ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Б.Я. Штейнберг

ИНФОРМАЦИОННЫЕ ЗАВИСИМОСТИ И ВЫСОКОУРОВНЕВЫЕ РАСПАРАЛЛЕЛИВАЮЩИЕ ПРЕОБРАЗОВАНИЯ ПРОГРАММ

Учебные материалы к спецкурсу
«Параллельные вычисления и преобразования программ»

Ростов-на-Дону
2007

СОДЕРЖАНИЕ

1. ВВЕДЕНИЕ

2. РАСПАРАЛЛЕЛИВАЮЩИЕ/ОПТИМИЗИРУЮЩИЕ ПРЕОБРАЗОВАНИЯ ПРОГРАММ

ПРОГРАММНЫЕ ЗАВИСИМОСТИ И БАЗОВЫЕ ПРЕОБРАЗОВАНИЯ
- Управляющий граф программы
- Информационная зависимость в программе
- Базовые понятия о преобразованиях программ
- Поиск и удаление недостижимых и неиспользуемых по управлению операторов
- Пустой оператор. Удаление и вставка операторов
- Приведение подобных (приведение выражения к стандартному виду)

ПРЕОБРАЗОВАНИЯ ЛИНЕЙНЫХ УЧАСТКОВ
- Перестановка фрагментов программы
- Подстановка вперед
- Вынос вычисления выражений и замена общих подвыражений
- Переименование безиндексных переменных

ПРЕОБРАЗОВАНИЯ ЦИКЛОВ
- Раскрутка цикла
- Удаление оператора (заголовка) цикла
- Канонизация циклов
- Расщепление одномерного цикла
- Разрыв итераций, гнездование цикла и подобные им преобразования
- Развертка цикла
- Расщепление вершин (Введение временных массивов)
- Растягивание скаляров (Повышение размерности массива)
- Разбиение цикла
- Слияние циклов
- Приведение цикла к разбиваемому виду
- Инверсия цикла
- Вынос оператора из цикла
- Индуктивные и охватывающие переменные

РЕШЕТЧАТЫЙ ГРАФ, КАК МОДЕЛЬ ИНФОРМАЦИОННОЙ ЗАВИСИМОСТИ В ГНЕЗДАХ ЦИКЛОВ
- Линейный класс программ
- Пространство итераций гнезда циклов
- Решетчатый граф и его соотношение с графом информационных связей
- Перестановка циклов в тесном гнезде

РАСПАРАЛЛЕЛИВАНИЕ
- Разновидности распараллеливания программ
- Векторизация циклов
- Асинхронное параллельное выполнение блоков
- Асинхронное распараллеливание циклов с независимыми итерациями
- Асинхронное распараллеливание циклов с зависимыми итерациями
- Распараллеливание на архитектуру VLIW
- Распараллеливание на архитектуру SIMD

КОНВЕЙЕРНЫЕ ВЫЧИСЛЕНИЯ
- Граф вычислений программы
- Граф вычислений и информационные зависимости
- Отображение цикла на архитектуру компьютера
- Синхронизация конвейера

3. ПАРАЛЛЕЛЬНОЕ РЕШЕНИЕ СИСТЕМ СЛАУ С ЛЕНТОЧНЫМИ МАТРИЦАМИ И РАСПАРАЛЛЕЛИВАНИЕ РЕКУРРЕНТНЫХ ЦИКЛОВ

- СЛАУ с двухдиагональными матрицами
- Применения алгоритма параллельного решения системы уравнения с ленточной двухдиагональной матрицей
- СЛАУ с блочными двухдиагональными матрицами
- СЛАУ с ленточными треугольными матрицами
- СЛАУ с блочной трехдиагональной нетреугольной матрицей
- СЛАУ с ленточной матрицей с преобладанием по главной диагонали
- Метод окаймления
- Параллельное вычисление дробнолинейных рекуррентных последовательностей
- Параллельный алгоритм LU-разложения трехдиагональной матрицы
- Распараллеливание рекуррентных циклов
- Циклы с нелинейной рекуррентностью
- Рекуррентные циклы, не имеющие эффективного распараллелливания.
- Распараллеливание алгоритмов решения СЛАУ для компьютеров с сегментированной памятью
- Оценки погрешностей решения СЛАУ с двухдиагональной матрицей

4. СЕТИ ПЕТРИ

- Определение сети Петри
- Граф разметок
- Свойства и виды сетей Петри
- Бесконечные метки
- Покрывающее дерево сети Петри
- Полное покрывающее дерево
- Проблемы ограниченности и безопасности
- Анализ сетей Петри
- Задача о пяти китайских обедающих философах
- Описание взаимодействия параллельно выполняемых процессов с помощью сетей Петри

СПИСОК ЛИТЕРАТУРЫ

СПИСОК СОКРАЩЕНИЙ

МВС – многопроцессорная вычислительная система

ПЭ – процессорный элемент

ПЛИС – программируемая логическая интегральная схема

ОРС – открытая распараллеливающая система

MIMD – many instructions, multiple data

SIMD – single instructions, multiple data

SPMD – single program multiple data

VLIW – vary long instruction word

SUIF – Stanford University Intermediate Format

MPI – massage passing interface

SMP – Symmetric Multi Processing

FPGA – Field Processing Gate Array – ПЛИС

1. ВВЕДЕНИЕ

Данный электронный учебник посвящен параллельным вычислениям, которые, особенно в последние годы, занимают все большую долю в теории и практике программирования.

Закон Мура об ускорении быстродействия вычислительной техники подтверждается уже десятилетиями. Но изменилось не только быстродействие компьютеров, но и соотношение времени на подготовку данных и времени на их обработку. А это означает, что разные программы стали работать быстрее в разной степени.

У сравнительно массового отечественного суперкомпьютера 80-х годов ПС-2000 обращение к памяти (чтение/записать) длилось 3 такта, сложение 1 такт, пересылка числа из одного процессорного элемента в другой (соседний) по кольцевому сдвиговому регистру 1 такт. В современных массовых процессорах умножение в 15 раз быстрее обращения к оперативной памяти и на два порядка быстрее пересылки по сети. Если на ПС-2000 был смысл распараллеливать цикл, вычисляющий сумму 100 слагаемых, то на кластере даже из двух компьютеров такое распараллеливание вместо ускорения вызовет замедление.

Таким образом, соотношение времён подготовки и обработки данных влияет на эффективность распараллеливающих преобразований.

Переменчивы приоритеты и в скалярной оптимизации. Рассмотрим вопрос об эффективности замены оператора

```
X(i+2) = X(i+2)+5
```

следующим кодом

```
k = i+2
X(k) = X(k)+5
```

Такая замена уменьшает количество операций, но вводит новую переменную k. Если для этой переменной не окажется свободного регистра, то придется обращаться к памяти и потерять быстродействие существенно большее, чем от выигрыша одной операции. (На старых компьютерах такая замена однозначно давала ускорение и использовалась в пакетах прикладных программ).

