Параллельное программирование

Виды параллелизма, эквивалентные преобразования и зависимости

20.04.2016 (продолжение лекции)

Виды параллелизма в алгоритмах и программах

В настоящее время принято подразделять параллелизм на два вида: *конечный* и *массовый*.

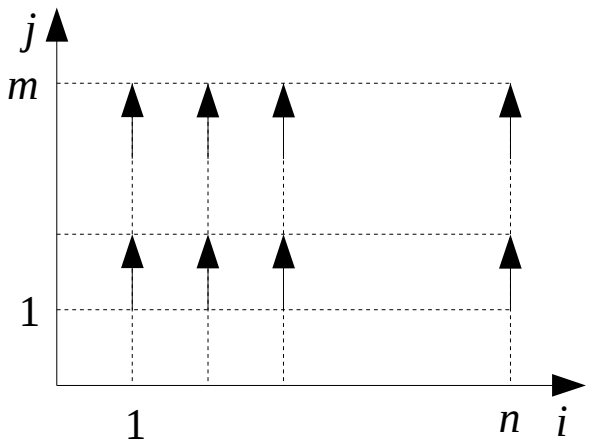
Конечный параллелизм определяется информационной независимостью некоторых фрагментов в тексте программы.

Массовый параллелизм определяется информационной независимостью операций циклов программ.

Массовый параллелизм подразделяется на *координатный* и *скошенный*.

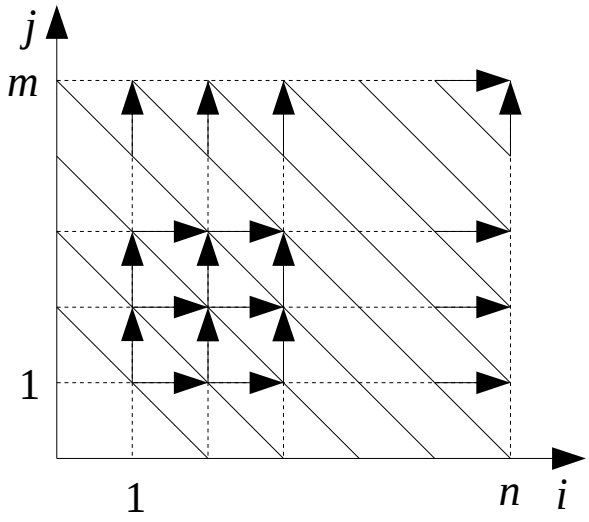
Координатный параллелизм:

```
for (i = 0; i < n; ++i)
for (j = 0; j < m; ++j)
     A[i][j] = A[i][j – 1] + c[i][j] * x;
```



Скошенный параллелизм:

```
for (i = 0; i < n; ++i)
for (j = 0; j < m; ++j)
     A[i][j] = A[i][j – 1] + A[i – 1][j] * x;
```



Эквивалентные преобразования программ

Определение линейной программы

Будем считать, что алгоритм записан с помощью следующих средств языка:

1. В программе может использоваться любое число скалярных переменных и переменных с индексами.
2. Единственным видом исполняемого оператора может быть оператор *присвоить*, правая часть которого представляет собой некоторое арифметическое выражение. Допускается любое число таких операторов.
3. Все повторяющиеся операторы записываются только с помощью цикла do, структура вложенности которого может быть произвольной. Параметр цикла меняется всегда только с шагом +1. Если нижняя граница больше верхней, цикл не выполняется.
4. Допускается использование любого числа условных и безусловных операторов перехода, передающих управление вниз по тексту, и не допускается использование побочных выходов из циклов.
5. Все индексные выражения переменных, границы изменения параметров циклов и условия передачи управления задаются в общем случае неоднородными формами, линейными как по параметрам цикла, так и по внешним переменным программы. Все коэффициенты линейных форм являются целыми числами.
6. Внешние переменные программы всегда являются целочисленными, и вектора их значений принадлежат некоторым целочисленным многогранникам. Конкретные значения внешних переменных известны только перед началом работы программы и неизвестны в момент её исследования.

