Глава 5

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ПАРАЛЛЕЛЬНЫХ ЭВМ

§ 5.1. Особенности программного обеспечения параллельных ЭВМ

Параллелизм оказывает существенное влияние на архитектуру вычислительных средств.

Под вычислительными средствами будем понимать аппаратуру ЭВМ и ее программное обеспечение. Влияние параллелизма и прежде всего системы команд на аппаратуру было описано в предыдущих главах, в этой главе будет рассмотрено влияние параллелизма на программное обеспечение.

Программное обеспечение традиционной универсальной ЭВМ можно разделить на три части, часто называемые системным программным обеспечением: операционные системы, системы программирования и прикладные пакеты.

Ядро операционной системы (ОС) мультипрограммной ЭВМ содержит следующие компоненты:

1. Средства управления ресурсами системы (драйверы ввода-вывода, динамическое управление памятью, система прерывания процессора и др.).
2. Средства управления процессами (создание, уничтожение, синхронизация процессов).
3. Средства планирования и распределения ресурсов между задачами, заданиями, процессами.
4. Системы управления файлами.
5. Интерфейс пользователя, оператора (управляющие средства, командные языки).
6. Средства восстановления и реконфигурации ЭВМ и ОС после появления сбоев и неисправностей.

В зависимости от режима работы мультипрограммная ОС приобретает специализацию. В частности, для пакетного режима усиливаются функции планирования, оптимизируется распределение времени между пользователями в системах разделения времени, совершенствуются средства прерывания в системах реального времени.

Системы программирования включают языки программирования, компиляторы, отладчики, средства объединения и загрузки программ, редакторы, библиотеки подпрограмм и функций разных этапов выполнения программы. Интегрированные системы программирования обычно включают также собственную систему управления файлами, интерфейс программиста и некоторые другие функции, характерные для ОС.

Прикладные пакеты разрабатываются и поставляются как фирмами, которые разрабатывают оборудование, так и фирмами-разработчиками программного обеспечения. Существует большое количество пакетов различного назначения, например, пакеты для автоматизации проектирования, текстовые редакторы, графические пакеты, пакеты научной математики, базы данных, пакеты для цифровой обработки сигналов, для обработки экономической информации и др. Эти пакеты, как правило, имеют хорошую научную основу, имеют большой объем программ и сложную организацию. В них используются специальные языки описания предметной области, интерфейс пользователя и некоторые элементы управления, свойственные операционным системам.

Параллелизм в основном затрагивает систему программирования, причем, в наибольшей степени это проявляется при программировании одной задачи, поскольку основное назначение параллельных систем — уменьшение времени выполнения отдельных трудоемких задач. Поэтому, если в последовательной ЭВМ процесс программирования можно, образно говоря, считать одномерным, так как в этом случае достаточно только обрабатывать последовательность арифметико-логических операций, то в параллельной ЭВМ необходимо еще отслеживать размещение процессов в пространстве ресурсов и их синхронизацию во времени, то есть здесь процесс программирования является, по крайней мере, трехмерным.

На систему программирования ложатся функции создания языков программирования и трансляторов, которые во многих случаях должны обеспечивать функции автоматического распараллеливания последовательных программ (программ, а не алгоритмов, так как распараллеливание алгоритмов относится к области параллельной вычислительной математики и рассматривается в главе 6).

При появлении параллельных ЭВМ приходится перерабатывать ранее написанные прикладные пакеты, однако вопросы их мобильности в настоящем пособии не рассматриваются.

Таким образом глава 5 в основном посвящена системам программирования.

§ 5. 2. Векторные языки

Наиболее важным и трудоемким для векторных ЭВМ является создание языков и трансляторов.

На рис. 5.1 показана зависимость быстродействия конвейерной ЭВМ от размера L матрицы при выполнении одного и того же алгоритма (отыскание LU-разложения), но записанного на ассемблере (кривая 1), машинно-ориентированном параллельном ЯВУ (кривая 2), проблемно-ориентированном параллельном ЯВУ (кривая 3), последовательном Фортране с подпрограммами на параллельном ассемблере (кривая 4), последовательном Фортране с векторизацией (кривая 5), на Фортране без векторизации (кривая 6). Несмотря на очень большой разрыв в производительности, обеспечиваемой различными языками, на практике по тем или иным причинам используются все названные языки.

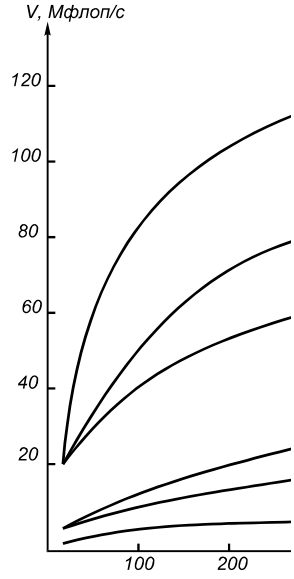


Рис.5.1. Производительность конвейерной Э
ричной операции по программам, написан

Языки типа ассемблер в символической форме отражают машинную систему команд параллельной ЭВМ, т. е. все особенности структуры ЭВМ: число процессоров, состав и распределение регистров и ОП, систему коммутации, особенности структуры ПЭ, средства специализированного размещения данных и т. д. Языки такого типа используются на всех параллельных ЭВМ и позволяют детально приспособить па

раллельный алгоритм к особенностям структуры конкретной параллельной ЭВМ. Программы, написанные на ассемблере, обеспечивают наивысшую производительность ЭВМ.

Ассемблер, несмотря на высокую эффективность программ, написанных на этом языке, не может использоваться для массового прикладного программирования по следующим причинам:

1) программирование на ассемблере является очень трудоемким даже для последовательных ЭВМ, не говоря уже о параллельных, где необходимо дополнительно программировать размещение данных и синхронизацию процессов;

2) в вычислительной технике происходит регулярная смена элементной базы, что сказывается на системе команд ЭВМ, в особенности параллельных (это означает переработку всего запаса программ, написанных на ассемблере);

3) программы, написанные на ассемблере, не мобильны, т. е. их нельзя выполнять на параллельных ЭВМ других классов, а часто и внутри одного класса ЭВМ, но для разных размеров ПП.

Поэтому, как правило, программы на ассемблере используются:

1) при создании системного математического обеспечения;

2) для написания программных библиотек стандартных функций и численных методов;

3) для оформления гнезд циклов с наибольшим временем счета в программах, написанных на ЯВУ;

4) для написания небольшого числа программ постоянного пользования в специализированных ЭВМ.

Параллельные ЯВУ в большинстве случаев строятся как расширение стандартных вариантов Фортрана, хотя в некоторых случаях используются расширения на базе языков Паскаль, Ада, Си. Иногда параллельный язык строится как новый, не имея в качестве базы стандартного последовательного языка.

Проблемно-ориентированные языки не имеют структур данных и операций, отражающих свойства конкретной параллельной ЭВМ. Если ЯВУ является векторным, то написанные на нем программы должны с равной эффективностью выполняться как на конвейерных ЭВМ, так и на ПМ. В проблемно-ориентированных языках все конструкции данных и операции над ними соответствуют математическому понятию этих данных и операций. Размер

ности данных, как правило, не ограничены, в таких языках часто отсутствуют средства для размещения пользователем данных в ПП. Проблемно-ориентированные языки достаточно просты для программирования.

Машинно-ориентированные языки в сравнении с проблемно-ориентированными языками позволяют создавать значительно более эффективные программы, предоставляя пользователю средства, отражающие структуру параллельной ЭВМ.

В машинно-ориентированных языках часто вводятся:

1) переменные типа “суперслово”. Число элементарных слов в суперслове строго соответствует размеру ПП (64 для ЭВМ ILLIAC-IV). Пользователь имеет возможность представить данные в виде суперслова, обеспечивая, таким образом, лучшее использование ОП и увеличенное быстродействие;

2) операторы для размещения в ПЭ данных по “срезам” различных индексов массивов;

3) средства типа “масок” для включения-выключения отдельных ПЭ.

Написание программ на машинно-ориентированных ЯВУ является более трудоемким процессом, чем написание программ на проблемно-ориентированных ЯВУ.

Для специализированных применений ЭВМ часто используются системы программирования, содержащие библиотеку программ ограниченного размера, написанную на ассемблере. Подпрограммы библиотеки вызываются из основной программы, оформленной на последовательном Фортране или другом ЯВУ и предназначенной для выполнения в базовой ЭВМ. Этот метод использования библиотечных программ обладает средними характеристиками по трудоемкости, эффективности и универсальности.

При разработке параллельного ЯВУ необходимо:

- выбрать новые структуры данных и определить методы доступа к структурам и элементам этих структур данных;
- определить допустимые арифметико-логические и управляющие операции над введенными структурами данных;
- определить базовые методы размещения и коммутации данных в ПП;

- предложить изобразительные средства для оформления структур данных и операций над ними.

Рассмотрим наиболее общие методы решения этих вопросов в параллельных ЯВУ [2].

Основными объектами параллельной обработки в параллельных ЭВМ являются массивы различной размерности, а также подструктуры этих массивов: плоскости строки, столбцы диагонали и отдельные элементы. Следовательно, для выборки данных объектов в параллельном ЯВУ должны существовать специальные механизмы.

Выборка объектов пониженного ранга. Элементы массива разного ранга выбираются посредством индексации одной или нескольких размерностей массива. В частности, если задана матрица $A(N, N)$, то:

- обозначение A относится ко всей матрице;
- $A(I)$ относится ко всей I-й строке матрицы A (вектор);
- $A(J)$ относится к J-му столбцу матрицы A (вектор);
- $A(I, J)$ относится к одному элементу матрицы A (скаляр).

Наибольший интерес в языках представляет процесс получения векторов (одномерных массивов), так как именно они являются основным объектом обработки в векторных ЭВМ.

В некоторых языках синтаксис механизма задания векторов определяется специальным символом ∗, который устанавливается в позиции опущенного индекса. Так, для приведенного выше примера получаем: $A(*, *)$, $A(I, *)$, $A(*, J)$, $A(I, J)$. Однако часто соответствующий индекс просто опускается.

Примеры подобных выборок даны на рис. 5.2.

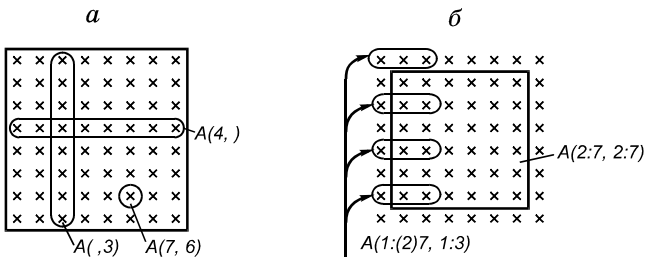


Рис. 5.2. Пример выборки, понижающей ранг (а)
и сужающей диапазон (б), из массива 8×8

Выборка диапазона значений. Выборка диапазона значений производится для матрицы $A(N\times N)$ с помощью конструкций вида $A(2 : N\text{-}1,\ 2 : N\text{-}1)$ или $A(1 : (2)N,\ 1 : 3)$. В обоих примерах используемый синтаксис указывает начало, шаг (:), если он есть, и конечную границу диапазона изменения данного индекса.

Поэтому первое выражение позволяет осуществить выборку внутренних точек матрицы, а второе — выбрать первые три элемента из нечетных строк (см. рис. 5.2).

Выборка с помощью целочисленных массивов. Применение этого метода предоставляет большие возможности для выделения подструктур. Пусть даны масссивы целых чисел *IV*, *JV*. Размерность этих массивов может быть произвольной, но числа в них должны быть меньше или равны *N*, так как они являются индексами элементов массива *A*.

