Теорія оптимальних рішень

УДК 004-931

В.В. ГОРИН, В.М. ЛЮТЕНКО

ИМПЛЕМЕНТАЦИЯ И ОПТИМИЗАЦИЯ АЛГОРИТМА РИДА-СОЛОМОНА ДЛЯ СОЗДАНИЯ КОДОВ ВОССТАНОВЛЕНИЯ ПОТЕРЬ В ДАННЫХ

Рассмотрены основные аспекты построения, реализации и оптимизации алгоритма Рида-Соломона для создания кодов восстановления потерь в данных. Подробно рассмотрены возможности ускорения работы алгоритма, вопрос об эффективности использования 32 и 64-битной арифметики. Показано, что для классической версии алгоритма, использование длинных слов (32 и 64 бита), является неэффективным, не смотря на тот факт, что объем обрабатываемых процессором данных за один такт прямо пропорционален величине битности арифметики.

Техника распределенного хранения данных в нескольких хранилищах для достижения большей скорости чтения и записи и использования одной или нескольких техник для обнаружения и исправления ошибок не нова. Она была представлена Петтерсоном (David A. Patterson), Гибсоном (Garth A. Gibson) и Катцом (Randy H. Katz) в 1987 году [1]. И известна под аббревиатурой RAID «redundant array of inexpensive disks» («избыточный (резервный) массив недорогих дисков», так как они были гораздо дешевле RAM). Но основная проблема RAID – подобных систем в том, что если данные храняться на *n* хранилищах, то вероятность того, что некоторые из хранилищ выйдут из строя довольно высока. Как раз для хранения и восстановления данных в RAID-подобных системах хорошо подходят коды восстановления потерь, речь о которых пойдёт в этой статье.

Обнаружение и исправление ошибок. Обнаружение ошибок в технике связи –- действие, направленное на контроль целостности данных при записи/воспроизведении информации или при её передаче по линиям связи. Исправление ошибок (коррекция ошибок) – процедура восстановления информации после чтения её из устройства хранения или канала связи. Для обнаружения ошибок используют коды обнаружения ошибок, для исправления - корректирующие коды (коды, исправляющие ошибки, коды с коррекцией ошибок, помехоустойчивые коды).

В системах связи возможны несколько стратегий борьбы с ошибками:

- обнаружение ошибок в блоках данных и *автоматический запрос повторной передачи* повреждённых блоков – этот подход применяется в основном на канальном и транспортном уровнях
- обнаружение ошибок в блоках данных и отбрасывание повреждённых блоков – такой подход иногда применяется в системах потокового мультимедиа, где важна задержка передачи и нет времени на повторную передачу.
- *исправление ошибок* (*forward error correction*) применяется на физическом уровне.

Корректирующие коды - коды, служащие для обнаружения или исправления ошибок, возникающих при передаче информации под влиянием помех, а также при её хранении.

Для этого при записи (передаче) в полезные данные добавляют специальным образом структурированную *избыточную* информацию (контрольное число), а при чтении (приёме) её используют для того, чтобы обнаружить или исправить ошибки. Естественно, что число ошибок, которое можно исправить, ограничено и зависит от конкретного применяемого кода.

С ***кодами, исправляющими ошибки***, тесно связаны *коды обнаружения ошибок*. В отличие от первых, последние могут только установить факт наличия ошибки в переданных данных, но не исправить её.

В действительности, используемые коды обнаружения ошибок принадлежат к тем же классам кодов, что и коды, исправляющие ошибки. Фактически, любой код, исправляющий ошибки, может быть также использован для обнаружения ошибок (при этом он будет способен обнаружить большее число ошибок, чем был способен исправить).

По способу работы с данными коды, исправляющие ошибки, делятся на *блоковые*, делящие информацию на фрагменты постоянной длины и обрабатывающие каждый из них в отдельности, и *свёрточные*, работающие с данными как с непрерывным потоком.

Блоковые коды. Пусть кодируемая информация делится на фрагменты длиной k бит, которые преобразуются в кодовые слова длиной n бит. Тогда соответствующий блоковый код обычно обозначают (n,k). При этом число $R=k/n$ называется скоростью кода. Если исходные k бит код оставляет неизменными, и добавляет $n-k$ проверочных, такой код называется систематическим, иначе несистематическим.Задать блоковый код можно по-разному, в том числе таблицей, где каждой совокупности из k информационных бит сопоставляется n бит кодового слова. Однако, хороший код должен удовлетворять, как минимум, следующим критериям:

– способность исправлять как можно большее число ошибок,

– как можно меньшая избыточность,

– простота кодирования и декодирования.