На одних параллельных компьютерах могут эффективно выполняться одни программы, а на других компьютерах – другие. Векторное выполнение следующего цикла равносильно его последовательному выполнению.

```
For I = 1 to N do
X[i] = 2*X[i+1]
```

Асинхронное параллельное выполнение этого же цикла может дать и другие результаты, например, если вторая итерация завершится до начала первой ите-

рации. С другой стороны условные операторы в теле цикла препятствуют векторизации, но не препятствуют асинхронному параллельному выполнению.

На конвейерные архитектуры эффективно отображаются другие программные циклы. Например, эти циклы могут допускать рекуррентные зависимости, которые препятствуют распараллеливанию и на векторных и на многоядерных компьютерах.

Большой вклад в многообразие архитектур вносят ПЛИС (программируемые логические интегральные схемы). Все чаще появляются сообщения об ускорителях и суперкомпьютерах с программируемой архитектурой.

Повышается не только многообразие архитектур, но и их сложность. Глядя на две разных программных реализации задачи не всегда можно угадать, какая из них будет работать быстрее. Нужен опыт – запустить программы – и посмотреть.

Итак, многообразие архитектур приводит к тому, что понятие эффективной программы или алгоритма становится архитектурно ориентированным. Это, в свою очередь, приводит к тому, что, либо новые архитектуры следует снабжать автоматически распараллеливающими компиляторами, либо программистов следует учить многовариантности программирования. Последнее означает, что, получая задачу, программист должен видеть не один вариант алгоритма и реализации программы, а целое семейство, из которого выбирать под конкретную архитектуру.

Теория преобразования программ как раз является, с одной стороны, математическим аппаратом для разработки распараллеливающих компиляторов, а, с другой стороны, средством развития видения многовариантности реализации программы.

Данный электронный учебник состоит из трех частей, которые можно изучать независимо: теория распараллеливающих и оптимизирующих преобразований программ, параллельные алгоритмы решения СЛАУ с ленточными матрицами и сети Петри.

Теория преобразования программ ориентирована на преобразование последовательных программ в параллельные и на развитие многовариантного представления о программной реализации алгоритма.

Параллельные алгоритмы решения систем линейных алгебраических уравнений с ленточными матрицами ориентированы на развитие параллельного алгоритмического мышления, поскольку прямые методы для решения таких систем сводятся к вычислению циклов с рекуррентными зависимостями, которые, казалось бы, представляют собой исключительно последовательные процессы.

Сети Петри интересны с точки зрения развития представлений о недетерминированных процессах. Представления о недетерминированности очень важны при программировании, например, многоядерных или кластерных архитектур. Но развивать такие представления проще на сетях Петри, которые представляют наглядные картинки с изображением графов. Функционирование сетей Петри иллюстрируется анимацией – данная глава учебника содержит два десятка технических мультфильмов. Завершается эта глава моделированием

(описанием) с помощью сетей Петри некоторых видов параллельных процессов.

Этот учебник не претендует на полноту. Знания о параллельных вычислениях могут быть пополнены из многочисленных источников, часть которых приводится в списке литературы.

2. РАСПАРАЛЛЕЛИВАЮЩИЕ/ОПТИМИЗИРУЮЩИЕ ПРЕОБРАЗОВАНИЯ ПРОГРАММ

Автоматическая оптимизация программ – одна из наиболее горячих точек развития науки. Действительно, во всем Мире в каждый момент времени работают миллионы программ. Предположим, что среди них 100 000 программ такие, что пользователи ждут окончания их работы. Предположим, что все компиляторы в Мире станут выдавать код всего лишь на 1% более быстрый, чем прежде. Это означает, что сто тысяч программ за сутки сэкономят около четверти часа. Итого, экономия порядка 24 000 человеко-часов в сутки! А если внедренные оптимизирующие преобразования можно будет переносить на будущие компьютеры, то эффект об их разработки астрономический.

Теория оптимизирующих преобразований программ насчитывает несколько десятилетий. Десятилетиями уже может исчисляться и практика применения этих преобразований в компиляторах, как в нашей стране, так и за рубежом. При этом описываются преобразования, как правило, на примерах высокоуровневых программ. Более всего литература по теории преобразования программ ориентируется на программы языка ФОРТРАН. А в компиляторах используются, в основном, низкоуровневые преобразования. Чаще всего это преобразования ассемблерных текстов в трехадресном внутреннем представлении. Этот дуализм, видимо, связан с тем, что теорию легче писать (и читать!) на языке высокого уровня. А ассемблер – это промежуточный язык, который, с одной стороны, удобен для отображения в него языка высокого уровня, а с другой стороны, для генерации кода. И чем ближе язык по уровню к архитектуре, тем более оптимально можно к архитектуре подогнать программу.

С течением времени накапливаются все новые и новые преобразования программ. В старых компиляторах блок оптимизации занимал сравнительно небольшую долю всего программного кода, по сравнению с парсером и генератором кода. Теперь блок оптимизации может занимать объем превосходящий остальную часть компилятора. По уровню использованного интеллекта блок оптимизации никак не уступает остальной части компилятора, особенно в последние годы при появлении автоматически распараллеливающих преобразований. Это приводит к проблеме понижения стоимости компиляторов при повышении их сложности.

Проблема удешевления компиляторов становится все более острой с расширением разнообразия вычислительных архитектур. Это разнообразие диктуется ростом потребности в высокопроизводительных вычислениях для различных прикладных задач в таких областях, как обработка сигналов, искусственный интеллект, научные исследования, управление большими базами данных, управление сложными объектами и других. При этом различные задачи могут предъявлять различные требования к архитектуре используемого компьютера: целые или вещественные числа, длина разрядной сетки, синхронность или асинхронность, конвейерность и т.д. Разнообразие малосерийных высоко-

производительных вычислительных архитектур реализуется в значительной степени на основе ПЛИС – программируемых логических интегральных схемах, которые в последние годы набирают все большую популярность.

Очевидным способом удешевления компиляторов является их переносимость, хотя бы частичная. В частности, переносимость блока оптимизации. Чем выше уровень внутреннего представления, на котором оптимизирующие преобразования реализованы, тем легче они переносимы. Разумеется, речь может идти только о машинно-независимых оптимизациях. Машинно-ориентированные преобразования могут быть дописаны к каждому конкретному компьютеру отдельно. Чем больше преобразований можно поднять на высокий уровень, тем легче можно будет создавать семейство компиляторов на их основе. Например, по такому пути пошли разработчики распараллеливающих компиляторов фирмы Portland Group, взяв за основу высокоуровневое внутреннее представление SUIF. Высокоуровневое внутреннее представление для преобразований программ лежит и в основе ОРС – Открытой распараллеливающей системы, которая разрабатывается в Южном федеральном университете www.ops.rsu.ru.