Программы, удовлетворяющие описанным выше условиям, называют линейными.

Будем называть два арифметических выражения эквивалентными, если их записи совпадают с точностью до обозначения переменных.

Перенумерует одинаковым образом входные переменные эквивалентных выражений и рассмотрим две линейные программы. Предположим, что между их пространствами итераций установлено такое взаимно однозначное соответствие, при котором соответствующие точки выдают срабатывания оператора с эквивалентными выражениями в правых частях. Подобные пространства итераций будем называть эквивалентными, а указанное соответствие – соответствием эквивалентности.

Пусть теперь фиксировано правило, по которому для линейной программы однозначно строится какой-либо граф зависимостей. Возьмём две программы с эквивалентными пространствами итераций и предположим, что отношение эквивалентности является изоморфным для графов, и что сопоставляемые при изоморфизме дуги графов зависимостей относятся к входным переменным с одними и теми же номерами. Такие программы назовём эквивалентными по графам зависимостей, а сами графы – графами эквивалентности. Любое преобразование программы в эквивалентную ей будем называть эквивалентным.

Такое понятие эквивалентности зависит от того, какие графы принимаются во внимание при преобразовании программ. Все эквивалентные программы выполняют одно и то же множество операций, описанных арифметическими выражениями, однако при последовательной реализации программ порядки выполнения операций могут различаться. В общем случае разные порядки могут приводить к разным результатам.

При специальном выборе графов эквивалентности эквивалентные программы дают в одинаковых условиях реализации одни и те же результаты, включая всю совокупность ошибок округления.

Множества используемых переменных в эквивалентных программах могут быть по-разному организованы как для переменных, задающих входные данные, так и для тех, которые формируют результаты.

В реальных условиях при преобразовании программ приходится выполнять и неэквивалентные преобразования, например заменять один оператор последовательностью операторов, реализующих ту же самую операцию. С формальной точки зрения такое преобразование не является эквивалентным, так как меняется пространство операций. В практическом отношении такая замена может привести к изменению результата вычислений из-за влияния ошибок округления, что также не позволяет считать подобное преобразование эквивалентным.

Программы могут быть неэквивалентными и в случае эквивалентности их пространства итераций. Например, рассмотрим суммирование элементов массива. Пусть в одной программе выполняется последовательное суммирование, а в другой применяется принцип сдваивания. И пусть графами эквивалентности являются графы алгоритма. Пространства итераций обеих программ совпадают как по числу точек, так и по содержанию

соответствующих им операций. Тем не менее, программы не эквивалентны, так как их графы алгоритмов не изоморфны. При этом, при точных вычислениях программы будут выполнять одинаковые результаты, а в условиях влияния ошибок округления результаты будут различными. Отметим, что сохранение результатов вычисления имеет безусловный приоритет при преобразованиях программ.

По-существу, программы, эквивалентные по графам алгоритма, представляют собой различные записи одного и того же алгоритма. Единственное, чем они могут принципиально отличаться друг от друга – это объёмом памяти, необходимой для хранения результатов промежуточных вычислений, а также формой организации переменных в массивы.

Эквивалентные программы обладают одним и тем же параллелизмом, но он может быть по-разному записан.

27.04.2016

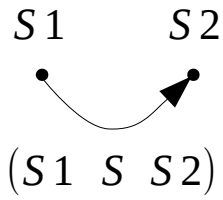
Виды зависимостей по данным

А. `S1: x := 2 y + z`

`S2: y := a – x`

Видно, что результат выполнения оператора S1 используется при выполнении оператора S2. И, если такие операторы следуют непосредственно друг за другом, то их распараллеливание невозможно. Такой тип зависимости называется потоковой зависимостью или истинной зависимостью.

