8. Матричное умножение

Операция умножения матриц является одной из основных задач матричных вычислений. В данном разделе рассматриваются несколько разных параллельных алгоритмов для выполнения этой операции. Два из них основаны на ленточной схеме разделения данных. Другие два метода основаны на блочной схеме разделения - это широко известные алгоритмы Фокса (Fox) и Кэннона ($Cannon$).

8.1. Постановка задачи

Умножение матрицы A размера $m \times n$ и матрицы B размера $n \times l$ приводит к получению матрицы C размера $m \times l$, каждый элемент которой определяется в соответствии с выражением:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l. \quad (8.1)$$

Как следует из (8.1), каждый элемент результирующей матрицы C есть скалярное произведение соответствующих строки матрицы A и столбца матрицы B:

$$c_{ij} = \left(a_i, b_j^T\right), a_i = \left(a_{i0}, a_{i1}, ..., a_{in-1}\right), b_j^T = \left(b_{0j}, b_{1j}, ..., b_{n-1j}\right)^T. \quad (8.2)$$

Этот алгоритм предполагает выполнение $m \cdot n \cdot l$ операций умножения и столько же операций сложения элементов исходных матриц. При умножении квадратных матриц размера $n \times n$ количество выполненных операций имеет порядок $O(n^3)$. Известны последовательные алгоритмы умножения матриц, обладающие меньшей вычислительной сложностью (например, алгоритм Штрассена ($Strassen's\ algorithm$)), но эти алгоритмы требуют определенных усилий для их освоения и, как результат, в данном разделе при разработке параллельных методов в качестве основы будет использоваться приведенный выше

последовательный алгоритм. Также будем предполагать далее, что все матрицы являются квадратными и имеют размер $n \times n$.

8.2. Последовательный алгоритм

Последовательный алгоритм умножения матриц представляется тремя вложенными циклами:

```
// Алгоритм 8.2
// Последовательный алгоритм умножения матриц
double MatrixA[Size][Size];
double MatrixB[Size][Size];
double MatrixC[Size][Size];
int i,j,k;
...
for (i=0; i<Size; i++){
  for (j=0; j<Size; j++){
    MatrixC[i][j] = 0;
    for (k=0; k<Size; k++){
      MatrixC[i][j] = MatrixC[i][j] + MatrixA[i][k]*MatrixB[k][j];
    }
  }
}
```

Алгоритм 8.1. Последовательный алгоритм умножения двух квадратных матриц

Этот алгоритм является итеративным и ориентирован на последовательное вычисление строк матрицы C. Действительно, при выполнении одной итерации внешнего цикла (цикла по переменной i) вычисляется одна строка результирующей матрицы (см. рис. 8.1)

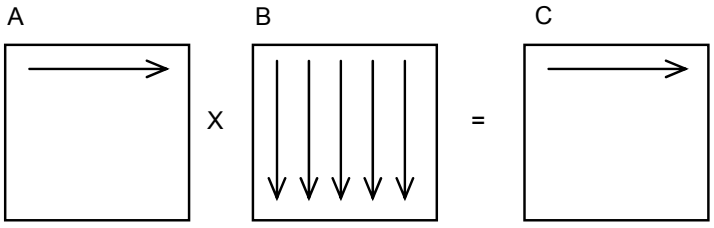

Рис. 8.1. На первой итерации цикла по переменной *i* используется первая строка матрицы *A* и все столбцы матрицы *B* для того, чтобы вычислить элементы первой строки результирующей матрицы *C*

Поскольку каждый элемент результирующей матрицы есть скалярное произведение строки и столбца исходных матриц, то для вычисления всех элементов матрицы C размером $n \times n$ необходимо выполнить $n^2 \cdot (2n-1)$ скалярных операций и затратить время

$$T_1 = n^2 \cdot (2n-1) \cdot \tau \qquad (8.3)$$

где τ есть время выполнения одной элементарной скалярной операции.

8.3. Умножение матриц при ленточной схеме разделения данных

Рассмотрим два параллельных алгоритма умножения матриц, в которых матрицы A и B разбиваются на непрерывные последовательности строк или столбцов (*полосы*).

8.3.1. Определение подзадач

Из определения операции матричного умножения следует, что вычисление всех элементов матрицы C может быть выполнено независимо друг от друга. Как результат, возможный подход для организации параллельных вычислений состоит в использовании в качестве базовой подзадачи процедуры определения одного элемента результирующей матрицы C. Для проведения всех необходимых вычислений каждая подзадача должна содержать по одной строке матрицы A и одному столбцу матрицы B. Общее количество получаемых при таком подходе подзадач оказывается равным n^2 (по числу элементов матрицы C).

Рассмотрев предложенный подход, можно отметить, что достигнутый уровень параллелизма является в некоторой степени избыточным. Обычно при проведении практических расчетов количество сформированных подзадач превышает число имеющихся процессоров и, как результат, неизбежным является этап укрупнения базовых задач. В этом плане может оказаться полезным агрегация вычислений уже на шаге выделения базовых подзадач. Возможное решение может состоять в объединении в рамках

одной подзадачи всех вычислений, связанных не с одним, а с несколькими элементами результирующей матрицы C. Для дальнейшего рассмотрения в рамках данного подраздела определим базовую задачу как процедуру вычисления всех элементов одной из строк матрицы C. Такой подход приводит к снижению общего количества подзадач до величины n.

Для выполнения всех необходимых вычислений базовой подзадаче должны быть доступны одна из строк матрицы A и все столбцы матрицы B. Простое решение этой проблемы – дублирование матрицы B во всех подзадачах – является, как правило, неприемлемым в силу больших затрат памяти для хранения данных. Как результат, организация вычислений должна быть построена таким образом, чтобы в каждый текущий момент времени подзадачи содержали лишь часть данных, необходимых для проведения расчетов, а доступ к остальной части данных обеспечивался бы при помощи передачи сообщений. Два возможных способа выполнения параллельных вычислений подобного типа рассмотрены далее в п. 8.3.2.

8.3.2. Выделение информационных зависимостей

Для вычисления одной строки матрицы C необходимо, чтобы в каждой подзадаче содержалась строка матрицы A и был обеспечен доступ ко всем столбцам матрицы B. Возможные способы организации параллельных вычислений состоят в следующем.

1. Первый алгоритм. Алгоритм представляет собой итерационную процедуру, количество итераций которой совпадает с числом подзадач. На каждой итерации алгоритма каждая подзадача содержит по одной строке матрицы A и одному столбцу матрицы B. При выполнении итерации проводится скалярное умножение содержащихся в подзадачах строк и столбцов, что приводит к получению соответствующих элементов результирующей матрицы C. По завершении вычислений в конце каждой итерации столбцы матрицы B должны быть переданы между подзадачами с тем, чтобы в каждой подзадаче оказались новые столбцы матрицы B и могли быть вычислены новые элементы матрицы C. При этом данная передача столбцов между подзадачами должна быть организована таким образом, чтобы после завершения итераций алгоритма в каждой подзадаче последовательно оказались все столбцы матрицы B.

Возможная простая схема организации необходимой последовательности передач столбцов матрицы B между подзадачами состоит в представлении топологии информационных связей подзадач в виде кольцевой структуры. В этом случае, на каждой итерации подзадача i, $0 \le i < n$, будет передавать свой столбец матрицы B подзадаче с номером $i+1$ (в соответствии с кольцевой структурой подзадача $n-1$ передает свои данные подзадаче с номером 0) – см. рис. 8.2. После выполнения всех итераций алгоритма необходимое условие будет обеспечено – в каждой подзадаче поочередно окажутся все столбцы матрицы B.

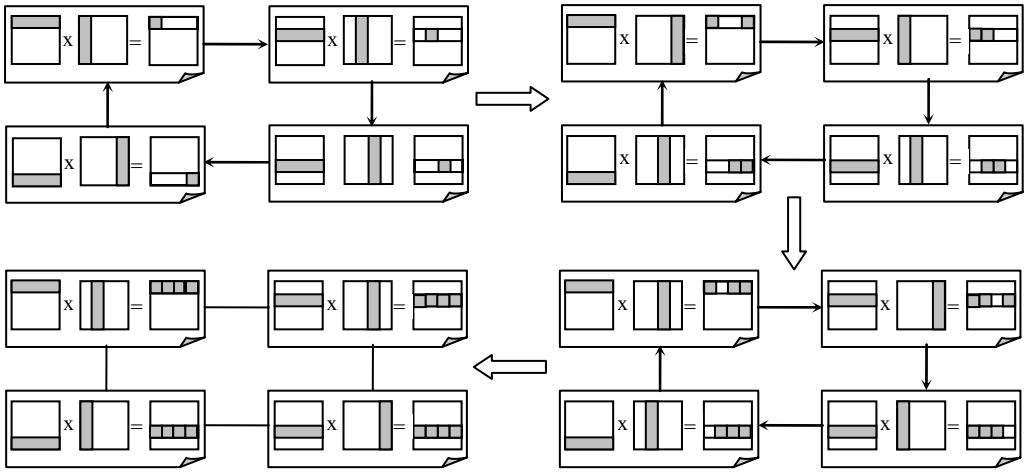

Рис. 8.2. Общая схема передачи данных для первого параллельного алгоритма матричного умножения при ленточной схеме разделения данных

На рис. 8.2 представлены итерации алгоритма матричного умножения для случая, когда матрицы состоят из четырех строк и четырех столбцов ($n=4$). В начале вычислений в каждой подзадаче i, $0 \le i < n$, располагаются i строка матрицы A и i столбец матрицы B. В результате их перемножения подзадача получает элемент c_{ii} результирующей матрицы C. Далее подзадачи осуществляют обмен столбцами, в ходе которого каждая подзадача передает свой столбец матрицы B следующей подзадаче в соответствии с кольцевой структурой информационных взаимодействий. Далее выполнение описанных действий повторяется до завершения всех итераций параллельного алгоритма.

