完成填充；后m个是输出数据buff。编解码都需按照这个逻辑进行数据组织。

2.6.1 核心编码逻辑((LMatCode(...))

由于我们的编码矩阵是列优先逻辑矩阵，故编码逻辑需按照该矩阵进行设计，这么做是为了有更好的cache hit率。我们在进行计算的时候，用了CPU avx2指令集，这样可以提高数据的并发计算。我们传入一个int型指针数组`*codep[widthStp]`，每个数组成员`codep[i]`（int指针）是一个数据buff，相当于一个disk上的数据块，前k个作为用户的输入数据，后m个作为输出数据。在对每一列数据进行计算时，需保存列中每个成员的计算的过程值`__m256i sum[row]`，因为我们要的结果是将每一行的有效数据的亦或值。最后恢复`codep`成员的值。

2.6.2 编码（Gf_Encode(...), Gf_GenCodingMatrix(...)）

编码逻辑`Gf_Encode（）`和核心编码逻辑`LMatCode()`一致，通过`Gf_GenCodingMatrix()`接口生成编码矩阵`lmat`。对`LMatCode()`进行调用而已。

2.6.3 解码(Gf_Decode(...), Gf_GenDecodingMatrix(...))

解码需要根据erasure的信息生成不同的`eMap`映射数组,数组大小等于冗余度，且对应下标即为校验节点在数据buff中的下标值，比如3冗余，条带宽度为12，eMap[0]代表codep[9]校验节点,往后顺延，codep[9]校验节点实际的落盘位置由上层逻辑决定（目前dm中的实现是生成eraCodeMap，根据在map进行数据的下发和重读rebuild），但是在rebuild重读条带时，需将校验节点数据读回codep[9]这个buff中，上文中的**约定**。数组值有三种状态`{-1，-2，i}`, i是损坏的数据节点在条带中的位置（由0开始）。-1代表该校验节点损坏，-2代表该校验节点未使用。比如`eMap [0，-1，-2]`，代表codep[0]数据损坏由codep[9]校验节点进行rebuild，codep[10]校验节点损坏，codep[11]校验节点未使用。-1的情况在发生rebuild失败进行尝试其他校验节点时使用，正常情况下无法知道校验节点是否损坏。

3 测试

3.1 测试内容

测试主要分三个部分：

```
有限域操作测试，包括加法、乘法、乘法逆元
柯西矩阵相关操作，包括位矩阵扩展、求逆矩阵的验证
各raid等级编解码正确性测试，以及编码带宽
```

3.2 测试方法

3.2.1 有限域操作测试

遍历`GF(2^8)`中所有元素，进行加法、乘法、和求逆运算

3.2.2 柯西矩阵相关操作测试

根据生成的柯西矩阵表，按照不同的raid等级

```
1 生成编码矩阵和相应的位矩阵
2 生成每个raid等级下的所有erasure错误组合
3 根据erasure类型，求解相应的逆矩阵和位逆矩阵
4 对每个raid等级每个erasure类型的矩阵和逆矩阵进行乘法运算
5 验证所得的矩阵是否为单位矩阵
```

3.2.3 编解码测试

编解码测试主要测试编解码功能和带宽，根据线程数和测试次数要求，每次测试重新分配内存、填充数据、编码、解码，然后取结果的平均值，来提高数据的真实度。

```
1 输入测试参数，‘-c’为必选项，其他raid、erasure信息、线程数、测试次数等，无输入则用默认值
2 根据线程数，创建同样workload的工作线程
3 每个工作线程根据输入的测试次数，运行多次编解码操作
4 填充数据块，根据编码矩阵进行编码
5 指定erasure信息的话，执行该erasure类型，否则执行测试系统生成的所有erasure类型
6 将相应erasure的节点数据擦除，进行解码
7 验证恢复的结果是否和填充的数据一致性
8 每次编解码返回时间，作为带宽统计时间
9 生成工作load时也加了时间统计，在所有线程完成时统计结束，模拟IO路径上的编解码时间。
存在数据验证时不太准确，增加了关闭验证的参数’-s‘。该结果只提供粗略的模拟，应在实际系统中在做具体测试
10 所有线程所有测试验证通过方为成功
```