На диаграмме изображается так:

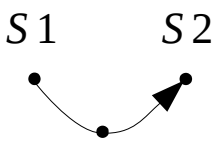


Б. `S1: x := 2 y + z`

`S2: x := a – b`

Зависимость по выходным данным. Она не мешает распараллеливанию, так как в одной из команд (например, S2) можно идентификатор x заменить другим идентификатором.

На диаграмме изображается так:

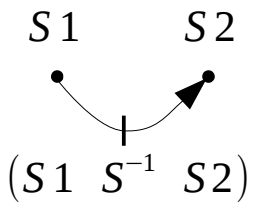


В. `S1: x := 2 y + z`

```
S2: y := a - b
```

Антизависимость. Если S1 выполняется раньше S2, то распараллеливание безопасно. В других случаях значение y может быть сохранено или скопировано в локальную память исполнителей, и распараллеливание здесь возможно.

На диаграмме изображается:



Выделяют и другие виды зависимостей:

1. Зависимость по управлению

```
S1: x := cos(z)
     if (x > 0) then
          S2: a := b + c
     else
          S3: a := b - c
```

Её можно свести к зависимости по данным:

```
S1: x := cos(z)
S2: if x > 0 then a := b + c
S3: if x <= 0 then a := b - c
```

2. Зависимость по ресурсам

```
S1: x := 2 y / z
S2: c := a / b
```

Если в параллельной вычислительной системе есть только один исполнитель, умеющий делить, то распараллеливание невозможно.

Зависимости во вложенных циклах

Будем рассматривать вложенные циклы вида:

do $j_1=1,u_1$

do $j_2=1,u_2$

...

```
                do   j_n=1,u_n
                      Тело цикла
                end do
            . . .
end do
```

В таких циклах итерация определяется совокупностью значений всех счётчиков j_1, j_n .

Набор таких значений будем рассматривать как n-мерный вектор $J=(j_1,\dots,j_n)$.

Вектор J называют итерационным вектором.

Множество всех допустимых значений итерационных векторов образует итерационное пространство цикла. В этом пространстве можно ввести отношение порядка между векторами.

Будем говорить, что $I=J$ (два итерационных вектора равны), если
$\forall k\ (1\le k\le n),\ i_k=j_k$

Будем говорить, что $I<J$ (один итерационный вектор меньше другого), если
$\exists S(1\le S\le n),\ \forall k(1\le k\le n)\ i_k=j_k,\ i_S<j_S$

Предположим, что тело цикла состоит из двух операторов S1 и S2. Наборы входных и/или выходных данных, в которых будет обращение к элементам одного и того же массива данных A с размерностью, совпадающей с количеством уровней вложенности циклов.

Пусть индексы массива могут принимать произвольные целочисленные значения, при этом для простоты предположим, что для оператора S1 элемент массива A принадлежит к выходным переменным оператора, а для оператора S2 – входным переменным:

```
do   j_1=1,u_1
     do   j_2=1,u_2
          . . .
               do   j_n=1,u_n
                    S1: A[f_1(J),...,f_n(J)]:=...
                    S2: ...:=...A[g_1(J),...,g_n(J)]
               end do
          . . .
end do
```

Где $f_k(J), g_k(J),\ 1\le k\le n$ есть целочисленные функции для n переменных.

Ставится задача выяснить возможность разбиения итерационного пространства такого цикла на так называемые *зоны ответственности* для параллельного выполнения. Для этого введём

понятие *вектора расстояний зависимости* или просто *вектора расстояний*.

Обозначается: $D=L-K$, где $(0,0,...,0)\le L\le(u_1,u_2,...,u_n)$,
$(0,0,...,0)\le K\le(u_1,u_2,...,u_n)$.

