Глава 6

ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ

§ 6.1. Принципы создания параллельных алгоритмов

Для последовательных ЭВМ накоплен большой запас алгоритмов вычислительной математики, оформленных в виде программ на ЯВУ, ассемблере, языках описания алгоритмов.

К алгоритмам предъявлялся ряд требований, например: минимальное время исполнения задачи, минимальный объем памяти, минимизация обращений к внешним устройствам и т. д. Эти требования часто противоречили друг другу и приводили, в конце концов, к появлению ряда вычислительных алгоритмов для решения одной и той же задачи.

В связи с появлением параллельных ЭВМ возникла необходимость в разработке *параллельных алгоритмов*. Следует заметить, что признаки параллельных алгоритмов можно обнаружить и в пакетах программ для последовательных ЭВМ там, где параллелизм присутствует в естественном виде, например, в матричной алгебре, при решении систем уравнений и т. д. В этих случаях возможным является даже эффективное автоматическое преобразование последовательных программ в параллельные с помощью автоматических распараллеливателей или векторизующих компиляторов. Однако чаще необходимо специально разрабатывать и обосновывать параллельные алгоритмы. Это и является задачей новой и бурно развивающейся отрасли вычислительной математики — *параллельной вычислительной математики*.

Для описания свойств параллельного алгоритма в литературе используется ряд характеристик. Внешне некоторые из них сходны с характеристиками параллельных ЭВМ, описанных в § 1.2, однако отличаются от последних по существу, поскольку не связаны с аппаратурой. Рассмотрим некоторые характеристики алгоритмов.

1. Параллелизм алгоритма, задачи, степень параллелизма. Эти понятия определяют число операций l, которые можно выполнять одновременно, параллельно. Причем при определении l считается, что алгоритм выполняется на идеализированном параллельном компьютере, в котором отсутствуют задержки на коммутацию и

конфликты при обращении к памяти, а число процессоров не ограничивает параллелизм. В таком компьютере параллелизм зависит только от внутренних свойств алгоритма. Будем называть такие алгоритмы параллельными математическими алгоритмами в отличие от прикладных параллельных алгоритмов, которые выполняются на параллельных ЭВМ с конечным и часто небольшим числом процессоров. В главе 6 рассматриваются в основном математические алгоритмы.

Если l характеризует часть задачи с основным объемом вычислений, эту величину называют потенциальным параллелизмом и выражают как функцию $f(e)$ или $f(n)$, где e — общее количество арифметико-логических операций в задаче; n — размерность задачи, например, длина вектора, размер стороны квадратной матрицы, число уравнений в системе.

Представление $l = f(e)$ позволяет выбрать один из численных методов среди множества возможных методов для решения данной задачи. Представление $l = f(n)$ позволяет оценить варианты внутри одного численного метода.

Величина l оценивается только по порядку и может принимать, например, такие значения: $0(\log n)$, $0(n)$, $0(n^2)$, где $0(Z)$ обозначает "порядка величины Z".

2. Время исполнения параллельного алгоритма (глубина параллелизма, алгоритмическая сложность). Эти понятия отражают время исполнения параллельного алгоритма q, выраженное в тактах. Величина q информативнее ширины параллелизма l. Однако и q не несет полной информации о качестве параллельного алгоритма. Если l описывает алгоритм по горизонтали, то q — по вертикали.

Качественно q можно определить как

$$q = e/l \qquad (6.1)$$

Формула (6.1) указывает на средства, позволяющие получить лучший параллельный алгоритм: необходимо уменьшать e и увеличивать l. В некоторых случаях при реорганизации алгоритма эффект можно получить и увеличением e, когда l растет быстрее, чем e: $q' = (e + \Delta e)/(l + \Delta l)$, $q' < q$.

Глубину параллелизма также выражают как функцию $f(e)$ или $f(n)$. Параллельные алгоритмы можно разделить на алгоритмы с

фиксированной глубиной параллелизма, алгоритмы с глубиной типа 0(log n), алгоритмы с большей глубиной параллелизма.

В алгоритмах с фиксированной глубиной параллелизма величина q не зависит от объема вычислений или размера структуры данных.

Примерами таких алгоритмов являются простейшие бинарные операции над векторами $C(*) = A(*) \otimes B(*)$, где A, B, C — векторы длиной n; $\otimes$ — символ произвольной арифметико-логической операции. Такая операция будет выполняться за один такт при любом n.

Особенное значение получили алгоритмы с $q = 0(\log n)$. Примеры таких алгоритмов даны в § 6.2. Это алгоритмы каскадного суммирования, циклической редукции и некоторые другие, связанные с распараллеливанием рекурсивных алгоритмов.

Алгоритмы с большей глубиной параллелизма $l = 0(n \log n)$, $l = 0(n^2 \log n)$ и другие малоэффективны, и необходимо искать более быстрые алгоритмы.

3. Наиболее информативной характеристикой параллельного алгоритма является ускорение r, показывающее, во сколько раз применение параллельного алгоритма уменьшает время исполнения задачи по сравнению с последовательным алгоритмом: $r = e_{nc}/q$.

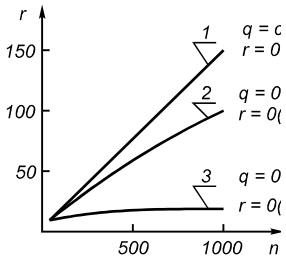

Рис. 6.1. Характер ускорения для параллельн
глубиной параллелиз

На рис. 6.1 в качественной форме отражено ускорение для задач с различной глубиной параллелизма. Кривая *1* относится к операциям типа $C(*) = A(*) \otimes B(*)$, кривая 2 — к задачам типа каскадных сумм, где общее число операций равно L, а кривая 3 — к умножению матриц одним из методов, для которого $e_{nc} = n$.

Для реальных параллельных ЭВМ на основе математических параллельных алгоритмов разрабатываются параллельные прикладные алгоритмы, характеристики которых хуже, чем у математических алгоритмов. Прикладные алгоритмы должны учитывать наличие в реальной ЭВМ оперативной памяти, системы коммута

ции и ограниченного числа процессоров. Это значительно усложняет создание прикладных алгоритмов, так как при их разработке по сравнению с математическими алгоритмами используются другие критерии. Для математических алгоритмов требовалось уменьшать e и максимально увеличивать l. Для прикладных алгоритмов нет смысла в слишком большом l, так как оно должно быть равно N. В результате если за счет ограничения l удастся уменьшить e, то это повысит качество разрабатываемого прикладного алгоритма.

Параллельные системы предназначены для решения задач большой размерности. Поэтому среди прочих источников погрешностей заметной становится погрешность округления. Известно, что среднеквадратичное значение погрешности округления $\sigma = 0(\beta\sqrt{e})$, где β — значение младшего разряда мантиссы числа. Для чисел с плавающей запятой одинарной точности $\beta = 2^{-24}$. Величина e для ЭВМ с быстродействием 100 млн. оп/с за время работы около 1 ч достигает значения примерно 10. В результате σ может иметь порядок $0(2^{-24} \cdot 10^6) = 0(2^{-4})$, т. е. погрешность составляет несколько процентов и более, что не всегда допустимо, поэтому для параллельных вычислений часто используется двойная точность.

Далее будут рассмотрены способы построения параллельных алгоритмов для некоторых задач вычислительной математики: суммирование чисел, умножение матриц, решение дифференциальных уравнений в частных производных. Эти задачи характеризуются тем, что они охватывают основные методы вычислительной математики: рекурсию, прямые и итерационные методы решения.

§ 6.2. Некоторые виды параллельных алгоритмов

Рассмотрим отдельные методы распараллеливания рекурсивных алгоритмов, алгоритмов умножения матриц и решения дифференциальных уравнений в частных производных. Эти задачи характерны для практики и предъявляют особые требования к структуре параллельных ЭВМ [2].

Рекурсивные алгоритмы. *Рекурсия* — это последовательность вычислений, при которых значение очередного члена в последовательности зависит от одного или нескольких ранее вычисленных членов последовательности.

Вычисление рекурсии по определению имеет последовательный характер и параллелизм здесь достигается только реорганизацией вычислений. При этом реорганизация вычислений, как правило, приводит к увеличению общего числа операций.

Рассмотрим методику разработки параллельного алгоритма на примере линейной рекурсии первого порядка.

Пусть имеется рекуррентное соотношение

$$x_j = x_{j-1} + d_j,\ j=1,2,\ ...,\ n \qquad (6.2)$$

где $x_j = \sum_{k=1}^{j} d_k$. Этот метод вычислений последовательных сумм можно реализовать с помощью n — сложений в простой программе на Фортране:

```
      X(1) = D(1)
      DO 1 J = 2, N
1     X(J) = X(J - 1) + D(J)
```

На рис. 6.2 для n = 8 представлена зависимость между способом хранения, арифметическими операциями и операциями коммутации. Имеющаяся последовательность перемещается снизу вверх: операции, которые могут вычисляться параллельно, показаны на одном и том же уровне. Ясно, что на каждом временном уровне может быть выполнена только одна операция. Таким образом, алгоритм последовательных сумм имеет n - 1 операцию сложения со степенью параллелизма 1.

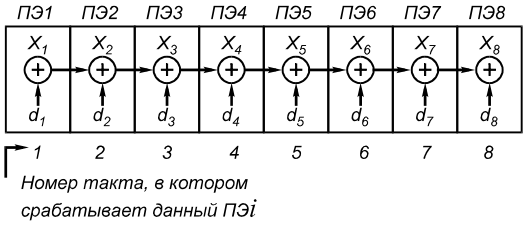

Рис. 6.2. Формирование всех частных сумм для восьми чисел

Чтобы дать количественную оценку схеме коммутации, предположим, что горизонтальная ось на рис. 6.2 указывает относительное размещение данных в памяти последовательной или параллельной ЭВМ. Единичная операция коммутации означает параллельный сдвиг всех элементов массива в соседние блоки памяти последовательной или параллельной ЭВМ. Единичный метод последовательных сумм требует коммутации одной единицы вправо на каждом временном уровне, поэтому метод последовательных сумм приводит к n - 1 операции коммутации со степенью параллелизма 1.

Параллельный метод решения рекурсивных задач изображен на рис. 6.3 для n = 8. Набор из n аккумуляторов (регистров, ячеек памяти) сначала загружается данными, которые должны суммироваться. На первом уровне копия содержимого аккумуляторов сдвигается на одну позицию вправо и складывается с несдвинутым содержимым аккумуляторов, чтобы сформировать сумму данных от смежных пар. На следующем уровне процесс повторяется, но со сдвигом на две позиции вправо, тем самым получаем сумму от

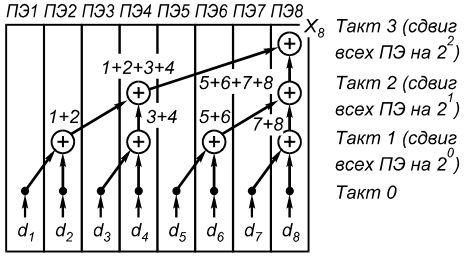

Рис. 6.3. Схема формирования сумм методом параллельных каскадных сумм

групп по четыре числа. Поскольку производятся сдвиги, в левую часть заносятся нули. В дальнейшем на l-м временном уровне выполняется сдвиг на 2^l позиций, и на уровне $l = \log_2 n$ аккумуляторы содержат требуемые частные суммы.