Возьмем для примера следующие исходные массивы и векторы:

$A \rightarrow$	*a*	*b*	*c*	*d*		*IV*	1	1	1	3
	e	*f*	*g*	*h*						
	i	*j*	*k*	*l*						
	m	*n*	*o*	*p*		*JV*	3	2	1	

Тогда возможны следующие выражения:

а) $A(I, JV(I)) = A(3, JV(3)) \Rightarrow i$
б) $A(*, IV(*)) \Rightarrow a\ e\ i\ o$
в) $A(IV(*), *) \Rightarrow a\ b\ c\ l$

Выражение а) позволяет выбрать единичный элемент матрицы *A*, если задан номер строки *I*. Этот элемент имеет индексы $A(I, K)$, где *K* — число, записанное на *I*-м месте массива *JV*. Выражения б) и в) позволяют из матрицы *A* извлечь проекцию в виде вектора. Каждый элемент вектора получается аналогично случаю а), но для всех индексов данной размерности.

Выборка с помощью элементов булевых массивов. Выборка осуществляется в зависимости от состояния логического селектора, который может являться элементом логического массива или

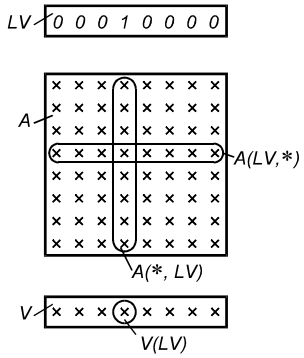


Рис. 5.3. Пример выборки булева вектора *LV*

результатом вычисления логического выражения.

Несколько примеров такой выборки дано на рис. 5.3.

Индексация сдвига. Эта процедура выравнивает положение массивов относительно друг друга с помощью индексных операций и часто используется при решении дифференциальных уравнений в частных производных сеточным методом. Здесь каждый узел сетки корректируется по среднему значению от четырех соседних узлов, что можно выразить следующим образом:

$$A=(A(*+1, *) + A(* - 1, *) + A(*, * - 1) + A(*, * + 1))/4.$$

Все элементы *A* обновляются одновременно в соответствии со значением четырех соседних элементов, выбираемых путем сдвига массива *A* влево (+) и вправо (–) в каждой размерности. Для внутренних значений *A* это можно выразить последовательной программой:

```
      DO 1 I = 2, N - 1
      DO 1 J = 2, N - 1
1     ANEW(I, J) = (A(I+1, J)+A(I-1, J)+A(I, J+1)+A(I, J-1))/4.
```

Краевые эффекты, как правило, рассматриваются отдельно в последовательной программе, но это можно сделать путем сдвигов

и в параллельной программе, если принять некоторую геометрию соединений краев ПМ. (Примеры таких соединений приведены в § 2.3). Конечно, сдвиг может осуществляться в случае необходимости на любую величину путем записи выражения $*+K$, где K — величина сдвига.

Повышение ранга структуры данных. Часто для упрощения вычислений необходимо многократно повторить, размножить экземпляр скаляра или массива. Такая функция называется *XPND* (“расширитель”). Выражение *XPND* (*A*, *K*, *N*) обозначает массив, сформированный путем повторения *N* раз вектора *A* по своей *K*-й размерности. Например, если *V* — вектор из *N* элементов, то *XPND* (*V*, 1, *N*) — матрица *N*×*N*, сформированная повторением *V* в качестве строк матрицы, а *XPND* (*V*, 2, *N*) сформирована повторением *V* в качестве столбцов матрицы.

Переформирование массивов. Иногда необходимы операции между массивами различной размерности, но содержащими одинаковое общее число элементов. Для переформирования массивов используется оператор *MAP* $A(m_1,\text{K},m_q)$, который преобразует массив $A(n_1,\text{K},n_p)$ в матрицу $A(m_1,\text{K},m_q)$, где q и p — число размерностей нового и старого представления матрицы A соответственно, при этом всегда $\sum_{k=1}^{p} n_k = \sum_{k=1}^{q} m_k$.

Ниже дается пример преобразования одномерного массива *A* в двумерный:

```
DIMENSION A(N)
. . .
MAP A (N/2, 2)
. . .
```

Часто возникает и другая задача: большой массив разделяется на части, каждая из которых используется под собственным именем; при этом все данные хранятся в памяти только в области исходного массива, не повторяясь. Операция переименования производится с помощью оператора DEFINE, который записывается следующим образом:

DEFINE $a_1 \cdot e_1 = b_1,\ a_2 \cdot e_2 = b_2,\ \ldots,\ a_k \cdot e_k = b_k$,

где каждое a_i— имя массива, помещаемого на массив b_i; e_i — мерность массива; массив b_i записывается в виде $n_i(p_1,\text{K}\,,p_m)$; n_i — имя массива; p_j — выражение вида Fl, где $1 \le l \le m$.

Например, запись

```
DIMENSION B (10, 10, 3)
. . .
DEFINE A (10, 10) = B (F1, F2, 3)
. . .
```

означает, что массив $A(10, 10)$ есть часть массива B, причем $F1$ и $F2$ обозначают соответственно совпадение первой размерности массива A с первой размерностью массива B и второй размерности A — со второй размерностью B. Тогда обращение к элементу $A(4, 5)$ будет в действительности вызывать обращение к элементу $B(4, 5, 3)$.

Размещение данных. От того, каким образом размещены данные в ПП, зависит время выполнения программы. В параллельных ЯВУ размещение данных обычно задается неявно и извлекается транслятором из векторных выражений. Например, следующие выражения обозначают:

а) $A(I, *)$ — матрица A должна быть размещена так, чтобы все элементы любой ее строки находились в разных ПЭ. Это главное условие параллельного доступа к элементам строки, во всяком случае, если длина строки меньше или равна числу процессоров. Если это не так, то элементы строки располагаются слоями. Элементы столбца располагаются в одном ПЭ;

б) $A(*, J)$ — матрица A должна быть размещена параллельно по столбцам;

в) $A(*, *)$ — необходим параллельный доступ как по столбцам, так и по строкам. В этом случае транслятор строит скошенное размещение матрицы A.

Размещение массивов не всегда является очевидным, как в приведенных выше случаях, поэтому в некоторых языках имеются операторы явного задания способов размещения данных. Например, в языке *IVTRAN* (параллельное расширение Фортрана) ис

пользуются следующие символы: $ — признак основной размерности; # — признак подстрочной размерности. Если параметр основной, то увеличение его на единицу будет увеличивать на единицу номер ПЭ, адрес в модуле памяти увеличиваться не будет. Если параметр подстрочный, то увеличение его на единицу не меняет номера ПЭ, а увеличивает адрес в модуле памяти на единицу.

С учетом сказанного приведенные ниже выражения обозначают следующее:

1) [$(1), (2)] — размерность на первой позиции будет основной, хотя по обеим координатам имеется параллельный доступ к срезам. Такое размещение целесообразно, когда в программе чаще обращаются к первой и реже ко второй координате;

2) [$(2), (1)] — обращение чаще производится ко второй координате;

3) [(2), # (1)] — по первой координате параллельный доступ не нужен, что и отмечено символом #.

Арифметико-логические операции. Параллельные расширения ЯВУ используют выражения и операторы присваивания, имеющие ту же структуру, что и в последовательных языках, но операндами в них являются векторы. Приведем примеры бинарных векторных операторов:

а) $C1(*) = A1(*) + B1(*)$ — поэлементное сложение двух векторов, т. е. $C1(I) = A1(I) + B1(I)$, $I = 1, 2, ..., N$;

б) $A2(*) = K*A1(*)$ — умножение каждого элемента вектора на константу, т. е. $A2(I) = K*A1(I)$, $I = 1, 2, ..., N$;

в) $C(I, *) = A(I, *) * B(*, J)$ — поэлементное умножение векоров, полученных из матриц A и B, т. е. $C = (I, K) = = A(I, K)*B(K, J)$, где $K = 1, 2, ..., N$.

Бинарные (с двумя операндами) операции в подавляющем большинстве параллельных ЯВУ выполняются только над векторами, как в приведенных примерах. В некоторых языках эти операции могут выполняться и над массивами большей размерности. Однако при любой размерности массивов-операндов над ними выполняются только поэлементные операции. Поэтому оператор

$$C(*, *) = A(*, *) * B(*, *)$$

означает всего лишь попарное умножение элементов двух матриц с одинаковыми индексами, но ни в коем случае не операцию умножения матриц в математическом смысле, которая должна выражаться программой, состоящей из векторных унарных и бинарных операторов.

Унарные (с одним операндом) векторные операции не являются поэлементными и имеют специфический смысл. Примеры некоторых унарных операций приведены в табл. 5.1. Эти операции могут выполняться и над векторами, получаемыми из многомерных массивов.

Таблица 5.1.

Унарные операции в расширениях Фортрана

Функция	Операция	Математический смысл
SUM(A)	+	$A_1.+\ldots+A_k$
PROD(A)	*	$A_1*\ldots*A_k$
AND(B)	$\wedge$	$B_1\wedge\ldots\wedge B_k$
OR(B)	$\vee$	$B_1\vee\ldots\vee B_k$
MAX(A)	$\geq$	MAX($A_1,\ldots,A_k$)
MIN(A)	$\leq$	MIN($A_1,\ldots,A_k$)

Условные переходы в параллельных ЭВМ организуются с помощью оператора IF следующим образом

$$\text{IF } (f\{l_1,\text{K},l_k\})\ m_1,m_2$$

где m_1,m_2 — метки перехода по адресам программы; f — логическая функция k переменных (обычно k равно размерности процессорного поля, например $k = 64$).

Каждый ПЭi имеет бит режима, который устанавливается в 0 или 1 в зависимости от результата выполнения операции именно в этом процессоре, например:

$$l_i = f_1(A(I).LT.0.0) \qquad (5.1)$$

Логическая переменная l_i = 1, если условие в правой части выполняется, и l_i = 0, если условие не выполняется. Таким образом, после выполнения в каждом процессоре операции (5. 1) в ПП формируется логический вектор $\{l_1, \text{K}, l_k\}$, который затем считывается в ЦУУ. В качестве функции $f\{l_1, \text{K}, l_k\}$могут применяться различные логические операции, например:

1) ALL $\{l_1, \text{K}, l_k\}$ означает, что логическое выражение в операторе является истинным, если все l_i = 1;

2) ANY $\{l_1, \text{K}, l_k\}$ означает, что функция истинна, если хотя бы один l_i = 1.

Конечно, в параллельных расширениях Фортрана должны быть обеспечены возможности вызова параллельных подпрограмм, написанных на различных языках, в том числе и ассемблере. При этом минимальная длина подпрограммы может равняться одному оператору языка ассемблер.

Описанные выше принципы построения параллельных ЯВУ являются основой для построения стандарта параллельного ЯВУ и в различных сочетаниях применяются в языках для существующих параллельных машин CRAY-1, CYBER-205, BSP, DAP, ПС-2000.

Как уже говорилось, мобильность программ для параллельных ЭВМ имеет принципиальное значение. Для обеспечения мобильности применяются два метода: метод стандартного Фортрана и метод трансляции программ с одного диалекта на другой.

Стандартный последовательный Фортран является средой, в которой относительно легко осуществляется перенос программ. Однако использование векторизующего компилятора для последовательного языка все-таки дает программы с низкой производительностью. Кроме того, стандартный Фортран не всегда позволяет эффективно закодировать специфические для данной ЭВМ операции.

Например, цикл

```
      DO 10 I = 1, 64                                    (5.2)
10    IF (A(I).LE.1.OE-6) A(I) = 1.0E-6
```

стандартного Фортрана записывается на Фортране для ЭВМ CRAY-1 с помощью нестандартной функции слияния *CUMPG*, иначе цикл не векторизуется:

```
      DO 10 I = 1, 64
10  A(I) = CUMPG(A(I), 1.0E-6, 1.0E-6 - A(I))
```

Таким образом, даже среда стандартного Фортрана не обеспечивает переноса программ без их изменения.

