Лекция 7. Эквивалентные преобразования и «устранение» истинных зависимостей в циклах

В предыдущей лекции прозвучали два замечания, не получившие пока своего дальнейшего развития. Во-первых, исследовался вопрос о возможности перестановки в многомерном цикле циклов, входящих в его состав. Во-вторых, говорилось, что распараллеливание для циклов с истинной зависимостью может быть проблематично. Сейчас мы подробнее остановимся на этих двух темах.

Тема первая. Почему я неоднократно подчеркивал важность возможности перестановки порядка циклов по различным индексам внутри многомерного цикла? Рассмотрим следующий **пример 7.1**:

```
do i = 1, u1
    do j = 1,u2
        do k = 1,u3
        S:    a[i, j, k] = a[i, j-1, k+1]*2
        enddo
    enddo
enddo
```

Вектор расстояний суть **D** = (0, 1, –1), а вектор направлений **d** = (“=”, “<”, “>”). У нас истинная зависимость. Распараллеливание возможно либо по индексу *i*, либо по индексу *k*, (либо по обоим) без ограничений. Допустим, что по соображениям математической модели эффективно распараллеливание именно по индексу *k* (например, для правильной балансировки вычислений). Тогда по окончании цикла по индексу *k* необходима барьерная синхронизация исполнителей. Эта операция требует определенных затрат времени, приводя к увеличению накладных расходов на распараллеливание и уменьшая полученное ускорение. И таких операций потребуется $u_1 \times u_2$.

В то же время, в приведенном выше трехмерном цикле можно изменить порядок составляющих циклов (без изменения графа алгоритма!):

```
do j = 1, u2
    do k = 1,u3
        do i = 1,u1
        S:    a[i, j, k] = a[i, j-1, k+1]*2
        enddo
    enddo
enddo
```

Вектор расстояний суть **D** = (1, –1, 0), а вектор направлений **d** = (“<”, “>”, “=”). Анализ распараллеливания не меняется. По-прежнему, при распараллеливании по индексу *k* необходима барьерная синхронизация по окончании этого цикла. Но теперь требуется только u_2 таких операций. Налицо сокращение накладных расходов и, как следствие, — увеличение эффективности работы параллельной программы.

Рассмотренное преобразование цикла приводит к изменению динамического порядка выполнения операторов развернутой конструкции без изменения графа алгоритма и является одним из примеров эквивалентного преобразования программ.

Определение. Эквивалентным преобразованием последовательной программы будем называть изменение динамического порядка ее операторов, сохраняющее граф алгоритма.

Теперь мы созрели для формулировки **фундаментальной теоремы о зависимости по данным**:

Любое изменение динамического порядка операторов последовательной программы, сохраняющее все зависимости по данным, не изменяет граф алгоритма программы.

Такое преобразование сохраняет в программе порядок всех операций обращения к памяти и записи в память за исключением, быть может, операций ввода-вывода.

Тема вторая. Почему при наличии истинных зависимостей в циклах мы говорим не о невозможности распараллеливания программы, а лишь о проблематичности такого распараллеливания? Ну, частично, ответ был дан выше. В рассмотренном трехмерном цикле присутствовала истинная зависимость, а, тем не менее, согласно теории, распараллеливание было возможно.

Однако, существуют ситуации, в которых по теории распараллеливание в циклах недопустимо, но с помощью эквивалентных преобразований мы можем это распараллеливание осуществить.

Пример 7.2. Рассмотрим цикл из примера 6.5

```
do i = 1, u1
        do j = 1,u2
                S:      a[i, j] = a[i-1, j-1]*2
        enddo
enddo
```

Анализируя его, мы пришли к выводу, что любое распараллеливание цикла невозможно. Вспомним все же, что операторы цикла в программах есть просто

сокращение формы записи. Развернем циклы, и изобразим возникающие зависимости между элементами массива a на плоскости (i, j) (рис. 7.1).

