Лекция 4. Недостатки ярусно-параллельных форм и зависимости операторов последовательной программы

А, собственно, зачем нам последующие лекции? Мы все уже выяснили и решили! Каноническая параллельная форма алгоритма, реализующего программу, позволяет нам определить, какое максимальное ускорение можно получить для нее на параллельной вычислительной системе, и, стало быть, принять решение: распараллеливать ли старую программу с заранее известным ускорением или нужно выбирать новый алгоритм. Но! Всегда существует «но!». В нашем случае их целых два.

Во-первых, каноническая параллельная форма дает только теоретическую оценку ускорения на машинах в модели PRAM при наличии неограниченного количества ресурсов. Параллельных компьютеров, подчиняющихся модели PRAM, в реальности не существует, тем более, не бывает неограниченных ресурсов. Для приближения модели к действительности необходимо проставить на узлах графа времена выполнения операций (возможно, разные для разных исполнителей) и на дугах — времена передачи данных (если в пределах одного исполнителя, то одни значения, если между разными исполнителями — то другие), учесть ограниченное количество исполнителей. И вот уже на таком параметризованном графе (а, точнее, на семействе графов, решая проблему распределения операций по исполнителям) оценивать ускорение, отыскивая наилучший вариант. Это задача, к сожалению, NP-полная.

Концепция распараллеливания в рамках PRAM машин без оглядки на ресурсы получила название концепции *неограниченного параллелизма*. В свое время появление теории ярусно-параллельных форм вызвало в мире программирования эйфорию, дескать, все понятно, распараллелим, если возможно, посчитаем и закидаем всех шапками! Не тут-то было! Тем не менее, концепцию неограниченного параллелизма нельзя недооценивать. Она позволяет понять, чего вообще можно ожидать от алгоритма при его реализации на параллельной архитектуре.

Во-вторых, построение графа программы, реализующей алгоритм, всегда имеет большую трудоемкость. Давайте возьмем простой программный фрагмент (рис 4.1) и построим для него граф алгоритма, рассматривая цикл просто как форму сокращения записи программы. Заметим, что если значение i далее в программе не используется, то нижнюю линейку в графе можно опустить.

```
b(1) = a(1)*2
do i=2,5
  b(i) = b(i-1)+a(i)*2
enddo
```

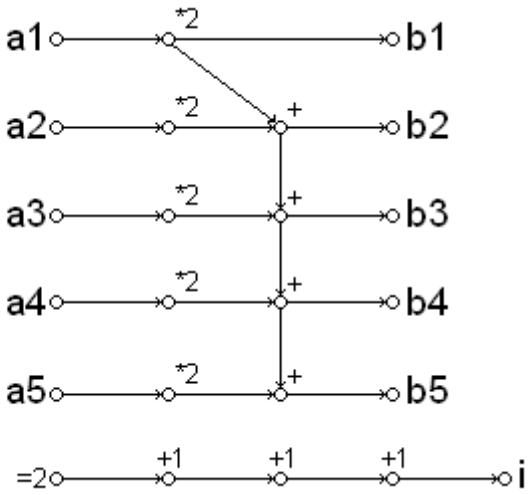

Рис. 4.1 Граф алгоритма, соответствующий программному фрагменту.

Вы видите, что для простейшего цикла с количеством итераций 4, граф получается довольно громоздким. А если количество итераций 2000000? А количество циклов порядка сотни (не так уж много для реальной задачи)? Приплыли… Соответствующий граф займет площадь небольшого концертного зала, ну а далее можно в этом зале строить каноническую параллельную форму и определять максимально возможное ускорение. Проделав эту процедуру человек «получает нечто формально правильное, а по сути — издевательство» .

Понятно, что анализ реальной программы нельзя разумно свести к построению канонической параллельной формы алгоритма, реализуемого этой программой. Граф алгоритма использует понятия слишком низкого уровня — отдельных операций, выполняемых над данными, — или, как принято говорить, декомпозиция алгоритма с помощью графа операций обладает слишком большой *гранулярностью*.

Для того, чтобы можно было проанализировать последовательную программу на параллельность, затратив приемлемое количество усилий, нам следует перенести анализ с уровня операций на более высокий уровень — уровень групп операций (например, отдельных операторов или их блоков), тем самым понизив степень гранулярности проводимой декомпозиции. Нечто похожее мы уже проделывали на предыдущей лекции, когда ликвидировали недетерминированность графа, связанную с небольшими условными ветвлениями в программе, с помощью укрупнения операций (рис. 3.3а и 3.3б).

