2013 Вычислительные методы в дискретной математике №4(22)

DOI 10.17223/20710410/22/9

УДК 519.7

О РЕАЛИЗАЦИИ ОСНОВНЫХ ЭТАПОВ БЛОЧНОГО АЛГОРИТМА ВИДЕМАНА — КОППЕРСМИТА ДЛЯ ДВОИЧНЫХ СИСТЕМ ЛИНЕЙНЫХ УРАВНЕНИЙ НА ВЫЧИСЛИТЕЛЯХ КЛАСТЕРНОГО ТИПА

А. С. Рыжов

Лаборатория ТВП, г. Москва, Россия

E-mail: ryzhovalexander@mail.ru

Рассматриваются вопросы реализации наиболее трудоёмких этапов алгоритма Видемана — Копперсмита поиска решений сильно разреженных систем линейных уравнений на современных ЭВМ. Исследуются вопросы эффективной реализации отдельных операций алгоритма. Отдельно рассмотрены проблемы, возникающие при реализации алгоритма на вычислителях кластерного типа.

Ключевые слова: *системы линейных уравнений, алгоритм Видемана — Копперсмита.*

Введение

Необходимость нахождения решений системы однородных линейных уравнений (СОЛУ) с сильно разреженной матрицей возникает в задачах вычислительной алгебры достаточно часто. Не являются исключением и теоретико-числовые задачи, имеющие непосредственное отношение к криптографии. Например, при нахождении дискретного логарифма в поле $\mathrm{GF}(2^n)$ (при простом $p = 2^n - 1$) методом Копперсмита [1] необходимо найти решения системы линейных уравнений над простым полем $\mathbb{Z}_p$. При факторизации чисел методом решета числового поля одним из этапов является нахождение нескольких решений большой разреженной системы линейных уравнений над полем $\mathrm{GF}(2)$.

В настоящий момент для решения данной задачи обычно используется один из двух алгоритмов: Ланцоша — Монтгомери [2] и Видемана — Копперсмита [3]. Оба алгоритма имеют свои преимущества и недостатки, а также ограничения на используемые вычислители. Алгоритм Ланцоша — Монтгомери реализован, например, в пакете GGNFS. Он ориентирован на выполнение на одном вычислителе кластерного типа с высокоскоростным внутренним соединением узлов. Алгоритм Видемана — Копперсмита, напротив, позволяет наиболее трудоёмкие этапы выполнять на нескольких не связанных друг с другом кластерах. Именно этот алгоритм был выбран авторами работы [4] для реализации линейного этапа при факторизации 768-битного числа.

При реализации математических алгоритмов на современных вычислительных системах возникает ряд вопросов, связанных со способом представления данных, выполнением отдельных шагов и координацией работы узлов вычислителя. Знание и грамотное использование особенностей вычислителя, на котором предполагается выполнение алгоритма, помогают существенно снизить время его работы по сравнению с «лобовой» реализацией. В то же время в работах, содержащих результаты экспериментальных исследований математических алгоритмов, эти вопросы, как правило, не освещаются. Например, в [4] приводится время выполнения отдельных этапов факторизации 768-битного числа и характеристики используемых вычислительных кла-

стеров, однако не затрагиваются вопросы, связанные с реализацией этих этапов на указанных вычислителях.

В данной работе рассматриваются вопросы реализации наиболее трудоёмких этапов алгоритма Видемана — Копперсмита для двоичных систем линейных уравнений на современных вычислителях. В п. 1 излагается сам алгоритм Видемана — Копперсмита; п. 2 посвящён вопросам реализации ряда операций, выполняющихся на отдельных шагах алгоритма. В п. 3 рассмотрены нюансы выполнения алгоритма Видемана — Копперсмита на вычислителях кластерного типа; п. 4 посвящён вопросам применения алгоритма в рамках факторизации больших целых чисел методом решета числового поля.

Работу можно рассматривать как обзор подходов к программированию отдельных процедур алгоритма Видемана — Копперсмита и реализации его в целом на вычислителях кластерного типа. Автору не удалось найти публикаций, в которых бы детально анализировались вопросы эффективной реализации приводимых методов на современных многопроцессорных системах, и настоящая работа преследует цель восполнить этот пробел.

1. Алгоритм Видемана — Копперсмита

Блочный алгоритм Видемана — Копперсмита является обобщением алгоритма Видемана поиска решения разреженной СОЛУ [5], ориентированным на вычислительную технику, работающую со словами в несколько байтов. Он подробно описан в [3, 6]. Приведём краткое описание основных шагов алгоритма.

Пусть A — матрица размера $M \times N$ над конечным полем P и необходимо найти ненулевое решение СОЛУ

$$Ax = 0. \tag{1}$$

Будем считать, что $P = \mathrm{GF}(2)$, параметры алгоритма $m, n \in \mathbb{N}$ $(m \geqslant n)$ кратны длине машинного слова в битах: $m = 64m'$, $n = 64n'$, и что выполнено неравенство $N \geqslant M + m$.

Выберем случайную матрицу $Z \in P_{m,M}$ размера $m \times M$ над полем P. Пусть $B \in P_{M,M}$ — матрица из первых M столбцов матрицы A СОЛУ (1), $Y \in P_{M,n}$ — матрица, составленная из столбцов матрицы A с номерами $M{+}1, M{+}2, \ldots, M{+}n$. Выполняем следующие три этапа:

1) Вычислим матрицы $a_i = ZB^iY$, $i = 0, 1, \ldots, L$ (здесь L — параметр метода; см. далее).
2) Найдём такие векторы $f_0, \ldots, f_d \in P^{(n)}$, что $\sum\limits_{j=0}^{d} a_{i+j} f_j = 0 \in P^{(m)}$ для всех $i \geqslant 0$.