Это означает, что из вектора итераций, соответствующего итерации стока зависимостей вычитаем вектор итерации, соответствующий итерации источника зависимости.

```
do   i=1,u1
     do   j=1,u2
          S1: a[i, j] := b[i, j] * 2;
          S2: c[i, j] := a[i, j – 1] + 1;
     end do
end do
```

Развернём цикл в итерационном пространстве:

$S_1^{(1,1)}: a[1,1]:=b[1,1]*2$ (исток зависимости)

$S_2^{(1,1)}: c[1,1]:=a[1,0]+1$

$S_1^{(1,2)}: a[1,2]:=b[1,2]*2$

$S_2^{(1,2)}: c[1,2]:=a[1,1]+1$ (сток зависимости)

...

$S_1^{(2,1)}: a[2,1]:=b[2,1]*2$

$S_2^{(2,1)}: c[2,1]:=a[2,0]+1$

Здесь видно, что операторы $S_1^{(1,1)}$ и $S_2^{(1,2)}$ обращаются к одному и тому же элементу. При этом $S_1^{(1,1)}$ является *истоком зависимости*, а $S_2^{(1,2)}$ *– стоком*.

Поэтому $K=(1,1)$, $L=(1,2)$, $D=(0,1)$

Введём понятие вектора направлений. Его компоненты будут определяться следующим образом:

$$d_i=\begin{cases}=, & D_i=0\\ >, & D_i<0\\ <, & D_i>0\end{cases}$$

На основании вектора направлений можно определить тип зависимостей по данным между операторами.

Пусть $d=(=,=)$, $D=(0,0)$. И пусть имеется цикл, в котором вычисления осуществляются следующим образом: ...

```
do   i=1,u1
     do   j=1,u2
          S1: a[i, j] := b[i, j] * 2;
          S2: c[i, j] := a[i, j] + 1;
     end do
end do
```

$S_1^{(1,1)}: a[1,1] := b[1,1] * 2$

$S_2^{(1,1)}: c[1,1] := a[1,1] + 1$

$S_1^{(1,2)}: a[1,2] := b[1,2] * 2$

$S_2^{(1,2)}: c[1,2] := a[1,2] + 1$

...

$S_1^{(2,1)}: a[2,1] := b[2,1] * 2$

$S_2^{(2,1)}: c[2,1] := a[2,1] + 1$

Тип зависимости – истинная зависимость, но она не связана с циклом, то есть это зависимость в пределах одной итерации. Распараллеливание здесь возможно по любой компоненте итерационного вектора и по двум компонентам одновременно. В первом случае зоны ответственности нарезаются полосами в одном или другом направлении, а во втором случае зоны ответственности представляют собой прямоугольники в итерационном пространстве.

Если операторы в теле цикла поменять местами, то для нового примера, решающего другую задачу, изменится тип зависимости, а всё остальное останется справедливо. В этом примере также можно поменять местами внешний и внутренний цикл – на результаты вычисления это не повлияет.

Утверждение

Если многомерный цикл имеет $d=(=,=,\ldots,=)$, то цикл может быть распараллелен по произвольному количеству индексов без всяких ограничений. При этом циклы, соответствующие различным уровням вложенности, можно безопасно менять местами.

Пример

```
do   i=1,u1
     do   j=1,u2
          S: a[i, j] := a[i, j + 1] * 2;
     end do
end do
```

Развернём цикл:

```
(1,1) a[1,1] := a[1,2] * 2
(1,2) a[1,2] := a[1,3] * 2
...
(2,1) a[2,1] := a[2,2] * 2
(2,2) a[2,2] := a[2,3] * 2
...
```

Ясно, что вектор расстояний $D=(0,-1)$, $d=(=,>)$

В данном случае значения элементов массива сначала используются, а потом вычисляются заново. Это антизависимость.

Если рассматривать внутренний цикл как один большой оператор, то все зависимости будут скрыты внутри этого большого оператора и, следовательно, внешний цикл может быть распараллелен без ограничений по итерациям.

Распараллеливание по внутреннему циклу и по двум циклам одновременно может быть осуществлено при условии размножения необходимых входных данных перед выполнением операции.

