Reparare de Corupție în Fișiere
Coduri Reed-Solomon

Theodor Negrescu

2025-01-16

Cuprins

Introducere

- **Codurile Reed-Solomon** sunt o clasă bine cunoscută de coduri de corectare a erorilor, inventate în 1960 de Irving S. Reed și Gustave Solomon. Sunt utilizate într-o gamă largă de aplicații - CD-uri, coduri QR, RAID arrays, comunicare satelit, etc.
- Acest program, numit **RSARC**, pentru **R**eed-**S**olomon **Arc**hive, repară corupția fișierelor folosind date redundante generate în avans folosind coduri Reed-Solomon.

Justificare

- Corectarea erorilor este o problemă importantă în sisteme de backup, stocare și transfer de date. Deși drive-uri moderne au mecanisme interne de corectare a erorilor, este posibil să se piardă date.
- Un mod comun de utilizare a codării Reed-Solomon pentru stocarea datelor este în RAID arrays. Se presupune că orice drive poate eșua total, iar datele trebuie să fie recuperabile. În plus, tot procesul trebuie să funcționeze live.
- Acest program urmărește un model diferit. Se presupune că modificare live nu este necesară, iar datele sunt stocate pe un singur drive (deși pot fi manual copiate pe mai multe). Se cere repararea unui fișier dacă drive-ul devine parțial corupt.

Principii Matematice - Polinoame

- Datele sunt re-interpretate ca fiind valori y al unui polinom, unde valorile au 64 de biți. Un set de puncte (x, y), cu toate x distince, determină un polinom unic de grad cel mult $n - 1$. Pentru a genera date redundante, polinomul este calculat prin interpolare și evaluat pentru a obține k puncte redundante.
- Orice combinație de n puncte din cele $n + k$ (date + redundante) va determina același polinom unic. Pentru a recupera datele, se interpolează din nou polinomul folosind orice n puncte non-corupte. Este important să se țină cont de coordonatele x când reconstruim polinomul. Această codare este de tip **erasure code**, deoarece trebuie să se știe care date sunt corupte.

Principii Matematice - Câmpuri Finite / Galois

- Există o problemă în calculul polinoamelor. Nu se poate folosi aritmetica naturală a procesorului, deoarece nu exista invers pentru înmulțire (i.e. înmulțirea nu este inversabilă / pierde informație). Exemplu: $0 * 2 = 0$, dar și $2^{63} * 2 = 0$ când lucrăm cu 64 de biți.
- Interpolarea și evaluarea polinomului trebuie să fie într-un **câmp**, structură matematică care necesită o operație de înmulțire inversabilă. Din fericire, există câmpuri finite notate $GF(p^n)$, unde p este prim (vom folosi 2).
  În campul finit $GF(2^{64})$, valorile sunt polinome de grad cel mult 63, unde coeficienții sunt 0 sau 1 (elemente ale $GF(2)$).
  Nu este același polinom ca în secțiunea anterioară. Acela are coeficienții valori din $GF(2^{64})$.
- Deoarece adunarea se face element cu element, este echivalentă cu XOR. Înmulțirea este mai complicată și consistă în înmulțirea polinoamelor și reducerea modulo un polinom ireductibil arbitrar.

Implementare portabilă a înmulțirii

Pseudocod pentru înmulțirea a două polinoame *lhs* și *rhs* de 64 de biți: Toate variabilele sunt de 64 de biți. Acest algoritm nu necesită un procesor cu operații de 128 de biți, deoarece reducerea se face incremental. Variabila IRREDUCIBLE_POLY nu conține coeficientul de grad 64. Acela este implicit 1.

```
in: lhs, rhs, IRREDUCIBLE_POLY
let product = 0
repeat 64 times:
    if rhs & 1:
        product ^= lhs
    rhs >>= 1
    const carry = lhs >> 63
    lhs <<= 1
    if carry:
        lhs ^= IRREDUCIBLE_POLY
return product
```

Implementare folosind carry-less multiplication

Operația de înmulțire fără carry (CLMUL) valabilă pe procesoarele moderne înmulțește două polinoame de 64 de biți și returnează un polinom de 128 de biți. Înmulțirea părții mai puțin semnificative de 64 de biți a rezultatului cu polinomul ireductibil face o reducere parțială.
Folosind polinomul $x^{64} + x^4 + x^3 + x + 1$, două înmulțiri sunt garantate să elimine biții peste 64.

```
in: lhs, rhs, IRREDUCIBLE_POLY
const full_product = CLMUL(lhs, rhs)
const almost_reduced = CLMUL(full_product >> 64, IRREDUCIBLE_POLY)
const fully_reduced = CLMUL(almost_reduced >> 64, IRREDUCIBLE_POLY)
return lower_64_bits(fully_reduced) ^
       lower_64_bits(almost_reduced) ^
       lower_64_bits(full_product)
```

Această metodă este mai rapidă, dar cea portabilă e necesară ca fallback pentru procesoare mai vechi.

