ГЛАВА 2. ПРИНЦИПЫ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

Этапы подготовки любой задачи для параллельного исполнения таковы:

1. Разработка параллельного алгоритма решаемой задачи. Создаваемый алгоритм должен учитывать особенности параллельной аппаратуры и, кроме того, иметь наименьший объем вычислений среди других параллельных алгоритмов.
2. Разработка программы и ее оптимизация. Здесь наибольшее значение имеют способы разделения частей программы и данных на отдельные блоки и распределение этих блоков по вычислительным устройствам.
3. Планирование вычислений для конкретной аппаратуры. Только учет тонких особенностей параллельной аппаратуры позволяет реализовать тот потенциал эффективности, который создан на предыдущих этапах подготовки задачи.

2.1. Алгоритмизация

Рассмотрим некоторые свойства и особенности параллельных алгоритмов. Имеется большое разнообразие параллельных алгоритмов для решения одной и той же задачи. То обстоятельство, что алгоритм является параллельным, еще не гарантирует его эффективности. Это можно показать на основе *дискретного преобразования Фурье (ДПФ)* – центрального алгоритма в ЦОС [3]. Основными направлениями использования ДПФ является цифровая фильтрация и спектральный анализ.

ДПФ может быть вычислено тремя совершенно различными способами. Первым методом получения ДПФ является решение системы уравнений. Он хорош для понимания ДПФ, но слишком неэффективен. Другой метод – корреляция, основанный на выявлении известной волновой формы в другом сигнале. Третий метод – *быстрое преобразование Фурье (БПФ).* Конечно, все три метода дают одинаковый результат. Мы сравним только корреляцию и БПФ.

В методе корреляции вычисление одной точки в частотной области заключается в попарном умножении точек исследуемого временного сигнала и синусоиды (косинусоиды) заданной частоты с последующим суммированием этих произведений, что и показано в приводимом ниже уравнении:

$$X[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[n]\left(\cos(2\pi kn / N) - j\sin(2\pi kn / N)\right)$$

Здесь: $x[n]$ – комплексная амплитуда n-го отсчета во временной области, $n = 0,1,...N-1$; $X[k]$ – комплексная амплитуда отсчета с номером k в частотной области.

Для метода корреляции программа использует два вложенных цикла, каждый обрабатывает N точек. Полное время работы программы определяется выражением:

$$ВремяКорреляции = O(N^2)$$

Алгоритм БПФ выполняется за несколько шагов [3]. Первый шаг состоит в разделении N-точечного входного сигнала из временной области в N сигналов временной области. Второй шаг заключается в вычислении N частотных спектров, соответствующих этим N сигналам временной области. Наконец, N спектров синтезируются в один частотный спектр.

Для БПФ время выполнения операций во временной области пренебрежимо мало. Синтез в частотной области требует всего

$$ВремяБПФ = O(N*log_2N)$$

тактов, что в сотни раз меньше, чем на основе метода корреляции.

Разработка параллельных алгоритмов является сложным разделом вычислительной математики, но на этом пути уже получено много результатов. В таблице 2.1 приведена для примера часть этих результатов[16]. В ней n и m являются параметрами

структуры данных. Естественно, реальные результаты в сильной степени зависят от конкретной реализации параллельной ЭВМ.

Таблица 2.1.Порядок времени вычислений для различных алгоритмов

№ пп	Наименование алгоритма	Порядок времени вычислений	Число процессоров
	Алгебра		
2	Решение треугольной системы уравнений, обращение треугольной матрицы	$0(\log^2 n)$	$0(n^{\omega} / \log n)$
3	Вычисление коэффициентов характеристического уравнения матрицы	$0(\log^2 n)$	$0(n^4 / \log^2 n)$
4	Решение системы линейных уравнений, обращение матрицы	$0(\log^2 n)$	$0(n^4 / \log^2 n)$
5	Метод исключения Гаусса	$0(\log^2 n)$	$0(n^{\omega+1})$
6	Вычисление ранга матрицы	$0(\log^2 n)$	полиномиальное число
7	Подобие двух матриц	$0(\log^2 n)$	
8	Нахождение LU-разложения симметричной матрицы	$0(\log^3 n)$	$0(n^4 / \log^2 n)$
	Комбинаторика		
1	ε — оптимальный рюкзак, n — размерность задачи	0(log n log (n/ε))	$0(n^3 / \varepsilon^2)$
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	$0(\log^2 n)$	

4	Транзитивное замыкание	$0(\log^2 n)$	$0(n^{\omega} / \log n)$

2.2. Оптимизация программ

Будем называть *оптимизирующими трансформациями* такие преобразования программы, которые повышают ее рабочие свойства за счет всех имеющихся возможностей. Под «имеющимися возможностями» понимается не только распараллеливание, но и все средства, которые используются для улучшения программ для последовательных машин. Эти средства также будут рассматриваться в последующих разделах, поскольку без них распараллеливание теряет эффективность.

Чтобы выполнить оптимизацию программы, надо решить, на какой части программы будет выполняться оптимизация и какая оптимизация нужна, затем удостовериться, что трансформация не меняет смысла программы, и только тогда преобразовать программу.

Объектами приложения преобразований являются:

- Арифметические выражения и базовые блоки. *Базовый блок* – это участок программы, в котором отсутствуют переходы;
- Внутренний цикл. В таких циклах как правило сосредоточен главный объем работ. Они имеют обычно небольшой размер, но большое число повторений;
- Совершенное гнездо циклов. Такое гнездо содержит ряд вложенных циклов. Гнездо называется совершенным, если все циклы имеют одно и те же тело. Совершенный цикл удобен для преобразований;
- Произвольное гнездо циклов;
- Процедура. Некоторые преобразования, в частности, связанные с доступом к памяти, дают лучшие результаты, если они прилагаются сразу к целой процедуре. Часто в литературе процедурные оптимизации относят к глобальному уровню.

Наибольший интерес представляют трансформации за пределами базового блока.Они и будут рассмотрены дальше.

Оптимизации для последовательных ЭВМ в первую очередь касаются обращений к памяти. В частности, дистанция по адре

сам памяти при последовательном к ней обращениям оказывает большое влияние на эффективность. Эта дистанция называется страйдом (stride). Если цикл обращается в память через каждые 4 элемента, это цикл со страйдом-4. Страйд-1 означает, что цикл обращается в память за каждым следующим по адресу элементом. Страйд-1 ниболее желателен, поскольку он делает максимальной локальность, а, следовательно, и эффективность кэша, буфера трансляции адресов TLB и систем страничной памяти. Это также исключает конфликты в банках памяти векторных машин.

Оптимизации имеют место на трех различных ступенях, соответствующих трем различным представлениям программы: высокоуровневый промежуточный язык, низкоуровневый промежуточный язык и объектный код.

Главным объектом внимания для высокоуровневых архитектур являются циклы. Сначала выполняется последовательность преобразований, которая пытается конвертировать цикл в форму, которая более пригодна для оптимизации: где возможно создаются совершенные гнезда циклов из несовершенных, индексные выражения переписываются в терминах индуктивных переменных (переменные внутри индексных выражений) и так далее. Затем итарации цикла реорганизуются, чтобы сделать максимальным параллелизм и локальность. После этой обработки организуются результирующие циклы, чтобы уменьшить накладные расходы на организацию циклов.