Метод параллельных каскадных сумм можно описать на векторном фортраноподобном языке:

```
      X = D
      DO 1 L = 1, LOG 2N
1     X = X + SHIFTR (X, 2 ** (L - 1))
```

где X и D — векторы из n элементов; знак "+" — параллельное сложение по n элементам; *SHIFTR* (X, L) — векторная функция, которая помещает копию вектора X, полученную смещением L ячеек памяти вправо (*RIGHT*), во временный вектор *SHIFTR*. Элементы вектора X не затрагиваются. Поэтому метод каскадных частных сумм требует $\log_2 n$ сложений со степенью параллелизма n.

Если мы используем связи типа "к ближнему соседу", то на уровне l потребуется 2^{l-1} единичных операций коммутации, что дает в целом 1 + 2 + 4 +...+ n/2 или n - 1 операций коммутации со степенью параллелизма n.

Для каждой операции метода каскадных частных сумм характерна максимально возможная степень параллелизма n, т. е. имеет место 100%-й параллелизм. Однако общее число скалярных арифметических операций увеличилось с n - 1 для метода последовательных сумм до $n \log_2 n$. Поэтому для последовательных ЭВМ метод каскадных сумм не может использоваться.

Метод каскадных частных сумм значительно упрощается в каждом специальном случае, когда требуется только один результат общей суммы (в нашем случае X_8). На рис. 6.3 отмечены те схемы перемещения и арифметические операции, которые связаны с подсчетом X_8, и они составляют только часть общего объема вычислений. Вычисления, связанные с общей суммой, формируют двоичное дерево (в виде каскада), где число операций и параллелизм делятся пополам на каждом l-м уровне вычисления. Для простоты примем, что n — степень 2. Тогда число операций для метода каскадных сумм составит одно сложение со степенью параллелизма $n \cdot 2^{-l}$ для l = 1, 2, ..., $\log_2 n$. Следовательно, число скалярных сложений в алгоритме: n/2 + n/4 +...+ 2 + 1 = n - 1, т. е. метод каскадной общей суммы имеет такое же число скалярных операций, что и метод последовательных сумм. Метод каскадной общей суммы обычно называют методом каскадной суммы.

Рассмотрим параллельный алгоритм для линейной рекурсии более общего характера, чем в (6.2):

$$x_j = a_j x_{j-1} + d_j,\ j=1,2,\ ...,\ n$$

Последовательная программа будет выглядеть так:

```
      X(1) = A(1) ∗ X(0) + D(1)
      DO 1 J = 2, N
1     X(J) = A(J) ∗ X(J - 1) + D(J)
```

Она требует $2n$ арифметических операций со степенью параллелизма 1 и n операций коммутации со степенью параллелизма 1 (рис. 6.4). Для простоты представлены различные типы арифметических операций, хотя можно было бы ограничиться и одним типом, так как, например, для конвейерных ЭВМ выдача результата сложения или умножения занимает один такт одинаковой длительности.

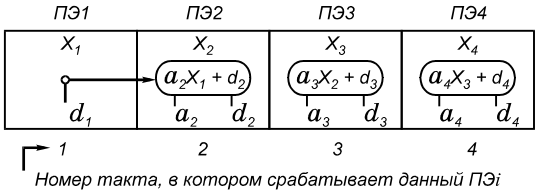

Рис. 6.4. Схема вычисления рекурсии общего вида последовательным методом

Для распараллеливания рекурсий подобного вида используется алгоритм циклической редукции, который позволяет выполнить указанную рекурсию за $3\log_2 n$ операций со степенью параллелизма n.

Умножение матриц. Матричное умножение хорошо иллюстрирует различные способы реорганизации вычислений для согласования с архитектурой ЭВМ, на которой этот алгоритм должен выполняться. Вычисление элементов результирующей матрицы C производится по следующей базовой формуле:

$$C_{i,j} = \sum_{k=1}^{n} A_{i,k} \cdot B_{k,j},\ 1 \le i;\ j \le n,$$

где i — номер строки; j — номер столбца.

Существует несколько методов умножения матриц.

Если правую часть приведенной формулы обозначить *R*, то все методы умножения матрицы описываются в виде гнезда из трех вложенных циклов (рассматриваются для простоты квадратные матрицы):

```
      DO 1 I = 1, N                DO 1 J = 1, N
      DO 1 J = 1, N                DO 1 I = 1, N
      DO 1 K = 1, N                DO 1 K = 1, N          (6.3, а, б)
1     C (I, J) = R           1     C (I, J) = R

      DO 1 I = 1, N                DO 1 J = 1, N
      DO 1 K = 1, N                DO 1 K = 1, N
      DO 1 J = 1, N                DO 1 I = 1, N          (6.3, в, г)
1     C (I, J) = R           1     C (I, J) = R

      DO 1 K = 1, N                DO 1 K = 1, N
      DO 1 I = 1, N                DO 1 J = 1, N
      DO 1 J = 1, N                DO 1 I = 1, N          (6.3, д, е)
1     C (I, J) = R           1     C (I, J) = R
```

В зависимости от расположения индекса *K* методы умножения матриц называются методами внутреннего произведения (6.3 а, б), среднего произведения (6.3 в, г), внешнего произведения (6.3 д, е).

Метод внутреннего произведения для перемножения матриц на последовательных ЭВМ основан на использовании гнезда циклов (6.3 а), где предполагается, что все элементы *C* (*I*, *J*) матрицы перед занесением программы устанавливаются в нуль. Оператор присваивания в программе (6.3 а) формирует внутреннее произведение *i*-й строки матрицы *A* и *j*-го столбца матрицы *B*. Это последовательное вычисление суммы множества чисел, которое описывалось ранее (пусть $d_k = A_{i,k} \cdot B_{k,j}$ в уравнении (6.2), тогда $X_n = C_{i,j}$). Здесь можно производить вычисления последовательно, как в программе (6.3 а), или использовать метод каскадных сумм. В некоторых ЭВМ предусмотрены даже команды внутреннего или скалярного произведения.

Умножению матриц присуща большая степень параллелизма, чем в случае вычисления одиночной суммы. Умножение матриц

включает в себя вычисление n внутренних произведений. В частности, внутренний цикл по K в программе (6.3 а) может быть заменен векторной операцией:

```
      DO 1 I = 1, N
      DO 1 J = 1, N                                          (6.4)
1     C (I, J) = C (I, J) + A (I, *) * B (*, J)
```

где A (I, $*$) — вектор-строка матрицы A; B ($*$, J) — вектор столбец матрицы B. Степень параллелизма этой векторной операции равна n при умножении. При сложении членов суммы результирующего вектора может применяться метод каскадных сумм.

Замена индексов I и J согласно (6.3) степени параллелизма не изменяет.

Метод среднего произведения основывается на перестановке порядка циклов *DO* в программе (6.3 а). Если переставить цикл по строкам в самую внутреннюю позицию, то получится программа, которая подсчитывает параллельно внутреннее произведение по всем элементам столбца матрицы C (см. (6.3 г)).

Члены в цикле по I могут вычисляться параллельно, поэтому цикл (6.3 г) может быть заменен векторным выражением:

```
      DO 1 J = 1, N
      DO 1 K = 1, N                                          (6.5)
1     C (*, J) = C (*, J) + A (*, K) * B (K, J)
```

где $C(*, J)$ и $A(*, K)$ — векторы, составленные из J-го и K-го столбцов матриц C и A. Операция сложения представляет собой параллельное сложение n элементов, а операция умножения — умножение скаляра $B(K, J)$ на вектор $A(*, K)$. Здесь за каждое выполнение последнего оператора (K не меняется) все элементы столбца $C(I, J)$ увеличиваются на одно произведение $A(I, K)*B(K, J)$. За K тактов полностью формируется столбец матрицы J.

Степень параллелизма программы для метода среднего произведения равна n, а степень параллелизма для исходного метода внутреннего произведения равна единице. Этому способствовало изменение порядка циклов *DO*. Можно было бы переместить цикл по J в середину, обеспечив тем самым параллельное вычисление всех внутренних произведений строки. Однако элементы столбца

матрицы обычно хранятся в смежных ячейках памяти (в Фортране матрицы хранятся в памяти по столбцам), следовательно, конфликты на уровне блоков памяти уменьшатся, если векторные операции будут производиться над векторами столбцов. Поэтому лучше использовать программу (6.3).

Метод внешнего произведения основывается на вынесении цикла по K наружу, т. е. рассматривается вариант (6.3 д, е). Последний оператор можно заменить матричноподобным одиночным оператором, где один член внутреннего произведения подсчитывается параллельно для всех n^2 элементов матрицы C. Если всю матрицу обозначим C (∗, ∗), то получим

```
      DO 1 K = 1, N
1     C (∗, ∗) = C (∗, ∗) + A (∗, K) ∗ B (K, ∗)          (6.6)
```

где операция умножения представляет собой поэлементное перемножение матрицы $n \times n$, полученной дублированием K-го столбца матрицы A, и матрицы $n \times n$, полученной дублированием K-й строки матрицы B. Операция сложения представляет собой поэлементное сложение в матрице C (∗, ∗).

Дифференциальные уравнения в частных производных. Дифференциальное уравнение в частных производных (ДУЧП) второго порядка в двух размерностях можно представить в общем виде

$$A(x,y)\frac{\partial^2 j}{\partial x^2} + B(x,y)\frac{\partial j}{\partial x} + C(x,y)\frac{\partial^2 j}{\partial y^2} + \\ + D(x,y)\frac{\partial j}{\partial y} + E(x,y)j = r(x,y), \qquad (6.7)$$

где коэффициенты A, B, C, D, E — произвольные функции позиции.

Это уравнение включает в себя основные уравнения математической физики и техники (уравнения Гельмгольца, Пуассона, Лапласа, Шредингера и уравнение диффузии) в общепринятых координатных системах (декартовой (x, y), полярной (r, θ), цилиндрической (r, z), аксиально-симметричной сферической (r, θ) и сферической (θ, φ)). Если уравнение (6.7) продифференцируем на $n \times n$ сетке точек с помощью стандартных процедур, то получим

множество алгебраических уравнений, каждое из которых имеет отношение к значениям переменных по пяти соответствующим точкам сетки:

$$\begin{matrix} a_{p,q}\varphi_{p,q-1} + b_{p,q}\varphi_{p,q+1} + c_{p,q}\varphi_{p-1,q} + \\ + d_{p,q}\varphi_{p+1,q} + e_{p,q}\varphi_{p,q} = f_{p,q} \end{matrix}, \qquad (6.8)$$

где целочисленные индексы p, q = 1, 2, ..., n обозначают точки сетки в направлениях x и y соответственно. Коэффициенты a, b, c, d, e изменяются от точки к точке сетки и относятся к функциям A, B, C, D, E. Для разделения точек сетки можно использовать частную разностную аппроксимацию. Переменная в правой части $f_{p,q}$ представляет собой линейную комбинацию значений $\rho(x, y)$ в точках сетки вблизи (p, q). В простейшем случае она является значением $\rho(x, y)$ в точке сетки (p, q).