Метод трансляции программ позволяет применять различные параллельные диалекты ЯВУ и автоматическую трансляцию с одного диалекта на другой. При этом требуется M^2 трансляторов, где M — число диалектов ЯВУ. Если использовать промежуточный язык, то необходимо только около M трансляторов. Главный недостаток метода трансляции заключается в низкой эффективности трансляции (на уровне “среднего” программиста).

Лучшим средством, гарантирующим автоматический перенос программ, является разработка стандартов на параллельные ЯВУ, учитывающих особенности структур различных ЭВМ.

§ 5.3. Векторизующие компиляторы

Один из наиболее распространенных методов программирования для параллельных ЭВМ является метод оформления параллельных алгоритмов на последовательном ЯВУ. Полученные таким образом программы автоматически преобразуются в параллельные программы на машинном языке. Эффективность автоматического преобразования последовательных программ в параллельные относительно невелика, но для многих параллельных ЭВМ названный метод программирования является основным (совместно с программированием на ассемблере) по следующим трем причинам.

1. Параллельные алгоритмы, оформленные в программы на стандартных последовательных языках (чаще всего на Фортране), относительно легко переносимы не только внутри семейства параллельных ЭВМ одного типа, но и между ЭВМ различных типов. В частности, такие программы могут отлаживаться и выполняться на последовательных ЭВМ.

2. Программирование на распространенных последовательных языках хорошо освоено, существует много методической литера

туры. Отладка программ может производиться на ЭВМ разных типов: ЕС ЭВМ, персональных ЭВМ и др.

3. Автоматические распараллеливатели последовательных программ позволяют транслировать не только новые программы, но и большой запас пакетов прикладных программ, написанных ранее для последовательных машин. Если стоимость ПП невелика по сравнению со стоимостью всей установки, то экономически оправдано распараллеливание и эксплуатация пакетов с низким уровнем машинного параллелизма.

Анализ последовательной программы с целью извлечения параллелизма из программных конструкций называется *векторизацией*, а программные средства, предназначенные для этого, — *векторизующими компиляторами* (ВК).

Векторизующие компиляторы строятся для векторных ЭВМ, т. е. для конвейерных ЭВМ и ПМ.

Наибольшим внутренним параллелизмом, который можно использовать для векторизации программ, являются циклические участки, так как на них приходится основное время вычислений, и в случае распараллеливания вычисления в каждой ветви производится по одному и тому же алгоритму.

В методах распараллеливания циклов используется довольно сложный математический аппарат. Наиболее распространенными методами распараллеливания являются: метод параллелепипедов, метод координат, метод гиперплоскостей. Объектами векторизации в этих методах являются циклы типа DO в Фортране.

При постановке в общем виде задачи распараллеливания циклов вводится понятие “пространство итераций” — n-мерное целочисленное пространство с координатными осями I_1, K, I_n, соответствующими индексным переменным исходного цикла. Каждая итерация (повторение тела цикла при различных значениях индексов) представляет собой точку в этом пространстве и характеризуется значением вектора (i_1, K, i_n), где i_j — номер выполнения итерации на j-м уровне вложенности. Исходный тесногнездовой цикл имеет вид

```
DO 1 I1 = 1, M1
. . .
DO 1 In = 1, Mn                                    (5. 3)
```

```
      S(i1, ..., in)
1     CONTINUE
```

Шаг изменения цикла предполагается равным единице. Тело цикла $S(i_1,\text{K},i_n)$ состоит из последовательности пронумерованных по порядку операторов. На тело цикла, как правило, накладываются некоторые ограничения, зависящие от типа ЭВМ и метода распараллеливания: все индексные выражения должны быть линейными функциями от параметров цикла; не допускаются условные и безусловные переходы из тела цикла за его пределы, а внутри цикла передача управления может осуществляться только вперед, т. е. операторы IF и GO TO нельзя применять для организации цикла. Не допускается использование в теле цикла операторов ввода-вывода и обращений к подпрограммам.

Для цикла (5.3) пространство итераций представляет собой набор целочисленных векторов $I = \{i_1,\text{K},i_n\},\ 1 \le i_j \le M_j$. Распараллеливание цикла (5.3) заключается в разбиении этого пространства на подобласти, внутри которых все итерации могут выполняться одновременно и при этом будет сохраняться порядок информационных связей исходного цикла. Порции независимых итераций можно искать в виде n-мерных параллелепипедов, гиперплоскостей и т. д. в соответствии с методом распараллеливания.

Необходимое условие параллельного выполнения i-й и j-й итераций цикла записывается в виде

$$(\text{OUT(i)}\wedge\text{IN(j)})\vee(\text{IN(i)}\wedge\text{OUT(j)})\vee(\text{OUT(i)}\wedge\text{OUT(j)})=\varnothing \qquad (5.4)$$

Здесь IN(i) и OUT(i) — множества входных и выходных переменных i-й итерации. Два первых дизъюнктивных члена из (5.2) задают учет информационной зависимости между итерациями, т. е. параллельное выполнение может быть невозможным, если одна и та же переменная используется в теле цикла как входная и как выходная. Третий член учитывает конкуренционную зависимость, которая появляется в теле цикла в качестве выходной больше одного раза.

Отношение зависимости между итерациями можно представить в виде графа D. Вершины графа соответствуют итерациям цикла. Вершины i и j соединяет дуга, если они зависимы, и тогда $D(i, j) = 1$. Задачу распараллеливания цикла можно сформулиро

вать как задачу разбиения графа *D* на несвязные подграфы *Di*. Вершины, входящие в один подграф, должны быть независимыми и иметь смежные номера.

Проиллюстрируем применение *метода параллелепипедов* на примере следующего цикла [10]:

```
      DO 1 I = 2, 5
      DO 1 J = 2, 4                                    (5. 5)
1     X (I, J) = X (I - 1, J - 1) ** 2
```

Пространство итераций и информационные связи приведены на рис. 5.4, а. В данном случае вершины, лежащие вдоль диагонали прямоугольника, информационно связаны. Так, в итерации с номером (2. 2) генерируется переменная *X*(2, 2), которая затем используется в качестве входной переменной в итерации (3. 3). Разбиение итераций на параллелепипеды можно осуществить вдоль осей *I* и *J* (см. рис. 5.4, а).

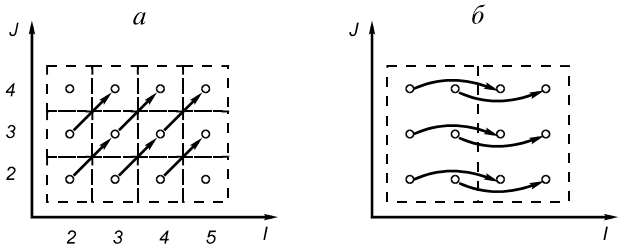


Рис.5.4. Разбиение итераций:
а — для цикла (5.5); *б* — для цикла (5.6)

Проиллюстрируем применение метода параллелепипедов для другого цикла:

```
      DO 1 I = 2, 5
      DO 1 J = 2, 4                                    (5. 6)
1     X (I, J) = X (I - 2, J) ** 2
```

Пространство итераций и информационные связи приводятся на рис. 5.4, б. Разбиение на параллелепипеды можно провести вдоль оси *I* с шагом, равным двум, а вдоль оси *J* — с максимально возможным шагом (в данном случае равным трем). Таким образом,

внутри каждого параллелепипеда оказалось по шесть независимых итераций, которые можно выполнить параллельно.

Если в цикле есть информационные связи между операторами, запрещающие параллельное выполнение, то можно попытаться устранить эти связи, применив *метод координат*. Данный метод предполагает осуществление в теле цикла некоторых преобразований, а именно: перестановку операторов, введение дополнительных переменных и массивов. Подобные преобразования производятся таким образом, чтобы изменить направление или устранить запрещающие связи между операторами в различных итерациях. При этом результат вычислений не должен изменяться.

В качестве примера использования метода координат рассмотрим следующий цикл:

```
      DO 4 I = 2, N
1     X (I) = Y (I) + Z (I)
2     Z (I) = Y (I - 1)
3     Y (I) = X (I + 1) ** 2
4     CONTINUE
```

Запрещающими связями в данном примере является связь между использованием переменной Y(*I*-1) во втором и генерацией *Y*(*I*) в третьем операторах, а также связь генерации *X*(*I*) в первом операторе с использованием генерации *X*(*I* + 1) в третьем операторе. Направление первой связи по *Y* можно изменить на противоположное перестановкой второго и третьего операторов. Вторую связь по *X* можно устранить введением дополнительного массива *T*. Таким образом, эквивалентный исходному преобразованный цикл, который можно выполнить одновременно для всех значений индекса *I*, будет иметь такой вид:

```
      DO 4 I = 2, N
      R (I) = X (I + 1)
1     X(I) = Y (I) + Z (I)
3     Y (I) = R (I) ** 2
2     Z (I) = Y (I - 1)
4     CONTINUE
```

Необходимо отметить, что методом параллелепипедов и методом координат можно распараллеливать как одномерные, так и многомерные тесногнездовые циклы.

В рассматриваемых методах распараллеливания индексные выражения элементов массивов должны быть линейными функциями относительно параметров цикла, т. е. иметь вид $A*I\pm B$, где A и B — целые константы; I — параметр цикла *DO*. Необходимо также, чтобы в индексных выражениях использовались одноименные индексные переменные. Не допускается применение нелинейных индексов типа $X(I*J*K)$ или косвенной индексации, например $X(N(I))$.

Основным препятствием к распараллеливанию циклов является так называемое *условие Рассела* — использование в теле цикла простой неиндексированной переменной раньше, чем этой переменной присваивается в цикле некоторое значение, например:

```
      DO 1 I = 1, N                    DO 1 I = 1, N
1     S = S + X (I)        или         X (I) = S * Y (I)
                                   1   S = Z (I) ** 2
```

Распараллеливанию препятствуют и обратные информационные и конкуренционные связи между операторами тела цикла. Рассмотрим пример:

```
      DO 1 I = 1, N
1     X (I) = X (I - 1)
```

Итерации этого цикла связаны обратной информационной зависимостью с шагом, равным единице. Вообще говоря, конструкции вида

```
      DO 1 I = 1, N
      . . .
      X (I) = X (I + K)
      . . .
1     CONTINUE
```

где K — целочисленная переменная, векторизуются только в том случае, если знак переменной K совпадает со знаком инкремента цикла. В большинстве же случаев знак числа K не может быть определен на этапе компиляции, поэтому такие конструкции счита

ются невекторизуемыми. Из соотношения (5.4) следует, что если в теле цикла нет одноименных входных и выходных переменных, то подобный цикл векторизуется. Если в цикле генерация простой переменной встречается раньше, чем ее использование, то этот цикл также векторизуется:

```
      DO 1 I = 1, N
      S = X (I) + Y (I)
1     Z (I) = Z (I) - S
```

Информационная зависимость между итерациями появляется только в том случае, если в теле цикла имеются одноименные входные и выходные переменные с несовпадающими индексными выражениями, например *X*(*I*) и *X*(*I* - 1). Направление связи (прямое или обратное) зависит от номеров операторов, в которых используются эти переменные. Если совпадающие индексные выражения и связи между итерациями отсутствуют, то такие циклы распараллеливаются:

```
      DO 1 I = 1, N
      X (I) = Y (I + 1) + Z (I)
1     Y (I + 1) = X (I) ** 2
```

В этом случае одноименные пары *X*(*I*) и *Y*(*I* + 1) имеют совпадающие индексы и не препятствуют векторизации.