После оптимизации цикла программа преобразуется в промежуточный язык. Многие оптимизации, которые использовались для ЯВУ, снова применяются к этой программе, чтобы исключить неэффективность выражений, сгенерированных из высокоуровневых конструкций. Дополнительно выполняется оптимизация индуктивных переменных, которая часто важна для однопроцессорных машин. В конце выполняется оптимизация вызовов процедур.

Генератор кода преобразует программу из промежуточного языка в ассемблер. Эта фаза ответственна за выбор и планирование команд и распределение регистров. На этой фазе компи

лятор использует низкоуровневые оптимизации для дальнейшего повышения эффективности. Наконец, ассемблер генерирует объектный код, объектные файлы линкуются и выполняются.

Далее описаны трансформации различного назначения. Все описанные ниже преобразования могут быть реализованы как вручную при написании некоторой программы, так и вложены в компилятор для автоматического преобразования на множестве программ.

Поскольку базой для проведения большинства оптимизаций является информация о зависимостях между операторами по данным или управлению, то в следующем разделе прежде всего будут рассмотрены методы анализа зависимостей.

2.2.1. *Анализ зависимостей*

В последовательных программах команды выполняются в порядке их расположения в программе или осуществляется переход по адресам ветвлений. В параллельных программах этот порядок в первую очередь определяется *зависимостями между операторами по данным или по управлению.* Среди различных форм анализа, используемых для оптимизирующих компиляторов, анализ зависимостей наиболее важен. Далее дается терминология и соответствующая теория [17,18,19]. При этом сама техника определения зависимостей не приводится.

Зависимость есть отношение между двумя вычислениями, ограничивающее порядок их выполнения. Анализ зависимостей вскрывает этот порядок, чтобы определить, меняет ли некоторое преобразование смысл вычислений.

Типы зависимостей. Имеется два вида зависимостей: зависимости по управлению и зависимости по данным. Между двумя предложениями 1 и 2 имеется зависимость по управлению, обозначаемая как S_1 (c) $\rightarrow$ S_2, если предложение S_1 определяет, будет ли выполняться S_2. Например:

```
1   if (a = 3) then
2         b = 10
```

```
end if
```

Описанные ниже три типа зависимостей по данным уже введены в разделе 1.1, здесь будут введены обозначения для них, поскольку эти обозначения будут использоваться при дальнейшем изучении материала.

Если некоторое предложение S_3 записывает значение, которое затем читается S_4, то S_4 должно выполняться только после S_3. Такая зависимость по данным называется прямой (или потоковой - flow dependence) и обозначается $S_3 \rightarrow S_4$, например:

```
3   a = c*10
4   d = 2*a + c
```

S_6 имеет обратную зависимость от S_5 (antidependence), обозначаемую $S_5 \leftarrow S_6$, когда S_6 записывает переменную, которая читается S_5:

```
5   e = f*4 + g
6   g = 2*h
```

Обратная зависимость не так сильно ограничивает исполнение, как прямая зависимость по данным. Чтобы избежать ее, надо задержать исполнение S_6. Для этого достаточно ввести две ячейки памяти g_5 и g_6, чтобы хранить g для S_5 и принимать значение из S_6.

Выходная зависимость имеет место, когда оба предложения записываю ту же самую переменную:

```
7   a = b*c
8   a = d + e
```

Снова, как и с обратной зависимостью, использование дополнительных переменных может позволить предложениям выполняться одновременно.

Анализ зависимостей цикла (Loop Dependence Analysis). По сравнению с базовым блоком анализ зависимостей в цикле более сложная задача. В цикле каждое предложение выполняется много раз и для многих преобразований необходимо описать зависимости, которые существуют между итерациями, называемые межитерационными.

Простой пример межитерационной зависимости представлен ниже. Внутри одной итерации цикла между S_1 и S_2 не имеется никаких зависимостей, однако, имеется зависимость между двумя смежными итерациями. Когда $i = k$, S_2 читает значение $a[k\text{-}1]$, записанное S_1 в итерации k-1.

```
   do i = 2, n
1      a[i] = a[i] + c
2      b[i] = a[i–1] * b[i]
   end do
```

Чтобы выявить зависимости в гнезде циклов, необходимо определить, может ли любая из итераций записывать значение, которое читается или записывается в любой другой итераций.

Ниже представлено обобщенное совершенное гнездо циклов. Тело гнезда циклов читает и записывает элементы m-мерного масива a. Шаг индексации является единичным, поскольку это упрощает определение зависимостей. Если шаг не единичный, следует нормализовать циклы, то есть привести их к единичному шагу.

```
do i1 = l1, u1
   do i2 = l2, u2
      ...
         do id = ld, ud
1              a[f1(i1,…,id),…,fm(i1,…,id)] = …
2              … = a[g1(i1,…,id),…,gm(i1,…,id)]
         end do
   end do
end do
```

Итерации могут быть уникально обозначены вектором из d элементов $I=(i_1,\ldots,i_d)$, где каждый индекс находится внутри итерационного диапазона своего цикла в гнезде циклов. Самый внешний цикл соответствует самому левому индексу.

Покажем, какие межитерационные зависимости существуют между двумя обращениями к a, и опишем их. Ясно, что обращение в итерации J может зависеть только от другого обращения в итерации I, которое было выполнено перед ним, но не после. Можно формализовать обозначение «перед» значком $\subset$, тогда:

$$I \subset J \text{ iff } \exists p: (i_p < j_p \wedge \forall q < p: i_q = j_q)$$

Обращение в той же самой итерации J зависит от обращения в итерации I тогда и только тогда, когда по крайней мере одно обращение есть запись и

$$I \subset J \wedge \forall p: f_p(I) = g_p(J)$$

Другими словами, зависимость имеется, когда значения индексных выражений те же самые в разных итерациях. Если не существует таких I и J, два обращения независимы для всех итераций цикла. В случае выходной зависимости, вызванной той же самой записью в различных итерациях, условие простое: $f_p(I) = f_p(J)$.

Для примера предположим, что мы пробуем описать поведение цикла из рис.2.1а.

```
do  i = 2, n
  do j = 1, n–1
        a[i,j] = a[i,j] + a[i –1,j+1]
   end do
end do
        (a) {(1, –1)}

do i = 2, n
  do j = 1, n–1
        a[j] = (a[j] + a[j –1] + a[j+1] ) /3
   end do
```

end do

(b) {(0,1), (1,0), (1,–1)}

Рис. 2.1. Дистантные вектора

Каждая итерация внутреннего цикла записывает элемент $a[i,j]$. Зависимость имеет место, если любая другая итерация читает или записывает тот же самый элемент. В этом случае имеется много пар итераций, которые зависят одна от другой. Рассмотрим итерацию $I = (1,3)$ и $J = (2,2)$. Итерация I выполняется первой и записывает значение $a[1,3]$. Это значение читается в итерации J, следовательно имеется зависимость между итерациями I и J. Расширяя нотации для зависимостей, мы запишем $I \rightarrow J$.

Когда $X \Rightarrow Y$, мы определяем дистанцию зависимости (dependence distance) как $Y - X = (y_1 - x_1,\ldots,y_d - x_d)$.