Нетрудно видеть, что приведённые требования противоречат друг другу. Именно поэтому существует большое количество кодов, каждый из которых пригоден для своего круга задач. Практически все используемые коды являются линейными. Это связано с тем, что нелинейные коды значительно сложнее исследовать, и для них трудно обеспечить приемлемую лёгкость кодирования и декодирования[2].

Свёрточные коды, в отличие от блоковых, не делят информацию на фрагменты и работают с ней как со сплошным потоком данных. Свёрточные коды, как правило, порождаются дискретной линейной инвариантной во времени системой. Поэтому, в отличие от большинства блоковых кодов, свёрточное кодирование – очень простая операция, чего нельзя сказать о декодировании. Кодирование свёрточным кодом производится с помощью регистра сдвига, отводы от которого суммируются по модулю два. Таких сумм может быть две (чаще всего) или больше. Декодирование свёрточных кодов, как правило, производится по алгоритму Витерби, который пытается восстановить переданную последовательность согласно критерию максимального правдоподобия[3].

Прямая коррекция ошибок (FEC): техника кодирования/декодирования, позволяющая исправлять ошибки методом упреждения. Применяется для исправления сбоев и ошибок при передаче данных, путём передачи избыточной служебной информации, на основе которой может быть восстановлена первоначальное содержание посылки. На практике широко используется в компьютерных ЛВС, LAN и различных телекоммуникационных сетях. Коды, обеспечивающие прямую коррекцию ошибок, требуют введения большей избыточности в передаваемые данные, чем коды, которые только обнаруживают ошибки.

Коды Рида - Соломона: недвоичные циклические коды, позволяющие исправлять ошибки в блоках данных. Элементами кодового вектора являются не биты, а группы битов (блоки). Очень распространены коды Рида – Соломона, работающие с байтами (октетами). В настоящее время широко используется в системах восстановления данных с компакт-дисков, при создании архивов с информацией для восстановления в случае повреждений, в помехоустойчивом кодировании.

Код восстановления от потерь (Erasure code): это разновидность кода прямой корекции ошибок (FEC), который трансформирует сообщения длиной k символов в более длинное сообщение (кодовое слово) длинной n символов, такое что изначальное сообщение может быть восстановлено по подмножеству n символов. Отношение $r = k/n$ называется коэффициент кода.

Постановка задачи. Пусть имеется n устройств хранения информации, D_1, D_2, …, D_n, каждое из которых содержит по k байт. Назовем эти устройства *«Устройствами Данных»*. Пусть также имеется m дополнительных устройств C_1, C_2, …, C_m, каждое из которых также содержит по k байт. Эти устройства назовем «*Кодирующими Устройствами»*. Содержимое каждого кодирующего

устройства будет вычисленно из содержимого устройств данных. Наша цель – определить кодирующую функцию для каждого устройства C_i такую, что если при отказе любых m из $D_1, D_2, \ldots, D_n, C_1, C_2, \ldots, C_m$ устройств, содержимое отказавших устройств может быть реконструировано из содержимого оставшихся неповрежденных устройств.

Классический алгоритм восстановления от потерь – алгоритм Рида-Соломона. Модель отказа устройства, о которой мы говорим, формально говоря, называется *стиранием* (англ. *erasure*). Когда устройство становится неисправным – оно отключается, и система распознает это отключение. Эта ситуация противоположна *ошибке*, когда неисправное устройство продолжает функционировать, но сохраняет и выдает некоректные данные, которые могут быть обнаружены при помощи одного из алгоритмов обнаружения ошибок, речь о которых шла выше [2,5].

Вычисление содержимого каждого кодирующего устройства C_i требует определения функции F_i, и ее применения ко всем устройствам данных:

$$C_i = F_i(D_1, D_2, \ldots, D_n),\ i = 1, \ldots, m.$$

Обычно, метод кодирования Рида-Соломона разбивает каждое устройство хранения информации на *слова*. Размер каждого слова равен w бит. Каждое устройство хранения содержит $l = 8k/w$ слов. Кодирующие функции F_i оперируют со словами.