ПРОГРАММНЫЕ ЗАВИСИМОСТИ И БАЗОВЫЕ ПРЕОБРАЗОВАНИЯ

УПРАВЛЯЮЩИЙ ГРАФ ПРОГРАММЫ

Управляющий граф [**16**], [**21**], [**22**] – это одна из основ преобразования программ. Даже если в описании преобразования управляющий граф не упоминается, то он, как правило, подразумевается. В явном или неявном виде управляющий граф присутствует во внутренних представлениях оптимизирующих/распараллеливающих компиляторов.

Перед описанием преобразований и управляющего графа программ следует определиться, о каких программах будет идти речь.

Программы – это предложения языков программирования. В научной литературе высокоуровневые преобразования более всего описаны для программ языка ФОРТРАН. Встречаются также описания преобразований для языков Си и Паскаль, реже – для других высокоуровневых языков. Абстрагируемся от этих конкретных языков, чтобы преобразования носили как можно более общий характер. Программу будем рассматривать, как последовательность единиц языка. Будем считать, что рассматриваемые программы содержат следующие операторы:

операторы присваивания,
циклы со счетчиком (параметром, считающим итерации),
блочные условные операторы,
операторы безусловного перехода на метку GOTO,
пустые операторы,
Блочные операторы (операторные скобки),
Операторы вызова подпрограммы (процедуры или функции).

Предполагается, что в рассматриваемом языке программирования допускается конкатенация операторов.

Данные описываются только константами, скалярными переменными и массивами.

Программы могут исполняться, изменяя этим самым состояние памяти компьютера.

Исполнение программы состоит в чтении единиц языка (операторов, линейных участков), сопоставлении переменных, входящих в выражения с их значениями и изменении значений переменных (соответствующих им состояний ячеек памяти). Последовательность чтения единиц языка определяется семантикой языка программирования. Если две единицы языка могут прочитаться при некотором исполнении программы одна после другой, то говорят, что они связаны передачей управления.

Фрагмент программы – это такая часть подряд идущих по тексту операторов, заключение в скобки которой приводит к программе, эквивалентной исходной.

Базовый блок или линейный участок программы – это фрагмент, состоящий только из операторов присваивания.

В частности, у программы выделяются входные данные и выходные данные. Две программы считаются эквивалентными, если при одних и тех же входных данных в результате исполнения выходные данные у этих программ тоже совпадают.

Вершины управляющего графа – это некоторые элементы языка программирования (в научной литературе чаще всего операторы или линейные участки), а дуги – это передачи управления.

Отметим, что ранее в литературе в качестве вершин управляющего графа предполагалось использовать только операторы программы. Такой подход вполне удобен для программ языка ФОРТРАН, но уже не удобен для языка Паскаль, в котором циклу с постусловием «repeat-until» неудобно сопоставлять одну вершину (вместо «repeat» или вместо «until»?). Если вершины управляющего графа – операторы, то управляющий граф, соответствующий линейному участку имеет вид простого пути. Поэтому такие фрагменты программы и называются «линейными участками». Иногда линейные участки рассматривают в качестве вершин управляющего графа, чтобы сократить размеры графа, сохраняя его топологию.

Если операторы программы связаны конкатенацией, то передача управления между ними происходит в порядке следования этих операторов. Условный оператор и оператор цикла имеют по две передачи управления.

В языках программирования встречаются сложные операторы, которые определяются через другие операторы. Обычно, такие операторы имеют вид

<сложный оператор> ::= $term_1$ $statement_1$ $term_2$ $statement_2$ … $term_n$ $statement_n$ $term_{n+1}$

Здесь $term_1$, $term_2$, … $term_n$, $term_{n+1}$ – некоторые зарезервированные слова языка, $statement_1$, $statement_2$, … $statement_n$ – операторы или выражения. Такой вид имеют операторы цикла, условные операторы, блочный оператор, оператор case в языке Паскаль, оператор switch в языке Си.

Пусть G ориентированный граф, вершинами которого являются единицы языка $term_1$, $statement_1$, $term_2$, $statement_2$,…, $term_n$, statement , $term_{n+1}$. Пусть этот граф имеет единственный источник и единственный сток $term_1$ и $term_{n+1}$ соответственно. Дуги этого графа будем считать передачами управления в сложном операторе.

Следует отметить, что при внешней схожести операторов case в языке Паскаль и switch в языке Си, передачи управления у них организованы по разному. Вершины, как бы одинаковые, а дуги связывают эти вершины по разному. Графы передач управления этих операторов не изоморфны.

Приведенное выше аксиоматическое определение передач управления в сложных операторах – элемент семантики этих операторов.

При преобразованиях программ следует контролировать сохранение синтаксической и семантической корректности.

Пример 1. Если в программе языка ФОРТРАН поменять местами заголовок цикла и последний оператор тела цикла, то результирующая программа будет содержать нарушения синтаксиса языка. Следует отметить, что от таких ошибок компиляторы, как правило, застрахованы, поскольку преобразования в них проводятся не над текстом, а над специальным внутренним представлением.

Конец примера.

Пример 2. Если скопировать фрагмент программы, содержащий оператор с меткой, то в результирующем фрагменте получится два оператора с одинаковой меткой, что нарушает семантическую корректность. Копирования операторов встречаются в таких преобразованиях циклов, как развертка, раскрутка, гнездование, расщепление.

Конец примера.

Причины такой некорректности часто связаны с пренебрежением к управляющему графу программы при проведении преобразований.

Вход фрагмента программы – это передача управления, начало которой находится вне рассматриваемого фрагмента, а конец – внутри фрагмента. Аналогично определяется выход фрагмента.

Пример 3. В данном примере у фрагмента, состоящего из одного оператора 999, по крайней мере, два входа: один от предыдущего присваивания, а другой – от условного оператора.

```
      IF BOOLEXPR THEN
      A = 1
999   X = 1
```

Конец примера.

Иногда перед выполнением преобразования рассматриваемый фрагмент необходимо заключать в операторные скоби. В работе [**54**] преобразуемые циклы приводятся к такому виду, в котором перед заголовком и после тела ставятся пустые операторы с метками, которые перехватывают передачи управления, входящие в рассматриваемый цикл и выходящие из него.

Пример 4. В этом примере все операторы информационно независимы. Управление передается от каждого предыдущего оператора к последующему. Но операторы 3 и 4 можно менять местами, а операторы 1 и 2 – нельзя. Заметим, что перестановка операторов 1 и 2 приводит к программе синтаксически и семантически корректной, но не эквивалентной исходной. По этой причине

иногда приходиться оговаривать, какие именно операторы входят в рассматриваемый фрагмент.

```
1       GOTO 2
2       X = 1
3       Y = 2
4       Z = 3
```

Конец примера.

Строгое определение корректности преобразования программ обсуждается в [**17**]. В данной работе исследуются эквивалентность или эффективность преобразований программ. Корректность преобразований если не оговаривается, то подразумевается.

При автоматических преобразованиях программ в оптимизирующих компиляторах программы, как правило, находятся в древовидных внутренних представлениях. И преобразования проводятся во внутреннем представлении и представляют собой преобразования деревьев. Этим самым устраняются некоторые некорректности, которые возможны при преобразованиях текста. Например, невозможно переставить местами тело цикла и заголовок цикла, поскольку заголовок цикла – это корень куста, на котором тело цикла находится, а переставлять можно только целые кусты.