Итерационные процедуры описываются сначала предположительными значениями φ^{p-q} во всех точках сетки и дифференциальным уравнением (6.8). Итерации повторяются, и если успешно, то значения φ стремятся к решению уравнения (6.8) во всех точках сетки.

В самой простой процедуре значения φ во всех точках сетки одновременно настраиваются на значения, которые имели бы по уравнению (6.8), если допустить, что все соседние значения правильные, т.е. φ в каждой точке сетки заменяются новым значением:

$$\varphi^*_{p,q} = \frac{f_{p,q} - a_{p,q}\varphi_{p,q-1} - b_{p,q}\varphi_{p,q+1} - c_{p,q}\varphi_{p-1,q} - d_{p,q}\varphi_{p+1,q}}{e_{p,q}}. \qquad (6.9)$$

Поскольку замена должна производиться одновременно, то все значения φ в правой части являются старыми значениями от последней итерации, а отмеченные звездочкой значения в левой части — новыми уточненными значениями.

Описанный метод называется *методом Якоби*, он идеально подходит для исполнения на параллельных ЭВМ. Одновременное выравнивание означает, что для всех точек уравнение (6.9) может выполняться параллельно с максимально возможным уровнем параллелизма n^2. Типичное значение n = 32...256, так что уровень

параллелизма изменяется от 1024 до 65 536. Тот факт, что одиночная итерация метода одновременной замены может быть эффективно реализована на параллельных ЭВМ, вовсе не означает, что метод хорош для использования, так как необходимо учитывать число итераций, требуемых для получения удовлетворительной сходимости. Скорость сходимости нельзя найти для общего уравнения (6.8), но известны аналитические результаты для простого случая уравнения Пуассона в квадрате с нулевыми граничными значениями (условие Дирихле). Сказанное соответствует принятию $a_{p,q} = b_{p,q} = c_{p,q} = d_{p,q} = 1$ и $e_{p,q} = -4$ для всех точек сетки и часто называется модельной задачей.

Обычно для модельной задачи используются прямые, а не итерационные методы решения, но на ней удобно продемонстрировать методику оценки и выбора алгоритма для параллельной ЭВМ.

Известно, что для модельной задачи число итераций t, необходимое для уменьшения погрешности на коэффициент 10^{-p}, равняется $t = pn^2/2$. Таким образом, чтобы по методу Якоби достичь уменьшения ошибки, например, в 10^{-3} раз, на типичной сетке 128×128 требуется 24 000 итераций. Такой медленный характер сходимости делает метод Якоби бесперспективным для параллельных ЭВМ, несмотря на его удобства для организации параллельных вычислений.

Наиболее широко для последовательных ЭВМ используется *метод последовательной повторной релаксации* (SOR). Для этого метода характерно использование взвешенного среднего от старых и новых (отмеченных звездочкой) значений:

$$\varphi_{pq}^{i} = \omega\varphi_{p,q}^{*} + (1-\omega)\varphi_{p,q}^{cm}, \qquad (6.10)$$

где ω — постоянный коэффициент релаксации (обычно лежащий в диапазоне $1 \le \omega \le 2$), который выбирается для увеличения скорости сходимости. Можно показать, что для модельной задачи наилучшая скорость сходимости достигается при

$$\omega = \frac{2}{1+(1-\rho^2)^{1/2}}, \qquad (6.11)$$

где ρ — коэффициент сходимости соответствующей итерации Якоби. Отсюда для модельной задачи $\rho = \cos(\sigma/n)$. Как правило, точки сетки обрабатываются последовательно, точка за точкой и строка за строкой. Улучшение сходимости связано с тем, что новые значения, как только их вычислили, заменяют старые; следовательно, значения в правой части уравнения (6.9), используемые для вычисления φ^*, представляют собой смесь старых и новых значений, и уравнение (6.9) нельзя решить для всех точек в параллель, как это делалось по методу Якоби.

Из сказанного может сложиться мнение, что метод SOR, несмотря на хорошую сходимость, непригоден для реализации на параллельных ЭВМ.

Существуют варианты метода SOR для применения на параллельных ЭВМ. Лучшим из вариантов является четное-нечетное упорядочение по *методу Чебышева*. В соответствии с методом Чебышева точки делятся на две группы; четной или нечетной является сумма $p + q$ (рис. 6.5). Метод возобновляет половину итераций, во время каждой из которых выравнивается только половина точек (выборочно нечетное и четное множество точек) в соответствии с уравнениями (6.9)...(6.11). Кроме того, значение ω изменяется в каждой полуитерации:

$$\omega^0 = 1,$$

$$\omega = 1/(1 - 1/2 \cdot \rho^2),$$

$$\omega^{(k+1/2)} = 1/(1 - 1/4 \cdot \rho^2 \omega^{(k)}),\ \text{k=1/2, 1/4, ..., } \infty,$$

где k — номер итерации.

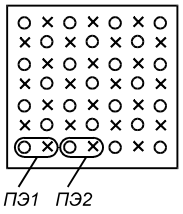

Рис. 6.5. Распределение данных при решении ДУЧП методом Чебышева.
Кружками обозначены четные, крестиками — нечетные точки

Чтобы уменьшить погрешность до уровня 10^{-p}, число итераций для метода нечетного-четного упорядочения должно быть

равно $t_{SOR} \approx np/3$. На основе этого уравнения можно подсчитать, что уменьшение погрешности до уровня 10^{-3} на 128×128 сетке потребовало бы 128 итераций по сравнению с 24 000 по методу Якоби. Это делает очевидным необходимость использования для параллельных ЭВМ хорошо сходящихся методов.

Метод Чебышева имеет дополнительные преимущества при рассмотрении его реализации на параллельных ЭВМ. Как следует из уравнения (6.9), обозначенное звездочкой значение в нечетной точке сетки зависит только от старых значений в соответствующих четных точках, которые были подсчитаны во время последней полуитерации. Таким образом, все нечетные точки могут быть выравнены параллельно при уровне параллелизма $n^2/2$ за время одной полуитерации и аналогично все четные точки — параллельно в течение следующей полуитерации. Следовательно, время одной полной итерации пропорционально $n^2/2$.

Описанные методы могут быть расширены и для трех размерностей. Таким образом, при использовании параллельных ЭВМ следует учитывать методы с разной степенью параллелизма n, n^2, n^3.

Далее в сжатом виде приводится ряд известных результатов о параллелизме алгоритмов в следующих областях [23]: алгебра, целочисленная арифметика, ряды и многочлены, комбинаторика, теория графов, вычислительная геометрия, сортировка и поиск.

Все приводимые ниже алгоритмы позволяют получить представление о состоянии параллельной математики.

В качестве модели параллельной ЭВМ, на которой выполняются рассматриваемые алгоритмы, используется широко распространенная модель *параллельной машины с произвольным доступом к памяти* — PRAM (Parallel Random Access Machine). PRAM является эталонной моделью абстрактной параллельной ЭВМ. Она состоит из p идентичных RAM (называемых процессорами), имеющих общую память. Процессоры работают синхронно, и каждый из них имеет возможность переписать в свой сумматор содержимое любой ячейки памяти. Будем считать, что каждый процессор работает по своей программе, хранящейся в локальной памяти.

Машина PRAM удобна тем, что наиболее полно отражает черты реальных параллельных ЭВМ, абстрагируясь от технических ограничений. Имеются различные типы PRAM в зависимости от ответа на вопрос: что происходит, если несколько процессоров обращаются к одной ячейке памяти? В модели EREW PRAM одновременная запись или считывание из одной ячейки сразу в несколько процессоров запрещены. Это слабая модель PRAM. В модели CREW PRAM нескольким процессорам разрешается одновременное чтение одной ячейки памяти, но не разрешается одновременная запись в ячейку памяти.

Результаты научных исследований отражены в табл. 6.1. В ней n и m являются параметрами структуры данных.

Разумеется, приведенный перечень не исчерпывает все классы и подклассы уже разработанных алгоритмов. Кроме того, имеется и ряд нерешенных с точки зрения распараллеливания задач, среди которых следует выделить задачу о максимальном паросочетании в графе, о построении дерева поиска в глубину в произвольном графе и задачу о нахождении наибольшего общего делителя двух натуральных чисел. По-видимому, продвижение в указанных направлениях потребует разработки новых методов построения параллельных алгоритмов.

Таблица 6.1.

Характеристики некоторых параллельных алгоритмов

N пп	Наименование алгорита	Тип PRAM	Порядок времени вычислений	Число процессоров
1	2	3	4	5
	Алгебра			
1	вычисление линейной рекурсии первого порядка:	EREW	$0(\log n)$	$0(n/\log n)$
	$x_j - a_j x_{j-1} + b_j$,	EREW	$0(\log k \log n)$	
	j=1, ..., n линейная рекурсия k-го порядка			
2	Решение треугольной системы уравнений, обращение треугольной матрицы	EREW	$0(\log^2 n)$	$0(n^{\omega} / \log n)$

3	Вычисление коэффициентов характеристического уравнения матрицы	EREW	$0(\log^2 n)$	$0(n^4 / \log^2 n)$
4	Решение системы линейных уравнений, обращение матрицы	EREW	$0(\log^2 n)$	$0(n^4 / \log^2 n)$
5	Метод исключения Гаусса	-	$0(\log^2 n)$	$0(n^{\omega+1})$
6	Вычисление ранга матрицы	-	$0(\log^2 n)$	полиномиальное
7	Подобие двух матриц	-	$0(\log^2 n)$	
8	Нахождение *LU*-разложения симметричной матрицы	EREW	$0(\log^3 n)$	$0(n^4 / \log^2 n)$
9	Отыскание собственных значений Якобиевой матрицы с $\varepsilon > 0$	-	$0(\log^4 n + \log^3 n (\log\log 1/\varepsilon)^2)$	$0(n(\log^4 n + \log^3 n(\log\log$
	Ряды и многочлены			
1	Умножение многочленов $A_1(x)\ldots A_m(x)$, степени n	-	$0(\log(m\cdot n))$	$0(mn\log(mn))$
1	2	3	4	5
2	Вычисление степенных рядов (и членов)	-	$0(\log n)$	
3	Вычисление корней многочлена с погрешностью $\varepsilon > 0$, L — длина записи многочлена	-	$0(\log^3 n(\log n + +\log\log^2(L/\varepsilon))$	
	Комбинаторика			
1	ε — оптимальный рюкзак, n — размер	-	$0(\log n \log(n/\varepsilon))$	$0(n^3 / \varepsilon^2)$

	ность задачи			
2	Задача о покрытии с гарантированной оценкой отклонения не более, чем в (1+ε)log *d* раз	-	$0(\log^2 n \log m)$	$0(n)$
3	Нахождение ε — хорошей раскраски в задаче о балансировке множеств	-	$0(\log^3 n)$	полиномиальное число
	Теория графов			
1	Ранжирование списка	-	$0(\log n)$	$0(n/\log n)$
2	Эйлеров путь в дереве	-	$0(\log n)$	$0(n/\log n)$
3	Отыскание основного дерева минимального веса	CREW	$0(\log^2 n)$	
4	Транзитивное замыкание	EREW	$0(\log^2 n)$	$0(n^{\omega} / \log n)$
5	Раскраска вершины в Δ + 1 и Δ цветов	CREW	$0(\log^3 n \log\log n$	$0(n+m)$
6	Дерево поиска в глубину для графа	-	$0(\log^3 n)$	$0(n)$
	Сортировка и поиск			
1	Сортировка	EREW	$0(\log n)$	$0(n)$
2	Слияние для двух массивов размера *n* и *m*, *N* = *m*+*m*	EREW	$0(\frac{N}{P} + \log N)$	$P = 0(N / \log$

§ 6.3. Особенности выполнения параллельных алгоритмов на ЭВМ различных типов

Рассмотрим особенности выполнения задачи умножения матриц на конвейерных ЭВМ, процессорных матрицах и многопроцессорных ЭВМ типа МКМД. Эта задача выбрана из-за простоты, краткости и наглядности представления.