Рассмотрим часто встречающуюся на практике циклическую конструкцию, в которой переменная целого типа используется в качестве неявного параметра цикла:

```
      J = 0
      DO 1 I = 1, N
      J = J + 2
1     X (I) = Y (I)
```

В этом случае в массив *X* заносятся четные элементы из массива *Y*. Роль второго параметра цикла играет переменная *J,* и хотя для *J* формально выполняется условие Рассела, этот цикл можно распараллелить.

Иногда цикл, в котором выполняется условие Рассела, оказывается вложенным в другой цикл, например:

```
      DO 1 I = 1, N
      S = 0
      DO 2 J = 1, N
2     S = S + X (I, J)
      Y (I) = S
1     CONTINUE
```

В вектор *Y* заносятся значения суммы элементов строк матрицы *X*. Внутренний цикл по *J* не распараллеливается (выполняется условие Рассела для переменной *S*), а внешний цикл по *I* можно распараллелить, так как переменная *S* в нем получает значение перед входом во внутренний цикл.

Рассмотрим общую структуру ВК (рис. 5.5). Как правило, ВК состоит из синтаксического анализатора, распараллеливателя циклов, модуля оценки качества распараллеливания, генераторов программ на параллельном ЯВУ или в машинных кодах.

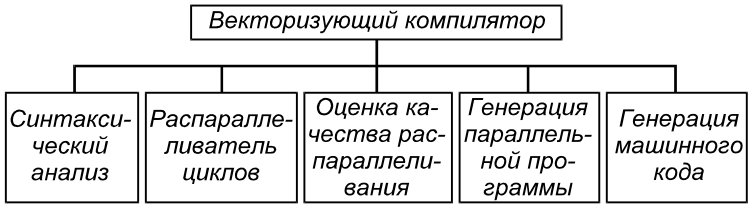


Рис.5.5. Состав функций векторизующего компилятора

На первом этапе работы ВК производится синтаксический анализ исходной последовательной программы и выдаются сообщения о найденных ошибках, заполняются таблицы входных и выходных переменных для всех циклов, а также проверяется выполнение необходимых требований: линейность индексных выражений, отсутствие операторов и обращений к подпрограммам и т. п. Кроме того, на этапе синтаксического анализа строится так называемый временной профиль программы, в котором оценивается время выполнения программы. Недостающую информацию для такой оценки компилятор может запрашивать в диалоговом режиме. Очевидно, что циклы, имеющие наибольший вес, необходимо распараллеливать в первую очередь.

На втором этапе циклы, для которых выполняются все ограничения, анализируются блоком распараллеливания. В этом блоке

реализован один или несколько методов распараллеливания. Анализ производится начиная с вложенных циклов. Для циклов, которые не распараллеливаются, выдается список “узких мест”, т. е. информация о конкретных причинах: условие Рассела, информационная зависимость итераций и т. д.

На третьем этапе оценивается достигнутая степень распараллеливания для каждого цикла и на основе этой информации вычисляется ускорение параллельной программы по сравнению с исходной последовательной.

На четвертом этапе, если достигнутое ускорение удовлетворяет программиста, ВК может сгенерировать параллельную программу на ЯВУ (например, на параллельном Фортране) или программу в машинных кодах, уже готовую к выполнению. Если же достигнутая степень параллелизма неудовлетворительна, то программист, используя полученный список “узких мест”, может попробовать преобразовать циклы таким образом, чтобы они стали векторизуемыми. Если это не удается сделать, то необходимо менять алгоритм решения задачи.

Рассмотренная структура ВК является весьма обобщенной. В реальных ВК некоторые модули могут отсутствовать (например, модуль оценки качества или генератор программ на параллельном ЯВУ). Кроме того в ВК могут быть реализованы и некоторые дополнительные функции.

Как правило, ВК обрабатывает программу, написанную на стандартном ЯВУ: Фортран, PL-1, Паскаль, Си и др. В этом случае, чтобы включить в систему программирования ЭВМ новый язык, для него приходится реализовывать свой ВК.

Однако существуют и многоязыковые ВК. При использовании таких ВК программа с ЯВУ сначала транслируется в некоторую промежуточную форму (промежуточный язык), удобную для распараллеливания, и вся дальнейшая обработка производится на уровне промежуточного языка. В данном случае для включения нового языка в систему программирования необходимо к существующему ВК добавить транслятор с этого языка в промежуточную форму.

Векторизующий компилятор может иметь и несколько модулей генерации машинного кода, каждый из которых генерирует код для своего вида ЭВМ. Таким образом, возможно построение

универсальных многоязыковых ВК, производящих распараллеливание программ на нескольких ЯВУ для разных типов ЭВМ.

§ 5.4. Языки программирования для транспьютерных систем

Разнообразие и широкое распространение транспьютерных систем предъявляют повышенные требования к системам программирования для транспьютеров.

Прежде чем перейти к рассмотрению систем программирования, опишем концентуальную модель вычислений в мультитранспьютерной системе:

1. Вычислительный процесс представляет собой параллельное выполнение нескольких подпроцессов на различных транспьютерах в асинхронном режиме. Синхронизация осуществляется путем обмена сообщениями. Каждый параллельный процесс в мультитранспьютере может, в свою очередь, быть разбит на более мелкие подпроцессы, которые выполняются квазипараллельно в одном транспьютере в режиме разделения времени.

2. В однотранспьютерной системе запуск параллельных ветвей осуществляется командой “стартовать процесс” (операция *THREAD* в ЯВУ), которая по своим функциям подобна оператору FORK: заводится критический блок, и новый процесс планировщиком включается в очередь процессов, претендующих на время процессора. Команда “завершить процесс” соответствует оператору JOIN: процесс исключается из очереди, из счетчика критического блока вычитается единица и, если содержимое счетчика оказывается равным нулю, завершается вышестоящий процесс. Обмен между квазипараллельными процессами осуществляется через программные каналы, следовательно, общие переменные отсутствуют, то есть реализуется модуль МКМД-ЭВМ с раздельной памятью. При этом конкуренции за ресурсы нет, так как их распределение производится программистом на этапе написания программы.

Процессы разделяются на: срочные, которые выполняются по очереди до полного завершения каждого из них; и на несрочные, которые обслуживаются по кольцу с равным интервалом времени. Управление ими осуществляется встроенным планировщиком, реализованным микропрограммно или аппаратно.

В случае необходимости реализовать разделяемую общую память (например, буфер для нескольких процессов) пользователь обязан сам организовать синхронизацию с помощью семафоров.

3. Реальный физический параллелизм процессов достигается в мультитранспьютере (МТ). Чтобы обеспечить этот параллелизм, пользователь должен создать сеть транспьютеров и разместить части задачи в узлах этой сети.

Программное обеспечение транспьютерных систем включает системы программирования, операционные системы, пакеты прикладных программ и некоторые специальные средства. Основным элементом систем программирования являются языки программирования. В дальнейшем будут рассмотрены: язык Оккам-2, особенности языков высокого уровня Фортран и Си для транспьютеров, возможности программирования средством операционной системы Helios.

Оккам-2. Идеология транспьютеров содержит две составляющие: традиционную и параллельную. Традиционная часть соответствует работе процессора одиночного транспьютера, который является обычной последовательной ЭВМ. Если в системе появляется более одного транспьютера, необходим параллелизм в программах, каналы, распределение ресурсов. Эти две составляющие выражены и в языках программирования, в частности, в Оккаме.

В Оккаме имеются все средства для последовательного програмирования: константы, переменные, массивы, операторы присваивания, операторы циклов, вызывы подпрограмм и другое. Эту часть в дальнейшем мы рассматривать не будем.

Новое, “параллельное” в Оккаме — это в первую очередь [16]:

1. Конструкции выбора ALT, параллельного выполнения *PAR* и повторители *FOR*.

2. Операторы ввода *c*?*x* и выводы *c*!*x*, используемые для синхронизации взаимодействующих процессов по однонаправленным линиям связи.

3. Средства для описания конфигурации мультитранспьютера, распределения параллельных частей задачи по элементам МТ, то есть средства конфигурирования.

Поскольку конструкции по пунктам 1 и 2 рассматривались уже в § 3.2, то более подробно остановимся на средствах конфигу

рирования, в связи с тем, что в указанном параграфе они рассмотрены кратко.

Распределение процессов между несколькими процессорами в транспьютерной сети называется *конфигурированием* программы. В Оккам принят статический подход к конфигурированию, при котором программист заранее на этом этапе программирования должен определить, на каких узлах сети необходимо разместить процессы. В некоторых областях применения может быть более подходящим динамический подход, где до выполнения не определяется, на каком процессоре должен выполняться каждый процесс. Если необходимо, язык Оккам можно использовать для кодирования механизмов динамической загрузки и выполнения процессов.

Для описания того, как программа распределяется по процессорам в сети, можно воспользоваться несколькими различными подходами. Например, программист может подготовить три отдельных файла, содержащих:

1) описание топологии сети;

2) описание структуры программы, то есть описание состава и связей независимых процессов;

3) описание того, как отображаются процессы и каналы на процессоры и линии связи.

В языке Оккам-2 эти три аспекта фактически объединены в одно описание, которое называется "*Описание конфигурации"*. Текст описывает Оккам-программу пользователя на самом высоком уровне и содержит достаточно информации для отображения программы на транспьютерную сеть. Это описание затем используется компилятором для создания загружаемого файла пользовательской программы.

Для того, чтобы сконфигурированная программа была загружена в сеть, необходимо, чтобы один из процессоров в сети был соединен с главной ЭВМ. Этот процессор называется *корневым процессором*. Между корневым процессором и всеми остальными в сети должен быть путь через транспьютерные линии связи. Корневой процессор любой сети должен быть описан как первый процессор в конфигурации. Нет необходимости включать в описание конфигурации линию связи, соединяющую главную ЭВМ с корневым процессором.

При описании конфигурации используются три оператора языка Оккам:

PLACED PAR
PROCESSOR *номер тип. процессора*
PLACE *канал* AT *адрес*

Операторы PLACED PAR и PROCESSOR совместно описывают распределение логических частей задачи по физическим транспьютерам, а оператор PLACE AT закрепляет логические связи между частями задачи, называемые каналами, за физическими связями транспьютеров.

Рассмотрим пример описания конфигурации для распределения кода программы между четырьмя транспьютерами в сети. Пример состоит из двух различных процессов: control и work, которые должны быть описаны на уровне описания конфигурации.

Процедура control выполняется на корневом транспьютере и включает процессы, соединенные с главной ЭВМ для посылки управляющих сообщений программе, выполняющейся на главной ЭВМ. Процедура work выполняется на остальных трех транспьютерах.

На рис. 5.6 показана логическая структура программы, хотя она может выполняться и на меньшем числе транспьютеров, в том числе и на одном. Все зависит от распределения частей программы по процессорам, задаваемым описанием конфигурации. Дальнейшее рассмотрение относится к исполнению программы четырьмя транспьютерами. Разбиение программы на логические блоки, которые можно выполнять параллельно, производится программистом.

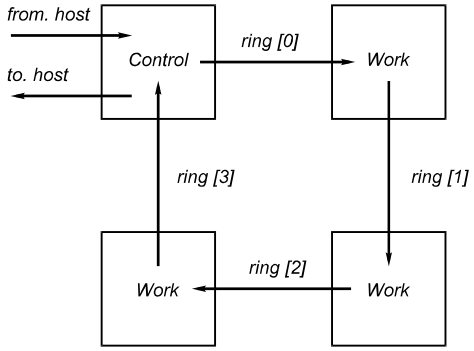


Рис.5.6. Логическая структура программы

Описание конфигурации содержит следующие компоненты:

- описание процедуры control;
- описание процедуры work;
- операторы PLACED PAR;
- отображение каналов на линии связи.