Легальный (допустимый) *дистантный вектор* должен быть лексикографически положительным, то есть первый ненулевой элемент дистантного вектора должен быть положительным. Негативный элемент в дистантном векторе означает, что зависимость в соответствующем цикле имеется на итерации с более высоким номером. Если первый ненулевой элемент был отрицательным, это будет означать зависимость на будущей итерации.

Дистантный вектор описывает зависимости между итерациями, а не между элементами массива. Например, гнездо циклов, которое обрабатывает одномерный массив a на рис.2.1b, имеет зависимости, описываемые набором двухэлементных векторов {(0,1),(1,0),(1,-1)}.

В некоторых случаях невозможно точно определить дистанции зависимости на этапе компиляции или эти дистанции могут варьироваться от итерации к итерации, но имеется достаточно информации для частичной характеристики зависимости. Для описания таких зависимостей используется *вектор направления* (direction vector).

Для зависимости $I \Rightarrow J$ вектор направления определяется так: $W = (w_1,\ldots,w_d)$, где

W_p равен < if $I_p < J_p$
W_p равен = if $I_p = J_p$
W_p равен > if $I_p > J_p$

Можно использовать обобщенный термин «вектор зависимости» для обоих типов векторов. На рис.2.1 дистантный вектор для цикла (а) есть (< , >), и вектор направления для цикла (в) есть {(= , <) , (< , =) , (< , >)}. Заметим, что вход вектора направления < соответствует входу дистантного вектора, который больше, чем 0.

Поведение зависимостей цикла описывается рядом векторов зависимости для каждой пары возможно конфликтующих обращений. Это можно объединить в суммарный вектор направления цикла за счет потери некоторой информации (и потенциально – для оптимизации). Зависимости цикла на рис 2.1в могут быть суммированы как (≤ , *).

При анализе межитерационнных зависимостей важен следующий вопрос: как компилятор определяет, относятся ли два обращения в разных итерациях к одному и тому же элементу массива. Это делается путем анализа линейных индексных выражений. Если индексные выражения слишком сложны для анализа, считают, что предложения зависят друг от друга.

2.2.2. *Трансформации общего назначения*

В дальнейшем рассмотрен ряд преобразований общего характера, используемых для чистки программ как для последовательных, так и для параллельных машин. Для каждого преобразования представлены два названия: на русском и английском языках, русские названия приведены согласно [18].

Ряд классических трансформаций основан на анализе потока данных через программные переменные, который используются при выполнении программы.

Понижение силы операций (Loop-Based Strength Reduction).В этом случае некоторые выражения в цикле заме

щаются эквивалентными по результату вычислений, но имеющими меньший вычислительный вес. Рис.2.2а представляет преобразованную версию цикла, в которой умножение заменено на сложение.

```
do i = 1,  n
     a[ i ] = a[ i ] + c*i
end do
```

(a) исходный цикл

```
T = c
do i = 1,  n
     a[i] = a[i] + T
     T = T + c
end do
```

(b) после понижения силы оперций

Рис.2.2 Пример понижения силы операций

Многие операции, например, такие как c*i, c^i, $(-1)^i$ можно заменить сложением или другими более простыми действиями.

Обычно индуктивные переменные, чье значение используется рядом итераций, являются индексными перемеными цикла. Наиболее полезным случаем понижения силы операций является как раз замена умножения на сложение при вычислении очередного значения индекса.

Чистка цикла (Loop-Invariant Code Motion). Если в цикле обнаружен код, не изменяющийся в процессе выполнения итераций, он может быть вынесен за пределы цикла.

Рис.2.3 представляет пример, в котором вызовы длительных в вычислительном отношении трансцедентных функций выносятся за пределы цикла.

```
do i = 1, n
  a[i] = a[i] + sqrt(x)
 end do
```

(a) Исходный цикл

```
if (n > 0) C = sqrt(x)
do i = 1, n
  a[i] = a[i] + C
```

```
end do
```
(b) После перемещения

Рис.2.3. После перемещения кода

Хотя вынесение кода иногда называют подъемом кода (code hoisting), подъем есть более общий термин, относящийся к любому преобразованию, которое перемещает вычисления на более раннюю точку программы. К таким преобразованиям относятся, например, предварительное вычисление выражений, заблаговременная загрузка данных из памяти.

Втягивание констант (Constant Propogation). Втягивание констант является одной из наиболее важных оптимизаций, которую может выполнять компилятор. Типичные программы содержат много констант, втягивая их на протяжении программы, компилятор может выполнить существенный объем предвычислений. Более важно то, что втягивание открывает путь другим оптимизациям. В дополнение к очевидным возможностям, таким как исключение мертвого кода, втягивание констант влияет на оптимизации циклов, поскольку константы часто появляются в их индуктивном диапазоне. Зная диапазон цикла, компилятор может использовать в цикле наиболее эффективные преобразования.

```
n = 64
c = 3
do i = 1, n
  a[i] = a[i] + c
end do
```
(a) Исходный код

```
do i = 1, 64
  a[i] = a[i] + 3
end do
```
(b) После втягивания констант

Рис.2.4. Втягивание констант

Рис.2.4 представляет простой пример втягивания констант. Здесь результирующий цикл может быть преобразован в единственную векторную операцию, поскольку цикл имеет ту же длину, что и аппаратный векторный регистр.

Подстановка вперед (Forward Substitution). Подстановка вперед есть обобщенный случай втягивания констант. В этом случае использование переменной замещается определяющим ее выражением. Замена может изменять зависимости между переменными или улучшать анализ индексных выражений в цикле.

Например, цикл на рис.2.5a не может быть параллелизован, поскольку используется неизвестный элемент массива *a*. После подстановки вперед, как показано на рис.2.5b, индексное выражение определяется через индексы цикла, и это прямолинейно позволяет выразить цикл через параллельную редукцию.

Подстановка вперед обычно выполняется на индексных выражениях массивов одновременно с нормализацией цикла. Индексные выражения должны быть линейной функцией индуктивных переменных, тогда техника анализа индексных выражений будет работать эффективно.

```
np1 = n + 1
do i = 1,  n
  a[np1] = a[np1] + a[i]
end do
```

(a) Исходный код

```
do all i = 1, n
  a[n+1] = a[n+1] + a[i]
end do
```

(b) После подстановки вперед

Рис.2.5. Подстановка вперед

Исключение избыточности (Redundancy Elimination). Существует много оптимизаций, предназначенных для выявления и удаления избыточности. Это относится, например, к удалению кода, инвариантного по отношению к циклу. Здесь речь

пойдет о недостижимых или бесполезных операциях, а также об исключении общих подвыражений.

Вычисление недостижимо, если оно никогда не выполняется. Недостижимый код создается программистом или преобразованием, которое оставляет после себя коды «сироты». Например, если уже известно, что условное предложение истинно или ложно, одна ветвь условного предложения никогда не будет выполняться и соответствующий код можно исключить. Другим общим источником недостижимого кода является цикл, который не выполняет ни одной итерации.

Во многих случаях ряд вычислений будет содержать идентичные подвыражения. Избыточность может возникнуть как в пользовательском коде, так и в адресных вычислениях, сгенерированных компилятором. Компилятор может однажды вычислить значение подвыражения, сохранить его и затем повторно его использовать.