Мы будем считать, что наша программа состоит из набора операторов S_1, S_2, …, S_k, расположенных в тексте программы в определенном порядке. Например,

S_1: a = 2*b + 15;

S_2: b = a + 10*x;

S_3: d = b – c;

S_4: d = a/c;

Расположение операторов в тексте программы задает их *статический* порядок. Мы всегда, глядя на исходный текст программы, можем сказать, какой из двух операторов в тексте расположен выше, а какой ниже. Но статический порядок операторов не всегда совпадает с порядком их исполнения. Например, для следующего программного фрагмента

S_1: goto S_4;

S_2: c = a + 10*b;

S_3: goto S_6;

S_4: d = b – c;

S_5: goto S_2;

S_6: b = a/d;

статический порядок операторов есть «S_1, S_2, S_3, S_4, S_5, S_6», а порядок их последовательного выполнения в программе — «S_1, S_4, S_5, S_2, S_3, S_6». Порядок выполнения операторов в программе (при заданных начальных данных) принято называть *динамическим* порядком операторов. Наша основная задача заключается в том, чтобы определить, какие операторы в последовательной программе могут быть выполнены различными исполнителями на параллельной вычислительной системе, и какое ускорение при этом можно получить.

Прежде чем приступать к решению этой задачи, давайте вспомним некоторые сведения, обычно излагаемые в курсах операционных систем [7].

Мы временно отвлечемся от программ и компьютеров и поговорим о некоторых *активностях* вообще. При этом под активностью будем понимать последовательное выполнение ряда действий, направленных на достижение определенной цели. Типичным примером активности является процесс приготовления бутерброда. Последовательность действий (не обязательно единственно правильную), необходимую для создания бутерброда, можно описать следующим образом:

1. Отрезать ломтик хлеба.
2. Отрезать ломтик колбасы.
3. Намазать хлеб маслом (если на масло хватает средств).
4. Положить колбасу на намазанный ломтик хлеба (или, может быть, хлеб на колбасу — кот Матроскин считает, что второй вариант предпочтительнее).

Все действия, входящие в состав активности, будем считать атомарными или неделимыми, т.е. исполнитель, приступивший к выполнению действия, не может отвлечься и заняться чем-либо другим, пока действие не завершено. В то же время между

действиями исполнителю разрешено подрабатывать на стороне. Предположим, что у нас есть две активности P и Q, состоящие из атомарных действий abc и efg соответственно. Обе активности поручено осуществить одному и тому же исполнителю. Что произойдет в результате?

Если активности P и Q выполняются строго в последовательном порядке, то мы получаем следующую совокупность атомарных операций: abcdef. Но в наших допущениях исполнитель, выполнив атомарную операцию активности P, может далее переключиться на выполнение атомарной операции активности Q и т.д. Активности расслаиваются на неделимые действия с различным их чередованием, при этом порядок атомарных операций внутри одной активности сохраняется: aebcfg, aebfcg, …, efgabc. Полученное явление на английском языке получило название *interleaving*. Так как псевдопараллельное выполнение двух активностей приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения.

Рассмотрим пример. Пусть у нас имеется две активности P и Q, состоящие из двух атомарных операций каждая

P:	Q:
x = 2	x = 3
y=x-1	y = x+1

При их последовательном выполнении (PQ) мы получим результат x = 3, y = 4. Но при различном чередовании атомарных действий активностей мы можем также получить результаты (x = 2, y = 1), (x = 2, y = 3) и (x = 3, y = 2).

Мы будем говорить, что набор активностей (например, программ) *детерминирован*, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные. В противном случае он *недетерминирован*. Выше приведен пример недетерминированного набора программ. Понятно, что детерминированный набор активностей можно безбоязненно выполнять в режиме разделения времени. Для недетерминированного набора такое исполнение нежелательно. Можно ли до получения результатов определить, является ли набор активностей детерминированным или нет? Для этого существуют достаточные условия Бернстайна. Изложим их применительно к программам с разделяемыми переменными.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы входных и выходных переменных – это наборы переменных, которые атомарная операция считывает и записывает. Набор входных переменных программы R(P) (R от слова read) суть объединение наборов входных

переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова write) суть объединение наборов выходных переменных для всех ее неделимых действий. Например, для программы

P: x=u+v

y=x*w

получаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что переменная x присутствует как в R(P), так и в W(P).

Теперь сформулируем условия Бернстайна.