Процедура control имеет два канала — to. host и from. host, которые соединяют ее с главной ЭВМ. Эти каналы имеют канальный протокол MONITOR. Кроме того, имеется один канал для связи с конвейером, состоящим из процедур work, и один канал, связанный с последним процессом в конвейере. Эти каналы имеют канальный протокол PIPELINE. Интерфейс процедуры control выглядит следующим образом:

```
PROC control (CHAN OF MONITOR from. host, to. host
CHAN OF PIPELINE to. ring, from. ring)
```

Процедура work имеет вводной канал и выводной канал, каждый из которых имеет протокол PIPELINE. Интерфейс процедуры work имеет следующий вид:

```
PROC work (CHAN OF PIPELINE in, out)
```

Для описания конфигурации необходимо определить некоторые каналы, соединяющие все процессы друг с другом. Должны быть описаны два канала для связи прикладной программы с главной ЭВМ; они получили название from. host и to. host. Также

должны быть описаны четыре канала для связи четырех транспьютеров друг с другом; они описаны как массив каналов от ring[0] до ring[3]. Все эти каналы показаны на рис. 5.6. Первый канал ring[0] соединяет процессор 0 с процессором 1 и так далее по кольцу. Ниже показано простое описание конфигурации без отображения каналов на линии связи:

```
CHAN OF MONITOR from. host, to. host:
[4]CHAN OF PIPELINE ring:
PLACED PAR
    PROCESSOR 0 T414
        control (from. host, to. host, rihg[0], ring[3])
    PROCESSOR 1 T414
        work (ring[0], ring[1])
    PROCESSOR 2 T414
        work (rihg[1], ring[2])
    PROCESSOR 3 T414
        work (ring[2], ring[3])
```

Теперь надо отобразить каналы на линии связи. Для этого необходимо для каждого процессора перечислить входные и выходные каналы. На рис. 5.6 даны имена каналов. Что же касается физических связей между транспьютерами, то они определяются топологией сети транспьютеров. Топология может разрабатываться как для решения одной задачи, так и для решения нескольких задач. Во втором случае физические связи не меняются при переходе от задачи к задаче. Но в любом случае перед выполнением задачи физические связи должны быть установлены вручную с помощью разъемов. В некоторых случаях физические связи между транспьютерами устанавливаются с помощью программируемого коммутатора.

Если “интеллект” компилятора позволяет производить закрепление логических каналов за физическими связями, то описание топологии должно включаться в состав описания конфигурации. В противном случае это закрепление производится программистом вручную.

Добавив полученное таким образом отображение линий связи к приведенному выше частичному описанию, получим следующее описание конфигурации:

```
CHAN OF MONITOR from. host, to. host:
[4] CHAN OF PIPELINE ring:
PLACED PAR
  PROCESSOR 0 T414
    PLACE from. host AT link0in:
    PLACE to. host AT link0out:
    PLACE ring[0] AT link2out:
    PLACE ring[3] AT link3in:
    control(from. host, to. host, ring[0], ring[3])
  PROCESSOR 1 T414
    PLACE ring[0] AT link3in:                          (5.5)
    PLACE ring[1] AT link2out:
    work (ring[0], ring[1])
  PROCESSOR 2 T414
    PLACE ring[1] AT link3in:
    PLACE ring[2] AT link2out:
    work (ring[1], ring[2])
  PROCESSOR 3 T414
    PLACE ring[2] AT link3in:
    PLACE ring[3] AT link2out:
    work (ring[2], ring[3])
```

Наконец, заметим, что все три процессора, на которых выполняются процессы work, имеют очень похожую структуру. Можно воспользоваться этой регулярностью и описать все три процессора одним оператором с использованием конструкции *replicated* PLACE PAR. Ниже приведена конечная версия описания конфигурации, где используется конструкция повторения с индексом *i*, которая принимает значения 1, 2 и 3:

```
CHAN OF MONITOR from. host, to. host:
[4]CHAN OF PIPELINE ring:
PLACED PAR
  PROCESSOR 0 T414
    PLACE from. host AT link0in:
    PLACE to. host AT link0out:
    PLACE ring[0] AT link2out:
    PLACE ring[3] AT link3in:                    (5.6)
    control(from. host, to. host, ring[0], ring[3])
  PLACED PAR i = 1 FOR 3
    PROCESSOR i T414
      PLACE ring[i -1] AT link3in:
      PLACE ring[i] AT link2out:
      work (ring[i - 1], ring[i])
```

Языки высокого уровня. Несмотря на эффективность языка Оккам постоянно существует стремление использовать для программирования транспьютеров традиционные последовательные ЯВУ, особенно если учесть, что вычисления в пределах одного транспьютера являются чисто последовательными. Здесь осуществлено несколько подходов [15].

Наиболее очевидный способ заключается в том, чтобы описывать взаимодействие транспьютеров на Оккаме, а описание работы внутри одного транспьютера на Фортране или Си включать как фрагмент в эту программу. Этот способ особенно привлекателен, потому что имеется уже большой объем прикладного программного обеспечения на ЯВУ. Однако включение фрагментов на ЯВУ в программу на Оккаме распространения не получил по следующим причинам: необходимым оказалось изучение нового и достаточно необычного языка Оккам. Кроме того, включение фрагментов на ЯВУ в Оккам является сложной, кропотливой процедурой, чреватой ошибками.

Другой подход состоял в том, чтобы создавать языки, в которых содержались бы собственные средства для описания параллелизма процессов и распределения ресурсов между этими процессами.

Главное требование к таким ЯВУ, пусть даже в ущерб эффективности, чтобы они строились на базе распространенных последовательных ЯВУ с минимумом изменений и дополнений.

Средства для описания параллелизма процессов могут вводиться двумя способами: путем внесения изменения в синтаксис языка и расширением языка за счет введения специальных процедур. Примером первого подхода является использование операторов типа par {операторы}. Операторы в скобках задают процессы, которые будут выполняться квазипараллельно (в режиме разделения времени). Совместно с *par* { } могут использоваться повторители *for*.

Для работы с каналами транспьютера введен новый тип данных — канал, который используется в операторах присваивания.

Присваивание обычной переменной значения переменной типа канал означает чтение из канала значения и размещение его в первой из упомянутых переменных. Обратное присваивание задает помещение в канал, имя которого определяется именем переменной типа канал, значения переменной обычного типа, указанной справа от знака присваивания. При описании канальной переменной с указанием адреса физического канала можно таким образом определить передачу сообщения между процессами в разных транспьютерах, связанных этим каналом.

В этом же языке имеется конструкция *CASE*, аналогичная по функциям конструкции *ALT* в Оккаме.

Другой подход в описании параллелизма процессов состоит в расширении обычных ЯВУ за счет специальных процедур. Для определенности будем рассматривать язык Фортран. В нем используются следующие процедуры этапа исполнения: *THREAD*, *SEMA*, *TIMER*, *CHAN*, *NET*, *ALT*. Рассмотрим функции, выполняемые некоторыми из этих процедур.

Процедура *THREAD* имеет разновидности *F*77 *THREAD START*, *F*77 *THREAD CREATE* и ряд других.

Вызов процедуры сопровождается набором аргументов, например:

CALL F77 THREAD START (SUB, WSARRAY, WSSIZE, FLAGGS, NARGS, ARG1, . . . , ARGN)

Здесь *SUB* — имя запускаемой процедуры, которая будет выполняться в режиме разделения времени на одном транспьютере совместно с основной ветвью; *WSARRAY*, *WSSIZE* — начальный адрес и размер рабочей области; *FLAGGS* — приоритет ветви, *NARG* — количество аргументов процедуры, *ARG1*, ..., *ARGN* — формальные параметры процедуры. Из приведенных выше описаний видно, что процедура *THREAD* в общем выполняет те же функции, что и команды “стартовать процесс” и “завершить процесс” языка Оккам.

Управление запускаемой ветвью осуществляется встроенным планировщиком транспьютера.

Все ветви, запускаемые поочередно с помощью *THREAD*, выполняются в одном транспьютере.

Своеобразие языка Фортран состоит в том, что подпрограммы языка Фортран не реентерабельны, поэтому процедуры *THREAD* являются разделяемым многими ветвями ресурсом и защищаются от одновременного использования семафором, который создается с помощью процедуры *F*77 *SEMA*.

Процедура *CHAN* также имеет ряд модификаций. Модификация:

F77 CHAN IN MESSAGE (LENGTH, BUFFER, ICHANADDR)

позволяет ввести сообщение через канал с именем *ICHANADDR*, а модификация:

F77 CHAN OUT MESSAGE (LENGTH, BUFFER, ICHANADDR)

— вывести сообщение через канал. Таким образом, обе процедуры выполняют те же функции, что и операторы *c*?*x* и *c*!*x* в языке Оккам.

С помощью процедуры *CHAN* обеспечивается связь как между ветвями (*THREAD*) одной задачи, выполняемой в одном транспьютере, так и между ветвями, выполняемыми в разных транспьютерах. Эти ветви называются в параллельном Фортране *TASK*.

Выше были представлены два способа описания параллельных процессов: путем расширения синтаксиса за счет новых опе

раторов и путем использования библиотечных процедур. Однако, для обоих языков остался нерешенным вопрос о распределении ресурсов между параллельными частями задач. В МКМД-ЭВМ с общей памятью ресурсы распределяются динамически (см. § 3.1). В языке Оккам распределение задается пользователем статически на этапе написания программы с помощью языка конфигурации. Аналогичный язык используется и в упоминавшихся выше ЯВУ.

В языке Фортран имеются средства, позволяющие организовать выполнения по принципу “фермы”. В “ферме” имеется программа “мастер”, которая выдает порции работы программам, называемым “работники” по мере их освобождения от предыдущей порции работы. Количество программы “работник” столько, сколько транспьютеров в системе. В режиме “ферма” от пользователя не требуется распределять ресурсы, это делается автоматически программным обеспечением мультитранспьютера.

Helios. Одной из наиболее распространных операционных систем для транспьютеров является система *Helios*, разработанная фирмой Perihelion Software Ltd. Helios называется распределенной, поскольку ее ядро находится в каждом транспьютере мультитранспьютерной сети.

Helios аккумулирует в себе все возможности параллельного программирования, описанные выше, и добавляет новые.

Helios построена на базе распространенной ОС *UNIX* и включает в себя все возможности *UNIX*, в том числе и язык управления и программирования SHELL. С другой стороны, HELIOS является надстройкой над операционной системой базовой машины, которая используется мультитранспьютером как средство ввода-вывода и для связи с пользователем. Обычно, это ПЭВМ типа IBM PC. Следовательно, HELIOS является надстройкой над MS DOS.

Helios выполняет все функции обычных операционных систем: организацию ввода-вывода, поддержку дисков, управление файлами, редактирование, связь с пользователем и др.

В Helios имеется ряд возможностей для параллельного программирования: на языке Shell, на языках Фортран, Си. Однако, особенностью этой ОС является наличие *языка CDL* (Component Distribution Lauguage), предназначенного для описания связей между подзадачами, которые в CDL называются компонентами.

Компоненты могут быть написаны на традиционных ЯВУ или на языке Shell.

Программа на CDL перечисляет компоненты, подлежащие выполнению в транспьютерной сети, задает порядок их выполнения, описывает связи между ними. Рассмотрим основные конструкции CDL.

CDL содержит описательную и исполнительную части. Описательная часть позволяет описать компоненты задачи с нестандартными требованиями, например, описание:

```
component numbercrunch {
    processor T800;
    memory 32768;
    streams < left, > right
    }
```

требует, чтобы компонент numbercrunch был расположен на транспьютере T800 с объемом памяти не мене 32 Кбайт. Кроме того, этот компонент связан с внешним миром двумя потоками сообщений, имена которых указаны после ключевого слова streams: из потока left производится чтение сообщения (знак “<” указывает на чтение), а в поток right производится запись сообщений (знак “>” указывает на запись сообщений).

