Нижегородский государственный университет им. Н.И. Лобачевского
Факультет вычислительной математики и кибернетики

Образовательный комплекс «Параллельные численные методы»

Лекционные материалы

Баркалов К.А.

При поддержке компании Intel

Нижний Новгород
2011

Содержание

3. Прямые методы решения СЛАУ

Методы решения систем линейных алгебраических уравнений (СЛАУ) относятся к численным методам алгебры. При формальном подходе решение подобных задач не встречает затруднений: решение системы можно найти, раскрыв определители в формуле Крамера. Однако при непосредственном раскрытии определителей решение системы с n неизвестными требует $O(n!)$ арифметических операций; уже при n порядка 20 такое число операций недоступно для современных компьютеров. При сколько-нибудь больших n применение методов с таким порядком числа операций будет невозможно и в обозримом будущем. Другой причиной, по которой этот классический способ неприменим даже при малых n, является сильное влияние на окончательный результат округлений при вычислениях.

Методы решения алгебраических задач можно разделить на точные и итерационные. Классы задач, для решения которых обычно применяют методы этих групп, можно условно назвать соответственно классами задач со средним и большим числом неизвестных. Изменение объема и структуры памяти вычислительных систем, увеличение их быстродействия и развитие численных методов приводят к смещению границ применения методов в сторону систем более высоких порядков.

Например, в 80-х годах прошлого века точные методы применялись для решения систем до порядка 10^4, итерационные – до порядка 10^7, в 90-х – до порядков 10^5 и 10^8 соответственно. Современные суперкомпьютеры способны использовать точные методы при решении еще больших систем.

Мы будем рассматривать систему из n линейных алгебраических уравнений вида

$$\begin{aligned} a_{11}x_1 &+ a_{12}x_2 &+ \ldots + a_{1n}x_n &= b_1 \\ a_{21}x_1 &+ a_{21}x_2 &+ \ldots + a_{2n}x_n &= b_2 \\ &\ldots & & \\ a_{n1}x_1 &+ a_{n2}x_2 &+ \ldots + a_{nn}x_n &= b_n \end{aligned} \qquad (3.1)$$

В матричном виде система может быть представлена как

$$Ax=b, \qquad (3.2)$$

где $A=(a_{ij})$ есть вещественная матрица размера $n\times n$; b и x – вектора из n элементов.

Под задачей решения системы линейных уравнений для заданных матрицы A и вектора b мы будем считать нахождение значения вектора неизвестных x, при котором выполняются все уравнения системы.

3.1. Метод исключения Гаусса

В первую очередь рассмотрим алгоритмыы, предназначенные для решения системы

$$Ax=b, \tag{3.3}$$

с произвольной квадратной матрицей A. Основой для всех них служит широко известный *метод последовательного исключения неизвестных*, или же *метод Гаусса*.

Метод Гаусса основывается на возможности выполнения преобразований линейных уравнений, которые не меняют при этом решение рассматриваемой системы (такие преобразования носят наименование эквивалентных). К числу таких преобразований относятся:

- умножение любого из уравнений на ненулевую константу,
- перестановка уравнений,
- прибавление к уравнению любого другого уравнения системы.

Метод Гаусса включает последовательное выполнение двух этапов. На первом этапе, который называется *прямой ход*, исходная система линейных уравнений при помощи последовательного исключения неизвестных приводится к верхнему треугольному виду. При выполнении *обратного хода* (второй этап алгоритма) осуществляется определение значений неизвестных.

3.1.1. Последовательный алгоритм

Прямой ход состоит в последовательном исключении неизвестных в уравнениях решаемой системы линейных уравнений.

На итерации i, $1\le i<n$, метода производится исключение неизвестной i для всех уравнений с номерами k, больших i (т.е. $i<k\le n$). Для этого из этих уравнений осуществляется вычитание строки i, умноженной на константу a_{ki}/a_{ii} с тем, чтобы результирующий коэффициент при неизвестной x_i в строках оказался нулевым – все необходимые вычисления могут быть определены при помощи соотношений:

$$\begin{aligned} a_{kj}^{'} &= a_{kj} - \mu_{ki} a_{ij}, \\ b_{k}^{'} &= b_{k} - \mu_{ki} b_{i}, \end{aligned} \quad i \le j \le n, i < k \le n, 1 \le i < n \ , \tag{3.4}$$

где $\mu_{ki} = a_{ki} / a_{ii}$ *– множители Гаусса*.

В итоге приходим к системе $Ux=c$ с верхней треугольной матрицей

$$U = \begin{pmatrix} u_{1,1} & u_{1,2} & \dots & u_{1,n} \\ 0 & u_{2,2} & \dots & u_{2,n} \\ & & \dots & \\ 0 & 0 & \dots & u_{n,n} \end{pmatrix},$$

При выполнении прямого хода метода Гаусса строка, которая используется для исключения неизвестных, носит наименование *ведущей*, а диагональный элемент ведущей строки называется *ведущим элементом*. Как можно заметить, выполнение вычислений является возможным только, если ведущий элемент имеет ненулевое значение. Более того, если ведущий элемент a_{ii} имеет малое значение, то деление и умножение строк на этот элемент может приводить к накоплению вычислительной погрешности и вычислительной неустойчивости алгоритма.

Избежать подобной проблемы можно, если при выполнении каждой очередной итерации прямого хода метода Гаусса определить коэффициент с максимальным значением по абсолютной величине в столбце, соответствующем исключаемой неизвестной, т.е.

$$y = \max_{i \le k \le n} |a_{ki}| ,$$

и выбрать в качестве ведущей строку, в которой этот коэффициент располагается (данная схема выбора ведущего значения носит наименование *метода главных элементов*).

Обратный ход алгоритма состоит в следующем. После приведения матрицы коэффициентов к верхнему треугольному виду становится возможным определение значений неизвестных. Из последнего уравнения преобразованной системы может быть вычислено значение переменной x_n, после этого из предпоследнего уравнения становится возможным определение переменной x_{n-1} и т.д. В общем виде выполняемые вычисления при обратном ходе метода Гаусса могут быть представлены при помощи соотношений:

$$x_i = \left(c_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) \Big/ u_{ii} \ , \ i=n,\dots,1.$$

Оценим трудоемкость метода Гаусса. При выполнении прямого хода число операций составит

$$\sum_{i=1}^{n-1} 2(n-i)^2 = \frac{n(n-1)(2n-1)}{3} = \frac{2}{3}n^3 + O(n^2).$$

Для выполнения обратного хода потребуется

$$\sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2} = O(n^2).$$

Таким образом, общее время выполнения метода Гаусса при больших n можно оценить как

$$T_1 = \frac{2}{3}n^3\tau,$$

где τ – время выполнения одной операции.

3.1.2. Параллельный алгоритм

При внимательном рассмотрении метода Гаусса можно заметить, что все вычисления сводятся к однотипным вычислительным операциям над строками матрицы коэффициентов системы линейных уравнений. Как результат, в основу параллельной реализации алгоритма Гаусса может быть положен принцип распараллеливания по данным. В качестве *базовой подзадачи* можно принять тогда все вычисления, связанные с обработкой одной строки матрицы A и соответствующего элемента вектора b. Рассмотрим общую схему параллельных вычислений и возникающие при этом информационные зависимости между базовыми подзадачами.

Для выполнения прямого хода метода Гаусса необходимо осуществить (n–1) итерацию по исключению неизвестных для преобразования матрицы коэффициентов A к верхнему треугольному виду. Выполнение итерации i, $1 \le i \le n$, прямого хода метода Гаусса включает ряд последовательных действий. Прежде всего, в самом начале итерации необходимо выбрать ведущую строку, которая при использовании метода главных элементов определяется поиском строки с наибольшим по абсолютной величине значением среди элементов столбца i, соответствующего исключаемой переменной x_i. Зная ведущую строку, подзадачи выполняют вычитание строк, обеспечивая тем самым исключение соответствующей неизвестной x_i.

При выполнении обратного хода метода Гаусса подзадачи выполняют необходимые вычисления для нахождения значения неизвестных. Как только какая-либо подзадача i, $1 \le i \le n$, определяет значение своей переменной x_i, это значение должно быть использовано всеми подзадачам с номерами k,

k<*i*: подзадачи подставляют полученное значение новой неизвестной и выполняют корректировку значений для элементов вектора *b*.

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью. Однако размер матрицы, описывающей систему линейных уравнений, является существенно большим, чем число потоков в программе (т.е., $p << n$), и базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько строк матрицы. При этом применение последовательной схемы разделения данных для параллельного решения систем линейных уравнений приведет к неравномерной вычислительной нагрузке между потоками: по мере исключения (на прямом ходе) или определения (на обратном ходе) неизвестных в методе Гаусса для большей части потоков все необходимые вычисления будут завершены и они окажутся простаивающими. Возможное решение проблемы балансировки вычислений может состоять в использовании ленточной циклической схемы для распределения данных между укрупненными подзадачами. В этом случае матрица *A* делится на наборы (полосы) строк вида (см. рис. 1.5).

$$A = (A_1, A_2, ..., A_p)^T,$$

$$A_i = (a_{i_1}, a_{i_2}, ..., a_{i_k}), i_j = i + jp, 1 \le j \le k, k = n/p.$$

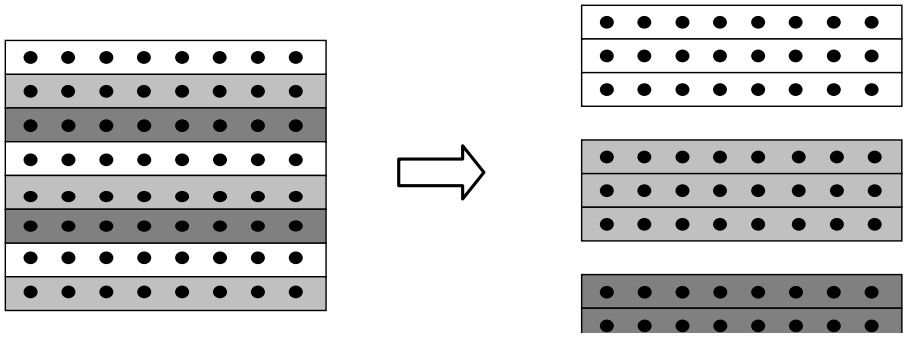


Рис. 3.1 Ленточная схема

Сопоставив схему разделения данных и порядок выполнения вычислений в методе Гаусса, можно отметить, что использование циклического способа формирования полос позволяет обеспечить лучшую балансировку вычислительной нагрузки между подзадачами.

Итак, проведя анализ последовательного варианта алгоритма Гаусса, можно заключить, что распараллеливание возможно для следующих вычислительных процедур:

- поиск ведущей строки,
- вычитание ведущей строки из всех строк, подлежащих обработке,
- выполнение обратного хода.

Оценим трудоемкость рассмотренного параллельного варианта метода Гаусса. Пусть n есть порядок решаемой системы линейных уравнений, а p, $p<n$, обозначает число потоков. При разработке параллельного алгоритма все вычислительные операции, выполняемые алгоритмом Гаусса, были распределены между потоками параллельной программы. Следовательно, время, необходимое для выполнения вычислений на этапе прямого хода, можно оценить как

$$T_p = T_1 / p .$$

Подставив выражение T_1 получим, что время выполнения вычислений для параллельного варианта метода Гаусса описывается выражением:

$$T_p = \frac{2n^3\tau}{3p} .$$

Теперь можно оценить величину накладных расходов, обусловленных организацией и закрытием параллельных секций. Пусть δ – время, необходимое на организацию и закрытие параллельной секции. Параллельная секция создается при каждом выборе ведущей строки, при выполнении вычитания ведущей строки из остальных строк линейной системы, подлежащих обработке, а также при выполнении каждой итерации обратного хода метода Гаусса. Таким образом, общее число параллельных секций составляет $3(n–1)$.

Сводя воедино все полученные оценки можно заключить, что время выполнения параллельного метода Гаусса описывается соотношением

$$T_p = \frac{2n^3\tau}{3p} + 3(n-1)\delta . \qquad (3.5)$$

3.1.3. Связь метода Гаусса и LU-разложения

LU-разложение – представление матрицы A в виде

$$A=LU, \qquad (3.6)$$

где L – нижняя треугольная матрица с диагональными элементами, равными единице, а U – верхняя треугольная матрица с ненулевыми диагональными элементами. *LU*-разложение также называют *LU*-факторизацией. Из-

вестно [4], что *LU*-разложение существует и единственно, если главные миноры матрицы *A* отличны от нуля.

Алгоритм *LU*-разложения тесно связан с методом исключения Гаусса. В самом деле, пусть мы решаем систему уравнений вида (3.2). Непосредственно проверяется, что преобразования *k*-го шага метода Гаусса равносильны домножению системы (3.2) слева на матрицу

$$M^{(k)} = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 1 & & & & \\ & & -\mu_{k+1,k} & 1 & & & \\ & & -\mu_{k+2,k} & & 1 & & \\ & & & & & \ddots & \\ & & -\mu_{n,k} & & & & 1 \end{bmatrix},$$

где μ_{ik} – множители Гаусса из (3.4). Как было рассмотрено в п. 3.1.1, прямой ход метода Гаусса преобразует исходную систему уравнений к виду

$$Ux=c,$$

с верхней треугольной матрицей *U*. Зная матрицы $M^{(i)}$, можно записать матрицу *U* и вектор *c* как

$$U = M^{(n-1)}M^{(n-2)}...M^{(1)}A,$$

$$c = M^{(n-1)}M^{(n-2)}...M^{(1)}b.$$

Обозначим $L^{-1} = M^{(n-1)}M^{(n-2)}...M^{(1)}$. Можно непосредственно проверить, что

$$L = \begin{bmatrix} 1 & & & & \\ \mu_{21} & 1 & & & \\ \vdots & \vdots & \ddots & & \\ \mu_{n-1,1} & \mu_{n-1,2} & \cdots & 1 & \\ \mu_{n,1} & \mu_{n,2} & \cdots & \mu_{n,n-1} & 1 \end{bmatrix}.$$

Отсюда получаем *A*=*LU*.

Таким образом, матрицу *L* можно получить как нижнюю треугольную матрицу коэффициентов Гаусса, а матрицу *U* – как верхнюю треугольную матрицу, получаемую в результате работы метода Гаусса. При этом оче-

видно, что трудоемкость получения *LU*-факторизации будет такой же – $2/3n^3 + O(n^2)$.

Рассмотренный нами алгоритм *LU*-факторизации реализован с помощью *исключения по столбцу*. Следует отметить, что можно сформулировать аналогичный алгоритм, основанный на *исключении по строке*. В самом деле, основная идея алгоритма с помощью исключения по столбцу заключается в том, что на *i*-й итерации ведущая строка с подходящими множителями вычитается из строк, лежащих ниже, чтобы занулить все элементы матрицы, расположенные в *i*-м столбце ниже диагонали. Между тем возможно и другое: на каждой *i*-й итерации можно вычитать из *i*-й строки все строки, расположенные выше, умноженные на подходящие коэффициенты, так, чтобы занулить все элементы *i*-й строки левее диагонали. При этом элементы матрицы *L* ниже главной диагонали и элементы матрицы *U* на главной диагонали и выше нее можно вычислять на месте матрицы *A*. Как и в случае исключения по столбцу, приведенная схема требует проведения $2/3n^3 + O(n^2)$ операций.

Рассмотрим теперь еще один способ *LU*-факторизаци, называемый *компактной схемой*. Пусть матрица $A(n\times n)$ допускает *LU*-разложение (3.6), где

$$L = \begin{bmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & \dots & 0 \\ . & . & \dots & . \\ l_{n1} & l_{n2} & \dots & 1 \end{bmatrix}, \; U = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ . & . & \dots & . \\ 0 & 0 & \dots & u_{nn} \end{bmatrix},$$

т.е. $l_{ii} = 1$, $l_{ij} = 0$ при $i<j$, а $u_{ij} = 0$ при $j<i$. Из соотношения (3.6) следует, что

$$a_{ij} = \sum_{k=1}^{n} l_{ik} u_{kj}, \; i, j = \overline{1,n}.$$

Преобразуем эту сумму двумя способами:

$$\sum_{k=1}^{n} l_{ik} u_{kj} = \sum_{k=1}^{i-1} l_{ik} u_{kj} + l_{ii} u_{ij} + \sum_{k=i+1}^{n} l_{ik} u_{kj} = \sum_{k=1}^{i-1} l_{ik} u_{kj} + l_{ii} u_{ij},$$

$$\sum_{k=1}^{n} l_{ik} u_{kj} = \sum_{k=1}^{j-1} l_{ik} u_{kj} + l_{ij} u_{jj} + \sum_{k=j+1}^{n} l_{ik} u_{kj} = \sum_{k=1}^{j-1} l_{ik} u_{kj} + l_{ij} u_{jj}.$$

Отсюда находим

$$l_{11} = 1, \; u_{11} = a_{11},$$

$$u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}\,,\ i \le j, i = \overline{1, j-1},\ j = \overline{2, n},$$

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right) \Big/ u_{jj}\,,\ i > j,\ j = \overline{1, n-1},\ i = \overline{2, n}.$$

Оценка числа операций данного алгоритма *LU*-факторизаци также составляет

$$\frac{2}{3} n^3 + O(n^2).$$

Если разложение (3.6) получено, то решение системы (3.2) сводится к последовательному решению двух систем уравнений с треугольными матрицами (*обратный ход*)

$$Ly = b,\ Ux = y. \qquad (3.7)$$

Обратный ход требует $O(n^2)$ операций.

Как следует из приведенных оценок, вычислительная сложность метода исключения Гаусса и метода *LU*-разложения одинакова. Однако если необходимо решить несколько систем с одинаковыми матрицами коэффициентов, но различными векторами свободных членов (правая часть СЛАУ), то метод *LU*-разложения окажется предпочтительным, так как в этом случае нет необходимости производить разложение матрицы коэффициентов многократно. Достаточно лишь сохранить полученные треугольные матрицы в памяти и, подставляя различные вектора свободных членов, получать решения методами прямой и обратной подстановки. Это позволит значительно сократить объем вычислений по сравнению с методом Гаусса.

3.1.4. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода Гаусса для решения систем линейных уравнений проводились при условиях, указанных во введении.

Результаты вычислительных экспериментов приведены в таблице 3.1 (время работы алгоритмов указано в секундах).

Табл. 3.1. Результаты экспериментов (параллельный метод Гаусса)

n	Посл.	Параллельный алгоритм							
		2 потока		4 потока		6 потоков		8 потоков	
		T	*S*	*T*	*S*	*T*	*S*	*T*	*S*

1000	0,44	0,48	0,90	0,28	1,56	0,23	1,93	0,20	2,15
2000	4,45	4,20	1,06	3,01	1,48	2,86	1,56	2,75	1,62
3000	16,29	14,27	1,14	10,98	1,48	10,89	1,50	10,87	1,50
4000	34,62	33,90	1,02	26,18	1,32	26,10	1,33	26,18	1,32
5000	72,20	68,19	1,06	51,45	1,40	51,11	1,41	51,87	1,39

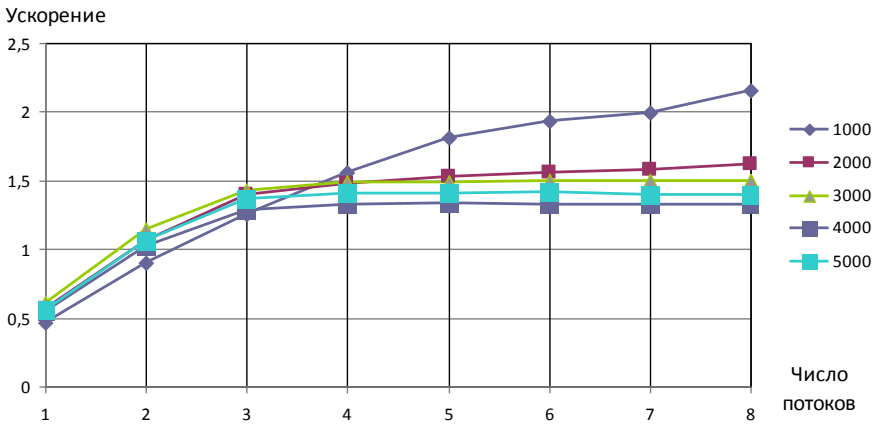


Рис. 3.2. Зависимость ускорения от числа потоков (метод Гаусса)

Как следует из приведенных результатов, при $p \geq 3$ получается ускорение около 1.5, которое не зависит ни от числа потоков, ни от размера матрицы, тогда как оценка (3.5) позволяла нам надеяться на лучшее (исключение составляет случай N=1000). Отсутствие значительного ускорения обусловлено следующим эффектом.

Во-первых, при N=1000 размер матрицы позволяет целиком загрузить ее в кэш процессора, и использовать для проведения операций быстродействующую память. А при N>1000 матрица не помещается в кэш-память целиком, и возрастает число *кэш-промахов*, что приводит к частым обращениям в относительно медленную оперативную память. Эффективно использовать кэш-память можно при блочном разделении данных, что будет показано в следующем пункте.

Во-вторых, компилятор с большей эффективностью оптимизирует чисто последовательную программу, чем более сложную параллельную. И в некоторых случаях однопоточная версия параллельной программы будет работать в 1.5-2 раза медленнее, чем ее чисто последовательный аналог, поэтому здесь и далее мы будем оценивать ускорение параллельной про-

граммы, запущенной в p потоков, по отношению к той же самой программе, запущенной в один поток.

Для иллюстрации сказанного приведем результаты сравнения параллельного алгоритма не с чисто последовательной программой, а с параллельной, запущенной в один поток. Результаты такого сравнения приведены в таблице 3.2 (время работы алгоритмов указано в секундах).

Табл. 3.2. Результаты экспериментов (параллельный метод Гаусса)

N	1 поток	Параллельный алгоритм							
		2 потока		4 потока		6 потоков		8 потоков	
		T	*S*	*T*	*S*	*T*	*S*	*T*	*S*
1000	0,94	0,48	1,94	0,28	3,34	0,23	4,14	0,20	4,61
2000	7,82	4,20	1,86	3,01	2,60	2,86	2,74	2,75	2,85
3000	26,72	14,27	1,87	10,98	2,43	10,89	2,45	10,87	2,46
4000	63,77	33,90	1,88	26,18	2,44	26,10	2,44	26,18	2,44
5000	130,09	68,19	1,91	51,45	2,53	51,11	2,55	51,87	2,51

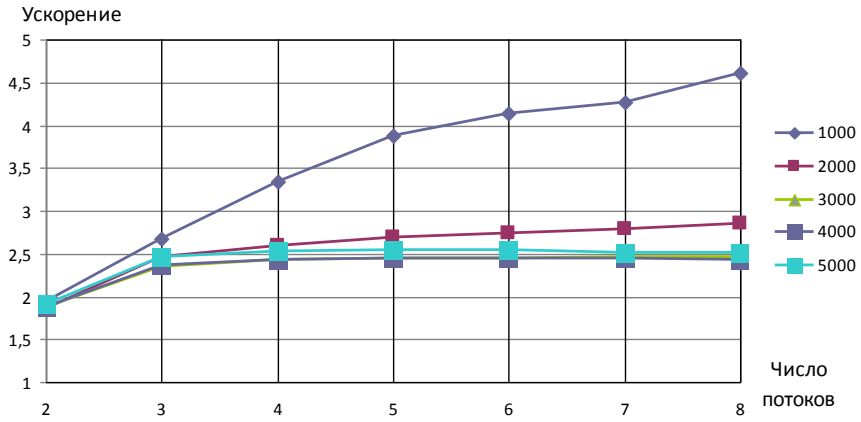


Рис. 3.3. Зависимость ускорения от числа потоков (метод Гаусса)

Как следует из приведенных результатов, относительно хорошее ускорение мы получили лишь для задачи размера 1000 (до 4.6 при решении в 8 потоках), в остальных задачах ускорение составляет примерно 2.5 при $p \geq 3$. В следующем параграфе будет рассмотрен метод, обладающий значительно большим масштабированием.