3.3 测试结果

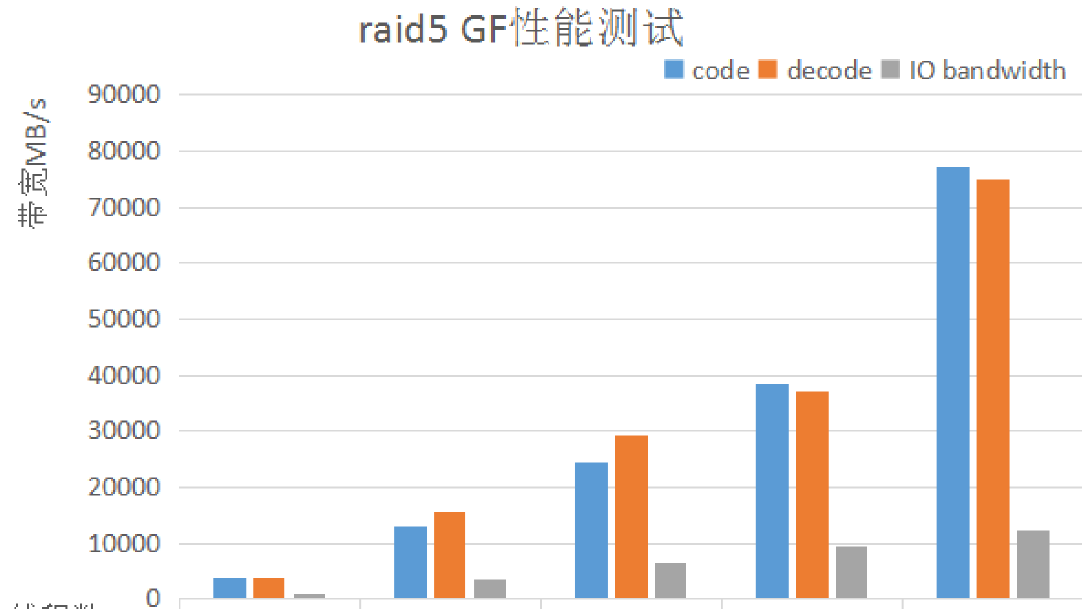
3.3.1 有限域运算&柯西矩阵操作测试结果

```
[root@afa_primary ~]# ./gf_test -g
Add in GF(2^8) test: pass!
Invert in GF(2^8) test: pass!
Multiply in GF(2^8) test: pass!
[root@afa_primary ~]# ./gf_test -M
Matrix operation test of (1,11,8) pass!
Matrix operation test of (2,10,8) pass!
Matrix operation test of (3,9,8) pass!
```

3.3.2 编解码测试结果

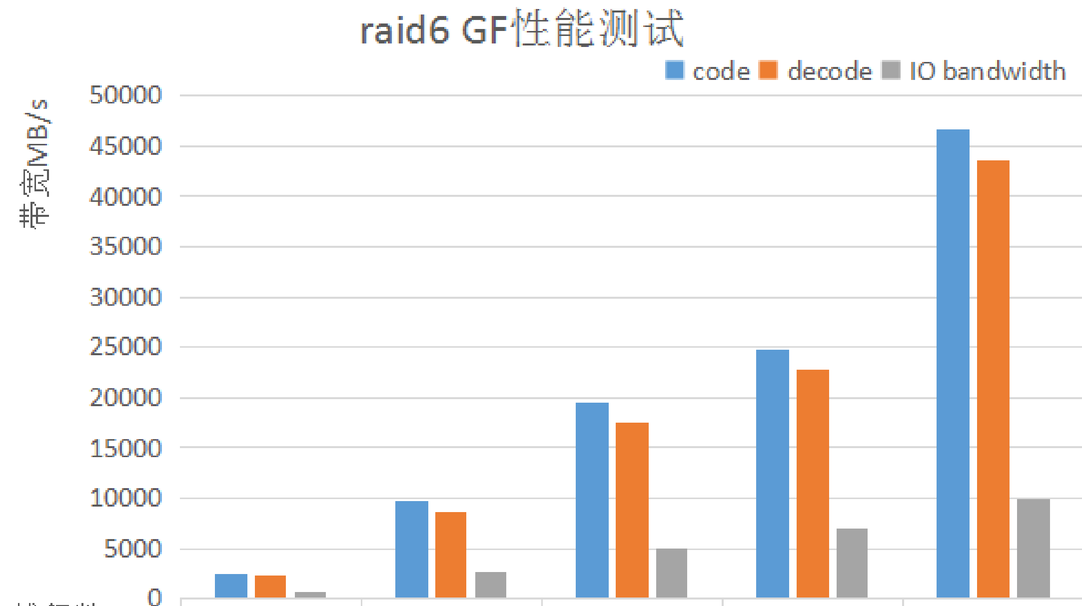
性能的计算取多次综合的平均值，多线程计算简单以pthread_join返回的结果作为总时间，总数据量为所有线程的IO总和。其中IO-bandwidth包含了数据填充、数据编码、数据解码和校验的时间，相对值不是很准确只是作为粗略的估计IO路径上的时间消耗。decode取得是所有erasure case的平均值，若在正常情况下，比如3冗余实际只有一块数据失效，那么性能接近提升1倍，两块数据失效也能提升50%。