Если для двух данных активностей P и Q:

• пересечение W(P) и W(Q) пусто,

• пересечение W(P) с R(Q) пусто,

• пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, псевдопараллельное выполнение P и Q детерминировано, а, может быть, и нет. Случай двух активностей естественным образом обобщается на их большее количество.

Вот теперь мы можем вернуться к анализу на параллельность последовательных программ, состоящих из набора операторов (или их блоков).

Первое, что мы должны выяснить, справедливы ли условия Бернстайна для операторов одной последовательной программы на параллельной вычислительной системе. Пусть в качестве активностей P и Q у нас выступают два оператора последовательной программы S_1 и S_2. Если для них выполнены условия Бернстайна, то псевдопараллельно (при наличии одного исполнителя) — в режиме разделения времени — они дают детерминированный набор активностей. Кажется, что и при наличии нескольких исполнителей детерминированность должна сохраниться. Рассмотрим следующий пример. Пусть операторы S_1 и S_2 динамически предшествуют друг другу и имеют вид:

S_1: x = 2*y+z;

S_2: d = a-b;

Входные и выходные данные операторов S1 и S2 вообще не пересекаются. Их можно исполнять псевдопараллельно и они образуют детерминированный набор активностей. Но можно ли выполнять их параллельно в рамках работы одной программы? С первого взгляда — ничего не препятствует! Но предположим, что строгий динамический порядок выполнения операторов выглядит так:

S_1: x = 2*y+z;

$S_{3/2}$: a = x;

S_2: d = a-b;

И все! Если для этого фрагмента построить укрупненный граф алгоритма, то выяснится, что S_2 нельзя выполнить до вычисления $S_{3/2}$, а тот, в свою очередь до исполнения S_1.Параллельности нет…

В формулировку условий Бернстайна для анализа возможности одновременного выполнения операторов S_1 и S_2 последовательной программы на параллельном компьютерном комплексе необходимо внести уточнение:

Если для двух операторов (или блоков операторов) S_1 и S_2 последовательной программы, **непосредственно** динамически следующих друг за другом, выполнено:

• пересечение $W(S_1)$ и $W(S_2)$ пусто,

• пересечение $W(S_1)$ с $R(S_2)$ пусто,

• пересечение $R(S_1)$ и $W(S_2)$ пусто,

то операторы S_1 и S_2 могут быть выполнены одновременно разными исполнителями на параллельной вычислительной системе.

К сожалению, условия Бернстайна являются достаточными, но не необходимыми, и предполагают, что у операторов, непосредственно следующих друг за другом, могут пересекаться только наборы входных данных. Так в реальных последовательных программах почти не бывает.

Давайте разберемся, что будет происходить в том случае, когда для двух операторов S_1 и S_2, динамически следующих друг за другом, условия Бернстайна нарушаются (непосредственное следование здесь не играет роли). Здесь мы будем следовать изложению материала в [16].

Начнем с нарушения первого условия, Итак, пусть для двух операторов (или блоков операторов) S_1 и S_2 последовательной программы, динамически следующих друг за другом, выполнено:

• пересечение $W(S_1)$ и $W(S_2)$ не пусто,

• пересечение $W(S_1)$ с $R(S_2)$ пусто,

• пересечение $R(S_1)$ и $W(S_2)$ пусто,

Рассмотрим пример:

S_1: x = 2*y+z;

S_2: x = a-b;

В большинстве случаев, в последовательной программе (а я напомню, что мы работаем в понятиях выполняемой программы, а не исходных текстов) такой случай связан либо с экономией памяти, когда под разные данные отводится одна и та же

область памяти, либо с небрежностью (не сказать плохого слова) программистов. Простое переименование переменной «x» в операторе S_2 снимает возникающие вопросы. Тем не менее, в первоначальном виде операторы зависят друг от друга, и такой вид зависимости мы будем называть *зависимостью по выходным данным* (output dependence). Обычно такая зависимость записывается как $S_1 \delta^o S_2$ и графически изображается следующим образом (рис 4.2):

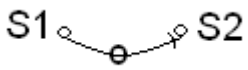

Рис 4.2 Зависимость по выходным данным.

Как правило, зависимость по выходным данным не мешает распараллеливанию. Переименуйте переменные и, если, оба оператора исполняются непосредственно один за другим, — распараллеливайте на здоровье!