Конвейерные ЭВМ. В качестве образца конвейерной ЭВМ для экспериментов над параллельными алгоритмами возьмем упрощенный вариант ЭВМ CRAY-1. Напомним ее основные характеристики: такт синхронизации 12,5 нс; память состоит из 16 блоков, обращение к каждому блоку занимает 4 такта; имеется ряд функциональных устройств (векторные АЛУ для чисел с плавающей запятой и длиной конвейера 6 и 7 тактов; временем подключения этих устройств к векторным регистрам для простоты будем пренебрегать) и восемь векторных регистров, каждый по 64 слова. Более подробная информация о машине CRAY-1 представлена в § 2.2.

Приведем примеры стандартного размещения вектора длиной 16 элементов и матрицы размерностью 16×16 (рис. 6.6). Стандартным для расположения матриц в Фортране является размещение по столбцам. Если размер матрицы кратен числу блоков памяти, то стандартное размещение матрицы приводит к тому, что элементы столбцов расположены в разных блоках памяти и их можно считывать по очереди с максимальной скоростью: один элемент за один такт. Но каждая строка оказывается полностью размещенной в одном блоке памяти, поэтому последовательные элементы строк считываются с минимальной скоростью: один элемент за четыре такта (см. рис. 6.6, а).

Определенные проблемы существуют и в размещении векторов (см. рис. 6.6, б). Последовательные элементы вектора $a_1, a_2, a_{3,}$ ... и $a_1, a_5, a_9, ...$ с кратностью 4 считываются с максимальной скоростью элементы $a_1, a_9, a_{17}, ...$ с кратностью 8 считываюся с половинной скоростью, так как за время цикла памяти (четыре такта) в каждый блок поступает два запроса; элементы $a_1, a_{17}, ...$ с кратностью 16, расположенные в одном блоке, считываются со скоростью в четыре раза ниже максимальной. Эти ситуации могут встречаться при использовании методов каскадных сумм или циклической редукции.

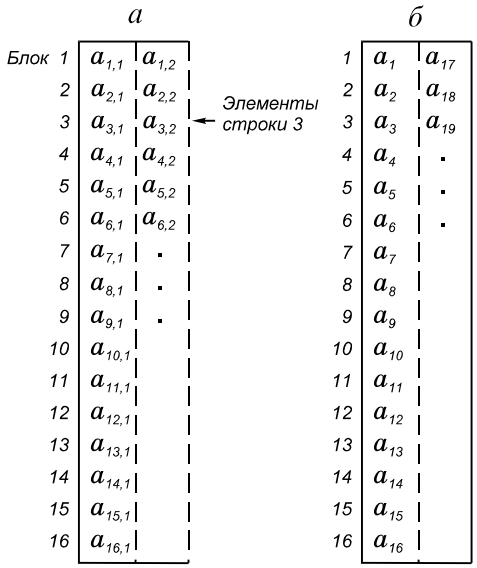

Рис. 6.6. Размещение матриц (*а*) и векторов (*б*) в многоблочной памяти конвейерной ЭВМ

В конвейерной ЭВМ выделяют три вида производительности:

1) скалярная производительность достигается только при использовании скалярных команд и регистров;

2) векторная производительность достигается при использовании программ с векторными командами, скорость выполнения которых ограничивается скоростью обмена с памятью;

3) сверхвекторная производительность достигается при использовании программы с векторными командами, скорость выполнения которых ограничивается готовностью векторных регистров или функциональных устройств.

В CRAY-1 скорости выполнения программ для скалярных и векторных команд заметно отличаются и равны 12 и 153 Мфлоп/с соответственно.

При умножении матриц нас будут интересовать только векторная и сверхвекторная производительности, причем основными средствами ускорения вычислений являются:

- обеспечение максимальной скорости обращения к памяти;
- уменьшение объема обращений к памяти и соответственно увеличение объема работы с векторными регистрами;
- увеличение длины векторов, чтобы уменьшить влияние времени “разгона” (определяется длиной конвейера функциональных устройств);
- максимальное использование зацепления операций.

Для простоты дальнейшего анализа рассмотрим матрицы размерностью 64×64. При использовании метода внутреннего произведения (см. (6.4)) необходимо выполнить следующий алгоритм для вычисления одного элемента матрицы C (рис. 6.7, а).

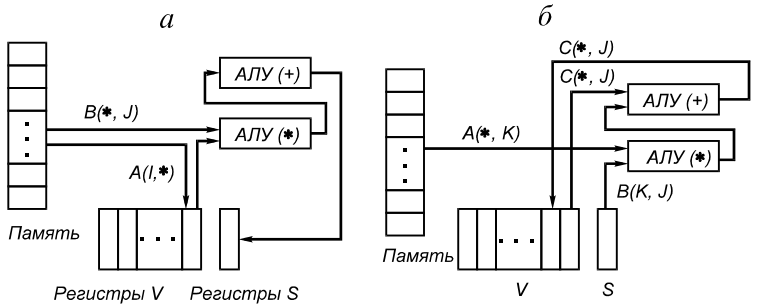

Рис. 6.7. Схема соединений конвейерной ЭВМ для выполнения операций умножения матриц:
а — метод внутреннего произведения; б — метод среднего произведения

Из памяти в векторный регистр выбирается строка $A(I, *)$ матрицы A. При стандартном размещении матриц все элементы строки оказываются в одном блоке памяти, поэтому на выборку строки затрачивается время (в тактах) $t_1=l_1+4n$, где n — размер стороны матрицы, а l_1 — время “разгона” (длина конвейера) памяти (для CRAY-1 $l_1 = 11$). Затем производится поэлементное умножение векторов в АЛУ (∗). Суммирование получаемых произведений в АЛУ (+) выполняется в зацеплении с предыдущей операцией. Результат C (I, J) заносится в один из скалярных регистров S. Это потребует времени $t_2=l_2+l_3+n$, где $l_2 = 7$ и $l_3 = 6$ — время “разгона” конвейерных АЛУ сложения и умножения соответст

венно. Тогда время вычисления всей матрицы методом внутреннего произведения будет:

$$t_{вн} = (t_1 + t_2)n^2 = (l_1 + l_2 + l_3 + 5n)n^2 = 5{,}3n^3, \qquad (6.12)$$

если предположить, что $l_1 + l_2 + l_3 = 22 \approx 0{,}3$n.

При использовании описанного метода большое замедление вызывает выборка строки матрицы A. Если применим специальное (нестандартное) размещение матрицы A таким образом, чтобы смежные элементы строки были расположены в соседних блоках памяти, то обеспечим считывание каждого элемента строки за один такт, т. е. с максимальной скоростью. Тогда

$$t_{âi} = 2{,}3n^3. \qquad (6.13)$$

Но при этом требуется изменение стандартных трансляторов.

Более перспективными являются методы среднего произведения. Рассмотрим реализацию программы (6.5). Соответствующая схема для вычисления одного столбца матрицы приведена на рис. 6.7, б. При использовании метода среднего произведения столбец $C(*, J)$ постоянно располагается в регистрах V, а из блоков памяти выбираются только столбцы $A(*, K)$. Поскольку при стандартном расположении матриц элементы столбца располагаются в различных блоках памяти, то элементы $A(*, K)$ будут выбираться с максимальной скоростью. При вычислении элементов столбца матрицы C методом среднего произведения совмещают три операции: считывание $A(*, K)$ из блоков памяти; умножение этого вектора на константу $B(K, J)$, получаемую из скалярного регистра S; сложение столбца $C(*, J)$ с результирующим вектором из АЛУ $(*)$. Поэтому среднее время вычисления всей матрицы C

$$t_{ср} = n^2(l_1 + l_2 + l_3 + n) = 1{,}3n^3. \qquad (6.14)$$

При определении времени умножения матриц опущены различного рода вспомогательные операции. В частности, в последнем случае не учтены операции считывания из памяти и запись в память столбцов матрицы C. Таким образом, полученные оценки времени умножения следует считать приблизительными.

Оценим быстродействие, получаемое по формуле (6.14). При вычислении произведения матриц выполняется n^3 операций умножения и n^3 операций сложения. Для конвейерных ЭВМ длительности тактов сложения и умножения равны, поэтому будем считать, что выполняется $2n^3$ операций и, следовательно, на одну арифметико-логическую операцию затрачивается

$$\frac{t_{\text{ср}}}{2n^3} = \frac{1{,}3}{2} = 0{,}65 \text{ такта.}$$

Для CRAY-1 при длительности такта 12,5 нс это соответствует быстродействию

$$V \frac{10^9}{0{,}65 \cdot 12{,}5} = 120 \text{ Мфлоп/с.}$$

Таким образом, метод среднего произведения обеспечивает сверхвекторную производительность и считается, будучи запрограммированным на ассемблере, наилучшим методом для ЭВМ типа CRAY-1. Его преимущества обеспечиваются большой глубиной зацепления и тем, что один из векторов (столбец матрицы *C*) постоянно расположен в векторных регистрах.

Метод внутреннего произведения обеспечивает векторную (см. (6.12)) и промежуточную между векторной и сверхвекторной (см. (6.13)) производительности.

Метод внешнего произведения не имеет преимуществ перед методом среднего произведения. Более того, возможность работы с векторными регистрами и глубокое зацепление делают метод среднего произведения предпочтительным.

Процессорные матрицы. При выборе наилучшего алгоритма для ПМ необходимо учитывать следующие структурные отличия процессорных матриц от конвейерных ЭВМ.

1. Быстродействие в ПМ достигается за счет числа процессоров *N*, а не за счет глубины конвейера.

2. Основными факторами, увеличивающими время выполнения алгоритма, являются конфликты в памяти и затраты на коммутацию, поэтому размещение данных в памяти является одним из наиболее существенных моментов для процессорных матриц.

Примем для дальнейших расчетов следующее время выполнения отдельных операций в ПЭ: сложение двух чисел — 1 такт, выборка чисел из памяти — 1 такт; умножение двух чисел — 2 такта; операция коммутации для N процессоров ($N \le 2^5$) — 1 такт; операция типа SUM — $2\log_2 N$ тактов.