Здесь также можно поменять местами внешний и внутренний циклы без изменения результатов вычислений. Ясно, что после этого вектор расстояний станет $D=(-1,0)$, $d=(>,=)$. Тип зависимости при этом не изменится, тогда без ограничений можно будет распараллелить внутренний цикл, а для распараллеливания внешнего цикла и распараллеливания по двум направлениям нужно дублировать входные данные.

Пример

```
do   i=1,u1
     do   j=1,u2
          S: a[i, j] := a[i + 1, j + 1] * 2;
     end do
```

```
end do
```

Развернём цикл:

```
(1,1) a[1,1] := a[2,2] * 2
(1,2) a[1,2] := a[2,3] * 2

...
(2,1) a[2,1] := a[3,2] * 2
(2,2) a[2,2] := a[3,3] * 2

...
```

$D=(-1,-1)$, $d=(>,>)$

Значения элементов массива снова сначала используются, а затем заново вычисляются. Ясно, что это антизависимость. Распараллеливание по внутреннем циклу, или внешнему циклу, или по двум циклам одновременно можно осуществить при условии копирования необходимых входных данных перед выполнением операций. Внешний и внутренний циклы можно поменять местами без изменения результатов вычисления и с сохранением анализа на возможность распараллеливания.

Утверждение

Пусть многомерный цикл имеет вектор направлений d, в состав которого входят элементы > и =. Тогда он может быть распараллелен без всяких ограничений по любому количеству индексов, соответствующим компонентам >, возможно при дублировании необходимых входных данных. Циклы, соответствующие различным уровням вложенности, можно безопасно менять местами.

Пример

```
do   i=1,u1
     do   j=1,u2
          S: a[i, j] := a[i, j – 1] * 2;
     end do
end do
```

Развернём цикл:

```
(1,1) a[1,1] := a[1,0] * 2
(1,2) a[1,2] := a[1,1] * 2
```

```
...
(2,1) a[2,1] := a[2,0] * 2
(2,2) a[2,2] := a[2,1] * 2

...
```

$D=(0,1)$, $d=(=,<)$

Значения элементов сначала вычисляются, а потом используются. Тип зависимости – истинная.

Если рассматривать цикл как один большой оператор, то все зависимости будут скрыты внутри него и, следовательно, внешний цикл может быть распараллелен по итерациям без ограничений. Распараллеливания по внутреннему циклы и по двум циклам одновременно невозможно.

В данном примере внешний и внутренний циклы можно менять местами без изменения результатов вычислений. Ясно, что после такого станет: $D=(1,0)$, $d=(<,=)$. При этом тип зависимости не изменится, и без проблем можно будет распараллелить внутренний цикл.

Пример

```
do   i=1,u1
     do   j=1,u2
          S: a[i, j] := a[i – 1, j – 1] * 2;
     end do
end do
```

Развернём цикл:

```
(1,1) a[1,1] := a[0,0] * 2
(1,2) a[1,2] := a[0,1] * 2

...
(2,1) a[2,1] := a[1,0] * 2
(2,2) a[2,2] := a[1,1] * 2

...
```

$D=(1,1)$, $d=(<,<)$

Тип зависимости – истинная. Внутренний цикл не содержит зависимостей и может быть распараллелен без ограничений, но при этом все исполнители, завершившие очередную

итерацию во внешнем цикле, должны начинать новую внешнюю итерацию синхронно. Это означает, что требуется барьерная синхронизация по окончанию внутреннего цикла.

Распараллеливания по внешнему циклу и по двум циклам невозможны, но, как и в предыдущем примере, внутренний и внешний циклы можно менять местами без изменения результатов вычислений.

Пусть многомерный цикл имеет вектор направлений, в состав которого входят только элементы <,=. Такой цикл может быть распараллелен без ограничений по любому количеству индексов, соответствующих компонентам =. Распараллеливание по индексам, соответствующим компонентами < проблематично, а именно, может потребоваться барьерная синхронизация. Перед распараллеливанием циклы, соответствующие различным уровням, можно менять местами.