3.1.5. Блочное *LU*-разложение

Недостаток рассмотренного стандартного алгоритма *LU*-разложения обусловлен тем, что его вычисления плохо соответствует правилам использования *кэш-памяти* – быстродействующей дополнительной памяти компьютера, используемой для хранения копии наиболее часто используемых областей оперативной памяти. Эффективное использование кэша может существенно (в десятки раз) повысить быстродействие вычислений. Размещение данных в кэше может происходить или предварительно (при использовании тех или иных алгоритмов предсказания потребности в данных) или в момент извлечения значений из основной оперативной памяти. При этом подкачка данных в кэш осуществляется не одиночными значениями, а небольшими группами – строками кэша. Загрузка значений в строку кэша осуществляется из последовательных элементов памяти; типовые размеры строки кэша обычно равны 32, 64, 128, 256 байтам (см., например, [7]). Как результат, эффект наличия кэша будет наблюдаться, если выполняемые вычисления используют одни и те же данные многократно и осуществляют доступ к элементам памяти с последовательно возрастающими адресами.

В рассмотренном нами алгоритме *LU*-разложения размещение данных в памяти осуществляется по строкам, а вычисления проводятся – по столбцам, и это приводит к низкой эффективности использования кэша. Возможный способ улучшения ситуации – укрупнение вычислительных операций, приводящее к последовательной обработке некоторых прямоугольных подматриц матрицы *A*.

LU-разложение можно организовать так, что матричные операции (реализация которых допускает эффективное использование кэш-памяти) станут основными. Для этого представим матрицу $A \in R^{n\times n}$ в блочном виде

$$A = \begin{matrix}\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix} \\ \begin{matrix} r & n-r \end{matrix}\end{matrix},$$

где r – блочный параметр, A_{11} – подматрица матрицы A размера $r\times r$, A_{12} – размера $r\times(n–r)$, A_{21} – размера $(n–r)\times r$, A_{22} – размера $(n–r)\times(n–r)$. Компоненты L и U искомого разложения также запишем в блочном виде

$$L = \begin{matrix}\begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix} \\ \begin{matrix} r & n-r \end{matrix}\end{matrix}, \quad U = \begin{matrix}\begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} \begin{matrix} r \\ n-r \end{matrix} \\ \begin{matrix} r & n-r \end{matrix}\end{matrix},$$

где L_{11}, L_{21}, L_{22}, U_{11}, U_{12}, U_{22} – соответствующего размера подматрицы матриц L и U.

Рассмотрим теперь связь между исходной матрицей и ее разложением в блочном виде. Используя соотношение (3.6), запишем

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{bmatrix} = \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & L_{21}U_{12} + L_{22}U_{22} \end{bmatrix}$$

Так как L_{11} и U_{11} являются блоками LU-разложения матрицы A, и

$$A_{11} = L_{11}U_{11},$$

то их можно найти, применив стандартный метод Гаусса для разложения блока A_{11}. Затем можно найти блоки L_{21}, U_{12}, решив треугольные системы с несколькими правыми частями

$$L_{11}U_{12} = A_{12},$$

$$L_{21}U_{11} = A_{21}.$$

Следующий шаг алгоритма состоит в вычислении редуцированной матрицы $\tilde{A}_{22}$, в процессе которого используются ставшие известными блоки L_{21} и U_{12}, блок A_{22} и соотношение $A_{22} = L_{21}U_{12} + L_{22}U_{22}$,

$$\tilde{A}_{22} = A_{22} - L_{21}U_{12} = L_{22}U_{22}.$$

Как следует из данной формулы, LU-разложение редуцированной матрицы $\tilde{A}_{22}$ совпадает с искомыми блоками L_{22}, U_{22} матрицы A, и для его нахождения можно применить описанный алгоритм рекурсивно.

Так же как и иные рассмотренные процедуры разложения, приведенная блочная схема требует порядка $2/3n^3$ операций. Оценим долю матричных операций.

Пусть размер матрицы кратен размеру блока, т.е. $n = rN$. Операции, не являющиеся матричными, используются при выполнении разложения

$$A_{11} = L_{11}U_{11}$$

что требует порядка $2/3r^3$ операций. Так как в процессе блочного разложения приходится решать N подобных систем, то долю матричных операций можно оценить как

$$1-\frac{N\,2r^3/3}{2n^3/3}=1-\frac{1}{N^2}.$$

Значит, при правильном подборе размера блока $r<<n$ почти все операции в данном алгоритме будут матричными, и к их реализации нужно подойти с особой тщательностью. В данном случае для проведения операции матричного умножения целесообразно воспользоваться блочной схемой умножения матриц, выбрав для этого свой размер блока, меньший *r*. Описание блочного алгоритма умножения матриц см., например, в [16].

Из сказанного следует, что распараллеливание блочного алгоритма *LU*-разложения должно осуществлять на уровне матричных операций, распараллеливание которых также подробно рассмотрено в работе [16].

3.1.6. Результаты вычислительных экспериментов

Сначала приведем результаты экспериментов, подтверждающие эффективность блочного *LU*-разложения по сравнению с его исходным неблочным вариантом. В таблице 3.3 и на рисунке 3.4 приведено время работы блочного алгоритма в зависимости от размера матрицы *n* и размера блока *r* (столбец, отмеченный прочерком, соответствует исходному неблочному алгоритму).

Табл. 3.3. Время работы блочного *LU*-разложения

n	Время работы, сек						
	---	*r*=2	*r*=5	*r*=10	*r*=20	*r*=50	*r*=100
1000	0,44	0,568	0,49	0,474	0,464	0,565	0,64
2000	4,45	4,846	3,994	3,76	3,869	4,29	4,992
3000	16,29	16,364	13,556	12,76	13,9	14,352	17,035
4000	34,62	39,921	32,043	30,561	34,055	34,944	40,513
5000	72,20	75,77	62,93	59,842	65,645	66,987	82,15

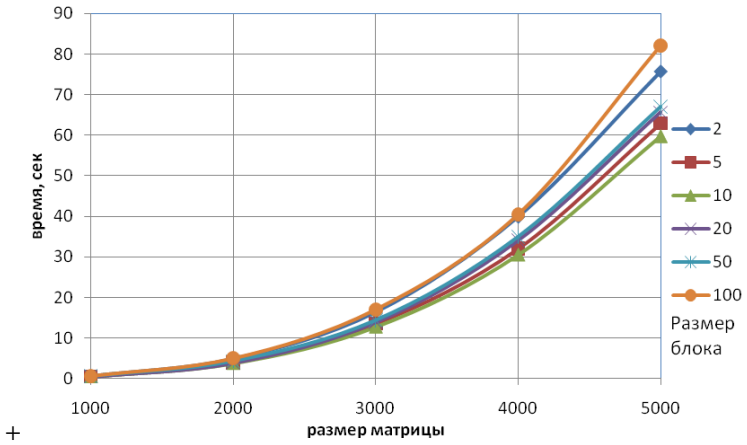


Рис. 3.4. Время работы блочного *LU*-разложения

Результаты показывают, что блочный алгоритм при r=10 работает быстрее как своего прототипа, так и блочного алгоритма при других размерах блока. Поэтому дальнейшие эксперименты будем проводить для блока размера r=10.

Приведем теперь характеристики работы (время и ускорение) параллельного блочного *LU*-разложения.

Табл. 3.4. Время работы параллельного блочного LU-разложения

N	1 поток	Параллельная версия							
		2 потока		4 потока		6 потоков		8 потоков	
		T	*S*	*T*	*S*	*T*	*S*	*T*	*S*
1000	0,62	0,31	2,00	0,17	3,65	0,11	5,72	0,08	8,00
2000	5,07	2,54	1,99	1,34	3,78	0,91	5,60	0,69	7,39
3000	17,02	8,53	1,99	4,52	3,76	3,03	5,62	2,29	7,42
4000	40,59	20,58	1,97	10,81	3,75	7,29	5,57	5,51	7,37
5000	78,80	39,89	1,98	20,83	3,78	14,01	5,62	10,61	7,43

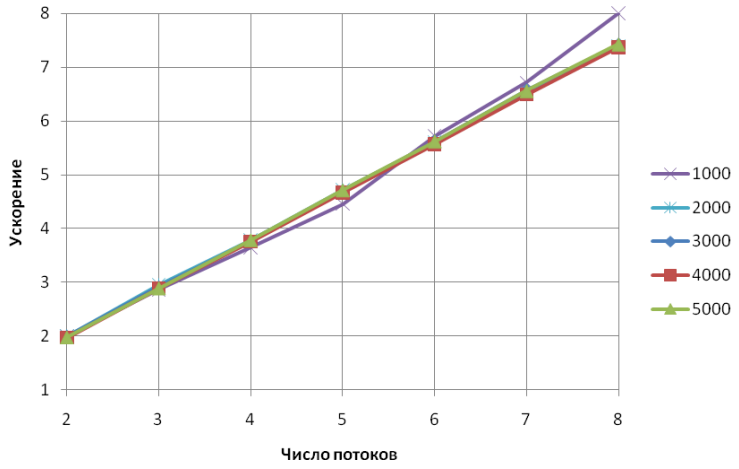


Рис. 3.5. Ускорение параллельного блочного *LU*-разложения

Приведенные результаты свидетельствуют о хорошей (практически линейной) масштабируемости блочного алгоритма, в отличие от стандартной версии метода Гаусса.

3.2. Метод Холецкого

Метод Холецкого (известный также как *метод квадратного корня*) предназначен для решения систем уравнений вида (3.2) с действительной симметричной положительно определенной матрицей. Напомним, что матрица называется *положительно определенной*, если для любого вектора $x \in R^n$ выполнено неравенство

$$(Ax, x) \geq 0.$$

Матрицы с такими свойствами возникают, например, при использовании метода наименьших квадратов и численном решении дифференциальных уравнений.

Метод Холецкого основан на разложении матрицы A в произведение

$$A = LL^T \tag{3.10}$$

где L – нижняя треугольная матрица с положительными элементами на главной диагонали. Известно [4], что разложение (3.10) можно получить для любой матрицы, удовлетворяющей указанным свойствам. Существует также обобщение этого разложения на случай комплекснозначных матриц [4].

Если разложение (3.10) получено, то решение системы (3.2) сводится к последовательному решению двух систем уравнений с треугольными матрицами

$$Ly=b,\ L^Tx=y. \tag{3.11}$$

Следует отметить, что на практике часто необходимо решить последовательность систем вида (3.2), отличающихся лишь правой частью b. Учитывая этот факт и соотношения (3.11), однократная факторизация матрицы A значительно упрощает решение оставшихся систем.

3.2.1. Последовательный алгоритм

Получим расчетные формулы метода. Из условия (3.10) следует, что

$$a_{ij}=\sum_{k=1}^{n} l_{ik}l_{kj}^T\ ,\ i,j=\overline{1,n}\,.$$

Так как матрица A – симметричная, можно рассмотреть лишь случай $i\le j$. Так как $l_{ik}=0$ при $i<k$, данные условия можно записать в виде

$$a_{ij}=\sum_{k=1}^{i-1} l_{ik}l_{kj}^T+l_{ii}l_{ij}+\sum_{k=i+1}^{n} l_{ik}l_{kj}^T=\sum_{k=1}^{i-1} l_{ik}l_{kj}^T+l_{ii}l_{ij}\,,$$

В частности, при $i=j$ получаем

$$a_{ii}=\sum_{k=1}^{i-1} l_{ik}^2-l_{ii}^2\,.$$

откуда

$$l_{ii}=\sqrt{a_{ii}-\sum_{k=1}^{i-1} l_{ik}^2}\ .$$

Далее, для $i<j$ получим

$$l_{ji}=\frac{1}{l_{ii}}\left(a_{ji}-\sum_{k=1}^{i-1} l_{ik}l_{jk}\right). \tag{3.12}$$

Итак, если предположить, что метод квадратного корня применяется к системе уравнений с действительной симметричной положительно определенной матрицей, то элементы матрицы L можно вычислить, начиная с ее левого угла, по следующим расчетным формулам:

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}\ ,\ i = \overline{1,n}\,. \tag{3.13}$$

$$l_{ji} = \frac{1}{l_{ii}}\left(a_{ji} - \sum_{k=1}^{i-1} l_{ik} l_{jk} \right),\ j = \overline{i+1,n}\,. \tag{3.14}$$

Подсчитаем число операций, требующихся для выполнения разложения. Вычисления по формуле (3.13) требуют

$$\sum_{i=2}^{n} 2(i-1) = n(n-1) = O(n^2)$$

операций. Вычисления по формуле (3.14) при каждом фиксированном j требуют

$$\sum_{i=2}^{j-1} 2(i-1) = (j-2)(j-1)$$

операций, а всего потребуется

$$\sum_{j=2}^{n} (j-2)(j-1) = \sum_{k=1}^{n-1} k(k-1) = \frac{n(n-1)(n-2)}{3} = \frac{1}{3} n^3 + O(n^2)$$

операций. Следует отметить, что в дополнение к указанным действиям для расчетов по формулам (3.13), (3.14) потребуется n операций извлечения корня.

Если матрица A факторизована в виде $A=LL^T$, то обратный ход метода квадратного корня состоит в последовательном решении двух систем уравнений

$$Ly=b,\ L^T x=y.$$

Решения этих систем находятся по формулам, аналогичным формулам обратного хода метода Гаусса

$$y_i = \frac{1}{l_{ii}}\left(b_i - \sum_{j=1}^{i-1} l_{ij} y_j \right),\ i = \overline{1,n}\,. \tag{3.15}$$

$$x_i = \frac{1}{l_{ii}}\left(y_i - \sum_{j=i+1}^{n} l_{ji} x_j \right),\ i = \overline{n,1}\,. \tag{3.16}$$

Вычисления по формулам (3.15), (3.16) потребуют $2n(n+1)$ операций, что не оказывает существенного влияния на кубическую трудоемкость метода.

Таким образом, общее время работы метода можно оценить как

$$T_1 = \frac{1}{3} n^3 \tau ,$$

где τ – время выполнения одной операции.

При больших n время работы метода Холецкого будет примерно в два раза меньше, чем метода Гаусса, это сокращение объясняется тем, что A – симметричная матрица, и метод Холецкого учитывает данную особенность задачи.

3.2.2. Организация параллельных вычислений

Как и в методе исключения Гаусса, в данном алгоритме все вычисления сводятся к однотипным вычислительным операциям над строками матрицы L. Как результат, в основу параллельной реализации разложения может быть положен принцип распараллеливания по данным. В качестве базовой подзадачи можно принять тогда все вычисления, связанные с обработкой одной строки матрицы L.

Рассмотрим общую схему параллельных вычислений и возникающие при этом информационные зависимости между базовыми подзадачами.

При выполнении прямого хода метода необходимо осуществить (n–1) итерацию для получения нижней треугольной матрицы L. Выполнение итерации i, $1 \le i \le n$, включает ряд последовательных действий. Прежде всего, в самом начале итерации необходимо вычислить диагональный элемент l_{ii}. Зная диагональный элемент, подзадачи выполняют вычисление i-го столбца матрицы L.

При выполнении обратного хода метода Холецкого подзадачи выполняют необходимые вычисления для нахождения значений сначала вспомогательного вектора y, а затем вектора неизвестных x. Как только какая-либо подзадача i, $1 \le i \le n$, определяет значение своей переменной y_i (или x_i), это значение должно быть использовано всеми подзадачами с номерами k, где $k > i$ при решении системы относительно y, и $k < i$ при решении системы относительно x; подзадачи подставляют полученное значение новой неизвестной и выполняют корректировку значений для элементов вектора правой части.

При использовании систем с общей памятью размер матрицы, описывающей систему линейных уравнений, является большим, чем число потоков (т.е., $p < n$). Следовательно, базовые подзадачи нужно укрупнить, объединив в рамках одной подзадачи несколько строк матрицы. Здесь также можно использовать циклическую схему распределения данных, аналогично методу Гаусса (см. п. 3.1.2).

Итак, проведя анализ последовательного варианта метод Холецкого, можно заключить, что распараллеливание возможно для следующих вычислительных процедур:

- вычисление диагонального элемента l_{ii} в соответствии с (3.13),
- вычисление элементов i-го столбца в соответствии с (3.14),
- выполнение обратного хода в соответствии с (3.15), (3.16).

Оценим трудоемкость рассмотренного параллельного варианта метода Холецкого. Пусть n есть порядок решаемой системы линейных уравнений, а p, $p<n$, обозначает число потоков в программе.

При разработке параллельного алгоритма все вычислительные операции, были распределены между потоками параллельной программы. Пусть δ – время, необходимое на организацию и закрытие параллельной секции. Параллельная секция создается при каждом вычислении диагонального элемента, при вычислении элементов текущего столбца матрицы L, а также на каждой итерации при решении двух систем с треугольными матрицами. Таким образом, общее число параллельных секций составляет $4n$.

Следовательно, теоретическая оценка времени, необходимого для решения системы, составит

$$T_p = \frac{n^3}{3p}\tau + 4n\delta .$$

3.2.1. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта блочного метода Холецкого проводились при стандартных условиях, указанных во введении. С целью формирования симметричной положительно определенной матрицы A размера $n \times n$ для тестовой системы уравнений диагональный элемент матрицы генерировался в диапазоне от n до $2n$, остальные элементы генерировались в диапазоне от 0 до 1 (с учетом симметрии).

Результаты вычислительных экспериментов приведены в таблице 1.6 (время работы алгоритмов указано в секундах).

Табл. 3.5. Результаты экспериментов (параллельное разложение Холецкого)

Размер	1 поток	Параллельный алгоритм							
		2 потока		4 потока		6 потоков		8 потоков	
		t	*S*	*t*	*S*	*t*	*S*	*t*	*S*

1000	0,12	0,06	1,86	0,04	3,16	0,03	3,90	0,03	4,68
2000	1,07	0,52	2,04	0,29	3,66	0,20	5,30	0,17	6,26
3000	4,20	2,64	1,59	2,00	2,10	1,87	2,24	1,75	2,40
4000	10,11	6,76	1,50	5,54	1,83	5,54	1,83	5,49	1,84
5000	19,75	13,53	1,46	11,37	1,74	11,28	1,75	11,28	1,75

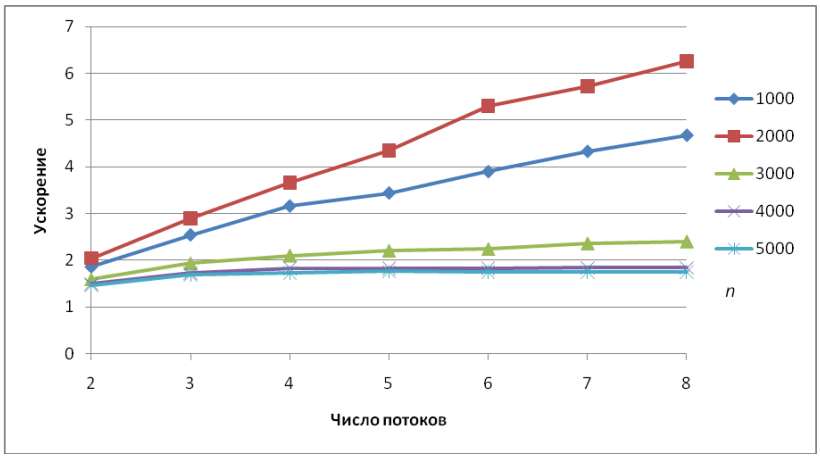


Рис. 3.6. Зависимость ускорения от числа потоков (разложение Холецкого)

Как показывают эксперименты, алгоритм хорошо масштабируется для матриц размера не более 2000. Действительно, при таком размере матрицы A ее нижний треугольник (а только он и нужен для проведения расчетов в силу симметрии матрицы) целиком помещается в кэш-память компьютера. При больших размерах матрицы возрастает число кэш-промахов. Сгладить данный эффект и получить масштабируемый алгоритм нам поможет, как и в случае метода Гаусса, блочный подход к обработке данных.

3.2.2. Блочный алгоритм

Рассмотрим теперь, как и в п. 3.1.5, вычислительную процедуру, основанную на идее разбиения матрицы на блоки и ориентированную на эффективную работу с кэш-памятью. Разложение осуществляется путем переписывания исходной матрицы элементами искомого фактора Холецкого сверху вниз по блокам.

Первый шаг блочного алгоритма заключается в следующем. Пусть мы определили размер блока как r, тогда исходную матрицу A можно представить в виде

$$A = \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix},$$

где A_{11} – подматрица матрицы A размера $r \times r$, A_{21} – размера $(n-r) \times r$, A_{22} – размера $(n-r) \times (n-r)$. Искомый фактор Холецкого также можно записать в блочном виде как

$$L = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix},$$

где L_{11}, L_{21}, L_{22} – соответствующего размера подматрицы фактора L.

Рассмотрим теперь связь между блоками фактора и исходной матрицы. Используя соотношение (3.10), запишем

$$\begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \cdot \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} = \begin{bmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{bmatrix},$$

откуда получим

$$A_{11} = L_{11}L_{11}^T, \tag{3.17}$$

$$A_{21} = L_{21}L_{11}^T, \tag{3.18}$$

$$A_{22} = L_{21}L_{21}^T + L_{22}L_{22}^T. \tag{3.19}$$

Используя данные матричные равенства, можно найти блоки L_{11}, L_{21} из искомого разложения.

Блок L_{11} может быть получен с помощью обычного неблочного алгоритма, изложенного в п. 3.2.1, т.к. формула (3.17) соответствует разложению Холецкого для матрицы A_{11}.

Далее, используя известный блок A_{21} и найденный блок L_{11}, из соотношения (3.18) можно найти блок L_{21}. Для этого потребуется решить r вспомогательных систем из r уравнений с одинаковой матрицей и разными правыми частями. Но так как матрица L_{11} является треугольной, то решение одной системы потребует лишь $O(r^2)$ операций. Всего же для нахождения блока L_{21} потребуется $O(r^3)$ операций с плавающей точкой.

Следующий шаг алгоритма состоит в вычислении редуцированной матрицы $\tilde{A}_{22}$, при котором используется ставший известным блок L_{21}, блок A_{22} и соотношение (3.19),

$$\tilde{A}_{22} = A_{22} - L_{21}L_{21}^T = L_{22}L_{22}^T. \qquad (3.20)$$

Как следует из данной формулы, фактор Холецкого для матрицы $\tilde{A}_{22}$ совпадает с искомым блоком L_{22}, и для его нахождения можно применить описанный алгоритм рекурсивно.

По построению матрица $\tilde{A}_{22}$ является симметричной положительно определенной, и при реализации алгоритма нет необходимости вычислять ее полностью, достаточо вычислить в соответствии с формулой (3.20) ее нижний треугольник.

Процесс блочной факторизации проиллюстрирован на рис. 3.7.

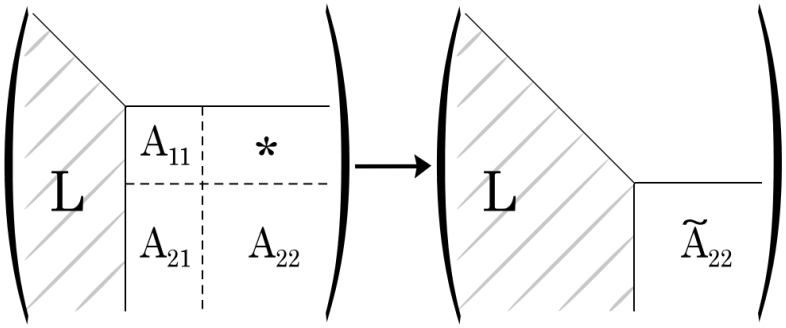


Рис. 3.7. Метод Холецкого, блочный алгоритм

Известно [8], что данная вычислительная процедура включает в себя

$$\frac{n^3}{3} + O\left(n^2\right)$$

операций, как и другие возможные реализации метода Холецкого; при этом вклад матричных операций в общее число действий (аналогично блочному *LU*-разложению) аппроксимируется величиной

$$1 - \frac{1}{N^2},$$

где $n = rN$. Отсюда следует, что матричные операции, в частности, в соответствии с формулой (3.20), будут составлять большую часть вычислений, и к реализации матричного умножения нужно подойти с особой тщательностью. В данном случае для проведения операции матричного умножения целесообразно воспользоваться блочной схемой умножения матриц, выбрав для этого свой размер блока, меньший *r*. Описание блочного алгоритма умножения матриц см., например, в [16].