Пусть нарушено третье условие Бернстайна. Для двух операторов (или блоков операторов) S_1 и S_2 последовательной программы, динамически следующих друг за другом, выполнено следующее:

- пересечение $W(S_1)$ и $W(S_2)$ пусто,
- пересечение $W(S_1)$ с $R(S_2)$ пусто,
- пересечение $R(S_1)$ и $W(S_2)$ не пусто ,

Рассмотрим пример:

S_1: x = 2*y+z;

S_2: y = a-b;

Переменная «y» в последовательной программе сначала используется для вычислений в операторе S_1 ,а затем переопределяется в операторе S_2. Если исполнитель, взявший на себя выполнение оператора S_1, использует значение «y» до того, как это значение переопределит исполнитель, выполняющий S_2, то распараллеливание безопасно. Давайте поступим следующим образом. Перед выполнением соответствующих операторов скопируем значение данной переменой в локальную память исполнителей и лишь потом разрешим им выполнить операции. Тогда расчеты окажутся верными. Эту форму зависимости операторов называют антизависимостью (antidependence). Обычная форма ее записи выглядит как $S_1 \delta^{-1} S_2$, а графическое представление имеет вид (рис 4.3):

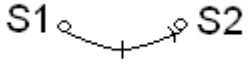

Рис 4.3 Антизависимость.

Наконец, осталось рассмотреть нарушение второго условия Бернстайна, т.е. ситуацию, когда для двух операторов (или блоков операторов) S_1 и S_2 последовательной программы, динамически следующих друг за другом, выполнено следующее:

• пересечение $W(S_1)$ и $W(S_2)$ пусто,

• пересечение $W(S_1)$ с $R(S_2)$ не пусто,

• пересечение $R(S_1)$ и $W(S_2)$ пусто ,

Рассмотрим пример:

S_1: x = 2*y+z;

S_2: y = a-x;

Это самый тяжелый случай с точки зрения распараллеливания. Результат выполнения оператора S_1 используется для выполнения оператора S_2. Если операторы непосредственно динамически следуют друг за другом, то их распараллеливание невозможно. Такой тип зависимости называют *потоковой зависимостью* или *истинной зависимостью*. Ее принято записывать $S_1 \delta S_2$ и изображать так (рис 4.4):

Рис. 4.4.Потоковая или истинная зависимость

Используя введенные обозначения, зависимости в программном фрагменте из четырех операторов, приведенном выше, можно графически изобразить следующим образом (рис. 4.5):

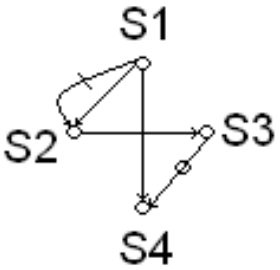

Рис. 4.5 Графическое представление зависимостей в программном фрагменте

Конечно, зависимости, влияющие на возможность распараллеливания последовательных программ, не исчерпываются тремя рассмотренными видами зависимостей. В программе могут встречаться условные операторы, изменяющие динамический порядок других операторов при различных наборах входных данных. В примере

S_1: x = cos(z);

```
        if (x > 0) {
S2:             a = b + c;
        } else {
S3:             a = b - c;
        }
```

выполнение операторов S_2 и S_3 зависит от результата выполнения оператора S_1, но не по данным, а *по управлению*. Этот вид зависимости записывают как $S_1 \delta^c S_2$ и $S_1 \delta^c S_3$.

Зависимость по управлению можно свести к зависимости по данным, если ввести новые специфические операторы, изменив вид программы:

S_1: x = cos(z);

S_2: where (x > 0) a = b + c;

S_3: where (x ≤ 0) a = b - c;

Здесь переменная «x» становится входной переменной для операторов S_2 и S_3, и зависимости по управлению заменяются зависимостями по данным.

Еще один вид зависимости в последовательном фрагменте программного кода может возникнуть при его распараллеливании на вычислительной системе с ограниченными ресурсами. Пусть у нас есть 2 оператора, динамически непосредственно следующих друг за другом и не имеющих зависимостей по данным.

S_1: x = 2*y/z;

S_2: c = a/b;

Условия Бернстайна выполнены, но если у вас на параллельной вычислительной системе существует единственный исполнитель, умеющий делить одно данное на другое, то параллельное исполнение операторов оказывается невозможным. Это — *зависимость по ресурсам*, $S_1 \delta^R S_2$. Мы далее будем работать в модели неограниченного параллелизма, где зависимость по ресурсам не существует.

Итак, построив граф зависимостей по данным для операторов программы, можно выяснить, какие операторы могут быть выполнены одновременно на различных исполнителях, какие являются потенциально распараллеливаемыми, и для каких параллельное выполнение невозможно или требует дополнительного анализа.