Характер взаимодействия между компонентами определяется параллельными конструкторами и повторителями.

Если в задании больше двух компонентов, то каждая пара рядом стоящих конструкторов должна быть разделена одним из четырех конструкторов. Список конструкторов в порядке возрастания старшинства выглядит следующим образом: ^^, |||, |, <>.

Простой конструктор задает параллельное исполнение двух компонент, которые не обмениваются сообщениями, например:

```
square ^^ square
```

Здесь показано, что два независимых компонента случайным образом получают данные из входного потока, а их выходные данные также будут перемежаться в выходном потоке.

Конструктор “|||” определяет режим работы “ферма”, например, выражение:

power [4] ||| square

обозначает, что выходные данные от компонента power будут по мере поступления автоматически пересылаться случайным образом выбранному одному из четырех компонентов square, которые ожидают поступления информации на свой вход.

Конструктор "|" описывает паралельный конвейер из двух компонент, например, выражение:

square | square

позволяет получить на выходе величину, если на вход подается значение *x*.

Двухсторонний конструктор "<>" описывает двухсторонний обмен между двумя компонентами. Левый компонент считается главным, правый — подчиненным. Главный компонент передает сообщение подчиненному, который затем, подобно подпрограмме, возвращает ответ. При этом, главный компонент может продолжать работу сразу после пересылки сообщения, то есть параллельно с подчиненным.

Для удобства определения в задании несколько подзаданий из одинаковых и однотипно связанных компонент используются повторители. Повторитель содержит число в квадратных скобках, например: задание

A [3] <> B

эквивалентно заданию

A <> B <> B <> B

Повторитель может быть также задан с помощью диапазона значений. Например, повторитель [*i*<1, *j*<1] задает четыре значения: {0, 0}, {0, 1}, {1, 0}, {1, 1}. Символ "<" здесь указывает на верхнюю (включаемую) границу диапазона.

Используя такого рода поименованные повторители, можно строить сложные массивы копий компонента и определять их взаимные связи. Например, с использованием поименованных повторителей *i* и *j* двумерный массив размера 4×4 копий компонента *B*, связанных двусторонними потоками сообщение в top, мог бы выглядеть так:

```
component B [i, j]{
    streams, , , ,
                    < riqht {i, j}              > riqht {i, (j+1)%4},
                    < left {i, (j+1)%4},        > left {i, j}
                    < down {i, j},              > down {(i+1)%4, j}
                    < up {(i+1)%4, j}           > up {i, j},
    ;                                                                  (5.7)
}
<> [i<4, j<4] B{i, j}
```

В (5.7) знак %4 означает “по модулю 4”.

После ключевого слова *streams* имеется ряд запятых, между которыми не содержатся имена потоков. Это означает, что в CDL эти имена подразумеваются по умолчанию: первое пропущенное имя обозначает поток, принимаемый со стандартного устройства ввода *stdin*; второе пропущенное имя определяет поток, выводимый на стандартное устройство вывода *stdout*; третье — описывает поток, выводимый на стандартное устройство сообщений об ошибках *stderr*; наконец, четвертое место соответствует потоку, выводимому на стандартное устройство для отладки *stddbg*.

Последний оператор в (5.7) соответствует исполнительной части этой программы и задает характер взаимодействия всех компонентов.

Программа (5.7) на языке CDL затем с помощью компилятора CDL, входящего в состав Helios, автоматически преобразуется в программу загрузки элементов транспьютера соответстующими компонентами, между которыми устанавливаются физические связи, реализующие связи из (5.7). Загрузка производится с учетом физической конфигурации ресурсов, хранящихся в конфигурационном файле, о котором упоминалось ранее. Естественно, физические связи должны допускать те связи, которые требуются согласно программе на CDL. Если это не выполняется, то данная программа на CDL не может быть выполнена. В этом случае необходимо либо менять схему физических связей, либо “подгонять” программу на CDL на существующую систему физических связей.

Среди прикладных пакетов имеются многочисленные пакеты, написанные на ЯВУ и ассемблере для научных и инженерных рас

четов, для обработки графических изображений, для цифровой обработки сигналов и др.

§ 5. 5. Программное обеспечение и служебные алгоритмы суперскалярных процессоров

Наиболее очевидным достоинством суперскалярных вычислительных систем является традиционная система программирования на последовательных языках. Это делает возможным массовое использование суперскалярного параллелизма. Однако параллельный характер вычислений предъявляет дополнительные требования к системному программному обеспечению. Системное обеспечение должно выполнять следующие новые функции:

1. Выявлять и представлять в явной форме скрытый параллелизм последовательных программ, то есть обеспечить построение ЯПФ. Этот этап обычно является машинно-независимым.

2. Обеспечивать планирование (упаковку) полученной ЯПФ на имеющиеся ресурсы. Этот этап является машинно-зависимым. В результате планирования также получается ЯПФ, но в каждом ее ярусе содержатся только те операции, для которых в данном такте имеются ресурсы. Планирование производится с учетом длительности выполнения операций.

3. Поскольку уровень параллелизма обычных пользовательских программ невысок и не всегда этот параллелизм позволяет загрузить имеющиеся ресурсы (конвейеры), то одной из наиболее важных задач суперскалярного программирования является задача повышения уровня параллелизма последовательных программ.

4. Суперскалярные МП, как правило, имеют новую систему команд и “выжить” они могут только в том случае, если для них будет быстро создан большой объем не только системного, но и прикладного программного обеспечения. Поскольку разработка огромного числа прикладных пакетов для новой системы команд слишком трудоемка, возникает задача автоматического переноса этих пакетов, уже созданных в основном для МП *i*80×86 и МС 680×0.

Некоторые способы решения этих задач и будут рассмотрены далее.

Алгоритмы построения ЯПФ в значительной степени зависят от того, на языке какого уровня записано распараллеливаемое выражение.

Существует много способов распараллеливания арифметико-логических выражений на ЯВУ. Все они прямо или косвенно (через польскую запись) используют анализ приоритетов выполняемых арифметико-логических операций. Рассмотрим один из прямых методов.

Присвоим операциям следующие приоритеты P:

1. P (↑) = 3
2. P (∗), P(/) = 2
3. P (+), P(-) = 1

Алгоритм состоит в том, что исходное выражение многократно сканируется слева направо. При каждом сканировании выделяются наиболее приоритетные операции. Рассмотрим пример:

ВХОД	: a+b+c∗d∗l∗f+g	
ПРОХОД	1: T1+T2∗T3+g;	T1=a+b; T2=c∗d; T3=l∗f
	2: T4+t5; T4=T2∗T3;	T5=T1+g
	3: T6;	T6=T4+T5

Полученная ЯПФ представлена на рис. 5.7.

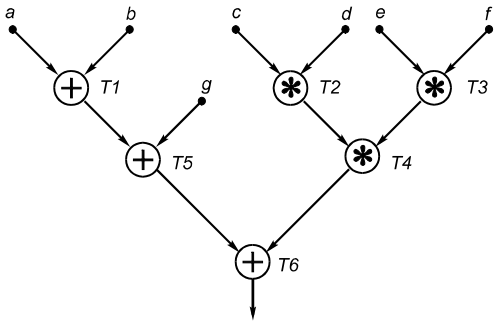


Рис. 5.7. Пример ЯПФ построенной по методу приоритетов

Параллелизм команд программы, представленной на уровне ассемблера, также определяется графом информационных зависи

мостей. При этом на этапе распараллеливания могут применяться различные методы устранения зависимостей с целью повышения параллелизма.

Построение ЯПФ для отрезка программы производится путем определения зависимости пар смежных команд при анализе программы сверху вниз согласно алгоритму на рис. 5.8. Для своего выполнения алгоритм требует порядка n^2 операций, где n — число команд в отрезке программы.

Ниже приводится пример последовательной программы (слева) и ее ЯПФ, построенной по описанному алгоритму (справа)

```
mov eax, [esi]
imul eax, [ebx]
add [edi], eax
add ebx, 4
add esi, 4
dec ecx
```

```
mov eax, [esi]    dec ecx
imul eax, [ebx]   add esi, 4
add [edi], eax    add ebx, 4
```

Построение ЯПФ и упаковка на ЯВУ производятся в общем случае для многоместных операторов, в то время как в реальной вычислительной системе параллельные АЛУ или процессоры выполняют только двухместные операции, соответствующие командам машинного языка. Отсюда следует, что в реальной системе построение ЯПФ и упаковка должны производиться на уровне системы команд (ассемблер, двоичный код), а не на ЯВУ.

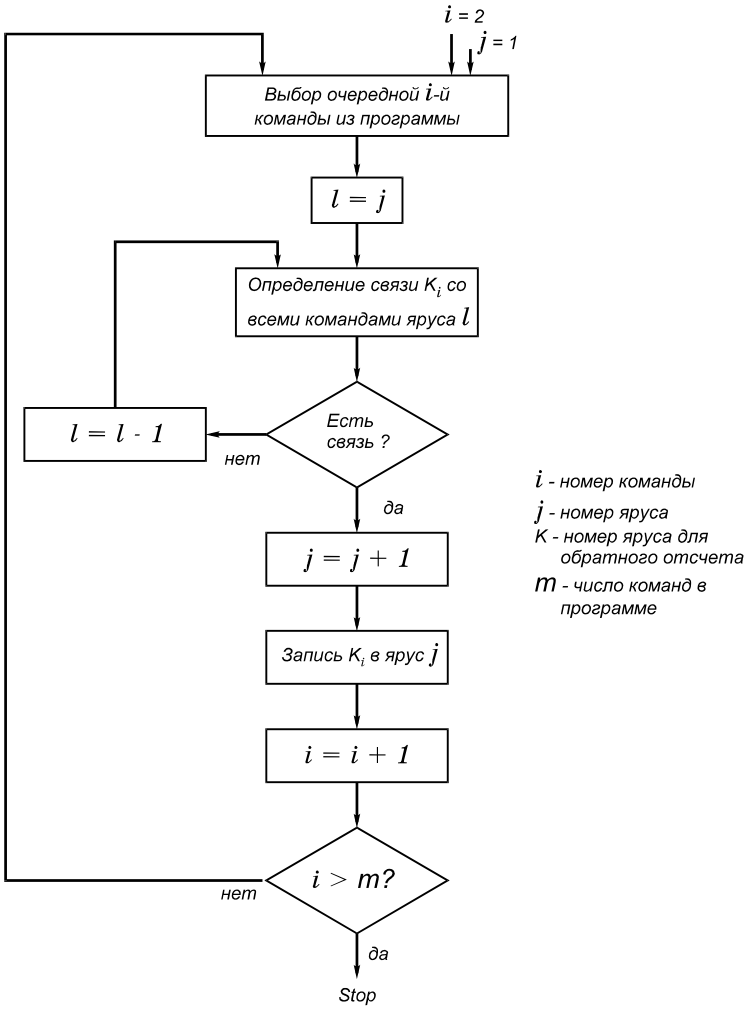


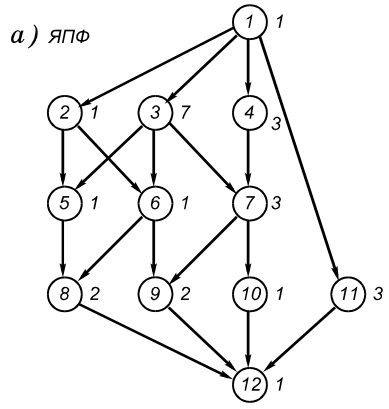
Рис.5.8. Алгоритм построения ЯПФ на языке машинного уровня

Планирование является оптимизационной задачей и уже при небольшом числе вершин информационного графа, точнее, ЯПФ, время перебора при распределении ресурсов будет недопустимо большим.