Рассмотрим теперь способы параллельной реализации блочного метода Холецкого. После внимательного анализа данного алгоритма можно заключить, что распараллеливание возможно для следующих вычислительных процедур:

- вычисление блока L_{11} в соответствии с (3.17) (параллельная версия неблочного алгоритма);
- вычисление блока L_{21} в соответствии с (3.18) (параллельное решение набора систем линейных уравнений с треугольной матрицей);
- выполнение матричного умножения в соответствии с (3.20);
- выполнение обратного хода в соответствии с (3.15), (3.16).

В силу большей трудоемкости прямого хода алгоритма (т.е. собственно разложения) эффективность параллельного блочного алгоритма будет определяться эффективностью распараллеливания матричных операций, составляющих основную долю операция прямого хода.

3.2.3. Результаты вычислительных экспериментов

Сначала приведем время работы блочного алгоритма Холецкого и его исходного неблочного варианта (время работы *t* указано в секундах, столбец, отмеченный прочерком, соответствует неблочному алгоритму).

Табл. 3.6. Сравнение блочного и неблочного разложения Холецкого

n	Время работы *t*, с					
	---	*r*=10	*r*=20	*r*=50	*r*=100	*r*=200
1000	0,11	0,28	0,21	0,16	0,16	0,15
2000	1,05	2,62	1,99	1,39	1,23	1,09
3000	4,16	9,81	7,25	4,84	4,17	3,71
4000	10,00	25,05	18,22	11,74	10,00	9,63
5000	19,59	49,23	35,82	23,24	20,39	20,22

Из таблицы следует, что блочный алгоритм значительно проигрывает неблочному при малых размерах блока факторизации, и почти сравнивается по скорости при увеличении размера блока. Данный факт объясняется тем, что при реализации формулы (3.20) мы использовали алгоритм умножения матриц по определению. Посмотрим, что изменится, если применить блочный алгорим матричного умножения, который более эффективно использует кэш-память.

Для выбора оптимального размера блока при выполнении матричного умножения нами были проведены эксперименты при размерах блока от

10×10 до 200×200 с шагом 5. Лучшие результаты приведены в таблице ниже.

Табл. 3.7. Сравнение блочного и неблочного разложения Холецкого

n	Время работы *t*			
	---	*r*=50 (25×50)	*r*=100 (25×50)	*r*=200 (10×200)
1000	0,12	0,11	0,12	0,13
2000	1,07	0,78	0,80	0,83
3000	4,20	2,57	2,61	2,64
4000	10,11	5,98	5,99	6,10
5000	19,75	12,57	11,50	11,84

Ситуация значительно улучшилась – теперь блочный алгоритм показывает такое же быстродействие на матрицах малого размера (которые полностью умещаются в кэш-память), и значительно обгоняет неблочный на матрицах большого размера. При этом время, затрачиваемое алгоритмом, является примерно одинаковым для любых размеров блоков факторизации *r* (отличие есть лишь в сотых долях секунды).

Осталось убедиться в том, что наше решение будет масштабируемым – проведем разложение с использованием параллельного блочного алгоритма с размером блока факторизации *r*=200 и блоком матричного умножениия 10×200 в соответствии с описанной схемой распараллеливания. Как уже обсуждалось ранее, время работы последовательной программы и параллельной программы, запущенной в один поток, будет разным. Этим объясняются разные времена работы алгоритмов, приведенных в табл. 1.8 и 1.9.

Табл. 3.8. Параллельное блочное разложение Холецкого

n	1 поток	Параллельный алгоритм							
		2 потока		4 потока		6 потоков		8 потоков	
		t	*S*	*t*	*S*	*t*	*S*	*t*	*S*
1000	0,14	0,08	2,15	0,05	3,77	0,05	4,98	0,03	4,98
2000	0,87	0,48	1,90	0,28	3,52	0,22	4,84	0,19	5,80
3000	2,79	1,50	1,95	0,83	3,55	0,62	4,94	0,51	6,10
4000	6,43	3,35	1,94	1,90	3,51	1,37	4,91	1,11	6,03
5000	12,39	6,44	1,94	3,56	3,52	2,54	4,87	2,04	6,05

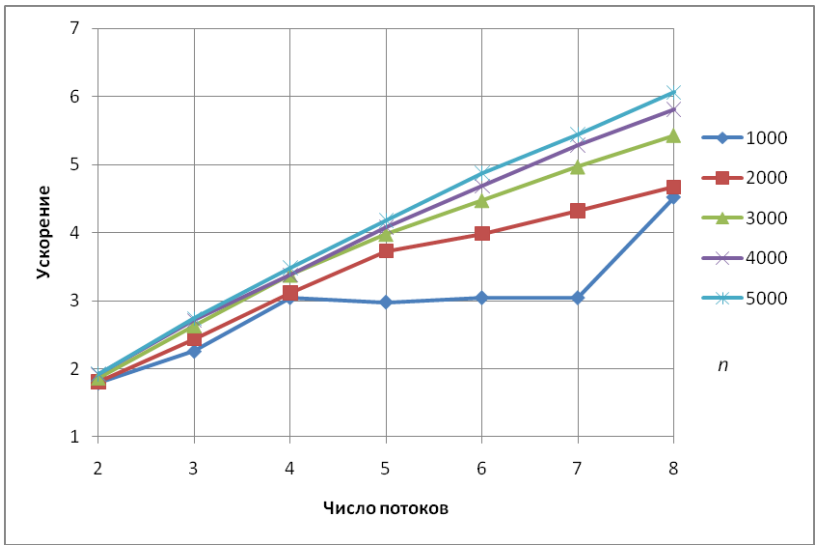


Рис. 3.8. Зависимость ускорения от числа потоков
(блочное разложение Холецкого)

Как показывают эксперименты, блочный алгоритм хорошо масштабируется – ускорения достигает значения 6 для 8-и поточной программы.

3.3. Метод прогонки

Одним из частных (но, тем не менее, часто встречающихся) видов системы (3.2) является система

$$Ax=f,$$

с ленточной матрицей A. Матрица A называется *ленточной*, когда все ее ненулевые элементы находятся вблизи главной диагонали, т.е. $a_{ij}=0$, если $|i-j|>l$, где $l<n$. Число l называется *шириной ленты*. Примером является трехдиагональная матрица (при $l=1$) вида:

$$A=\begin{bmatrix} c_1 & b_1 & 0 & \dots & 0 \\ a_2 & c_2 & b_2 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & a_{n-1} & c_{n-1} & b_{n-1} \\ 0 & \dots & 0 & a_n & c_n \end{bmatrix}$$

Матрицы такого вида возникают, например, при решении задачи сплайн-интерполяции [1].

Рассмотрим *метод прогонки*, применимый для решения систем с трехдиагональной матрицей. Предположим, что имеет место соотношение

$$x_i=\alpha_{i+1}x_{i+1}+\beta_{i+1} \tag{3.21}$$

с неопределенными коэффициентами α_{i+1} и β_{i+1}, и подставим выражение $x_{i-1}=\alpha_i x_i+\beta_i$ в i-е уравнение системы:

$$(\alpha_i a_i+c_i)x_i+b_i x_{i+1}=f_i-a_i\beta_i.$$

Сравнивая полученное выражение с (3.21), находим

$$\begin{aligned}\alpha_{i+1}&=\frac{-b_i}{a_i\alpha_i+c_i},\ i=2,\ldots,n-1,\\ \beta_{i+1}&=\frac{f_i-a_i\beta_i}{a_i\alpha_i+c_i},\ i=2,\ldots,n-1.\end{aligned} \tag{3.22}$$

Из первого уравнения системы

$$c_1x_1+b_1x_2=f_1$$

находим

$$\alpha_2=-b_1/c_1,\ \beta_2=f_1/c_1.$$

Зная α_2,β_2 и переходя от i к $i+1$ в формулах (3.22), определим α_i,β_i для всех $i=3,\ldots,n$.

Определим x_n из последнего уравнения системы и условия (3.21) при $i=n-1$.

$$\begin{aligned}x_{n-1}&=\alpha_n x_n+\beta_n\\ a_n x_{n-1}+c_n x_n&=f_n\end{aligned} \tag{3.23}$$

Решив систему из двух уравнений с двумя неизвестными, находим

$$x_n=\frac{f_n-a_n\beta_n}{a_n\alpha_n+c_n}.$$

После того, как значение x_n найдено, определяем все остальные значения x_i в обратном порядке, используя формулу (3.21). Соберем теперь все формулы прогонки и запишем их в порядке применения.

Прямой ход:

$$\alpha_2=-b_1/c_1,\ \alpha_{i+1}=\frac{-b_i}{a_i\alpha_i+c_i},\ i=2,\ldots,n-1, \tag{3.24}$$

$$\beta_2=f_1/c_1,\ \beta_{i+1}=\frac{f_i-a_i\beta_i}{a_i\alpha_i+c_i},\ i=2,\ldots,n-1.$$

Обратный ход:

$$x_n=\frac{f_n-a_n\beta_n}{a_n\alpha_n+c_n},\ x_i=\alpha_{i+1}x_{i+1}+\beta_{i+1},\ i=n-1,\ldots,1. \tag{3.25}$$

Известно [4], что для вычислительной устойчивости метода прогонки необходимо выполнение условия *диагонального преобладания*

$$|c_1|\ge|b_1|,\ |c_n|\ge|a_n|,$$

$$|c_i|>|a_i|+|b_i|,\ i=2,\ldots,n-1.$$

Оценим трудоемкость метода прогонки. При выполнении прямого хода по формулам (3.24) потребуется $8(n-2)+2$ операций. Для выполнения обратного хода по формулам (3.25) потребуется $2(n-1)+5$ операций. Таким образом, общее число операций можно оценить величиной

$$10n+O(1), \tag{3.26}$$

а время решения системы методом прогонки при больших n будет определяться как

$$T_1=10n\tau,$$

где τ – время выполнения одной операции.

3.3.1. Метод встречной прогонки и его распараллеливание

Рассмотренный в предыдущем пункте метод прогонки, определяемый соотношениями (3.24) и (3.25), при котором определение x_i происходит последовательно справа налево, называют *правой прогонкой*. Аналогично выписываются формулы *левой прогонки*.

Прямой ход:

$$\begin{aligned}&\xi_n=-a_n/c_n,\ \xi_i=\frac{-a_i}{c_i+b_i\xi_{i+1}},\ i=n-1,\ldots,2;\\ &\eta_n=f_n/c_n,\ \eta_i=\frac{f_i-b_i\eta_{i+1}}{c_i+b_i\xi_{i+1}},\ i=n-1,\ldots,2.\end{aligned} \tag{3.27}$$

Обратный ход:

$$x_1 = \frac{f_1 - b_1\eta_2}{b_1\xi_2 + c_1},\ x_{i+1}=\xi_{i+1}x_i+\eta_{i+1},\ i=1,\ldots,n-1. \qquad (3.28)$$

В самом деле, предполагая, что $x_{i+1}=\xi_{i+1}x_i+\eta_{i+1}$, исключим из i-го уравнения системы переменную x_{i+1}, получим

$$a_ix_{i-1}+c_ix_i+b_i(\xi_{i+1}x_i+\eta_{i+1})=f_i,$$

или

$$x_i = \frac{-a_i}{c_i + b_i\xi_{i+1}}x_{i-1} + \frac{f_i - b_i\eta_{i+1}}{c_i + b_i\xi_{i+1}}.$$

Сравнивая с формулой $x_i=\xi_ix_{i-1}+\eta_i$, получим расчетные формулы (3.27). Значение x_1 находим из первого уравнения и условия $x_2=\xi_2x_1+\eta_2$, затем, используя условие $x_i=\xi_ix_{i-1}+\eta_i$ и известные коэффициенты ξ_i, η_i, можно найти все остальные значения неизвестных.

Нетрудно видеть, что трудоемкость левой прогонки составляет также $10n+O(1)$.

Комбинация левой и правой прогонок дает метод *встречной прогонки*, который допускает распараллеливание на два потока. Разделим систему между двумя потоками – первый будет оперировать уравнениями с номерами $1\le i\le p$, второй – уравнениями $p\le i\le n$, где $p = \lceil n/2 \rceil$.

При параллельном решении системы в первом потоке по формулам (3.22) вычисляются прогоночные коэффициенты α_i, β_i, при $1\le i\le p$, а во втором потоке по формулам (3.27) находятся ξ_i, η_i, при $p\le i\le n$. При $i=p$ проводится сопряжение решений в форме (3.25) и (3.28): находим значение x_p из системы

$$\begin{cases} x_p = \alpha_{p+1}x_{p+1} + \beta_{p+1} \\ x_{p+1} = \xi_{p+1}x_p + \eta_{p+1} \end{cases}.$$

Найдя указанное значение, в первом потоке можно по формуле (3.25) найти все x_i, при $1\le i<p$, а во втором – по формуле (3.28) – все x_i, при $p<i\le n$.

Трудоемкость метода параллельной встречной прогонки можно оценить как

$$T_2 = 5n\tau + \delta,$$

где δ – время, необходимое на организацию и закрытие параллельной секции. Следует отметить, что расчеты и при прямом, и при обратном ходе производятся независимо, теоретическое ускорение здесь должно быть равно двум.

3.3.2. Параллельный вариант метода прогонки

Рассмотрим теперь схему распараллеливания метода прогонки при использовании p потоков. Пусть нужно решить трехдиагональную систему линейных уравнений

$$a_i x_{i-1} + c_i x_i + b_i x_{i+1} = f_i,\ i=1,\ldots,n,\ x_0=x_{n+1}=0, \tag{3.29}$$

с использованием p параллельных потоков.

Применим блочный подход к разделению данных: пусть каждый поток обрабатывает $m = \lfloor n/p \rfloor$ строк матрицы A, т.е. k-й поток обрабатывает строки с номерами $1+(k-1)m \le i \le km$. Для простоты изложения мы предполагаем, что число уравнений в системе кратно числу потоков, в общем случае изменится только число уравнений в последнем потоке. Ниже представлено разделение данных для трех потоков в случае системы из 12 уравнений.

$$\left(\begin{array}{cccc:cccc:cccc|c}
c_1 & b_1 & & & & & & & & & & & f_1 \\
a_2 & c_2 & b_2 & & & & & & & & & & f_2 \\
 & a_3 & c_3 & b_3 & & & & & & & & & f_3 \\
 & & a_4 & c_4 & b_4 & & & & & & & & f_4 \\ \hdashline
 & & & a_5 & c_5 & b_5 & & & & & & & f_5 \\
 & & & & a_6 & c_6 & b_6 & & & & & & f_6 \\
 & & & & & a_7 & c_7 & b_7 & & & & & f_7 \\
 & & & & & & a_8 & c_8 & b_8 & & & & f_8 \\ \hdashline
 & & & & & & & a_9 & c_9 & b_9 & & & f_9 \\
 & & & & & & & & a_{10} & c_{10} & b_{10} & & f_{10} \\
 & & & & & & & & & a_{11} & c_{11} & b_{11} & f_{11} \\
 & & & & & & & & & & a_{12} & c_{12} & f_{12}
\end{array}\right)$$

В пределах полосы матрицы, обрабатываемой k-м потоком, можно организовать исключение поддиагональных элементов матрицы (прямой ход метода). Для этого осуществляется вычитание строки i, умноженной на константу a_{i+1}/c_i, из строки $i+1$ с тем, чтобы результирующий коэффициент при неизвестной x_i в $(i+1)$-й строке оказался нулевым.

Если исключение первым потоком поддиагональных переменных не добавит в матрицу новых коэффициентов, то исключение поддиагональных элементов в остальных потоках приведет к возникновению столбца отличных от нуля коэффициентов: во всех блоках (кроме первого) число ненулевых элементов в строке не изменится, но изменится структура уравнений.

Модификации также подвергнутся элементы вектора правой части. Матрица (3.30) иллюстрирует данный процесс, чертой сверху отмечены элементы, которые будут модифицированы.

$$\left(\begin{array}{cccccccccccc|c}
c_1 & b_1 & & & & & & & & & & & f_1 \\
 & \bar{c}_2 & b_2 & & & & & & & & & & \bar{f}_2 \\
 & & \bar{c}_3 & b_3 & & & & & & & & & \bar{f}_3 \\
 & & & \bar{c}_4 & b_4 & & & & & & & & \bar{f}_4 \\
\hdashline
 & & & a_5 & c_5 & b_5 & & & & & & & f_5 \\
 & & & d_6 & & \bar{c}_6 & b_6 & & & & & & \bar{f}_6 \\
 & & & d_7 & & & \bar{c}_7 & b_7 & & & & & \bar{f}_7 \\
 & & & d_8 & & & & \bar{c}_8 & b_8 & & & & \bar{f}_8 \\
\hdashline
 & & & & & & & a_9 & c_9 & b_9 & & & f_9 \\
 & & & & & & & d_{10} & & \bar{c}_{10} & b_{10} & & \bar{f}_{10} \\
 & & & & & & & d_{11} & & & \bar{c}_{11} & b_{11} & \bar{f}_{11} \\
 & & & & & & & d_{12} & & & & \bar{c}_{12} & \bar{f}_{12}
\end{array}\right) \qquad (3.30)$$

Затем выполняется обратный ход алгоритма – каждый поток исключает наддиагональные элементы, начиная с последнего.

$$\left(\begin{array}{cccccccccccc|c}
c_1 & & & g_1 & & & & & & & & & \bar{f}_1 \\
 & \bar{c}_2 & & g_2 & & & & & & & & & \bar{f}_2 \\
 & & \bar{c}_3 & g_3 & & & & & & & & & \bar{f}_3 \\
 & & & \bar{c}_4 & b_4 & & & & & & & & \bar{f}_4 \\
\hdashline
 & & & \bar{a}_5 & c_5 & & & & g_5 & & & & \bar{f}_5 \\
 & & & \bar{d}_6 & & \bar{c}_6 & & & g_6 & & & & \bar{f}_6 \\
 & & & \bar{d}_7 & & & \bar{c}_7 & & g_7 & & & & \bar{f}_7 \\
 & & & d_8 & & & & \bar{c}_8 & b_8 & & & & \bar{f}_8 \\
\hdashline
 & & & & & & & \bar{a}_9 & c_9 & & & & \bar{f}_9 \\
 & & & & & & & \bar{d}_{10} & & \bar{c}_{10} & & & \bar{f}_{10} \\
 & & & & & & & \bar{d}_{11} & & & \bar{c}_{11} & & \bar{f}_{11} \\
 & & & & & & & d_{12} & & & & \bar{c}_{12} & \bar{f}_{12}
\end{array}\right)$$

После выполнения обратного хода матрица стала блочной. Исключим из нее внутренние строки каждой полосы, в результате получим систему

уравнений относительно части исходный неизвестных, частный вид которой представлен ниже.

$$\left(\begin{array}{ccccccc}
c_1 & g_1 & & & & & \bar{f}_1 \\
 & \bar{c}_4 & b_4 & & & & \bar{f}_4 \\
 & \bar{a}_5 & c_5 & g_5 & & & \bar{f}_5 \\
 & & d_8 & \bar{c}_8 & b_8 & & \bar{f}_8 \\
 & & & \bar{a}_9 & c_9 & & \bar{f}_9 \\
 & & & & d_{12} & \bar{c}_{12} & \bar{f}_{12}
\end{array}\right)$$

Данная система будет содержать $2p$ уравнений, и будет трехдиагональной. Ее можно решить последовательным методом прогонки. После того, как эта система будет решена, станут известны значения неизвестных на границах полос разделения данных. Далее можно за один проход найти значения внутренних переменных.

Рассмотренный способ распараллеливания уже дает хорошие результаты, но можно использовать лучшую стратегию исключения неизвестных. Прямой ход нового алгоритма будет таким же, а во время обратного хода каждый поток исключает наддиагональные элементы, начиная со своего предпоследнего, и заканчивая последним для предыдущего потока. Матрица (3.31) иллюстрирует данный процесс.

$$\left(\begin{array}{ccccccccc}
c_1 & & & g_1 & & & & & \bar{f}_1 \\
 & \bar{c}_2 & & g_2 & & & & & \bar{f}_2 \\
 & & \bar{c}_3 & b_3 & & & & & \bar{f}_3 \\
 & & & \bar{c}_4 & & & g_4 & & \bar{f}_4 \\
\hdashline
 & & & \bar{a}_5 & c_5 & & g_5 & & \bar{f}_5 \\
 & & & \bar{d}_6 & & \bar{c}_6 & g_6 & & \bar{f}_6 \\
 & & & d_7 & & & \bar{c}_7 \; b_7 & & \bar{f}_7 \\
 & & & d_8 & & & \bar{c}_8 & g_8 & \bar{f}_8 \\
\hdashline
 & & & & & & \bar{a}_9 & c_9 \; g_9 & \bar{f}_9 \\
 & & & & & & \bar{d}_{10} & \bar{c}_{10} \; g_{10} & \bar{f}_{10} \\
 & & & & & & d_{11} & \bar{c}_{11} \; b_{11} & \bar{f}_{11} \\
 & & & & & & d_{12} & \bar{c}_{12} & \bar{f}_{12}
\end{array}\right) \qquad (3.31)$$

Изменение порядка исключения переменных в обратном ходе алгоритма приводит к тому, что можно сформировать вспомогательную задачу меньшего размера. Исключим из матрицы все строки каждой полосы, кроме

последней, в результате получим систему уравнения относительно части исходный неизвестных, частный вид которой представлен ниже.

$$\left|\begin{array}{ccc|c} \bar{c}_4 & g_4 & & \bar{f}_4 \\ d_8 & \bar{c}_8 & g_8 & \bar{f}_8 \\ & d_{12} & \bar{c}_{12} & \bar{f}_{12} \end{array}\right|$$

Даная система будет содержать всего p уравнений, и также будет трехдиагональной. Ее можно решить последовательным методом прогонки (так как для систем с общей памятью число потоков p будет не слишком велико, применять для решения вспомогательной системы даже параллельный метод встречной прогонки нецелесообразно). После того, как эта система будет решена, станут известны значения неизвестных на нижних границах полос разделения данных. Далее можно за один проход найти значения внутренних переменных в каждом потоке.

Оценим трудоемкость рассмотренного параллельного варианта метода прогонки. В соответствии с введенными ранее обозначениями n есть порядок решаемой системы линейных уравнений, а p, $p<n$, обозначает число потоков. Тем самым, матрица коэффициентов A имеет размер $n \times n$ и, соответственно, $m=n/p$ есть размер полосы матрицы A на каждом процессоре.

При выполнении прямого хода алгоритма на каждой итерации каждый процессор должен осуществить исключение в пределах своей полосы поддиагональных элементов (что требует $8(m-1)$ операций) и наддиагональных элементов (что требует $7m$ операций).

Затем следует произвести сборку вспомогательной трехдиагональной системы уравнений в одном потоке, и осуществить ее решение методом прогонки. В соответствии с оценкой (3.26) затраты на выполнение этого чисто последовательного этапа составят порядка $10p$ операций.

На следующем этапе алгоритма каждый процессор выполняет обратный ход алгоритма, который потребует $5(m-1)$ операций. Таким образом, общую трудоемкость параллельного метода прогонки можно оценить как

$$T_p = 20m + 10p\,. \tag{3.32}$$

Как результат выполненного анализа, показатели ускорения и эффективности параллельного варианта метода прогонки могут быть определены при помощи соотношений следующего вида:

$$S_p = \frac{T_1}{T_p} = \frac{10n}{20\dfrac{n}{p} + 10p} = p\frac{10n}{20n + p^2}; \tag{3.33}$$

$$E_p = \frac{S_p}{p} = \frac{10n}{20n + p^2}.$$

Из приведенных соотношений видно, что в случае решения системы уравнений с большим числом неизвестных, при котором $p << n$, показатели ускорения и эффективности будут определяться как

$$S_p \approx \frac{p}{2},\ E_p \approx 0.5. \tag{3.34}$$

3.3.3. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного варианта метода прогонки для решения трехдиагональных систем линейных уравнений проводились на аппаратуре, технические характеристики которой указаны во введении. Также следует отметить, что для простоты написания параллельных программ нами рассматривались задачи размера, кратного 8, чтобы размер блоков был одинаков для всех потоков. С целью формирования матрицы с диагональным преобладанием элементы на побочных диагоналях матрицы генерировались в диапазоне от 0 до 100, а элемент на главной диагонали был равен удвоенной сумме элементов в строке.