2. Второй алгоритм. Отличие второго алгоритма состоит в том, что в подзадачах располагаются не столбцы, а строки матрицы B. Как результат, перемножение данных каждой подзадачи сводится не к скалярному умножению имеющихся векторов, а к их поэлементному умножению. В результате подобного умножения в каждой подзадаче получается строка частичных результатов для матрицы C.

При рассмотренном способе разделения данных для выполнения операции матричного умножения нужно обеспечить последовательное получение в подзадачах всех строк матрицы *B*, поэлементное умножение данных и суммирование вновь получаемых значений с ранее вычисленными результатами. Организация необходимой последовательности передач строк матрицы *B* между подзадачами также может быть выполнена с использованием кольцевой структуры информационных связей (см. рис. 8.3).

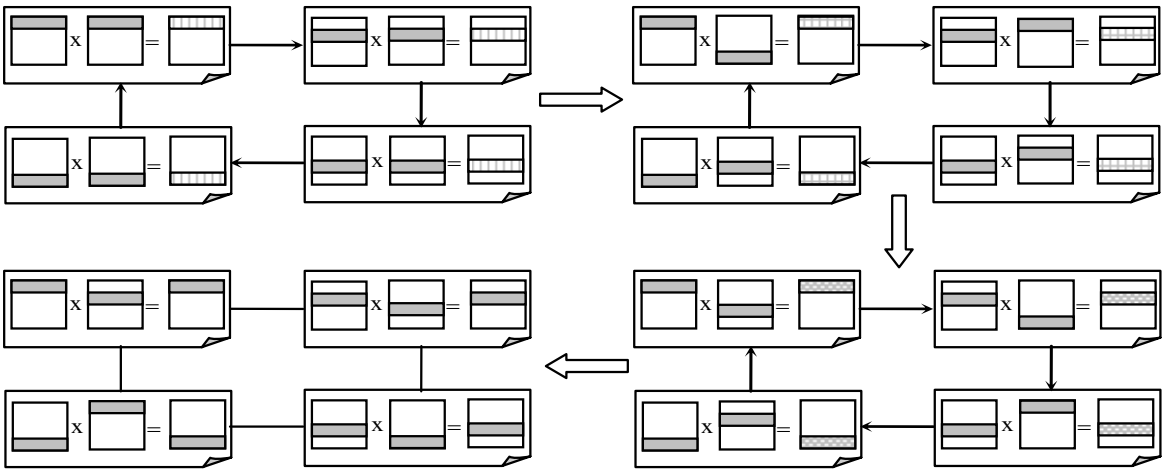

Рис. 8.3. Общая схема передачи данных для второго параллельного алгоритма матричного умножения при ленточной схеме разделения данных

На рис. 8.3 представлены итерации алгоритма матричного умножения для случая, когда матрицы состоят из четырех строк и четырех столбцов ($n=4$). В начале вычислений в каждой подзадаче *I*, $0 \le i < n$, располагаются i строки матрицы *A* и матрицы *B*. В результате их перемножения подзадача определяет i строку частичных результатов искомой матрицы *C*. Далее подзадачи осуществляют обмен строками, в ходе которого каждая подзадача передает свою строку матрицы *B* следующей подзадаче в соответствии с кольцевой структурой информационных взаимодействий. Далее выполнение описанных действий повторяется до завершения всех итераций параллельного алгоритма.

8.3.3. Масштабирование и распределение подзадач по процессорам

Выделенные базовые подзадачи характеризуются одинаковой вычислительной трудоемкостью и равным объемом передаваемых данных. В случае, когда размер матриц n оказывается больше, чем число процессоров p, базовые подзадачи можно укрупнить, объединив в рамках одной подзадачи несколько соседних строк и столбцов перемножаемых матриц. В этом случае, исходная матрица *A* разбивается на ряд горизонтальных полос, а матрица *B* представляется в виде набора вертикальных (для первого алгоритма) или горизонтальных (для второго алгоритма) полос. Размер полос при этом следует выбрать равным $k=n/p$ (в предположении, что n кратно p), что позволит по-прежнему обеспечить равномерность распределения вычислительной нагрузки по процессорам, составляющим многопроцессорную вычислительную систему.

Для распределения подзадач между процессорами может быть использован любой способ, обеспечивающий эффективное представление кольцевой структуры информационного взаимодействия подзадач. Для этого достаточно, например, чтобы подзадачи, являющиеся соседними в кольцевой топологии, располагались на процессорах, между которыми имеются прямые линии передачи данных.

8.3.4. Анализ эффективности

Выполним анализ эффективности первого параллельного алгоритма умножения матриц.

Общая трудоемкость последовательного алгоритма, как уже отмечалось ранее, является пропорциональной n^3. Для параллельного алгоритма на каждой итерации каждый процессор выполняет умножение имеющихся на процессоре полос матрицы *A* и матрицы *B* (размер полос равен n/p и, как результат, общее количество выполняемых при этом умножении операций равно n^3/p^2). Поскольку число итераций алгоритма совпадает с количеством процессоров, сложность параллельного алгоритма без учета затрат на передачу данных может быть определена при помощи выражения

$$T_p = (n^3 / p^2) \cdot p = n^3 / p . \qquad (8.4)$$

С учетом этой оценки показатели ускорения и эффективности данного параллельного алгоритма матричного умножения принимают вид:

$$S_p = \frac{n^3}{(n^3/p)} = p \quad \text{и} \quad E_p = \frac{n^3}{p \cdot (n^3/p)} = 1. \tag{8.5}$$

Таким образом, общий анализ сложности дает идеальные показатели эффективности параллельных вычислений. Для уточнения полученных соотношений оценим более точно количество вычислительных операций алгоритма и учтем затраты на выполнение операций передачи данных между процессорами.

С учетом числа и длительности выполняемых операций время выполнения вычислений параллельного алгоритма может быть оценено следующим образом:

$$T_p(calc) = (n^2 / p) \cdot (2n-1) \cdot \tau \tag{8.6}$$

(здесь, как и ранее, τ есть время выполнения одной элементарной скалярной операции).

Для оценки коммуникационной сложности параллельных вычислений будем предполагать, что все операции передачи данных между процессорами в ходе одной итерации алгоритма могут быть выполнены параллельно. Объем передаваемых данных между процессорами определяется размером полос и составляет *n/p* строк или столбцов длины *n*. Общее количество параллельных операций передачи сообщений на единицу меньше числа итераций алгоритма (на последней итерации передача данных не является обязательной). Тем самым, оценка трудоемкости выполняемых операций передачи данных может быть определена как

$$T_p(comm) = (p-1) \cdot (\alpha + w \cdot n \cdot (n/p)/\beta), \tag{8.7}$$

где α – латентность, β – пропускная способность сети передачи данных, а *w* есть размер элемента матрицы в байтах.

С учетом полученных соотношений общее время выполнения параллельного алгоритма матричного умножения определяется следующим выражением:

$$T_p = (n^2 / p)(2n-1) \cdot \tau + (p-1) \cdot (\alpha + w \cdot n \cdot (n/p)/\beta). \tag{8.8}$$

8.3.5. Результаты вычислительных экспериментов

Эксперименты проводились на вычислительном кластере на базе процессоров Intel XEON 4 EM64T, 3000 Мгц и сети Gigabit Ethernet под управлением операционной системы Microsoft Windows Server 2003 Standart x64 Edition (см. п. 1.2.3).

Для оценки длительности τ базовой скалярной операции проводилось решение задачи умножения матриц при помощи последовательного алгоритма и полученное таким образом время вычислений делилось на общее количество выполненных операций – в результате подобных экспериментов для величины τ было получено значение 6.4 нсек. Эксперименты, выполненные для определения параметров сети передачи данных, показали значения латентности α и пропускной способности β соответственно 130 мкс и 53.29 Мбайт/с. Все вычисления производились над числовыми значениями типа double, т. е. величина *w* равна 8 байт.

Результаты вычислительных экспериментов приведены в таблице 8.1. Эксперименты выполнялись с использованием двух, четырех и восьми процессоров.