4.05.2016

Пример

```
do   i=1,u1
     do   j=1,u2
          S: a[i, j] := a[i + 1, j – 1] * 2;
     end do
end do
```

Развернём цикл:

```
(1,1) a[1,1] := a[2,0] * 2
(1,2) a[1,2] := a[2,1] * 2
...
(2,1) a[2,1] := a[3,0] * 2
(2,2) a[2,2] := a[3,1] * 2
...
```

$D=(-1,1)$, $d=(>,<)$

Здесь значения элементов сначала используются, а затем определяются. Вид зависимости – антизависимость.

При этом внутренний цикл не содержит зависимостей и может быть без ограничений распараллелен. Распараллеливание по внешнему циклу или по двум одновременно допустимо при предварительном резервировании входных данных.

В этом примере невозможно поменять местами внешний и внутренний циклы без изменения результатов вычислений.

Пример

```
do   i=1,u1
     do   j=1,u2
          S: a[i, j] := a[i – 1, j + 1] * 2;
     end do
end do
```

Развернём цикл:

```
(1,1) a[1,1] := a[0,2] * 2
(1,2) a[1,2] := a[0,3] * 2
...
(2,1) a[2,1] := a[1,2] * 2
(2,2) a[2,2] := a[1,3] * 2
...
```

$D=(1,-1)$; $d=(<,>)$

Здесь значения элементов массива сначала определяются, а затем используются, то есть это потоковая зависимость. При этом внутренний цикл не содержит зависимостей и может быть распараллелен без ограничений, при этом требуется барьерная синхронизация по окончании внутреннего цикла.

Распараллеливание по внешнему циклу невозможно.

Итоги всех примеров

Рассмотренные примеры позволяют сделать следующий вывод:

Для произвольного цикла возможно распараллеливание по любому индексу, соответствующему компоненте "=" в векторе направления. Уровень вложенности, соответствующий этой компоненте, можно поменять местами с любым соседним уровнем вложенности с сохранением результатов вычислений. Два соседних уровня вложенности, которым соответствуют одинаковые компоненты вектора направлений, также можно поменять местами.

Если в цикле существует антизависимость, то распараллеливание возможно по

произвольному количеству индексов при дублировании необходимых входных данных.

Распараллеливание в циклах с истинной зависимостью может оказаться проблематичным, то есть здесь необходим дополнительный анализ.

Для цикла, в котором есть зависимости по элементам нескольких массивов, решение о возможности распараллеливания принимается по результатам анализа всей совокупности зависимостей.

Эквивалентные преобразования циклов

Пример

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

Вектор расстояния и вектор направления:

$D=(0,1,-1)$

$d=(=,<,>)$

Истинная зависимость. Распараллеливание возможно либо по i, либо по k, либо по обоим вместе без ограничений.

Предположим, что, по соображениям математической модели, эффективно распараллеливание по индексу k (например для правильной балансировки вычислений).

Тогда по окончании цикла по k необходима барьерная синхронизация исполнителей. Ясно, что такая операция требует определённых затрат времени и приводит к увеличению накладных расходов на распараллеливание, уменьшая таким образом полученное ускорение.

Количество таких операций (операций барьерной синхронизации) $u_1 \times u_2$.

В то же время для этого примера можно изменить порядок циклов следующим образом:

```
do   j=1,u2
     do   k=1,u3
```

```
        do   i=1,u1
             S: a[i, j, k] := a[i, j – 1, k + 1] * 2
        end do
    end do
end do
```

$D=(1,-1,0)$

$d=(<,>,=)$

Анализ распараллеливания не меняется. По параметру k при распараллеливании всё равно понадобится барьерная синхронизация, но теперь таких операций понадобится только u_2 . Следовательно, накладные расходы будут сокращены, эффективность параллельной программы увеличена.