ИНФОРМАЦИОННАЯ ЗАВИСИМОСТЬ В ПРОГРАММЕ

Вхождением переменной, как обычно, будем называть всякое появление переменной в тексте программы вместе с тем местом в программе, в котором эта переменная появилась. Всякому вхождению (при конкретном значении индексного выражения дл массивов) соответствует обращение к некоторой ячейке памяти. Если при этом обращении происходит запись в ячейку памяти, то такое вхождение называется генератором. Остальные вхождения называются использованиями.

Пример 5. В следующем операторе присваивания

```
A(I+B(J+2))=D(I-1,B(I))+3*A(1)-I+5
1  2  3 4        5 6    7 8        9     10
```

десять вхождений переменных (их номера проставлены снизу) – и только первое является генератором.

Конец примера.

Пример 6. В следующем выражении языка С

```
A = B++
```

Вхождение переменной B является одновременно и использованием и генератором.

Конец примера.

Говорят, что два вхождения порождают информационную зависимость, если при некоторых допустимых значениях индексных выражений они обращаются к одной и той же ячейке памяти.

Пример 7.

```
DO 99 I=1,100
99  A(I)=D(I,I-1)+A(I-1)
```

В операторе с меткой 99 вхождения переменной A информационно зависимы, поскольку оба обращаются к ячейке памяти A(10): вхождение A(I) на десятой итерации, а A(I-1) - на одиннадцатой итерации цикла.

Конец примера.

Граф информационных связей – это ориентированный граф, вершины которого соответствуют вхождениям, а дуга соединяет пару вершин (v,u), если выполняется одно из следующих условий:

1. Эти вхождения обращаются к одной и той же ячейке памяти (т.е. порождают информационную зависимость), причем вхождение v раньше, чем u и хотя бы одно из этих вхождений является генератором.

2. Вхождение u является генератором, а вхождение v принадлежит этому же оператору присваивания. Такие дуги будем называть тривиальными.

Генератор будем обозначать – out (output), а использование - in (input). Дуги графа информационных связей бывают трех типов в зависимости от типов инцидентных им вершин: out-in - истинная информационная зависимость (true dependence), in-out - антизависимость (antidependence), out-out - выходная зависимость (output dependence) [**8**, с.95].

Если информационная зависимость связывает два использования, то мы будем говорить об in-in зависимости. Эти зависимости не влияют на эквивалентность преобразований программ и используются только при распределении данных в параллельной памяти.

Иногда бывает трудно определить, существует или нет информационная зависимость между парой вхождений. При преобразованиях программ в таких случаях предполагают худшее – считают, что такая зависимость существует, и на графе соответствующие вершины соединяют дугой.

Пример 8.

```
DO 99 I=1,N
99  X(2*I)=X(2*I+k)
```

В этом примере, если число k нечетное, то между обоими вхождениями переменной X нет информационной зависимости; если k четное и отрицательное, то есть истинная информационная зависимость, делающая цикл рекуррентным; если k четное и неотрицательное, то имеет место антизависимость. Если до выполнения программы неизвестно, какое значение примет k к началу выполнения цикла, то и характер зависимости тоже нельзя определить. В этом случае в графе информационных зависимостей должны быть проведены обе дуги между двумя вхождениями переменной X.

Конец примера.

Пример 9.

```
      DO 333 J=2,N
      DO 333 I=2,N
      A(I-2,J)=X(2*I-1)
c        v1          v2
      X(2*I)=X(2*I+3)+A(I,J-1)
c       v3         v4          v5
      X(2*I+1)=A(J,I-1)
c       v6          v7
333   X(2*I+2)=A(I+K,J)+A(I-2,J)
          v8              v9          v10
```

Выпишем список всех нетривиальных дуг графа информационных связей этого цикла:

(v1;v7), (v1;v9), (v1;v10), (v4;v6), (v5;v1), (v6;v2), (v7;v1), (v8;v3), (v9;v1)

Если до выполнения программы неизвестно, какое значение примет K к началу выполнения цикла, то и характер зависимости между вхождениями v1 и v9 нельзя определить. В этом случае в графе информационной зависимости должны быть проведены обе дуги между этими двумя вхождениями.

Конец примера.

Пример 10.

```
      DO 333 I=1,N
333   A(I+2)=A(N-I)
c     v1     v2
```

Для данного цикла список дуг графа имеет вид:

v1 – v2

v2 – v1

Конец примера.

Пример 11. В следующем цикле дуга графа информационных связей ведет от первого вхождения A(I) ко второму, но не наоборот. Вхождения переменной X соединяются двумя дугами в обе стороны: дуга антизависимости, поскольку на каждой итерации сначала X используется в первом операторе тела цикла, а потом перезаписывается; дуга истинной зависимости, поскольку на

каждой итерации, начиная со второй, используется значение X, полученное на предыдущей итерации.

```
      DO 99 I = 1,N
      A(I) = B+X
      X = A(I)
   99 CONTINUE
```

Конец примера.

Информационная зависимость между вхождениями называется циклически независимой (loop independent dependence), если эти вхождения обращаются к одной и той же ячейке памяти на одной и той же итерации цикла. Иначе зависимость называется циклически порожденной (loop carried dependence) [**8**, с.96].

В предыдущем примере дуга зависимости между вхождениями A(I) циклически независима, дуга зависимости между вхождениями X, ведущая сверху вниз тоже циклически независима, а дуга, ведущая снизу вверх – циклически зависима.

Пусть v1 и v2 – два вхождения массива X в тело цикла со счетчиком I, связанные информационной зависимостью. Информационную зависимость между v1 и v2 будем называть РЕГУЛЯРНОЙ относительно этого цикла, если разность индексных выражений вхождений v1 и v2 не зависит от счетчика цикла I.

Особым образом строится граф зависимости фрагмента программы, содержащего вызовы процедур и функций. Операция (оператор) вызова процедуры или функции рассматривается, как одна вершина графа. Если после замены формальных переменных фактическими, в теле процедуры (функции), имеется генератор (использование) некоторой переменной X, то из данной вершины (в данную вершину) графа идут дуги, как из генератора (использования) X. Более точно, вызов процедуры следует временно заменить соответствующим текстом программы, построить граф зависимостей для рассматриваемого фрагмента с этим текстом, потом весь подграф, соответствующий вставленному тексту, стянуть в одну вершину (построить фактор-граф по подграфу).

В данной работе будем считать, что рассматриваемые фрагменты программы не содержат вызовов функций с побочными эффектами. То есть, если в каком либо выражении вызывается функция, то эта функция возвращает только одно основное значение и не меняет значения глобальных переменных. В противном случае вхождение вызова этой функции пришлось бы связывать информационными зависимостями с некоторыми вхождениями.

Очевидно, если графы вычислений двух программ изоморфны, как помеченные графы, то рассматриваемые программы эквивалентны.