Сначала приведем результаты сравнения реализованной нами правой и левой прогонки со специальной функцией библиотеки Intel MKL, решающей трехдиагональные системы с диагональным преобладанием. Результаты, отражающие зависимость времени решения задачи *T* от ее размера *N*, приведены на рис. 3.9.

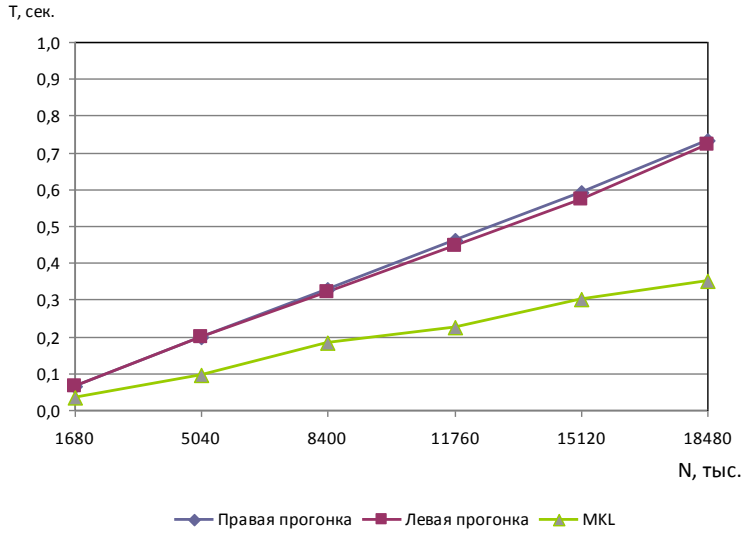


Рис. 3.9. Сравнение с библиотекой MKL

Результаты экспериментов демонстрируют двукратное отставание от библиотеки Intel MKL по времени, что является неплохим показателем.

Далее рассмотрим эффект, который дает использование встречной прогонки в двух потоках: проведем сравнение параллельной встречной прогонки с правой и левой прогонками. Результаты, отражающие зависимость ускорения по отношению к правой и левой прогонкам от размера задачи N приведены на рис. 3.10.

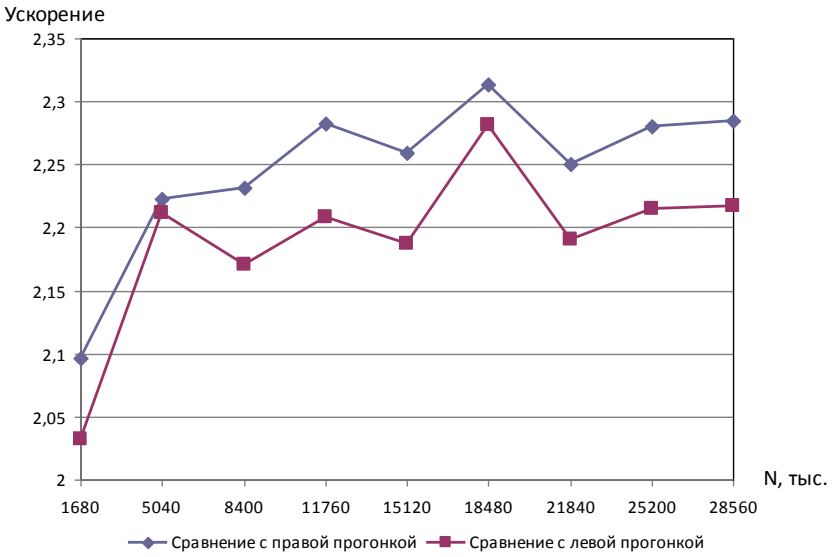


Рис. 3.10. Зависимость ускорения встречной прогонки от размера задачи

Перед тем, как переходить к экспериментам с большим числом потоков, выясним, какой из алгоритмов является наиболее быстрым в двухпоточной программе. Для сравнения будем использовать метод встречной прогонки, и два способа распараллеливания, описанных в п. 3.3.2.

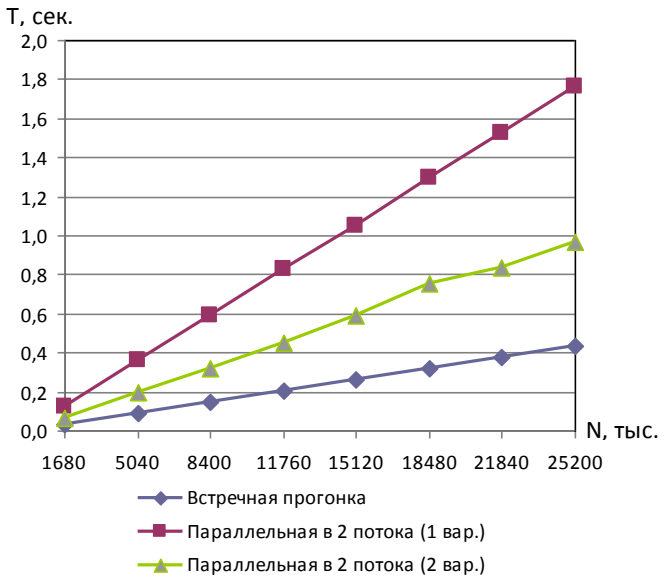


Рис. 3.11. Время работы различных вариантов метода прогонки

Как и следовало ожидать, наиболее быстродействующим оказался метод встречной прогонки. Однако встречную прогонку можно использовать только в двух потоках, для распараллеливания на большее число потоков нужно использовать вторую модификацию алгоритма, описанную в п. 3.3.2, которая обладает большей трудоемкостью, но и большей масштабируемостью. Ниже приведены результаты вычислительных экспериментов, полученные при использовании данной модификации параллельного метода прогонки.

Таблица 3.9. Результаты экспериментов (параллельная прогонка)

n, тыс.	1 поток	Параллельный алгоритм							
		2 потока		4 потока		6 потоков		8 потоков	
		T	*S*	*T*	*S*	*T*	*S*	*T*	*S*
1680	0,06	0,07	0,94	0,03	2,03	0,03	2,10	0,03	2,03
5040	0,20	0,20	1,00	0,11	1,83	0,09	2,29	0,09	2,14

8400	0,32	0,32	0,99	0,17	1,85	0,15	2,18	0,14	2,26
11760	0,45	0,45	0,98	0,23	1,91	0,20	2,25	0,20	2,20
15120	0,57	0,59	0,97	0,31	1,84	0,26	2,18	0,27	2,16
18480	0,72	0,76	0,95	0,41	1,78	0,32	2,25	0,31	2,31
21840	0,83	0,84	0,99	0,45	1,84	0,38	2,20	0,37	2,22
25200	0,96	0,97	0,99	0,53	1,80	0,43	2,25	0,44	2,19

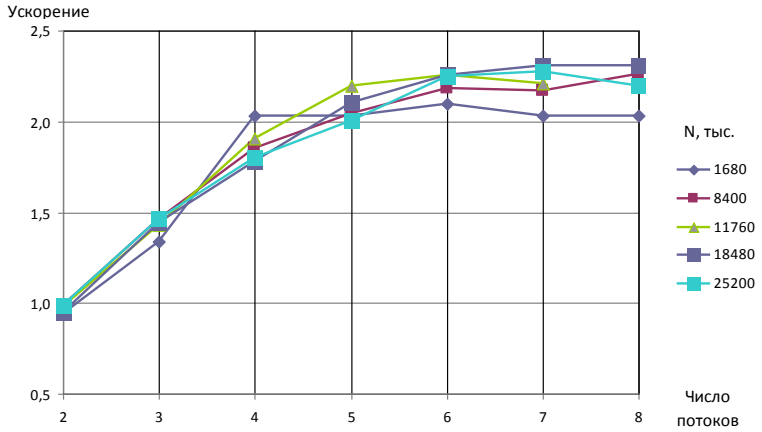


Рис. 3.12. Зависимость ускорения от числа потоков для метода прогонки

На рис. 3.12 приведен график зависимости ускорения от числа потоков при использовании модификации алгоритма из п. 1.2.2. Видно, что при $p<6$ ускорение в целом соответствует оценке (3.34), полученной с помощью подсчета числа операций, необходимых для работы метода.

3.4. Метод редукции

Вернемся опять к рассмотрению системы линейных уравнений

$$Ax=f,$$

с трехдиагональной матрицей A, удовлетворяющей условию строгого диагонального преобладания. В п. 3.3 мы уже познакомились с методом прогонки для решения подобных систем. Анализ формул метода прогонки показывает, что в некоторых случаях метод может давать большую погрешность. Потенциальным источником погрешности являются формулы для

вычисления «прогоночных» коэффициентов, которые содержат операцию деления на разность близких по значению величин.

Метод редукции, который будет рассмотрен ниже, свободен от этого недостатка и, кроме того, при реализации на современных вычислительных системах он показывает большую эффективность по сравнению с методом прогонки. Основное ограничение метода редукции состоит в том, что он применим лишь для матриц размера, равного степени двойки, в отличие от метода прогонки, применимого для матриц любого размера.

Следует отметить, что существует обобщение метода редукции на случай блочных трехдиагональных матриц (см. [4]). В этом случае число блоков должно быть равно степени двойки, а основная идея метода остается неизменной (лишь операция деления заменяется на операцию умножения на обратную матрицу, что подразумевает хорошую обратимость блоков матрицы A).

3.4.1. Последовательный алгоритм

Для удобства последующих обозначений запишем систему в виде

$$a_i x_{i-1} + c_i x_i + b_i x_{i+1} = f_i,\ 1 \le i \le n-1, \qquad x_0 = 0,\ x_n = 0 \tag{3.35}$$

Идея метода редукции состоит в последовательном исключении из системы (3.35) неизвестных сначала с нечетными номерами, затем с номерами, кратными 2 (но не кратными 4), и т.д., (*прямой ход*) и восстановлении значений нечетных переменных на основании известных значений переменных с четными номерами (*обратный ход*).

Выпишем три идущие подряд уравнения системы (3.35) с номерами $i-1$, i, $i+1$, где i – четное число

$$a_{i-1} x_{i-2} + c_{i-1} x_{i-1} + b_{i-1} x_i = f_{i-1}$$

$$a_i x_{i-1} + c_i x_i + b_i x_{i+1} = f_i$$

$$a_{i+1} x_i + c_{i+1} x_{i+1} + b_{i+1} x_{i+2} = f_{i+1}$$

Умножая первое из указанных уравнений на коэффициент $\alpha_i^{(1)} = -a_i / c_{i-1}$, последнее – на коэффициент $\beta_i^{(1)} = -b_i / c_{i+1}$ и складывая полученные уравнения со вторым, получим

$$a_i^{(1)} x_{i-2} + c_i^{(1)} x_i + b_i^{(1)} x_{i+2} = f_i^{(1)},\ i = 2, 4, 6, \ldots, n-2 \tag{3.36}$$

где $a_i^{(1)} = \alpha_i^{(1)} a_{i-1}$, $b_i^{(1)} = \beta_i^{(1)} b_{i+1}$, $c_i^{(1)} = \alpha_i^{(1)} b_{i-1} + c_i + \beta_i^{(1)} a_{i+1}$, $f_i^{(1)} = \alpha_i^{(1)} f_{i-1} + f_i + \beta_i^{(1)} f_{i+1}$. Если предположить, что неизвестные с четными номерами найдены из системы (3.36), то остальные неизвестные с нечетными номерами можно найти по формулам

$$x_i = \frac{f_i - a_i x_{x-1} - b_i x_{i+1}}{c_i},\ i=1, 3, 5, \ldots, n-1.$$

Очевидно, что систему (3.36) можно решить, используя описанный процесс рекурсивно (так как число переменных в ней будет 2^{n-1}).

Таким образом, на втором шаге алгоритма из системы будут (3.36) исключены переменные с номерами, кратными 2, но не кратными 4. В результате l-го шага процесса исключения получим систему

$$a_i^{(l)} x_{i-2^l} + c_i^{(l)} x_i + b_i^{(l)} x_{i+2^l} = f_i^{(l)},\ i = 2^l, 2\cdot 2^l, 3\cdot 2^l, \ldots, n-2^l \qquad (3.37)$$

где

$$\alpha_i^{(l)} = \frac{-a_i^{(l-1)}}{c_{i-2^{l-1}}^{(l-1)}},\ \beta_i^{(l)} = \frac{-b_i^{(l-1)}}{c_{i+2^{l-1}}^{(l-1)}},\ a_i^{(l)} = \alpha_i^{(l)} a_{i-2^{l-1}}^{(l-1)},\ b_i^{(l)} = \beta_i^{(l)} b_{i+2^{l-1}}^{(l-1)}$$

$$c_i^{(l)} = \alpha_i^{(l)} b_{i-2^{l-1}}^{(l-1)} + c_i^{(l)} + \beta_i^{(l)} a_{i+2^{l-1}}^{(l-1)},\ f_i^{(l)} = \alpha_i^{(l)} f_{i-2^{l-1}}^{(l-1)} + f_i^{(l)} + \beta_i^{(l)} f_{i+2^{l-1}}^{(l-1)}, \qquad (3.38)$$

$$i = 2^l, 2\cdot 2^l, 3\cdot 2^l, \ldots, n-2^l,\ l \geq 1.$$

Здесь использованы обозначения $a_i^{(0)} = a_i$, $b_i^{(0)} = b_i$, $c_i^{(0)} = c_i$, $f_i^{(0)} = f_i$.

Графическая иллюстрация схемы исключения переменных для случая n=8 приведена на рис. 3.13. При l=0 система уравнений совпадает с исходной и содержит все неизвестные. На рис. 3.13 номера неизвестных переменных, входящих в систему, отмечены в кружочках. На первом этапе (l=1) происходит исключение неизвестных с нечетными номерами, в результате чего получаем систему, содержащую только четные переменные. Стрелки указывают, какие переменные участвовали в исключении. На последнем этапе (l=2) остается только одно уравнение, связывающее x_0, x_4 и x_8.

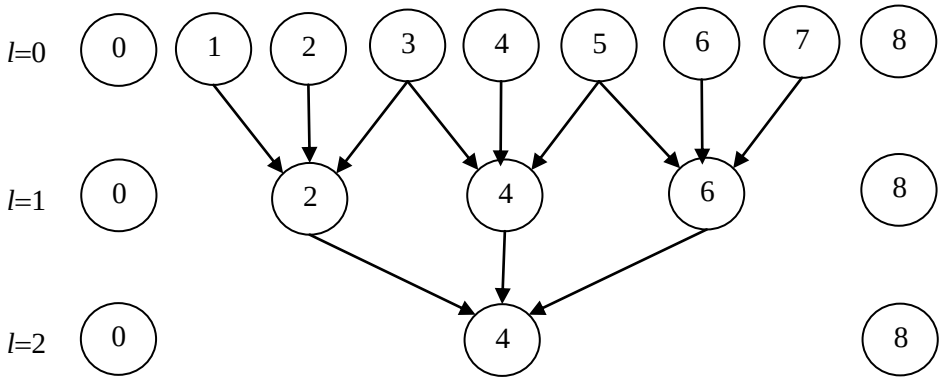


Рис. 3.13. Схема исключения переменных при n=8

В общем случае процесс исключения закончится на (q–1)-м шаге, $q = \log_2 n$, когда система (3.37) будет состоять из одного уравнения относительно переменной $x_{n/2} = x_{2^{q-1}}$. Из этого уравнения, учитывая $x_0 = x_n = 0$, найдем

$$x_{2^{q-1}} = \frac{f_{2^{q-1}}^{(q-1)}}{c_{2^{q-1}}^{(q-1)}}.$$

Остальные переменные определяются по формулам

$$x_i = \frac{f_i^{(l)} - a_i^{(l)} x_{i-2^l} - b_i^{(l)} x_{i+2^l}}{c_i^{(l)}},\ i = 2^l, 3\cdot 2^l, 5\cdot 2^l, ..., n - 2^l. \qquad (3.39)$$

Для иллюстрации на рис. 3.14 приведена схема вычисления значений неизвестных при n=8. Согласно данной схеме переменные пересчитываются последовательно снизу вверх (пересчитываемые на каждом шаге переменные выделены, стрелками от них указаны переменные, значения которых используются при вычислении согласно формуле (3.39)).

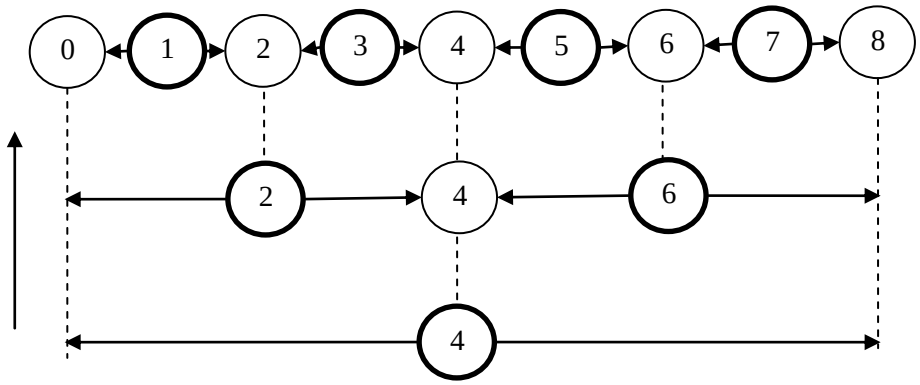


Рис. 3.14. Схема вычисления неизвестных при n=8

Итак, прямой ход метода редукции состоит в вычислении по формулам (3.38) коэффициентов $a_i^{(l)}, b_i^{(l)}, c_i^{(l)}, f_i^{(l)}$ для l=1, 2,…, q–1, а обратный ход состоит в нахождении решения по формуле (3.39) для l= q–1, q–2,…, 0.

Общее число операций, требующееся для решения системы уравнений методом редукции, – 12n сложений, 8n умножений, 3n делений [5], т.е. метод является примерно таким же по трудоемкости, как и метод прогонки.

3.4.2. Параллельный алгоритм

Сформулируем теперь параллельный вариант метода редукции.

Анализ вычислительной схемы метода редукции показывает, что каждый последующий шаг прямого и обратного хода зависит от предыдущего. Зависимости по данным для этих операций на примере задачи при n=8 отражены на рис. 3.13 (прямой ход) и рис. 3.14 (обратный ход). Таким образом, распараллелить целиком прямой или обратный хода не получается.

С другой стороны, исключение неизвестных на отдельном шаге прямого хода можно проводить независимо, т.к. в этом случае зависимостей по данным нет. Аналогично может быть распараллелен отдельный шаг обратного хода, т.к. значения неизвестных находятся независимо.

Например, в задаче, проиллюстрированной на рис. 3.13 и рис. 3.14, на первом шаге прямого хода можно параллельно вычислить коэффициенты редуцированной системы уравнений относительно переменных x_2, x_4, x_6; второй шаг прямого хода выполняется последовательно. А на первом шаге обратного хода можно параллельно вычислить значения переменных x_2, x_6, а на втором шаге – значения x_1, x_3, x_5, x_7.

В заключение отметим, что при реализации рассмотренного параллельного алгоритма в системах с общей памятью не должен возникать эффект «гонки данных», связанный с доступом потоков к одинаковым областям памяти, т.к. распараллеливаемые отдельные шаги прямого и обратного хода не имеют зависимостей по данным.

3.4.3. Результаты вычислительных экспериментов

Для проведения вычислительных экспериментов выберем иную стратегию – многократное решение системы уравнений с одинаковой матрицей относительно небольшого размера и различными правыми частями (в отличие от предыдущего пункта, в котором решалась одна задача с матрицей большого размера). Решение указанной последовательности задач возникает, например, при численном решении дифференциальных уравнений в частных производных сеточными методами.

Вычислительные эксперименты для оценки эффективности метода редукции проводились на аппаратуре, технические характеристики которой указаны во введении. Для простоты написания параллельных программ нами рассматривались задачи размера, кратного степени двух, и решалась серия из 10 тыс. задач с одинаковой матрицей и разными правыми частями (подробная постановка данной задачи описана в лабораторной работе «Дифференциальные уравнения в частных производных»). Здесь же отметим лишь некоторую специфику данной серии задач. Матрица СЛАУ не зависит от номера задачи в серии, причем на диагонали расположены одинаковые числа, т.е. в системе (3.35) $a_i = a$, $b_i = b$, $c_i = c$; а правая часть j-й задачи нелинейно зависит от решения (j–1)-й задачи (правая часть первой задачи в серии – известна).

Теперь перейдем к результатам экспериментов. В табл. 3.10 приведено время решения серии из 10 тыс. задач с использованием методов прогонки и редукции на разных размерностях матрицы, а на рис. 3.15 показана зависимость времени решения от размерности матрицы СЛАУ.

Таблица 3.10. Время решения серии задач
с использованием методов прогонки и редукции

N	**Время работы метода прогонки (сек)**	**Время работы метода редукции (сек)**
256	0,202	0,031
512	0,405	0,078
1024	0,795	0,14
2048	0,53	0,265

4096	1,045	0,514
8192	2,074	1,092
16384	4,227	2,169
32768	8,47	4,399
65536	19,11	9,111

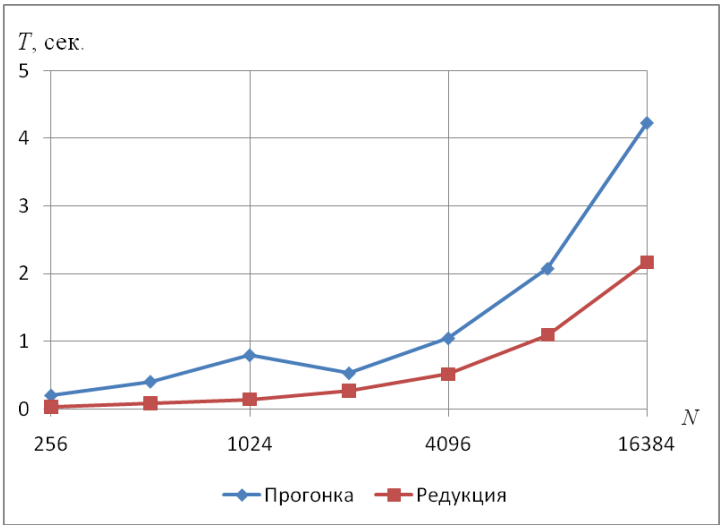


Рис. 3.15. Зависимость времени решения от размера матрицы

Приведенные результаты показывают, что время решения задачи с помощью метода прогонки на всех размерностях матрицы более чем в 2 раза превышает время решения с использованием метода редукции, что противоречит теоретическим оценкам трудоемкости методов.

Полученный эффект можно объяснить существенным влиянием на результат архитектуры компьютера, на котором проводились эксперименты. В самом деле, если воспользоваться инструментами анализа производительности (напр., Intel Parallel Amplifier XE), можно увидеть, что в среднем операция метода прогонки выполняется почти за 2 такта, операция метода редукции – примерно за 1 такт. При этом количество инструкций в редукции приблизительно в 1.3 раза больше. Поскольку СЛАУ решается многократно, то время при использовании редукции меньше, чем при использовании метода прогонки. Подробное описание данного исследования приведено в лабораторной работе «Дифференциальные уравнения в частных производных».

Теперь выполним анализ масштабируемости параллельной реализации метода циклической редукции. Ниже приведены показатели ускорения (отно-

сительной однопоточной реализации), соответствующие работе метода на 2, 4 и 8 потоках.