Для простоты, будем считать, что каждое устройство содержит только одно слово. Тогда наша задача сводится к вычислению m кодирующих слов $c_1, \ldots, c_n$ из n слов данных $d_1, \ldots, d_n$ таких, что потеря любых m слов допустима и может быть восстановлена.

Вычисление кодирующих слов (контрольных сумм). Определим каждую функцию F_i как линейную комбинацию слов данных::

$$c_i = F_i(d_1, d_2, \ldots, d_n) = \sum_{j=1}^{n} d_j f_{i,j}$$

Другими словами, если мы представим слова данных и кодирующие слова как векторы D и C, а функции F_i как строки матрицы F, то состояние системы будет удовлетворять следующему уравнению::

$$FD = C.$$

Мы определяем матрицу F как матрицу Вандермонда размерности $m \times n$: $f_{i,j} = j^{i-1}$, и, таким образом, вышестоящее уровнение принимает следующий вид:

$$\begin{bmatrix} f_{1,1} & f_{1,2} & \cdots & f_{1,n} \\ f_{2,1} & f_{2,2} & \cdots & f_{2,n} \\ \vdots & \vdots & & \vdots \\ f_{m,1} & f_{m,2} & \cdots & f_{m,n} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \cdots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

Восстановление устройств данных при отказе произвольных *m* устройств. Для объяснения процесса восстановления системы, мы определяем матрицу A и вектор E как: $A = \begin{bmatrix} I \\ F \end{bmatrix}$, и $E = \begin{bmatrix} D \\ C \end{bmatrix}$. Тогда мы получаем следующее уравнение ($AD = E$):

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 3 & \cdots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \cdots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}.$$

Мы можем сопоставить каждому устройству в системе соответствующую строку матрицы A и вектор E. Если устройство отказывает, мы удаляем соответствующую строку из матрицы A и соответствующий элемент из E. Это дает нам новую матрицу A' и новый вектор E', которые удовлетворяют следующему уравнению:

$$A'D = E'.$$

Предположим, что ровно m устройств отказывают. Тогда, A' – это матрица размерности $n \times n$. Мы определили F как матрицу Вандермонда и, поэтому, каждое подмножество из n строк матрицы A является невырожденной матрицей. Т.е, матрица A' не сингулярна и значения D могут быть посчитаны из выражения $A'D = E'$, например, методом Гаусса. Таким образом, все устройства данных могут быть восстановлены.

Предположим теперь, что отказывают k устройств и $k < m$. Тогда мы можем просто выбросить $m - k$ «лишних» устройств и эта ситуация сведется к предыдущей.

Арифметика в конечных полях Галуа. Основная проблема с описанным выше алгоритмом заключается в том, что область и пределы всех вычислений – это бинарные слова фиксированной длины w. Не смотря на то, что вся приведенная алгебра справедлива для «длинной арифметики», т.е., для арифметики над вещественными числами, мы должны убедиться, что она работает для слов фиксированной длины. Типичной ошибкой при работе с описанными кодами является проведение всей арифметики над целыми числами по модулю 2^w. Это *не работает*, т.к. деление не определено для всех пар элементов (например, отношени 3/2 не определено по модулю 4); метод Гауса не применим во множестве таких случаев. Вместо этого мы должны выполнять сложение и умножение на поле с большим количеством элементов, чем $n + m$ [2].

Поля с количеством элементов 2^w называются *полями Галуа (Galois Fields,* обозначаются как $GF(2^w)$), они являются фундаментальным разделом в алгебре (например, [5, 6, 7]).

Поле $GF(n)$ это множество из n элементов, замкнутое относительно операций сложения и умножения, а котором для каждого элемента x существует противоположный элемент $-x$ и обратный элемент $1/x$ (за исключением нуля, для которого нет обратного элемента). Если n целое число, то поле $GF(n)$ можно представить как множество $\{0, 1, \ldots, n-1\}$, где и сложение и умножение, оба, выполняются по модулю n.

Однако, если n не простое число, множество $\{0, 1, \ldots, n-1\}$, где сложение и умножение выполняются по модулю n, *не* является *полем*. Поэтому мы *не можем* выполнять наше кодирование на бинарных словах размера $w>1$, использую сложение и умножение по модулю 2^w. Вместо этого мы используем конечные поля Галуа.