Заметим, что если в графе информационных связей есть дуга истинной информационной зависимости, то этой дуге соответствует дуга в графе вычислений.

Скалярные переменные, которые не меняют своего значения в рассматриваемом фрагменте программы, называются внешними (по отношению к данному фрагменту).

БАЗОВЫЕ ПОНЯТИЯ О ПРЕОБРАЗОВАНИЯХ ПРОГРАММ

Преобразование программы – это замена одной программы другой программой.

Преобразование фрагмента программы – это замена одного фрагмента другим фрагментом.

Преобразованию фрагмента сопоставляется такое преобразование программ, что для каждой программы, содержащей исходный фрагмент, сопоставим программу, в которой этот исходный фрагмент заменен результирующим.

Эквивалентное преобразование – это преобразование, в котором результирующая программа эквивалента исходной.

Преобразование фрагментов будем называть эквивалентным, если соответствующее ему преобразование программ всегда эквивалентно. Аналогично, преобразование фрагментов будем называть корректным, если соответствующее ему преобразование программ всегда корректно.

Далее под преобразованиями программ будем понимать преобразования фрагментов.

Преобразования программ сводятся к элементарным преобразованиям, которые используются при редактированиях текстов. Это удаление, вставка, и замена фрагментов программы, а также удаление, вставка и замена подвыражений.

ПОИСК И УДАЛЕНИЕ НЕДОСТИЖИМЫХ И НЕИСПОЛЬЗУЕМЫХ ПО УПРАВЛЕНИЮ ОПЕРАТОРОВ

При разработке большой программы может случиться, что программа будет содержать избыточный код, в частности, такие участки, на которые исполнение программы никогда не выйдет. Теоретически, вследствие ошибок, могут появиться и такие участки, попав в которые, программа из них никогда не выйдет (зациклится). Например, такие ошибки возможны при некорректно выполненной специализации большой программы. (Специализация – это сужение функций программы, используется для ускорения выполнения полезных для конкретного приложения функций программы за счет удаления ненужных функций.)

Некоторые из описанных недостатков программы можно выявить, используя только управляющий граф программы.

Для управляющего графа программы следует построить конденсацию, т.е. фактор-граф по компонентам сильной связности. Источники на этом графе, не соответствующие входу программы, представляют недостижимые участки. Их следует удалить. После этого, в управляющем графе программы опять следует найти источники, не соответствующие входу – и их удалить также. Повторять эту процедуру до тех пор, пока в программе останется единственный источник, соответствующий входу. После этого, из исходной программы следует удалить участки, соответствующие удаленным вершинам фактор-графа.

Аналогично, на конденсации управляющего графа следует найти стоки, не соответствующие выходам программы. Этим стокам соответствуют участки зацикливания.

Следует отметить, что программа может зацикливаться и при других условиях, зависящих от входных данных. Такие зацикливания с помощью управляющего графа можно и не обнаружить.

Участки зацикливания, которым на конденсации (конденсация – это фактор-граф по компонентам сильной связности) управляющего графа соответствуют посторонние тупиковые вершины, естественно называть участками структурного зацикливания.

Аналогично, в программе могут быть недостижимые участки, которые не соответствуют недостижимым источникам конденсации управляющего графа.

Участки программы, которые соответствуют недостижимым из начала вершинам конденсации управляющего графа естественно называть структурно недостижимыми.

Итак, с помощью конденсации управляющего графа можно находить структурно недостижимые участки и участки структурного зацикливания.

ПУСТОЙ ОПЕРАТОР. УДАЛЕНИЕ И ВСТАВКА ОПЕРАТОРОВ

Абстрактный пустой оператор – это оператор, который «ничего не делает, ничего не меняет». Можно говорить о пустом операторе присваивания, либо пустом условном операторе, либо пустом оператор цикла. Но абстрактный пустой оператор лучше всего представлять себе, как пустой блочный оператор – блочный оператор, не содержащий ни одного оператора. Абстрактный пустой оператор будем обозначать EMPTY.

Пример 12. Следующий оператор присваивания не изменяет состояния памяти

```
X = X
```

Но вряд ли стоит этот оператор отождествлять с абстрактным пустым оператором, поскольку данное присваивание, если его не удалить, обращается к памяти и может потребовать регистр.

Пустой блочный оператор не обращается ни к какой памяти и, в зависимости от языка программирования FORTPAN, PASCAL или C, может иметь вид

```
CONTINUE

BEGIN
END
{
}
```

Можно считать, что EMPTY – это и есть пустой блочный оператор.

От языка следует потребовать, чтобы пустой блочный оператор не изменял состояние памяти программы и «не менял передачи управления».

Блочный оператор – имеет синтаксис скобок и содержит конкатенацию каких-либо операторов.

Пустой оператор присваивания. Это оператор присваивания, который не меняет состояния памяти. Например,

```
x = x
```

Аксиома. Если в программе пустой оператор присваивания заменить пустым блочным оператором EMPTY, то получится программа, эквивалентная исходной.

Пустой условный оператор – это условный оператор, у которого на ветках стоят пустые операторы.

```
IF P THEN
        EMPTY
ELSE
        EMPTY
ENDIF
```

или

```
IF P THEN
        EMPTY
ENDIF
```

Аксиома. Если в программе пустой условный оператор заменить пустым блочным оператором EMPTY, то получится программа, эквивалентная исходной.

Пустой оператор цикла – это либо цикл с пустым заголовком, либо цикл с пустым телом. (Напомним, что под оператором цикла понимается цикл DO языка ФОРТРАН).

Цикл с пустым телом – это оператор цикла с конечным числом итераций, у которого тело цикла – пустой оператор.

Цикл с пустым заголовком – это оператор цикла с нулевым количеством итераций.

Аксиома. Если в программе цикл с пустым заголовком заменить пустым блочным оператором EMPTY, то получится программа, эквивалентная исходной.

Отметим, что для циклов с пустым телом используются языки программирования, удовлетворяющие одной из следующих двух аксиом.

Аксиома. Если в программе цикл с пустым телом заменить пустым блочным оператором EMPTY, то получится программа, эквивалентная исходной.

Аксиома. Если в программе цикл с пустым телом заменить присваиванием счетчику цикла значения последней итерации, то получится программа, эквивалентная исходной.

Для привычных для нас языков программирования семантика соответствующих операторов обеспечивает выполнение этих аксиом пустых операторов.

Абстрактный пустой оператор можно формально определить следующим свойством

Аксиома (об удалении пустого оператора). Предположим, что пустой оператор входит в конкатенацию в блочном операторе и не является единственным оператором этого блока. Пусть в управляющем графе программы на этот оператор ведет только одна дуга, определенная конкатенацией (нет передач управления на метку этого оператора). Тогда удаление этого оператора приводит к программе, эквивалентной исходной.

Вставка пустого оператора является преобразованием, обратным к удалению. Поэтому условие вставки можно сформулировать следующим образом.