Чтобы уменьшить время планирования, используют эвристические алгоритмы, среди которых наибольшее распространение получили *списочные расписания* [21], которые при линейном росте времени планирования при увеличении числа вершин, дают результаты, отличающиеся от оптимальных всего на 10-15%.

В таких расписаниях оператором присваиваются приоритеты по тем или иным эвристическим правилам, после чего операторы упорядочиваются по убыванию или возрастанию приоритета в виде линейного списка. В процессе планирования затем осуществляется назначение операторов процессорам в порядке их извлечения из списка. Ниже дан пример построения списочного расписания.

На рис. 5.9, а представлена исходная ЯПФ. Номера вершин даны внутри кружков, а время исполнения — около вершин. На рис. 5.9, б приведены уровни вершин. Под уровнем вершины понимается длина наибольшего пути из этой вершины в конечную, то есть длина критического по времени пути из данной вершины в конечную.



б) Уровни вершин

1	2	3	4	5	6	7	8	9	10	11	12	*номера вершины*
14	5	13	9	4	4	6	3	3	2	4	1	*уровни*

Рис.5.9. Примеры двухпроцессорных списочных расписаний

Наиб. кр. вр. путь КОН		*Наиб. кр. оп. путь КОН*		*Произв. выбор*
П	*В*	*П*	*В*	*В*
14	*1*	*5*	*1*	*9*
13	*3*	*4*	*4*	*7*
9	*4*	*4*	*2*	*11*
6	*7*	*4*	*3*	*2*
5	*2*	*3*	*5*	*12*
4	*6*	*3*	*7*	*10*
4	*5*	*3*	*6*	*5*
4	*11*	*2*	*10*	*3*
3	*9*	*2*	*9*	*6*
3	*8*	*2*	*8*	*1*
2	*10*	*2*	*11*	*8*
1	*12*	*1*	*12*	*4*

в) Приоритеты и списки вершин
(П — приоритеты, В — номера вершин)

г) Реализация расписаний
(P1, P2 — процессоры, X — простой процессора)

Наиб. кр. вр. путь КОН

P1	*1*	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*7*	*7*	*7*	*9*	*9*	*12*
P2	*X*	*4*	*4*	*4*	*2*	*11*	*11*	*11*	*6*	*5*	*8*	*8*	*10*	*X*

Наиб. кр. оп. путь КОН

P1	*1*	*4*	*4*	*4*	*11*	*11*	*11*	*X*	*X*	*5*	*6*	*8*	*8*	*9*	*9*	*12*
P2	*X*	*2*	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*7*	*7*	*7*	*10*	*X*	*X*	*X*

Произвольный выбор

P1	*1*	*11*	*11*	*11*	*4*	*4*	*4*	*X*	*X*	*7*	*7*	*7*	*9*	*9*	*12*
P2	*X*	*2*	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*5*	*6*	*8*	*8*	*10*	*X*

Рис.5.9. Примеры двухпроцессорных списочных расписаний (продолжение)

Построим для примера три расписания из множества возможных.

Для каждого из них найдем характеристики всех вершин по соответствущим алгоритмам и примем значения этих характеристик в качестве величин приоритетов этих вершин.

В первом расписании величина приоритета вершины есть ее уровень. Это расписание НАИБ. КР. ВР. ПУТЬ КОН (наибольший критический по времени путь до конечного оператора). Во втором расписании величина приоритета вершины есть ее уровень без учета времени исполнения, то есть здесь веса всех вершин полагаются одинаковыми (единичными). Это расписание НАИБ. КР. ОП. ПУТЬ КОН (наибольший критический по количеству операторов путь до конечного оператора). В третьем расписании величины приоритетов полагаются одинаковыми и последовательность выборки вершин определяется случайно на равновероятной основе, но с учетом зависимостей вершин. Это расписание ПРОИЗВОЛЬНЫЙ ВЫБОР используется в целях сравнения с другими для определения их эффективности.

Об оптимальности расписания можно судить по количеству простоев процессоров. Первое расписание дает 2, второе — 5, третье — 4 такта простоя.

Серьезным препятствием для использования суперскалярных МП является низкий уровень параллелизма прикладных программ. Рассмотрим этот вопрос на примерах.

Скалярный параллелизм определяется внутри базового блока, под которым понимают линейный участок программы. Ниже приведена программа, содержащая три базовых блока, а на рис. 5.10 построены ЯПФ этих ББ.

```
        DO 1 I = 1, 100
ББ1     Z(I) = A(I)**2*X + B(I) * X + C(I)
        IF (I - 30) 2, 2, 3
ББ2   2 R1(I) = Z1*I
        R(I) = R1(I) * Z(I)
        GO TO 1                                          (5.8)
ББ3   3 R1(I) = Z1 * I
        R2(I) = Z2 * I
        R3(I) = Z3 * I
        R1 = R3(I)*Z(I)**3 + R2(I)*Z(I)**2 + R1(I)*Z(I)
```

1 CONTINUE

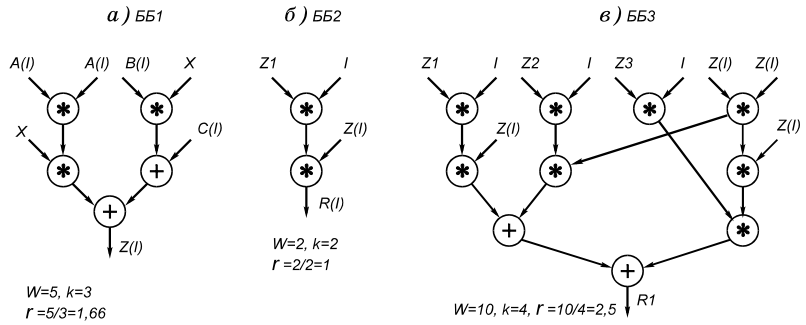


Рис.5.10. ЯПФ исходных базовых блоков

Если не учитывать длительность операций, то из рис.5.10 следует, что $w_{ББ3}>w_{ББ1}>w_{ББ2}$ (w — количество операций в ББ, k — число ярусов его ЯПФ) и $r_{ББ3}>r_{ББ1}>r_{ББ2}$. Это приводит к мысли, что параллелизм растет с увеличением размера ББ, следовательно, надо увеличивать длину ББ. Этого можно достигнуть двумя способами: путем объединения базовых блоков и путем развертки итераций цикла.

Рассмотрим обе возможности. Очевидно, цикл (5.8) можно разбить по параметру I на два независимых цикла (5.9) и (5.10).

```
      . . .
      DO 1 I = 1, 30
ББ4   Z(I) = A(I)**2*X + B(I)*X + C(I)
      R1(I) = Z1*I (5. 9)                          (5.9)
      R(I) = R1(I)*Z(I)
    1 CONTINUE

      . . .
      DO 1 I = 31, 100
ББ5   Z(I) = A(I)**2*X + B(I)*X + C(I)
      R1(I) = Z1*I                                 (5.10)
      R2(I) = Z2*I
      R3(I) = Z3*I
```

```
      R(I) = R3(I)*Z(I)**3 + R2(I)*Z(I)**2 + R1(I)*Z(I)
1     CONTINUE
```

В (5.9) блок ББ4 объединяет блоки ББ1 и ББ2, а в (5.10) блок ББ5 объединяет ББ1 и ББ3. Соответствующие ЯПФ представлены на рис. 5.11, а, б.

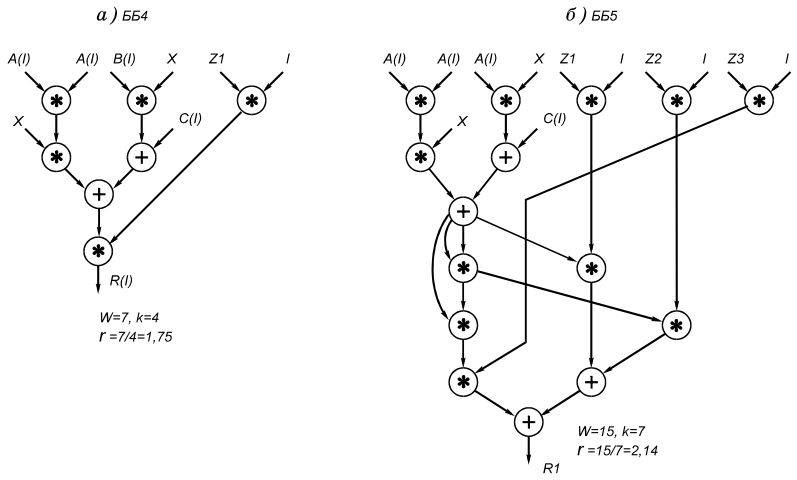


Рис. 5.11. ЯПФ укрупненных базовых блоков

Из рисунка следует, что ББ4 обладает лучшими характеристиками, чем ББ1 и ББ2, взятыми по отдельности, поскольку $r_{ББ4} > r_{ББ1,ББ2}$. Однако, для ББ5 дела обстоят по другому:

$w_{ББ5} = w_{ББ1} + w_{ББ3} = 5+10 = 15$
$k_{ББ5} = k_{ББ1} + k_{ББ3} = 3+4 = 7$,

поэтому и ускорение ББ5 занимает промежуточное значение: $r_{ББ1} > r_{ББ5} > r_{ББ3}$. Это означает, что для ББ5 улучшения характеристик не произошло. Однако, эксперименты показывают, что в большинстве случаев объединение блоков дает увеличение ускорения.

Цикл (5.11) является разверткой трех итераций цикла (5.9).

Соответствующая ЯПФ приведена на рис. 5.12. Ускорение возрастает более, чем линейным образом: $r_{ББ6} > 3 \cdot r_{ББ1}$. Правда, при подсчете операций в ББ6 индексные операции не учитывались.

```
      . . .
      DO 1 I = 1, 30, 3
ББ6   Z(I) = A(I)**2*X + B(I)*X + C(I)
      R (I) = Z1*I                                          (5. 11)
      R(I) = R1(I)*Z(I)
      Z(I+1) = A(I+1)**2*X + B(I+1)*X + C(I+1)
      R1(I+1) = X1 * (I+1)
      R(I+1) = R1(I+1) * Z(I+1)
      Z(I+2) = A(I+2)**2*X + B(I+2)*X + C(I+2)
      R1(I+2) = Z1*(I+2)
      Z(I+2) = R1(I+2)*Z(I+2)
    1 CONTINUE
```

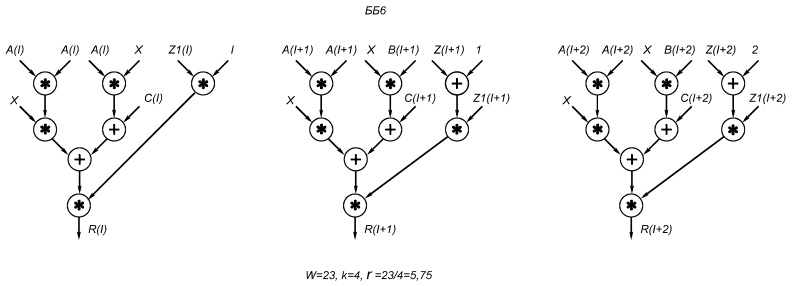


Рис. 5.12. ЯПФ развертки итераций

Развертка (5.11) включает не только параллелизм одного ББ, но и параллелизм трех смежных итераций, то есть в обращение вовлекается и векторный параллелизм, который, однако, рассматривается здесь как частный случай скалярного.

