Распараллеливание матричных алгоритмов вычисления базисов Грёбнера

Д. Е. АЛЕКСАНДРОВ
Московский государственный университет им. М. В. Ломоносова
e-mail: denis57@bk.ru

В. В. ГАЛКИН
Московский государственный университет им. М. В. Ломоносова
e-mail: galkin-vv@ya.ru

А. И. ЗОБНИН
Московский государственный университет им. М. В. Ломоносова
e-mail: Alexey.Zobnin@gmail.com

М. В. ЛЕВИН
Московский государственный университет им. М. В. Ломоносова
e-mail: Michael.Levin@gmail.com

УДК 512.622+519.6

Ключевые слова: базисы Грёбнера, алгоритм F4, параллельные алгоритмы, матричная редукция.

Аннотация

В работе описываются последовательные и параллельные реализации алгоритма F4 для построения базисов Грёбнера полиномиальных идеалов.

Abstract

D. E. Alexandrov, V. V. Galkin, A. I. Zobnin, M. V. Levin, Parallelization of matrix algorithms for Gröbner basis computation, Fundamentalnaya i prikladnaya matematika, vol. 14 (2008), no. 4, pp. 35—64.

Sequential and parallel implementations of the F4 algorithm for computing Gröbner bases of polynomial ideals are discussed.

1. Введение

Задачи исследования систем уравнений полиномиального типа, часто решаемые с помощью базисов Грёбнера, возникают в различных теоретических и прикладных разделах математики, механики, криптографии и т. п. Вопросы, которые можно ставить о таких системах, не обязательно требуют поиска всех

Фундаментальная и прикладная математика, 2008, том 14, № 4, с. 35—64.
© 2008 *Центр новых информационных технологий МГУ, Издательский дом «Открытые системы»*

решений системы; они могут быть связаны с размерностью пространства решений, с исключением из системы определённых неизвестных, с приведением системы к более удобному виду и т. д. Базис Грёбнера — традиционный инструмент для конструктивного исследования таких систем — позволяет теоретически дать ответ на поставленные вопросы, но его практическое вычисление обычно сопряжено с колоссальными вычислительными затратами. Кроме традиционного алгоритма Бухбергера с различными оптимизациями [3, 8, 9, 13], существуют также инволютивные алгоритмы [14], алгоритмы, основанные на вычислении базиса модуля сизигий [17], матричные алгоритмы [11] и алгоритмы с исключением нулевых редукций [12]. Большинство алгоритмов работают с S-полиномами — специальными многочленами, построенными по исходным образующим. Некоторые из этих алгоритмов имеют дополнительные ограничения на исходные многочлены, при которых возможна их эффективная работа. Основное время тратится на редукцию этих S-полиномов; если в результате получается ненулевой остаток, то он добавляется к промежуточному базису. Все эти алгоритмы являются последовательными. В научной литературе в 1990-х гг. было распространено мнение, что алгоритмы построения базисов Грёбнера в принципе плохо поддаются распараллеливанию [4, 10]. На сегодняшний день известно сравнительно немного параллельных реализаций тех или иных алгоритмов [7, 15, 22, 23].

В работе рассмотрены варианты распараллеливания алгоритма F4. Он относится к так называемым алгоритмам матричного типа и хорошо зарекомендовал себя на практике [11, 19]. Главное отличие матричных алгоритмов от традиционных методов заключается в том, что процесс редукции не формулируется в терминах многочленов, а переводится на линейно-алгебраический язык и сводится к задаче приведения большой разреженной матрицы к ступенчатому виду. Методы работы с разреженными матрицами изучены намного лучше и потому могут быть реализованы эффективнее. Другое отличие состоит в том, что на очередном шаге алгоритма рассматривается не один S-полином, а сразу некоторое подмножество S-полиномов. Критерии формирования этого подмножества могут давать различную эффективность в зависимости от задачи. Так, для однородных систем уравнений хорошим выбором является нормальная стратегия. Распараллеливание алгоритмов матричного типа производилось в их линейно-алгебраической части, поскольку именно матричная редукция занимает в среднем 90 % времени работы последовательной версии алгоритма.

Были разработаны специальные методы параллельного приведения сильно разреженной матрицы к ступенчатому виду. Для этого пришлось преодолеть ряд сложностей. Во-первых, столбцы таких матриц соответствуют мономам исходных многочленов. Они должны быть упорядочены согласно выбранному мономиальному порядку, поэтому элементарные преобразования столбцов матрицы запрещены. Из-за этого многие специальные параллельные матричные алгоритмы оказываются неприменимыми. Кроме того, несмотря на свою разреженность, матрицы, возникающие при вычислении базисов Грёбнера, имеют особую структуру, полученную в результате специальной процедуры формирования матрицы — препроцессинга. Так, приведение случайной матрицы к ступенчато-

му виду с таким же коэффициентом разреженности может оказаться в среднем эффективней, чем приведение матрицы, реально появившейся в алгоритме. Все эти особенности были учтены при распараллеливании. В результате был создан настраиваемый параллельный блочный метод, в котором элементарные преобразования производятся не относительно отдельной строки, а сразу относительно специально выбранного блока. Особое внимание было уделено настройке стратегии выбора S-пар, стадии формирования матрицы и препроцессинга: они были переписаны таким образом, чтобы по возможности сохранить разреженность матрицы и уменьшить время вычислений.

Отдельное исследование было посвящено алгоритмам работы с системами уравнений над полем из двух элементов, решение которых также ищется в этом поле (а не в его расширении). Такие системы часто встречаются, например, в криптографических задачах. Это эквивалентно исследованию системы с добавленными уравнениями поля $x_i^2 = x_i$ для каждой переменной. Тогда с учётом редукции можно считать, что все мономы обрабатываемых многочленов свободны от квадратов. Для различных последовательных и параллельных алгоритмов были реализованы специальные высокоэффективные методы для хранения и обработки таких многочленов.

Распараллеливание осуществлялось с использованием программного интерфейса MPI. Была создана специальная библиотека libF4mpi на языке программирования C++. Реализованные параллельные алгоритмы показали довольно неплохие результаты, что подтверждается экспериментальными вычислениями на многопроцессорных кластерах с распределённой памятью как для стандартных тестовых примеров, на которых традиционно проверяют работу систем компьютерной алгебры, так и на специфических прикладных и теоретических задачах.

2. Базисы Грёбнера и алгоритм Бухбергера

Мы будем рассматривать идеалы в кольце многочленов $F[x_1, \ldots, x_n]$, где F — конечное поле F_p из p элементов. Мы называем мономом выражение $x_1^{a_1} \ldots x_n^{a_n}$, где $a_i \geqslant 0$, а термом — моном с коэффициентом (одночлен). Через $\preccurlyeq$ мы будем обозначать мономиальное упорядочение — линейный порядок на множестве мономов $\mathbb{M}$, удовлетворяющий следующим свойствам для произвольных мономов M, N и Q:

— $M \prec N \iff M \cdot Q \prec N \cdot Q$;
— $M \succcurlyeq 1$.

Мы будем предполагать, что переменные упорядочены так, что $x_1 \succ x_2 \succ \ldots \succ x_n$. Классические примеры мономиальных упорядочений — лексикографическое (Lex), степенное лексикографическое (DegLex) и степенное обратное лексикографическое (DegRevLex). Пусть $M = \prod_{i=1}^{n} x_i^{a_i}$ и $N = \prod_{i=1}^{n} x_i^{b_i}$. Тогда

– $M \prec_{\mathrm{Lex}} N \iff (a_1, \ldots, a_n) <_{\mathrm{Lex}} (b_1, \ldots, b_n)$, т. е. $a_1 < b_1$, или $a_1 = b_1$ и $a_2 < b_2$, или $a_1 = b_1, a_2 = b_2$ и $a_3 < b_3$ и т. д.;

– $M \prec_{\mathrm{DegLex}} N \iff \Big(\sum\limits_{i=1}^{n} a_i,\ a_1, \ldots, a_n\Big) <_{\mathrm{Lex}} \Big(\sum\limits_{j=1}^{n} b_j,\ b_1, \ldots, b_n\Big)$;

– $M \prec_{\mathrm{DegRevLex}} N \iff \Big(\sum\limits_{i=1}^{n} a_i,\ b_n, \ldots, b_1\Big) <_{\mathrm{Lex}} \Big(\sum\limits_{j=1}^{n} b_j,\ a_n, \ldots, a_1\Big)$.

Если зафиксировано мономиальное упорядочение, то можно вычислить старший моном $\mathrm{HM}(f)$, старший коэффициент $\mathrm{HC}(f)$ и старший терм $\mathrm{HT}(f)$ ненулевого многочлена f. Более того, можно ввести понятие алгоритма редукции многочлена f по множеству многочленов G. Редукция последовательно заменяет очередное подходящее слагаемое $c \cdot M$ многочлена f на многочлен вида $c \cdot M - hg$, где $g \in G$, $h \in F[x_1, \ldots, x_n]$ и $\mathrm{HT}(hg) = c \cdot M$. Результатом редукции может быть либо 0, либо многочлен f', старший моном которого не больше $\mathrm{HM}(f)$. Мы будем записывать это так: $f \underset{G}{\longrightarrow} f'$.

Существует несколько эквивалентных определений базиса Грёбнера [3, 5, 8]. Приведём здесь два из них.

Определение. Множество $G \subset I$ называется базисом Грёбнера идеала I, если идеал, порождённый старшими мономами G, совпадает с идеалом, порождённым старшими мономами I.

Определение. Множество $G \subset I$ называется базисом Грёбнера идеала I, если любой $f \in I$ редуцируется к нулю относительно G каким-либо фиксированным алгоритмом редукции.