   Формальная сумма $\sum\limits_{j=0}^{d} f_j X^j$ называется *векторным аннулирующим многочленом последовательности* $(a_i : i = 0, \ldots)$.
3) Вычислим вектор w, составленный из векторов $\sum\limits_{j=0}^{d-1} B^j Y f_{j+1}$ и f_0 и дополненный нулями до длины N. Можно показать [3, 6], что w с большой вероятностью является решением системы (1).

Для нахождения векторного аннулирующего многочлена Копперсмитом предложен отдельный алгоритм [3, 6], основанный на последовательном вычислении приближений Паде. При этом в результате работы алгоритма может быть получено до n различных линейно независимых аннулирующих многочленов, что в результате при-

водит к получению до n линейно независимых решений системы (1). Из теоретико-вероятностных соображений [3] достаточно иметь $L = \lfloor M/m \rfloor + \lfloor M/n \rfloor + \Delta$ элементов последовательности (a_i), где $\Delta \approx 100$.

Заметим, что на этапах 1 и 3 алгоритма при вычислении матриц $W_i = B^i Y$ и $a_i = ZB^i Y$ вместо вычисления степеней матрицы B достаточно выполнять последовательное умножение матрицы Y на матрицу B: $W_{i+1} = BW_i$. При этом трудоёмкость этого умножения, равно как и вся трудоёмкость этапов 1 и 3, полностью определяется размером матрицы B и её весом, т. е. числом ненулевых элементов. Если ω — среднее число ненулевых элементов в строке матрицы B, то трудоёмкость вычисления очередной матрицы W_{i+1} равна $\mathrm{O}(M\omega)$; соответственно трудоёмкость этапов 1 и 3 равна $\mathrm{O}(M^2\omega)$.

Важно отметить, что на 64-разрядных ЭВМ умножение одного двоичного вектора на разреженную матрицу B выполняется с такой же трудоёмкостью, что и умножение на матрицу B одновременно 64 векторов. За счёт этого в блочном алгоритме Видемана — Копперсмита достигается существенное снижение трудоёмкости, так как сокращается число шагов на 1-м и 3-м этапах по сравнению с исходным алгоритмом Видемана (частный случай при $m = n = 1$). Например, при $m = n = 64$ число шагов L уменьшится в 64 раза! Если $n = 64n'$, то вычисления матриц W_i можно выполнять параллельно на n' несвязанных вычислителях, умножая по 64 столбца матрицы W_i на матрицу B независимо на каждом вычислителе.

Замечание 1. Не стоит забывать о том, что вместо алгоритма Берлекэмпа — Мэсси, используемого на втором этапе при $m = n = 1$, возникает необходимость применения алгоритма Копперсмита для построения векторного аннулирующего многочлена. Трудоёмкость этого алгоритма равна $\mathrm{O}(m^3L^2) = O\left(mM^2\right)$ (при $m \geqslant n$), и при сокращении числа шагов на первом и третьем этапах алгоритма растет трудоёмкость второго этапа. Однако при небольших значениях m и n трудоёмкость второго этапа в разы меньше, чем трудоёмкость этапов 1 и 3. Кроме того, в работе [7] предложена рекурсивная версия алгоритма Копперсмита, имеющая трудоёмкость порядка $\mathrm{O}\left(m^3 L \log^2 L\right)$. При факторизации 768-битного числа [4] для нахождения решений большой разреженной СОЛУ с помощью алгоритма Видемана — Копперсмита были выбраны параметры $m = 16 \cdot 64$, $n = 8 \cdot 64$. Размер матрицы системы был равен $N \approx 193 \cdot 10^6$. Время работы алгоритма на трёх мощных вычислительных кластерах составило 119 сут, причём на выполнение второго этапа с использованием лишь одного из кластеров ушло менее 18 ч.

Замечание 2. Использование параметров m и n, больших 64, позволяет выполнять этапы 1 и 3 на независимых вычислителях и снижает число шагов на этих этапах, за счёт чего снижается общее время выполнения первого и третьего этапов алгоритма. Но трудоёмкость этих этапов не снижается: при $m = n = 128$ число шагов L в 2 раза меньше, чем при $m = n = 64$, но эти шаги необходимо выполнить для каждой из двух частей матрицы Y (хотя их и можно выполнять параллельно на двух независимых вычислителях).

2. Вопросы реализации первого и третьего этапов алгоритма на современных ЭВМ

Итак, на каждом шаге первого этапа алгоритма Видемана — Копперсмита выполняются следующие операции:

— умножение матрицы W_i размера $M \times n$ на сильно разреженную матрицу B размера $M \times M$: $W_{i+1} = BW_i$;

— умножение матрицы W_{i+1} на матрицу Z размера $m \times M$: $a_{i+1} = ZW_{i+1}$.