1 raid5



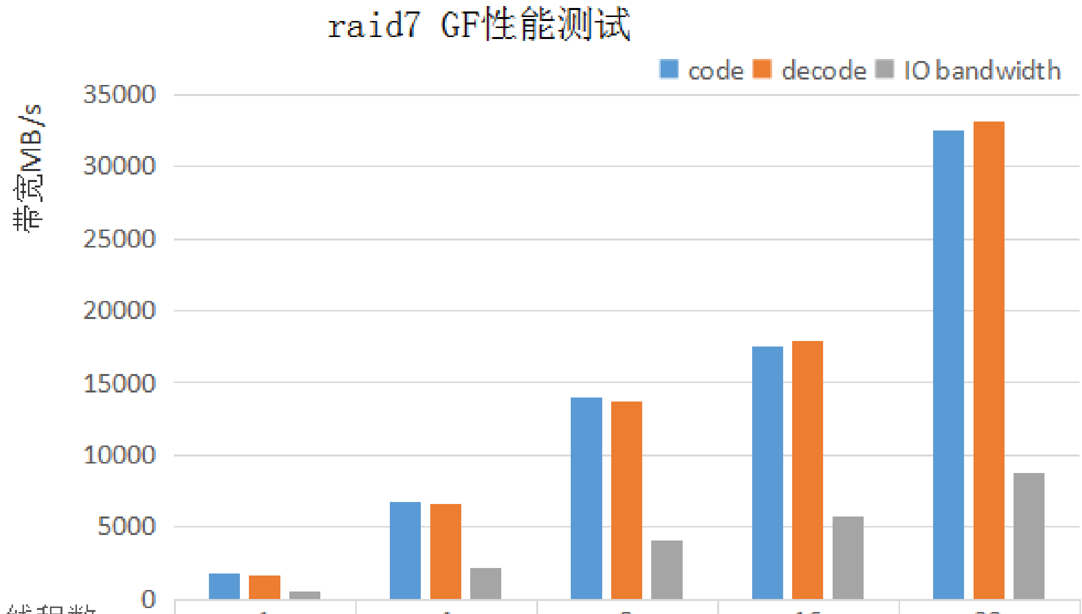
线程数	1	4	8	16	32
code	3956.25	12969.38	24404.43	38498.92	77309.8
decode	3748.57	15501.35	29158.46	37017.46	75018.64
IO bandwidth	993.34	3385.56	6331.45	9229.36	12293.83