Базисы Грёбнера в кольце многочленов от конечного числа переменных над полем используются для конструктивного описания идеалов и фактор-колец [3, 5, 8]. В частности, с их помощью решается задача принадлежности произвольного многочлена рассматриваемому идеалу. Имеет место теорема Гильберта о базисе, утверждающая, что кольцо многочленов нётерово (т. е. каждый идеал в нём имеет конечный набор образующих). Именно этот факт позволяет алгоритмически вычислить базис Грёбнера за конечное число шагов. Имеется несколько алгоритмов вычисления базисов Грёбнера. Каждый из них использует в той или иной мере следующий факт: существуют «хорошие» конечные подмножества идеала I, такие что достаточно проверить редуцируемость к нулю элементов этих множеств относительно G, вместо того чтобы проверять редуцируемость к нулю *каждого* элемента идеала. Классическим примером таких подмножеств является набор S-полиномов исходных образующих.

Определение. Пусть зафиксирован мономиальный порядок. S-многочленом (или S-полиномом) многочленов f и g называется многочлен

$$\frac{\mathrm{HOK}(\mathrm{HM}(f), \mathrm{HM}(g))}{\mathrm{HT}(f)} f - \frac{\mathrm{HOK}(\mathrm{HM}(f), \mathrm{HM}(g))}{\mathrm{HT}(g)} g.$$

Смысл этого определения такой: многочлены f и g сначала домножаются минимальным образом на некоторые мономы так, чтобы у них оказался одинаковый старший моном (равный НОК исходных старших мономов). Затем из

одного домноженного многочлена вычитается другой с таким подходящим коэффициентом, чтобы эти старшие мономы сократились.

Основное утверждение теории базисов Грёбнера состоит в следующем (см. [3, 5, 8]).

Теорема. *Набор многочленов $f_1, f_2, \ldots, f_n$ является базисом Грёбнера идеала $(f_1, f_2, \ldots, f_n)$ тогда и только тогда, когда для любой пары (f_i, f_j) S-полином $S(f_i, f_j)$ редуцируется относительно этого набора к нулю.*

Отсюда получается алгоритм Бухбергера: до тех пор пока есть нерассмотренные пары многочленов, выбираем одну из них, строим для неё S-полином, редуцируем его по множеству текущих образующих идеала и если получился не ноль, то добавляем результат к множеству образующих. При каждом пополнении множества образующих увеличивается строго возрастающая цепочка идеалов их старших мономов. Поэтому ввиду нётеровости кольца $F[x_1, \ldots, x_n]$ через конечное число шагов алгоритм закончит работу.

Алгоритм Бухбергера является исторически первым алгоритмом построения базиса Грёбнера. Описанная простая версия этого алгоритма является крайне неэффективной. Другие алгоритмы построения базиса Грёбнера тоже основаны на переборе S-полиномов и применении редукции. Повышение эффективности алгоритмов достигается, как правило, за счёт более грамотного перебора S-полиномов и за счёт применения техники линейной алгебры.

Хорошо известны два критерия Бухбергера, которые гарантируют, что те или иные пары многочленов (критические пары) заведомо рассматривать не нужно, так как полученный S-полином будет редуцироваться к нулю по текущему множеству многочленов:

1) если старшие мономы многочленов f_i и f_j взаимно просты, то $S(f_i, f_j)$ редуцируется к нулю по множеству $\{f_i, f_j\}$ (первый критерий Бухбергера);
2) если существует тройка многочленов f_i, f_j, f_k, такая что

$$\mathrm{HM}(f_k) \mid \mathrm{HOK}\big(\mathrm{HM}(f_i), \mathrm{HM}(f_j)\big)$$

и пары (f_i, f_k) и (f_j, f_k) уже рассмотрены, то пару (f_i, f_j) можно не рассматривать (второй критерий Бухбергера).

Эти два критерия существенно уменьшают количество рассматриваемых критических пар и на порядки ускоряют алгоритм.

Каждый нетривиальный идеал при фиксированном упорядочении обладает бесконечным множеством базисов Грёбнера. Однако можно ввести понятие минимального и редуцированного базиса Грёбнера. Редуцированный базис уже однозначно определяется идеалом и мономиальным упорядочением. Его можно получить из любого базиса Грёбнера с помощью процесса авторедукции (замены слагаемых на эквивалентные им по модулю идеала выражения с меньшими мономами с помощью вычитания многочленов с подходящими старшими мономами) и последующей нормировкой старших коэффициентов.

Основным недостатком метода базисов Грёбнера является высокая сложность вычислений. Поэтому важной задачей является создание быстрых и эффективных программных библиотек для построения базисов Грёбнера. Повысить эффективность программ можно как за счёт оптимизации самих алгоритмов, так и благодаря их распараллеливанию. Алгоритмы могут работать с разной эффективностью на различных входных данных, однако существует общепризнанный набор стандартных тестовых примеров* (как правило, возникших из конкретных прикладных задач), на которых традиционно проверяется скорость работы таких алгоритмов.

В данной статье описывается параллельная реализация алгоритма F4 для вычисления базисов Грёбнера в кольце многочленов над простым конечным полем с помощью матричных вычислений.

3. Алгоритм F4

Алгоритм F4 был предложен французским математиком Ж.-Ш. Фожером в 1999 году [11]. Основная идея алгоритма F4 — заменить шаг редукции многочлена по множеству шагом редукции *множества* многочленов по множеству. Для этого вычисление очередных результатов редукций S-полиномов сводится к линейно-алгебраической задаче — приведению большой разреженной матрицы к ступенчатому виду. Строки этой матрицы соответствуют многочленам редуцируемого множества X и того множества, по которому производится редукция Y (они собираются вместе). Столбцы матрицы соответствуют мономам, которые встречаются в слагаемых многочленов из $X \cup Y$. Элементами матрицы являются коэффициенты соответствующих многочленов. Поскольку операции над строками данной матрицы соответствуют линейным комбинациям исходных многочленов с коэффициентами в основном поле (нельзя брать комбинации с полиномиальными коэффициентами), то для моделирования полиномиальной редукции приходится использовать так называемый *препроцессинг*. Эта операция состоит в добавлении к множеству $X \cup Y$ многочленов этого же множества, домноженных на подходящие мономы (если старший моном такого домноженного многочлена потенциально может быть использован при редукции). Эта процедура выполняется сравнительно быстро. Гораздо большее время занимает приведение матрицы к ступенчатому виду (по которому можно восстановить результаты редукции), так как размеры матрицы могут быть колоссально большими из-за добавления «домноженных» многочленов. В [11] сообщается, что при вычислении базисов Грёбнера некоторых идеалов размеры матрицы достигали 750000×750000. Поэтому эффективность алгоритма F4 во многом зависит от того, какими методами линейной алгебры матрица приводится к ступенчатому виду. Эти методы подробно рассматриваются ниже.

*См., например, `http://www.math.uic.edu/~jan/demo.html`.

Алгоритм F4 предполагает некоторый произвол в выборе редуцируемого за один раз на каждом шаге множества S-полиномов. Так, если на каждом шаге выбирается один S-полином, то он повторяет классический алгоритм Бухбергера. Другая крайность — когда на очередном шаге редуцируется множество всех имеющихся S-полиномов. Это тоже не очень эффективно из-за больших размеров матриц. Автор алгоритма Ж.-Ш. Фожер предложил *нормальную стратегию* выбора S-полиномов для редукции, согласно которой выбираются S-полиномы с наименьшей степенью левых и правых частей. Она даёт хорошие эмпирические результаты для упорядочения DegRevLex, и её выбор является естественным для однородных идеалов.

В алгоритм можно внести несколько естественных усовершенствований. Как и в классическом алгоритме вычисления базиса Грёбнера, можно применять критерии Бухбергера (метод Гебауэра—Мёллера [8, 13]) для отсеивания заведомо ненужных S-полиномов.

Псевдокод алгоритма F4

Алгоритм F4, строящий по множеству многочленов `givenPolynomials` его базис Грёбнера `result`, имеет следующий вид:

```
(basis, sPairs) ← Update(∅, ∅, givenPolynomials)
while sPairs ≠ ∅ do
    selectedPairs ← SelectPairs(sPairs)
    sPairs ← sPairs \ selectedPairs
    newElements ← Reduce(selectedPairs, basis)
    (basis, sPairs) ← Update(basis, sPairs, newElements)
end while
result ← Autoreduce(basis)
```

Функция

`Update: (basis, sPairs, newElements)` → `(newBasis, newSPairs)`

строит на основании текущего промежуточного базиса `basis`, множества нерассмотренных S-пар `sPairs` и множества `newElements` «новых» (не рассматривавшихся ранее) элементов идеала новый промежуточный базис `newBasis` и множество S-пар для рассмотрения (`newSPairs`) с учётом критериев Бухбергера.

Функция `Reduce`

Наиболее трудоёмкой частью алгоритма F4 с вычислительной точки зрения является процедура редукции набора S-пар `selectedSPairs` по множеству многочленов `basis`. Про множество редукторов при этом дополнительно известно, что оно является топ-авторедуцированным (т. е. никакие два элемента

множества старших мономов многочленов из `basis` не делятся друг на друга) и что S-пары порождены тем же множеством `basis`, по которому происходит редукция.