На третьем этапе выполняется вычисление вектора $v = \sum_{j=0}^{d-1} B^j Y f_{j+1}$, где f_j — векторы длины n (коэффициенты аннулирующего многочлена). Данные вычисления можно также выполнять параллельно на независимых вычислителях, разбивая B^jY на n' блоков по 64 столбца, а каждый вектор f_j — на n' составляющих длины 64. Поскольку на втором этапе может быть получено до n различных векторных аннулирующих многочленов $f_i(x) = \sum_{j=0}^{d_i} f_j^{(i)} x^j$, $i = 1, \ldots, n$, на третьем этапе можно одновременно вычислять n векторов $v_i = \sum_{j=0}^{d_i-1} B^j Y f_{j+1}^{(i)}$. Соответственно на каждом шаге третьего этапа выполняются следующие операции:

— умножение матрицы $W_{t-1} = B^{t-1}Y$ размера $M \times n$ на сильно разреженную матрицу B размера $M \times M$: $W_t = BW_{t-1}$;

— умножение матрицы W_t на набор векторов-коэффициентов $f_{t+1}^{(i)}, i = 1, \ldots, n$, каждый длины n, и побитное сложение с результатами предыдущих шагов, т. е. с суммами $\sum_{j=0}^{t-1} B^j Y f_{j+1}^{(i)}$.

Очевидно, что наиболее трудоёмкой операцией каждого шага первого и третьего этапов алгоритма является умножение на разреженную матрицу B: $W_{i+1} = BW_i$. Тем не менее неэффективная реализация других операций может существенно повлиять на время работы алгоритма, поэтому необходимо оптимизировать каждую операцию, выполняемую на шагах различных этапов алгоритма.

2.1. Умножение вектора на разреженную матрицу

Двоичная разреженная матрица обычно представляется в виде последовательности строк (или столбцов); каждая строка определяется числом ненулевых элементов и их номерами. Например, матрица

$$B = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

записывается в виде массива $(\mathbf{2}, 1, 3, \mathbf{1}, 2, \mathbf{2}, 1, 2)$ (жирным выделены веса строк, т. е. число ненулевых элементов). При построчном хранении матрицы B (размера $M \times M$) умножение $W_{i+1} = BW_i$, где W_i — матрица $M \times 64$, выполняется следующим образом (здесь и далее приводится пример реализации на языке C++).

Алгоритм 1. Умножение (слева) на разреженную матрицу, представленную построчно в виде списка ненулевых элементов.

Вход: массив `B` (представление разреженной матрицы), массив `W` (матрица W_i, представленная в виде массива 64-разрядных чисел), число M.

Выход: массив `W2` (представление матрицы $W_{i+1} = BW_i$).

```
1 int st, i, len;    //st - номер строки матрицы B
2                    //len - вес очередной строки
3 __int64 tmp;       //результат умножения на очередную строку
4                    //(64-разрядное число)
5 i = 0;             //индекс в массиве B
6 for (st=0; st<M; st++)  //проходим все строки матрицы
```

```
 7 {
 8     len=B[i];           //вес очередной строки
 9     i++;
10     tmp = 0;
11     while (len--)       //проходим все ненулевые элементы
12     {
13         tmp = tmp ^ W[B[i]];    // ^ - побитное сложение
14         i++;
15     }
16     W2[st] = tmp;       //очередной элемент результата
17 }
```

Конец алгоритма.

Хранение матрицы B можно реализовать и в виде набора весов и номеров ненулевых элементов её столбцов. Для приведённой выше матрицы B её представление по столбцам — $(\mathbf{2}, 1, 3, \mathbf{2}, 2, 3, \mathbf{1}, 1)$. В этом случае умножение $W_{i+1} = BW_i$ можно выполнить с помощью следующего алгоритма.

Алгоритм 2. Умножение (слева) на разреженную матрицу, представленную по столбцам в виде списка ненулевых элементов.

Вход: массив `B` (представление разреженной матрицы по столбцам), массив `W` (матрица W_i, представленная в виде массива 64-разрядных чисел), число M.

Выход: массив `W2` (представление матрицы $W_{i+1} = BW_i$).

```
 1 int st,i,len;   //st - номер столбца матрицы B
 2                 //len - вес очередного столбца
 3 __int64 tmp;    //очередной элемент массива W
 4 for (st=0; st<M; st++) W2[st] = 0;     //обнуляем результат
 5 i = 0;          //индекс в массиве B
 6 for (st=0; st<M; st++)  //проходим все столбцы матрицы
 7 {
 8     len=B[i];           //вес очередного столбца
 9     tmp = W[st];        //следующий элемент массива W
10     i++;
11     while (len--)       //проходим все ненулевые элементы
12     {
13         W2[B[i]] = W2[B[i]] ^ tmp;  //побитное сложение
14         i++;
15     }
16     W2[str] = tmp;      //очередной элемент результата
17 }
```

Конец алгоритма.

Если ω — среднее число ненулевых элементов в строке (столбце) матрицы B, то трудоёмкость обоих алгоритмов равна $\mathrm{O}(M\omega)$. Проведены исследования по сравнению времени работы алгоритмов 1 и 2 для различных размеров входных данных. Результаты экспериментов приведены в табл. 1 (время работы указано в миллисекундах).

Из табл. 1 видно, что алгоритм 1 выигрывает по скорости работы. Это объясняется тем, что внутри основного цикла алгоритма 2 выполняется долгая (по сравнению с остальными) операция записи в память (прибавление элемента массива `W` к соответствующему элементу результата). В алгоритме 1 промежуточное значение очередного элемента выходного массива `W2` хранится в переменной `tmp`, и если в машинном коде

Таблица 1
Время работы алгоритмов 1 и 2

M	ω	Алгоритм 1	Алгоритм 2
100 000	100	24	29
200 000	100	51	62
500 000	100	137	164
1 000 000	100	484	648
2 000 000	100	1445	2269
5 000 000	100	4537	6949
10 000 000	100	11677	17135
200 000	500	251	303
500 000	200	272	321
1 000 000	100	484	648
2 000 000	50	712	1050

эта переменная реализована с помощью одного из регистров процессора, то запись в неё происходит во много раз быстрее, чем запись в ячейку оперативной памяти.