Теорема 1. (о вставке пустого оператора). Если после вставки пустого оператора его удаление является эквивалентным преобразованием, то вставка является эквивалентным преобразованием.

Доказательство очевидно.

Замечание. При вставке нового оператора присваивания следует обратить внимание на корректность. Нельзя вставить оператор с неописанной меткой или с меткой, которая уже существует у некоторого оператора.

Теорема 2. (о замене оператора присваивания пустым оператором). Предположим, что генератор оператора присваивания не используется никакими использованиями, т.е. в графе информационных зависимостей из этого генератора нет дуг. Тогда этот оператор присваивания можно заменить пустым оператором (сохранив метку, если такая была) и новая программа будет эквивалентна исходной.

Доказательство вытекает из определения эквивалентности программ.

Теорема 3. (о замене оператора присваивания пустым оператором). Пустой оператор можно заменить оператором присваивания, генератор которого нигде не используется (сохранив метку, если такая была) и новая программа будет эквивалентна исходной.

Доказательство вытекает из предыдущей теоремы.

Теорема 4. (об удалении оператора присваивания). Предположим, что оператор присваивания входит в блочный оператор и не является единственным оператором этого блока. Предположим, что на этот оператор ведет только одна передача управления, определенная конкатенацией, и из этого оператора ведет только одна передача управления (тоже определенная конкатенацией). Предположим, что генератор этого оператора присваивания нигде не используется. Тогда удаление этого оператора приводит к программе, эквивалентной исходной.

Доказательство вытекает из теоремы об удалении пустого оператора и теоремы о замене присваивания пустым оператором.

Вставка оператора является преобразованием, обратным к удалению. Поэтому условие вставки можно сформулировать следующим образом.

Теорема 5. Если после вставки оператора присваивания его удаление является эквивалентным преобразованием, то и вставка является эквивалентным преобразованием.

Доказательство очевидно.

Замечание. При вставке нового оператора присваивания следует обратить внимание на корректность. Нельзя вставить оператор с неописанной меткой или с меткой, которая уже существует у некоторого оператора.

Теоремы об удалении/вставке оператора присваивания можно очевидным образом обобщить на случай фрагмента программы.

Теорема 6. (об удалении фрагмента программы). Предположим, что фрагмент программы входит в блочный оператор и этот блок содержит еще хотя бы один оператор. Предположим, что у этого фрагмента есть единственный вход и единственный выход и они определяются конкатенацией. Предположим, что из вхождений этого фрагмента не выходят дуги истинной информационной зависимости (то есть генераторы этого фрагмента не используются за его пределами). Тогда удаление этого оператора приводит к программе, эквивалентной исходной.

Теорема 7. Если после вставки фрагмента программы его удаление является эквивалентным преобразованием, то и вставка является эквивалентным преобразованием.

Пример 13. Фрагмент программы

```
T = A+B
X = T+C
Y = T-C
write(Y)
```

равносилен следующему

```
      T = A+B
      Y = T-C
      write(Y)
```
Конец примера.

Пример 14. Фрагмент программы
```
      DO 99 I = 1, N
      IF (X > N) goto 88
      T = I*A
      X = T+C
      Y = T-C
   99   write(Y)
```
НЕ равносилен следующему
```
      DO 99 I = 1, N
      IF (X > N) goto 88
      T = I*A
      Y = T-C
   99   write(Y)
```
(Из генератора удаляемого оператора выходит дуга информационной зависимости).

Конец примера.

Пример 15. Фрагмент программы
```
      IF (X > N)
      D = T+C
      T = I*A
      Y = T-C
   99   write(Y)
```
НЕ равносилен следующему
```
      IF (X > N)
      T = I*A
      Y = T-C
   99   write(Y)
```
поскольку удаляемый оператор принадлежит фрагменту с двумя входами. Удаление оператора меняет зону действия условного оператора.

Конец примера.

ПРИВЕДЕНИЕ ПОДОБНЫХ (ПРИВЕДЕНИЕ ВЫРАЖЕНИЯ К СТАНДАРТНОМУ ВИДУ)

Для некоторых преобразований важно, чтобы выражения программы были приведены к некоторому стандартному виду. Особенно это необходимо при построении графа информационных связей. Без такого стандарта трудно понять, что вхождения X(2*I+1) X(1+I*2) X(2*I+2-1) X(I+1+I) информационно

зависимы. В данной работе предлагаются следующие стандартные требования к записи выражений.

1. Аргументы коммутативной бинарной операции должны следовать в алфавитном порядке. Например, P*Q а не Q*P.
2. Если в выражении некоторая одновременно и коммутативная и ассоциативная операция должна выполняться несколько раз подряд, то все аргументы должны быть отсортированы по алфавиту. Например, A+B-X (вычитание в этой записи следует понимать, как сложение с переменной с унарным минусом A+B+(-X)).
3. Сумма двух одинаковых выражений с (может быть разными) числовыми множителями заменяется таким выражением с множителем, который равен сумме исходных. 3*I-I => 2*I.
4. Можно выносить общий множитель-переменную, пользуясь законом дистрибутивности. Но при вызове такой процедуры следует указывать, какие множители нам следует выносить. Такая процедура может выполняться благополучно лишь тогда, когда исходное выражение линейно зависит от совокупности указанных переменных (если их несколько). Например, A*B+A*C+B*C => A*B+(A+B)*C, если выражение надо приводить относительно переменной C. Заметим, что в этом примере невозможно привести исходное выражение относительно переменных A и B одновременно, поскольку выражение нелинейно зависит от их совокупности. Если выносимый множитель является не константой, а переменной или подвыражением, то он пишется слева или справа в зависимости от алфавитного порядка с переменной, относительно которой множитель выносится.
5. При записи отношений будем использовать «<» вместо «>» и «≤» вместо «≥». (x+2)<0 вместо 0>(x+2).
6. Булевы выражения следует приводить к ДНФ (дизъюнктивной нормальной форме), в которой аргументы конъюнкций и дизъюнкций расположены в алфавитном порядке. При этом выражение без отрицания должно записываться раньше такого же выражения с отрицанием. Например, Not(A) and B and Z or B and Y or not(B) and C.
7. Арифметические подвыражения над константами следует заменять их числовыми значениями. Например, числом –5 можно заменить выражение (3-4*2).
8. Унарный минус «-» можно выносить, как множитель, если это особо оговорено в параметрах процедуры.

Пример 16.

X(I)*2-2*X(I-1) = > 2*X(I)-2*X(I-1)

X(I)*A(I)-2*X(I) => (A(I)-2)* X(I)

X(I)*Y(I)-2*X(I) => X(I) *(Y(I)-2)

Конец примера.

Для реализации пункта 4 требований к стандартному виду выражения может быть полезным предварительное раскрытие скобок по закону дистрибутивности.

Пример 17. 5*(X-Y)+(A+1)*(X+Y) => 5*X-5*Y+(A+1)*X+(A+1)*Y => (5+(A+1))*X +(-5+(A+1))*Y => (A+6)*X +(A-4)*Y
Конец примера.