Для того чтобы объяснить, как работают поля Галуа, рассмотрим полиномы от x, коэффициенты которых лежат в $GF(2)$. Это означает, например, что если $r(x)=x+1$, и $s(x)=x$, то $r(x)+s(x)=x+1+x=(1+1)x+1=0x+1=1$. Более того, мы можем считать остаток от деления таких полиномов друг на друга, использую следующее определение: если $r(x) \bmod q(x) = s(x)$, то $s(x)$ это полином степени меньшей, чем $q(x)$, и $r(x) = q(x)t(x) + s(x)$, где $t(x)$ – это любой полином от x.

Пусть $q(x)$ – *примитивный* многочлен степени w, коэффициенты которого лежат в $GF(2)$. Это означает, что $q(x)$ не раскладывается на множители (не факторизуется), и полином x является *генератором* $GF(2^w)$. Чтобы увидеть как x генерирует $GF(2^w)$, мы можем начать строить множество с элементов 0, 1, и x, и потом продолжить перебор элементов путем умножения последнего элемента на x и взятия остатка от деления на $q(x)$, если степень резльтата умножения больше либо равна w. В построенном множестве будет 2^w элементов – последний элемент, умноженный на x по модулю $q(x)$ равен 1. Сложение в таком поле выполняется путем сложения полиномов, произведение – путем умножения полиномов по модулю $q(x)$.

Теперь чтоб использовать $GF(2^w)$ для алгоритма Рида-Соломона нам нужно отобразить элементы поля $GF(2^w)$ в слова длины w. Пусть $r(x)$ – полином в $GF(2^w)$. Тогда мы можем отобразить $r(x)$ в бинарное слово b длины w, устанавливая i-ый бит b в значение коэффициента при x^i в $r(x)$.

Сложение бинарных элементов в $GF(2^w)$ может быть выполнено как побитовое сложение по модулю 2 (XOR). Умножение немного сложнее. Нужно преобразовать бинарные числа в их полиномиальное представление, перемножить полиномы по модулю $q(x)$, и потом преобразовать результат обратно к бинарному виду. Это можно сделать путем использования двух логарифмических таблиц: первая преобразует бинарный элемент b к его степени j такой, что x^j эквивалентно b (“дискретное логарифмирование”), и вторая преобразует степень j к

бинарному элементу b. Каждая таблица состоит из $2^w - 1$ элементов. Тогда произведение двух бинарных слов состоит из конвертирования каждого из них к дискретному логарифму, сложения логарифмов по модулю $2^w - 1$ (что эквивалентно перемножению полиномов по модулю $q(x)$) и конвертирования результата обратно к бинарному элементу. Деление производится таким же образом, за исключением того, что логарифмы вычитаются, а не складываются. Очевидно, что элементы, для которых $b = 0$ должны рассматриваться отдельно. Таким образом, умножение и деление двух бинарных элементов занимает три операции поиска в таблице и одно сложение по модулю.

Имплементация и оптимизация. Итак, имея n устройств данных и m кодирующих устройств, алгоритм Рида-Соломона для создания отказоустойчивости системы по отношению к отказу не более чем m устройств выглядит следующим образом.

1. Выбираем длину слова w такую, что $2^w > n + m$. Легче всего использовать $w = 8$ либо $w = 16$, т.к. слова тогда ложатся прямо на границы байта (2^8). Для $w = 16$, $n + m$ может быть не больше 65 535.
2. Строим две логарифмические таблицы для умножения и деления слов в $GF(2^w)$, как это было показано выше.
3. Строим матрицу F – матрицу Вандермонда размерности $m \times n$: $f_{i,j} = j^{i-1} \quad (1 \le i \le m, 1 \le j \le n)$, где умножение производится в $GF(2^w)$.
4. Используя матрицу F, вычисляем каждое слово контрольных устройств из слов устройств с данными. Опять, сложение и умножение выполняются в поле $GF(2^w)$.
5. При неисправности не более чем m устройств: выбираем любые n из уцелевших устройств и строим матрицу A' и вектор E', как это было описано выше; затем находим D из $A'D = E'$. Это позволяет восстановить содержимое устройств данных, после чего можно пересчитать недостающие кодирующие устройства при помощи матрицы F.

Авторами были созданы имплементации классического алгоритма для $w = 8$ на C, Java, C# и JavaScript. Исходный код Java имплементации доступен по адресу http://users.i.com.ua/~vgrn/rs_transform/. Основной особенностью разработанной имплементации является отказ от логарифмических таблиц для умножения и деления. Вместо них используются две другие таблицы – таблица умножения и таблица деления. О них речь пойдет ниже.