Следует также отметить разницу во времени работы алгоритмов в четырёх последних экспериментах. Несмотря на то, что теоретическая трудоёмкость алгоритмов $O(M\omega)$ для выбранных параметров одинакова, умножение на более плотные матрицы работает быстрее за счёт того, что реже выполняется операция записи в память. Этим также объясняется нелинейный рост времени работы алгоритмов относительно размера матрицы при одинаковом среднем числе ненулевых элементов в строке (столбце).

Алгоритм 1 допускает эффективное распараллеливание по строкам матрицы B: умножение на строки с номерами $n_1+1, n_1+2, \ldots, n_1+k$ можно выполнять независимо от умножения на другие строки; при этом будут получены k элементов выходного массива `W2` с номерами $n_1+1, n_1+2, \ldots, n_1+k$. Тем не менее из-за особенностей работы потоков с общей оперативной памятью не рекомендуется распараллеливать данный алгоритм на большое число потоков.

2.2. Умножение двух векторов

В исходном алгоритме Видемана (т. е. при $m = n = 1$) на каждом шаге первого этапа после умножения вектора на матрицу вычисляется скалярное произведение двух векторов длины M: $a_i = ZW_i$. Трудоёмкость вычисления скалярного произведения равна $2M$ операций в поле GF(2). В общем случае в алгоритме Видемана — Копперсмита требуется перемножить две матрицы размеров $m \times M$ и $M \times n$. При фиксированных m и n это также можно сделать за $O(M)$ операций, используя тривиальный алгоритм. Однако с увеличением параметров m и n трудоёмкость тривиального алгоритма быстро растет, поэтому для ускорения данной процедуры целесообразно использовать более быстрые алгоритмы, например LUT-алгоритм на основе таблиц поиска (Look-Up Tables). Описание LUT-алгоритма можно найти, например, в [8, гл. 6].

Идея метода заключается в следующем. Пусть, для начала, $m = n = 8$, а матрицы Z и W размеров $8 \times M$ и $M \times 8$ представлены в виде массивов байтов `Z` и `W` длины M. Результат умножения $a = ZW$ есть матрица 8×8, представляемая массивом из 8 байтов. Каждая строка результата есть линейная комбинация строк второго множителя: $a_i = \sum\limits_j z_{ij} W_j$. Выделение отдельных элементов z_{ij}, т. е. отдельных битов элементов массива `Z`, является достаточно долгой операцией. Вместо этого удобнее сохранять промежуточный результат вычисления линейных комбинаций $\sum\limits_j z_{ij} W_j$ сра-

зу для всех $i = 1, \ldots, 8$, а именно: в дополнительный массив LUT из $2^8 = 256$ байтов в ячейку с номером $k \in \{0, \ldots, 255\}$ запишем сумму строк второго множителя с такими номерами $j \in \{1, \ldots, M\}$, для которых значение столбца первого множителя (т. е. элемента массива Z) равно k:

$$\mathtt{LUT}_k = \sum_{\substack{j \in \{1,\ldots,M\}: \\ Z_j = k}} W_j.$$

Создание массива LUT выполняется за один проход по массивам Z и W: для $j = 1, \ldots, M$ выполняем побитное сложение $\mathtt{LUT}_{Z_j} := \mathtt{LUT}_{Z_j} \oplus W_j$. Получение результата умножения из массива LUT тривиально. Например, первая строка результата есть линейная комбинация элементов массива LUT, первый (младший) бит номеров которых равен 1.

Для случая $m = n = 64$ невозможно выделить дополнительную память LUT объёмом 2^{64} 64-разрядных элементов. Вместо этого будем использовать 8 различных таблиц LUT, разбив вычисления по номерам байтов в строке результата. При больших значениях m и n удобнее решать задачу, разбивая на подзадачи с $m = n = 64$.

Итоговый алгоритм на языке C++ можно записать следующим образом.

Алгоритм 3. Умножение матриц размеров $64 \times M$ и $M \times 64$: $a = ZW$.

Вход: массивы Z и W 64-разрядных чисел, представляющие матрицы Z и W соответственно, число M.

Выход: массив A из 64-х 64-разрядных чисел, представляющий результат (матрицу a размера 64×64).

```
 1 __int64 LUT[256 * 8];
 2 __int64 zj, wj;
 3 int i, j, k;
 4 for (j = 0; j < 256*8; j++) LUT[j] = 0;
 5 for (j = 0; j < M; j++)
 6 {
 7     zj = Z[j];
 8     wj = W[j];
 9     for (i=0; i<8; i++)
10         LUT[i*256 + (zj >> 8*i) & 0xFF] ^= wj;
11     //при m=n=8 здесь вместо цикла по i было бы только
12     //одно побитное сложение: LUT[Z[j]] ^= W[j];
13 }
14 for (i = 0; i < 64; i++) a[i] = 0;
15 for (i = 0; i < 8; i++)
16 {
17     for (j = 0; j < 256; j++)
18         if ((j >> i) & 1)
19             for (k = 0; k < 8; k++)
20                 a[8*k + i] ^= LUT[k*256 + j];
21 }
```

Конец алгоритма.

Алгоритм 3 работает на ЭВМ существенно быстрее тривиального алгоритма (см. табл. 2). Выигрыш достигается за счёт того, что результатом является небольшая двоичная матрица размера 64×64, а также за счёт эффективного использования дополнительной памяти и многоразрядности ЭВМ.