Сформулируем спецификацию функции `Reduce` на алгебраическом языке. Пусть $M =$ `selectedSPairs` (можно считать, что это набор левых и правых частей S-полиномов) и $B =$ `basis`. Положим $Q = \max\limits_{f \in M} \mathrm{HM}\, f$. Рассмотрим множество

$$\bar{B} = \{mb \mid m \in \mathbb{M},\ m \preccurlyeq Q,\ b \in B\},$$

представляющее собой базисные многочлены, домноженные на мономы. Для каждого старшего монома элемента $\bar{B}$ выберем каким-либо способом одного представителя из $\bar{B}$ с таким же старшим мономом и составим из этих представителей множество редукторов R. Таким образом, $\mathrm{HM}\, R = \mathrm{HM}\, \bar{B}$ и старшие мономы различных элементов R не совпадают. Далее в векторном пространстве $\langle M \cup R \rangle \subset F[x_1, \ldots, x_n]$ выберем базис $\{g_i\}$ с условием, что все старшие мономы $\mathrm{HM}\, g_i$ различны. Выделим подмножество базисных элементов

$$G = \{g_i \mid \mathrm{HM}\, g_i \text{ не делится ни на какой моном из множества } \mathrm{HM}(B \cup \{g_j\}_{j \neq i})\}.$$

Это и есть новые многочлены, которые следует добавить на очередном шаге к промежуточному базису.

Итак, в алгоритме F4 в качестве редукторов используются только элементы заранее подготовленного множества $R = \{mb\}$, $b \in B$, все старшие мономы которого различны. После начала преобразований никакие элементы в него уже не добавляются. Этот подход и называется препроцессингом. Он делает информацию о мультипликативных свойствах мономов ненужной в основном цикле преобразований. Перед началом преобразований можно перенумеровать все встречающиеся мономы по убыванию (чтобы старшим мономам соответствовали первые столбцы матрицы). После получения результата мультипликативные свойства вновь становятся нужны, поэтому перед выдачей ответа нужно провести обратную замену номеров на мономы. Выигрыш от замены мономов на номера состоит исключительно в повышении эффективности — как с точки зрения памяти, так и с точки зрения производительности.

В итоге задача свелась к редукции матрицы M (соответствующей S-парам) по матрице R (соответствующей редукторам). Благодаря тому что по построению матрица R уже имеет ступенчатый вид, задача эквивалентна нахождению ступенчатого вида для объединённой матрицы, составленной из строк M и R. Из полученного результата нужно выкинуть нулевые строки, а также строки, у которых старшие мономы совпадают со старшими мономами R. Заметим, что благодаря предварительному препроцессингу в окончательном результате вычислений не могут появиться строки, старший моном которых делится на какой-то старший моном редуктора из R, но не равен никакому старшему моному редуктора.

Возможности варьирования результата

Вообще говоря, искомое множество G не единственно. Это объясняется тем, что, хотя редуктор mb с заданным старшим мономом единствен в течение всей редукции, он может быть изначально выбран разными способами. Рассмотрим пример, где M состоит из единственного S-полинома, образованного двумя первыми многочленами из B (порядок на мономах лексикографический, $a > b > c > d > e$):

$$B = \{a^2 + 1,\ ab + de,\ ad - b,\ ae - 1\}, \quad M = \{ade - b\}.$$

В качестве редуктора можно взять как $e(ad-b)$, так и $d(ae-1)$. В первом случае получится $be - b$, во втором $-b + d$. В обоих случаях полученные многочлены не редуцируются дальше по B и, следовательно, уже образуют результирующее множество G. Из этого примера видно, что набор старших мономов G неоднозначен. Более того, если бы в B был ещё и многочлен $be-b$, то в первом случае произошла бы редукция до нуля, в то время как во втором редукции до нуля не произошло бы. Таким образом, неоднозначно определено даже то, есть ли в G ненулевые элементы.

Постановка задачи оставляет свободными четыре параметра:

1) способ формирования множества редуцируемых многочленов M по набору S-пар;
2) способ выбора множества редукторов R;
3) порядок, в котором происходят разрешённые преобразования;
4) проведение дополнительных преобразований (например, авторедукции).

Рассмотрим их подробнее.

Формирование множества редуцируемых многочленов M

Рассмотрим два способа формирования набора строк матрицы M из множества S-пар:

1) добавление отдельно левых и правых частей S-пары;
2) добавление отдельных S-полиномов (обозначим их множество через M_0).

Заметим, что в первом случае в матрице будет в два раза больше строк, что может несколько увеличить объём вычислений при проведении редукций не по ведущему столбцу. Левые и правые части S-пары имеют такой же вид, как и редукторы (они представляют собой исходные базисные многочлены, домноженные на мономы). Поэтому в первом случае при препроцессинге их не следует учитывать. Заметим также, что, в отличие от второго способа, в первом случае операции сложения и вычитания достаточно реализовать только для строк матриц, но не для многочленов. Кроме того, в первом случае могут появиться дополнительные многочлены в результирующем множестве G.

Эффективность этих способов зависит от количества S-пар с одинаковыми старшими мономами: если их мало, то выгоднее пользоваться вторым методом,

а если много — первым. Количество таких пар зависит от стратегии выбора подмножества S-пар на каждой итерации цикла F4 и от конкретной задачи. На практике немного более эффективным оказывается второй способ.

В общем случае M обязано обладать следующими двумя свойствами:

1) все элементы M_0 выражаются через элементы M с помощью линейных преобразований;
2) все многочлены из M принадлежат идеалу, порождённому B.

Первое условие необходимо для того, чтобы G содержало те многочлены (с точностью до линейных преобразований), которые появились бы, если бы вместо M рассматривалось M_0. Второе — для того, чтобы все полученные после редукции многочлены также принадлежали идеалу, порождённому B.

Способ выбора множества редукторов R

Можно использовать два подхода к выбору R. Первый подход — добавлять в R новые элементы (если подходящие многочлены существуют) всякий раз, когда в нём не находится редуктора с заданным старшим мономом (т. е. добавлять редукторы по мере необходимости). Так работает классический алгоритм Бухбергера. Второй подход — заранее вычислять и помещать в R множество всех многочленов, которые могут потенциально пригодиться при редукции, т. е. проводить препроцессинг. Этот подход применяется в классическом алгоритме F4. В обоих случаях, если в R уже есть многочлен с рассматриваемым старшим мономом, его и следует использовать для редукции.

Фиксирование множества редукторов R позволяет практически полностью избавиться от неоднозначности. В самом деле, для фиксированных множеств M и R и количество многочленов, полученных в результате редукции, и их старшие мономы определены однозначно. В свою очередь, при фиксированном R выбором M можно пытаться получить в результате редукции некоторые дополнительные многочлены по сравнению с теми, что получились бы при взятии M_0. Поскольку над элементами множества M разрешены линейные преобразования, то существенную роль с точки зрения возможного результата играет не само M, а лишь его линейная оболочка. Но на объём вычислений может влиять и конкретный выбранный набор векторов, порождающий векторное пространство $\langle M \rangle$.

Порядок преобразований и авторедукция

При заданных M и R на набор старших мономов результата уже ничего не влияет, и все преобразования имеют вид $a \leftarrow a + mb$, где a — многочлен из преобразуемого множества, m — элемент поля, а b может быть как из преобразуемого множества M, так и из R. Хотя результат от порядка проведения этих преобразований существенно не зависит, последовательность выполнения операций может существенно влиять на объём вычислений из-за большой разреженности.

Единственное, что всё ещё неоднозначно определено в результате, — это мономы многочлена, следующие после ведущего (мономы «хвоста»). Могут существовать нестаршие мономы, совпадающие с ведущими элементами какого-то другого многочлена. Они могут быть редуцированы (обнулён коэффициент) с помощью вычитания этого многочлена. Такое преобразование называется *авторедукцией*. В зависимости от порядка проводимых операций часть авторедукций может быть уже неявно проведена (отсюда и возникает несущественная неоднозначность результата). Можно дополнительно применить авторедукции к множеству, которое уже удовлетворяет требованию на результат о неделимости старших мономов G на старшие мономы B. Если провести все возможные авторедукции элементов M по элементам M и R (*полную авторедукцию*), то результат будет удовлетворять усиленному требованию: теперь *все* его мономы не будут делиться ни на старшие мономы B, ни на старшие мономы результата. Такой *авторедуцированный вид* будет строго единствен при фиксированных множествах M и R. Смысл проведения авторедукций состоит в том, что, несмотря на увеличение объёма вычислений на данном шаге, другой вид младших мономов результата может существенно снизить объём вычислений на последующих шагах F4.

4. Идеи распараллеливания и реализация

Непосредственно распараллеливание применяется к задаче нахождения ступенчатого вида одной матрицы, поскольку она занимает основную часть времени работы алгоритма. Сначала все строки матрицы равномерно делятся на блоки и рассылаются по разным процессорам. Далее применяется блочный метод Гаусса, работа в котором идёт в терминах блоков строк. Операции редукции блоков, находящихся на разных процессорах, выполняются параллельно.

Поскольку в исходных многочленах присутствовали далеко не все мономы, матрица $M \cup R$ обладает большой разреженностью: большинство её элементов равно 0. На практике заполнение составляет примерно 0,3—3 %, хотя в отдельных случаях оно падает до 0,1 % или даже увеличивается до 20 %. Для эффективного использования такой большой разреженности требуется учитывать её при хранении строк и в реализации метода Гаусса. Представление строк реализовано по аналогии с многочленами с заменой монома на номер столбца: строка является массивом, содержащим только ненулевые коэффициенты и соответствующие им номера столбцов. Элементы массива упорядочены по возрастанию номеров столбцов так, чтобы элемент с наименьшим номером (ведущий) оказался в начале. В этом случае доступ к нему будет наиболее эффективен, что важно, поскольку ведущий элемент строк при работе алгоритма редукции просматривается чаще остальных.