Далее будет рассмотрена операция умножения матриц для следующих вариантов в предположении, что n — размер стороны матрицы данных; N — общее число процессоров в ПМ:

1) $N = n$. Этот вариант позволяет рассмотреть некоторые специальные методы размещения данных;

2) $N = n^2$, $N = n^3$. В большинстве случаев данные варианты имеют теоретический интерес, так как при n = 100 (это реальное число) требуется 10^4 или 10^6 процессоров, что пока еще нельзя осуществить;

3) $N <$ n. Этот вариант является наиболее интересным с точки зрения практики.

Очевидно, что для любого N следует выбирать методы умножения матриц с наибольшим потенциальным параллелизмом.

В а р и а н т $N = n$. Для этого варианта следует применять алгоритмы с уровнем параллелизма 0 (n) и выше. Рассмотрим метод внутренних произведений (см. программу (6.4)). В основе метода лежит параллельная выборка строк матрицы A и столбцов матрицы B. Тогда алгоритм вычисления одного элемента $C(I, J)$ может быть таким:

1) параллельное считывание из памяти в регистры всех ПЭi-й строки матрицы A — 1 такт;

2) считывание J-го столбца матрицы B в регистры ПЭ — 1 такт;

3) поэлементное перемножение выбранных строки и столбца —2 такта;

4) суммирование n частных произведений — $2\log_2 n$ тактов.

Следовательно, вычисление всех элементов матрицы C займет время (в тактах)

$$t_{вн} = n^2(4 + \log_2 n).$$

При использовании метода среднего произведения (см. (6.5)) матрицы A и C располагаются горизонтально по столбцам, а матрица B — в памяти ЦУУ. Поэтому алгоритм вычисления одного столбца матрицы C будет следующим:

1) запись очередного элемента B (K, J) во все ПЭ — 1 такт;

2) считывание из памяти в ПЭ K-го столбца матрицы A — 1 такт;

3) умножение $A(*, K)$ $B(K, J)$ — 2 такта;

4) суммирование полученных частных произведений с соответствующими частными суммами элементов J-го столбца матрицы C, который постоянно располагается в регистрах ПЭ — 1 такт.

Таким образом, одна итерация вычисления $C(*, J)$ занимает 5 тактов; полное вычисление столбца $C(*, J)$ займет 5 тактов, а вычисление всей матрицы C

$$t_{ср} = 5n^2 \text{ тактов.} \qquad (6.15)$$

Очевидно, что $t_{ср} < t_{вн}$ для всех $n \geq 2$. Преимущество метода средних произведений — в отсутствии итерации вычисления каскадной суммы.

Вариант $N = n^2$. Для этого варианта следует применять алгоритмы с уровнем параллелизма $0(n^2)$ и выше. Для достижения такого параллелизма могут быть использованы различные варианты метода внешнего произведения (см. (6.6)) и комбинации метода среднего произведения. Рассмотрим метод внешнего произведения. Прямая реализация данного метода предполагает конфигурацию ПП в виде матрицы процессоров размерностью $n \times n$, причем в каждом ПЭi, j вычисляется один элемент матрицы $C(I, J)$. Для этого в памяти каждого ПЭi, j должны находится все элементы i-й строки матрицы A и j-го столбца матрицы B. Тогда вычисление всех элементов матрицы C потребует n итераций. За время одной итерации в каждом ПЭi, j выполняются следующие действия:

1) считывание из памяти элемента $A(I, K)$ — 1 такт;

2) считывание из памяти элемента $B(K, J)$ — 1 такт;

3) умножение $A(I, K) * B(K, J)$ в ПЭ — 2 такта;

4) подсуммирование полученного произведения к ранее накопленной сумме из K - 1 произведения — 1 такт.

Следовательно, вычисление матрицы займет время $t_{вн\ ш} = 5n$ тактов.

К сожалению, этот метод требует слишком большого объема памяти процессорного поля. Если суммарное число элементов в матрицах A и B равно $2n^2$, то вследствие многократного дублирования строк и столбцов матриц A и B в различных ПЭ суммарное число элементов возрастет до величины $2n^3$. Данное условие требует для матрицы размерностью 100×100 памяти 16 Мбайт, которая не всегда может быть доступна. Поэтому используются методы со степенью резервирования $\beta < n$, как в описанном выше варианте. Тогда $t_{вн\ ш} = f(\beta)$.

Для уменьшения резервирования данных часто используют модификации метода среднего произведения.

В а р и а н т $N = n^3$. Этот вариант имеет больше теоретическое, чем прикладное, значение. В частности, он может использоваться при умножении матриц размерностью 16×16 на ЭВМ ICL DAP (размерность ПП 64×64). В этом случае все размещения матриц в ПП аналогичны размещению этих матриц в описанном выше методе внешнего произведения, но поскольку вместо одной матрицы процессоров будет 16 слоев ПМ, то все n^3 операции умножения могут быть выполнены одновременно, а затем все частные произведения суммируются за $0(\log_2 n)$ тактов. Следовательно, общее время выполнения операции умножения матриц составит $t = 0(1 + \log_2 n)$. Этот метод требует такого же избыточного резервирования данных, как и метод среднего произведения при $N = n^2$.

В а р и а н т $N < n$. Следует сказать, что это наиболее реалистичная ситуация, так как обычно N колеблется от 8...16 до 100, в то время как размер матрицы n может колебаться от 100 до 1000.

Оценим ситуацию, когда n кратно числу процессоров N. Тогда можно использовать *метод клеточного умножения матриц*. Рассмотрим для простоты умножение матриц $A \times B = C$ размерностью 4×4:

$$\begin{array}{cccc} a_{11} & a_{12} & \mathrm{M}a_{13} & a_{14} \\ a_{21} & a_{22} & \mathrm{M}a_{23} & a_{24} \\ \ldots & \ldots & \mathrm{M}\ldots & \ldots \\ a_{31} & a_{32} & \mathrm{M}a_{33} & a_{34} \\ a_{41} & a_{42} & \mathrm{M}a_{43} & a_{44} \end{array} \times \begin{array}{cccc} b_{11} & b_{12} & \mathrm{M}b_{13} & b_{14} \\ b_{21} & b_{22} & \mathrm{M}b_{23} & b_{24} \\ \ldots & \ldots & \mathrm{M}\ldots & \ldots \\ b_{31} & b_{32} & \mathrm{M}b_{33} & b_{34} \\ b_{41} & b_{42} & \mathrm{M}b_{43} & b_{44} \end{array} = \begin{array}{cccc} c_{11} & c_{12} & \mathrm{M}c_{13} & c_{14} \\ c_{21} & c_{22} & \mathrm{M}c_{23} & c_{24} \\ \ldots & \ldots & \mathrm{M}\ldots & \ldots \\ c_{31} & c_{32} & \mathrm{M}c_{33} & c_{34} \\ c_{41} & c_{42} & \mathrm{M}c_{43} & c_{44} \end{array} \quad (6.16)$$

Вычисление любого элемента матрицы C производится по известной формуле

$$C(I, J) = C(I, J) + A(I, K) * B(K, J),\ K = 1, 2, 3, 4$$

В частности, для c_{11} получаем

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} \quad (6.17)$$

Разобьем исходные матрицы на клетки (как показано в (6.16) штриховыми линиями), которые обозначим $a_{i,j}$, где i, j — индексы, указывающие положение клеток в соответствии с обычными обозначениями для матриц. Тогда выражение (6.16) для матриц A, B, C заменяется следующим выражением для матриц *A', B', C*':

$$\begin{vmatrix} a'_{11} & a'_{12} \\ a'_{21} & a'_{22} \end{vmatrix} \times \begin{vmatrix} b'_{11} & b'_{12} \\ b'_{21} & b'_{22} \end{vmatrix} = \begin{vmatrix} c'_{11} & c'_{12} \\ c'_{21} & c'_{22} \end{vmatrix} \quad (6.18)$$

Для (6.18) по обычным правилам выполнения матричных операций имеем:

$$\begin{aligned} c'_{11} &= a'_{11}b'_{11} + a'_{12}b'_{21},\ \ c'_{12} = a'_{11}b'_{12} + a'_{12}b'_{22}, \\ c'_{21} &= a'_{21}b'_{11} + a'_{22}b'_{21},\ \ c'_{22} = a'_{21}b'_{12} + a'_{22}b'_{22}. \end{aligned} \quad (6.19)$$

Выражения (6.19) позволяют вычислить все элементы исходной матрицы C на основе матриц меньшего размера, в частности для c_{11} можно записать:

$$c'_{11} = \begin{vmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{vmatrix} = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \times \begin{vmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{vmatrix} + \begin{vmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{vmatrix} \times \begin{vmatrix} b_{31} & b_{32} \\ b_{41} & b_{42} \end{vmatrix} \quad (6.20)$$

откуда, например, по обычным правилам для умножения матриц следует:

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41} \qquad (6.21)$$

Выражения (6.17) и (6.21) идентичны, что подтверждает правильность приведенных выкладок.

Таким образом, при использовании метода клеток для умножения матриц необходимо в общем случае выполнить перечисленные ниже действия:

1. Исходные матрицы большого размера, когда сторона матрицы $n>>N$, разбиваются на клетки размерностью $N\times N$. Число таких клеток в каждой матрице будет $(n/N)^2$. Эти клетки размещаются в памяти и могут в дальнейшем обрабатываться с параллелизмом N.

2. Для получения разбиения составляются системы выражений типа (6.19). Каждое выражение будет содержать n/N слагаемых. Алгоритмы вычислений по системе выражений типа (6.19) закладываются в программу пользователя или автоматически генерируются транслятором с языка ЯВУ.

3. В процессе выполнения программы в соответствии с выражениями типа (6.20) из памяти выбираются и обрабатываются с параллелизмом N клетки, т. е. матрицы значительно меньшего размера, чем исходные, что и обеспечивает эффективность обработки. При этом умножение матриц в выражении (6.20) может производиться любым подходящим методом, например, методом среднего произведения.

Для некоторых методов вычислений метод клеточных матриц не может быть использован, в подобном случае применяется размещение данного вектора в памяти процессорного поля по слоям (см. § 2.4).

Обобщая полученные результаты, сравним время умножения матриц для конвейерных ЭВМ $t_к$ и матриц процессоров $t_м$. Предположим, что для каждого типа ЭВМ используются лучшие методы (см. (6.14) и (6.15) соответственно) и время одного такта равно 10 нс для конвейерной ЭВМ и 100 нс для ПМ. Вопрос о требуемых объемах оборудования при этом не рассматривается. Соотношение времени умножения матриц

$$r = \frac{t_м}{t_к} = \frac{5 \cdot n^2 \cdot 100}{1{,}3 \cdot n^3 \cdot 10} \approx \frac{39}{n}. \qquad (6.22)$$

Из формулы (6.22) следует, что при $n \geq 39$ процессорные матрицы предпочтительнее. Необходимо еще раз подчеркнуть, что подобные расчеты носят качественный характер.