Таблица 3.11. Результаты решения серии задач параллельным методом редукции

N	1 поток	Параллельный алгоритм					
		2 потока		4 потока		8 потоков	
	T	*T*	*S*	*T*	*S*	*T*	*S*
2048	0,31	0,31	1,00	0,52	0,61	0,66	0,48
4096	0,59	0,51	1,15	0,78	0,76	0,83	0,72
8192	1,14	0,91	1,26	1,01	1,12	1,09	1,04
16384	2,22	1,47	1,51	1,47	1,51	1,55	1,43
32768	4,46	3,12	1,43	2,51	1,78	2,47	1,81
65536	9,19	5,55	1,65	4,38	2,10	3,96	2,32

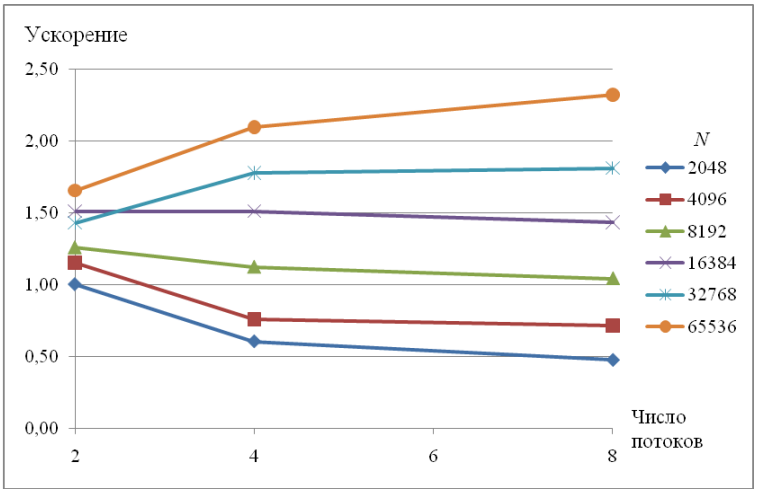


Рис. 3.16. Зависимость ускорения от числа потоков для метода редукции

Представленные результаты экспериментов свидетельствуют о плохой масштабируемости приложения, т.к. только при размерах задачи более 32 тыс. ускорение составляет немногим более двух в лучшем случае.

Данный факт можно объяснить тем, что распараллеливание выполнено на уровне внутреннего цикла прямого и обратного хода редукции, т.е. на каждой итерации редукции порождается или возобновляется несколько потоков, выполняется ожидание их завершения (фактически, точка синхрони-

зации), после чего главный поток продолжает последовательные вычисления, т.е. значительную часть времени программа работает в 1 поток. Таким образом, значительное влияние на время работы программы оказывают накладные расходы, связанные с организацией параллелизма.

Если же посмотреть на параллельный метод редукции с точки зрения работы с данными, то можно сказать, что отсутствие масштабируемости также связано с неэффективной организацией работы с памятью. Пересчет правых частей СЛАУ и вычисление решения в методе осуществляется не последовательно, а с некоторым регулярным шагом на каждой итерации прямого и обратного хода редукции, это приводит к многочисленным кэш-промахам при увеличении числа потоков.

Таким образом, результаты экспериментов показывают, что решать системы с ленточной матрицей с использованием параллельных алгоритмов целесообразно лишь при достаточно большом размере матрицы, при этом в любом случае мы будем получать не очень значительное ускорение.

3.5. Методы решения систем с разреженной матрицей

В данном разделе будут рассмотрены вопросы, касающиеся решения СЛАУ с разреженными матрицами. В предыдущих разделах неявно подразумевалось, что мы работаем с плотными матрицами.

Понятие разреженной матрицы можно определить многими способами, суть которых состоит в том, что в разреженной матрице «много» нулевых элементов. Обычно говорят, что матрица *разрежена*, если она содержит $O(n)$ отличных от нуля элементов. В противном случае матрица считается *плотной*. Типичным случаем разреженности является ограниченность числа ненулевых элементов в одной строке от 1 до k, где $k<<n$. Задачи линейной алгебры с разреженными матрицами возникают во многих областях, например, при решении дифференциальных уравнений в частных производных, при решении многомерных задач локальной оптимизации. В п. 3.3 мы уже познакомились с методами решения СЛАУ с трехдиагональной матрицей, которая, являясь ленточной, относится также и к классу разреженных матриц.

Очевидно, что любую разреженную матрицу можно обрабатывать как плотную, и наоборот. При правильной реализации алгоритмов в обоих случаях будут получены правильные результаты, однако вычислительные затраты будут существенно отличаться. Поэтому приписывание матрице свойства разреженности эквивалентно утверждению о существовании алгоритма, использующего ее разреженность и делающего операции с ней эффективнее по сравнению со стандартными алгоритмами.

Многие алгоритмы, тривиальные для случая плотных матриц, в разреженном случае требуют более тщательного подхода. Во многих алгоритмах обработки разреженных матриц можно выделить два этапа: *символический* и *численный*. На символическом этапе формируется *портрет* результирующей матрицы (т.е. определяются места ненулевых элементов в структуре матрицы); на численном этапе определяются значения ненулевых элементов результирующей матрицы. В качестве примера типовых операций с разреженными матрицами в данной главе будет рассмотрен алгоритм умножения разреженной матрицы на плотный вектор, а также круг вопросов, возникающих при использовании метода Холецкого для решения систем линейных уравнений с разреженной матрицей.

3.5.1. Хранение разреженной матрицы

Существуют различные форматы хранения разреженных матриц. Одни предназначены для хранения матриц специального вида (например, ленточных), другие обеспечивают работу с матрицами общего вида. Ниже рассмотрим некоторые весьма распространенные способы представления разреженных матриц, информацию о других способах можно найти, например, в [10].

По-видимому, наиболее очевидным способом хранения произвольной разреженной матрицы является *координатный формат*: хранятся только ненулевые элементы матрицы, и их координаты (номера строк и столбцов). При данном подходе хранение матрицы *A* можно обеспечить в трех одномерных массивах:

- массив ненулевых элементов матрицы *A* (обозначим его как *values*);
- массив номеров строк матрицы *A*, соответствующих элементам массива *values* (обозначим его как *rows*);
- массив номеров столбцов матрицы *A*, соответствующих элементам массива *values* (обозначим его как *cols*);

Данный способ представления называют *полным*, поскольку представлена вся матрица A, и *неупорядоченным*, поскольку элементы матрицы могут храниться в произвольном порядке.

В качестве примера рассмотрим разреженную матрицу

$$A=\begin{bmatrix} 1 & -1 & 0 & -3 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{bmatrix}, \qquad (3.40)$$

которая может быть представлена в координатном формате как

values=(1, –1, –3, –2, 5, 4, 6, 4, –4, 2, 7, 8, –5);

rows=(1, 1, 1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5);

cols=(1, 2, 4, 1, 2, 3, 4, 5, 1, 3, 4, 2, 5).

Хотя многие математические библиотеки поддерживают матрично-векторные операции в координатном формате, данный формат обеспечивает медленный доступ к элементам матрицы, и является затратным по используемой памяти. В рассмотренном выше примере избыточность по памяти образом проявляется в массиве *rows*, в котором строчные координаты хранятся неоптимальным образом.

Перейдем далее к рассмотрению более экономных форматов хранения. *Разреженный строчный формат* – это одна из наиболее широко используемых схем хранения разреженных матриц. Эта схема предъявляет минимальные требования к памяти и в то же время оказывается очень удобной для нескольких важных операций над разреженными матрицами: сложения, умножения, перестановок строк и столбцов, транспонирования, решения линейных систем с разреженными матрицами коэффициентов как прямыми, так и итерационными методами и т. д.

В соответствии с рассматриваемой схемой для хранения матрицы *A* требуется три одномерных массива:

- массив ненулевых элементов матрицы *A*, в котором они перечислены по строкам от первой до последней (обозначим его опять как *values*);
- массив номеров столбцов для соответствующих элементов массива *values* (обозначим его как *cols*);
- массив указателей позиций, с которых начинается описание очередной строки (обозначим его *pointer*). Описание *k*-й строки хранится в позициях с *pointer*[*k*]-й по (*pointer*[*k*+1]–1)-ю массивов *values* и *cols*. Если *pointer*[*k*]=*pointer*[*k*+1], то *k*-я строка пустая. Если матрица *A* состоит из *n* строк, то длина массива pointer будет *n*+1.

Данный способ представления также является полным, и *упорядоченным*, поскольку элементы каждой строки хранятся в соответствии с возрастанием столбцовых индексов.

Для примера рассмотрим представление матрицы (3.40) в разреженном строчном формате:

$$values=(1, -1, -3, -2, 5, 4, 6, 4, -4, 2, 7, 8, -5);$$

$$cols=(1, 2, 4, 1, 2, 3, 4, 5, 1, 3, 4, 2, 5);$$

$$pointer=(1, 4, 6, 9, 12, 14).$$

Очевидно, что объем памяти, требуемый для хранения вектора *pointer*, значительно меньше, чем для хранения вектора *rows*. Более того, разреженный строчный формат обеспечивает эффективный доступ к строчкам матрицы; доступ к столбцам по прежнему затруднен. Поэтому предпочтительно использовать этот способ хранения в тех алгоритмах, в которых преобладают строчные операции.

Иногда бывает удобно использовано *полный неупорядоченный* способ хранения, при котором внутри каждой строки элементы могут храниться в произвольном порядке. Результаты многих матричных операций получаются неупорядоченными, и упорядочивание может быть весьма затратным. В то же время, многие алгоритмы для разреженных матриц не требуют, чтобы представление было упорядоченным.

После рассмотрения строчного формата хранения очевидным является и *разреженный столбцовый формат*. В этом случае ненулевые элементы матрицы *A* перечисляются в порядке их появления в столбцах матрицы, а не в строках. Все ненулевые элементы хранятся по столбцам в массиве *values*; индексы строк ненулевых элементов – в массиве *rows*; элементы массива *pointer* указывают на позиции, с которых начинается описание очередного столбца.

Столбцовые представления могут рассматриваться как строчные представления транспонированных матриц. Разреженный столбцовый формат обеспечивает эффективный доступ к столбцам матрицы; доступ к строкам затруднен. Поэтому предпочтительно использовать этот способ хранения в тех алгоритмах, в которых преобладают столбцовые операции.

В случае если обрабатываемая матрица симметрична, достаточно хранить лишь ее верхнюю треугольную подматрицу. При этом для хранения можно использовать любой из рассмотренных форматов. Например, симметричная матрица

$$A = \begin{bmatrix} 1 & -1 & 0 & -3 & 0 \\ -1 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -3 & 0 & 6 & 7 & 0 \\ 0 & 0 & 4 & 0 & -5 \end{bmatrix}$$

может быть представлена в разреженном строчном формате как

$$values=(1, -1, -3, 5, 4, 6, 4, 7, -5);$$

$$cols=(1, 2, 4, 2, 3, 4, 5, 4, 5);$$

$$pointer=(1, 4, 5, 8, 9, 10).$$

Если большинство диагональных элементов заданной симметричной матрицы отличны от нуля (например, у симметричной положительно определенной матрицы все диагональные элементы положительны), то они могут храниться в отдельном массиве *BD*, а разреженным форматом представляется только верхний треугольник *B*.

Завершая обзор форматов хранения разреженных матриц, можно еще раз отметить, что выбор способа хранения матрицы полностью определяется алгоритмами, которые планируется в дальнейшем использовать для обработки данной матрицы.

3.5.2. Базовые алгоритмы обработки разреженных матриц

Реализация матричных операций является тривиальной в случае плотной или ленточной матрицы, но не столь очевидна для разреженных матриц, хранящихся в одном из экономичных форматов (здесь и далее мы будем рассматривать разреженный строчный формат хранения матрицы).

3.5.2.1 Умножение матрицы на вектор

Рассмотрим операцию умножения разреженной матрицы на плотный вектор. Результатом этой операции будет заполненный вектор, а не разреженная матрица. Поэтому в алгоритмах умножения вычисления выполняются прямо, без символического этапа.

Среди алгоритмов, в которых встречается данная операция, можно отметить итерационные методы решения систем линейных уравнений. Достоинство данных методов, с вычислительной точки зрения, состоит в том, что единственная требуемая матричная операция, – это повторное умножение матрицы на последовательность заполненных векторов; сама матрица не меняется.

Итак, рассмотрим умножение разреженной матрицы общего вида, хранимой в строчном формате посредством массивов *values, cols, pointer* на заполненный вектор-столбец *b*, хранимый в одномерном массиве. Результатом будет новый заполненный вектор

$$c=Ab,$$

также размещаемый в одномерном массиве.

Пусть n – число строк матрицы. Порядок, в котором накапливаются скалярные произведения, определяется порядком хранения элементов матрицы. Для каждой ее строки i мы находим с помощью массива индексов *pointer* значения первой *pointer*[i] и последней *pointer*[i+1]–1 позиций, занимаемых элементами строки i в массивах *values* и *cols*. Затем, чтобы вычислить скалярное произведение строки i и вектора b, мы просто просматриваем *values* и *cols* на отрезке от *pointer*[i] до *pointer*[i+1]–1; каждое значение, хранимое в *cols*[*pointer*[j]], есть столбцовый индекс и используется для извлечения из массива b элемента, который должен быть умножен на соответствующее число из массива *values*. Результат каждого умножения прибавляется к c[i].

Рассмотрим теперь параллельный алгоритм умножения разреженной матрицы на плотный вектор, матрица представлена в строчном формате. При таком способе представления данных в качестве базовой подзадачи может быть выбрана операция скалярного умножения одной строки матрицы на вектор. После завершения вычислений каждая базовая подзадача определяет один из элементов вектора результата c.

Как правило, количество элементов в одной строке разреженной матрицы размера $n \times n$ ограничено некоторой константой k, $k << n$. Значит, в процессе умножения такой матрицы на вектор количество вычислительных операций для получения скалярного произведения примерно одинаково для всех базовых подзадач. При использовании систем с общей памятью число потоков p будет меньше числа базовых подзадач n, и мы можем объединить базовые подзадачи таким образом, чтобы каждый поток выполнял несколько таких задач, соответствующих непрерывной последовательности строк матрицы A. В этом случае по окончании вычислений каждая базовая подзадача определяет набор элементов результирующего вектора c. Распределение подзадач между потоками может быть выполнено произвольным образом.

3.5.2.2 Транспонирование матрицы

Рассмотрим задачу транспонирования разреженной матрицы A. Формально матрица A^T может быть определена как

$$A^T[i,j]= A^T[j,i],$$

и простейшая реализация данной операции в «плотном» случае оказывается эффективной. Однако в случае разреженной матрицы ситуация не столь очевидна, и простейшая реализация может существенно ухудшить показатели эффективности.

Будем формировать результирующую матрицу построчно. Для этого можно брать столбцы исходной матрицы и создавать из них строки результирующей матрицы. Но операция выделения из CRS-матрицы столбца №*i* является трудоемкой, т.к. данные в векторе *values* хранятся по строкам и для выборки данных по столбцу нужно просмотреть всю матрицу, что приводит к квадратичной (от числа ненулевых элементов) трудоемкости алгоритма. Необходимо другое решение. Подробно проблема транспонирования разреженной матрицы обсуждается в книге [10], здесь же рассмотрим основные идеи описанного в [10] алгоритма.

1. Сформируем N одномерных векторов для хранения целых чисел (*IntVectors*), а также N векторов для хранения вещественных чисел (*RealVectors*). N в данном случае соответствует числу столбцов исходной матрицы.
2. В цикле просмотрим все строки исходной матрицы, для каждой строки – все ее элементы. Пусть текущий элемент находится в строке i, столбце j, его значение равно v. Тогда добавим числа i и v в j-ые вектора для хранения целых и вещественных чисел (соответственно). Тем самым в векторах мы сформируем строки транспонированной матрицы.
3. Последовательно скопируем данные из векторов в CRS-структуру транспонированной матрицы (*cols* и *values*), попутно формируя массив *pointer*.

Рассмотрим пример (3.17).

При обходе исходной матрицы A формируются вектора *IntVectors* и *RealVectors* (число 3 расположено в строке 0 и столбце 1, следовательно, в *IntVectors[1]* добавляется 0, в *RealVectors[1]* добавляется 3 и т.д.). Далее вектора последовательно формируют структуру A^T: обратите внимание на порядок следования элементов в массиве *values* матрицы A^T и его соответствие порядку элементов в массиве *RealVectors* (аналогично *cols* и *IntVectors*). Для формирования *pointer* достаточно подсчитать количество элементов в каждом из N векторов. *pointer[0]* всегда равно нулю, *pointer[i]=pointer[i–1]* + «Количество элементов в векторе *i–1*».

Таким образом, алгоритм транспонирует матрицу за линейное время, что значительно лучше исходного тривиального алгоритма. Недостатком же алгоритма в том виде, как он изложен, является использование дополнительной памяти. В книге [10] приводится описание алгоритма, лишенного

этого недостатка. Основная идея состоит в использовании структур данных матрицы A^T для промежуточных результатов вычислений.

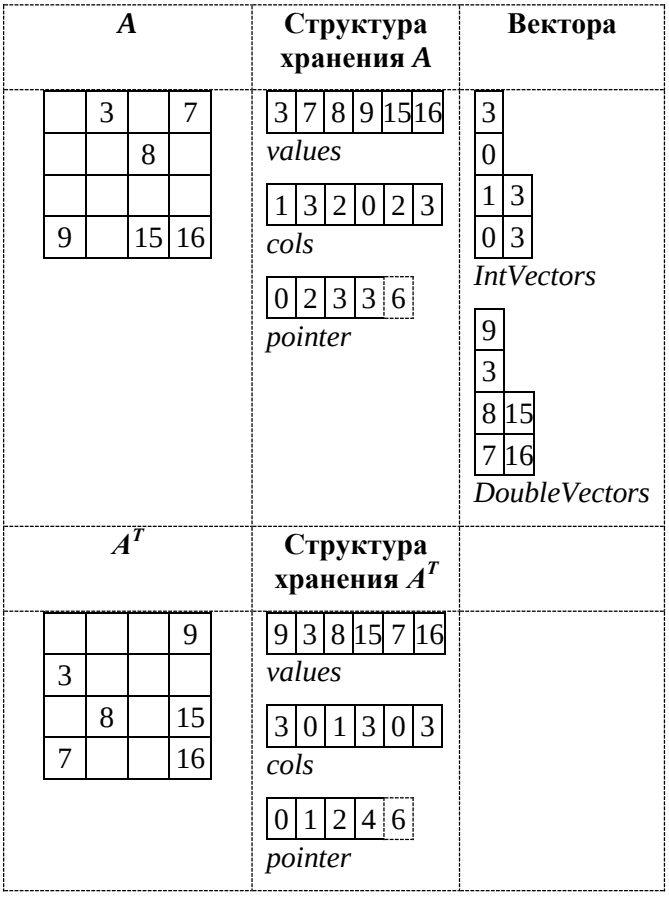
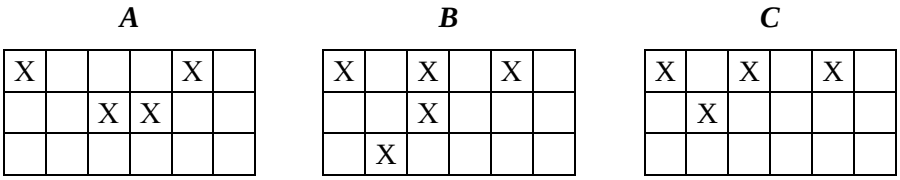


Рис. 3.17. Транспонирование разреженной матрицы

3.5.2.3 Матричное умножение

Приступим к обсуждению алгоритмов умножения разреженных матриц. Прежде всего рассмотрим алгоритм умножения непосредственно по определению.



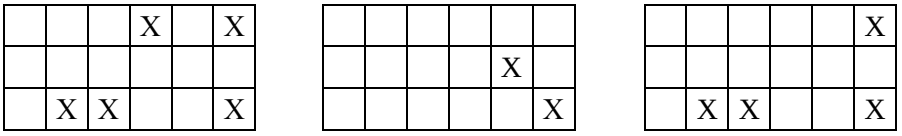


Рис. 3.18. Умножение матриц

Определение предполагает, что элемент в строке *i* и столбце *j* матрицы *C* вычисляется как скалярное произведение *i*-ой строки матрицы *A* и *j*-го столбца матрицы *B*. Какие особенности вносит разреженное представление?

Во-первых, используемая структура данных, построенная на основе формата CRS, предполагает хранение только ненулевых элементов, что усложняет программирование вычисления скалярного произведения, но одновременно уменьшает количество арифметических операций. При вычислении скалярных произведений нет необходимости умножать нули и накапливать полученный нуль в частичную сумму, что положительно влияет на сокращение времени счета. Пусть, например, в первом и втором векторах находится по 1% ненулевых элементов, при этом только десятая часть этих элементов расположена на соответствующих друг другу позициях. В этом случае расчет с использованием информации о структуре векторов может использовать в 1000 раз меньшее число умножений и сложений. Учитывая, что таких пар векторов N^2, получается существенное сокращение объема вычислений. К сожалению, не все так просто. Учет структуры векторов тоже требует машинного времени. Необходимо выполнить сопоставление номеров ненулевых элементов с целью обнаружения пар значений, которые необходимо перемножить и накопить в частичную сумму.

Во-вторых, необходимо научиться выделять вектора в матрицах *A* и *B*. В соответствии с определением, речь идет о строках матрицы *A* и столбцах матрицы *B*. Выделить строку матрицы в формате CRS не представляет труда: *i*-я строка может быть легко найдена, так как ссылки на первый элемент *poiner[i]* и последний элемент (*pointer[i+1]–1*) известны, что позволяет получить доступ к значениям элементов и номерам столбцов, хранящихся в массивах *values* и *cols* соответственно. Таким образом, проход по строке выполняется за время, пропорциональное числу ненулевых элементов в указанной строке, а проход по всем строкам – за время, пропорциональное *NZ*, где *NZ* – количество ненулевых элементов в матрице.

Проблема возникает с выделением столбца. Чтобы найти элементы столбца *j* необходимо просмотреть массив *cols* (~*NZ* операций) и выделить все элементы, у которых в соответствующей ячейке массива *cols* записано число *j*. Если это нужно проделать для каждого столбца, необходимо ~*NZ*·*N* операций, что является неэффективным.

Возможное, но не единственное решение проблемы состоит *в транспонировании матрицы B*: после вычисления B^T появится возможность эффективно работать со столбцами исходной матрицы *B*, а сам алгоритм транспонирования работает достаточно быстро. Другой вариант – для каждой CRS-матрицы, которая может понадобиться в столбцовом представлении, дополнительно хранить транспонированный портрет. Сэкономив время на транспонировании, придется смириться с расходами времени на поддержание дополнительного портрета в актуальном состоянии. Есть и другие варианты решения проблемы, никак не связанные с транспонированием матрицы *B* (см., например, эффективный алгоритм Густавсона [11]). В дальнейшем мы будем использовать подход, основанный на транспонировании матрицы *B*.

B-третьих, необходимо научиться записывать посчитанные элементы в матрицу *C*. Учитывая, что *C* хранится в формате CRS, важно избежать перепаковок. Для этого нужно обеспечить пополнение матрицы *C* ненулевыми элементами последовательно, по строкам – слева направо, сверху вниз. Это означает, что часто используемый в вычислениях на бумаге метод «возьмем первый столбец матрицы *B*, запишем его над матрицей *A* и перемножим на все строки…» не подходит. Нужно делать наоборот – брать первую строку матрицы *A* и умножать ее по очереди на все столбцы матрицы *B* (строки матрицы B^T). В этом случае обеспечивается последовательное пополнение матрицы *C*, позволяющее дописывать элементы в массивы *values* и *cols*, а также формировать массив *pointer*.

Таким образом, для выполнения матричного умножения разреженных матриц необходимо сделать следующее:

1. Реализовать транспонирование разреженной матрицы и применить его к матрице *B*.
2. Инициализировать структуру данных для матрицы *C*, обеспечить возможность ее пополнения элементами.
3. Последовательно перемножить каждую строку матрицы *A* на каждую из строк матрицы B^T, записывая в *C* полученные результаты и формируя ее структуру.