Рассмотренное преобразование цикла приводит к изменению динамического порядка выполнения оператора развернутой конструкции без изменения графа алгоритма и является примером эквивалентного преобразования программ.

Таким образом, эквивалентное преобразование программы – это изменение динамического порядка его операторов, сохраняющее граф алгоритма.

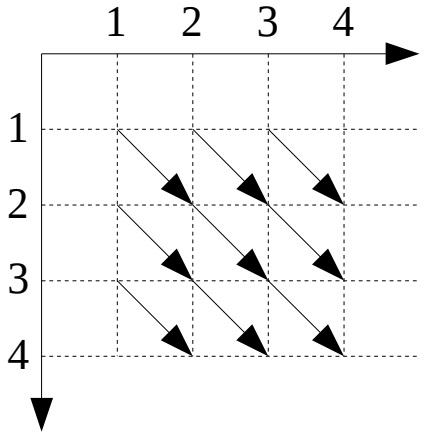
Любое изменение динамического порядка оператора последовательной программы, сохраняющее все зависимости по данным, не изменяют граф алгоритма программы.

Такое преобразование сохраняет в программе порядок всех операций обращения к памяти (чтение/запись), за исключением может быть операции ввода/вывода.

Пример

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

Если развернуть итерационное пространство этого цикла и изобразить зависимости между элементами на плоскости, то получим следующее:



(стрелочками изображена зависимость по данным)

Видно, что зависимости по данным расположены по диагоналям, при этом каждая диагональ может быть выполнена исполнителем вне зависимости от другой диагонали. Таким образом, сделав соответствующую замену индексных переменных циклов, сохраняющую граф алгоритма, код можно распараллелить и обеспечить правильный баланс загрузки исполнителей.

В этом примере зависимость не зависит от цикла.

Некоторые стандартные приёмы распараллеливания циклов

1. Разделение цикла

```
do   i=1,u1
      S1: a[i] := d[i] + 5 * i
      S2: c[i] := a[i − 1] * 2
end do
```

Здесь расстояние зависимости равно 1. Тип зависимости: истинная зависимость. По теории такой цикл не распараллеливается.

Однако этот цикл можно разбить на два:

```
do   i=1,u1
      S1: a[i] := d[i] + 5 * i
end do
do   i=1,u1
```

```
     S2: c[i] := a[i − 1] * 2
end do
```

Каждый из полученных циклов может быть распараллелен без ограничений. Выполненное преобразование не изменяет зависимости между операторами программы в целом, и следовательно допустимо. Но необходимо помнить о барьерной синхронизации между циклами.

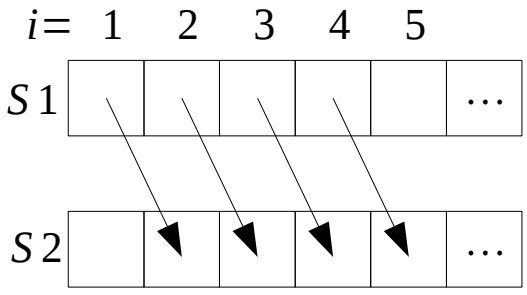
Такое распределение по двум циклам возможно, если между операторами в теле цикла нет рекурсивных зависимостей.

Например, для следующего цикла такой приём неприменим:

```
do   i=1,u1
     S1: a[i] := c[i − 1] + 5 * i
     S2: c[i] := a[i − 1] * 2
end do
```

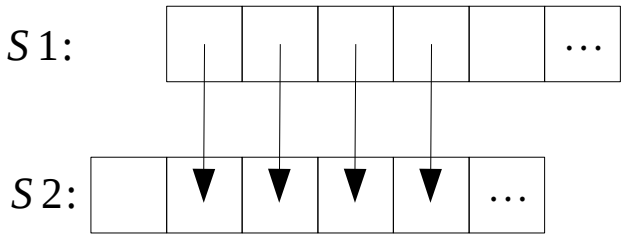
2. Выравнивание цикла

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



Стрелками показана зависимость

Сделаем так:



Преобразованный код программы:

```
do i = 0, u
     S1: if (i > 0) a[i] := d[i] + 5 * i
     S2: if (i < u) c[i + 1] := a[i] * 2
end do
```

Либо сделать так:

```
c[1] := a[0] * 2
do i = 1, u - 1
     S1: a[i] := d[i] + 5 * i
     S2: c[i + 1] := a[i] * 2
end do
a[u] := d[u] + 5 * u
```

Полученный цикл может быть распараллелен по итерациям, так как зависимость локализована в теле цикла и расстояние зависимости равно единице.

Однако наличие двух и более зависимостей между одними и теми же операторами цикла делает выравнивание невозможным.

Например, для следующего цикла выравнивание применить нельзя:

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

3. Репликация кода

Чтобы выполнить распараллеливание последнего примера, введём временную переменную и несколько ухудшим исходный код, заставляя программу выполнять определённые операции дважды.

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

Такое преобразование не изменяет граф алгоритма, но при этом получается зависимость, локализованная внутри одной итерации, следовательно, преобразованный цикл можно распараллелить.

Для не вложенных циклов приёмы разделения цикла, выравнивания и допустимая перестановка операторов в теле цикла достаточны для устранения зависимостей, связанных с циклом, если выполнены условия:

1. В теле цикла нет рекурсивной зависимости.
2. Расстояние для каждой зависимости есть величина постоянная, не зависящая от индекса цикла.

Подобные приёмы можно обобщить и на случай вложенных циклов.

Помимо зависимостей по элементам массивов могут встречаться зависимости по скалярным переменным, препятствующие распараллеливанию циклов. В некоторых случаях от них можно избавиться, не всегда сохраняя при этом граф алгоритма. Но необходимо быть осторожными, так как в результате распараллеливания программа может выдавать результаты, отличные от последовательной версии.

11.05.2016

4. Приватизация скалярных переменных

Пример

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

```
end do
```

По векторным переменным антизависимость, локализованная в пределах одной итерации. Скалярная переменная t препятствует распараллеливанию, что наиболее очевидно на машинах с общей памятью.

Справиться с задачей распараллеливания выполнения этого цикла можно, если на каждой итерации завести собственную переменную, то есть:

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

5. Индукционные переменные

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

Между всеми итерациями существует истинная (потоковая) зависимость по переменной j. С точки зрения математики понятно, что $j = k \cdot n$ на n-ой итерации. Поэтому можно записать:

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

Этот цикл можно распараллеливать, но эта операция рискованная, потому что граф алгоритма изменяется. То есть такой приём надо использовать осторожно.

Переменная j здесь называется *индукционной переменной*. А вообще индукционными переменными называют такие, значения которых на n-ой итерации представляют собой функции от номера итерации и, возможно, некоторого начального значения, присвоенного этой переменной вне цикла.

6. Редукционные операции

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

Здесь между всеми срабатываниями оператора существует истинная зависимость по скалярной переменной j, и следовательно, распараллеливание невозможно. Однако операция сложения является редукционной, то есть можно посчитать парные суммы на количестве процессоров $u/2$, или можно посчитать суммы по четыре элемента на количестве процессоров $u/4$, и так далее. Этот процесс даёт выигрыш по сравнению с последовательными вычислениями, но он изменяет граф алгоритма. Приём можно применять ко всем операциям, которые обладают свойством ассоциативности (ассоциативность: $(a+b)+c=a+(b+c)$), но пользоваться этим приёмом надо осторожно.

Пример (когда редукция невозможна)

$10^{20}-10^{20}+10^{-20}=10^{-20}$

$10^{20}+10^{-20}-10^{20}=0$ (из-за разности порядков при выполнении первой суммы 10^{-20} обращается в ноль – проверил на компе, это действительно так :))