Многопроцессорные ЭВМ. В многопроцессорных ЭВМ каждый процессор — полноценный процессор последовательного типа и выполняет независимую ветвь параллельной программы. Ветвь является обычной последовательной программой, как правило, не содержащей в себе операторов разделения на другие ветви или операторов векторной обработки. Однако при выполнении параллельной программы ветви обмениваются информацией. Это и есть главный источник увеличения времени выполнения параллельной программы.

Число процессоров в многопроцессорной системе обычно невелико: $N = 2...10$, но каждый процессор обладает большой вычислительной мощностью и содержит конвейер команд или арифметический конвейер.

При разбиении задачи на ветви для исполнения на многопроцессорной ЭВМ (декомпозиция задачи) соблюдаются определенные принципы.

1. Задача должна разбиваться по возможности на наиболее крупные и редко взаимодействующие блоки. При таком подходе схема используемого для декомпозиции последовательного алгоритма рассматривается как иерархическая структура, высший уровень которой — крупные блоки, следующий уровень — подблоки крупных блоков и т. п. Процесс распараллеливания начинается с самого высокого уровня. Переход на следующий уровень происходит только в том случае, если не удалось достичь нужной степени распараллеливания на данном уровне.

В качестве блоков одного уровня обычно рассматривают циклические структуры одинаковой вложенности (пример крупноблочно-иерархического представления алгоритма дан на рис. 6.8).

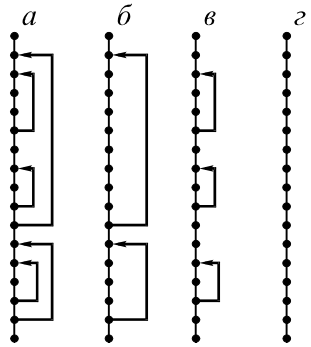

Рис. 6.8. Иерархический принцип построения алгоритма:
а — схема алгоритма; *б* — структурная единица первого уровня иерархии; *в*, *г* — структурные единицы второго уровня иерархии

Аналогично используется иерархия в структуре данных, если параллельный алгоритм строится на основе распределения по ветвям обрабатываемых данных, а не на основе операторов последовательного алгоритма.

2. Должно быть обеспечено однородное распределение массивов по ветвям параллельного алгоритма, поскольку при этом уменьшается время, затрачиваемое на взаимодействие ветвей. При таком распределении имеет место:

а) равенство объемов распределяемых частей массивов;

б) соответствие нумерации распределяемых частей массивов нумерации ветвей;

в) разделение массивов на l частей параллельными линиями или параллельными плоскостями (обычно l равно числу процессоров);

г) дублирование массивов или частей одинаковым образом.

Если рассмотреть двумерный массив, то к нему применимы следующие основные способы однородного распределения: горизонтальные полосы (ГП-распределение), циклические горизонтальные полосы, вертикальные полосы (ВП-распределение), скошенные полосы, горизонтальные и вертикальные полосы с дубли

рованием, горизонтальные полосы с частичным дублированием и т. п. Некоторые из этих распределений показаны на рис. 6.9.

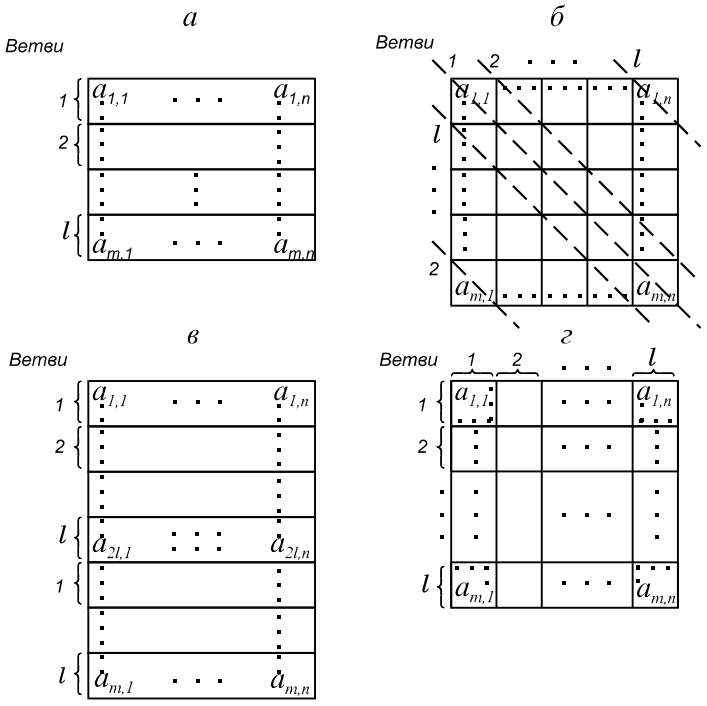

Рис. 6.9. Способы распределения двумерного массива:
а — горизонтальные полосы; *б* — скошенные полосы; *в* — циклические горизонтальные полосы; *г* — вертикальные и горизонтальные полосы с дублированием

Практикой установлено, что для многопроцессорных систем используются следующие пять основных схем обмена между ветвями [1]:

- трансляционный обмен (ТО) — “один — всем”;
- трансляционно-циклический обмен (ТЦО) — “каждый — всем”;
- коллекционный обмен (КО) — “все — одному”;
- парный стационарный обмен (ПСО) — “каждый — соседу”;

- парный нестационарный обмен (ПНО) — “каждый — любому”.

Продолжим рассмотрение задачи умножения матриц, но для многопроцессорных систем. Исходя из принципа крупноблочного разбиения, можно предположить, что наиболее подходящим для многопроцессорных систем будет метод внутреннего произведения с разбиением по внешним индексам. Разберем следующие варианты умножения матриц размерностью $n \times n$:

```
      DO 1 J = 1, n
      DO 1 I = 1, n
      DO 1 K = 1, n
1     C (I, J) = R
```

где R соответствует $\sum_{k=1}^{n} a_{i,k} b_{k,j}$. На первом уровне анализа имеется цикл по j. Его различные итерации независимы. Распределение индекса j по ветвям можно провести различным способом, в частности массивы B и C можно распределить ВП-способом. Дублирование матрицы A в каждой ветви обеспечивает ей возможность параллельных вычислений всех элементов матрицы C без обмена с другими ветвями, а ГП-распределение массива A и ведение обмена между ветвями позволяет избежать расхода памяти на дублирование. Схема вычислений приобретает следующий вид (для одного процессора):

```
      DO 1 J = 1, n/1
      DO 1 I = 1, n
      DO 2 K = 1, n
2     C(I, J) = R
1     O(I)
```

Здесь O(I) — оператор обмена, организующий рассылку строки i матрицы A во все ветви. Одна рассылка обеспечивает параллельное вычисление l элементов матрицы C. Долю обменов можно уменьшить, если поменять местами циклы по J и I:

```
      DO 1 I = 1, n
      DO 2 J = 1, n/1
```

```
      DO 2 K = 1,n
2     C(I, J) = R
1     O(I)
```

При такой схеме одна рассылка строки обеспечивает вычисление *n* элементов матрицы *C*. Тогда рассылка осуществляется по трансляционной схеме обмена (ТО).

Максимальное число ветвей, а значит, и параллелизм равны числу повторений тела цикла. Большая степень параллелизма возможна при переходе к следующему уровню, блоки которого определяются структурой отдельного “витка”. Для матриц — это переход к среднему, а затем и к внутреннему циклу.

Задача умножения матриц носит явно выраженный векторный характер и для ее решения более предпочтительным являются векторные ЭВМ (конвейерные ЭВМ и ПМ). Для многопроцессорных систем в настоящем параграфе задача умножения матриц использовалась только из-за простоты оценки эффективности различных вариантов вычислений.

Многопроцессорные системы более эффективны для задач с неоднородными по составу операторов ветвями. Эта неоднородность отражает природу какого-либо явления и может быть известна заранее. Тогда в параллельной программе для запуска неидентичных ветвей используется оператор FORK. Неоднородность может проявляться и в процессе счета, в таком случае она отображается в программе с помощью операторв IF. Подобная ситуация встречается, например, при имитационном моделировании цифровых схем.

§ 6.4. Методы параллельной обработки нечисловой информации

Обработка нечисловой информации приобретает все большее значение в связи с созданием систем автоматизации и интеллектуализации труда в производственной и научной сферах.

В этом параграфе будут рассмотрены два примера:

1) построение распределенной базы данных реляционного типа на ПМ;

2) реализация некоторых алгоритмов автоматического программирования для машин с управлением от потока данных.

Распределенные базы данных. Основной операцией в БД является *транзакция* — получение ответа на запрос, представленный в определенной форме. В современных БД число запросов достигает нескольких сотен в секунду, и на каждый запрос требуется выполнить около миллиона арифметико-логических операций. Следовательно, ЭВМ для обработки БД должны иметь быстродействие в десятки и сотни миллионов операций в секунду, а объем памяти может достигать нескольких гигабайт.

Структура современных ЭВМ для обработки БД, как правило, строится по принципу процессорной матрицы. Рассмотрим структуру, функционирование и программирование одной из таких ЭВМ, сходной по системе команд с известным процессором для реляционных баз данных *RAP* [1].

Существуют сетевые, иерархические и реляционные БД. Последние наиболее перспективны. Ознакомимся с некоторыми понятиями *реляционной алгебры*.

Основным понятием реляционной алгебры является *отношение* (рис. 6.10). Отношение содержит ряд столбцов, называемых доменами. Каждый домен является набором некоторых величин и имеет свое имя. Например, на рис. 6.10 домен Γ_1 может означать номера учебных групп, домен C_2 — номера зачетных книжек студентов, чьи фамилии находятся в домене A_3.

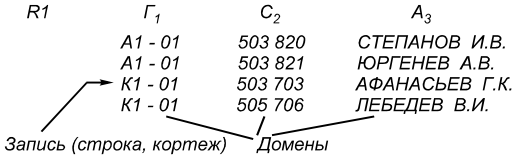

Рис. 6.10. Структура отношения:
$R1$ — имя отношения; Γ_1, C_2, A_3 — имена доменов

База данных есть совокупность отношений, связанных общими доменами и изменяющихся во времени. Отношения имеют различную размерность и обрабатываются для извлечения неявно заданной информации. Одинаковых строк в отношении не может быть.

Система обработки данных в БД называется системой управления базой данных (СУБД).

Над отношениями могут выполняться различные операции.

Пусть имеются два отношения R и S:

$R\,(A, B)$	$S\,(C, D)$
$a_1 b_1$	$c_1 d_1$
$a_2 b_2$	$a_1 b_1$

Тогда возможны следующие основные операции над отношениями R и S.