Еще один непростой момент: в процессе вычисления скалярного произведения на бумаге (в точной арифметике) может получиться нуль, причем не только в том случае, когда при сопоставлении векторов соответствующих друг другу пар не обнаружилось, но и просто как естественный результат. В арифметике с плавающей точкой нуль может получиться еще и в связи с ограничениями на представление чисел и погрешностью вычислений (см. стандарт IEEE-754). Нули, получившиеся в процессе вычислений, можно как хранить в матрице *C* (в векторе *values* в соответствующей позиции), так и не хранить; оба подхода имеют право на существование.

Как следует из приведенного описания, основной операцией является скалярное произведение двух разреженных векторов (обозначим их V_1 и V_2), и к ее реализации нужно подойти с особой тщательностью.

Алгоритм подразумевает для каждой отличной от нуля компоненты вектора V_1 проверку, есть ли отличная от нуля компонента с таким же номером в векторе V_2. Простейший вариант такой проверки имеет квадратичную трудоемкость, что существенно сказывается на эффективности.

Заметим, что каждый умножаемый вектор является строкой матрицы и упорядочен в соответствии с выбранной структурой хранения матрицы. Это означает, что для сопоставления компонент может быть применен алгоритм, весьма похожий на слияние двух отсортированных массивов в один с сохранением упорядочивания. Алгоритм выглядит следующим образом:

1. Встать на начало обоих векторов (**`ks =` …**, **`ls =` …**).
2. Сравнить текущие элементы **`A.Col[ks]`** и **`B.Col[ls]`**. Если значения совпадают, накопить в сумму произведение **`A.Value[ks] * B.Value[ls]`** и увеличить оба индекса, в противном случае – увеличить один из индексов, в зависимости от того, какое значение больше (например, **`A.Col[ks] > B.Col[ls]`** → **`ls++`**).

Шаг 2 выполняется до тех пор, пока не кончатся элементы хотя бы в одном из векторов.

Данный алгоритм имеет линейную трудоемкость, но он не очень хорошо соответствует архитектуре компьютера, т.к. содержит очень много ветвлений. Существуют более эффективные схемы: например, в книге [10] предлагается следующий алгоритм вычисления скалярного произведения разреженных векторов.

1. Создадим дополнительный целочисленный массив X длины N. Инициализируем его числом –1. Обнулим вещественную переменную S.
2. Просмотрим все ненулевые элементы первого вектора V_1. Пусть такой элемент с порядковым номером i расположен в столбце с номером $j=V_1.cols[i]$. В этом случае запишем i в j-ю ячейку массива X.
3. Просмотрим в цикле все ненулевые элементы второго вектора V_2. Пусть элемент с порядковым номером k расположен в столбце с номером $z=V_2.cols[k]$. Проверим значение $X[z]$. Если оно равно –1, в первом векторе нет соответствующего элемента, т.е. умножение выполнять не нужно. Иначе умножаем $V_2.values[k]$ и $V_2.values[X[z]]$ и накапливаем сумму в S.

Данный алгоритм также имеет линейную (относительно числа ненулевых элементов в векторе) трудоемкость, и не содержит избыточных ветвлений.

В заключение отметим, что рассмотренные нами базовые алгоритмы демонстрируют типичные трудности, которые возникают при реализации очевидных «плотных» алгоритмов в «разреженном» случае.

3.5.3. Метод Холецкого для разреженных матриц

В п. 3.2 был рассмотрен метод Холецкого для решения системы линейных уравнений с симметричной положительно определенной матрицей

$$Ax=b, \tag{3.41}$$

при этом подразумевалось, что матрица A – плотная. Данный алгоритм, естественно, будет применим и для разреженной матрицы A, однако в этом случае возникает ряд важных моментов, к обсуждению которых мы сейчас и приступаем.

В качестве примера рассмотрим систему уравнений

$$\begin{bmatrix} 4 & 1 & 2 & 0.5 & 2 \\ 1 & 0.5 & 0 & 0 & 0 \\ 2 & 0 & 3 & 0 & 0 \\ 0.5 & 0 & 0 & 0.625 & 0 \\ 2 & 0 & 0 & 0 & 16 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 17 \\ 3 \\ 7 \\ 6 \\ 12 \end{bmatrix} \tag{3.42}$$

Используя расчетные формулы метода (3.13) и (3.14), можно вычислить фактор Холецкого для данной задачи

$$L = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 & 0 \\ 0.25 & -0.25 & -0.5 & 0.5 & 0 \\ 1 & -1 & -2 & -3 & 1 \end{bmatrix},$$

а затем, выполнив обратный ход алгоритма по формулам (3.15), (3.16), найти искомое решение

$$x^T=[2, 2, 1, 8, 0.5].$$

Этот пример иллюстрирует наиболее важный факт, относящийся к применению метода Холецкого для разреженных матриц: матрица обычно пре-

терпевает *заполнение*. Это значит, что в матрице L появляются ненулевые элементы в позициях, где в нижней треугольной части A стояли нули.

Предположим теперь, что мы перенумеровали переменные в соответствии с правилом $y_i=x_{i+1}$, $i=1, 2,..., 4$, $y_5=x_1$ и переупорядочили уравнения так, чтобы первое стало последним, а остальные сдвинулись бы на единицу вверх. Тогда мы получим эквивалентную систему уравнений

$$\begin{bmatrix} 0.5 & 0 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 & 2 \\ 0 & 0 & 0.625 & 0 & 0.5 \\ 0 & 0 & 0 & 16 & 2 \\ 1 & 2 & 0.5 & 2 & 4 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 3 \\ 7 \\ 6 \\ 12 \\ 17 \end{bmatrix} \tag{3.43}$$

Очевидно, что эта перенумерация переменных и переупорядочение уравнений равносильны симметричной перестановке строк и столбцов матрицы A, причем та же перестановка применяется и к вектору b. Решение новой системы есть переупорядоченный вектор x. Указанную перенумерацию легко записать в виде произведения матицы A на матрицу перестановки P. Напомним, что матрица P называется матрицей перестановки, если в каждой строке и столбце матрицы находится лишь один единичный элемент. В нашем примере

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}, \tag{3.44}$$

а матрицу системы (3.43) можно получить как

$$\overline{A} = PAP^T.$$

При этом связь с исходной постановкой задачи выражается соотношением

$$(PAP^T)(Px) = Pb.$$

Ниже приведен фактор Холецкого для новой системы уравнений (с точностью до 10^{-3})

$$L = \begin{bmatrix} 0.707 & 0 & 0 & 0 & 0 \\ 0 & 1.732 & 0 & 0 & 0 \\ 0 & 0 & 0.79 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 1.414 & 1.154 & 0.632 & 0.5 & 0.129 \end{bmatrix}.$$

Выполнив обратный ход, можно получить переупорядоченный вектор решения

$$y^T=[2, 1, 8, 0.5, 2].$$

Важный момент состоит в том, что переупорядочение уравнений и переменных привело к треугольному множителю L, который разрежен в точности в той мере, что и нижний треугольник A.

Задача нахождения перестановки, минимизирующей заполнение при вычислении фактора Холецкого, является *NP*-трудной (т.к. вообще говоря, минимум должен вычисляться по всем $n!$ перестановкам). Поэтому на практике используют эвристические алгоритмы, которые хотя и не гарантируют, что находят оптимальную перестановку, но, как правило, дают приемлемый результат.

Таким образом, при решении разреженной системы методом Холецкого можно выделить следующие этапы:

1. *Переупорядочивание* – вычисление матрицы перестановки P;
2. *Символическое разложение* – построение портрета матрицы L и выделение памяти под хранение ненулевых элементов;
3. *Численное разложение* – вычисление значений матрицы L и размещение их в выделенной памяти.
4. *Обратный ход* – решение двух треугольных систем уравнений.

Этапы 1 и 2 специфичны именно для разреженных матриц, тогда как этапы 3 и 4 выполняются для любых задач.

Ниже мы рассмотрим два известных алгоритма определения матрицы перестановки – методы *минимальной степени* и *вложенных сечений*.

3.5.3.1 Метод минимальной степени

Для формулирования метода минимальной степени нам потребуются некоторые сведения из теории графов.

Рассмотрим неориентированный граф G. Формально граф G можно рассматривать как упорядоченную пару двух множеств

$$G=(V, E),$$

где $V=\{1,...,n\}$ – множество вершин, $E=\{(e_1,e_2)\subseteq V\times V\}$ – множество связей между вершинами (ребер). Так как рассматриваемый граф – *неориентированный*, то множество E удовлетворяет свойству

$$(e_1,e_2)\in E\leftrightarrow(e_2,e_1)\in E.$$

Пара вершин $x\in V$, $y\in V$ называется *смежной*, если существует ребро, их соединяющее, т.е.

$$(x,y)\in E.$$

Для любой вершины $x\in V$ можно определить множество $Adj(x)$ смежных ей вершин как

$$Adj(x)=\{y\in V:(x,y)\in E\}.$$

Мощность множества $Adj(x)$ называется *степенью* вершины

$$deg(x)=|Adj(x)|.$$

Граф, каждые две вершины которого соединены ребром, называется *полным*.

Подграф исходного графа – граф, содержащий некое подмножество вершин данного графа и некое подмножество инцидентных им рёбер.

Кликой в неориентированном графе называется подмножество вершин, каждые две из которых соединены ребром графа. Иными словами, это полный подграф первоначального графа.

Одной из форм задания графа $G=(V, E)$ является *матрица смежности* $A=(a_{ij})$ размера $n\times n$ такая, что

$$a_{ij}=\begin{cases}\neq 0, (i, j)\in E,\\ 0, \quad (i, j)\notin E.\end{cases}$$

В нашем случае матрицу A системы линейных уравнений (3.41) можно рассмотреть как матрицу смежности соответствующего графа $G(A)$, за исключением диагональных элементов, которым будут соответствовать петли. На рис. 3.19 приведен граф, соответствующий матрице из примера (3.42).

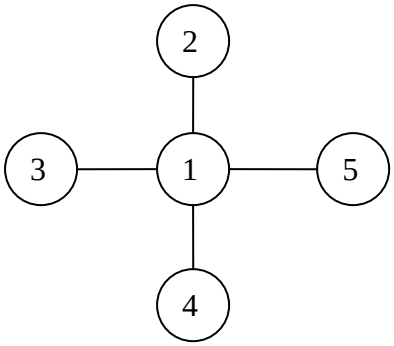
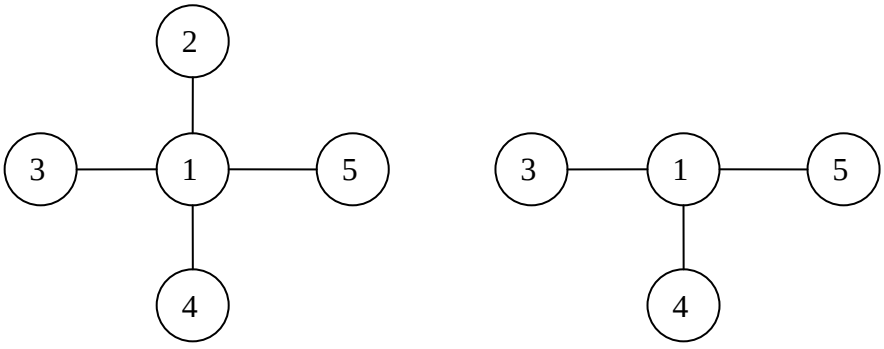


Рис. 3.19. Пример графа матрицы

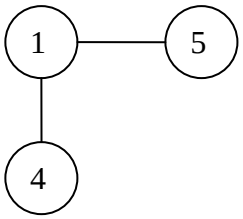
После того, как введены все необходимые определения, можно перейти к описанию алгоритма минимальной степени.

Пусть исходной матрице A из (3.41) соответствует граф $G(A)$. Алгоритм минимальной степени строит последовательность графов исключения G_i, каждый их которых получен из предыдущего удалением вершины с минимальной степенью и созданием клики между всеми вершинами, которые были смежными с удаленной. В случае, когда вершин с минимальной степенью несколько, выбирается любая. Алгоритм продолжается до тех пор, пока в очередном графе есть вершины. По мере удаления вершин, их номер записывается в перестановку π, по которой впоследствии строится матрица перестановки P.

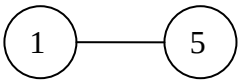
В качестве примера работы алгоритма на рис. 3.20 приведена последовательность графов исключения и соответствующая перестановка π, порождаемая алгоритмом минимальной степени для задачи (3.42).



$G_0,\pi=\{\varnothing\}$ $G_1,\pi=\{2\}$



G_2, π={2,3}



G_3, π={2,3,4}

G_4, π={2,3,4,1}

G_5=∅, π={2,3,4,1,5}

Рис. 3.20. Последовательность графов исключения

Полученная перестановка π={2,3,4,1,5} соответствует матрице

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

которая отличается от уже рассмотренной нами перестановки (3.44). Проверим ожидаемые свойства переупорядоченной матрицы: новая матрица системы уравнений, получаемая по формуле $\overline{A} = PAP^T$, будет

$$\overline{A} = PAP^T = \begin{bmatrix} 0.5 & 0 & 0 & 1 & 0 \\ 0 & 3 & 0 & 2 & 0 \\ 0 & 0 & 0.625 & 0.5 & 0 \\ 1 & 2 & 0.5 & 4 & 2 \\ 0 & 0 & 0 & 2 & 16 \end{bmatrix} \qquad (3.45)$$

Фактор Холецкого, соответствующий данной матрице, будет (с точностью до 10^{-3})

$$L = \begin{bmatrix} 0.707 & 0 & 0 & 0 & 0 \\ 0 & 1.732 & 0 & 0 & 0 \\ 0 & 0 & 0.79 & 0 & 0 \\ 1.414 & 1.154 & 0.632 & 0.516 & 0 \\ 0 & 0 & 0 & 3.873 & 1 \end{bmatrix}.$$

Видно, что при факторизации матрицы $\overline{A}$ не образуется порожденных элементов. Однако это не всегда выполняется, алгоритм лишь необходимым образом уменьшает количество порожденных элементов.

3.5.3.2 Результаты экспериментов

В качестве вычислительного примера рассмотрим разложение некоторых матриц, доступных в коллекции университета Флориды (www.cise.ufl.edu/research/sparse/matrices/). Ниже приведено описание выбранных матриц: указан размер *n*, число ненулевых элементов нижнего треугольника *nz*, портрет матрицы. Все выбранные матрицы являются симметричными положительно определенными.

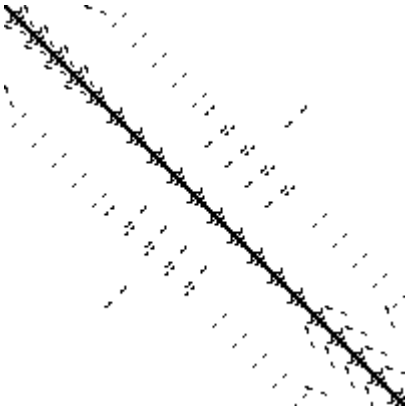


Рис. 3.21. Матрица shallow_water2, *n*=81 920, *nz*=204 800

Рис. 3.22. Матрица parabolic_fem, *n*=525 825, *nz* =2 100 225

Рис. 3.23. Матрица pwtk, *n*=217 918, *nz*=5 871 175

Рис. 3.24. Матрица cfd2, *n*=123 440, *nz*=1 604 423

Следует отметить, что при практической реализации метод минимальной степени в его исходном виде не используется в силу его значительной трудоемкости. Широко известны две модификации метода – *приближенный метод минимальной степени* (*Approximate Minimum Degree, AMD*), и *множественный метод минимальной степени* (*Multiple Minimum Degree, MMD*). Идея метода AMD состоит в вычислении приближенной степени вершины с помощью следующего эвристического правила: степень вершины не превосходит сумму степеней ее соседей. Метод MMD использует следующую модификацию: если на некотором шаге алгоритма нашлось несколько вершин с минимальной степенью, то можно одновременно удалить все из них, не являющиеся соседями.

Метод MMD реализован в библиотеке Intel MKL, и приведенные ниже результаты экспериментов получены с помощью данной библиотеки. Для каждой из наших тестовых матриц была получена перестановка методом MMD, портреты матриц после применения перестановки приведены ниже.

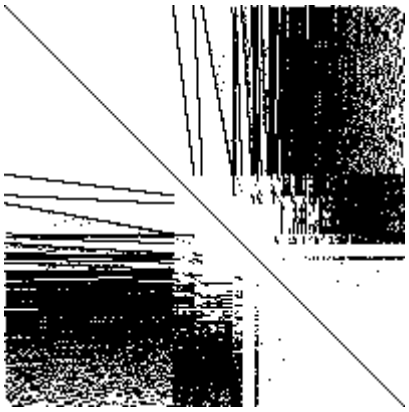


Рис. 3.25. Матрица shallow_water2, *n*=81 920, *nz*=204 800

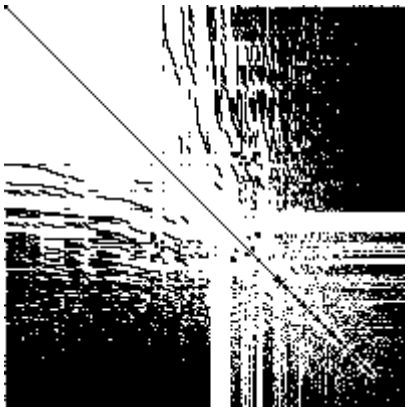


Рис. 3.26. Матрица parabolic_fem, *n*=525 825, *nz* =2 100 225

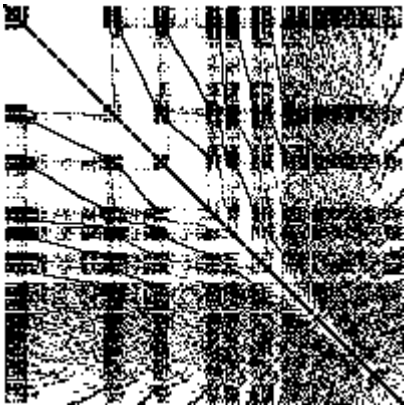


Рис. 3.27. Матрица pwtk, *n*=217 918, *nz*=5 871 175

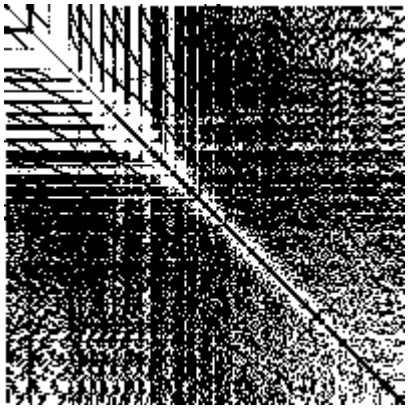


Рис. 3.28. Матрица cfd2, *n*=123 440, *nz*=1 604 423

Кажущаяся «плотность» полученных портретов объясняется масштабированием, ниже приведены портрет правого нижнего угла одной из матриц в высоком разрешении.

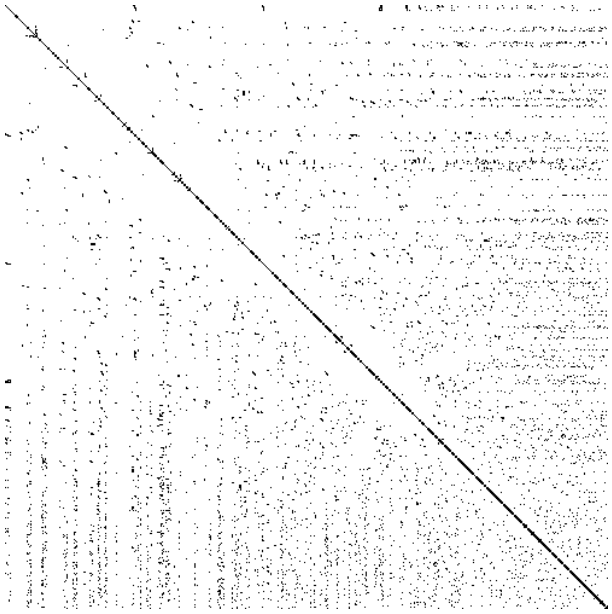


Рис. 3.29. Матрица parabolic_fem,
правый нижний угол.

Для каждой из матриц был вычислен коэффициент заполнения при факторизации в исходном виде (обозначим такой фактор L), и при факторизации после применения перестановки, найденной методом минимальной степени (обозначим полученный фактор L').

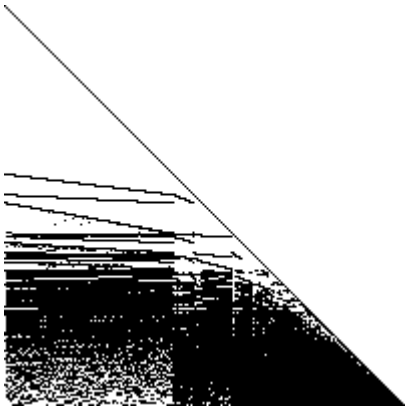


Рис. 3.30. Фактор для shallow_water2,
n=81 920, nz= 2 525 184

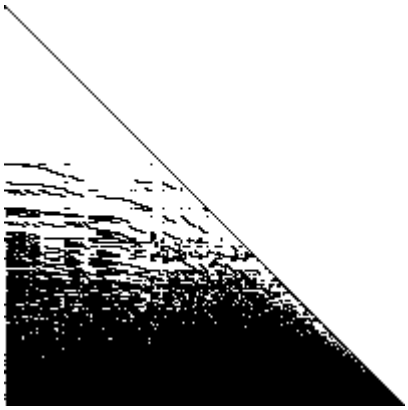


Рис. 3.31. Фактор для parabolic_fem,
n=525 825, nz = 23 582 508

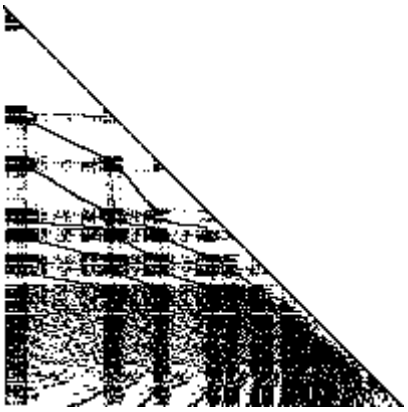


Рис. 3.32. Фактор для pwtk,
n=217 918, *nz*= 60 711 075

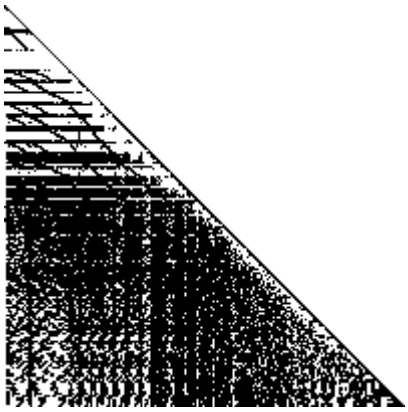


Рис. 3.33. Фактор для cfd2,
n=123 440, *nz*= 66 828 397

Здесь также присутствует эффект масштабирования, ниже приведен портрет правого нижнего угла одного из факторов в высоком разрешении

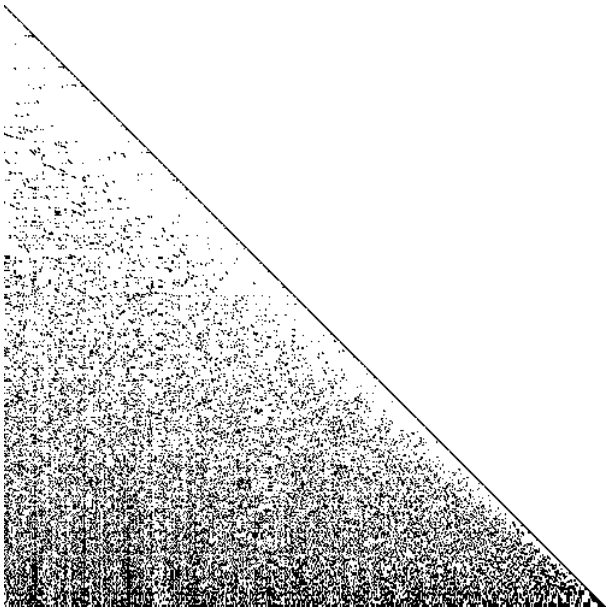


Рис. 3.34. Фактор для parabolic_fem,
правый нижний угол.