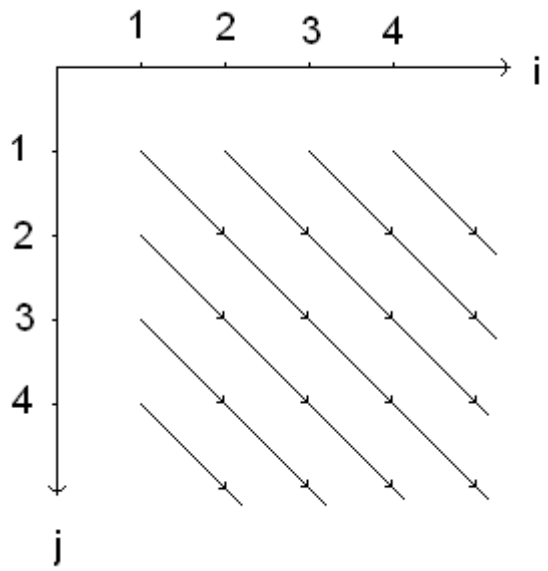

Рис. 7.1. Зависимости между элементами массива для примера 7.2

Как видим, зависимости по данным между элементами массива расположены по диагоналям данного графического представления. При этом каждая диагональ может быть выполнена на своем исполнителе независимо от другой диагонали. Таким образом, сделав соответствующую замену индексных переменных циклов (сохраняющую граф алгоритма), мы можем распараллелить фрагмент последовательного кода и даже, при хороших мозгах, обеспечить правильный баланс загрузки исполнителей.

Этот прием распараллеливания при наличии истинной зависимости в цикле, конечно, является нестандартным. Но его суть заключается в том, чтобы свести loop carried dependence к зависимости, независящей от цикла.

Существует набор аналогичных, но уже стандартных приемов. Для простоты мы будем рассматривать их на одномерных циклах.

Прием 1. Разделение цикла (loop distribution)

Возьмем следующий цикл

```
do i = 1, u
      S1:    a[i] = d[i]+5*i
      S2:    c[i] = a[i-1]*2
enddo
```

Легко видеть, что для этого цикла расстояние зависимости равно 1, стало быть, мы имеем дело с истинной зависимостью в самом плохом варианте. По теории такой цикл не распараллеливается. Прибегнем к эквивалентным преобразованиям программы и разобъем наш цикл на два цикла:

```
do i = 1, u
        S1:     a[i] = d[i]+5*i
enddo
do i = 1, u
        S2:     c[i] = a[i-1]*2
enddo
```

Такое преобразование не изменяет зависимости между операторами программы в целом, значит оно допустимо. При этом каждый из полученных циклов вообще не имеет зависимостей и может быть распараллелен без ограничений. Необходимо, конечно, не забыть о барьерной синхронизации между циклами.

Два набора операторов в цикле могут быть распределены по двум циклам, если в исходном цикле не существует циклов истинной зависимости между операторами (нет рекурсивных зависимостей). Для случая двух операторов это означает, что не должно быть таких $\kappa_1, \kappa_2, \lambda_1, \lambda_2$, что $S_1^{\kappa_1} \delta S_2^{\lambda_1}$ и $S_2^{\kappa_2} \delta S_1^{\lambda_2}$. Например, для цикла

```
do i = 1, u
        S1:     a[i] = c[i-1]+5*i
        S2:     c[i] = a[i-1]*2
enddo
```

нельзя использовать этот прием.

Разделение цикла увеличивает гранулярность декомпозиции и требует введения барьерной синхронизации — это неизбежные накладные расходы, оплачивающие возможность распараллеливания.

Прием 2. Выравнивание цикла (loop alignment)

Возьмем тот же самый цикл, что и в предыдущем примере

```
do i = 1, u
        S1:     a[i] = d[i]+5*i
        S2:     c[i] = a[i-1]*2
enddo
```

Заметим, что если изобразить реинкарнации операторов на разных итерациях по горизонтали, то зависимость по данным будет наклонно переходить из одной итерации в другую (рис. 7.2).

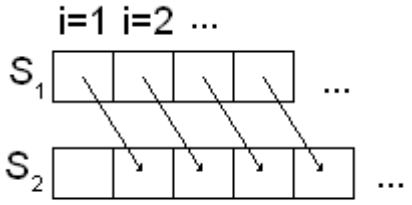

Рис 7.2. Зависимости между реинкарнациями операторов

Идея выравнивания заключается в сдвиге реинкарнаций операторов, так чтобы зависимости стали вертикальными и лежали внутри одной итерации (рис. 7.3).

Для этого можно сделать следующие эквивалентные преобразования

do i = 0, u

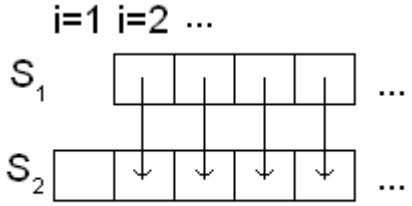

Рис 7.3. Идея модификации цикла

S_1: if (i > 0) a[i] = d[i]+5*i

S_2: if (i < n) c[i+1] = a[i]*2

enddo

Или, выкидывая условия для уменьшения накладных расходов,

c[1] = a[0]*2

do i = 1, u-1

S_1: a[i] = d[i]+5*i

S_2: c[i+1] = a[i]*2

enddo

a[u] = d[u]+5

В полученном цикле у нас зависимость локализована в теле цикла (расстояние зависимости равно 0), и цикл может быть распараллелен по итерациям.