1. Объединение

$R \cup S(M, N)$
$a_1 b_1$
$a_2 b_2$
$c_1 d_1$

2. Пересечение

$R \cap S(M, N)$
$a_1 b_1$

3. Разность

$R / S(M,N)$
$a_2 b_2$

4. Произведение

$R(A, B) \otimes S(C) = R \otimes S(A, B, C)$

$a_1 b_1$	c_1	$a_1 b_1 c_1$
$a_2 b_2$	c_2	$a_1 b_1 c_2$
		$a_2 b_2 c_1$
		$a_2 b_2 c_2$

5. Проекция (предназначена для изменения числа столбцов в отношении, например, для исключения чего-либо). Пусть дано отношение

$R(D_1,$	D_2	$D_3)$
a	2	f

b	1	*g*
c	3	*f*
d	3	*g*
e	2	*f*

Тогда пять его проекций будут таковы:

$R[1]M_1$	$R[2]M_1$	$R[3]M_1$
a	2	*f*
b	1	*g*
c	3	
d		
e		

$R[3, 2]M_1, M_2$		$R[3, 2, 2]M_1, M_2, M_3$		
f	2	*f*	2	2
g	1	*g*	1	1
f	3	*f*	3	3
g	3	*g*	3	3

6. Соединение (по условиям =, ≠, <, ≤, >, ≥). Пусть даны

$R(A, B, C)$		
a	2	1
b	1	2
c	2	5
c	2	3

$S(D, E)$	
2	*f*
3	*g*
4	*f*

Тогда возможны следующие результаты:

$R\ [C = D]\ S(A, B, C, D, E)$

b	1	2	2	*f*
c	5	3	3	*g*

$R\ [B \neq C]\ S(A, B, C)$

a	2	1
b	1	2
c	2	5
c	5	3

$R\ [C < D]\ S(A, B, C, D, E)$

a	2	1	2	*f*
a	2	1	3	*g*
a	2	1	4	*f*
b	1	2	4	*g*
b	1	2	4	*f*
c	5	3	4	*f*

$R\ [B > C]\ S(A, B, C)$

a	2	1
c	5	3

Эти операции реляционной алгебры в том или ином виде входят в системы команд различных реляционных процессоров баз данных.

Структура параллельного реляционного процессора (ПРП), аналогичного RAP, совпадает с общей структурой процессорной матрицы.

В ответ на запрос на специальном языке запросов ПРП выполняет следующие действия: преобразует запрос во внутреннюю форму; генерирует программу поиска ответа на запрос; ищет ответ; преобразует ответ во внешнюю форму; выдает ответ. Причем собственно ПРП ищет только ответ, остальные действия осуществляет базовая машина, формирующая программу поиска, которая реализуется ЦУУ и всеми ПЭ. Коммутатор необходим для сравнения результатов поиска в отдельных ПЭ.

В качестве ОП обычно используется сдвиговая память на приборах с зарядовой связью. К памяти возможен только последовательный доступ. Но с развитием методов поиска описаний отношений будет применяться и память с произвольным доступом, тогда роль коммутаторов возрастет.

Все ПЭ выполняют одну и ту же команду поиска, при этом каждый ПЭ*i* хранит имя отношения. По мере последовательного считывания записей из ОП*i* запись анализируется, и, если необходимо, заменяется новой, которая записывается в память. В каждом АЛУ имеется местная память для хранения доменов на время их обработки.

Отношение располагается в ОП*i* в форме последовательности кортежей (рис. 6.11). Поле *DF* используется для размещения флага удаления. Если *DF* = 1, то данный кортеж будет стерт, а программа “сбора мусора” учтет это место в памяти как свободное.

Кортеж считается *T*-маркированным, если в разрядах *ABCD* содержится определенная совокупность нулей и единиц, называемая *T*-образцом. Поле каждой величины имеет разрядность 8, 16 или 32, причем первые два бита задают его длину. Кортеж содержит не более 255 доменов. Последний кортеж в первом домене содержит комбинацию 11, которая задает конец записи. При наращивании записи эта комбинация перемещается к ее концу, при сокращении — к началу.

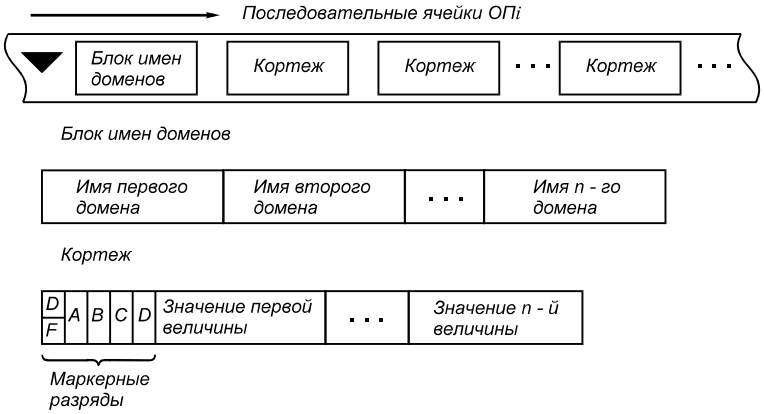

Рис. 6.11. Структура хранения отношения в памяти

Программа ПРП рассматривается как канальная программа БМ, только семантика этих команд другая. Команда имеет формат:

метка><КОП><маркеры>[<объект>:<условие>] [<параметр>]

Поле <метка> не обязательно. Оно используется только при переходах в программе. В поле < маркеры > указываются разряды, которые нужно установить, например, *MARK* (*A*) означает, что нужно разряд *A* установить в 1 во всех кортежах, удовлетворяющих заданному условию. Если задана операция *RESET* (*AB*), то надо сбросить разряды *A*, *B* во всех кортежах при выполнении условия.

Поле <объект> имеет вид: <Rn>(<D1>, <D2>, ..., <Dn>), где *Rn* — имя отношения; *Di* — имя домена отношения.

Логическое выражение может иметь форму: вырожденную; конъюнкции элементарных условий; дизъюнкции элементарных условий. Элементарное условие можно задавать в таком виде:

1) <имя> <операция сравнения> <операнд>

2) <*Rn*>. *MKED* (<*T*>)

3) <*Rn*>. *UNMKED* (<*T*>)

В поле <имя> задается выражение (<*Rn*>). <*Dn*>). В поле <операция сравнения> указываются следующие операции: =, ≠, <,

≤, >, ≥. В поле операнда может быть указан адрес операнда в БМ или ПЭ. Приведем пример условия:

СОТР. ЗАРП > Q & (СОТР. MKED(A))

Здесь выделяются кортежи отношения СОТР со значением зарплаты больше *Q*, в которых маркер *A* установлен в 1.

Для дальнейшего использования примем следующие сокращения:

СОТР — сотрудник,	УПР — управляющий,
ОТД — отдел,	НОМИЗ — шифр изделия,
ЗАРП — зарплата,	ФАМ — фамилия,
КОЛ — количество,	ИМИЗ — имя изделия.

Рассмотрим систему команд ПРП (всего 11 команд) на примере обработки БД (рис. 6.12):

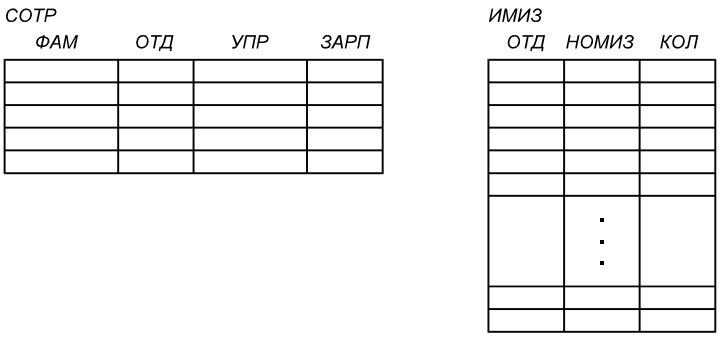

Рис. 6.12. Пример БД

1) *SELECT* <МАРКЕРЫ>[<*Rn*>: <УСЛОВИЕ>]

Пример:

S*ELECT MARK* (*AB*) [СОТР:СОТР.ОТД = 202]

В те кортежи, в которых имеются сотрудники из отдела 202, в *A* и *B* записывается 1.

2) *READ-ALL*<МАРКЕРЫ>[<*Rn*>(<*D*1>, ...): <УСЛОВИЕ>] [<АДРЕС>]

Пример 1:

READ-ALL [СОТР:УПР = КУЗНЕЦОВ] [*BUF*].

В буфер *BUF* посылаются данные обо всех сотрудниках отдела, возглавляемого Кузнецовым.

Пример 2:

READ-ALL [СОТР (ФАМ, ЗАРП): СОТР. *MKED*(*A*)] [*BUF*].

В буфер *BUF* записываются фамилии и зарплата отмеченных сотрудников.

3) *READ*[<*n*>]<МАРКЕРЫ>[<*Rn*>(<*Di*>,...): <УСЛОВИЕ>] [<АДРЕС>] — соответствует команде READ-ALL, но в буфер BUF посылается только *n* кортежей;

4) *READ-REG* [<список регистров>] [<адрес>] — записывает содержимое указанных регистров в БМ;

5) *CROSS-SELECT*<R1 МАРКЕРЫ>[<*R*1>:<*D*1> <ОПЕРАЦИЯ СРАВНЕНИЯ><*R*2>.<*D*2>] [<*R*2>. MKED (<T>)] — сложная команда. По ней устанавливаются маркеры в *R*1, если домен *D*1 из *R*1 сравнился с <*R*2>.<*D*2>, при этом рассматриваются только *T*-маркированные кортежи *R*2.

Пример:

SELECT MARK (A) [ИМИЗ. КОЛ > 100] CROSS SELECT MARK (A) [СОТР: СОТР. ОТД = ИМИЗ. ОТД [ИМИЗ. MKED (A)].

Будут промаркированы записи о служащих, которые продали более 100 единиц продукции.

6) *GET-FIRST-MARK* <МАРКЕРЫ> [<*R*n> (<*D*1>, ...): <*Da*>) <ОПЕРАЦИЯ СРАВНЕНИЯ> <*Db*>] [MKED(<*T*>] — выполняет операцию проектирования. Значение домена <*Db*> некоторого *T*-маркированного кортежа последовательно сравнивается со значением домена <*Da*> всех кортежей данного отношения. При совпадении устанавливаются маркеры кортежей, в которых произошло

совпадение; сбрасывается *T*-маркер опорного кортежа; значения доментов (<*D*1>, ...) записываются в память АЛУ;

7) *SAVE*(<*n*>)[<*Rn*>(<*D*1>, ...): <УСЛОВИЕ>] [<СПИСОК РЕГИСТРОВ>] — в регистры записываются для *n* кортежей домены, удовлетворяющие условию;

8) *SUM*<МАРКЕРЫ>[<*Rn*>(<*Dn*>):<УСЛОВИЕ>] [<РЕГИСТР>]

COUNT <МАРКЕРЫ> [<*Rn*>(<*Dn*>): <УСЛОВИЕ>] [<РЕГИСТР>]

где *SUM* суммирует значения домена; *COUNT* подсчитывает число доменов;

9) *DELETE* [<*Rn*>:<УСЛОВИЕ>] — исключает все кортежи, удовлетворяющие условию;

10) ВС<МЕТКА>,<УСЛОВИЕ ПЕРЕХОДА> — обычный условный переход;

11) *EOQ* — завершает выполнение программы.