Исключение общих подвыражений есть важное преобразование и выполняется почти универсально. Однако, здесь надо следить за ценой исключения. Если запоминание промежуточных значений вызывает дополнительный разброс адресов памяти, такое преобразование понижает эффективность программы.

2.2.3 Трансформации реорганизации циклов

В этом разделе будут рассмотрены трансформации, которые изменяют порядок выполнения гнезда или гнезд циклов. Эти преобразования используются, чтобы увеличить параллелизм и локальность обращения к памяти.

Наиболее просто эти преобразования применяются к совершенным гнездам циклов, поэтому часто перед реорганизацией несовершенное гнездо циклов преобразуется в совершенное, если такое преобразование можно выполнить.

Перед преобразованием предварительно надо исследовать зависимость итераций цикла, чтобы установить, может ли цикл выполняться параллельно. Очевидным случаем является ситуация, когда все дистанции зависимости для цикла равны 0 (direc

tion =). Это означает, что между итерациями цикла нет зависимостей. На рис.2.6a дистантный вектор для цикла равен (0, 1), то есть внешний цикл параллелизуем.

```
do i = 1, n
   do j = 2, n
     a[i,j] = a[i, j–1] + c
   end do
end do
```

(a) Внешний цикл параллелизуем

```
do i = 1, n
   do j=1, n
     a[i,j] = (a[i–1, j] + a[i–1, j+1 ]
   end do
end do
```

(b) Внутренний цикл параллелизуем

Рис.2.6. Влияние зависимостей на распараллеливание цикла

В более общем случае *p*-ый цикл в гнезде циклов параллелизуем, если для любой дистанции вектор:

$$v_p = 0 \vee \exists q < p : v_p > 0$$

На рис.2.6b дистантный вектор равен {(1,0),(1,-1)}, значит, внутренний цикл параллелизуем. Оба обращения с правой стороны выражения читают элементы из строк *i*-1, которые были записаны на предыдущей итерации внешнего цикла. Поэтому элементы строки *i* могут быть вычислены и записаны в произвольном порядке.

Перестановка циклов (Loop Interchange). При перестановке циклов обычно внешний цикл совершенного гнезда циклов перемещается на место одного из внутренних. Такая перестановка является одним из наиболее мощных преобразований и может улучшить характеристики программы по разным причинам. Перестановка может быть предназначена для того, чтобы:

• Ввести векторизацию заменой внутреннего зависимого цикла на внешний независимый;

• Повысить уровень векторизации путем перевода внутрь более векторизуемого внешнего цикла;
• Уменьшить шаг между параллельными итерациями, желательно, до единицы;
• Повысить зернистость параллелизма, перемещая внутренний параллельный цикл на место внешнего непараллельного.

Однако, необходимо следить, чтобы эти возможные выгоды не отменили друг друга. Например, обмен, который улучшает использование регистров, может изменить страйд-1 доступа к памяти на страйд-n, который сильно ухудшает общие характеристики вычислений из-за промахов кэша. Рис.2.7 демонстрирует значительное влияние различных страйдов на скорость вычислений:

```
double precision a[*]

do i = 1, 1024*stride,  stride
  a[i] = a[i] + c
end do
```

(a) Цикл с изменяемым страйдом

Страйд	Промахи КЭШ	Промахи TLB	Относительная скорость %
1	64	2	100
2	128	4	83
4	256	8	63
8	512	16	40
12	768	24	28
16	1024	32	23
64	1024	128	19
256	1024	512	12
512	1024	1024	8

(b) Влияние страйда

Рис.2.7. Пример влияния страйда на характеристики цикла, приведенного выше

На рис.2.8a внутренний цикл обращается к массиву со страйдом *n* (в Фортране обращение к массиву производится по столбцам). Замена циклов позволяет преобразовать внутренний цикл к страйду 1, как показано на рис.2.8b:

```
do i = 1, n
   do j = 1, n
     total[i]=total[i] + a[i,j]
   end do
end do
```

(a) Исходный цикл

```
do j = 1, n
   do i = 1, n
     total[i]=total[i] + a[i,j]
   end do
end do
```

(b) Гнездо с переставленными циклами

Рис.2.8. Перестановка циклов

Для больших массивов, в которых только часть столбца помещается в кэше, эта возможность сильно уменьшает количество промахов кэша.

Для векторных архитектур преобразованный цикл дает возможность векторизации благодаря исключению зависимостей на *total*[*i*] во внешнем цикле.

Перестановка циклов допустима, когда измененные зависимости легальны и границы циклов могут быть изменены**.** Если два цикла *p* и *q* в совершенном гнезде циклов *d* переставляются, каждый вектор зависимости в исходном гнезде циклов $V = (v_1,\ldots,v_p,\ldots,v_q,\ldots,v_d)$ становится $V' = (v_1,\ldots,v_q,\ldots,v_p,\ldots,v_d)$ в преобразованном гнезде циклов. Если V’ является лексикографически положительным, тогда отношения зависимости исходного цикла удовлетворены. Изменение границ цикла выполняется прямолинейно, когда границы пространства итераций прямоугольны, как в гнезде циклов на рис.2.8. В этом случае границы внутреннего цикла независимы от индексов, охватывающих

цикл, и обе могут быть просто переставлены. Когда пространство итераций не прямоугольно, вычисление границ более сложно.

Обращение цикла (Loop Reversal). Обращение изменяет направление, в котором цикл пересекает свое итерационное пространство. Оно часто используется в соединении с другими преобразованиями пространства итераций, поскольку оно изменяет вектора зависимости.

Рис.2.9 показывает, как обращение может открыть возможность для перестановки циклов: исходное гнездо циклов (a) имеет дистантный вектор (1,-1), который препятствует перестановке, поскольку результирующий дистантный вектор (-1, 1) не является лексикографически положительным. Обращенное гнездо (b) может быть легально переставлено.

```
do i = 1, n
   do j = 2, n
     a[i,j]=a[i–1, j+1] + 1
   end do
end do
```

(a) Исходное гнездо циклов: дистантный вектор (1,-1). Перестановка невозможна

```
do i=1, n
   do j=n, 1, –1
     a[i,j]=a[i–1 ,j+1] + 1
   end do
end do
```

(b) Внутренний цикл обращен: дистантный вектор (1,1). Циклы могут быть переставлены

Рис.9. Обращение цикла

Распределение цикла (Loop Distribution). Распределение цикла разбивает один цикл на несколько. Каждый из новых циклов имеет то же пространство итераций, что и исходный цикл, но содержит свое подмножество предложений исходного цикла.

Распределение позволяет: создать совершенные гнезда циклов; получить подциклы с меньшими зависимостями; улучшить локальность кэша команд и TLB за счет сокращения тела цик

лов; уменьшить требования к памяти благодаря уменьшению размеров массивов.