Равномерность загрузки процессоров определяется двумя основными факторами: близким числом строк на разных процессорах (оно может меняться в процессе работы за счёт обнуления некоторых строк, так как изначально матрица,

как правило, является вырожденной) и равномерным распределением строк по процессорам с точки зрения номеров ведущих элементов (это особенность разреженного случая). Попытки применения эвристик для балансировки в процессе работы алгоритма показали, что подобные методы хотя и дают большую равномерность распределения, однако временные затраты на синхронизацию для выявления необходимой перебалансировки приводят лишь к увеличению общего времени. Поэтому используется алгоритм, при котором все разделения строк на блоки и по процессорам рассчитываются единожды до начала редукции. Все последующие шаги проводятся строго в соответствии с этим разделением, а в процессе вычислений перебалансировки между процессорами не проводятся. Для того чтобы при этом не происходило потери равномерности в процессе работы, применяется следующее распределение строк матрицы: множество всех строк матрицы, отсортированных по номеру столбца ведущего элемента, разделяется на равные по числу строк блоки. Эти блоки рассылаются по процессорам с циклическим перебором номера процессора-получателя, т. е. процессор с номером k (при n процессорах всего) получает блоки, номера которых равны k по модулю n. Разделение должно быть таким, чтобы в каждом блоке оказалось достаточно много строк (и была получена эффективность от блочных операций), а также, одновременно с этим, самих блоков должно быть достаточно много (чтобы время обработки одного шага на процессорах, содержащих на один блок больше, чем другие, отличалось незначительно и доля времени простоя, вызванного разным числом блоков на процессорах, была невелика). Именно этим объясняется то, что хорошие коэффициенты распараллеливания достигаются начиная с матриц, содержащих несколько тысяч строк.

После стадии рассылки все вычисления ведутся уже строго в терминах этих блоков. Используется классическая блочная запись метода Гаусса с тем отличием, что разбиение на блоки зафиксировано изначально на стадии рассылки и не меняется в процессе работы — перераспределения строк не происходит не только между процессорами, но и между блоками на одном процессоре. В противном случае в пределах одного блока могли бы возникнуть строки с сильно различающимися ведущими столбцами, что привело бы к потере равномерности после редукции по нему. То есть если какие-то строки блока обнулились в процессе редукции, то они выкидываются, а число строк в блоке уменьшается (вплоть до 0). Заимствования из других блоков того же процессора никогда не происходит. В остальном алгоритм соответствует блочно-параллельному методу Гаусса: на каждом шаге выполняются одновременно операции редукции блоков, находящихся на разных процессорах по одному и тому же заранее выбранному и переданному на все процессоры авторедуцированному блоку.

Псевдокод

Приведём в самом общем виде псевдокод реализованного блочно-параллельного алгоритма нахождения ступенчатого вида разреженной матрицы. В алгоритме используются следующие переменные:

— `Finished` — множество уже отредуцированных строк;

— `Processing` — множество блоков строк, находящихся в обработке;

— `Current_block` — блок строк, по которому производится редукция на данном шаге. Это множество совпадает на всех процессорах после вызова функции `send_set_to_all_from_processor`;

— `Current_processor` — процессор, содержащий блок, по которому ведётся редукция на данной итерации. Значение совпадает на всех процессорах, поскольку переменная одинаково меняется на каждой итерации цикла.

Также для взаимодействия используются следующие коммуникационные функции:

— `this_processor()` — возвращает номер процессора, на котором она вызвана (т. е. «этот» процессор), в множестве всех процессоров, используемых в алгоритме (процессоры нумеруются последовательно начиная с 0);

— `total_processors()` — суммарное число процессоров в системе;

— `is_empty_all_processors(set)` — булева синхронная функция, которая вызывается одновременно на всех процессорах. Её возвращаемое значение также везде одинаково: оно равно истине тогда и только тогда, когда все множества, переданные ей в качестве аргумента, на всех процессорах пусты. Если хотя бы на одном процессоре `set` не пусто, то функция вернёт ложь;

— `send_set_to_all_from_processor(set, processor_sender)` — синхронная процедура, которая, будучи вызвана одновременно на всех процессорах с одинаковым аргументом `processor_sender`, запишет в переменную `set` значение, содержащееся в этой переменной на `processor_sender`. То есть после вызова этой процедуры значение переменной `set` на всех процессорах будет одинаково;

— `unite_all_processors(set)` — синхронная функция, возвращающая на каждом из процессоров множество, равное объединению аргументов `set` по всем процессорам.

Прочие процедуры, не являющиеся коммуникационными (выполняются на каждом из процессоров независимо):

— `select_row_blocks_for_processor(matrix, processor)` — возвращает подмножество блоков строк `matrix`, попадающих на `processor` при начальном разделении;

— `take_row_block(set_of_row_blocks)` — возвращает блок строк, содержащийся в `set_of_row_blocks`, убирая его оттуда;

— `full_reduce(matrix)` — производит приведение матрицы к сильно ступенчатому виду (последовательная версия);

— `reduce_matrix_by_matrix(matrix_to_reduce, reductor)` — производит редукцию всех строк матрицы `matrix_to_reduce` по всем строкам `reductor` (последовательная версия, учитывающая разреженность);

— `add_row_block_to_set(block, set_of_row_blocks)` — добавляет блок строк к заданному множеству.

Следующий код параллельно исполняется на всех процессорах с одним и тем же аргументом `Matrix`. Согласование поведения происходит за счёт вызываемых коммуникационных процедур.

```
function Reduce(Matrix) {
    Processing = select_row_blocks_for_processor(Matrix,
        this_processor())
    Finished = ∅
    Current_processor = 0
    while not is_empty_all_processors(Processing) {
        if this_processor() == Current_processor then {
            Current_block = take_row_block(Processing)
            full_reduce(Current_block)
        }
        send_set_to_all_from_processor(Current_block,
            Current_processor)
        reduce_matrix_by_matrix(Processing, Current_block)
        if this_processor() == Current_processor then {
            add_row_block_to_set(Current_block, Finished)
        }
        Current_processor =
            (Current_processor + 1) mod (total_processors())
    }
    return unite_all_processors(Finished)
}
```

Авторами были созданы библиотеки программных средств F4mpi и F_2F4mpi (написаны на языке C++ с использованием интерфейса MPI). Данные библиотеки позволяют производить параллельные матричные вычисления базисов Грёбнера при различных порядках на мономах над конечными полями и имеют дополнительные параметры настройки, такие как указание стратегии выбора редуцирующих строк, явное задание размеров блоков при редукции матриц и набор параметров, отвечающих за настройку MPI-подсистемы. Библиотеки были протестированы на многопроцессорных кластерах с распределённой памятью.

В качестве тестовых примеров использовались стандартные системы из набора PoSSo и ряд систем, возникающих в прикладных задачах. Среди этих задач — исследование систем уравнений, описывающих распределение нагрузки в электрических сетях; исследование уравнений из алгоритма шифрования с открытым ключом HFE; описание бифуркационной диаграммы интегрируемых гамильтоновых систем и т. д. Некоторые результаты тестирования приведены в таблицах. В каждой ячейке указано общее время работы программы (в секундах), время приведения матриц к ступенчатому виду и общее ускорение относительно запуска на одном процессоре.

Пример	1 процессор	2 процессора	3 процессора	4 процессора
Cyclic8	222/199	136/113	96/73	81/58
(поле F_{31013})		1,63 раз	2,30 раз	2,72 раз
Eco12	366/336	207/181	116/90	93/67
(поле F_2)		1,76 раз	3,14 раз	3,92 раз
Hf855	117/106	63/53	31/20	27/16
(поле F_{31013})		1,83 раз	3,74 раз	4,26 раз
Noon8	2185/2071	1369/1296	649/587	387/325
(поле F_{31013})		1,60 раз	3,36 раз	5,64 раз

Пример	1 проц.	12 проц.	16 проц.	24 проц.	48 проц.
Eco12	1350/1249	208/147	175/116	154/96	137/80
(поле F_{31013})		6,48 раз	7,71 раз	8,72 раз	9,83 раз
Katsura10	516/492	92/67	79/55	72/48	69/45
(поле F_{31013})		5,59 раз	6,46 раз	7,07 раз	7,42 раз
Noon8	3299/3247	357/306	255/204	204/152	153/101
(поле F_{31013})		9,22 раз	12,90 раз	16,13 раз	21,44 раз

5. Теоретический анализ эффективности авторедукции

Строки, добавленные в матрицу R в результате препроцессинга, в свою очередь можно разделить на две группы:

A: строки, у которых старший моном встречается среди мономов матрицы M (полученной из S-пар);

B: строки, старший моном которых встречается в R, но не в M.

Отличия между этими группами проявляются при проведении авторедукции строк R.

Для оценки эффективности временных затрат на авторедукцию положим, что прибавление к строке длины m строки длины n требует $r(m,n)$ операций. Для простейшей реализации склеиванием двух упорядоченных массивов $r(m,n) = O(m+n)$. Также для большинства реализаций (тех, где можно разбить строку на две отдельные части за константное число действий) будет выполнено соотношение $r(m, n_1+n_2) \leqslant r(m,n_1) + r(m,n_2)$, потому что длинную строку можно прибавлять по частям. Предположим также монотонность по обоим аргументам: при $N \geqslant n$, $M \geqslant m$ будет выполнено $r(m,N) \geqslant r(m,n)$ и $r(M,n) \geqslant r(m,n)$ (вообще, можно считать функцию r симметричной). Эти соотношения будут использоваться ниже при оценке времени работы при различном порядке операций. Через $\operatorname{len}(x)$ обозначим число ненулевых элементов в строке x.