Таблица 8.1. Результаты вычислительных экспериментов по исследованию первого параллельного алгоритма матричного умножения при ленточной схеме распределения данных

Размер матриц	Последовательный алгоритм	2 процессора		4 процессора		8 процессоров	
		Время	Ускорение	Время	Ускорение	Время	Ускорение
500	0,8752	0,3758	2,3287	0,1535	5,6982	0,0968	9,0371
1000	12,8787	5,4427	2,3662	2,2628	5,6912	0,6998	18,4014
1500	43,4731	20,9503	2,0750	11,0804	3,9234	5,1766	8,3978
2000	103,0561	45,7436	2,2529	21,6001	4,7710	9,4127	10,9485
2500	201,2915	99,5097	2,0228	56,9203	3,5363	18,3303	10,9813
3000	347,8434	171,9232	2,0232	111,9642	3,1067	45,5482	7,6368

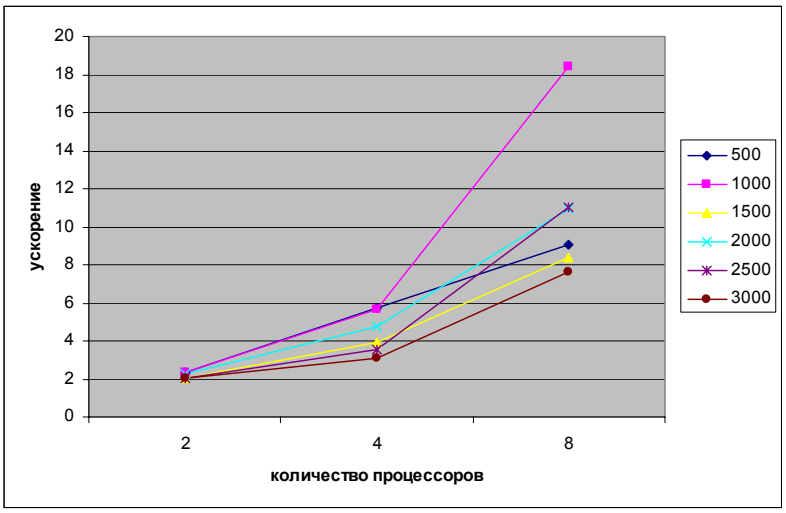

Рис. 8.4. Зависимость ускорения от количества процессоров при выполнении первого параллельного алгоритма матричного умножения при ленточной схеме распределения данных

Сравнение экспериментального времени T_p^* выполнения эксперимента и теоретического времени T_p из формулы (8.8) представлено в таблице 8.2 и на рис. 8.5.

Таблица 8.2. Сравнение экспериментального и теоретического времени выполнения первого параллельного алгоритма матричного умножения при ленточной схеме распределения данных

Размер матриц	2 процессора		4 процессора		8 процессоров	
	T_p	T_p^*	T_p	T_p^*	T_p	T_p^*
500	0,8243	0,3758	0,4313	0,1535	0,2353	0,0968
1000	6,51822	5,4427	3,3349	2,2628	1,7436	0,6998
1500	21,9137	20,9503	11,1270	11,0804	5,7340	5,1766
2000	51,8429	45,7436	26,2236	21,6001	13,4144	9,4127
2500	101,1377	99,5097	51,0408	56,9203	25,9928	18,3303
3000	174,6301	171,9232	87,9946	111,9642	44,6772	45,5482

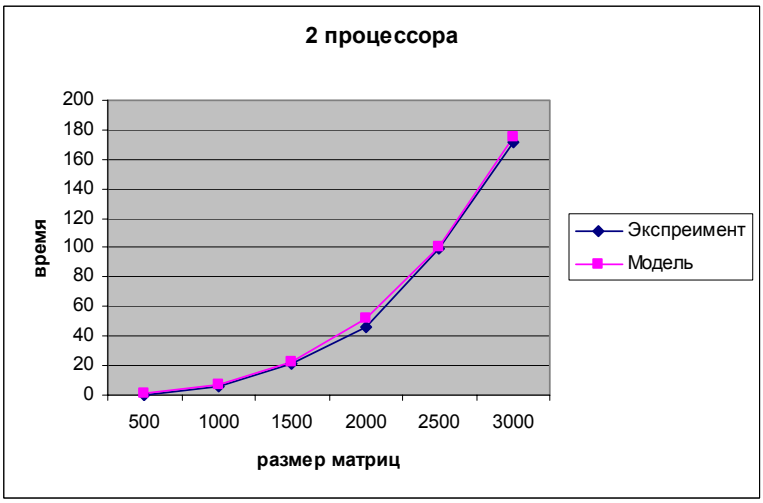

Рис. 8.5. График зависимости от объема исходных данных теоретического и экспериментального времени выполнения параллельного алгоритма на двух процессорах (ленточная схема разбиения данных)

8.4. Алгоритм Фокса умножения матриц при блочном разделении данных

При построении параллельных способов выполнения матричного умножения наряду с рассмотрением матриц в виде наборов строк и столбцов широко используется блочное представление матриц. Выполним более подробное рассмотрение данного способа организации вычислений.

8.4.1. Определение подзадач

Блочная схема разбиения матриц подробно рассмотрена в подразделе 7.2. При таком способе разделения данных исходные матрицы A, B и результирующая матрица C представляются в виде наборов блоков. Для более простого изложения следующего материала будем предполагать далее, что все матрицы являются квадратными размера $n \times n$, количество блоков по горизонтали и вертикали являются одинаковым и равным q (т.е. размер всех блоков равен $k \times k$, $k=n/q$). При таком представлении данных операция матричного умножения матриц A и B в блочном виде может быть представлена в виде:

$$\begin{pmatrix} A_{00} A_{01} \ldots A_{0q-1} \\ \cdots \\ A_{q-10} A_{q-11} \ldots A_{q-1q-1} \end{pmatrix} \times \begin{pmatrix} B_{00} B_{01} \ldots B_{0q-1} \\ \cdots \\ B_{q-10} B_{q-11} \ldots B_{q-1q-1} \end{pmatrix} = \begin{pmatrix} C_{00} C_{01} \ldots C_{0q-1} \\ \cdots \\ c_{q-10} C_{q-11} \ldots C_{q-1q-1} \end{pmatrix},$$

где каждый блок C_{ij} матрицы C определяется в соответствии с выражением

$$C_{ij} = \sum_{s=0}^{q-1} A_{is} B_{sj} \ .$$

При блочном разбиении данных для определения базовых подзадач естественным представляется взять за основу вычисления, выполняемые над матричными блоками. С учетом сказанного определим базовую подзадачу как процедуру вычисления всех элементов одного из блоков матрицы C.

Для выполнения всех необходимых вычислений базовым подзадачам должны быть доступны соответствующие наборы строк матрицы A и столбцов матрицы B. Размещение всех требуемых данных в каждой подзадаче неизбежно приведет к дублированию и к значительному росту объема используемой памяти. Как результат, вычисления должны быть организованы таким образом, чтобы в каждый текущий момент времени подзадачи содержали лишь часть необходимых для проведения расчетов данных, а доступ к остальной части данных обеспечивался бы при помощи передачи сообщений. Один из возможных подходов – алгоритм Фокса (Fox) – рассмотрен далее в данном подразделе. Второй способ – *алгоритм Кэннона (Cannon)* – приводится в подразделе 8.5.

8.4.2. Выделение информационных зависимостей

Итак, за основу параллельных вычислений для матричного умножения при блочном разделении данных принят подход, при котором базовые подзадачи отвечают за вычисления отдельных блоков матрицы C и при этом в подзадачах на каждой итерации расчетов располагаются только по одному блоку исходных матриц A и B. Для нумерации подзадач будем использовать индексы размещаемых в подзадачах блоков матрицы C, т.е. подзадача *(i,j)* отвечает за вычисление блока C_{ij} – тем самым, набор подзадач образует квадратную решетку, соответствующую структуре блочного представления матрицы C.

Возможный способ организации вычислений при таких условиях состоит в применении широко известного *алгоритма Фокса (Fox)*, - см., например, Fox et al. (1987), Kumar et al. (1994).

В соответствии с алгоритмом Фокса в ходе вычислений на каждой базовой подзадаче *(i,j)* располагается четыре матричных блока:

- блок C_{ij} матрицы C, вычисляемый подзадачей;
- блок A_{ij} матрицы A, размещаемый в подзадаче перед началом вычислений;
- блоки A'_{ij}, B'_{ij} матриц A и B, получаемые подзадачей в ходе выполнения вычислений.

Выполнение параллельного метода включает:

- этап инициализации, на котором каждой подзадаче *(i,j)* передаются блоки A_{ij}, B_{ij} и обнуляются блоки C_{ij} на всех подзадачах;
- этап вычислений, в рамках которого на каждой итерации l, $0 \le l < q$, осуществляются следующие операции:
  - для каждой строки i, $0 \le i < q$, блок A_{ij} подзадачи *(i,j)* пересылается на все подзадачи той же строки i решетки; индекс j, определяющий положение подзадачи в строке, вычисляется в соответствии с выражением

$$j = (i+l) \bmod q,$$

где *mod* есть операция получения остатка от целочисленного деления;

– полученные в результаты пересылок блоки A'_{ij}, B'_{ij} каждой подзадачи *(i,j)* перемножаются и прибавляются к блоку C_{ij}

$$C_{ij} = C_{ij} + A'_{ij} \times B'_{ij};$$

– блоки B'_{ij} каждой подзадачи *(i,j)* пересылаются подзадачам, являющимися соседями сверху в столбцах решетки подзадач (блоки подзадач из первой строки решетки пересылаются подзадачам последней строки решетки).

Для пояснения приведенных правил параллельного метода на рис. 8.6 приведено состояние блоков в каждой подзадаче в ходе выполнения итераций этапа вычислений (для решетки подзадач *2×2*).