2 raid6



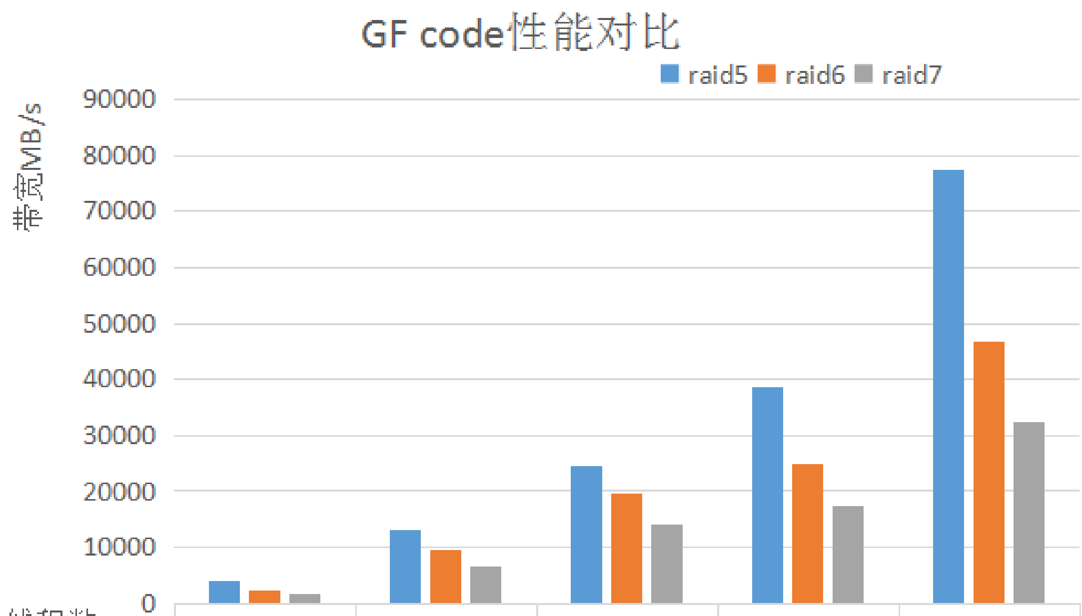
线程数	1	4	8	16	32
code	2503.03	9648.87	19521.02	24816.65	46727.29
decode	2261.34	8729.38	17590.44	22838.19	43618.79
IO bandwidth	730.78	2615.44	5006.73	6948.37	9996.49

3 raid7(3冗余)



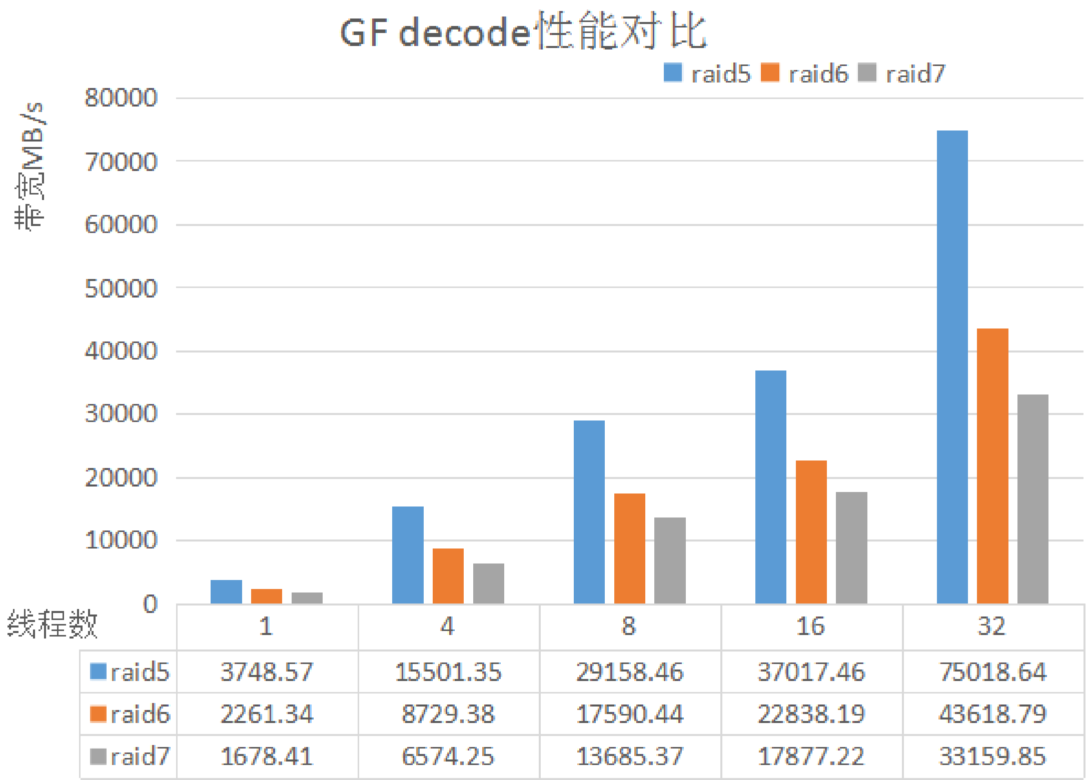
线程数	1	4	8	16	32
code	1730.91	6763.45	14015.48	17511.29	32530.07
decode	1678.41	6574.25	13685.37	17877.22	33159.85
IO bandwidth	572.05	2104.28	4027.65	5733.59	8784.2