Рассмотрим редукцию $x \underset{b}{\to} y$ строки x с ненулевым элементом в столбце c по строке $b \in B$ с ведущим элементом в столбце c до строки y. Тогда по построению матрицы R ведущие элементы у строк x и y совпадают. Пусть $U \subset M$ — некоторый набор строк из верхней части матрицы, которые нужно отредуцировать по x (и по y). Заметим, что при редукции $u \underset{y}{\to} u'$, где $u \in U$, в столбце c появляется ненулевой элемент, а при редукции $u \underset{x}{\to} u'$ не появляется. Число операций при редукции всех строк из U по x и b при проведении предварительной редукции $x \underset{b}{\to} y$ равно

$$C_1 = r\big(\operatorname{len}(x), \operatorname{len}(b)\big) + \sum_{u \in U} r\big(\operatorname{len}(u), \operatorname{len}(y)\big).$$

Без проведения предварительной редукции $x \underset{b}{\to} y$ получаем следующий результат:

$$C_2 = \sum_{u \in U} \Big(r\big(\operatorname{len}(u), \operatorname{len}(x)\big) + r\big(\operatorname{len}(u'), \operatorname{len}(b)\big)\Big),$$

где $u \underset{x}{\to} u'$. Сравним эти выражения. Одиночным слагаемым $r\big(\operatorname{len}(x), \operatorname{len}(b)\big)$ можно пренебречь. Для оставшихся слагаемых при условии $\operatorname{len}(u') \geqslant \operatorname{len}(u)$ будет выполнена следующая цепочка неравенств:

$$r\big(\operatorname{len}(u), \operatorname{len}(x)\big) + r\big(\operatorname{len}(u'), \operatorname{len}(b)\big) \geqslant r\big(\operatorname{len}(u), \operatorname{len}(x)\big) + r\big(\operatorname{len}(u), \operatorname{len}(b)\big) \geqslant$$
$$\geqslant r\big(\operatorname{len}(u), \operatorname{len}(x) + \operatorname{len}(b)\big) \geqslant r\big(\operatorname{len}(u), \operatorname{len}(y)\big).$$

Условие на длины u' и u будет выполнено практически во всех случаях из-за разреженности матрицы: при добавлении x к u вероятнее добавление новых ненулевых элементов, чем сокращение старых. Также соотношение $\operatorname{len}(x) + \operatorname{len}(b) \geqslant \operatorname{len}(y)$ будет всегда выполнено, поскольку в y, являющемся линейной комбинацией x и b, не может быть больше ненулевых элементов, чем в x и b, вместе взятых. Таким образом, от редукции строки x по строке $b \in B$ всегда имеется положительный эффект с точки зрения производительности.

Рассмотрим теперь другой случай, когда вместо строки $b \in B$ используется строка $a \in A$ с ведущим элементом в столбце c. Пусть $x \underset{a}{\to} y$. Снова по

построению множества R получаем, что c не является ведущим столбцом строки x. Покажем, что если большое количество строк $u \in U$ содержат в столбце c ненулевой элемент, то предварительная редукция $x \underset{a}{\to} y$ может привести к увеличению объёма вычислений. Здесь, в отличие от предыдущей ситуации, после редукции вида $u \underset{x}{\to} u'$ в столбце c могут по-прежнему остаться ненулевые элементы. Это произойдёт независимо от того, была ли строка x редуцирована по a или нет, так как изначально значение, содержащееся в столбце c строки u, в некотором смысле «случайно» и вероятность его обнулить, редуцировав по x, мала. То есть такую строку u придётся в любом случае редуцировать вначале по x (или y), а потом по a. Конечный результат редукции не будет зависеть от того, была ли проведена авторедукция x до y. Для оценки зависимости объёма вычислений в случаях проведения и непроведения предварительной редукции x до y от числа ненулевых элементов матрицы M в столбце c выпишем приближённое число операций для обоих случаев. Пусть $u \underset{x}{\to} u'$ и $u \underset{y}{\to} u''$. Оценка на число действий при проведении предварительной редукции $x \underset{a}{\to} y$ такова:

$$E_1 = r\big(\operatorname{len}(x), \operatorname{len}(a)\big) + \\ + \sum_{u \in K_1} \Big(r\big(\operatorname{len}(u), \operatorname{len}(y)\big) + r\big(\operatorname{len}(u''), \operatorname{len}(a)\big)\Big) + \sum_{K_2} r\big(\operatorname{len}(u), \operatorname{len}(y)\big).$$

Без проведения редукции получаем

$$E_2 = \sum_{u \in L_1} \Big(r\big(\operatorname{len}(u), \operatorname{len}(x)\big) + r\big(\operatorname{len}(u'), \operatorname{len}(a)\big)\Big) + \sum_{u \in L_2} r\big(\operatorname{len}(u), \operatorname{len}(x)\big).$$

Здесь

K_1 — множество строк M с ненулевым элементом в столбце c, которые редуцируются по x;

K_2 — множество строк M, которые редуцируются по x, но содержат 0 в столбце c;

L_1 — множество строк M, которые после редукции по x содержат ненулевой элемент в столбце c;

L_2 — множество строк M, которые после редукции по x содержат 0 в столбце c.

Для них выполнены следующие соотношения:

$$K_1 \cup K_2 = L_1 \cup L_2 = U,$$
$$K_1 \cap K_2 = L_1 \cap L_2 = \varnothing,$$
$$L_1 \supset K_2, \quad L_2 \subset K_1.$$

При вычислении над полями вычетов с большой характеристикой множество L_2 будет практически пусто, поскольку «случайное» появление нуля маловероятно. Однако при вычислениях по модулю 2 проявляется противоположный эффект: $L_1 = K_2$, $L_2 = K_1$, потому что в этом случае редукция по x всегда заменяет

значение в столбце c на противоположное. При большом числе строк в множестве $U = K_1 \cup K_2$ первым слагаемым, отвечающим за редукцию $x \underset{a}{\to} y$, можно пренебречь. Далее, $\operatorname{len}(y)$ превосходит $\operatorname{len}(x)$, поэтому для $i = 1, 2$ затраты на один элемент K_i больше, чем на элемент L_i. Но поскольку в обоих случаях затраты для элемента первого множества превышают затраты для второго, решающую роль играет распределение элементов по множествам. Рассмотрим наиболее простой случай $r(a, b) = a + b$ и предположим, что разреженность достаточно велика для того, чтобы у любых двух строк было пренебрежимо мало общих ненулевых столбцов. Тогда можно положить

$$\begin{gathered}\operatorname{len}(y) = \operatorname{len}(x) + \operatorname{len}(a),\\ \operatorname{len}(u') = \operatorname{len}(u) + \operatorname{len}(x),\\ \operatorname{len}(u'') = \operatorname{len}(u) + \operatorname{len}(x) + \operatorname{len}(a).\end{gathered}$$

Затраты при проведении предварительной редукции $x \underset{a}{\to} y$ будут равны

$$E_1 = \sum_{u \in K_1} \big(2\operatorname{len}(u) + 3\operatorname{len}(a) + 2\operatorname{len}(x)\big) + \sum_{u \in K_2} \big(\operatorname{len}(u) + \operatorname{len}(a) + \operatorname{len}(x)\big).$$

Без проведения предварительной редукции получаем следующую оценку на затраты:

$$E_2 = \sum_{u \in L_1} \big(2\operatorname{len}(u) + \operatorname{len}(a) + 2\operatorname{len}(x)\big) + \sum_{u \in L_2} \big(\operatorname{len}(u) + \operatorname{len}(x)\big).$$

Их разность, с учётом того что $L_1 \supset K_2$ и $L_2 \subset K_1$, даёт

$$\begin{aligned}E_1 - E_2 = \sum_{u \in K_2} \big(-\operatorname{len}(u) - \operatorname{len}(x)\big) +\\ + \sum_{u \in L_1 \cap K_1} 2\operatorname{len}(a) + \sum_{u \in L_2} \big(\operatorname{len}(u) + 3\operatorname{len}(a) + \operatorname{len}(x)\big).\end{aligned}$$

В случае большой характеристики поля последним слагаемым можно пренебречь (так как L_2 почти пусто), и редукцию $x \underset{a}{\to} y$ стоит проводить, только если число элементов K_2 «заметно» на фоне K_1. Если же K_2 значительно меньше K_1, то разность будет положительна, и редукцию проводить не следует. В случае характеристики 2 обнуляется второе слагаемое разности. Условие получается, по сути, то же самое: проводить редукцию, только если K_2 больше K_1 ($= L_2$ в случае модуля 2), но здесь K_1 входит с бóльшим весовым коэффициентом, чем в предыдущем случае. В обоих случаях конкретные числовые коэффициенты, на основе которых следует принимать решение о необходимости редукции x по a, нужно выбирать в зависимости от средней длины u, $\operatorname{len}(a)$, $\operatorname{len}(x)$ и мощности K_2.

Для строк из K_2 было бы эффективней использовать редуцированную строку x в любом случае. Но для реализации подобного метода, где часть строк редуцируются непосредственно x, а некоторые — строками, полученными из x, придётся хранить и вычислять для каждой исходной строки x много вариантов y

(результатов частичной редукции по различным поднаборам строк A). Их построение и выбор нужного варианта займёт существенно больше времени, чем будет выиграно за счёт чуть более эффективного проведения авторедукций. Из этого следует, что для принятия точного решения о проведении авторедукции нужно заключение о том, окажется ли множество K_2 больше K_1. Подобное заключение можно дать лишь при некотором дополнительном предположении о структуре матриц. Например, оно выполнено при равномерном распределении ненулевых элементов для матриц с малой заполненностью (потому что строк, у которых ненулевые элементы будут одновременно как в ведущем столбце строки x, так и в столбце c, окажется немного). В общем же случае ничего подобного утверждать нельзя, и вопрос эффективности проведения авторедукции матрицы существенно зависит от каждой конкретной матрицы. На практике несколько более эффективным обычно оказывается проведение предварительной авторедукции.