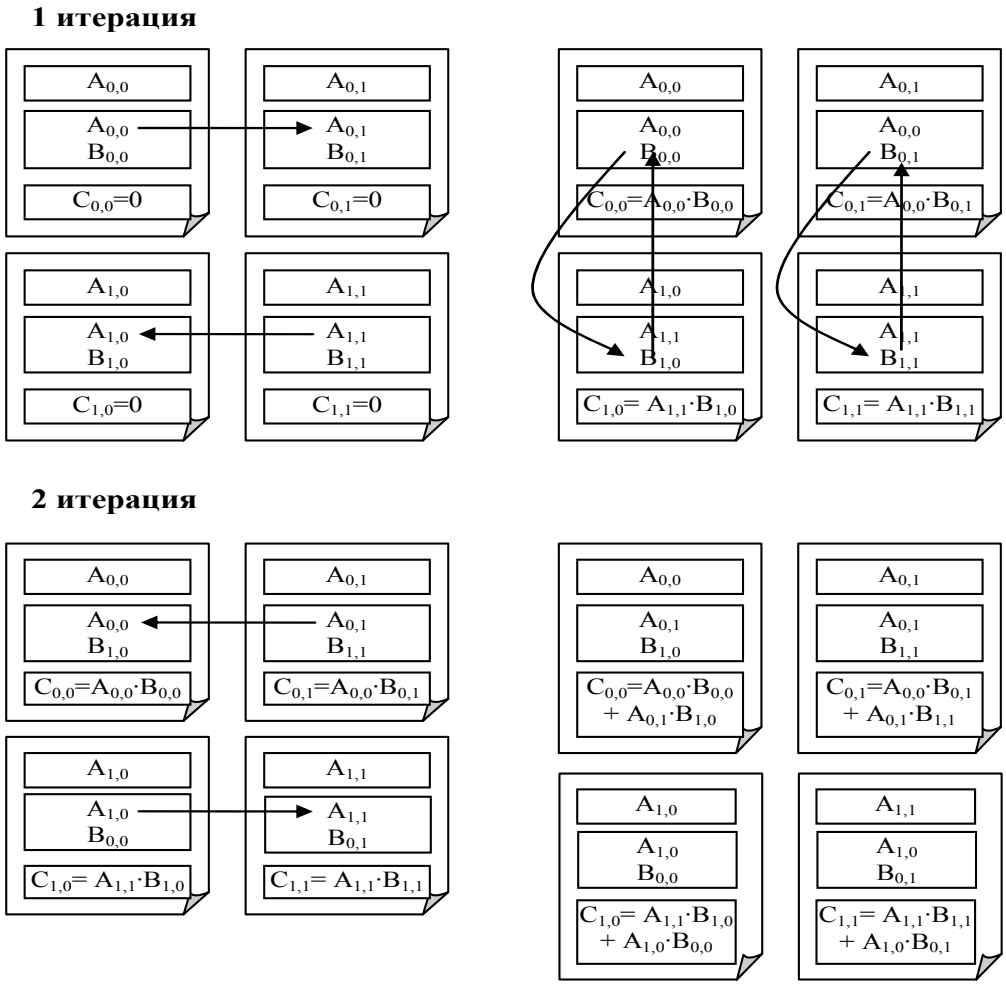

Рис. 8.6. Состояние блоков в каждой подзадаче в ходе выполнения итераций алгоритма Фокса

8.4.3. Масштабирование и распределение подзадач по процессорам

В рассмотренной схеме параллельных вычислений количество блоков может варьироваться в зависимости от выбора размера блоков – эти размеры могут быть подобраны таким образом, чтобы общее количество базовых подзадач совпадало с числом процессоров *p*. Так, например, в наиболее простом случае, когда число процессоров представимо в виде $p=\delta^2$ (т.е. является полным квадратом) можно выбрать количество блоков в матрицах по вертикали и горизонтали равным δ (т.е. $q=\delta$). Такой способ определения количества блоков приводит к тому, что объем вычислений в каждой подзадаче является одинаковым и, тем самым, достигается полная балансировка вычислительной нагрузки между процессорами. В более общем случае при произвольных количестве процессоров и размеров матриц балансировка вычислений может отличаться от абсолютно одинаковой, но, тем не менее, при надлежащем выборе параметров может быть распределена между процессорами равномерно в рамках требуемой точности.

Для эффективного выполнения алгоритма Фокса, в котором базовые подзадачи представлены в виде квадратной решетки и в ходе вычислений выполняются операции передачи блоков по строкам и столбцам решетки подзадач, наиболее адекватным решением является организация множества имеющихся

процессоров также в виде квадратной решетки. В этом случае можно осуществить непосредственное отображение набора подзадач на множество процессоров – базовую подзадачу (i,j) следует располагать на процессоре $P_{i,j}$. Необходимая структура сети передачи данных может быть обеспечена на физическом уровне, если топология вычислительной системы имеет вид решетки или полного графа.

8.4.4. Анализ эффективности

Определим вычислительную сложность данного алгоритма Фокса. Построение оценок будет происходить при условии выполнения всех ранее выдвинутых предположений - все матрицы являются квадратными размера $n \times n$, количество блоков по горизонтали и вертикали являются одинаковым и равным q (т.е. размер всех блоков равен $k \times k$, $k=n/q$), процессоры образуют квадратную решетку и их количество равно $p=q^2$.

Как уже отмечалось, алгоритм Фокса требует для своего выполнения q итераций, в ходе которых каждый процессор перемножает свои текущие блоки матриц A и B и прибавляет результаты умножения к текущему значению блока матрицы C. С учетом выдвинутых предположений общее количество выполняемых при этом операций будет иметь порядок n^3/p. Как результат, показатели ускорения и эффективности алгоритма имеют вид:

$$S_p = \frac{n^3}{(n^3/p)} = p \quad \text{и} \quad E_p = \frac{n^3}{p \cdot (n^3/p)} = 1. \tag{8.9}$$

Общий анализ сложности снова дает идеальные показатели эффективности параллельных вычислений. Уточним полученные соотношения - укажем для этого более точно количество вычислительных операций алгоритма и учтем затраты на выполнение операций передачи данных между процессорами.

Определим количество вычислительных операций. Сложность выполнения скалярного умножения строки блока матрицы A на столбец блока матрицы B можно оценить как $2(n/q)-1$. Количество строк и столбцов в блоках равно n/q и, как результат, трудоемкость операции блочного умножения оказывается равной $(n^2/p)(2n/q-1)$. Для сложения блоков требуется n^2/p операций. С учетом всех перечисленных выражений время выполнения вычислительных операций алгоритма Фокса может быть оценено следующим образом:

$$T_p(calc) = q[(n^2/p)\cdot(2n/q-1)+(n^2/p)]\cdot\tau. \tag{8.10}$$

(напомним, что τ есть время выполнения одной элементарной скалярной операции).

Оценим затраты на выполнение операций передачи данных между процессорами. На каждой итерации алгоритма перед умножением блоков один из процессоров строки процессорной решетки рассылает свой блок матрицы A остальным процессорам своей строки. Как уже отмечалось ранее, при топологии сети в виде гиперкуба или полного графа выполнение этой операции может быть обеспечено за $log_2 q$ шагов, а объем передаваемых блоков равен n^2/p. Как результат, время выполнения операции передачи блоков матрицы A при использовании модели Хокни может оцениваться как

$$T_{\text{п}}^1(comm) = \log_2 q\,(\alpha + w(n^2/p)/\beta) \tag{8.11}$$

где α – латентность, β – пропускная способность сети передачи данных, а w есть размер элемента матрицы в байтах. В случае же, когда топология строк процессорной решетки представляет собой кольцо, выражение для оценки времени передачи блоков матрицы A принимает вид:

$$\widetilde{T}_{\text{п}}^1(comm) = (q/2)(\alpha + w(n^2/p)/\beta).$$

Далее после умножения матричных блоков процессоры передают свои блоки матрицы B предыдущим процессорам по столбцам процессорной решетки (первые процессоры столбцов передают свои данные последним процессорам в столбцах решетки). Эти операции могут быть выполнены процессорами параллельно и, тем самым, длительность такой коммуникационной операции составляет:

$$T_p^2(comm) = \alpha + w\cdot(n^2/p)/\beta. \tag{8.12}$$

Просуммировав все полученные выражения, можно получить, что общее время выполнения алгоритма Фокса может быть определено при помощи следующих соотношений:

$$\begin{aligned} T_p &= q[(n^2/p)\cdot(2n/q-1)+(n^2/p)]\cdot\tau + q\log_2 q\,(\alpha + w(n^2/p)/\beta) + (q-1)\cdot(\alpha + w(n^2/p)/\beta) = \\ &= q[(n^2/p)\cdot(2n/q-1)+(n^2/p)]\cdot\tau + (q\log_2 q + (q-1))(\alpha + w(n^2/p)/\beta) \end{aligned} \tag{8.13}$$

(напомним, что параметр q определяет размер процессорной решетки и $q = \sqrt{p}$).

8.4.5. Программная реализация

Представим возможный вариант программной реализации алгоритма Фокса для умножения матриц при блочном представлении данных. Приводимый программный код содержит основные модули параллельной программы, отсутствие отдельных вспомогательных функций не сказывается на общем понимании реализуемой схемы параллельных вычислений.