Рис.2.10 представляет пример, на котором распределение цикла удаляет зависимости и позволяет частям цикла выполняться параллельно.

```
do i = 1 ,n
  a[i] = a[i] + c
  x[i+1] = x[i]*7 + x[i+1] + a[i]
end do
```

(a) Исходный цикл

```
do i = 1, n
  a[i] = a[i] + c
end do all
do i = 1 ,n
  x[i+1] = x[i]*7 + x[i+1] + a[i]
end do
```

(b) После распределения цикла

Рис.2.10. Распределение цикла

Распределение применимо к многим циклам, но все предложения, принадлежащие циклу с зависимостями, должны быть размещены в том же самом цикле. Если $S_1 \Rightarrow S_2$ в исходном цикле, тогда цикл, содержащий S_1, обязан предшествовать циклу с S_2.

Слияние циклов (Loop Fusion). Преобразование, обратное распределению, есть слияние. Оно улучшает характеристики благодаря: уменьшению накладных расходов на организацию цикла; повышению параллелизма команд; улучшению локальности регистров, векторов, кэша данных, TLB, страниц.

На рис.2.10 распределение позволяет параллелизовать часть цикла. Однако, слияние двух циклов повышает локальность регистров и кэша, поскольку $a[i]$ нужно загрузить только один раз. Слияние также повышает параллелизм команд, так как повышается отношение доли команд с плавающей точкой по отношению к целочисленным командам. Кроме того, до двух раз уменьшаются накладные расходы цикла. Для большого n рас

пределенный цикл будет быстрее выполняться на векторной машине, а цикл со слиянием – на суперскалярной.

Два цикла, предназначенные для слияния, должны иметь те же самые границы цикла. Два цикла с одинаковыми границами могут быть слиты, если не существует предложения S_1 в первом цикле и предложения S_2 во втором цикле таких, что они зависимы в слитом цикле следующим образом S_1 (<)→ S_2. Это означает, что S_1 выполняется перед S_2. После слияния эти операции выполняются вместе, что и приводит к ошибке.

2.2.4. *Трансформации, реструктуирующие цикл*

В этом разделе описаны трансформации, которые изменяют структуру цикла, но сохраняют неизменным относительный порядок вычислений в теле цикла.

Развертка циклов (Loop Unrolling). Развертка повторяет тело цикла *и* раз и выполняет итерации с шагом *и* вместо шага 1. Переменная *и* называется коэффициентом развертки. Развертка улучшает параметры благодаря: уменьшению накладных расходов на организацию цикла; повышению параллелизма команд; улучшению локальности регистров, кэша, TLB.

На рис.2.11 представлены все три варианта выигрыша. Накладные расходы цикла уменьшены наполовину, поскольку тест и ветвление осуществляются после двух итераций. Параллелизм команд повышен, поскольку второе присваивание может быть выполнено во время сохранения первого результата и коррекции переменной цикла.

```
do i = 2, n-1
  a[i] = a[i] + a[i – 1] ∗ a[i+1]
end do
```

(a) Исходный цикл

```
do i = 2, n -2, 2
  a[i] = a[i] + a[i – 1] ∗ a[i+1]
  a[i+1] = a[i+1] + a[i] ∗ a[i+2]
end do
if (mod (n–2, 2) =1) then
```

```
  a[n–1] = a[n–1] + a[n – 2] ∗ a[n]
end if
```

(b) Развернутый цикл

Рис.2.11. Развертка циклов

Если элементы массивов записываются в массивы, локальность регистров возрастет, поскольку $a[i]$ и $a[i+1]$ используются дважды в теле цикла, уменьшая число загрузок с 3-ех до 2-ух на одну итерацию.

Если используемая машина имеет двух- или многословные загрузки, развертка часто позволяет несколько загрузок объединить в одну.

Предложение *if* в конце рис.2.11b называется эпилогом (loop epilogue), который необходимо сгенерировать, если на этапе компиляции неизвестно, является ли число итераций кратным u. Если $u > 2$, сам эпилог является циклом.

Развертка имеет то преимущество, что она применима к любому циклу и дает улучшение как на вычислениях низкого, так и высокого уровня.

Большинство компиляторов для высокопроизводительных ЭВМ производят развертку по крайней мере внутреннего цикла гнезда циклов. Развертывание внешних циклов не является универсальной операцией.

Программный конвейер (Software Pipelining). Другим методом для улучшения параллелизма на уровне команд является *программный конвейер*. В аппаратном конвейере исполнение команды разбито на этапы, которые выполняются соответствующим оборудованием. При заполненном конвейере за 1 такт выполняется 1 команда. В программном конвейере операции одиночного цикла разбиты на S ступеней и очередная итерация конвейера выполняет ступень 1 из итерации i , ступень 2 из итерации i-1 и так далее. Перед циклом необходимо сгенерировать стартовый код, чтобы инициализировать конвейер для первых S-1 итераций, а после цикла нужно сгенерировать код для завершения последних S-1 итераций.

В разделе 2.4 этой главы программный конвейер рассмотрен более подробно, поскольку этот вид трансформации является одним из основных источников параллелизма для многих приложений.

Соединение циклов (Loop Coalescing). Эта оптимизация объединяет гнездо циклов в единственный цикл с новыми индексами, вычисленными из единственной результирующей индексной переменной. Соединение может улучшить планирование цикла на параллельной машине и также уменьшить накладные расходы цикла.

```
do all i = 1, n
  do all j = 1, m
     a[i,j] = a[i,j] + c
  end do all
end do all
```

(a) Исходный цикл

```
do all T=1, n*m
  i = ((T – 1)/ m) * m + 1
  j = MOD(T-1, m) + 1
  a[i,j] = a[i,j] + c
end do all
```

(b) Соединение циклов

Рис.2.12. Соединение циклов

На рис.2.12a, например, если n и m немного больше, чем число процессоров P, тогда никакой из циклов не планируется так же хорошо, как внешний параллельный цикл, поскольку выполнение последних $n - P$ итераций будет занимать столько же времени, как и первые P. Соединение двух циклов гарантирует, что P итераций могут быть выполны всегда, кроме последних ($nm \bmod P$) итераций, как показано на рис.2.12b.

Соединение само по себе всегда легально, поскольку оно не изменяет порядок итераций цикла. Итерации соединенного цикла могут быть параллелизованы, если все исходные циклы были параллелизуемы.

Разгрузка цикла (Loop Peeling). Когда цикл разгружен, то небольшое число итераций удалено из начала или конца цикла и выполняется отдельно. Если разгружается только одна итерация, в общем случае код этой итерации может быть вложен в условное предложение. При большом количестве итераций нужно ввести отдельный цикл. Разгрузка имеет две цели: удаление зависимостей, созданных первой и последней итерацией, что позволяет параллелизацию; и выравнивание итерационного управления смежных циклов, чтобы позволить слияние.

```
do  i = 2, n
  b[i] = b[i] + b[2]
end do all
do all i = 3 ,n
  a[i] = a[i] + c
end do all
```

(a) Исходный цикл

```
if  (2 < = n) then
   b[2] = b[2] + b[2]
end if
do all i = 3, n
  b[i] = b[i] + b[2]
  a[i] = a[i] + c
end do all
```

(b) После выгрузки одной итерации из первого цикла и слияния результирующих циклов

Рис.2.13. Разгрузка циклов

Цикл на рис.2.13a не параллелизуем, поскольку существует потоковая зависимость между итерациями $i = 2$ и $i = 3 \ldots n$. Выгрузка первой итерации позволяет параллелизовать остаток цикла и выполнить слияние со следующим циклом, как показано на рис.2.13b. Поскольку разгрузка просто разбивает цикл на секции без изменения порядка итераций, она может быть применима для любых циклов.