6. Специфика вычисления базисов Грёбнера над полем F_2

Работа с многочленами с коэффициентами из поля F_2 имеет свои особенности, связанные со структурами данных. Имеется несколько независимых исследований, учитывающих эти особенности (система PolyBoRi* (Polynomial over Boolean Rings), а также работа [1]).

Во-первых, мы можем опустить коэффициенты при мономах в многочленах. Благодаря этому можно существенно оптимизировать хранение многочленов и операции над ними.

Один из подходов — так называемая плотная запись многочленов. Она заключается в следующем: занумеруем каким-нибудь способом все возможные мономы, которые можно составить из наших переменных. Желательно, чтобы нумерация была согласована с порядком на мономах, который мы используем, т. е. если $p \prec q$, где p и q — мономы, то $\mathrm{Num}(p) < \mathrm{Num}(q)$, где Num — функция, ставящая в соответствие моному натуральное число. Также желательно, чтобы функция Num была «непрерывной»**, т. е. если существуют p и q, такие что $\mathrm{Num}(p) = k - 1$ и $\mathrm{Num}(q) = k + 1$, то найдётся r, для которого $\mathrm{Num}(r) = k$. Тогда будем хранить многочлен как битовую маску, т. е. как набор битов, причём в k-м по счёту бите стоит 1, если моном, которому соответствует число k, присутствует в многочлене, и 0, если он там отсутствует. При таком подходе в одной обычной переменной целого типа `int` можно будет хранить сразу 32 ячейки, определяющие, есть ли соответствующий моном в многочлене. Взяв достаточное количество таких переменных и считая, что они расположены «подряд», можно хранить полностью всю информацию о многочлене. При этом

*`http://polybori.sourceforge.net`.

**Это возможно сделать только при градуированных порядках (например, DegRevLex) или если есть оценка на максимально возможную степень вхождения переменной в моном.

сложение многочленов соответствует побитовой операции `xor`. Если в качестве функции, ставящей в соответствие мономам числа, взять функцию, возвращающую число, r-ичная запись которого представляет собой последовательность степеней вхождения всех имеющихся переменных в моном (где r — максимально возможная степень вхождения переменной в моном, которую нужно оценить заранее), то умножение монома на моном будет соответствовать сложению значений функции и умножение многочлена на моном будет соответствовать сдвигу всех цифр на одно и то же количество позиций, что также выполняется простым алгоритмом (такая нумерация соответствует порядку Lex). При таком способе мы получаем плотную запись многочлена, т. е. храним информацию и о тех мономах, которые в нём не содержатся, однако при этом тратим в 32 раза меньше памяти и вычислительных операций. При таком способе хранения в алгоритме F4 вообще не нужно переводить многочлены в матрицу, а потом обратно. Итак, если плотность матрицы больше, чем $1/32$, мы получаем здесь выигрыш.

Больший выигрыш даёт другая идея. Обычно, находя базис Грёбнера над конечным полем, мы предполагаем, что переменные, входящие в многочлены, принадлежат основному полю, а это означает, что они удовлетворяют уравнению поля $x^p - x = 0$, где p — характеристика. Для поля F_2 уравнения имеют вид $x^2 - x = 0$. В этом случае следует искать базис Грёбнера не для исходного набора многочленов, а для набора, к которому добавлены уравнения поля, записанные для каждой из переменных. Оказывается, что в этом случае можно сильно упростить действия над многочленами и мономами, фактически предполагая в них изначально уравнения поля, и при этом корректно получать базис Грёбнера для набора многочленов, используя немного видоизменённые алгоритмы Бухбергера или F4.

Итак, перейдём сразу к кольцу многочленов, отфакторизованному по идеалу $(x_1^2 - x_1, x_2^2 - x_2, \ldots, x_d^2 - x_d)$. В этом кольце останутся только мономы, у которых каждая переменная входит либо в степени 0, либо в степени 1. Каждому такому моному легко поставить в соответствие число, двоичная запись которого состоит из показателей степеней вхождения каждой из имеющихся переменных в моном. Если всего у нас имеется d переменных, то в двоичной записи числа, соответствующего каждому моному, будет содержаться ровно d цифр, возможно, при этом в записи будут содержаться лидирующие нули. В этом случае умножение мономов будет соответствовать операции `xor` над соответствующими им числами. Для исполнения алгоритма нам нужны следующие операции над мономами.

1. Создание монома по набору переменных. Для того чтобы добавить в моном k-ю переменную, необходимо вычислить $[k/32]$ — индекс элемента вектора, в котором находится эта переменная. Далее нужно выполнить операцию `xor` элемента вектора с числом $2^{k \pmod{32}}$. Эти операции реализованы как `(k >> 5)` и `(1 << (k & 31))` соответственно.
2. Копирование мономов: `a = b`.
3. Умножение мономов: `c[k] = a[k] | b[k]`.

4. Наименьшее общее кратное (совпадает с произведением).
5. Определение делимости одного монома на другой: для каждого индекса k векторов `(a[k] & b[k]) == b[k]`.
6. Деление монома на моном: `c[k] = a[k] ^ b[k]` (после проверки делимости).
7. Вычисление степени монома (соответствует вычислению размера битового множества). Этот размер равен сумме количества единичных битов в каждом из элементов вектора. В целях оптимизации для этой операции хранится специальная таблица из 65536 элементов, содержащая количество битов для каждого числа от 0 до $2^{16}-1$. Тогда вычисление количества битов в `unsigned int` выполняется с помощью двух обращений к таблице (для старшего и младшего байтов числа) и одного сложения.

Теперь опишем способ хранения полиномов. Будем считать, что порядок на мономах — это DegRevLex. Описываемый способ можно видоизменить и для использования других порядков. Так как в первую очередь мономы упорядочиваются по степени, то предлагается хранить многочлен в виде набора ячеек, в каждой из которых будут содержаться мономы одной и той же степени. Таким образом, всего нам может понадобиться не более $d+1$ ячейки для одного многочлена. Внутри ячеек мономы будем хранить в виде упорядоченного множества, в котором сравнение производится уже просто по порядку RevLex, так как степени мономов равны. Для многочленов нам нужно уметь выполнять две базовые операции:

1) умножение многочлена на моном. Для этого нужно просто пройтись по всем ячейкам, в каждой ячейке по всем мономам, домножить каждый из них на наш множитель, а затем положить в соответствующую ячейку результата, если там ещё нет такого монома, либо, наоборот, удалить из неё моном, если он там уже есть, так как два одинаковых монома сокращаются. Номер ячейки определяется степенью монома, которую мы умеем вычислять;
2) сложение двух многочленов. Для этого нужно уметь отдельно складывать их однородные части, лежащие в соответствующих друг другу ячейках. Ячейки представляют собой упорядоченные множества, которые мы можем обходить по возрастанию. Тогда сложение однородных частей сводится к нахождению симметрической разности между такими множествами, что можно сделать за один проход по этим множествам в силу их упорядоченности.

Заметим, что умножение на моном уже не сохраняет порядок мономов. Например, многочлен $x_1x_2+x_3$ после умножения на x_1x_2 превращается в $x_1x_2x_3+$ $+x_1x_2$, и мономы меняются местами. Вследствие этого изменяется сам ход алгоритма, независимо от того, базовый используемый алгоритм — это F4, алгоритм Бухбергера или другой алгоритм вычисления базисов Грёбнера.

Для того чтобы действия в отфакторизованном кольце приводили к правильному ответу, следует в начале выполнения алгоритма добавить к множеству

многочленов $f_1, f_2, \ldots, f_n$ все их кратные $x_i f_j$, такие что x_i входит в НМ(f_j). Эти многочлены являются S-полиномами пар $(f_j, x_i^2 - x_i)$, редуцированными относительно $\{x_i^2 - x_i\}_{i=1}^n$. Рассмотрим для доказательства два случая: когда старший моном f_j делится на x_i и когда не делится. Пусть m — старший моном f_j. Если m делится на x_i, то можно считать, что x_i входит в f_j в первой степени, так как иначе можно f_j редуцировать по $x_i^2 - x_i$. Тогда

$$S(f_j, x_i^2 - x_i) = x_i f_j - \frac{m}{x_i}(x_i^2 - x_i) \to x_i f_j.$$

Иначе

$$S(f_j, x_i^2 - x_i) = x_i f_j - m(x_i^2 - x_i) \to x_i f_j.$$

Соответственно, эти элементы добавить нужно. Далее, критерии Бухбергера применимы к новым многочленам. У каждого из многочленов в редуцируемом кольце есть прообраз в обычном кольце многочленов, однозначно определённый, так как все многочлены строятся как S-полиномы из уже имеющихся, а затем редуцируются по множеству многочленов вида $x_i^2 - x_i$. И хотя в нашем редуцированном кольце при домножении многочлена на моном порядок мономов в нём может измениться, а доказательства критериев используют свойства правильного порядка на мономах, откидываемые за счёт критериев пары все равно нужно откидывать. Действительно, операция умножения на моном в нашем кольце соответствует просто умножению на тот же моном в обычном кольце многочленов, а затем редукции. После редукции старшие мономы многочлена в обычном кольце многочленов и в нашем редуцированном совпадут по определению. Следовательно, если мы можем применять критерии Бухбергера к старшим мономам многочленов в обычном кольце многочленов, то можем применять их и для отбрасывания пар в редуцированном кольце, так как полученные S-полиномы будут редуцироваться в обычном кольце многочленов к нулю относительно имеющегося множества, а мы ищем именно базис Грёбнера для исходного кольца многочленов.