Портреты матриц после разложения в целом имеют ту же структуру, что и портреты переупорядоченных матриц – существенного заполнения не произошло. Числовые значения коэффициентов заполнения исходной матрицы A, и ее факторов L и L' приведены в табл. 3.12.

Таблица 3.12. Коэффициенты заполнения матрицы
до и после разложения

Матрица	*A*	*L*	*L'*
shallow_water2	4,88281E-05	0,00687	0,00075
pwtk	0,00024268	0,00803	0,00256
parabolic_fem	1,32902E-05	0,16134	0,00017
cfd2	0,000202489	0,02049	0,00877

Сравнение коэффициентов заполнения факторов L и L' очевидным образом демонстрирует преимущества использования метода минимальной степени.

3.5.3.3 Метод вложенных сечений

Рассмотрим теперь еще один подход к получению переупорядочивания матрицы – *метод вложенных сечений*. Как и в предыдущем пункте, для описания метода нам потребуются некоторые дополнительные сведения из теории графов.

Расстоянием $d(u,v)$ между вершинами графа u, v называется длина кратчайшего пути, соединяющего эти вершины.

Наибольшее расстояние между любыми двумя вершинами графа называется *диаметром* графа.

Наибольшее расстояние от вершины u до любой другой вершины называется *эксцентриситетом* вершины и обозначается $e(u)$. Таким образом,

$$Diam(G)=\max\{e(u_i),\ u_i \in V\}.$$

Если эксцентриситет вершины совпадает с диаметром графа, то такая вершина называется *периферийной*.

Псевдопериферийная вершина u определяется следующим условием: если v – любая вершина, для которой выполнено условие $d(u,v)=e(u)$, то будет выполнено и условие $e(v)=e(u)$. Указанное определение гарантирует, что эксцентриситет псевдопериферийной вершины будет «близким» к диаметру графа.

Граф называется *связным*, если для каждой пары его вершин найдется путь, который их связывает. В противном случае граф называют *несвязным*. Всякий несвязный граф состоит из двух или более *компонент связности*.

Разделителем называется множество вершин, удаление которых вместе с инцидентными им ребрами приводит к появлению несвязного графа (или к увеличению числа связных компонент, если граф уже был несвязным). Разделитель называют *минимальным*, если никакое его собственное

подмножество не является разделителем. Разделитель, состоящий из одной вершины, называется *разрезающей вершиной*.

Разбиением графа называется группировка вершин графа в попарно непересекающиеся подмножества S_0, S_1, … S_m. Если граф несвязный, и в качестве подмножеств выбраны его компоненты связности, то получаем *разбиение на компоненты*.

Одним из важных классов разбиений, использующихся при обработке разреженных матриц, является разбиение на *уровни смежности*, т.е. выявление *структуры уровней смежности* графа.

Структура уровней смежности L_0, L_1, … L_m, состоящая из m+1 уровня, получается, если указанные подмножества определены следующим образом:

$$\begin{gathered} Adj(L_0)\subseteq L_1, \\ Adj(L_m)\subseteq L_{m-1}, \\ Adj(L_i)\subseteq L_{i-1}\cup L_{i+1},\ 0<i<m. \end{gathered} \tag{3.46}$$

Напомним, что множество $Adj(x)$ для вершины $x\in V$ является множеством смежных ей вершин. Число m называют длиной структуры уровней смежности, а *ширина* структуры определяется как максимальное количество вершин, составляющих каждый уровень.

Важным следствием из определения является тот факт, что в структуре уровней смежности каждый уровень L_i, $0<i<m$, является разделителем графа.

Разбиение на уровни смежности называют *корневым* с корнем L_0, если $L_0\subset V$, а каждое следующее множество смежно с объединением предыдущих, т.е.

$$L_i = Adj\left(\bigcup_{j=0}^{i-1} L_j\right),\ i>0. \tag{3.47}$$

Если L_0 состоит из единственной вершины u, то говорят, что структура уровней имеет корень в вершине u.

Рассмотрим теперь алгоритм построения структуры уровней смежности.

1. В графе V выбирается некоторая вершина $u\subset V$ в качестве корневой. Полагаем i=0, L_0={u}.
2. Помечаем данную вершину (присваиваем ей первый номер)
3. Перебираем все вершины из множества L_i, и для каждой его вершины определяем непомеченную смежную ей вершину. Эти смежные

вершины образуют уровень L_{i+1}. Они помечаются (нумеруются по порядку) и помещаются в L_{i+1}.

4. Если множество непомеченных вершин пусто, то алгоритм завершен, иначе полагаем $i=i+1$, и переходим на шаг 3.

На рис. 3.35 приведена матрица разреженной системы линейных уравнений (ненулевые элементы отмечены символом *) и соответствующий ей граф, а на рис. 3.36 изображена структура уровней смежности для данного графа с корнем в вершине 10. Длина структуры равна 3, а ширина – 5. Отметим, что множество вершин {1, 11} первого уровня и множество вершин {6, 9, 7, 3, 8} второго уровня являются разделителями графа.

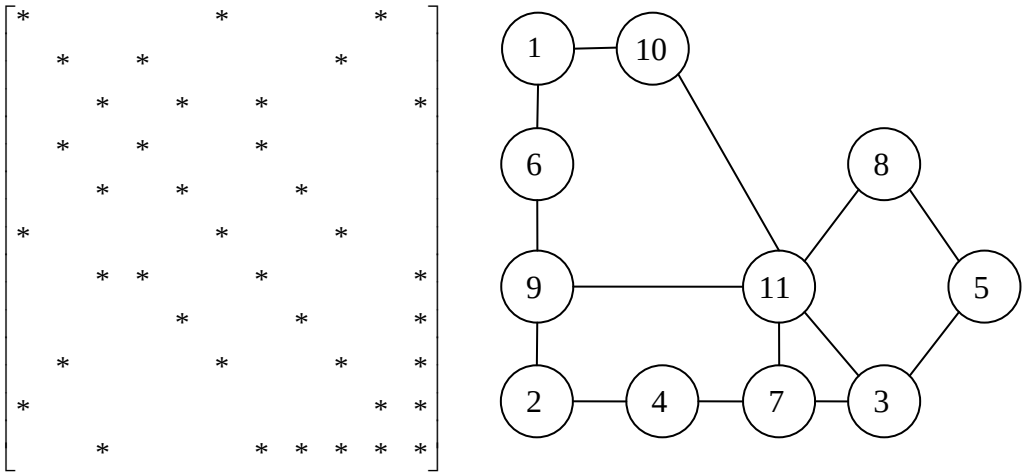


Рис. 3.35. Матрица и ее графовое представление

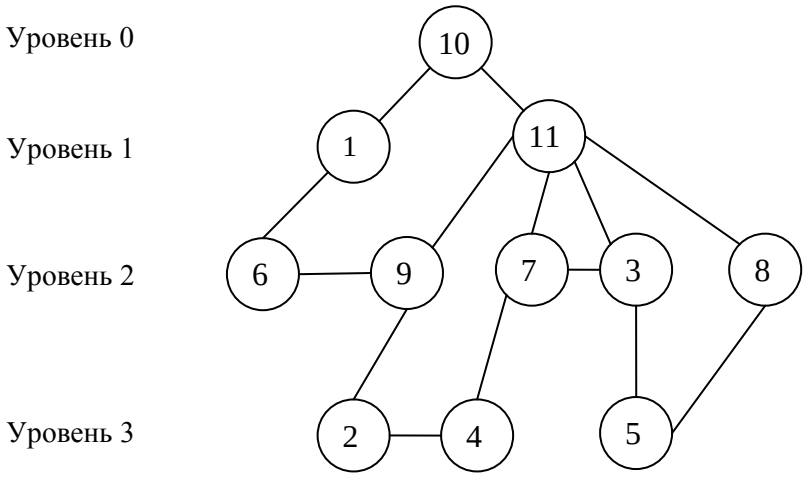


Рис. 3.36. Структура уровней смежности

В соответствии с книгами [9, 10] сформулируем алгоритм отыскания псевдопериферийной вершины r.

1. В графе V выбрать некоторую вершину $r \subset V$ в качестве корневой.
2. Определить структуру уровней смежности $\Lambda(r)=\{L_0, L_1, \ldots L_{m(r)}\}$ с корнем в r.
3. Выбрать в последнем уровне $L_{m(r)}$ вершину x с минимальной степенью.
4. Определить структуру уровней смежности $\Lambda(x)=\{L_0, L_1, \ldots L_{m(x)}\}$ с корнем в x.
5. Если $m(x)>m(r)$, то положить $r \leftarrow x$, и перейти на шаг 3.
6. Узел x является псевдопериферийным.

После того, как введены все требуемые понятия и рассмотрены вспомогательные алгоритмы, сформулируем идею метода вложенных сечений.

Пусть исходной матрице A из (3.41) соответствует граф $G(V,E)$. В графе G находим разделитель $S=\{v_k\}$, где $v_k \in V$. Будем считать, что вершины в графе перенумерованы так, что удаление вершины v_k (и инцидентных с ним ребер) приводит к появлению двух связных компонент G_1 и G_2 с вершинами $\{v_i\}$, $1 \le i \le k-1$, и $\{v_j\}$, $k+1 \le j \le n$, соответственно. Так как $S=\{v_k\}$ является разделителем, то любая i-я вершина G_1 не будет достижима ни для какой j-й вершины из G_2. В этом случае все элементы фактора Холецкого L_{ij} с соответствующими номерами будут равны 0, и заполнения не произойдет.

Далее введем новую нумерацию вершин (которая соответствует перестановке переменных в системе уравнений): перенумеруем вершины в компонентах последовательно, от 1 до $n-1$ (для этого достаточно уменьшить номера вершин в компоненте G_2, нумерация в G_1 останется прежней), а k-й вершине присвоим последний номер n, т.е. v_k перенумеруем как v_n.

Затем в каждой выделенной компоненте G_1 и G_2 проводим аналогичную процедуру: находим разделители $S_1=\{v_p\}$ в G_1 и $S_2=\{v_q\}$ в G_2, выделяем четыре новые компоненты связности: $G_{1,1}$ и $G_{1,2}$ с вершинами $\{v_i\}$, $1 \le i \le p-1$, и $\{v_j\}$, $p+1 \le j \le k-1$; $G_{2,1}$ и $G_{2,2}$ с вершинами $\{v_i\}$, $k+1 \le i \le q-1$, и $\{v_j\}$, $q+1 \le j \le n-1$. Следующий шаг алгоритма заключается в перенумерации вершин в пределах компонент указанным выше способом: вершины компонент связности нумеруются последовательно, разделителю присваивается последний номер. Если в какой-либо компоненте связности G^* нельзя найти разделитель,

то в качестве соответствующего множества S^* выбирается само множество G^* и нумеруются все его вершины. Выполнение процедуры прекращается, если все компоненты связности исчерпаны.

При практической реализации алгоритма нумерацию можно организовать следующим образом: нумеровать все вершины разделителей S в обратном порядке (от n до 1), как только такое множество будет получено; вершины компонент связности G остаются непронумерованными. Окончательная нумерация вершин (т.е. матрица перестановки) определяется непосредственно после выполнения алгоритма.

Следует отметить, что разделитель S можно выбирать разными способами. Для того, чтобы метод вложенных сечений был максимально эффективным (т.е. в результате переупорядочивания возникали бы большие нулевые блоки в результирующем факторе L), следует придерживаться следующих правил:

- по возможности выбирать минимальный из всех разделителей (идеальный случай – разделитель из одной вершины):
- возникающие при удалении разделителя компоненты связности должны быть примерно одинакового размера.

Известный метод выбора подходящего разделителя, описанный в [9], состоит в построении для графа структуры уровней смежности с корнем в псевдопериферийной вершине и выборе малого разделителя из «среднего» уровня.

Дадим формальное описание метода вложенных сечений. Пусть $G=G(V,E)$ – граф, ассоциированный с матрицей A. Тогда метод вложенных сечений будет состоять в следующем.

1. Положить $R=V$, $n=|V|$.
2. Найти в $G(R)$ связную компоненту $G(C)$ и построить для нее структуру уровней смежности с корнем в псевдопериферийной вершине r, т.е. $\Lambda(r)=\{L_0, L_1, \ldots L_m\}$.
3. Если $m\le 2$, то положить $S=C$ и перейти к шагу 4. В противном случае положить $j=\lfloor (m+1)/2 \rfloor$ и определить разделитель S как множество узлов $S\subset L_j$ такое, что
$$S=\left\{y\in L_j : Adj(y)\cap L_{j+1}\neq\varnothing\right\}$$
4. Перенумеровать узлы разделителя S числами от n до $n-|S|+1$. Положить $R\leftarrow R-S$ и $n\leftarrow n-|S|$. Если $R\neq\varnothing$, то перейти на шаг 2.

На шаге 3 алгоритма разделитель S можно получить простым отбрасыванием тех узлов из L_j, которые не смежны ни с одним узлом из L_{j+1}.

Подробное изучение свойств рассмотренных нами алгоритмов может быть найдено в книгах [9, 10].

3.5.3.4 Результаты экспериментов

Для иллюстрации мы вернемся к разложению рассмотренных в п.3.5.3.2 матриц из коллекции www.cise.ufl.edu/research/sparse/matrices/. Также как и в прошлый раз, сравним коэффициент заполнения исходной матрицы A, фактора L, полученного «в лоб», и фактора L', полученного после применения перестановки, найденной методом вложенных сечений (см. таблицу и рисунки ниже). Для сравнения приведен коэффициент заполнения факт L'', полученного после применения перестановки, найденной методом минимальной степени.

Таблица 3.13. Коэффициенты заполнения матрицы до и после разложения

Матрица	*A*	*L*	*L'*	*L"*
shallow_water2	4,88281E-05	0,00687	0,00065	0,00075
pwtk	0,00024268	0,00803	0,00281	0,00256
parabolic_fem	1,32902E-05	0,16134	0,00019	0,00017
cfd2	0,000202489	0,02049	0,02048	0,00877

Для наглядности приведем портреты матриц, переупорядоченных методом вложенных сечений.

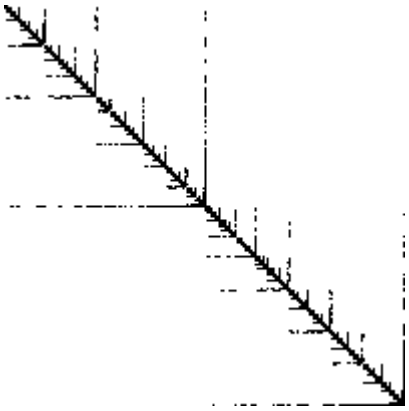


Рис. 3.37. Матрица shallow_water2, *n*=81 920, *nz*=204 800

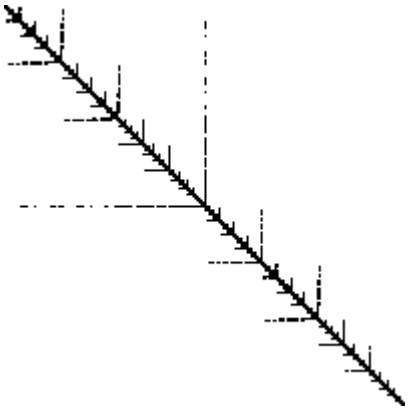


Рис. 3.38. Матрица parabolic_fem, *n*=525 825, *nz*=2 100 225

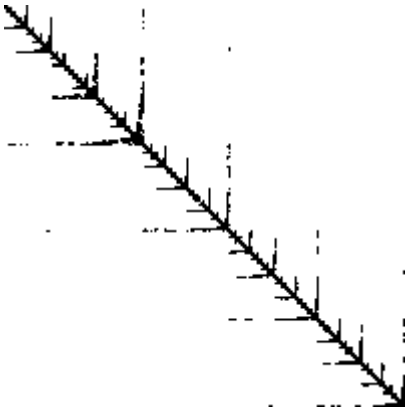


Рис. 3.39. Матрица pwtk,
n=217 918, *nz*=5 871 175

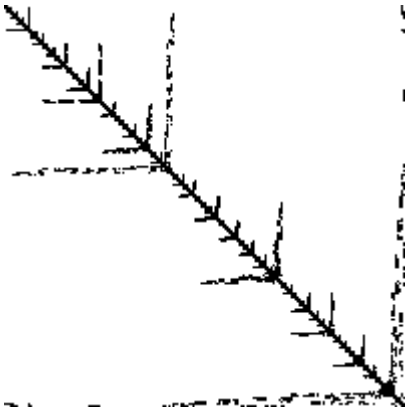


Рис. 3.40. Матрица cfd2,
n=123 440, *nz*=1 604 423

и матрицы факторов L'

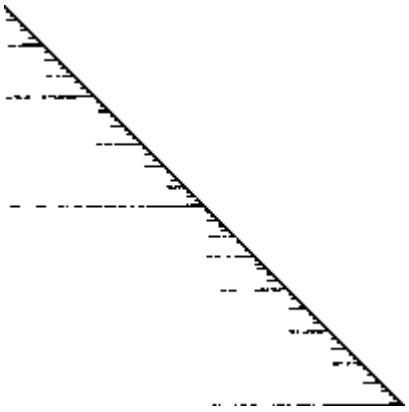


Рис. 3.41. Фактор для shallow_water2,
n=81 920, *nz*=2 183 332

Рис. 3.42. Фактор для parabolic_fem,
n=525 825, *nz*=26 494 693

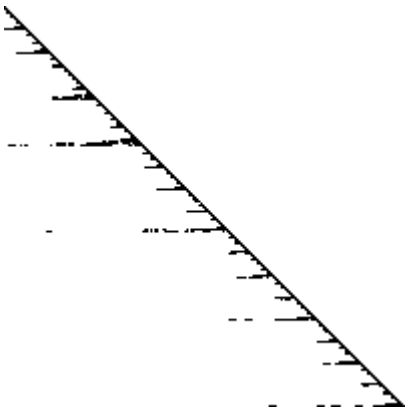


Рис. 3.43. Фактор для pwtk,

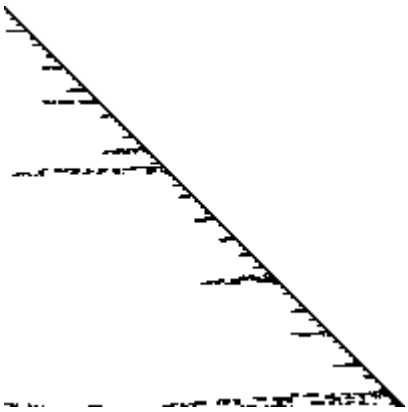


Рис. 3.44. Фактор для cfd2,

n=217 918, *nz*=66 843 598 *n*=123 440, *nz*=156 075 674

Эксперименты показывают, что метод вложенных сечений в целом дает примерно такие же коэффициенты заполнения, что и метод минимальной степени. Значительное отличие есть только для матрицы cfd2, которое объясняется ее особой структурой – элементы матрицы расположены близко к главной диагонали (см. портрет матрицы на рис. 3.24), поэтому перестановка не дает столь значительных (по сравнению с другими примерами) результатов.

3.5.3.5 Символическое и численное разложение

Вернемся теперь к исходной задаче решения системы уравнений

$$Ax=b,$$

Как уже было сказано, при решении задачи с разреженной матрицей A выделяют этапы переупорядочивания, символического разложения, численного разложения и обратного хода.

Методы переупорядочивания, т.е. вычисления матрицы перестановки P с целью минимизации заполнения фактора, были изложены в предыдущем пункте. Сейчас же мы коснемся вопросов, возникающих при символическом разложении, т.е. при построении портрета матрицы L.

Алгоритмические трудности, возникающие при разложении разреженной симметричной матрицы, лучше всего разъяснить с помощью примера. При этом конкретные численные значения нас не будут интересовать; для простоты в матрице, внешний вид которой приведен на рис. 3.45, будут обозначены только позиции ненулевых элементов нижнего треугольника.

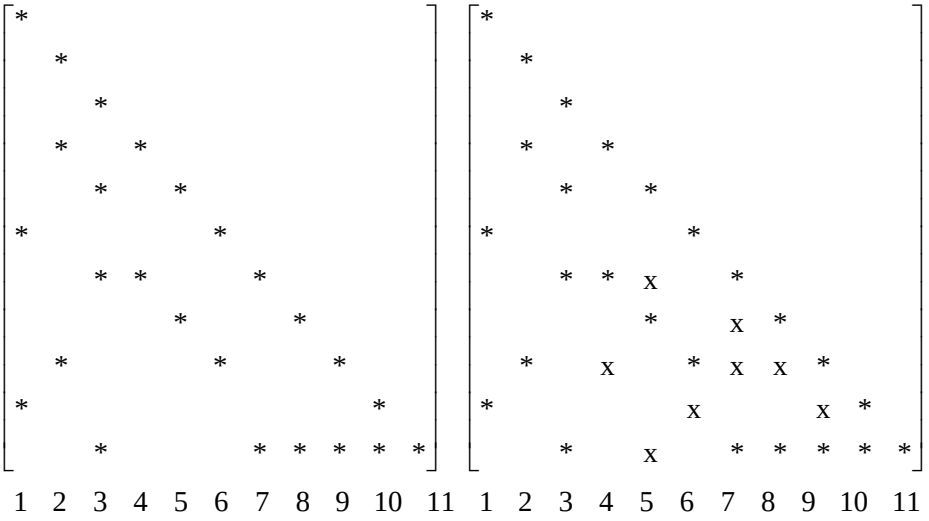


Рис. 3.45. Портрет матрицы A и фактора L

Разложение матрицы из примера начнется с четвертого столбца, поскольку в строках 1-3 все элементы слева от диагонали равны нулю. При этом в столбце 4 появляется новый ненулевой элемент, а именно элемент $a_{9,4}$. На рис. новые ненулевые элементы отмечены символом x. Далее обрабатывается столбец 5; при этом возникнут новые ненулевые элементы $a_{7,5}$ и $a_{11,5}$. Затем обрабатываем столбец 6, что приводит к появлению элемента $a_{10,6}$. Предполагая, что разложение доведено до этого места, мы подробно рассмотрим обработку столбца 7.

Запишем конкретный вид формул (3.13), (3.14) метода Холецкого, по которым будет вычисляться столбец 7. Элемент l_{77} вычисляется по формуле

$$l_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2}\ ,\ i = 7\ .$$

и будет всегда положительный (в силу положительной определенности матрицы A), а остальные элементы вычисляются как

$$l_{ji} = \frac{1}{l_{ii}}\left(a_{ji} - \sum_{k=1}^{i-1} l_{ik} l_{jk} \right),\ i = 7\ ,\ \ j = 8,9,10,11\ . \qquad (3.48)$$

Равенство нулю указанных элементов будет определяться как начальной структурой самого столбца 7, так и структурой предшествующих столбцов. Отметим, что здесь мы не рассматриваем случай обнуления элементов в результате операции вычитания. Если такой случай и произойдет, то будем хранить нуль как элемент матрицы в явном виде.

Следуя формуле вычисления элементов столбца, можно сделать вывод, что портрет столбца 7 при разложении получается как объединение портретов столбцов 1-6 с ненулевыми элементами в 7-й строке в факторе L и портрета исходного столбца 7.