Нормализация цикла (Loop Normalization). Нормализация конвертирует циклы так, чтобы начальное значение индуктив

ной переменной было 1 (или 0), а шаг индексации итераций равнялся 1. Это преобразование может предоставить возможности для слияния и упрощения анализа межцикловых зависимостей, как показано на рис.2.14. Наиболее важной возможностью, которую дает нормализация, является возможность применения компилятором тестов анализа индексных выражений, которые как правило требуют нормализованного диапазона итераций.

```
do  i = 1, n
  a[i] = a[i] + c
end do
do  i = 2, n+1
  b[i] = a[i –1] + b[i]
end do
```

(a) Исходный цикл

```
do  i =1, n
  a[i] = a[i] + c
end do
do  i =1, n
  b[i+1] = a[i] + b[i+1]
end do
```

(b) После нормализации. Два цикла могут быть слиты.

Рис.2.14. Нормализация циклов

Распознавание редукций (Reduction Recognition). Редукция это операция, вычисляющая скалярное значение массива. Общеизвестные редукции вычисляют сумму либо максимальное значение элементов массива. На рис.2.15a сумма элементов аккумулируется в скаляре S. Вектор зависимости для цикла равен (1) или (<). Хотя цикл с вектором направления (<) должен выполняться последовательно, редукция может быть параллелизована, если выполняемая операция ассоциативна. Коммутативность обеспечивает дополнительные возможности для реорганизации.

На рис.2.15b редукция была векторизована путем использования векторных сложений (внешний цикл DO ALL), чтобы вы

числить TS; конечный результат вычисляется из TS с помощью скалярного цикла.

Для полукоммутативных и полуассоциативных операций, подобных умножению с плавающей точкой, значимость преобразования зависит от языковой семантики и склонности программиста.

Максимальный параллелизм достигается при вычислении редукции по дереву: пары элементов суммируются, затем суммируются пары результатов и так далее. При этом число последовательных шагов уменьшается от *O(n)* до *O(logn)*.

```
do  i =1, n
   s = s + a[i]
end do
```

(a) Цикл суммирования с редукцией

```
real TS[64]
TS[1: 64] = 0.0
do  TI =1, n,  64
  TS[1: 64]=TS[1: 64] + a[TI: TI+63]
end do
do  TI = 1, 64
   s = s + TS[TI]
end do
```

(b) Цикл, преобразованный для векторизации

Рис.2.15. Распознавание редукций

Такие операции, как *and, or, min, max* являются ассоциативными и их редукция может быть параллелизована при всех обстоятельствах.

Преобразования для доступа к памяти (Memory Access Transformation). Эффективность высокоскоростных приложений зависит как от свойств процессора, так и от свойств памяти. Общеизвестно, что рост быстродействия памяти значительно отстает от роста скорости процессора. В результате оптимизация использования памяти становится очень важной проблемой. Факторы, улучшающие характеристики памяти, включают:

• Повторное использование данных, обозначаемое как Q, являющееся отношением числа использований единиц данных к числу из загрузок;
• Параллелизм. Векторные машины часто делят память на банки, что позволяет загружать векторные регистры параллельно или в конвейерном режиме. Скалярные машины часто поддерживают загрузку и сохранение полей из двух или четырех слов, что сокращает число обращений к памяти.

2.3. Размещение программ и данных в многопроцессорных системах

В многопроцессорных ЭВМ каждый процессор выполняет независимую ветвь параллельной программы. В определенных точках программы ветви обмениваются информацией. Это и есть главный источник увеличения времени выполнения параллельной программы.

При разбиении задачи на ветви для исполнения на многопроцессорной ЭВМ (декомпозиция задачи) соблюдаются определенные принципы [1]:

1. Задача должна разбиваться по возможности на наиболее крупные и редко взаимодействующие блоки. 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, представленные на рис.2.16.

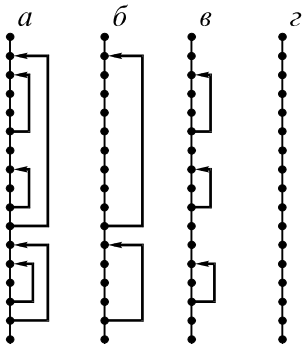

Рис.2.16. Иерархический принцип построения алгоритма: *а* - схема алгоритма; *б* — структурная единица первого уровня иерархии; *в*, *г* — структурные единицы второго уровня иерархии

2. Должно быть обеспечено однородное распределение массивов по ветвям параллельного алгоритма, поскольку при этом уменьшается время, затрачиваемое на взаимодействие ветвей.

Для двумерного массива, например, применимы следующие основные однородные распределения: горизонтальные полосы (ГП), циклические горизонтальные полосы, вертикальные полосы (ВП), скошенные полосы, горизонтальные и вертикальные полосы с дублированием и т. п. Некоторые из этих распределений показаны на рис.2.17.

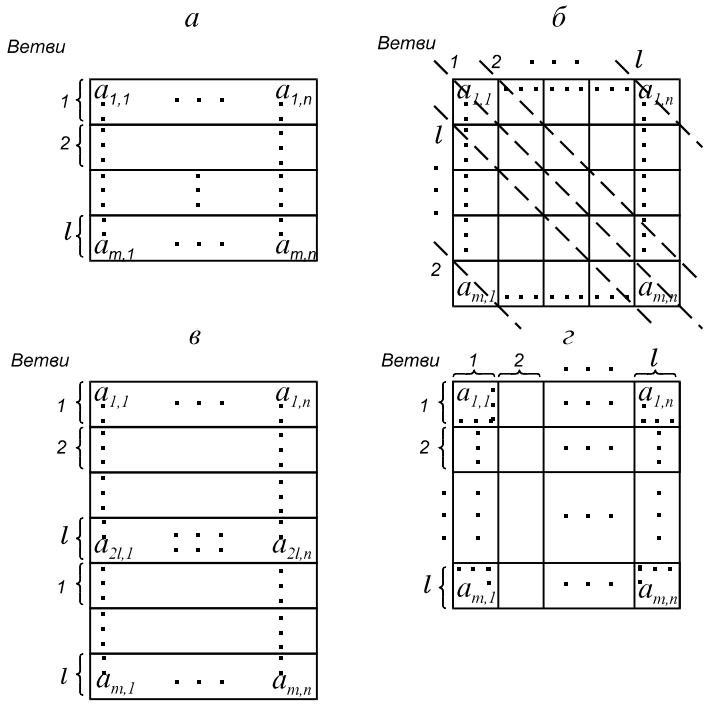

Рис.2.17. Способы распределения двумерного массива: *а* — горизонтальные полосы; *б* — скошенные полосы; *в* — циклические горизонтальные полосы; *г* — вертикальные и горизонтальные полосы с дублированием

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

- трансляционный обмен (“один – всем”);
- трансляционно-циклический обмен (“каждый – всем”);
- коллекционный обмен (“все – одному”);
- парный стационарный обмен (“каждый – соседу”);
- парный нестационарный обмен (“каждый – любому”).