Implementare invers multiplicativ

Algoritmul extins Euclidean găsește inversul multiplicativ.

```
in: x, IRREDUCIBLE_POLY
assert x != 0
if x == 1 return 1

let t = 0
let new_t = 1
let r = IRREDUCIBLE_POLY
let new_r = x
// x^64 nu e in IRREDUCIBLE_POLY (ar fi un al 65-lea bit)
// deci prima iterație a împărțirii e caz special
r ^= new_r << (new_r.leading_zeros() + 1)
let quotient = 1 << (new_r.leading_zeros() + 1)
while new_r != 0:
    while new_r.leading_zeros() >= r.leading_zeros():
        // impartire simpla de polinoame
        const degree_diff = new_r.leading_zeros() - r.leading_zeros()
        r ^= new_r << degree_diff
        quotient |= 1 << degree_diff
    (r, new_r) := (new_r, r)
    (t, new_t) := (new_t, t ^ gf64_multiply(quotient, new_t))
    quotient := 0
return t
```

Interpolare Newton

- Interpolarea Newton este o metodă de a găsi un polinom care trece prin un set de puncte. Am folosit această metodă în loc de interpolarea Lagrange, deoarece nu necesită împărțiri de polinoame.
- Se definesc polinoame de bază:
  $N_0 = 1$, $N_1 = x - x_0$, $N_2 = (x - x_0)(x - x_1), \ldots$.
- $N(x) = [y_0]N_0 + [y_0, y_1]N_1 + [y_0, y_1, y_2]N_2 + \ldots$.
- Notația $[y_0, y_1, y_2]$ reprezintă *divided differences* și se calculează recursiv:
  $[y_k] = y_k$
  $[y_k, \ldots, y_{k+j}] = \frac{[y_{k+1}, \ldots, y_{k+j}] - [y_k, \ldots, y_{k+j-1}]}{x_{k+j} - x_k}$.
  $x_{k+j} - x_k$ este o constantă, deci necesită doar operații de invers și înmulțirea polinomului cu o constantă.
- Algorithmul necesită timp $O(n^2)$. La orice moment dat, doar ultimul polinom de baza, și ultimul rând din tabelul de *divided differences* sunt necesare. Deci memoria necesară este doar $O(n)$.

Necesitatea blocurilor

Din multiple motive, fișierul input nu este considerat un singur cod Reed-Solomon.

- Algoritmul $O(n^2)$.
- Nevoile de memorie. Polinomul are aceeași dimensiune ca datele, deci pentru un fișier de 20 GB, avem nevoie de 20 GB de memorie pentru a genera datele redundante, și la fel pentru a repara fișierul.
- Orice reparare, chiar și pentru un singur byte, necesită citirea a 20 GB de date.
- Chiar dacă fișierul ar fi un singur cod, avem nevoie de hash-uri pentru a verifica integritatea datelor. Nu putem face hash a fiecărui 64 de biți! Deci oricum am avea efectiv blocuri de date, cu dimensiune determinată de câți bytes intră într-un hash.

Împărțirea în blocuri

- Fișierul este împărțit în blocuri de dimensiune fixă. Dacă fișierul nu e divizibil la dimensiunea blocului, se completează implicit cu 0-uri.
- Un bloc **nu** este un cod Reed-Solomon.
- Un cod este format din toate simbolurile pe o poziție dată din toate blocurile.
  Exemplu: dacă avem 3 blocuri, 8 simboluri pe bloc, 2 blocuri redundante, avem 8 coduri. Codul 7 va conține simbolurile 7, 15, 23 în input, și 7, 15 în output. Polinomul va fi interpolat folosind cele trei simboluri input, și evaluat pentru a obține output-ul.
  Un procedeu similar e folosit pentru reparare, doar ca citește și scrie simultan din ambele fișiere, în loc de a citi exclusiv din input și scrie exclusiv în output.