1. Главная функция программы. Определяет основную логику работы алгоритма, последовательно вызывает необходимые подпрограммы.

```
// Программа 8.1
// Алгоритм Фокса умножения матриц – блочное представление данных
//   Условия выполнения программы: все матрицы квадратные, размер блоков и их
//   количество по горизонтали и вертикали одинаковово, процессоры образуют
//   квадратную решетку
int ProcNum = 0;       // Number of available processes
int ProcRank = 0;      // Rank of current process
int GridSize;          // Size of virtual processor grid
int GridCoords[2];     // Coordinates of current processor in grid
MPI_Comm GridComm;     // Grid communicator
MPI_Comm ColComm;      // Column communicator
MPI_Comm RowComm;      // Row communicator

void main ( int argc, char * argv[] ) {
  double* pAMatrix;  // The first argument of matrix multiplication
  double* pBMatrix;  // The second argument of matrix multiplication
  double* pCMatrix;  // The result matrix
  int Size;          // Size of matricies
  int BlockSize;     // Sizes of matrix blocks on current process
  double *pAblock;   // Initial block of matrix A on current process
  double *pBblock;   // Initial block of matrix B on current process
  double *pCblock;   // Block of result matrix C on current process
  double *pMatrixAblock;
  double Start, Finish, Duration;

  setvbuf(stdout, 0, _IONBF, 0);

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
  MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

  GridSize = sqrt((double)ProcNum);
  if (ProcNum != GridSize*GridSize) {
    if (ProcRank == 0) {
      printf ("Number of processes must be a perfect square \n");
    }
  }
  else {
    if (ProcRank == 0)
      printf("Parallel matrix multiplication program\n");

    // Creating the cartesian grid, row and column communcators
    CreateGridCommunicators();

    // Memory allocation and initialization of matrix elements
    ProcessInitialization ( pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
      pCblock, pMatrixAblock, Size, BlockSize );


    DataDistribution(pAMatrix, pBMatrix, pMatrixAblock, pBblock, Size,
      BlockSize);

    // Execution of Fox method
    ParallelResultCalculation(pAblock, pMatrixAblock, pBblock,
```

```
      pCblock, BlockSize);

    ResultCollection(pCMatrix, pCblock, Size, BlockSize);

    TestResult(pAMatrix, pBMatrix, pCMatrix, Size);

    // Process Termination
    ProcessTermination (pAMatrix, pBMatrix, pCMatrix, pAblock, pBblock,
      pCblock, pMatrixAblock);
  }

  MPI_Finalize();}
```

2. Функция CreateGridCommunicators. Данная функция создает коммуникатор в виде двумерной квадратной решетки, определяет координаты каждого процесса в этой решетке, а также создает коммуникаторы отдельно для каждой строки и каждого столбца.

Создание решетки производится при помощи функции *MPI_Cart_create* (вектор *Periodic* определяет возможность передачи сообщений между граничными процессами строк и столбцов создаваемой решетки). После создания решетки каждый процесс параллельной программы будет иметь координаты своего положения в решетке; получение этих координат обеспечивается при помощи функции *MPI_Cart_coords*.

Формирование топологий завершается созданием множества коммуникаторов для каждой строки и каждого столбца решетки в отдельности (функция *MPI_Cart_sub*).

```
// Creation of two-dimensional grid communicator
// and communicators for each row and each column of the grid
void CreateGridCommunicators() {
  int DimSize[2];  // Number of processes in each dimension of the grid
  int Periodic[2]; // =1, if the grid dimension should be periodic
  int Subdims[2];  // =1, if the grid dimension should be fixed

  DimSize[0] = GridSize;
  DimSize[1] = GridSize;
  Periodic[0] = 0;
  Periodic[1] = 0;

  // Creation of the Cartesian communicator
  MPI_Cart_create(MPI_COMM_WORLD, 2, DimSize, Periodic, 1, &GridComm);

  // Determination of the cartesian coordinates for every process
  MPI_Cart_coords(GridComm, ProcRank, 2, GridCoords);

  // Creating communicators for rows
  Subdims[0] = 0;  // Dimensionality fixing
  Subdims[1] = 1;  // The presence of the given dimension in the subgrid
  MPI_Cart_sub(GridComm, Subdims, &RowComm);

  // Creating communicators for columns
  Subdims[0] = 1;
  Subdims[1] = 0;
  MPI_Cart_sub(GridComm, Subdims, &ColComm);
}
```

3. Функция ProcessInitialization. Для определения размеров матриц и матричных блоков, выделения памяти для их хранения и определения элементов исходных матриц реализуем функцию *ProcessInitialization*.

Данная функция определяет параметры решаемой задачи (размеры матриц и их блоков), выделяет память для хранения данных и осуществляет ввод исходных матриц (или формирует их при помощи какого-либо датчика случайных чисел). Всего в каждом процессе должна быть выделена память для хранения четырех блоков – для указателей на выделенную память используются переменные *pAblock, pBblock, pCblock, pMatrixAblock*. Первые три указателя определяют блоки матриц *A, B* и *C* соответственно. Следует отметить, что содержимое блоков *pAblock* и *pBblock* постоянно меняется в соответствии с пересылкой данных между процессами, в то время как блок *pMatrixAblock* матрицы *A* остается неизменным и используется при рассылках блоков по строкам решетки процессов (см. функцию *AblockCommunication*).

Для определения элементов исходных матриц будем использовать функцию *RandomDataInitialization*, реализацию которой читателю предстоит выполнить самостоятельно.

```
// Function for memory allocation and data initialization
void ProcessInitialization (double* &pAMatrix, double* &pBMatrix,
  double* &pCMatrix, double* &pAblock, double* &pBblock, double* &pCblock,
  double* &pTemporaryAblock, int &Size, int &BlockSize ) {
  if (ProcRank == 0) {
    do {
      printf("\nEnter size of the initial objects: ");
      scanf("%d", &Size);

      if (Size%GridSize != 0) {
        printf ("Size of matricies must be divisible by the grid size! \n");
      }
    }
    while (Size%GridSize != 0);
  }
  MPI_Bcast(&Size, 1, MPI_INT, 0, MPI_COMM_WORLD);

  BlockSize = Size/GridSize;

  pAblock = new double [BlockSize*BlockSize];
  pBblock = new double [BlockSize*BlockSize];
  pCblock = new double [BlockSize*BlockSize];
  pTemporaryAblock = new double [BlockSize*BlockSize];

  for (int i=0; i<BlockSize*BlockSize; i++) {
    pCblock[i] = 0;
  }
  if (ProcRank == 0) {
    pAMatrix = new double [Size*Size];
    pBMatrix = new double [Size*Size];
    pCMatrix = new double [Size*Size];
    RandomDataInitialization(pAMatrix, pBMatrix, Size);
  }
}
```

4. Функция DataDistribution для распределения исходных данных и функция ResultCollection для сбора результатов. После задания исходных матриц на нулевом процессе необходимо осуществить распределение исходных данных. Для этого предназначена функция *DataDistribution*. Может быть предложено два способа выполнения блочного разделения матриц между процессорами, организованными в двумерную квадратную решетку. С одной стороны, для организации передачи блоков в рамках одной и той же коммуникационной операции можно сформировать средствами MPI производный тип данных. С другой стороны, можно организовать двухэтапную процедуру. На первом этапе матрица разделяется на горизонтальные полосы. Эти полосы распределяются на процессы, составляющие нулевой столбец процессорной решетки. Далее каждая полоса разделяется на блоки между процессами, составляющими строки процессорной решетки.

Для выполнения сбора результирующей матрицы из блоков предназначена функция *ResultCollection*. Сбор данных также выполнить двумя способами: либо с использованием производного типа данных, либо при помощи двухэтапной процедуры, зеркально отображающей процедуру распределения матрицы.

Реализация функций *DataDistribution* и *ResultCollection* представляет собой задание для самостоятельной работы.