Заметим, что приведение выражений к стандартному виду всегда является эквивалентным преобразованием.

ПРЕОБРАЗОВАНИЯ ЛИНЕЙНЫХ УЧАСТКОВ

ПЕРЕСТАНОВКА ФРАГМЕНТОВ ПРОГРАММЫ

Данное преобразование состоит в том, чтобы фрагмент программы

```
S1
..........
S(k-1)
Sk                                          (1)
S(k+1)
..........
Sm
```

заменить на фрагмент программы

```
S(k+1)
..........
Sm
S1                                          (2)
..........
S(k-1)
Sk
```

Здесь S1,...,Sm - операторы программы.

Теорема 8. Пусть для фрагментов программы S1,..., Sk и S(k+1),..., Sm операторы S1 и S(k+1) являются единственными точками входа, а операторы Sk и Sm - единственными точками выхода соответственно. Будем считать, что между этими фрагментами нет условных или безусловных переходов. Кроме того, будем предполагать, что для фрагмента S1,..., Sk, S(k+1),...,Sm оператор S1 является единственным входом, а Sm - единственным выходом. Предположим, что в графе информационных связей НЕ существует такой дуги циклически независимой зависимости (v1,v2), что v1 принадлежит одному из переставляемых фрагментов программы, а v2 - другому. Тогда рассматриваемая перестановка фрагментов программы эквивалентна.

Доказательство. Корректность перестановки вытекает из того, что каждый из фрагментов S1,...,Sk, S(k+1),...,Sm, S1,...,Sk и S(k+1),...,Sm имеют по одной точке входа и одной точке выхода.

Из отсутствия переходов между фрагментами вытекает, что не существует цикла, содержащего хотя бы по одному оператору из каждого из двух рассматриваемых фрагментов, который мог бы разрушиться от перестановки фрагментов. С другой стороны, перестановка этих фрагментов не создаст новых циклов.

Два фрагмента программы эквивалентны тогда и только тогда, когда при одних и тех же исходных данных на их выходе получаются одинаковые значения результирующих данных. Предположим, что фрагменты программы S1,...,Sk, S(k+1),...,Sm и S(k+1),...,Sm,S1,...,Sk неэквивалентны. Тогда найдется некоторая переменная, обозначим ее A, принимающая разные значения после выполнения этих фрагментов. Это возможно лишь в двух случаях:

1) последнее значение переменной A в этих фрагментах вычисляют разные операторы;

2) последнее значение переменной A вычисляет один и тот же оператор, но этот оператор использует разные значения некоторой переменной, которую будем обозначать B.

Первый случай означает, что в каждом из фрагментов S1,...,Sk и S(k+1),...,Sm есть операторы присваивания, вычисляющие A в один и тот же момент выполнения фрагмента S1,...,Sm. Но это означает существование такой дуги (v1,v2) типа out-out в графе информационных связей, что v1 принадлежит фрагменту S1,...,Sk, v2 - фрагменту S(k+1),...,Sm и оба вхождения обращаются к одной и той же ячейке памяти в один и тот же момент выполнения фрагмента S1,..., Sm (т.е. существует циклически независимая дуга). Противоречие.

Второй случай означает, что в одном из фрагментов S1,..., Sk или S(k+1),...,Sm, находится оператор, вычисляющий A и использующий B, а в другом - оператор, вычисляющий B. Но, как и в предыдущем случае, это означает существование такой дуги (v1,v2) типа in-out либо out-in в графе информационных связей, что v1 принадлежит фрагменту S1,...,Sk, v2 - фрагменту S(k+1),...,Sm и оба вхождения обращаются к одной и той же ячейке памяти (в данном случае B) в один и тот же момент выполнения фрагмента S1,...,Sm (это дуга циклически независимой зависимости).

Конец доказательства.

Необходимость условий данной теоремы проиллюстрируем следующими примерами.

Пример 18. Фрагмент программы

```
10   X(K) = A(I+1)
15   A(I+1) = 1
```

НЕ равносилен следующему

```
10   A(I+1) = 1
15   X(K) = A(I+1)
```

(Здесь зависимость между вхождениями переменной A является циклически независимой). Действительно, при начальных значениях I=1; K=1; A(2)=2; первый фрагмент вычисляет X(1)=2 , а второй X(1)=1.

Конец примера.

Пример 19. Фрагмент программы

```
10    X(K) = A(I+1)
15    A(I) = 1
```

равносилен следующему

```
10    A(I) = 1
15    X(K) = A(I+1)
```

(Здесь зависимость между вхождениями переменной A не является циклически независимой).

Конец примера.

Пример 20. Фрагмент программы

```
10    X(K) = A(I+1)
15    K = 1
```

НЕ равносилен следующему

```
10    K = 1
15    X(K) = A(I+1)
```

(Здесь зависимость между вхождениями переменной K является циклически независимой). Действительно, при начальных значениях I=1; K=2; A(2)=1; X(1)=2, первый фрагмент вычисляет X(1)=2 , а второй X(1)=1.

Конец примера.

Пример 21. Фрагмент программы

```
10    IF (C<0)
15    L = A(I+1)
20    B = X(K)
```

НЕ равносилен следующему

```
10    IF (C<0)
20    B = X(K)
15    L = A(I+1)
```

(Здесь в исходном фрагменте у оператора 20 два входа).

Конец примера.

Пример 22. Фрагмент программы

```
IF (C<0)
L = A(I+1)
B = X(K)
```

равносилен следующему

```
B = X(K)
IF (C<0)
L = A(I+1)
```

Конец примера.

Пример 23. Фрагмент программы

```
      K = X+1
      goto 99
      K = I-1
   99 A = K
```

НЕ равносилен следующему

```
      K = X+1
      K = I-1
      goto 99
   99 A = K
```

(Здесь у всего фрагмента из двух переставляемых операторов в целом два выхода).

Конец примера.

Пример 24. Фрагмент программы

```
      C = 1
      goto 99
      ...
88      C = 2
99   A = K
```

НЕ равносилен следующему

```
      C = 1
      goto 99
   ...
99    A = K
 88    C = 2
```

(Здесь у фрагмента из оператора 99 два входа. У результирующего фрагмента на выходе всегда C=2, а у исходного может быть и C=1.).

Конец примера.

Пример 25. Фрагмент программы

```
     write(F,10)
     write(F,20)
```

НЕ равносилен следующему

```
     write(F,20)
     write(F,10)
```

Здесь вхождения F связаны выходной зависимостью.

Конец примера.

Предыдущую теорему несложно обобщить на случай перестановки фрагментов программы, между которыми есть еще один фрагмент.