Таблица 2

Время работы алгоритма умножения матриц $64 \times M$ и $M \times 64$

M	Тривиальный алгоритм	Алгоритм 3
100 000	4,69	0,42
1 000 000	47,02	4,27
10 000 000	470,03	42,59
100 000 000	4715,9	436,8

2.3. Умножение на коэффициенты аннулирующего многочлена

При вычислении вектора $v = \sum\limits_{j=0}^{d-1} B^j Y f_{j+1}$ на каждом шаге третьего этапа необходимо, помимо вычисления матрицы $W_j = B^j Y$, реализовать умножение этой матрицы размера $M \times n$ на вектор f_{j+1} длины n. Если $n = 64$ и W_j представляется в виде массива длины M, состоящего из 64-разрядных чисел, а вектор f_{j+1} записан как одно 64-разрядное число, то умножение W_j на f_{j+1} сводится к операциям побитного умножения и вычисления чётности двоичного веса Хэмминга 64-разрядных чисел. С помощью сдвига и побитного сложения в 2 раза уменьшается размер задачи вычисления чётности двоичного веса вектора, а для 16-разрядных чисел соответствующие значения можно вычислить заранее и хранить в памяти. В результате умножение W_j на f_{j+1} и сложение с результатом предыдущих шагов выполняется за $6M$ операций с 64-разрядными числами и M операций обращения к массиву двоичных весов 16-разрядных чисел. Однако при наличии достаточного объёма оперативной памяти можно выполнить требуемое умножение и сложение более эффективно. Поскольку отображение $\chi : V_{64} \to V_1$, реализующее вычисление чётности двоичного веса Хэмминга 64-разрядного вектора, является линейным, можно на каждом шаге вычислять только побитное произведение f_{j+1} и элементов матрицы W_j и складывать их (побитно) с соответствующими результатами предыдущих шагов, а отображение χ вычислить после выполнения всех d шагов третьего этапа. При этом необходимый объём памяти для хранения промежуточных сумм $\sum\limits_j B^j Y f_{j+1}$ возрастёт в 64 раза.

Заметим, что требуемый объём оперативной памяти можно уменьшить в 2 раза с помощью M сдвигов и M сложений на каждом шаге.

3. Реализация первого и третьего этапов алгоритма на вычислителях кластерного типа

Рассмотрим особенности, возникающие при реализации алгоритма Видемана — Копперсмита на кластерах. Под кластером, или вычислителем кластерного типа, будем понимать некоторый набор ЭВМ (узлов), соединённых между собой высокоскоростной сетью. Поскольку в этом случае отсутствует возможность использования быстрой общей памяти, необходимо решать вопросы о распределении обрабатываемых данных по узлам кластера, координации работы узлов, а также о способе обмена данными между узлами.

3.1. Распределение данных и распараллеливание вычислений

При выборе способа представления данных и распределения их между узлами кластера необходимо исходить из соображений эффективной реализации наиболее трудо-

ёмких операций алгоритма. В случае алгоритма Видемана — Копперсмита наиболее трудоёмкой операцией, выполняемой на каждом шаге первого и третьего этапов, является умножение векторов на разреженную матрицу. Если использовать алгоритм 1, то данная операция легко распараллеливается между узлами кластера путём разбиения разреженной матрицы на блоки строк примерно одинакового веса. Каждый узел хранит свою порцию строк разреженной матрицы B, а также всю матрицу W_j. После умножения на каждом узле находится соответствующая часть результата, т. е. матрицы $W_{j+1} = BW_j$. Перед следующим умножением необходимо выполнить обмен между узлами, чтобы на каждом из них вновь была полная копия правого множителя.

Строки матрицы B необходимо распределять по узлам кластера таким образом, чтобы каждый узел тратил примерно одинаковое время на умножение матрицы W_j на свою порцию строк матрицы B. Как правило, используются узлы кластера с одинаковой производительностью, поэтому время умножения определяется весом строк разреженной матрицы B, доставшихся данному узлу, и их количеством. Важно учитывать время записи результата умножения W_j на одну строку матрицы B, поэтому умножение на несколько «тяжелых» строк матрицы B выполняется быстрее, чем умножение на большее количество более «легких» строк с таким же суммарным количеством ненулевых элементов (см. п. 2.1).

Умножение двух матриц Z и W_j для получения очередного элемента последовательности Крылова $a_j = ZW_j = ZB^jY$, особенно с помощью алгоритма 3, выполняется намного быстрее, чем умножение W_j на разреженную матрицу B, поэтому данную операцию можно не распараллеливать и выполнять на одном из узлов после формирования на нем полной копии матрицы W_j. При этом на выбранном узле необходимо хранить всю матрицу Z.

Вычисление сумм $\sum\limits_j B^jYf_{j+1}$ на шагах третьего этапа также является достаточно быстрой операцией, к тому же её легко распараллелить, так как одновременно может вычисляться до n таких сумм.

3.2. Передача данных и координация работы узлов

Для выполнения алгоритма на кластере необходимо выделить один из узлов для координации действий. На этом управляющем узле хранится результат вычислений первого этапа — матричная последовательность (a_i). Этот же узел выполняет умножение матриц Z и W_j на каждом шаге первого этапа.

Управляющий узел должен следить за ходом выполнения шагов алгоритма. Он даёт команду остальным узлам на выполнение умножения на разреженную матрицу, ждёт отчёта узлов о завершении умножения, затем инициирует процедуру обмена данными между узлами. Как отмечалось выше, после умножения матрицы W_j на разреженную матрицу B на каждом узле будет получена часть матрицы $W_{j+1} = BW_j$, соответствующая доли матрицы B на этом узле. Перед следующим умножением, а также перед вычислением матрицы a_{j+1} необходимо получить полную копию W_{j+1} на каждом узле кластера.