5. Функция *AblockCommunication* обмена блоками матрицы A. Функция выполняет рассылку блоков матрицы A по строкам процессорной решетки. Для этого в каждой строке решетки определяется ведущий процесс *Pivot*, осуществляющий рассылку. Для рассылки используется блок *pMatrixAblock*, переданный в процесс в момент начального распределения данных. Выполнение операции рассылки блоков осуществляется при помощи функции *MPI_Bcast*. Следует отметить, что данная операция является коллективной и ее локализация пределами отдельных строк решетки обеспечивается за счет использования коммуникаторов *RowComm*, определенных для набора процессов каждой строки решетки в отдельности.

```
// Broadcasting matrix A blocks to process grid rows
void ABlockCommunication (int iter, double *pAblock, double* pMatrixAblock,
```

```
  int BlockSize) {

  // Defining the leading process of the process grid row
  int Pivot = (GridCoords[0] + iter) % GridSize;

  // Copying the transmitted block in a separate memory buffer
  if (GridCoords[1] == Pivot) {
    for (int i=0; i<BlockSize*BlockSize; i++)
      pAblock[i] = pMatrixAblock[i];
  }

  // Block broadcasting
  MPI_Bcast(pAblock, BlockSize*BlockSize, MPI_DOUBLE, Pivot, RowComm);
}
```

6. Функция перемножения матричных блоков *BlockMultiplication*. Функция обеспечивает перемножение блоков матриц *A* и *B*. Следует отметить, что для более легкого понимания рассматриваемой программы приводится простой вариант реализации функции – выполнение операции блочного умножения может быть существенным образом оптимизировано для сокращения времени вычислений. Данная оптимизация может быть направлена, например, на повышение эффективности использования кэша процессоров, векторизации выполняемых операций и т.п.

```
// Умножение матричных блоков
void BlockMultiplication (double *pAblock, double *pBblock,
  double *pCblock, int BlockSize) {
  // вычисление произведения матричных блоков
  for (int i=0; i<BlockSize; i++) {
    for (int j=0; j<BlockSize; j++) {
      double temp = 0;
      for (int k=0; k<BlockSize; k++ )
        temp += pAblock [i*BlockSize + k] * pBblock [k*BlockSize + j]
      pCblock [i*BlockSize + j] += temp;
    }
  }
}
```

7. Функция *BblockCommunication* обмена блоками матрицы B. Функция выполняет циклический сдвиг блоков матрицы B по столбцам процессорной решетки. Каждый процесс передает свой блок следующему процессу *NextProc* в столбце процессов и получает блок, переданный из предыдущего процесса *PrevProc* в столбце решетки. Выполнение операций передачи данных осуществляется при помощи функции *MPI_SendRecv_replace*, которая обеспечивает все необходимые пересылки блоков, используя при этом один и тот же буфер памяти *pBblock*. Кроме того, эта функция гарантирует отсутствие возможных тупиков, когда операции передачи данных начинают одновременно выполняться несколькими процессами при кольцевой топологии сети.

```
// Cyclic shift of matrix B blocks in the process grid columns
void BblockCommunication (double *pBblock, int BlockSize) {
  MPI_Status Status;
  int NextProc = GridCoords[0] + 1;
  if ( GridCoords[0] == GridSize-1 ) NextProc = 0;
  int PrevProc = GridCoords[0] - 1;
  if ( GridCoords[0] == 0 ) PrevProc = GridSize-1;

  MPI_Sendrecv_replace( pBblock, BlockSize*BlockSize, MPI_DOUBLE,
    NextProc, 0, PrevProc, 0, ColComm, &Status);
}
```

8. Функция ParallelResultCalculation. Для непосредственного выполнения параллельного алгоритма Фокса умножения матриц предназначена функция *ParallelResultCalculation*, которая реализует логику работы алгоритма.

```
void ParallelResultCalculation(double* pAblock, double* pMatrixAblock,
  double* pBblock, double* pCblock, int BlockSize) {
  for (int iter = 0; iter < GridSize; iter ++) {
    // Sending blocks of matrix A to the process grid rows
    ABlockCommunication (iter, pAblock, pMatrixAblock, BlockSize);
```

```
    // Block multiplication
    BlockMultiplication(pAblock, pBblock, pCblock, BlockSize);
    // Cyclic shift of blocks of matrix B in process grid columns
    BblockCommunication(pBblock, BlockSize);
  }
}
```

8.4.6. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма проводились при тех же условиях, что и ранее выполненные (см. п. 8.3.5). Результаты экспериментов с использованием четырех и девяти процессоров приведены в таблице 8.3.

Таблица 8.3 Результаты вычислительных экспериментов по исследованию параллельного алгоритма Фокса

Размер матриц	Последовательный алгоритм	Параллельный алгоритм			
		4 процессора		9 процессоров	
		Время	Ускорение	Время	Ускорение
500	0,8527	0,2190	3,8925	0,1468	5,8079
1000	12,8787	3,0910	4,1664	2,1565	5,9719
1500	43,4731	10,8678	4,0001	7,2502	5,9960
2000	103,0561	24,1421	4,2687	21,4157	4,8121
2500	201,2915	51,4735	3,9105	41,2159	4,8838
3000	347,8434	87,0538	3,9957	58,2022	5,9764

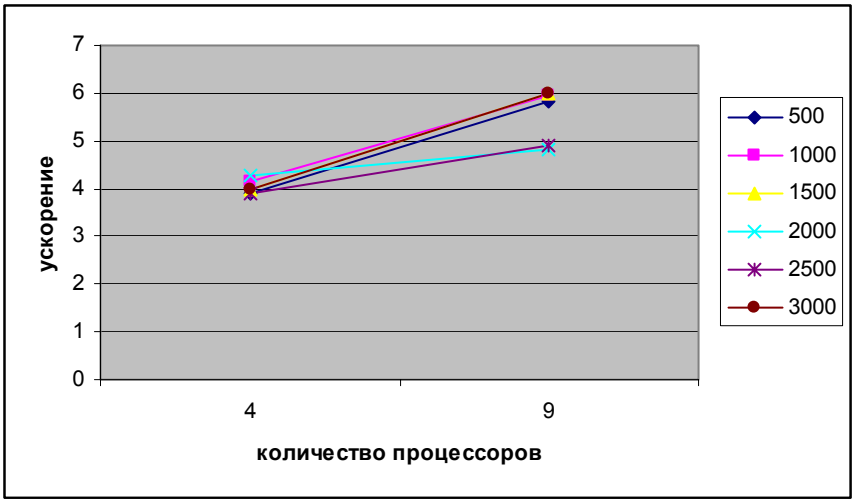

Рис. 8.7. Зависимость ускорения от размера матриц при выполнении параллельного алгоритма Фокса

Сравнение времени выполнения эксперимента T_p^* и теоретического времени T_p, вычисленного в соответствии с выражением (8.13), представлено в таблице 8.4 и на рис. 8.8.

Таблица 8.4. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма Фокса

Размер матриц	4 процессора		9 процессоров	
	T_p	T_p^*	T_p	T_p^*
500	0,4217	0,2190	0,2200	0,1468
1000	3,2970	3,0910	1,5924	2,1565
1500	11,0419	10,8678	5,1920	7,2502
2000	26,0726	24,1421	12,0927	21,4157
2500	50,8049	51,4735	23,3682	41,2159

3000	87,6548	87,0538	40,0923	58,2022

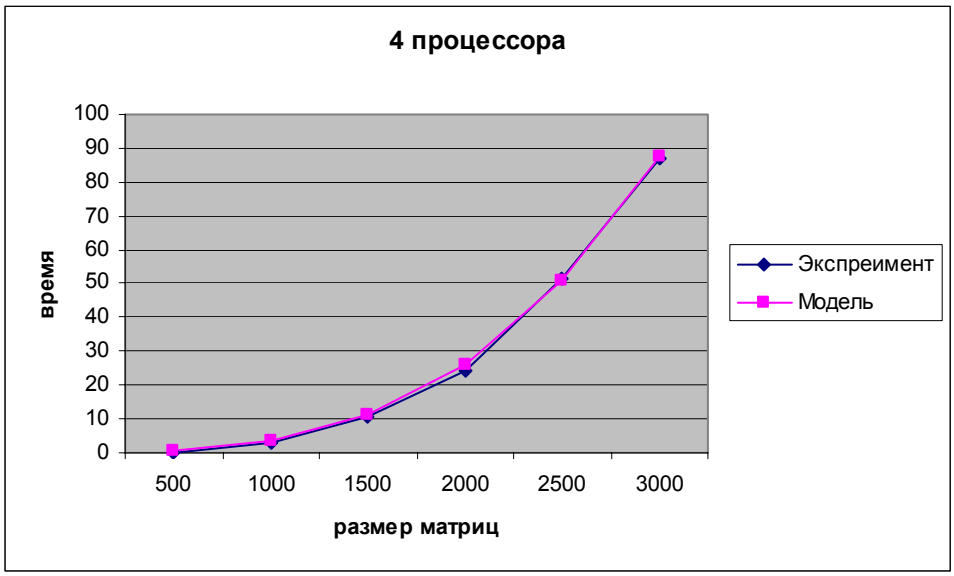

Рис. 8.8. График зависимости экспериментального и теоретического времени выполнения алгоритма Фокса на четырех процессорах

8.5. Алгоритм Кэннона умножения матриц при блочном разделении данных

Рассмотрим еще один параллельный алгоритм матричного умножения, основанный на блочном разбиении матриц.

8.5.1. Определение подзадач

Как и при рассмотрении алгоритма Фокса, в качестве базовой подзадачи выберем вычисления, связанные с определением одного из блоков результирующей матрицы C . Как уже отмечалось ранее, для вычисления элементов этого блока подзадача должна иметь доступ к элементам горизонтальной полосы матрицы A и элементам вертикальной полосы матрицы B.

8.5.2. Выделение информационных зависимостей

Отличие алгоритма Кэннона от представленного в предыдущем подразделе метода Фокса состоит в изменении схемы начального распределения блоков перемножаемых матриц между подзадачами вычислительной системы. Начальное расположение блоков в алгоритме Кэннона подбирается таким образом, чтобы располагаемые блоки в подзадачах могли бы быть перемножены без каких-либо дополнительных передач данных. При этом подобное распределение блоков может быть организовано таким образом, что перемещение блоков между подзадачами в ходе вычислений может осуществляться с использованием более простых коммуникационных операций.