На рис. 5.13 точками представлена зависимость $r = f(w)$ для рассмотренных ранее ББ. Для получения более достоверных результатов проводились специальные исследования на большом объеме программного материала. Полученные результаты ограничены контуром на этом же рисунке. На основе рис. 5.13 с учетом вероятностных характеристик контура результатов можно получить следующую качественную зависимость:

$$r = a + b\,w, \qquad (5.\ 12)$$

где a и b — константы ($a \approx 1$, $b \approx 0{,}15$)

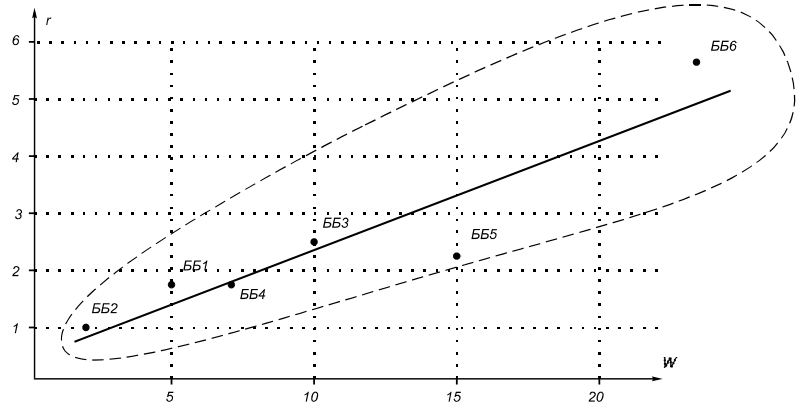


Рис.5.13 Рост ускорения в зависимости от размера базового блока

Таким образом, основной путь увеличения скалярного параллелизма программы — это удлинение ББ, а развертка — наиболее эффективный способ для этого. К сожалению, в большинстве случаев тело цикла содержит операторы переходов. Это препятствует как объединению ББ внутри тела цикла, так и выполнению развертки. Существуют частные способы преодоления этого препятствия. Один из них был продемонстрирован на примере разбиения программы (5.8) на два независимых цикла (5.9) и (5.10) благодаря тому, что оператор IF прямолинейно зависит от величины индекса цикла.

Более универсальный *метод планирования трасс* предложил в 80-е годы Фишер (Fisher, США) [22].

Рассмотрим этот метод на примере рис. 5.14, на котором представлена блок-схема тела цикла.

Как и прежде, в кружках представлены номера вершин, а

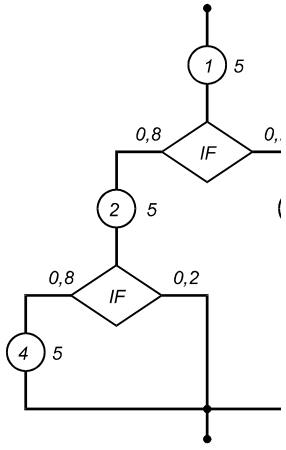


Рис. 5.14. Пример выбора

рядом — вес вершины (время ее исполнения); на выходах операторов переходов проставлены вероятности этих переходов.

Возможны следующие варианты исполнения тела цикла: 1-2-4, 1-2, 1-3. Путь 1-2-4 обладает наибольшим объемом вычислений (5+5+5) и является наиболее вероятным. Примем его в качестве главной трассы. Остальные пути будем считать простыми трассами.

Ограничимся рассмотрением метода планирования трасс только по отношению к главной трассе.

Если метод планирования трасс не применяется, то главная трасса состоит из трех независимых блоков, суммарное время выполнения которых будет в соответствии с (5.12) равным:

$$T_1 = 0{,}64\frac{3\cdot 5}{a+b\cdot 5} = 0{,}64\frac{3\cdot 5}{1+0{,}15\cdot 5} = 5{,}5$$

В методе планирования трасс предлагается считать главную трассу единым ББ, который выполняется с вероятностью 0,64. Тем не менее, при каждом выполнении этого объединенного блока проверяются условия выхода из главной трассы, и в случае необходимости выход осуществляется. Если переходов нет, то объединенный ББ выполняется за время

$$T_2 = 0{,}64\frac{3\cdot 5}{1+0{,}15\cdot 3\cdot 5} = 3$$

Таким образом, выигрыш во времени выполнения главной трассы составил $T_1 / T_2 = 1{,}8$ раз. В общем случае при объединении k блоков с равным временем исполнения w получаем:

$$\frac{T_1}{T_2} = (\frac{k\cdot w}{a+b\cdot w}) / (\frac{k\cdot w}{a+b\cdot k\cdot w}) = \frac{a+b\cdot kw}{a+b\cdot w} \xrightarrow[w\to\infty]{} k$$

При построении ЯПФ объединенного ББ и дальнейшем планировании команды могут перемещаться из одного исходного ББ в другой, оказываясь выше или ниже оператора перехода, что может привести к нарушению логики выполнения программы. Чтобы исключить возможность неправильных вычислений, вводятся компенсационные коды. Рассмотрим примеры (рис. 5.15)

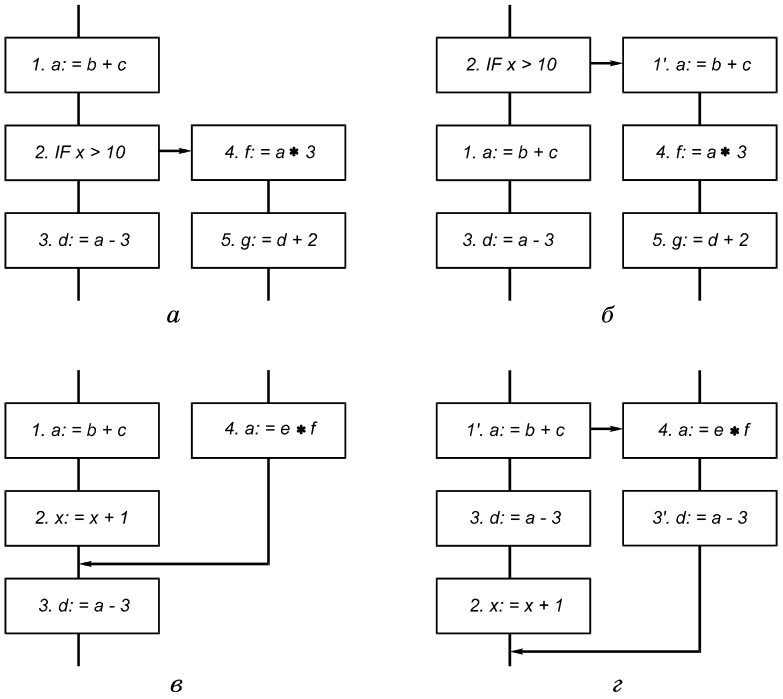


Рис.5.15. Способы введения компенсационных кодов

Пусть текущая трасса (рис. 5.15, а) состоит из операций 1, 2, 3. Предположим, что операция 1 не является срочной и перемещается поэтому ниже условного перехода 2. Но тогда операция 4 считает неверное значение *a*. Чтобы этого не произошло, компилятор водит компенсирующую операцию 1 (рис. 5.15, б).

Пусть теперь операция 3 перемещается выше IF. Тогда операция 5 считает неверное значение *d*. Если бы значение *d* не использовалось на расположенном вне трассы крае перехода, то перемещение операции 3 выше IF было бы допустимым.

Рассмотрим переходы в трассу извне. Пусть текущая трасса содержит операции 1, 2, 3 (рис. 5.15, в).

Предположим, что компилятор перемещает операцию 3 в положение между операциями 1 и 2. Тогда в операции 3 будет ис

пользовано неверное значение *a*. Во избежание этого, необходимо ввести компенсирующий код 3. (рис. 5.15, г).

Существует полный набор правил введения компенсационных кодов, которые используются в компиляторах для метода планирования трасс. Компенсационные коды увеличивают размер машинной программы, но не увеличивают числа выполняемых в процессе вычислений операций.

Задача переноса программного обеспечения наиболее просто решается, если суперскалярный МП строится на базе системы команд типа *CISC*, как в случае МП Пентиум, для которого без переделки пригодно все ПО, написанное для семейства МП *i*80×86. Однако, большинство суперскалярных МП строится на более перспективной системе команд типа *RISC*, например, семейство МП Power PC (совместная разработка фирм APPLE, IBM, MOTOROLA) и семейство Alpha (фирма DEC). В этом случае проблема переноса значительно усложняется.

Для системного ПО (ОС, компиляторы, редакторы и др.) перенос осуществляется относительно просто, поскольку объем системного ПО ограничен и для него существуют и доступны исходные программы на ЯВУ или ассемблере. В этом случае перенос осуществляется на основе прямой перекомпиляции в коды нового МП.

Объем прикладного ПО значительно больше, чем системного, причем, прикладные пакеты обычно существуют в виде двоичных кодов, а исходные тексты недоступны. В этом случае перенос может осуществляться на основе эмуляции. Под *эмуляцией* понимают преобразование команд исходной программы, представленной в двоичных кодах, в команды нового МП. Это преобразование выполняется в режиме интерпретации исходных команд непосредственно в процессе вычислений.

Пусть исходная программа в двоичных кодах представлена в системе команд CKi и ее требуется преобразовать в двоичный код программы в системе команд CKj. Тогда эмуляцией называется непосредственное выполнение исходной программы в режиме покомандной интерпретации, то есть каждая команда из CKi заменяется на одну или несколько команд из CKj, которые сразу же и выполняются. Эмулятор реализуется программно или микропро

граммно. Естественно, режим интерпретации значительно (в несколько раз) уменьшает скорость вычислений.

При переносе пакета, кроме непосредственной эмуляции кодов операции команд, нужно выполнить еще ряд условий, осложняющих эмуляцию:

1. Обычно пакет требует определенного (иногда, специфичного) набора ВнУ, библиотечных подпрограмм и функций, некоторых функций операционной системы, специализированного для пакета человеко-машинного интерфейса. Поэтому, если пакет переносится с ЭВМ с архитектурой Ai на ЭВМ с архитектурой Aj, то последняя должна удовлетворять всем требованиям пакета. Иногда это требует большой переработки пакета.

2. В процессе эмуляции должны быть отображены способы прерываний, способы вызова подпрограмм, система флагов и другие особенности исходной системы команд.

3. Обычно исходные программы представлены в системе команд типа *CISC*, которая использует небольшое число РОН (4...8). Такие программы обладают небольшим параллелизмом, поскольку из-за малого числа РОН в них внесены дополнительные зависимости по данным. В *RISC*-микропроцессорах число РОН равно 32 или больше, поэтому, чтобы увеличить параллелизм выполнений, динамически в процессе эмуляции производится переименование регистров.

Основной объем прикладных пакетов представлен в кодах МП семейств *i*80×86 и МС 680×0. Для перевода этих пакетов на суперскалярные МП Power PC и Alpha разработано большое количество эмуляторов.

Контрольные вопросы

1. В чем состоят особенности программного обеспечения для параллельных ЭВМ?
2. Перечислите механизмы преобразования структур данных в векторных языках.
3. Перечислите основные арифметико-логические операции векторных языков.
4. В чем отличие метода параллелепипедов от метода координат при автоматическом распараллеливании программ?

5. Какие особенности программ препятствуют их векторизации?
6. Охарактеризуйте структуру векторизующего компилятора.
7. Какова концептуальная модель вычислений в многотранспьютерной системе?
8. Опишите средства конфигурирования в Оккаме.
9. Охарактеризуйте основные подходы создания ЯВУ для транспьютеров. Каковы средства описания распределения ресурсов в них?
10. Каковы основные функции системного программного обеспечения в суперскалярных системах?
11. В чем состоят алгоритмы построения ЯПФ?
12. Что такое планирование? Опишите метод списочных расписаний.
13. Какие существуют методы повышения скалярного параллелизма?
14. В чем сущность метода планирования трасс Фишера?