Наличие 2-х и более зависимостей между одними и теми же операторами цикла делает невозможным его выравнивание. Например, для цикла

do i = 1, u

S_1: a[i+1] = d[i]+5*i

S_2: c[i] = a[i+1]*2+a[i]

enddo

выравнивание применить нельзя.

Прием 3. Репликация кода (code replication)

Для того, чтобы распараллелить последний пример, прибегнем к новому приему. Введем временную переменную и несколько ухудшим исходный код, заставляя программу определенные операции выполнять дважды

```
do i = 1, u
    S1:   a[i+1] = d[i]+5*i
          if(i == 1) t = a[1]
          else t = d[i-1]+5*(i-1)
    S2:   c[i] = a[i+1]*2+t
enddo
```

Заметим, что такое преобразование не изменяет графа алгоритма, но при этом мы получаем зависимость, локализованную внутри одной итерации. Следовательно, преобразованный цикл можно распараллелить.

Сформулируем **основную теорему** этой лекции для невложенных циклов.

Приемы разделения цикла, выравнивания цикла и допустимая перестановка операторов в теле цикла достаточны для устранения зависимостей, связанных с циклом, если:

- **В цикле нет рекурсивных зависимостей.**
- **Расстояния для каждой зависимости есть константа, не зависящая от индекса цикла.**

Доказательство теоремы оставляем для самостоятельной работы. Теорема, естественно, может быть обобщена для случая вложенных циклов.

Помимо зависимостей по элементам массивов в циклах могут встречаться и зависимости по скалярным переменным, препятствующие распараллеливанию циклов. В некоторых случаях с ними удается бороться не всегда правда, сохраняя граф алгоритма. Приемы борьбы со скалярными зависимостями могут оказаться опасными (например, последовательная программа будет выдавать результаты отличные от результатов работы параллельной программы). Будьте осторожны!

Прием 4. Приватизация скалярных переменных

Рассмотрим цикл

```
do i = 1, u
    t = a[i]
    a[i] = b[i]
    b[i] = t
```

```
enddo
```

По векторным переменным у нас сплошные антизависимости, локализованные в пределах одной итерации. Но скалярная переменная t препятствует распараллеливанию (наиболее очевидно это на машинах с общей памятью). Справиться с этой задачей можно, если на каждой итерации или, что более практично, завести свою собственную временную переменную.

```
do i = 1, u
        private t
        t = a[i]
        a[i] = b[i]
        b[i] = t
enddo
```

Это безопасный прием.

Прием 5. Индукционные переменные

Пусть дан следующий фрагмент кода

```
j = 0
do i = 1, u
        j = j+k
        a[i] = j
enddo
```

Здесь между всеми итерациями существует истинная зависимость по j. Присмотревшись, можно заметить, что с точки зрения чистой математики на n-ой итерации значение j = k*n. Так что можно (с этой точки зрения) написать

```
do i = 1, u
        j = k*i
        a[i] = j
enddo
```

Получаем параллельный цикл. Но — это рискованная операция. Мы изменяем граф алгоритма. Пользоваться таким приемом нужно с большой осторожностью. Переменная j в этом пример — это *индукционная* переменная. Вообще, индукционными называют переменные, значения которых на n-й итерации представляют собой функцию только от номера итерации и, возможно, некоторого начального значения, присвоенного вне цикла.

Прием 6. Редукционные операции

Возьмем пример

```
j = 0
do i = 1, u
        j = j+a[i]
enddo
```

Между всеми реинкарнациями оператора существует истинная зависимость по скалярной переменной. По теории распараллеливание невозможно. Но операция сложения (опять-таки, теоретически) является операцией редукционной. Если нам требуется посчитать сумму 8-ми элементов массива, то можно сделать это так. Посчитать парные суммы a_{12}=a[1]+a[2], …, a_{78}=a[7]+a[8] на четырех процессорах, затем суммы $a_{1234}=a_{12}+a_{34}$, $a_{5678}=a_{56}+a_{78}$ на двух процессорах и после этого — окончательную сумму. Этот процесс дает выигрыш по сравнению с последовательным расчетом, но изменяет граф алгоритма. Пользоваться им необходимо очень осторожно. Подобный прием можно применять ко всем ассоциативным операциям.

Вот вкратце все основные приемы, позволяющие бороться с зависимостями внутри циклов, препятствующими распараллеливанию программ.