Наиболее эффективным способом обмена данными между узлами в этом случае представляется передача данных по кольцу. Кратко изложим идею этого способа. Пусть имеется k узлов и на i-м узле хранится соответствующая часть X_i некоторого массива X, $i = 1, \ldots, k$. Заметим, что размер части массива X на некоторых узлах может быть равен нулю. Передача осуществляется следующим образом. Сначала каждый узел передаёт предыдущему узлу свою часть массива X, т. е. узел с номером i передает узлу с номером $i - 1$ часть X_i (первый узел передаёт X_1 на узел с номером k). По-

сле этого каждый узел передаёт предыдущему только что полученную от следующего узла часть: i-й узел передает часть X_{i+1}. И так далее. На j-м шаге узел с номером i передает часть X_{i+j-1} и получает часть X_{i+j}, $j = 1, \ldots, k$. Через $k-1$ шагов каждый узел будет иметь весь массив X. Применительно к алгоритму Видемана — Копперсмита, при равномерном распределении данных между k узлами каждый узел отправляет и получает $\dfrac{k-1}{k}64M$ битов. Заметим, что нет необходимости координировать данную процедуру на различных шагах: каждый узел самостоятельно продолжает передачу и приём данных в необходимом объёме, если перед началом процедуры ему известен размер части на предыдущем узле и общий размер массива X.

При использовании технологии `MPI` выполнить подобный обмен между процессами одной группы можно с помощью готовой процедуры `MPI_ALLGATHER` (см., например, [9]). Реализация обмена была подготовлена путём использования системных функций для установления соединения и пересылки данных между двумя узлами (библиотека `Winsock` под ОС Windows и `sys/socket` под ОС Linux). Заметим, что если сетевые карты узлов поддерживают режим полного дуплекса (что типично для современных вычислителей), то удаётся достичь максимально возможной скорости передачи для данной сети. Например, при $M = 10\,000\,000$ и равномерном распределении матрицы между $k = 10$ узлами, соединёнными сетью `gigabit Ethernet` (передача до 10^9 бит/с), сетевой обмен между узлами на каждом шаге выполняется примерно за 585 мс (каждый узел отправляет и получает $576 \cdot 10^6$ битов).

В случае, когда умножение узлами матрицы W_j на их порции разреженной матрицы B выполняется за время, сравнимое с временем обмена результатами умножения, имеет смысл выполнять умножение и передачу одновременно. При этом каждый узел начинает выполнять умножение и сразу же передаёт либо вычисленную им часть матрицы W_{j+1}, либо полученную от соседнего узла. Из-за нагрузки на оперативную память снижение времени работы алгоритма в 2 раза невозможно, однако при оптимальном балансе между временем умножения и передачи удаётся получить выигрыш примерно в 1,5 раза. Заметим, что подобный подход исключает возможность обмена результатами вычисления W_{j+1} между узлами с помощью готовых решений технологии `MPI`.

Результаты работы третьего этапа алгоритма Видемана — Копперсмита могут занимать большой объём памяти, и в этом случае их можно хранить на отдельно выделенных для этого узлах кластера. По окончании выполнения третьего этапа эти узлы передают результаты вычислений на управляющий узел.

3.3. Реализация проверки и сохранения промежуточных результатов

При решении задач больших размеров важными аспектами реализации алгоритма являются проверка и сохранение промежуточных результатов. Наибольшей вычислительной и ёмкостной трудоёмкостью обладает операция умножения на разреженную матрицу. Наибольший объём данных, передаваемых между узлами кластера, также связан с этой операцией. Поэтому проверка правильности вычисления матриц W_i, $i = 1, 2, \ldots$, представляет наиболее важную задачу с точки зрения контроля правильности работы алгоритма.

Проверка правильности вычисления матриц $W_i = B^i Y$ может быть реализована следующим образом. Пусть $i_{\text{test}} \in \mathbb{N}$ — параметр метода, Z_{test} — случайная матрица размера $m_{\text{test}} \times M$. Вычислим значение $Z_{\text{test}} B^{i_{\text{test}}}$. Далее выполняем шаги первого или третьего этапов алгоритма и на шаге $i = i_{\text{test}}$ выполняем проверку: $(Z_{\text{test}} B^{i_{\text{test}}})\, W_0 \stackrel{?}{=}$

$\stackrel{?}{=} Z_{\text{test}} W_{i_{\text{test}}}$. Если равенство не выполняется, то в процессе выполнения данного этапа алгоритма была допущена ошибка и необходимо начать выполнение этого этапа сначала. В случае выполнения сохраняем значение $W_{i_{\text{test}}}$ и выполняем следующие i_{test} шагов, после чего (на шаге $i = 2i_{\text{test}}$) снова выполняем проверку: $(Z_{\text{test}} B^{i_{\text{test}}})\, W_{i_{\text{test}}} \stackrel{?}{=} \stackrel{?}{=} Z_{\text{test}} W_{2i_{\text{test}}}$. Если равенство не выполняется, возвращаемся к шагу $i = i_{\text{test}}$, иначе сохраняем $W_{2i_{\text{test}}}$ и продолжаем выполнение шагов алгоритма. И так далее. На шаге $i = k \cdot i_{\text{test}}$, $k = 1, 2, \ldots$, проверяем равенство $(Z_{\text{test}} B^{i_{\text{test}}})\, W_{(k-1)i_{\text{test}}} \stackrel{?}{=} Z_{\text{test}} W_{k \cdot i_{\text{test}}}$ и сохраняем значение $W_{k \cdot i_{\text{test}}}$ либо возвращаемся на шаг $i = (k-1)i_{\text{test}}$.