Добавлять такие домноженные многочлены нужно только в начале алгоритма. Действительно, если на каком-то шаге мы получили многочлен f, то

$$f = \sum_{i=1}^{n} p_i f_i,$$

где p_i — некоторые многочлены, а f_i — исходный набор образующих. Тогда

$$x_i f = \sum_{i=1}^{n} x_i p_i f_i = \sum_{i=1}^{n} p_i (x_i f_i),$$

а многочлены $x_i f_i$ уже были добавлены на первом шаге. Итак, если многочлен f даёт вклад в ответ, то он всё равно будет получен другим путём, так как $x_i x_i f_i = x_i f_i$.

С другой стороны, изменения порядка мономов при домножении многочлена на моном несколько осложняет операции с многочленами. Было рассмотрено

несколько вариантов хранения многочленов в зависимости от заданного упорядочения на мономах.

Для степенных упорядочений, таких как DegLex и DegRevLex, был рассмотрен вариант хранения мономов многочлена в виде `vector<set<CMonomial>>`, где `CMonomial` — класс монома. При этом в i-й ячейке вектора хранится множество всех мономов степени i. Тогда сложение многочленов сводится к сложению покомпонентно, а в каждой компоненте можно параллельно пройти по двум множествам в порядке обхода по возрастанию и таким образом создать новое множество, являющееся их симметрической разностью. В этом случае легко выполнить операцию извлечения старшего монома: нужно взять наибольший элемент множества, соответствующего старшей по степени компоненте многочлена. При этом мономы внутри одной компоненты уже не нужно сравнивать по степени, достаточно сравнивать их по порядку Lex или RevLex.

Второй подход состоял в том, чтобы хранить все мономы в одном множестве `set<CMonomial>`. Реализация в этом случае похожа на первый случай, однако отличается тем, что мономы внутри множества уже нужно сравнивать вначале по степени, зато не нужно вычислять степень мономов при сложении и умножении.

Наконец, третий подход состоял в том, чтобы хранить все мономы просто в векторе `vector<CMonomial>`, упорядоченность которого следует восстанавливать после выполнения каждой операции. Здесь нет накладных расходов, которые есть у `set<CMonomial>` при хранении, но зато при добавлении монома к многочлену уже нужно пройти через весь список мономов, чтобы найти место для вставки.

Были реализованы все три варианта. На тестовых примерах лучше себя показал последний вариант, однако это, по всей видимости, связано с тем, что примеры относительно небольшие и количество мономов в многочлене в них невелико. При увеличении размера задачи один из первых двух подходов может дать преимущество.

Дополнительные методы оптимизации

Рассмотрим дополнительные методы ускорения программы в случае добавленных уравнений поля. В алгоритме Бухбергера основное время уходит на явную редукцию S-полинома. Если делается только top-редукция, т. е. редукция лишь старших слагаемых, то иногда можно ускорить выбор подходящего редуктора. Различных мономов в алгоритме встречается не так много, и старшие мономы редуцируемых многочленов часто будут повторяться. И если однажды уже был получен какой-то редуктор для такого старшего монома, то можно запомнить его и использовать при последующих вызовах функции. Для этого нужно в ассоциативном массиве хранить отображение из мономов в многочлены, ставящее в соответствие моному редуктор. Добавлять и искать элементы

в нём можно за логарифмическое время от его размера. Получив очередной старший моном, мы сначала определим, не найден ли редуктор для него заранее, а в случае неудачи будем просматривать список всех редукторов как обычно*.

Для ускорения одной из самых частых операций — выбора старшего монома у многочлена — также предпринято несколько шагов. Во-первых, в многочлене всегда хранится ровно столько ячеек, сколько необходимо, т. е. в последней ячейке обязательно есть хотя бы один моном. Для поддержания этого свойства при сложении многочленов и умножении их на мономы нужно в конце удалять пустые компоненты, однако это избавляет от необходимости каждый раз проходить по всему многочлену, чтобы найти последнюю ячейку, в которой хранится хотя бы один моном, а именно в ней и хранится старший моном многочлена в силу порядка, заданного на мономах. Сами же ячейки хранятся в виде упорядоченных множеств, благодаря чему мы можем найти наибольший элемент за количество операций, логарифмически зависящее от размера ячейки.

В алгоритме F4 есть несколько мест, в которых чётко не указано оптимальной стратегии действия, так как оптимальная стратегия для выполнения соответствующих шагов неизвестна. Одно из таких мест — стратегия выбора набора критических пар, по которым будут строиться S-полиномы, рассматриваемые на очередном шаге. Если не рассматривать дополнительных параметров, то имеется несколько разных стратегий:

1) выбирать одну какую-нибудь пару — получаем обычный алгоритм Бухбергера;
2) выбирать сразу все критические пары;
3) выбирать все критические пары наименьшей степени (нормальная стратегия);
4) выбирать все критические пары наименьшей степени после гомогенизации (сахарная стратегия).

Фожер рекомендует третью стратегию, для однородных идеалов она является естественным выбором. Однако при реализации было выяснено, что в случае вычисления базисов Грёбнера для серий cyclic с добавлением уравнений поля вторая стратегия даёт выигрыш в два с небольшим раза. Теоретическая зависимость эффективности вычислений определённых серий примеров от конкретных стратегий пока мало изучена.

На самом деле стратегия выбора критических пар на каждый шаг важна и в обычном алгоритме Бухбергера, если к нему применять оптимизации, связанные со вторым критерием Бухбергера. Дело в том, что критическая пара, добавленная в множество критических пар для рассмотрения, может так и не оказаться рассмотренной из-за какой-то будущей критической пары, вместе с которой они образуют бухбергеровскую тройку. Поэтому выбирать для рассмотрения все имеющиеся на данный момент критические пары на каждом шагу — это не то же самое, что выбирать их по одной по порядку. Оказалось,

*Заметим, что для вычислении инволютивных базисов Жане имеется специальная структура данных, дерево Жане, позволяющая быстро найти подходящий делитель [2].

что и в обычном алгоритме Бухбергера именно стратегия выбора сразу всех имеющихся критических пар даёт для случая с автоматическим добавлением уравнений поля выигрыш, причём, что удивительно, самый большой из всех применявшихся оптимизаций. После смены стратегии с выбора одной пары на выбор сразу всех алгоритм ускорился даже не в константу раз, а на порядок.

Результаты работы и временные показатели

Были реализованы два алгоритма: базовый алгоритм Бухбергера, снабжённый всеми описанными оптимизациями, и алгоритм F4, снабжённый теми из оптимизаций, которые к нему применимы. Программы называются PackedBuhberger и PackedF4, написаны на C++. Они запускались на трёх стандартных сериях примеров: cyclic, eco и katsura. Программы выполнялись на компьютере Intel® Celeron® 2,4 ГГц, 1 Гб RAM. Сравнение проводилось с пакетом FGb 1.34 для системы компьютерной алгебры Maple. FGb также реализован на C. Ниже приведены таблицы с временем работы описываемых алгоритмов и FGb на трёх сериях. В некоторых случаях в таблице стоит прочерк, что означает, что программа либо работает дольше разумного предела ожидания, либо не может отработать в связи с внутренними ограничениями (FGb начиная с cyclic13).

Cyclic			
Тест	PackedBuhberger	PackedF4	FGb 1.34
cyclic8	0,062	0,219	0,061
cyclic9	0,156	0,141	0,203
cyclic10	0,640	3,859	1,437
cyclic11	1,609	1,500	9,328
cyclic12	7,766	119,238	80,359
cyclic13	18,844	—	—
cyclic14	108,063	—	—
cyclic15	305,422	—	—

Eco			
Тест	PackedBuhberger	PackedF4	FGb 1.34
eco8	0,031	0,078	0,062
eco9	0,078	0,140	0,047
eco10	0,187	0,469	0,109
eco11	0,485	2,078	0,062
eco12	0,985	4,843	0,062
eco13	9,421	12,515	0,093
eco14	34,078	13,812	0,094
eco15	163,765	45,313	0,092

Katsura			
Тест	PackedBuhberger	PackedF4	FGb 1.34
katsura8	0,000	0,000	0,062
katsura9	0,000	0,000	0,047
katsura10	0,000	0,000	0,062
katsura11	0,000	0,000	0,061
katsura12	0,000	0,000	0,062
katsura13	0,000	0,000	0,094
katsura14	0,000	0,000	0,062
katsura15	0,000	0,000	0,062

Параллельная версия

Исследование показало, что при достаточном увеличении размера задачи на стандартных примерах специфическая реализация начинает обгонять по времени общую реализацию F4 при запуске на одном процессоре. Кроме того, выясняется, что в случае F_2 операция приведения матрицы к ступенчатому виду уже не занимает 99 % всего времени работы алгоритма, как было в случае общей реализации F4, а занимает 30—60 % времени на одном процессоре, однако другие операции, нераспараллеленные, начинают занимать более существенную часть времени. Соответственно, при запуске на нескольких процессорах специфическая для F_2 версия алгоритма начинает проигрывать общей, так как в ней распараллелена только операция приведения матрицы к ступенчатому виду, но, с другой стороны, появляется возможность общего ускорения за счёт распараллеливания других основных частей алгоритма.

За основу параллельной версии программы были взяты те же специфические структуры данных, что и в последовательной версии, а распараллеливаемая часть является общей с основной реализацией параллельного F4.

Запуск программы на тестовых примерах из серий cyclic, hcyclic, extcyc, redcyc показал, что на тестовых примерах маленького размера специфическая реализация немного проигрывает по времени общей из-за менее эффективных операций над многочленами. Однако при увеличении размера задачи при запуске на одном процессоре специфическая реализация начинает работать быстрее, и уже при временах порядка десятков секунд начинает выигрывать по времени, а на следующем порядке задачи выигрыш ещё увеличивается.