Metadate

Pe lângă blocurile de redundanță, fișierul de output conține și metadate.

- Header - un string hardcodat, hash pentru toate metadatele, numar de bytes într-un block, numărul de blocuri de date și, respectiv, de paritate, și lungimea așteptată a fișierului de input.
  Momentan, orice corupere de metadate este o eroare fatală.
- Hash-uri pentru toate blocurile de input și output. Aceastea sunt folosite pentru a detecta blocuri corupte, ceea ce e necesar pentru procesul de **erasure decoding**.
  Hash-urile conțin și primul simbol din fiecare bloc. Momentan nu e folosit, dar în viitor ar putea fi folosit pentru a găsi blocuri care nu sunt în poziția așteptată.

Arhitectura internă de codare și reparare

Arhitectura este multi-threaded. Se folosesc în principal canale pentru a comunica între thread-uri.

- Se citesc multiple coduri simultan, pe baza memoriei disponibile. Altfel, ar rezulta într-un apel de sistem pentru fiecare simbol citit.
- Pentru procesarea codurilor, se creează un thread pentru fiecare core.
  Deoarece buffer-ul primit de la cititor conține multiple coduri, multiple thread-uri îl folosesc simultan. Am folosit primitive read-write lock și reference counting pentru a coordona accesul la buffer și returnarea sa la cititor.
- Scrierea folosește mmap, fără nicio optimizare suplimentară. Scriitorul primește buffer plus indexul codului, și scrie datele în mmap pe pozițiile potrivite indexului.
  Deoarece datele vin de la citire în grupuri, nu rezulta în page-fault-uri excesive. În testele mele, nu am întâmpinat bottleneck-uri aici.

Cod împărtășit între codare și reparare

- În mod convenabil, comenzile de reparare și codare sunt destul de similare. Diferența primară este că repararea trebuie să poată citi din două fișiere simultan, de pe poziții arbitrare, calculate pe baza verificării hash-urilor, și, similar, să scrie în două fișiere simultan.
- Deoarece comenzile sunt similare, codul central care creează și controlează thread-urile poate fi folosit în ambele.
- Similar, codul pentru comanda de verificare e folosit și în reparare. Operația de reparare este o verificare urmată de o codare care folosește blocurile non-corupte găsite de verificare, și scrie în cele corupte.

Alegeri de tehnologie

Am folosit limbajul **Rust** pentru acest proiect. Mi s-au părut utile feature-urile de type system, package management, multi-threading, și unit testing. Am folosit următoarele biblioteci:

- **blake3** pentru hash-uri.
- **crossbeam-channel** pentru canale multi-producer, multi-consumer.
- **inficatif** pentru afișarea progresului în terminal.
- **memmap2** pentru mmap portabil.
- **num_cpus** pentru a afla numărul de core-uri.
- **sysinfo** pentru a afla memoria disponibilă.
- **positioned-io** pentru a citi din poziții arbitrare într-un fișier în mod convenabil, portabil, și eficient.
- **fastrand** pentru RNG, folosit doar pentru teste.

Testare

- Pentru testarea codului care nu face I/O, am folosit unit testing. Asta include codul de matematică $GF(2^{64})$, polinoame, și parsat header-ul.
- Pentru testarea codului care face I/O, am folosit un test end-to-end. Acesta generează un fișier de test, rulează codul de codare, verifică să nu fie corupt, introduce erori, verifică să fie detectate, rulează codul de reparare, și verifică să fie reparate corect fișierele.

Rezultate Test End-to-End

```
Current RNG seed: 8598835888182296900
696 data blocks
Using 20 CPUs, 1 buffers, reading 1748352 bytes per reading pass (314 symbols per code)
ETA 00:00:00                 218544/218544     one read pass [86,350,310.1663/s]
ETA 00:00:00                    314/314        read          [724.4234/s]
ETA 00:00:00                    314/314        process       [725.6852/s]
ETA 00:00:00                    314/314        write         [726.9966/s]
ETA 00:00:00                    696/696        hash data     [289,927.5181/s]
ETA 00:00:00                    256/256        hash parity   [223,093.6819/s]
Flush output...
Time: 439.2046ms
No corruption detected
Data file corrupted
Parity file corrupted
740 good blocks, 696 required for repair, 212 bad blocks
Using 20 CPUs, 1 buffers, reading 1858880 bytes per reading pass (314 symbols per code)
ETA 00:00:00                 232360/232360     one read pass [45,567,038.6131/s]
ETA 00:00:00                    314/314        read          [608.7438/s]
ETA 00:00:00                    314/314        process       [609.2221/s]
ETA 00:00:00                    314/314        write         [609.7641/s]
No corruption detected
```