Для обнаружения ошибок достаточно использовать матрицу Z_{test} размеров $64 \times M$. В качестве Z_{test} можно взять первые 64 столбца случайной матрицы Z. Однако иногда вместо случайной матрицы Z выбирают матрицу, состоящую из первых m столбцов единичной матрицы (см. далее), и в этом случае Z_{test} лучше выбирать случайно. Параметр i_{test} выбирают исходя из размеров задачи и надёжности вычислителя. При факторизации 768-битного числа, описанной в [4], проверка выполнялась каждые $i_{\text{test}} = 2^{14}$ шагов (всего на первом этапе было вычислено $L \approx 565000$ элементов последовательности (a_i), причём в работе не сказано о наличии сбоев при вычислении).

При выполнении первого этапа алгоритма Видемана — Копперсмита в качестве промежуточных результатов выступают элементы последовательности (a_i) и матрицы $W_i = B^i Y$, поэтому их сохранение реализуется автоматически при выполнении проверки правильности вычислений. В случае сбоя на шаге i (например, при отключении питания вычислителя и т. п.) достаточно вернуться к шагу $k \cdot i_{\text{test}}$, где $k = \lfloor i / i_{\text{test}} \rfloor$, и продолжить выполнение алгоритма. На третьем же этапе необходимо выполнять сохранение промежуточных сумм $\sum\limits_j B^j Y f_{j+1}$. Эта процедура также выполняется каждые i_{test} шагов, и промежуточные суммы можно хранить либо на узлах, которые эти суммы вычисляют, либо на управляющем узле кластера. В последнем случае необходимо выполнить передачу этих значений с соответствующих узлов кластера на управляющий узел. Заметим, что если применяется метод ускорения вычислений сумм $\sum\limits_j B^j Y f_{j+1}$, описанный в п. 2.3, то объём передаваемых и сохраняемых данных не возрастает, так как перед передачей и сохранением промежуточных результатов можно выполнить вычисление чётности двоичных весов Хэмминга и хранить собственно значения $\sum\limits_j B^j Y f_{j+1}$.

3.4. Некоторые способы снижения вычислительной и ёмкостной сложности алгоритма

Приведём некоторые незначительные изменения алгоритма Видемана — Копперсмита, которые позволяют снизить требуемый объём оперативной памяти вычислителя и сократить трудоёмкость отдельных операций.

Первое изменение связано с выбором матрицы Z. В исходном описании алгоритма сказано, что матрица Z должна выбираться случайным образом. При этом появляются затраты на её хранение и умножение на матрицы W_i. Действительно, если матрицу Y можно разбить на подматрицы по 64 столбца и выполнять первый и третий этапы алгоритма для каждой из этих подматриц на несвязанных между собой вычислителях, то матрица Z должна храниться на каждом вычислителе целиком, что при большом значении m приводит к существенным затратам оперативной памяти. Чтобы этого избежать, в качестве Z можно использовать первые m строк единичной матрицы. В этом случае матрицу Z хранить не нужно, а результат умножения $a_i = Z W_i$ — это первые

m строк матрицы W_i. Следует отметить, что в этом случае нарушается обоснование надёжности алгоритма [3], основанное на вероятностных соображениях, однако многочисленные эксперименты показывают, что для невырожденных матриц A алгоритм работает нормально. В качестве компромисса можно использовать матрицу Z, столбцы которой имеют вес 1. Если $m \leqslant 256$, то для хранения такой матрицы достаточно M байтов вместо $mM/8$, а умножение на неё матрицы W_i выполняется, очевидно, более эффективно, чем в общем случае.

Более важным является снижение трудоёмкости умножения матрицы W_i на разреженную матрицу B. При изложении алгоритма в п. 1 было сказано, что B состоит из первых M столбцов матрицы A. Тем не менее при неравномерном распределении веса столбцов имеет смысл составлять матрицу B из столбцов меньшего веса; иными словами, перед выполнением алгоритма выполнить перестановку столбцов матрицы A (перенумерацию неизвестных), чтобы снизить вес матрицы B и тем самым снизить трудоёмкость умножения на эту матрицу.

Вес строк матрицы A также может быть распределён неравномерно. В результате работы алгоритма Видемана — Копперсмита получается не одно, а сразу n решений системы. Если перед выполнением алгоритма отбросить одно уравнение системы, т. е. одну строку матрицы A, и найти n решений для такой системы, то можно получить не менее $n-1$ решений исходной системы. Действительно, пусть $w_1, \ldots, w_{k-1}, w_k$ — решения урезанной системы, не удовлетворяющие отброшенному уравнению, $1 \leqslant k \leqslant n$. Тогда $k-1$ векторов $w_1 \oplus w_k, \ldots, w_{k-1} \oplus w_k$ будут удовлетворять и отброшенному уравнению, и, очевидно, всем остальным. Следовательно, если требуется получить небольшое число решений (например, одно), наиболее тяжёлые строки матрицы A (т. е. строки большого веса) можно изначально исключить; при исключении t строк из полученных n решений можно сформировать не менее $n-t$ решений исходной системы уравнений.

4. Особенности применения алгоритма Видемана — Копперсмита в рамках метода решета числового поля факторизации целых чисел

Как уже было отмечено, нахождение решений большой разреженной системы однородных линейных уравнений над полем GF(2) является одним из этапов метода решета числового поля, применяемого для факторизации больших целых чисел. При этом матрица решаемой системы не является случайной и имеет ряд ярко выраженных особенностей, что необходимо учитывать при выполнении алгоритма Видемана — Копперсмита.