Задача приведения матрицы к ступенчатому виду сама по себе распараллеливается сравнительно плохо. Соответственно, если этот этап становится существенно быстрее сам по себе, то есть шанс, что распараллеливание всей программы целиком, а не только матричной части, даст здесь выигрыш.

Тестирование на одном процессоре

Тестирование проводилось на машине Intel Pentium 2,13 ГГц, 2 Гб RAM. Времена работы программ указаны в секундах. В двух последних столбцах

показана доля суммарного времени работы метода Гаусса по приведению матриц к ступенчатому виду.

Тест	F4	F4F2	F4 Gauss	F4F2 Gauss
cyclic12	26,70	19,08	22,08 (83 %)	7,21 (38 %)
cyclic13	260,88	102,89	245,33 (94 %)	54,01 (53 %)
hcyclic12	29,16	24,53	23,62 (81 %)	9,11 (37 %)
hcyclic13	258,36	142,89	241,11 (91 %)	74,44 (53 %)
redcyc13	23,00	25,14	18,53 (81 %)	8,96 (36 %)
redcyc14	331,45	197,75	310,29 (94 %)	111,95 (57 %)
extcyc12	20,61	23,31	16,62 (81 %)	8,22 (35 %)
extcyc13	774,99	210,83	736,97 (95 %)	133,88 (64 %)

Результаты показывают, что, во-первых, алгоритмы, учитывающие специфику поля F_2, работают существенно быстрее (кроме «простых» тестовых примеров redcyc13 и extcyc13, на которых имеется несущественный проигрыш). Во-вторых, доля именно матричных вычислений в специфическом алгоритме уменьшается по сравнению с обычной версией.

Тестирование на нескольких процессорах

В таблице приведены данные о запуске программы на некоторых тестовых примерах на 1, 2, 4 и 8 процессорах. В каждой ячейке приведено общее время работы программы и время в секундах, затраченное на приведение матрицы к ступенчатому виду (единственная распараллеленная часть на данный момент).

Тест	1 процессор	2 процессора	4 процессора	8 процессоров
cyclic14	613/360	460/190	380/109	350/75
hcyclic13	146/71	108/37	96/24	90/18
hcyclic14	843/480	621/247	493/128	485/119
redcyc14	201/105	150/50	135/36	133/30
redcyc15	871/523	635/282	541/192	491/147

Отсюда видим, что матричная часть действительно распараллеливается, но занимает она далеко не всё процессорное время, а время на выполнение остальных функций (в основном это `Preprocess` и `Update`) практически не меняется. Следовательно, потенциальное ускорение этого алгоритма можно получить из распараллеливания этих функций.

Программный код реализации основных операций над мономами

Ниже приведены комментарии к реализации важнейших операций.

```
// Моном хранится как целочисленный вектор:
vector<unsigned int> monom;

// Получение степени монома:
int Cmonomial::getDegree() {
    int res = 0;
    for (int i = 0; i < monom.size(); i++)
        res += CmonomialUtilities::getBitCount(monom[i]);
    return res;
}

vector<char>* CMonomialUtilities::integerBitCounts(NULL);

void CMonomialUtilities::initBitCounts() {
    integerBitCounts = new vector<char>;
    integerBitCounts->resize(1 << 16);
    integerBitCounts->at(0) = 0;
    for (int i = 1; i < (1 << 16); i++)
        integerBitCounts->at(i) =
            integerBitCounts->at(i >> 1) + (i & 1);
}

int CMonomialUtilities::getBitCount(unsigned int a) {
    return integerBitCounts->at(a >> 16) +
        integerBitCounts->at(a & ((1 << 16) - 1));
}

// Умножение двух мономов:
void operator*= (const CMonomial& givenMonomial) {
    for (int i = 0; i < monom.size(); i++)
        monom[i] |= givenMonomial.monom[i];
}

// Проверка делимости монома на моном:
bool divisibleBy(const CMonomial& m) const {
    for (int i = 0; i < monom.size(); i++)
        if ((monom[i] & m.monom[i]) != m.monom[i])
            return false;
    return true;
}
```

```
// Вычисление частного при делении монома на моном
// (предполагается, что проверка делимости уже произведена):
void operator/= (const CMonomial& givenMonomial) {
    for (int i = 0; i < monom.size(); i++)
        monom[i] ^= givenMonomial.monom[i];
}

// Вычисление наименьшего общего кратного двух мономов:
static const CMonomial lcm (const CMonomial& firstMonomial,
    const CMonomial& secondMonomial) {
    CMonomial res;
    for (int i = 0; i < firstMonomial.monom.size(); i++)
        res.monom[i] =
            firstMonomial.monom[i] | secondMonomial.monom[i];
    return res;
}
```

Благодарности

Коллектив авторов благодарит руководителя проекта д. ф.-м. н. профессора А. В. Михалёва. Авторы выражают глубокую признательность А. М. Чеповскому и А. А. Михалёву, а также П. Наливайко.

Работа посвящается нашему научному руководителю Евгению Васильевичу Панкратьеву, трагически погибшему 23 января 2008 года.

Литература

[1] Гердт В. П., Зинин М. В. Инволютивный метод вычисления базисов Грёбнера над F_2 // Программирование. — 2008. — № 4. — С. 8—24.

[2] Гердт В. П., Янович Д. А., Блинков Ю. А. Быстрый поиск делителя Жане // Программирование. — 2001. — № 1. — С. 32—36.

[3] Кокс Д., Литтл Дж., О'Ши Д. Идеалы, многообразия и алгоритмы. — М.: Мир, 2000.

[4] Митюнин В. А., Панкратьев Е. В. Параллельные алгоритмы построения базисов Грёбнера // Соврем. мат. и её прил. — 2005. — Т. 30. — С. 46—64.

[5] Панкратьев Е. В. Элементы компьютерной алгебры. — М.: Интернет-университет информационных технологий, 2007.

[6] Янович Д. А. Оценка эффективности распределённых вычислений базисов Грёбнера и инволютивных базисов // Программирование. — 2008. — № 4. — С. 32—40.

[7] Attardi G., Traverso C. Strategy-accurate parallel Buchberger algorithm // J. Symb. Comput. — 1996. — Vol. 21, no. 4-6. — P. 411—425.

[8] Becker T., Weispfenning V. Gröbner Bases. A Computational Approach to Commutative Algebra. — New York: Springer, 1993. — (Grad. Texts Math.; Vol. 141).

[9] Buchberger B. Gröbner bases: An algorithmic method in polynomial ideal theory // Multidimensional Systems Theory. — Reidel, 1985. — P. 184—232.

[10] Faugère J.-C. Parallelization of Gröbner bases // Parallel and Symbolic Computation. — World Scientific, 1994. — (Lect. Notes Comput.; Vol. 5). — P. 124—132.

[11] Faugère J.-C. A new efficient algorithm for computing Gröbner bases (F4) // J. Pure Appl. Algebra. — 1999. — Vol. 139, no. 1-3. — P. 61—88.

[12] Faugère J.-C. A new efficient algorithm for computing Gröbner bases without reduction to zero (F5) // Proc. of the 2002 Int. Symp. on Symbolic and Algebraic Computation (ISSAC). — ACM Press, 2002. — P. 75—83.

[13] Gebauer R., Möller H. M. On an installation of Buchberger's algorithm // J. Symbolic Comput. — 1988. — Vol. 6. — P. 275—286.

[14] Gerdt V. P. Gröbner bases and involutive methods for algebraic and differential equations // Math. Comput. Modelling. — 1997. — Vol. 25, no. 8/9. — P. 75—90.

[15] Gerdt V. P., Yanovich D. A. Parallel computation of involutive and Gröbner bases // Proc. of CASC 2003 / V. G. Ganza, E. W. Mayr, E. V. Vorozhtsov, eds. — Garching: Institute of Informatics, Technical University of Munich, 2004. — P. 185—194.

[16] Kipnis A., Shamir A. Cryptanalysis of the HFE public key cryptosystem by relinearization // Advances in Cryptology — Crypto'99. — Springer, 1999. — (Lect. Notes Comput. Sci.; Vol. 1666). — P. 19—30.

[17] Möller H. M., Mora T., Traverso C. Gröbner bases computation using syzygies // Int. Symp. on Symbolic and Algebraic Computation 92. ISSAC 92. Berkeley, CA, USA, July 27—29, 1992 / P. S. Wang, ed. — Baltimore: ACM Press, 1992. — P. 320—328.

[18] Reeves A. A. A parallel implementation of Buchberger's algorithm over $\mathbb{Z}_p$ for $p = 31991$ // J. Symbolic Comput. — 1998. — P. 229—244.

[19] Roune B. H. The F4 algorithm: Speeding up Gröbner basis computation using linear algebra. — 2005. — `http://www.broune.com/papers/f4.pdf`.

[20] Segers A. J. M. Algebraic attacks from a Gröbner bases perspective: MSc Thesis. — Technische Universiteit Eindhoven, 2004.

[21] Stegers T. Faugere's F5 algorithm revisited: Diplom Thesis. — Technische Universität Darmstadt, 2005. — `http://wwwcsif.cs.ucdavis.edu/~stegers/diplom_stegers.pdf`.

[22] Trân Q.-N. Parallel computation and Gröbner bases: An application for converting bases with the Gröbner walk // Gröbner Bases and Applications / B. Buchberger, F. Winkler, eds. — Cambridge Univ. Press, 1998. — P. 519—534.

[23] Yanovich D. A. Parallelization of an algorithm for computation of involutive Janet bases // Program. Comput. Software. — 2002. — Vol. 28, No. 2. — P. 66—69.