Рассмотрим задачу умножения матриц 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 введение 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.

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.

2.4. Планирование вычислений

Алгоритмы планирования могут быть классифицированы на основе двух критериев [20].

Первый критерий связан с природой *графа управления*: состоит ли он из одного или нескольких базовых блоков*; я*вляется ли граф управления циклическим или ациклическим графом в случае нескольких базовых блоков.

Алгоритмы для одиночных базовых блоков (ББ), называются *локальными*, остальные – *глобальными*. Алгоритмы ациклического планирования связаны с графами управления , которые не содержат циклов, или, что более типично, с циклическими графами, для которых существует самостоятельное управление на каждой обратной дуге графа управления.

Второй критерий связан со способом построения плана. На одном полюсе находятся однопроходные алгоритмы, на другом – сложные многопроходные ветвящиеся алгоритмы. В промежутке находится масса разнообразных алгоритмов.

2.4.1. *Планирование базовых блоков*

Планирование для единичного ББ заключается в построении как можно более короткого плана. Это называется упаковкой локального кода. Поскольку эта проблема имеет переборный характер, внимание было сосредоточено на эвристических алгоритмах планирования. Среди таких алгоритмов наиболее подходящими оказались *списочные расписания*[21], использующие схему с наивысшим приоритетом. Схема оказалась недорогой по

затратам времени, давала близкие к оптимальным результаты и сначала была развита для машинной модели, в которой все операции были однотактными и машина в каждом такте использовала только одно устройство. Далее списочные алгоритмы были распространены на случаи, когда команда использует несколько устройств, и устройства являются многотактными. Списочные алгоритмы получили общее распространение.

Алгоритмы планирования по списку при линейном росте времени планирования при увеличении числа планируемых объектов, 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.

2.4.2. *Метод программной конвейеризации для циклов*

В этом случае в роли ББ выступают итерации цикла. Наиболее простым способом увеличения размера базового блока для глобального планирования является развертка итераций цикла. Недостаток этого подхода – никак не охватывается обратные дуги управляющего графа. B.Rau, и J.Fisher [11] предложили способ решения этой проблемы, который заключается в том, что развертка продолжается до тех пор, пока не встретится повторяющийся образец, который сворачивается в цикл, чье тело составляет повторяющийся план. Это и составляет сущность метода программного конвейера (Software Pipelining) для циклов.

Принципы программной конвейеризации. Программная конвейеризация является одним из наиболее эффективных методов глобального планирования циклов для процессоров. Предположим, что мы имеется VLIW процессор, который позволяет одновременно начать выполнение команд: загрузки из

памяти, плавающей арифметики и записи в память. Команда загрузки (LOAD) выполняется 1 такт, конвейер плавающей арифметики (FMUL) имеет длину 7 тактов и команда записи (ST) выполняется 1 такт. Пустая операция (NOP) также выполняется 1 такт.

Ниже приведена простая схема реализации этого цикла, где каждое командное слово содержит лишь одну основную команду. Команды увеличения переменной цикла и вычисления условия окончания не изображены. Предполагается, что они выполняются параллельно с основными командами. Выполнение планирования базисного блока (тела цикла) не дает никакой возможности параллельного выполнения основных команд.

```
Do 1 i = 1,n              1: load
A( i ) = A ( i )*10.0        fmul
                             nop
                             nop
                             nop
                             nop
                             nop
                             nop
                             st ,  br1
```

Табл.2.2 представляет временную диаграмму возможного выполнения многократно развернутого цикла. По вертикали расположены команды последовательных итераций (копий тела цикла), соответствующих фактору развертки, равному 1,2,...,10, по горизонтальной –фактор развертки.

Таблица 2.2. Развертка итераций цикла

	I –номер итерации, U – фактор развертки (совпадает с I)										
t	1	2	3	4	5	6	7	8	9	10	
0	1: load										
1	fmul	load									
2	nop	F mul	load								
3	nop		fmul	load							

4	nop			fmul	load						
5	nop				fmul	load					
6	nop					fmul	load				
7	nop						fmul	load			
8	st							fmul	load		
9		st							fmul	load	
10			st							fmul	
11				st							
12					st						
13						st					
14							st				
15								st			
16									st		
17										st	br1

Зависимость по данным допускает параллельное выполнение команд разных итераций, которые лежат на одной горизонтали. Эти команды упакованы в одно командное слово. Такой код может быть получен после планирования исходного цикла, развернутого на 10. Время выполнения итерации исходного цикла – 9 тактов, что означает пропускную способность z = 1/9.

В развернутом цикле 10 (исходных) итераций выполняются за 18 тактов (z = 10/18). В полностью развернутом на N цикле будет достигнута идеальная пропускная способность z = N/(N + 8). Отметим, что дальнейшее увеличение фактора развертки лишь незначительно увеличивает пропускную способность (9/17, 10/18, 11/19), но может сильно увеличить размер программы.

После развертки на 10 и планирования в полученном теле цикла образуются два подобных командных слова. Эти слова включают команды итераций (1,8,9) и (2,9,10). Развертки на 11 добавит еще одно подобное командное слово (3,10,11). В общем виде подобные командные слова будут содержать команду ST(*i*) исходной итерации *i* и команды FMUL(*i*+7), LOAD(*i*+8). Эти повторяющиеся шаблоны приводят к следующей идее реализации исходного цикла в виде цикла по 1 из одного командного слова, которая представлена на табл.2.3.

Таблица 2.3. Выделение ядра для реализации программного конвейера

	I – номер итерации, U – фактор раскрутки (совпадает с I)										
t	1	2	3	4	5	6	7	8	9	10	
0	1:loa d										ПРОЛОГ
1	fmul	load									
2	nop	fmul	load								
3	nop		fmul	load							
4	nop			fmul	load						
5	nop				fmul	load					
6	nop					fmul	load				
7	nop						fmul	load			
8	11: st							fmul	load	br11	ЯДРО
n+0		st							fmul		ЭПИЛОГ
n+1			st								
n+2				st							
n+3					st						
n+4						st					
n+5							st				
n+6								st			
n+7									st		

Реализация цикла состоит из трех фрагментов: пролога, ядра (цикла) и эпилога. Первые 8 команд пролога инициируют 8 первых итераций исходного цикла. Ядро состоит из цикла, инициализирующего N-8 оставшихся исходных итераций. При этом каждое командное слово ядра задает параллельное выполнение трех команд из трех разных исходных итераций, включающих также вычисление и окончание двух исходных итераций. Эпилог цикла содержит 8 команд, завершающих последние 8 итераций.

Таким образом, на выполнение N итераций затрачивается 8+(N–8)+8 = N+8 тактов, что означает оптимальную пропускную способность N/(N +8) для исходного цикла, которая казалась достижимой только при полной законченной развертке и

конвейеризации итераций цикла. Преобразование цикла к такому виду и составляет цель программной конвейеризации. Результирующая программа, построенная на этом принципе, обычно весьма компактна и характеризуется оптимальной пропускной способностью.

2.4.3. *Метод планирования трасс для ациклических графов управления*

Базовые блоки невелики по размеру (5 – 20 команд) и даже при оптимальном планировании параллелизм не может быть большим, поэтому следует переходить к обработке с перекрытием последовательных базовых блоков.