Java имплементация алгоритма состоит из трех основных модулей: а) модуль арифметики на 8-ми битном конечном поле Галуа; б) модуль матричных вычислений; в) модуль кодека Рида-Соломона.

Модуль арифметики на конечных полях Галуа позволяет выполнять операции сложения, вычитания, умножения и деления в $GF(2^8)$. Для ускорения работы модуля создаются две таблицы размера 256×256 – таблица умножения и таб-

лица деления. Первая таблица состоит элементов вида $a_{i,j} = i \cdot j$, вторая $b_{i,j} = i \cdot j^{-1}$, $0 \le i \le 255, 0 \le j \le 255$. Для нахождения результата умножения или деления двух чисел i и j в $GF(2^8)$ достаточно найти элемент, находящийся на пересечении i-ой строки и j-го столбца соответствующей таблицы.

Модуль матричных вычислений отвечает за матричные операции: построение матрицы Вандермонда, вычисление обратной матрицы, перемножение матрицы на вектор, скалярное умножение векторов. Он использует модуль арифметики на конечном поле Галуа для выполнения операций сложения, вычитания, умножения и деления. Поиск обратной матрицы производится методом Гаусса [8].

Модуль кодека отвечает за подготовку к кодированию данных и непосредственно кодирование/декодирование (восстановление). Он использует предыдущие два модуля для построения кодирующей и декодирующей матриц, для вычисления содержимого кодирующих устройств, для восстановления потерь. Работа модуля, таким образом, состоит из четырех фаз (стадий):

1. Инициализация кодирования. На этой стадии строится кодирующая матрица необходимой размерности. Размерность зависит от инициализационных параметров – количества устройств данных и кодирующих устройств.
2. Кодирование. Эта стадия состоит из большого количества (примерно L/n, где L – объем кодируемых данных в байтах) перемножений кодирующей матрицы на вектор. Это составляет порядка $L \cdot m/n$ скалярных произведений, или $L \cdot m$ операций умножения и побитового сложения по модулю 2. В разработаной реализации алгоритма перемножение есть ни что иное, как чтение элемента в таблице умножения. Чтение/запись элемента таблицы можно реализовать двумя путями: как а) a[i][j], т.е. чтение сначала p = a[i] и потом чтение/запись p[j], либо как б) a[i*256+j], т.е. вычисление индекса p = i*256+j и потом чтение/запись a[p]. Первый способ состоит из двух операций чтения/записи памяти, второй – из одной такой операции, но это дается ценой вычисления индекса нужной ячейки памяти. В разработанной авторами имплементации используется второй способ, т.к. было замечено, что он работает быстрее. На прцессоре AMD Athlon 64 прирост производительности составил около 50%.
3. Инициализация декодирования. На этой стадии строится декодирующая матрица как обратная к кодирующей, после того, как из нее выброшено m строк. При больших n и m эта стадия может быть достаточно ресурсоемкой.
4. Декодирование. Эта стадия похожа на стадию кодирования, за тем лишь исключением, что вместо матрицы кодирования используется матрица декодирования, а вместо кодирующих устройств – оставшиеся довступные устройства.

Интересной особенностью классического алгоритма Рида-Соломона является то, что увеличение длины слова w не приводит к увеличению скорости рабо-

ты алгоритма, а наоборот. Действительно, рассмотрим, например, переход от $w = 8$ к $w = 16$. 64-битный процессор выполняет операции над 8ми и 16и битными словами одинаково быстро, поэтому на первый взгляд может показаться, что переход к $w = 16$ увеличит скорость работы алгоритма в двое. Вспомним, однако, что для $w = 8$ мы можем построить и закешировать таблицы умножения и деления (их суммарный объем составляет $2 \cdot 1 \cdot 2^8 \cdot 1 \cdot 2^8 = 128$ Кб, что помещается в кеш памяти современных процессоров), для $w = 16$ такие таблицы заняли бы $2 \cdot 2 \cdot 2^{16} \cdot 2 \cdot 2^{16} = 2^{35} = 32$ Гб, что даже сейчас является труднодоступным объемом оперативной памяти. Использование логарифмических таблиц для $w = 16$ добавляет дополнительне побитовое сложение по модулю 2 (XOR), что несколько снижает суммарную скорость [9]. Для $w = 32$ даже использование логарифмические таблиц становится недоступным, т.к. их объем был бы равен $2 \cdot 2^{32} = 2^{33} = 8$ Гб. Поэтому, для $w = 32$ и $w = 64$ пришлось бы пользоваться самым общим алгоритмом умножения и деления, что понижает суммарную скорость существенно [9, 10].