Для удобства последующего изложения будем использовать обозначение A_{i*} для i-й строки матрицы A, и A_{*j} – для j-го столбца. Введем также обозначения для портретов i-й строки и j-го столбца матрицы соответственно

$$struct(A_{i*}) = \{k<i : a_{ik} \neq 0\},$$

$$struct(A_{*j}) = \{k>j : a_{kj} \neq 0\}.$$

Предположим теперь, что все шаги, необходимые для определения портретов нужных столбцов проведены, и мы имеем следующие списки строчных индексов для ненулевых элементов, находящихся в строке 7 слева от диагонали:

$$struct(A_{*3}) = \{7, 11\},\ struct(A_{*4}) = \{7, 9\},$$

$$struct(A_{*5})=\{7, 8, 11\},\ struct(A_{*7})=\{7, 11\}$$

Чтобы получить портрет нового столбца 7, эти списки индексов нужно слить. Для этого можно использовать эффективный метод переменного переключателя [10]. Результатом будет неупорядоченное представление нового столбца 7: 7, 11, 9, 8.

$$struct(A_{*7})=\{7, 11, 9, 8\}$$

В общем случае для определения портрета некоторого j-го столбца приходится выполнить слияние, вообще говоря, j списков, что является трудоемкой операцией.

Замечательный факт, позволяющий существенным образом сократить трудоемкость вычисления портрета j-го столбца, состоит в том [9], что надо объединять не все портреты предшествующих столбцов с номерами 1, 2, …,(j–1), а только те из них, в которых ненулевой элемент в j-й строке является первым ненулевым элементом данного столбца ниже диагонали.

В рассмотренном примере для вычисления портрета столбца 7 нужно объединить лишь портреты столбцов 4, 5 и 7, при этом портрет столбца 5 уже будет содержать в себе все ненулевые элементы столбца 3:

$$struct(A_{*4})=\{7, 9\},\ struct(A_{*5})=\{7, 8, 11\},\ struct(A_{*7})=\{7, 11\}$$

При их слиянии получаем портрет столбца 7

$$struct(A_{*7})=\{7, 9, 8, 11\}.$$

Заметим, символический этап разложения не требует, чтобы представления столбцов были упорядоченными. В самом деле, мы должны лишь регистрировать, у каких столбцов первый ненулевой элемент под диагональю находится в данной строке. Это означает, что остальные ненулевые элементы таких столбцов расположены ниже рассматриваемой строки, и чтобы сформировать портрет нового столбца нам нужны все эти элементы, каков бы ни был их порядок.

Таким образом, символический этап разложения исключения не требует упорядоченности представления. Это выгодно для нас, поскольку, как видно из примера, символический алгоритм порождает неупорядоченные портреты.

Для регистрации столбцов, имеющих первый ненулевой элемент в данной строке, мы должны сопоставить каждой строке свое множество столбцов. Например, столбцы 4 и 5 принадлежат множеству, ассоциированному со строкой 7. При практической реализации наименьший строчный индекс каждого столбца определяется одновременно с формированием его портрета, после чего к множеству добавляется номер данного столбца. Например, при сборке портрета столбца 7 выясняется, что ниже диагонали наимень-

ший строчный индекс равен 8, и во множество, ассоциированное со строкой 8, вставляется номер 7. При обработке столбца 8 в соответствующем множестве будет обнаружен индекс 7 и будет использован портрет столбца 7.

Рассмотрим теперь этап численного разложения. Предположим, что символическое разложение и упорядочение портрета уже проделаны. Отметим, что упорядочить представление можно, дважды применив к матрице алгоритм символического транспонирования [10].

Вернемся опять к рассмотрению примера, изображенного на рис. 3.45. Предположим, что численное разложение выполнено вплоть до столбца 6, и исследуем процесс обработки столбца 7. Мы знаем, что ненулевые элементы столбца 7 расположены в строках 7, 8, 9 и 11. Чтобы вычислить их значения, нужно просмотреть строку 7, установить, что столбцы 3, 4 и 5 имеют ненулевые элементы в этой строке, найти в этих столбцах ненулевые элементы, строчные индексы которых не меньше 7, наконец, провести умножение и вычитание в соответствии с формулой (3.48). Рассмотрим одну из этих операций подробно (выполнение остальных операций основано на схожих идеях).

Предполагаем, что матрица хранится в разреженном столбцовом формате. Пусть нужно найти элементы данного столбца (например, столбца 3), у которого строчные индексы больше или равны номеру обрабатываемого столбца (например, столбца 7). Для описания столбца 3 в массивах *values* и *rows* начальной и конечной позициями являются *pointer[3]* и *pointer[4]–1*. Столбец 3 упорядочен:

$$struct(A_{*3})=\{5, 7, 11\}.$$

Однако нам нужна только та часть столбца 3, которая начинается со строки 7. Поэтому требуется другой указатель начала (например, *pointerP[3]*); в то же время *pointer[4]–1* по-прежнему можно использовать как указатель конца. В момент обработки столбца 7 *pointerP[3]* указывает позицию ненулевого элемента столбца 3, стоящего в строке 7. После того как столбец 3 использован для обработки столбца 7, значение *pointerP[3]* увеличивается на единицу. Так как столбец 3 упорядочен, *pointerP[3]* будет теперь указывать позицию следующего ненулевого элемента столбца 3, т. е. элемента из строки 11; это значение сохранится до обработки столбца 11. Дело в том, что при обработке столбцов 8, 9, 10 столбец 3 не будет использован, поскольку не имеет ненулевых элементов в строках 8, 9 и 10, а значит и значение *pointerP[3]* не будет изменено.

Подробное рассмотрение последовательных алгоритмов символического и численного разложения может быть найдено в книге [10].

3.5.3.6 Организация параллельных вычислений

Сейчас давайте обсудим, какие дополнительные возможности для распараллеливания возникают в случае разложения разреженных матриц по сравнению с плотными. Для этого нам опять потребуются некоторые сведения из теории графов.

В п. 3.5.3.1 было введено понятие графа $G(A)$, соответствующего матрице A. Рассмотрим также *граф заполнения* $F(A)$, который определяется как $G(A)$ с дополнительными ребрами, соединяющими вершины i и j, для которых $l_{ij}\neq 0$. Иными словами, $F(A)=G(L+L^T)$, т.е. это граф с матрицей смежности, образованной из фактора Холецкого L.

Для иллюстрации ниже приведена матрица L, и соответствующей ей граф заполнения $F(A)$. Дуги графа, которые соответствуют ребрам заполнения, выделены пунктиром.

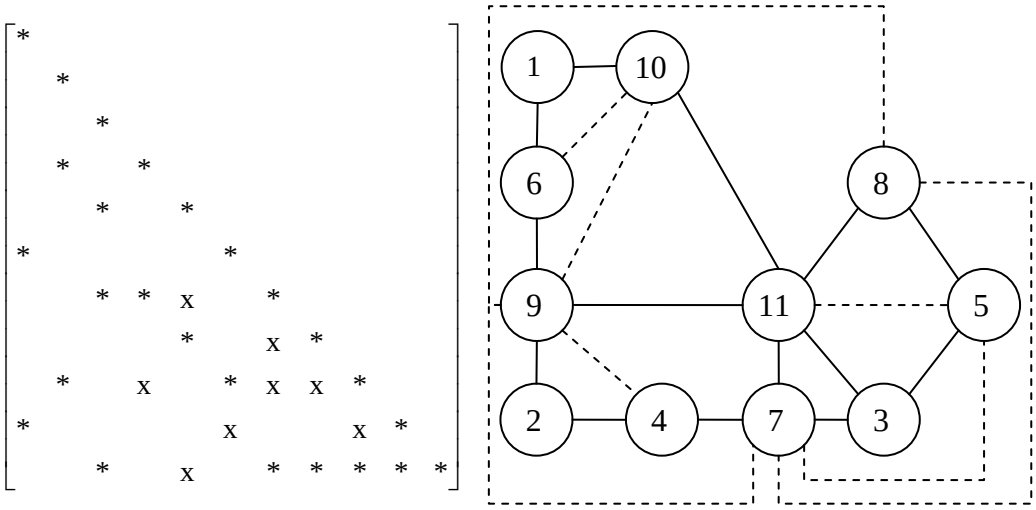


Рис. 3.46. Матрица L и граф заполнения $F(A)$

Используя введенное ранее обозначение для портрета столбца, определим на графе заполнения функцию *parent* следующим образом:

$$parent(j)=\begin{cases}\min\{i\in struct(L_{*j})\}, \text{если } struct(L_{*j})\neq\varnothing,\\ j, \text{иначе}.\end{cases}$$

Введенная нами функция определяет строчный индекс первого ненулевого внедиагонального элемента в столбце j матрицы L.

Определим теперь *дерево исключения* $T(A)$ как граф с n вершинами с дугами между вершинами i и j ($i>j$), где $i=parent(j)$.

Дерево исключения $T(A)$ является остовным деревом графа заполнения $F(A)$ с корнем в n-й вершине.

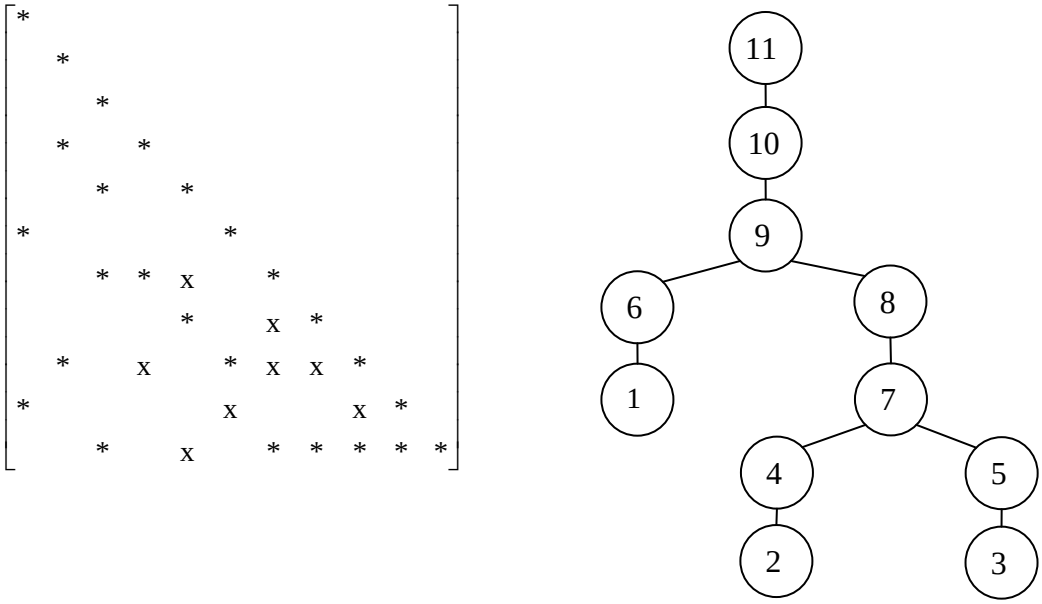


Рис. 3.47. Матрица L и дерево исключения $T(A)$

Отметим, что на этапе символического разложения фактически происходит построение дерева исключения $T(A)$, специальная процедура для его построения не требуется.

Важность дерева исключения состоит в том, что оно определяет зависимость по данным между столбцами матрицы L при численном разложении. Значения столбца с номером, соответствующим некоторой вершине дерева, зависят от всех столбцов, номера которых содержатся в поддереве с корнем в данной вершине. При этом если T_i и T_j – непересекающиеся поддеревья дерева исключения $T(A)$, то значения элементов столбцов с номерами, соответствующими вершинам поддерева T_i, не зависят от значений столбцов с номерами, соответствующими вершинам поддерева T_j. Указанное свойство дает возможность вычислить значения таких столбцов параллельно.

Например, в соответствии с деревом исключения, представленном на рис. 3.47, можно параллельно вычислить столбцы L_{*1}, L_{*2}, L_{*3}, затем – столбцы L_{*4}, L_{*5}, L_{*6}, и т.д.

Более того, дерево исключения определяет зависимость по данным и при проведении обратного хода. Для непересекающиеся поддеревьев T_i и T_j де-

рева исключения $T(A)$ значения неизвестных с номерами, соответствующими вершинам поддерева T_i, не зависят от значений неизвестных с номерами, соответствующими вершинам поддерева T_j, что позволяет выполнить обратный ход параллельно.

Конечно, при однократном решении СЛАУ распараллеливание обратного хода не дает ощутимого эффекта в силу его малой (по сравнению с прямым ходом) трудоемкости. Однако при решении серии СЛАУ с одинаковой матрицей и разными правыми частями ускорение может быть существенным.

Дерево исключения характеризует параллельный алгоритм разложения в целом, при этом высота дерева исключения является оценкой времени работы алгоритма (чем выше дерево, тем больше времени будет работать алгоритм), а ширина – оценкой степени параллелизма в задаче (чем шире дерево – тем больше степень параллелизма). Поэтому можно сказать, что для хорошего распараллеливания нужно иметь широкое, сбалансированное, относительно невысокое дерево исключения.

Для построения переупорядоченной матрицы A, обладающей деревом исключения $T(A)$ с указанными свойствами, наиболее подходит метод вложенных сечений. В самом деле, на каждом шаге метода выбирается разделитель S, который соответствует одной (или нескольким последовательно связанным) вершинам дерева исключения T_s, а подмножества, которые получились в результате разделения графа, соответствуют поддеревьям с корнем в T_s. Указанные поддеревья будут содержать примерно одинаковое количество узлов, что обеспечит хорошую сбалансированность дерева исключения.

Рассмотренный нами метод минимальной степени тоже можно применять для переупорядочивания матрицы A в параллельном алгоритме, однако при этом не будет гарантии хороших свойств дерева исключения. Это подтверждают приведенные ниже результаты экспериментов.

3.5.3.7 Результаты экспериментов

Для иллюстрации мы опять вернемся к разложению матриц из п.3.5.3.2, и рассмотрим методы переупорядочивания с точки зрения их применимости в параллельном алгоритме.

Одним из свойств, характеризующих применимость перестановки для параллельного алгоритма, является высота дерева исключения. В табл. 3.14 приведены высоты деревьев исключения, получающихся после работы методов минимальной степени (***M**inimum **D**egree*) и вложенных сечений (***N**ested **D**issection*) для рассматриваемых нами матриц.

Таблица 3.14. Высота дерева исключения
при использовании методов ND и MD

	ND	MD
shallow_water2	930	1459
Pwtk	5823	8062
parabolic_fem	2826	4891
cfd2	6802	10723

Результаты, приведенные в таблице, подтверждают, что переупорядочивание по методу вложенных сечений дает дерево исключения с меньшей высотой, чем с использованием метода минимальной степени. Поэтому в дальнейших экспериментах мы будем использовать перестановку, полученную методом вложенных сечений.

Теперь рассмотрим время работы последовательного алгоритма при проведении символического разложения (обозначим это время как T_1), численного разложения (T_2) и обратного хода (T_3).

Таблица 3.15. Время работы последовательных алгоритмов символического, численного разложения и обратного хода

	T_1	T_2	T_3
shallow_water2	0,12	0,37	0,01
pwtk	4,96	53,87	0,26
parabolic_fem	1,62	7,78	0,12
cfd2	4,78	91,04	0,23

Очевидно, что основные временные затраты приходятся на этап численного разложения, поэтому именно данный этап нуждается в распараллеливании. Также можно распараллеливать и обратный ход, т.к. во многих задачах требуется решать серию систем с одинаковым портретом, но разными значениями матрицы, что требует однократного проведения символьного разложения и многократного проведения численного разложения и обратного хода.

Распараллеливание численного разложения и обратного хода будем проводить, выделяя непересекающиеся поддеревья в дереве исключения и параллельно обрабатывая соответствующие столбцы матрицы. С целью определения оптимальной высоты выделяемых поддеревьев проведем серию экспериментов на матрице parabolic_fem, в которой будем варьировать высота в диапазоне от 50 до 100 (напомним, что общая высота дерева исключения – 2826); результаты экспериментов приведены ниже.

Таблица 3.16. Время работы параллельного численного разложения для матрицы parabolic_fem

Высота поддерева	1 поток	Параллельное численное разложение							
		2 потока		4 потока		6 потоков		8 потоков	
		t	*S*	*t*	*S*	*t*	*S*	*t*	*S*
50	8,36	5,37	1,56	5,32	1,57	4,13	2,02	5,46	1,53
75	8,47	5,90	1,44	4,51	1,88	4,66	1,82	4,24	2,00
100	8,36	5,35	1,56	4,40	1,90	4,60	1,82	4,65	1,80

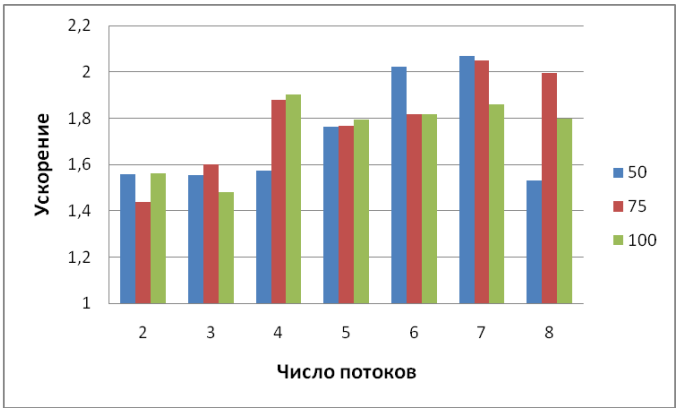


Рис. 3.48. Ускорение параллельного численного разложения

Таблица 3.17. Время работы параллельного обратного хода для матрицы parabolic_fem

Высота поддерева	1 поток	Параллельный обратный ход							
		2 потока		4 потока		6 потоков		8 потоков	
		t	*S*	*t*	*S*	*t*	*S*	*t*	*S*
50	0,15	0,08	1,82	0,07	2,30	0,06	2,63	0,05	2,87
60	0,15	0,09	1,69	0,06	2,49	0,06	2,39	0,06	2,57
100	0,17	0,09	1,85	0,06	2,68	0,06	2,74	0,05	3,06

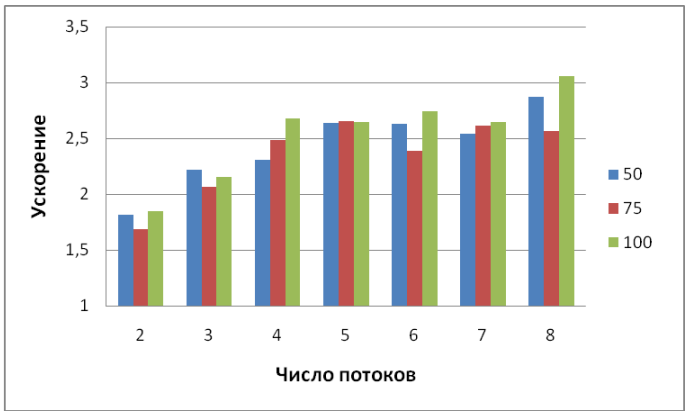


Рис. 3.49. Ускорение параллельного обратного хода

Эксперименты, проведенные на матрице parabolic_fem, показывают, что наибольшее ускорение обратного хода достигается при использовании поддеревьев высоты 100, а при численном разложении поддеревья высоты 75 и 100 дают примерно одинаковое ускорение, поэтому дальнейшие эксперименты будем проводить при использовании высоты 100.

Таблица 3.18. Время работы параллельного численного разложения

Матрица	1 поток	Параллельное численное разложение							
		2 потока		4 потока		6 потоков		8 потоков	
		t	S	t	S	t	S	t	S
shallow_water2	0,34	0,23	1,47	0,22	1,57	0,20	1,69	0,22	1,57
pwtk	61,42	36,94	1,66	32,76	1,87	32,26	1,90	31,54	1,95
parabolic_fem	8,36	5,35	1,56	4,40	1,90	4,60	1,82	4,65	1,80
cfd2	86,052	74,9	1,15	67,4	1,28	68,8	1,25	62,6	1,38

Таблица 3.19. Время работы параллельного обратного хода

Матрица	1 поток	Параллельный обратный ход							
		2 потока		4 потока		6 потоков		8 потоков	
		t	S	t	S	t	S	t	S
shallow_water2	0,01	0,01	1,67	0,01	2,21	0,01	2,16	0,01	2,30
pwtk	0,25	0,15	1,63	0,13	1,93	0,13	1,98	0,12	2,06
parabolic_fem	0,17	0,09	1,85	0,06	2,68	0,06	2,74	0,05	3,06
cfd2	0,28	0,2	1,64	0,1	2,26	0,2	1,79	0,2	1,50

Низкие показатели ускорения в данных задачах (максимальный – 3 при проведении обратного хода и 1,9 при численном разложении на 8 потоках) можно объяснить разбалансированностью дерева исключения. В задачах со сбалансированным деревом исключения можно ожидать больших показателей ускорения.

Литература

Использованные источники информации

1. Вержбицкий В.М. Численные методы (математический анализ и обыкновенные дифференциальные уравнения). – М.: Высшая школа, 2001.
2. Бахвалов Н.С., Жидков Н.П., Кобельков Г.М. Численные методы. – М.: Наука, 1987.
3. Тихонов А.Н., Самарский А.А. Уравнения математической физики. – М.: Наука, 1977.
4. Самарский А.А., Гулин А.В. Численные методы. – М.: Наука, 1989.
5. Самарский А.А. Введение численные методы. – СПб.: Лань, 2005.
6. Калиткин Н.Н. Численные методы. – М.: Наука, 1978
7. Хамахер К., Вранешич З., Заки С. Организация ЭВМ. –СПб: Питер, 2003.
8. Голуб Дж., Ван Лоун Ч. Матричные вычисления. – М.: Мир, 1999.
9. Джордж А., Лю Дж. Численное решение больших разреженных систем уравнений. – М.: Мир, 1984.
10. Писсанецки С. Технология разреженных матриц. — М.: Мир,1988.
11. Gustavson F. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition // ACM Transactions on Mathematical Software (TOMS), Volume 4 Issue 3, Sept. 1978. – Pp. 250-269.
12. Соболь И.М. Численные методы Монте-Карло. – М.: Наука, 1973.
13. Соболь И.М. Точки, равномерно заполняющие многомерный куб. – М.: Знание, 1985.
14. Д. Кнут. Искусство программирования. Том 2: получисленные алгоритмы. – М.: «Вильямс», 2007.
15. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем. – Н.Новгород, Изд-во ННГУ, 2003.
16. Гергель В.П. Теория и практика параллельных вычислений. – М.: БИНОМ, 2007.
17. Белов С.А., Золотых Н.Ю. Численные методы линейной алгебры. – Н.Новгород, Изд-во ННГУ, 2005.

18. J. Dongarra et al. Templates for the solution of linear systems: building blocks for iterative methods. SIAM, 1994.

19. G. Karniadakis, R. Kirby. Parallel scientific computing in C++ and MPI. Cambridge university press, 2003.

20. M. Quinn. Parallel programming in C with MPI and OpenMP. McGraw-Hill, 2004.

Дополнительная литература

21. P. Amestoy, T. Davis, and Iain S. Duff. An approximate minimum degree ordering algorithm. SIAM Journal on matrix analysis and applications, 17 (1996), pp. 886-905.

22. J.W.H. Liu. Modification of the Minimum-Degree algorithm by multiple elimination. ACM Transactions on Mathematical Software, 11(2):141-153, 1985.

23. Ширяев А. Н. Вероятность, – М.: Наука. 1989.

24. Metropolis N., Ulam S. The Monte Carlo method, J. Amer. statistical assoc., 1949, 44, N247, 335-341.

25. O. Percus, M. Kalos. Random number generators for MIMD parallel processors// Journal of parallel and distributed computing, v.6, 1989. pp. 477–479.

26. M. Matsumoto, T. Nishimura (1998). «Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator». ACM Trans. on Modeling and Computer Simulations v. 8(1).

27. M. Mascagni, A. Srinivasan. Algorithm 806: SPRNG: A scalable library for pseudorandom number generation. ACM Transactions on Mathematical Software, v. 26, № 3, 2000. pp. 436–461.

28. Niederreiter H. Random Number Generation and Quasi-Monte Carlo Methods. – SIAM, 1992. – 247 p.

Информационные ресурсы сети Интернет

29. Intel Math Kernel Library Reference Manual.

[http://software.intel.com/sites/products/documentation/hpc/mkl/mklman.pdf].