Rezultate Test Manual

```
PS $ cargo run --release encode .\test.txt parity.rsarc 1024 100
Finished `release` profile [optimized] target(s) in 1.91s
    Running `target\release\rsarc.exe encode .\test.txt parity.rsarc 1024 100`
1987 data blocks
Using 20 CPUs, 1 buffers, reading 2034688 bytes per reading pass (128 symbols per code)
ETA 00:00:00                254336/254336     one read pass [38,593,062.426/s]
ETA 00:00:00                   128/128        read          [83.4917/s]
ETA 00:00:00                   128/128        process       [83.5056/s]
ETA 00:00:00                   128/128        write         [83.5272/s]
ETA 00:00:00                  1987/1987       hash data     [405,444.0091/s]
ETA 00:00:00                   100/100        hash parity   [111,969.5443/s]
Flush output...
PS $ get-filehash .\test.txt
SHA256          2061DEEA850BEF5B3519FABBFDC0130C754D365CCE6A7DB77767F665468F8EE7
PS $ get-filehash .\parity.rsarc
SHA256          27CF447340575B44CA5E3D83A85AEBC7400D63EC0112C0DEC72CAA3CEA55313B
PS $ # messing with files in text and hex editor
PS $ cargo run --release repair .\test.txt .\parity.rsarc
    Finished `release` profile [optimized] target(s) in 0.08s
     Running `target\release\rsarc.exe repair .\test.txt .\parity.rsarc`
2085 good blocks, 1987 required for repair, 2 bad blocks
Using 20 CPUs, 1 buffers, reading 2135040 bytes per reading pass (128 symbols per code)
ETA 00:00:00                266880/266880     one read pass [44,686,301.7598/s]
ETA 00:00:00                   128/128        read          [74.1221/s]
ETA 00:00:00                   128/128        process       [74.1324/s]
ETA 00:00:00                   128/128        write         [74.1481/s]
PS $ get-filehash .\test.txt
SHA256          2061DEEA850BEF5B3519FABBFDC0130C754D365CCE6A7DB77767F665468F8EE7
PS $ get-filehash .\parity.rsarc
SHA256          27CF447340575B44CA5E3D83A85AEBC7400D63EC0112C0DEC72CAA3CEA55313B
```

Concluzii

Programul pare să funcționeze corect. Următoarele sunt posibile îmbunătățiri:

- Algoritmul $O(n^2)$ este un obstacol deoarece împiedică blocuri mici, deci reduce redundanța. Cercetare pe internet sugerează că există metode $O(n \log n)$, implementarea lor ar putea fi un viitor pas în semestrul II.
- Nu exista redundanță la metadate pentru moment. Nu ar trebui să fie prea complicat să inserez niște meta-blocuri de redundanță pentru metadate, pentru a nu suferi erori fatale de la singuri bytes corupti în header.
- Recuperarea blocurilor dacă sunt în locații necorespunzătoare ar fi un feature obscur, dar necesar pentru completitudine. Nu este acceptabil că a șterge sau adăuga un singur byte cauzează erori fatale. Deja am datele necesar în metadata de hash-uri, tot ce ar fi nevoie e o funcție de reasamblare care mută blocurile în locația corectă și apoi apelează repararea.
- Doar un singur fișier de input și unul de output sunt suportate. Ar fi util să se poată crea un singur fișier de output pentru structuri de foldere arbitrare. Header-ul ar trebui să fie adaptat pentru a include locația fișierelor.
- Arhive combinate ar fi un feature util, pentru a nu se putea separa accidental blocurile de date de cele de paritate.

- Despre CLMUL de la Intel:
  `https://cdrdv2-public.intel.com/836172/clmul-wp-rev-2-02-2014-04-20.pdf`
- Librării ce implementează Reed-Solomon in $O(n \log n)$:
  `https://github.com/paritytech/reed-solomon-novelpoly`
  `https://github.com/malaire/reed-solomon-16`
  `https://github.com/catid/leopard`
  `https://github.com/Bulat-Ziganshin/FastECC`