Рассмотрим три примера программ для ПРП. Первая программа исключает из отношения СОТР записи обо всех сотрудниках отдела Кузнецова:

```
SAVE [СОТР(УПР) : СОТР.ФАМ = КУЗНЕЦОВ] [REG 0]
DELETE [СОТР : СОТР. УПР = REG 0]
```

Вторая программа подсчитывает количество видов товарной продукции и общий объем товаров, проданных отделом, в котором служит Петров:

```
1. SELECT          MARK(A)[СОТР:СОТР.ФАМ = 'ПЕТРОВ':]
2. CROSS SELECT    MARK(A)[ИМИЗ:ОТД=СОТР.ОТД][СОТР]:
                   : MKED(A)]
3. COUNT           [ИМИЗ:MKED(A)][REG 1]
4. SUM             [ИМИЗ(КОЛ):MKED(A)][REG 2]
5. READ-REG        [REG 1, REG 2] [BUF]
6. SELECT          RESET (A) [ИМИЗ]
7. SELECT          RESET (A) [СОТР]
8. EOQ
```

По команде 1 для служащего Петрова маркируется кортеж СОТР.

По команде 2 маркируются кортежи ИМИЗ, относящиеся к отделу, в котором работает Петров. По командам 3,4 подсчитыва

ется количество видов маркированной товарной продукции и общий объем последней, результаты заносятся в регистры 1, 2. По команде 5 эти данные переписываются в область *BUF* базовой машины. Команды 6 и 7 устанавливают в 0 маркеры.

В третьей программе выполняются операции проектирования и условного перехода. Допустим нужно подсчитать число отделов, в которых заработок сотрудников больше *Q* руб. Используем такую программу:

```
1. SELECT                 MARK(AB)[COTP:COTP.ЗАРП > Q]
2. LOOP-GET-FIRST-MARK    MARK(C)[COTP:COTP.ОТД =
                          COTP.ОТД] [MKED(B)]
3. SELECT                 RESET(ABC)[COTP.COTP.MKED(C)]
4. BC                     A, TEST(COTP:MKED(B))
5. COUNT                  [COTP:COTP.MKED(A)][REG 1]
6. EOQ
```

Для конкретизации первая команда *SELECT* “маркирует”, например, Кузнецова и Вашкевича из четырнадцатого, Вершинину, Афанасьева и Лебедева из девятнадцатого и Ушакова из двадцатого отделов. При первом выполнении второй команды *GET-FIRST-MARK* будет найден один из пяти кортежей (например, Кузнецов). В результате станут *C*-маркированными все остальные *B*-маркированные кортежи того же отдела, при этом *B*-маркер Кузнецова будет сброшен. Третья команда *SELECT* сбросит *ABC* во всех *C*-маркированных кортежах (относящихся к Вашкевичу). При выполнении четвертой команды *BC* будут обнаружены *B*-маркированные кортежи. Затем все повторяется.

Пусть теперь вторая команда *GET-FIRST-MARK* найдет Афанасьева, выполнит *C*-маркирование кортежа Вершининой и Лебедева, сбросит *B*-маркеры кортежа Афанасьева. Третья команда *SELECT* сбросит *ABC* кортежей Вершининой и Лебедева. Опять следует повторение. Вторая команда *GET-FIRST-MARK* установит в 0 *B*-маркер Ушакова, однако кортежей, нуждающихся в маркировании, больше не обнаружит. Третья команда *SELECT* не найдет кортежей, удовлетворяющих условию. Четвертая команда *BC* не обнаружит *B*-маркированных кортежей. Тогда выполняется пятая команда *COUNT* и выдается ответ: 3 (Кузнецов, Афанасьев, Ушаков).

Автоматическая генерация программ. В ЭВМ пятого поколения предусматривается автоматическая генерация программ вычислений на основе заданного человеком условия задачи. Уже разработаны и экспериментально проверены некоторые из таких методов.

Большую известность получил *метод вычислительных моделей* [11], предназначенный для задач с четко определенными отношениями. Система на основе вычислительных моделей содержит базу знаний, предметную и операционную области. В базе знаний находятся наиболее общие понятия и правила вывода, обеспечивающие логику работы для предметных областей всех видов.

В предметной области содержатся знания, относящиеся к конкретной области научной или инженерной деятельности. Область операций хранит условия задачи, начальные данные, копию нужных разделов предметной области и т.д.

Рассмотрим пример. Пусть предметная область есть модель прямолинейного движения: тело массой m движется прямолинейно с ускорением a в интервале $t_1 \ldots t_2$ с начальной скоростью v_1 и конечной скоростью v_2; v— средняя скорость; Δs — пройденный путь; f— сила инерции; ΔE— выполненная силой f работа; E_1 и E_2 — кинетическая энергия в начале и конце движения соответственно. Физические уравнения для этой предметной области таковы:

$$\begin{array}{lll} f = ma, & \Delta v = v_2 - v_1, & E_2 = mv_2^2/2, \\ \Delta E = f\Delta s, & E_1 = mv_1^2/2, & 2v = v_1 + v_2 \\ \Delta t = t_2 - t_1, & \Delta s = v\Delta t, & \\ \Delta v = a\Delta t, & \Delta E = E_2 - E_1, & \end{array} \qquad (6.23)$$

Предметная область для прямолинейного движения в графической форме представлена на рис. 6.13. Графическая модель наглядна, но в памяти ЭВМ она должна представляться в форме текста (6.23).

Задание на вычисления в этой предметной области формируются двояко:

1) “Зная ДВИЖЕНИЕ вычислить $v_1, \Delta E$ по m, Δt, a, v_2”;

2) “Зная ДВИЖЕНИЕ составить программу вычисления $v, \Delta E$

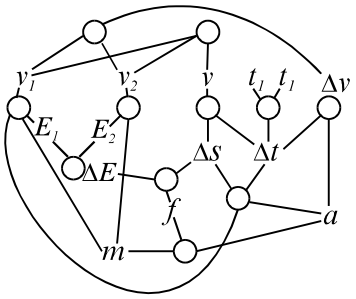

Рис. 6.13. Модель предметной области для прямолинейного движения. Кружками обозначены отношения между переменными

по $m, \Delta t, a, v_2$.”

В первом случае генерируется программа, по ней производятся вычисления и пользователь получает готовый результат. Во втором случае пользователь получает только программу вычислений.

Функционирование описанной выше вычислительной модели напоминает поиск одного или нескольких кратчайших путей в графе (см. рис. 6.13):

1) оператор вычислительной модели (отношение) может быть вычислен, если его входы определены;

2) оператор стоит применять только тогда, когда он дает новые результаты, которые ранее не вычислялись;

3) если есть различные пути вычисления результата, то с точки зрения правильности его получения не имеет значения, какой путь выбрать (но не с точки зрения длины программы).

Можно, например, получить две следующие программы (они являются различными путями в предметной области):

1. $\Delta v = a\Delta t$	1. $\Delta v = a\Delta t$
2. $f = m / a$	2. $v_1 = v_2 - \Delta v$

3. $v_1 = v_2 - \Delta v$	3. $E_1 = mv_1^2 / 2$
4. $v = v_1 + v_2 / 2$	4. $E_2 = mv_2^2 / 2$
5. $E_1 = mv_1^2 / 2$	5. $\Delta E = E_2 - E_1$
6. $E = mv_2^2 / 2$	
7. $\Delta E = E_2 - E_1$	

Существуют специальные приемы выбора более коротких (по некоторым критериям) программ (“прополки” различного рода). Принцип функционирования такой вычислительной модели строго соответствует принципу функционирования описанных в § 3.3 ЭВМ с управлением от потока данных. В обоих случаях срабатывание *A* оператора происходит после того, как на все его входы поступили данные. Поэтому, используя систему обозначений § 3.3, каждый оператор (отношение) вычислительной модели необходимо представить в виде команды для DF-ЭВМ. Само же графическое описание вычислительной модели, по существу, соответствует описанию на графическом языке Денниса.

Функционирование вычислительной модели можно обеспечить и на многопроцессорных ЭВМ (типа МКМД). Но и в этом случае необходимо реализовать запуск операторов после появления данных, т.е. опять же обеспечить управление от потока данных.

Однако поскольку в МКМД-ЭВМ отсутствуют необходимые аппаратные средства, такие машины будет менее эффективны, чем ЭВМ с управлением от потока данных, в которых этот вид управления на всех уровнях поддерживается аппаратурой.

Контрольные вопросы

1. Перечислите основные характеристики параллельных алгоритмов.
2. Охарактеризуйте основные свойства рекурсивных алгоритмов.
3. Перечислите основные способы умножения матриц на параллельных ЭВМ.
4. В чем заключаются особенности алгоритмов вычисления ДУЧП на параллельных ЭВМ?
5. В чем заключаются особенности умножения матриц на конвейерных ЭВМ, процессорных матрицах и многопроцессорных ЭВМ?

6. Как выполняются операции нечисловой обработки в параллельных ЭВМ?
7. Что такое автоматическая генерация программ?

ЗАКЛЮЧЕНИЕ

Прогноз роста характеристик микропроцессоров на ближайшие годы указывает на следующие тенденции [24]:

Характеристика	Темп удвоения характеристики	Состояние на 1995 год	Перспектива на 2000 год
Частота синхронизации	3,6 года	10 МГц	300 Мгц
Шаг сетки кристалла	7 лет	0,65 мк	0,4 мк
Размер кристалла	5 лет	450 мм^2	900 мм^2
Число транзисторов на кристалле (процессор)	2 года	6 млн	50-100 млн
Разрядность шин данных	5 лет	160	-

Таким образом, к 2000 году за счет параллелизма и повышения тактовой частоты возможно получение быстродействия триллионы оп/с. Это позволяет перейти к новому способу оценки быстродействия ЭВМ, определяя его как задержку (время реакции) при решении крупных задач.

Высокие характеристики микропроцессоров во многом решают проблему аппаратуры, но выдвигают ряд новых проблем в области архитектуры и программирования. Не касаясь проблем, общих для всей вычислительной техники, перечислим некоторые проблемы, связанные непосредственно с параллелизмом:

1. Исследования численных методов показывают (см. гл. 6), что в них имеется достаточно большой уровень параллелизма.

С другой стороны, характеристики микропроцессоров таковы, что представляется возможным строить системы с сотнями и тысячами процессоров. Таким образом, развитие параллельных ЭВМ обосновано как со стороны алгоритмов, так и со стороны аппаратуры. Однако большое разнообразие параллельных архитектур приводит к распылению сил в разработке параллельной техники. Это означает, что вопросы унификации параллельных архитектур становятся центральными. Одной из наиболее перспективных является архитектура МКМД-ЭВМ на базе *n*-кубов, в узлах которой используются векторные и суперскалярные процессоры.

2. Параллельное программирование является слишком сложным для массового пользователя, поэтому вопросы автоматизации распараллеливания, планирования и отладки пользовательских программ становятся ведущими в разработке программного обеспечения параллельных ЭВМ.

3. Выживаемость любой новой архитектуры в первую очередь определяется возможностью быстрого создания большого объема прикладного программного обеспечения, поэтому разработка унифицированных пакетов и средств автоматизации переноса прикладных пакетов приобретает большое значение.

Все упомянутые выше проблемы в той или иной мере нашли отражение в настоящем издании.