С учетом высказанных замечаний этап инициализации алгоритма Кэннона включает выполнение следующих операций передач данных:

- в каждую подзадачу *(i,j)* передаются блоки A_{ij}, B_{ij};
- для каждой строки i решетки подзадач блоки матрицы A сдвигаются на *(i-1)* позиций влево;
- для каждого столбца j решетки подзадач блоки матрицы B сдвигаются на *(j-1)* позиций вверх.

Выполняемые при перераспределении матричных блоков процедуры передачи данных являются примером операции *циклического сдвига* – см. раздел 3. Для пояснения используемого способа начального распределения данных на рис. 8.9 показан пример расположения блоков для решетки подзадач 3×3.

перераспределение блоков матриц A и B

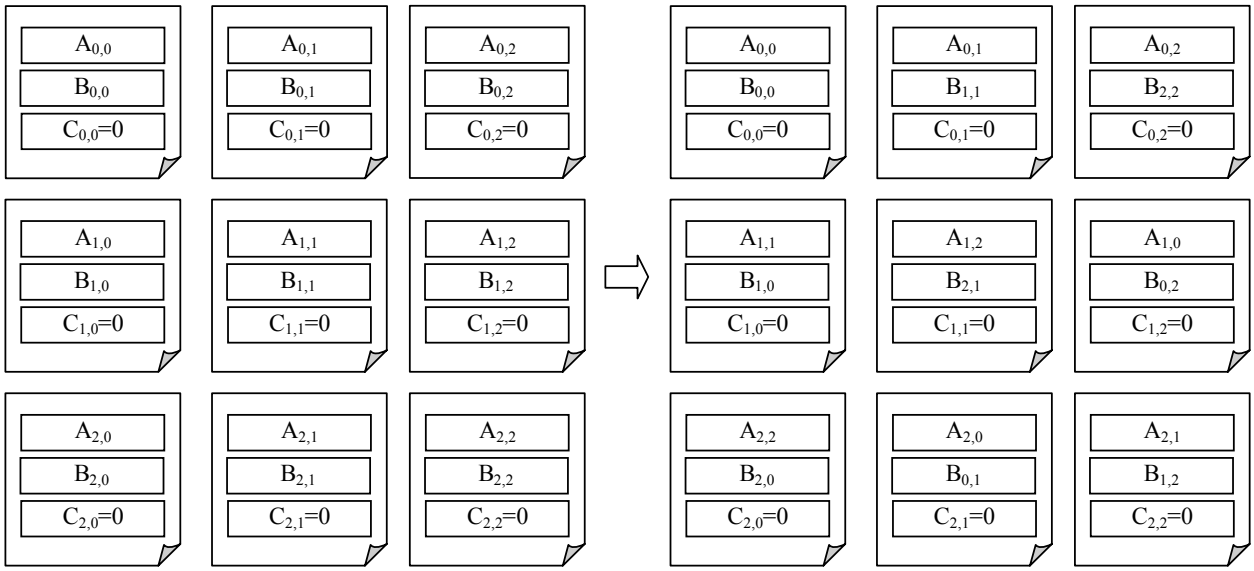

Рис. 8.9. Перераспределение блоков исходных матриц между процессорами при выполнении алгоритма Кэннона

В результате такого начального распределения в каждой базовой подзадаче будут располагаться блоки, которые могут быть перемножены без дополнительных операций передачи данных. Кроме того, получение всех последующих блоков для подзадач может быть обеспечено при помощи простых коммуникационных действий - после выполнения операции блочного умножения каждый блок матрицы A должен быть передан предшествующей подзадаче влево по строкам решетки подзадач, а каждый блок матрицы B – предшествующей подзадаче вверх по столбцам решетки. Как можно показать, последовательность таких циклических сдвигов и умножение получаемых блоков исходных матриц A и B приведет к получению в базовых подзадачах соответствующих блоков результирующей матрицы C.

8.5.3. Масштабирование и распределение подзадач по процессорам

Как и ранее в методе Фокса, для алгоритма Кэннона размер блоков может быть подобран таким образом, чтобы количество базовых подзадач совпадало с числом имеющихся процессоров. Поскольку объем вычислений в каждой подзадаче является равным, это обеспечивает полную балансировку вычислительной нагрузки между процессорами.

Для распределения подзадач между процессорами может быть применен подход, использованный в алгоритме Фокса – множество имеющихся процессоров представляется в виде квадратной решетки и размещение базовых подзадач (i,j) осуществляется на процессорах $P_{i,j}$ соответствующих узлов процессорной решетки. Необходимая структура сети передачи данных, как и ранее, может быть обеспечена на физическом уровне при топологии вычислительной системы в виде решетки или полного графа.

8.5.4. Анализ эффективности

Перед проведением анализа эффективности следует отметить, что алгоритм Кэннона отличается от метода Фокса только видом выполняемых в ходе вычислений коммуникационных операций. Как результат, используя оценки времени выполнения вычислительных операций, приведенные в п. 8.4.4, проведем только анализ коммуникационной сложности алгоритма Кэннона.

В соответствии с правилами алгоритма на этапе инициализации производится перераспределение блоков матриц A и B при помощи циклического сдвига матричных блоков по строкам и столбцам процессорной решетки. Трудоемкость выполнения такой операции передачи данных существенным образом зависит от топологии сети. Для сети со структурой полного графа все необходимые пересылки блоков могут быть выполнены одновременно (т.е. длительность операции оказывается равной времени передачи одного матричного блока между соседними процессорами). Для сети с топологией гиперкуба операция циклического сдвига может потребовать выполнения $log_2 q$ итераций. Для сети с кольцевой структурой связей необходимое количество итераций оказывается равным $q-1$ – более подробно методы выполнения операции циклического сдвига рассмотрены в разделе 3. Используем для построения оценки коммуникационной сложности этапа инициализации вариант топологии полного графа как более соответствующего кластерным вычислительным системам, время выполнения начального перераспределения блоков может оцениваться как

$$T_p^1(comm) = 2 \cdot \left(\alpha + w \cdot (n^2/p)/\beta\right) \qquad (8.14)$$

(выражение n^2/p определяет размер пересылаемых блоков, а коэффициент 2 соответствует двум выполняемым операциям циклического сдвига).

Оценим теперь затраты на передачу данных между процессорами при выполнении основной части алгоритма Кэннона. На каждой итерации алгоритма после умножения матричных блоков процессоры передают свои блоки предыдущим процессорам по строкам (для блоков матрицы *A*) и столбцам (для блоков матрицы *B*) процессорной решетки. Эти операции также могут быть выполнены процессорами параллельно и, тем самым, длительность таких коммуникационных действий составляет:

$$T_p^2(comm) = 2 \cdot \left(\alpha + w \cdot (n^2/p)/\beta\right). \qquad (8.15)$$

Поскольку количество итераций алгоритма Кэннона является равным q, то с учетом оценки (8.13) общее время выполнения параллельных вычислений может быть определено при помощи следующего соотношения:

$$T_p = q[(n^2/p) \cdot (2n/q - 1) + (n^2/p)] \cdot \tau + (2q+2)(\alpha + w(n^2/p)/\beta) \qquad (8.16)$$

(в используемых выражениях параметр $q = \sqrt{p}$ определяет размер процессорной решетки).

8.5.5. Результаты вычислительных экспериментов

Вычислительные эксперименты для оценки эффективности параллельного алгоритма проводились при тех же условиях, что и ранее выполненные эксперименты (см. п. 8.3.5). Результаты экспериментов для случаев четырех и девяти процессоров приведены в таблице 8.5.

Таблица 8.5 Результаты вычислительных экспериментов по исследованию параллельного алгоритма Кэннона

Размер объектов	Последовательный алгоритм	Параллельный алгоритм			
		4 процессора		9 процессоров	
		Время	Ускорение	Время	Ускорение
1000	12,8787	3,0806	4,1805	1,1889	10,8324
1500	43,4731	11,1716	3,8913	4,6310	9,3872
2000	103,0561	24,0502	4,2850	14,4759	7,1191
2500	201,2915	53,1444	3,7876	23,5398	8,5511
3000	347,8434	88,2979	3,9394	36,3688	9,5643

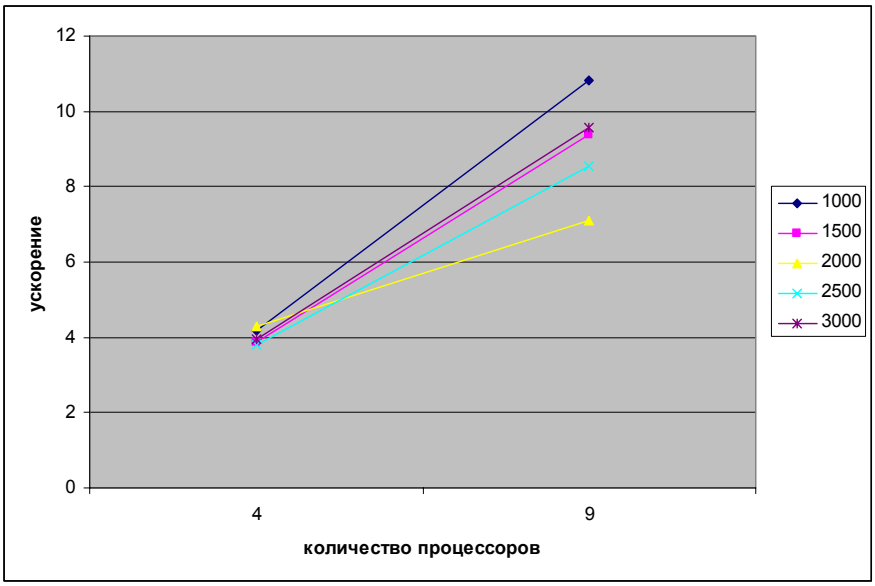

Рис. 8.10. Зависимость ускорения от размера матриц при выполнении параллельного алгоритма Кэннона

Сравнение времени выполнения эксперимента T_p^* и теоретического времени T_p, вычисленного в соответствии с выражением (8.16), представлено в таблице 8.6 и на рис. 8.13.