Выводы. Построенная авторами имплементация алгоритма Рида-Соломона на матрицах Вандермонда является простейшей, и в то же время, имеет довольно хорошие скоростные характеристики – это наиболее быстрая имплементация на матрицах Вандермонда. В процессе разработки были также обнаружены некоторые эффективные оптимизации алгоритма, например замена операции чтения/записи двумерного массива a[i][j] операцией чтения/записи одномерного массива a[i*n+j] (с вычислением индекса элемента вместо его считывания из памяти). Дальнейшее ускорение работы алгоритма возможно, если замнить классические матрицы Вандермонда на битовые матрицы Коши. Также, авторами планируется создание имплементации алгоритма для вычислений на видеокартах на платформе nVidia CUDA. Об этом пойдет речь в дальнейших публикациях.

В.В. Горін, В.М. Лютенко

ІМПЛІМЕНТАЦІЯ ТА ОПТИМІЗАЦІЯ АЛГОРИТМУ РІДА-СОЛОМОНА ДЛЯ СТВОРЕННЯ КОДІВ ВІДНОВЛЕННЯ ВТРАТ В ДАНИХ

Розглянуто основні аспекти побудови, реалізації та оптимізації алгоритму Ріда-Соломона для створення кодів відновлення втрат в даних. Детально розглянуто можливості прискорення роботи алгоритму, питання щодо ефективності використання 32 та 64-бітної арифметики. Показано, що для класичної версії алгоритму, використання довгих слів (32 і 64 біти), є неефективним, не дивлячись на той факт, що об'єм оброблюємих процесором даних за один такт прямо пропорційний величині бітності арифметики.

V.V. Gorin, V.M. Lyutenko

IMPLEMENTATION AND OPTIMIZATION OF REED-SOLOMON ALGORITHM FOR ERASURE CODING PURPUSE

Various aspects for building, implementing and optimization of classic Reed-Solomon erasure coding algorithm discussed. Detailed view into possibilities for algorithm speed improvement, question about 32 and 64-bit arithmetic usage efficiency is given. It is shown, that for classical algorithm version the use of long words (32 and 64 bits) is not efficient despite the fact, that size of data processed per one processor operation is proportional to the arithmetic word size.

1. *Patterson D., Gibson G., Katz R.,* A Case for Redundant Arrays of Inexpensive Disks (RAID) // SIGMOD '88 Proceedings – 1987. – **17** (3). – P. 109 – 116.
2. *Морелос-Сарагоса Р.,* Искусство помехоустойчивого кодирования. Методы, алгоритмы, применение. – М.: Техносфера, 2006. – 320 с.
3. *Никитин, Г.,* Сверточные коды: Учебное пособие. – СПбГУАП. СПб., 2001. – 78 с.
4. *Peterson W., Weldon E.,* Error-Correcting Codes, Second Edition. – Cambridge, Massachusetts: The MIT Press, 1972 – 373 p.
5. *MacWilliams F., Sloane N.,* The Theory of Error-Correcting Codes, Part I. (North-Holland Publishing Company). – NY: Oxford, 1977 – 782 p.
6. *Lin J.,* Introduction to Coding Theory. – NY: Springer-Verlag, 1982 – 227 p.
7. *Herstein I.,* Topics in Algebra, Second Edition – Lexington, Massachusetts, Xerox College Publishing 1975 – 400 p.
8. *Ильин В., Позняк Э.,* Линейная алгебра: Учебник для вузов. – 6-е изд., стер. – М.: ФИЗМАТЛИТ – 2004. – 280 с
9. *Schuman, C., Plank, J.,* A Performance Comparison of Open-Source Erasure Coding Libraries for Storage Applications // Technical Report UT-CS-08-625, Dept. of Electrical Engineering and Computer Science, University of Tennessee – 2008. – 13 p.
10. *Plank, J., Luo, J., Schuman, C., Xu, L., Wilcox-O'Hearn, Z.,* A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage // FAST-09: 7th USENIX Conference on File and Storage Technologies – 2009. – 14 p.

Получено 14.04.2012