4 coding性能对比



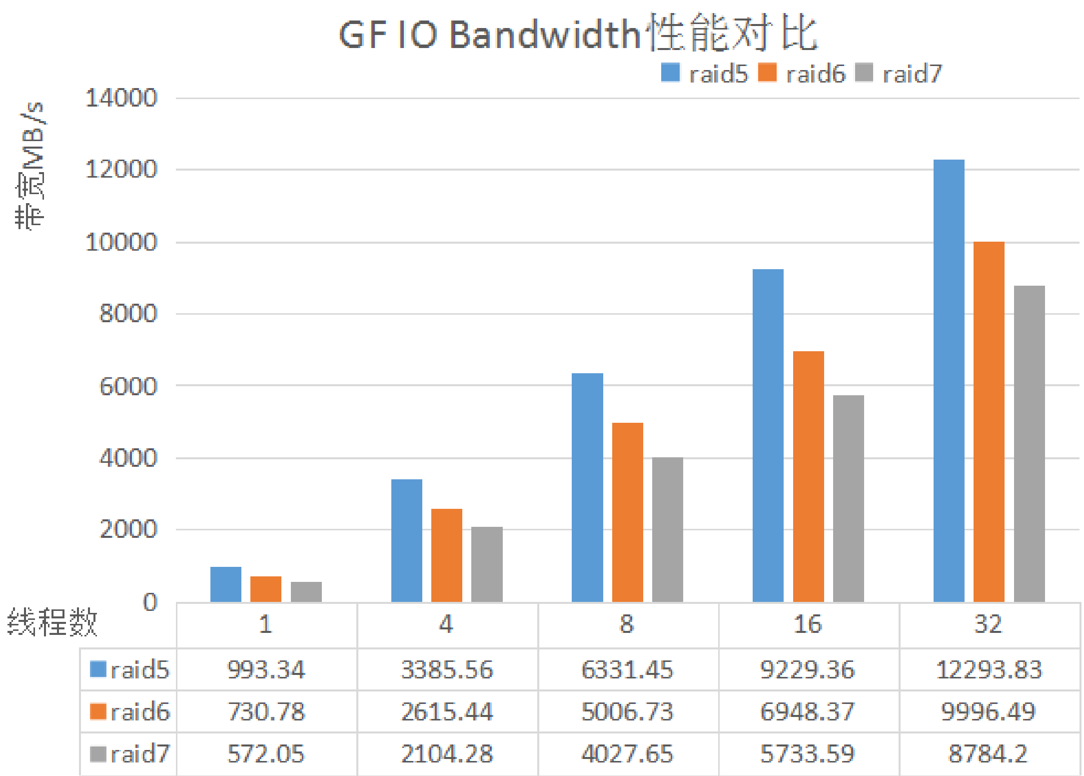
线程数	1	4	8	16	32
raid5	3956.25	12969.38	24404.43	38498.92	77309.8
raid6	2503.03	9648.87	19521.02	24816.65	46727.29
raid7	1730.91	6763.45	14015.48	17511.29	32530.07

5 decoding性能对比



	1	4	8	16	32
raid5	3748.57	15501.35	29158.46	37017.46	75018.64
raid6	2261.34	8729.38	17590.44	22838.19	43618.79
raid7	1678.41	6574.25	13685.37	17877.22	33159.85

4 IO bandwidth性能对比



	1	4	8	16	32
raid5	993.34	3385.56	6331.45	9229.36	12293.83
raid6	730.78	2615.44	5006.73	6948.37	9996.49
raid7	572.05	2104.28	4027.65	5733.59	8784.2