Таблица 8.6. Сравнение экспериментального и теоретического времени выполнения параллельного алгоритма Кэннона

Размер матриц	4 процессора		9 процессоров	
	T_p	T_p^*	T_p	T_p^*
1000	3,4485	3,0806	1,5669	1,1889
1500	11,3821	11,1716	5,1348	4,6310
2000	26,6769	24,0502	11,9912	14,4759
2500	51,7488	53,1444	23,2098	23,5398
3000	89,0138	88,2979	39,8643	36,3688

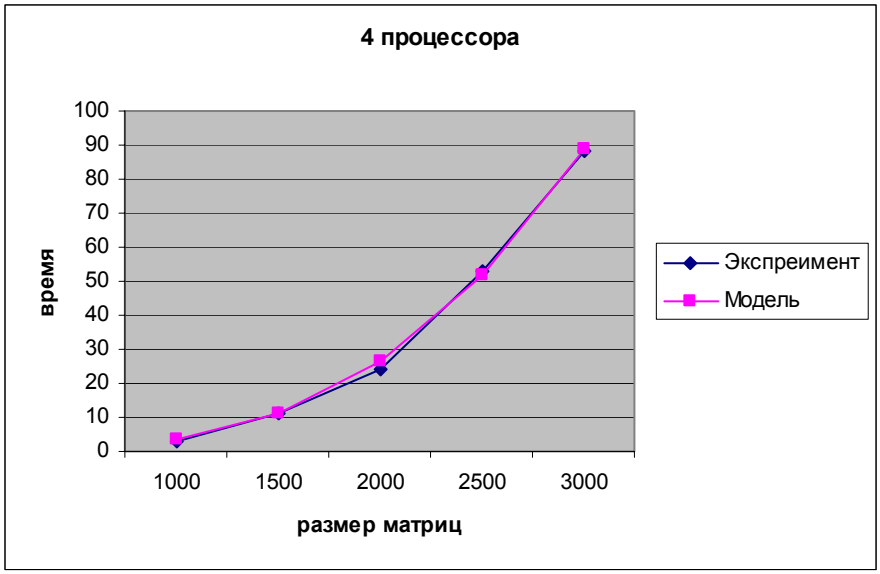

Рис. 8.11. График зависимости экспериментального и теоретического времени выполнения алгоритма Кэннона на четырех процессорах

8.6. Краткий обзор раздела

В данном разделе рассмотрены три параллельных метода для выполнения операции матричного умножения. Первый алгоритм основан на ленточном разделении матриц между процессорами. В разделе приведены два различных варианта этого алгоритма. Первый вариант алгоритма основан на различном разделении перемножаемых матриц – первая матрица (матрица ***A***) разбивается на горизонтальные полосы, а вторая матрица (матрица ***B***) делится на вертикальные полосы. Второй вариант ленточного алгоритма использует разбиение обеих матриц на горизонтальные полосы.

Далее в разделе рассматриваются широко известные алгоритмы Фокса и Кэннона, основанные на блочном разделении матриц. При использовании одинаковой схемы разбиения матриц данные алгоритмы отличаются характером выполняемых операций передачи данных. Для алгоритма Фокса в ходе вычислений осуществляется рассылка и циклический сдвиг блоков матриц, в алгоритме Кэннона выполняется только операция циклического сдвига.

Различие в способах разбиения данных приводит к разным топологиям коммуникационной сети, при которых выполнение параллельных алгоритмов является наиболее эффективным. Так, алгоритмы, основанные на ленточном разделении данных, ориентированы на топологию сети в виде гиперкуба или полного графа. Для реализации алгоритмов, основанных на блочном разделении данных, необходимо наличие топологии решетки.

На рис. 8.12 на общем графике представлены показатели ускорения, полученные в результате выполнения вычислительных экспериментов для всех рассмотренных алгоритмов. Как можно заметить, при использовании четырех процессоров некоторое преимущество по ускорению имеет параллельный алгоритм при ленточном разделении данных. Выполненные расчеты показывают также, что при большем количестве процессоров более эффективными становятся блочные алгоритмы умножения матриц.

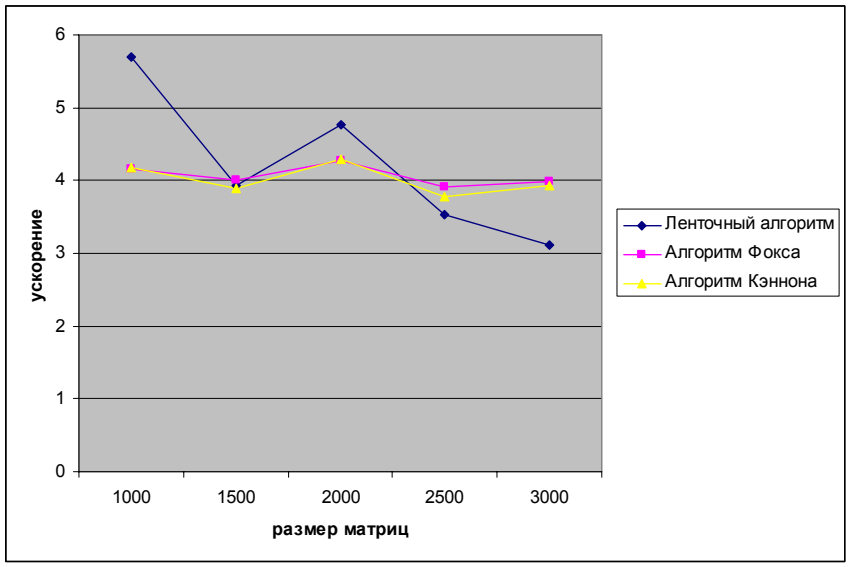

Рис. 8.12. Ускорение трех параллельных алгоритмов при умножении матриц с использованием 4 процессоров.

8.7. Обзор литературы

Задача умножения матриц широко рассматривается в литературе. В качестве дополнительного учебного материала могут быть рекомендованы работы Воеводин В.В. и Воеводин Вл.В. (2002), Kumar (1994) и Quinn (2003). Широкое обсуждение вопросов параллельного выполнения матричных вычислений выполнено в работе Dongarra, et al. (1999).

При рассмотрении вопросов программной реализации параллельных методов может быть рекомендована работа Blackford, et al. (1997). В данной работе рассматривается хорошо известная и широко используемая в практике параллельных вычислений программная библиотека численных методов ScaLAPACK.

8.8. Контрольные вопросы

1. В чем состоит постановка задачи умножения матриц?
2. Приведите примеры задач, в которых используется операция умножения матриц.
3. Приведите примеры различных последовательных алгоритмов выполнения операции умножения матриц. Отличается ли их вычислительная трудоемкость?
4. Какие способы разделения данных используются при разработке параллельных алгоритмов матричного умножения?
5. Представьте общие схемы рассмотренных параллельных алгоритмов умножения матриц.
6. Проведите анализ и получите показатели эффективности ленточного алгоритма при горизонтальном разбиении перемножаемых матриц.
7. Какие информационные взаимодействия выполняются для алгоритмов при ленточной схеме разделения данных?
8. Какие информационные взаимодействия выполняются для блочных алгоритмов умножения матриц?
9. Какая топология коммуникационной сети является целесообразной для каждого из рассмотренных алгоритмов?
10. Какой из рассмотренных алгоритмов характеризуется наименьшими и наибольшими требованиями к объему необходимой памяти?
11. Какой из рассмотренных алгоритмов обладает наилучшими показателями ускорения и эффективности?
12. Оцените возможность выполнения матричного умножения как последовательности операций умножения матрицы на вектор.
13. Дайте общую характеристику программной реализации алгоритма Фокса. В чем могут состоять различия в программной реализации других рассмотренных алгоритмов?
14. Какие функции библиотеки MPI оказались необходимыми при программной реализации алгоритмов?

8.9. Задачи и упражнения

1. Выполните реализацию двух ленточных алгоритмов умножения матриц. Сравните времена выполнения этих алгоритмов.

2. Выполните реализацию алгоритма Кэннона. Постройте теоретические оценки времени работы этого алгоритма с учетом параметров используемой вычислительной системы. Проведите вычислительные эксперименты. Сравните результаты реальных экспериментов с ранее полученными теоретическими оценками.

3. Выполните реализацию блочных алгоритмов умножения матриц, которые могли бы быть выполнены для прямоугольных процессорных решеток общего вида.

4. Выполните реализацию матричного умножения с использованием ранее разработанных программ умножения матрицы на вектор.