Для глобального планирования имеется одно очень важное обстоятельство: необходимо стремиться уменьшать длину кода базового блока тем сильнее, чем выше частота его исполнения. Поэтому эффективное глобальное планирование может в действительности увеличить размер программы за счет удлинения редко выполняемых блоков, чтобы слегка уменьшить длину высокочастотных блоков.

На рис.2.18 точками представлена экспериментальная зависимость ускорения от размера базового блока. Поле полученных в экспериментах результатов ограничено контуром (точками представлены значения для некоторых $ББ_i$). На основе рис.2.18 с учетом вероятностных характеристик контура результатов можно получить следующую качественную зависимость:

$$r = a + b\,w,$$

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

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 вы

полнению развертки. Существуют способы преодоления этого препятствия.

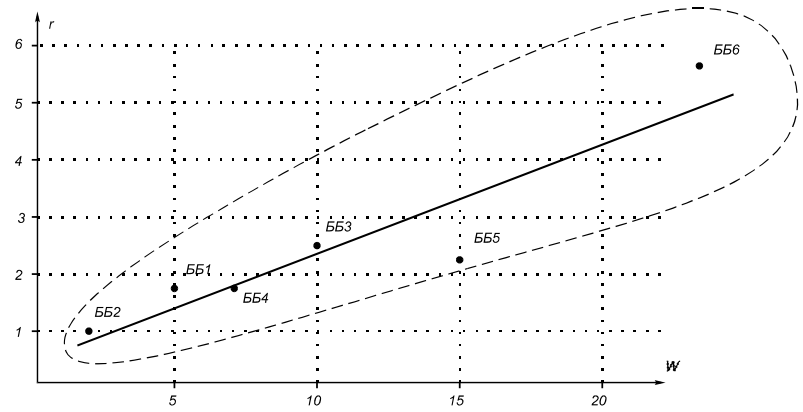

Рис.2.18. Рост ускорения в зависимости от размера базового блока

Достаточно универсальный метод планирования трасс предложил в 80-е годы J.Fisher[22]. Рассмотрим этот метод на примере рис.2.19, на котором представлена блок-схема тела цикла. На схеме в кружках представлены номера вершин, а 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

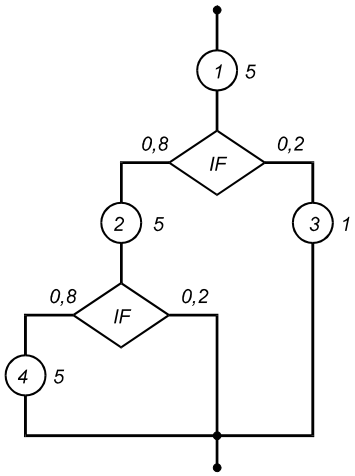

Рис.2.19. Пример выбора трасс

Суммарное время выполнения этих блоков будет в соответствии с вышеприведенными формулами 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. Если переходов из данной трассы в другие трассы 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 *T1/T2* = 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.2.20).

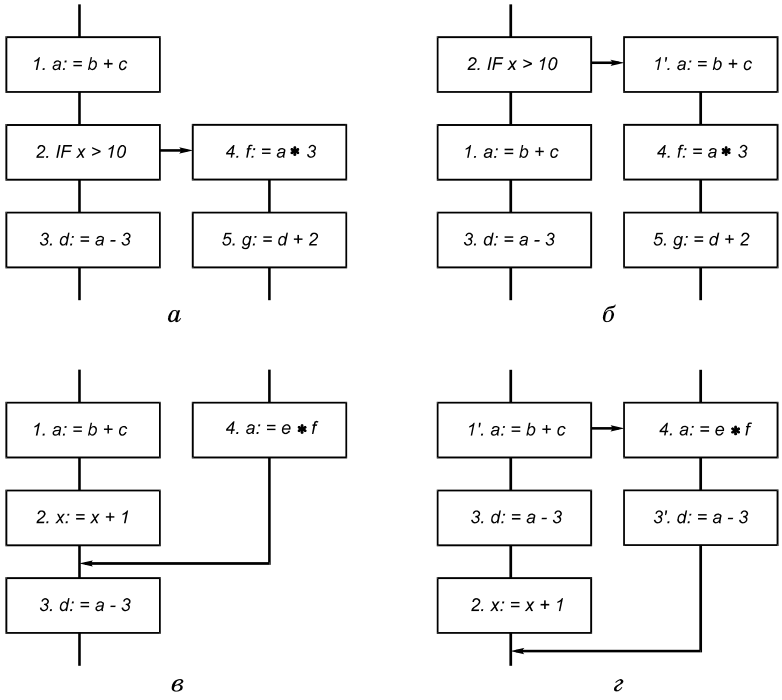

Рис.2.20. Способы введения компенсационных кодов

Пусть текущая трасса (рис.2.20а) состоит из операций 1, 2, 3. Предположим, что операция 1 не является срочной и перемещается поэтому ниже условного перехода 2. Но тогда операция 4 читает неверное значение *a*. Чтобы этого не произошло, компилятор водит компенсирующую операцию 1 (рис.2.20б).

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.2.20в). Предположим, что компилятор перемещает операцию 3 в положение между операциями 1 и 2. Тогда в операции 3 будет использовано неверное значение *a*. Во избежание этого, необходимо ввести компенсирующий код 3 (рис.2.20г).

Порядок планирования трасс для получения конечного результата таков:

1. Выбор очередной трассы и ее планирование.
2. Коррекция межтрассовых связей по результатам упаковки очередной трассы. Компенсационные коды увеличивают размер машинной программы, но не увеличивают числа выполняемых в процессе вычислений операций.
3. Если все трассы исчерпаны или оставшиеся трассы имеют очень низкую вероятность исполнения, то компиляция программы считается законченной, в противном случае осуществляется переход на пункт 1.

Если тело цикла создает слишком короткую трассу, то в качестве трассы может быть выбрано несколько (или несколько десятков) последовательных итераций цикла, что может заметно повысить параллелизм трассы. Например, из цикла

```
i: = 1
loop
if i > n  then exit
body  i = i+1
```

после трехкратной развертки можно получить трассу:

```
i:= 1
if i > n then exit
body
i = i+1
if i > n then exit
body
i = i+1
if i > n then exit
body
```

```
i = i+1
loop
```

В исходной трассе каждая строка содержит только одну операцию. Преобразование трассы в программу, состоящую из VLIW-команд заключается в перемещении операций исходной трассы вверх и вниз так, чтобы в большинстве строк оказалось несколько операций для одновременного исполнения.

Контрольные вопросы к главе 2

1. Опишите основные этапы подготовки задачи для параллельных вычислений.
2. Дайте определение понятию оптимизации программы.
3. Что такое анализ зависимостей?
4. Назовите виды трансформаций общего назначения.
5. Что выполняют трансформации реорганизации циклов?
6. Опишите трансформации, реструктуирующие цикл.
7. Какие методы размещения программ и данных используются в многопроцессорных системах?
8. Что такое планирование вычислений?
9. Опишите метод планирования по списку.
10. Что такое программная конвейеризация?
11. В чем сущность метода планирования трасс?
12. Что такое компенсационные коды?
13. Каков порядок планирования трасс?