Не вдаваясь в подробности, можно сказать, что каждая строка матрицы соответствует очередному простому числу, а элементы строки представляют чётность показателя степени, с которой данное простое число входит в разложение некоторых выбранных целых чисел. Для случайного натурального числа n и простого p вероятность того, что p^k делит n, а p^{k+1} не делит n, примерно равна $(p-1)/p^{k+1}, k = 0, 1, \ldots$ (понятно, что при $k > \log_p n$ эта вероятность равна нулю). Соответственно вероятность того, что элемент строки, соответствующей простому числу p, равен 1, примерно равна $\sum\limits_{k=1}^{\infty} \dfrac{p-1}{p^{2k}} = \dfrac{1}{p+1}$. Поэтому первые строки матрицы имеют достаточно большой вес (для первой строки, соответствующей $p = 2$, вес равен примерно $N/3$), а затем с ростом номера строки вес строки быстро снижается. Используя идею п. 3.4, можно отбросить несколько первых строк (т. е. уравнений), выполнить алгоритм, а затем из полученных решений урезанной системы составить несколько решений исходной системы.

Ещё один важный момент касается предварительной обработки матрицы решаемой разреженной СОЛУ. Например, если вес строки равен 1, то можно сразу выполнить исключение соответствующей переменной из решаемой системы. Аналогично, если некоторая переменная встречается только в одном уравнении (т. е. вес столбца равен 1), то соответствующие строку и столбец можно удалить перед выполнением алгоритма поиска решений. Применяются и другие операции (см., например, [10]), приводящие к снижению размеров M и N системы, но увеличивающие средний вес строки ω матрицы системы. Такая предварительная обработка матрицы называется *фильтрацией* и выполняется до того момента, пока эти операции приводят к снижению числа $M^2\omega$, определяющего трудоёмкость первого и третьего этапов алгоритма Видемана — Копперсмита (при реализации алгоритма на кластере необходимо также учитывать общее время передачи данных по сети, которое пропорционально M^2). При факторизации 768-битного числа [4] исходная матрица $35\,288\,334\,017 \times 47\,762\,243\,404$ была за 15 дней ужата до размеров $192\,795\,550 \times 192\,796\,550$ со средним весом строки $\omega = 144$.

При выполнении фильтрации решение о её прекращении принимается, исходя из оптимальных соотношений между размерами матрицы и её весом, приводящих к минимальной трудоёмкости алгоритма нахождения решений СОЛУ. Для одной и той же матрицы данные соотношения могут быть различными при использовании разных вычислителей для выполнения алгоритма Видемана — Копперсмита. Поэтому при решении сильно разреженной СОЛУ фильтрация является неотъемлемым этапом и должна согласовываться с выбранным вычислителем и параметрами алгоритма Видемана — Копперсмита.

Заключение

Рассмотрены вопросы, связанные с реализацией первого и третьего этапов алгоритма Видемана — Копперсмита. Наиболее трудоёмкой операцией этих этапов является умножение блока векторов на разреженную матрицу. Эта операция может быть эффективно распараллелена путем независимого умножения на различные строки разреженной матрицы, однако при реализации на вычислителях кластерного типа требуется дополнительный обмен между узлами для получения результата умножения.

Уделено внимание и другим операциям первого и третьего этапов алгоритма Видемана — Копперсмита, поскольку их неэффективная реализация может существенно увеличить время работы алгоритма. В рамках исследования особенностей реализации алгоритма на кластерах рассмотрены вопросы проверки и сохранения промежуточных результатов вычислений для защиты от сбоев в работе вычислителя. Приведены также некоторые особенности, возникающие при использовании алгоритма Видемана — Копперсмита в рамках факторизации больших целых чисел методом решета числового поля.

ЛИТЕРАТУРА

1. *Coppersmith D.* Fast evaluation of logarithms in fields of characteristic two // IEEE Trans. Inform. Theory. 1984. V. IT-30(4). P. 587–594.
2. *Montgomery P. L.* A block Lanczos algorithm for finding dependencies over GF(2) // EUROCRYPT'95. LNCS. 1995. V. 921. P. 106–120.
3. *Coppersmith D.* Solving linear equations over GF(2) via block Wiedemann algorithm // Math. Comp. 1994. V. 62(205). P. 333–350.
4. *Kleinjung T., Aoki K., Franke J., et al.* Factorization of a 768-bit RSA modulus // CRYPTO-2010. LNCS. 2010. V. 6223. P. 333–350.

5. *Wiedemann D. H.* Solving sparse linear equations over finite fields // IEEE Trans. Inform. Theory. 1986. V. IT-32(1). P. 54–62.
6. *Рыжов А. С.* О реализации алгоритма Копперсмита для двоичных матричных последовательностей на вычислителях кластерного типа // Прикладная дискретная математика. 2013. № 3. С. 112–122.
7. *Thomé E.* Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm // J. Symb. Comput. 2002. No. 33. P. 757–775.
8. *Ахо А., Хопкрофт Д., Ульман Дж.* Построение и анализ вычислительных алгоритмов. М.: Мир, 1979. 536 с.
9. `www.open-mpi.org/doc/v1.6` — Open MPI v1.6.4 documentation. 2013.
10. *Cavallar S.* Strategies for filtering in the number field sieve // Proc. ANTS IV. LNCS. 2000. V. 1838. P. 209–231.