Теорема 9. Пусть имеется фрагмент программы S1,...,Sm , представляющий собой последовательность трех подряд идущих фрагментов S1,...,Sk; S(k+1),...,Sr; S(r+1),...,Sm . Операторы S1, S(k+1), S(r+1) являются единственными точками входа, а операторы Sk, Sr и Sm - единственными точками выхода этих фрагментов соответственно. Кроме того, будем предполагать, что для всего фрагмента S1,...,Sm оператор S1 является единственным входом, а Sm - единственным выходом. Предположим, что в графе информационных связей НЕ существует такой циклически независимой дуги (v1,v2), что v1 принадлежит фрагменту S1,...,Sk; а v2 - S(k+1),...,Sr; S(r+1),...,Sm или v1 принадлежит S1,...,Sk; S(k+1),...,Sr; а v2 принадлежит фрагменту S(r+1),...,Sm. Тогда рассматриваемая перестановка фрагментов программы S1,...,Sk; и S(r+1),...,Sm синтаксически корректна и эквивалентна.

Для доказательства следует дважды воспользоваться предыдущей теоремой.

Теорема 10. (о произвольной перестановке фрагментов в соответствии с дугами информационных зависимостей). Пусть имеется последовательность фрагментов программы $F_1,..., F_n$ и перестановка чисел s(i), i = 1,…, n . Пусть каждый из этих фрагментов имеет один вход и один выход. Предположим, что для любых двух чисел i, j = 1,…, n, i < j выполняется условие

- Если из F_i в F_j есть дуга циклически независимой информационной зависимости, то s(i) < s(j) .

Тогда последовательность фрагментов $F_{s(1)},..., F_{s(n)}$ эквивалентна исходной последовательности $F_1,..., F_n$.

Доказательство. Будем преобразовывать последовательность фрагментов $F_{s(1)},..., F_{s(n)}$ эквивалентными перестановками к исходной последовательности $F_1,..., F_n$. Рассмотрим такое число k , для которого s(k)=1. Покажем, что для любого j такого, что j<k, не существует дуги циклически независимой зависимости из $F_{s(j)}$ в $F_{s(k)}$. Действительно, поскольку в исходной программе $F_{s(k)} = F_1$ по тексту раньше, чем $F_{s(j)}$, то в $F_{s(k)} = F_1$ не могут входить дуги циклически независимой зависимости. Поэтому фрагмент $F_{s(k)} = F_1$ можно переставить на первое место (где он и был раньше) и будет получена программа равносильная исходной $F_1, F_{s(1)},..., F_{s(k-1)}, F_{s(k+1)}, …, F_{s(n)}$. Теперь можно показать, что равносильной перестановкой фрагментов можно на второе место в последовательности переставить фрагмент F_2 . И т.д. методом математической индукции можно показать, что из результирующей последовательности $F_{s(1)},..., F_{s(n)}$ равносильными перестановками может быть получена исходная последовательность фрагментов.

Для векторных компьютеров перестановка операторов в теле цикла иногда позволяет цикл представить в виде последовательности векторных команд. Это происходит вследствие того, что циклически порожденные дуги графа информационных связей, которые были направлены "снизу-вверх", теперь будут направлены "сверху-вниз".

Пример 26. Фрагмент программы

```
DO 99 I = 1, N
   X(I) = A(I+1)
   K(I) = X(I+1)
99 CONTINUE
```

равносилен следующему, поскольку зависимость циклически порожденная.

```
DO 99 I = 1, N
   K(I) = X(I+1)
   X(I) = A(I+1)
99 CONTINUE
```

При этом операторы второго цикла могут выполняться одновременно для всех итераций.

Конец примера.

Пример 27. Фрагмент программы

```
DO 99 I = 1, N
   X(I) = A(I+1)
   IF X(I)<0 THEN   K(I) = X(I)
99 CONTINUE
```

не равносилен следующему, поскольку зависимость циклически независима

```
DO 99 I = 1, N
   IF X(I)<0 THEN    K(I) = X(I)
   X(I) = A(I+1)
99 CONTINUE
```

Конец примера.

Перестановка операторов в теле цикла иногда позволяет разбивать цикл, в теле которого много операторов присваивания на несколько циклов, в телах которых меньше операторов. Это преобразование может использоваться, например, в компьютерах с конвейерной архитектурой - (если для конвейеризации большого цикла недостаточно ресурсов).

Пример 28. Не разрушит ли следующий пример теорему о перестановке операторов?

```
   S1
   .........
      goto 99

   S(k-1)
   Sk                                          (3)
99    S(k+1)
   .........
   Sm
```

заменить фрагментом программы

```
99    S(k+1)
      .........
      Sm
      S1                                              (4)
      .........
         goto 99

      S(k-1)
      Sk
```

Здесь S1,...,Sm - операторы программы.

Конец примера.

ПОДСТАНОВКА ВПЕРЕД

Это преобразование подставляет правую часть оператора присваивания вместо последующих вхождений левой части этого же оператора (если это не нарушает равносильность) [**8**, с 26]. В частном случае, когда правая часть оператора присваивания является константой, это преобразование называется протягиванием констант. В результате протягивания констант уменьшается код программы, повышается быстродействие и исчезают некоторые дуги информационной зависимости, что может способствовать распараллеливанию. Подстановка вместо переменных их значений в индексных выражениях позволяет точнее строить граф информационных связей.

Пример 29. Фрагмент программы

```
      10    X(K) = A(I+1) + B
      20    B = X(K)
```

равносилен следующему

```
   10    X(K) = A(I+1)+B
      20    B = A(I+1) + B
```

Конец примера.

Пример 30. Фрагмент программы

```
         goto 77
.................
         T = 5
      77 A = T+E
         B = 2*T
```

НЕ равносилен следующему

```
         goto 77
....................
         T = 5
      77 A = 5+E
         B = 2*5
```

Конец примера.

Теорема 11. Пусть у фрагмента программы

```
S1    X(F1) = expr(A,B,...)
        v1
.....                                          (5)
Sm    ... = ...X(F1)...
                  v2
```

F1 - какое-либо индексное выражение, одинаковое в обоих случаях. Пусть для рассматриваемого фрагмента программы выполняются условия:

1. S1 - единственная точка входа фрагмента.
2. В графе информационных связей дуга (v1,v2) между вхождениями X(F1), является единственной дугой этого фрагмента, входящей во вхождение v2 переменной X.
3. Нет дуг, входящих во вхождения индексного выражения F1 оператора Sm из вхождений данного фрагмента.
4. Нет дуг, выходящих из вхождений выражения expr(A,B,...) в генераторы операторов S1,..,S(m-1).

Тогда подстановка вперед является эквивалентным преобразованием.

Доказательство. После выполнения подстановки из исходного фрагмента программы получается фрагмент

```
      S1    X(F1) = expr(A,B,...)
.................................              (6)
      Sm    ... = ...expr(A,B,...)...
```

Очевидно, что для эквивалентности исходного и результирующего фрагментов достаточно, чтобы к моменту выполнения оператора Sm переменная X(F1) и выражение expr(A,B,...) принимали одинаковые значения. Предположим, что это неверно. Тогда к моменту выполнения оператора Sm либо вхождение X(F1), либо выражение expr(A,B,...) поменяло свое значение.

Предположим, что свое значение поменяло вхождение X(F1). Тогда, ввиду единственности входа у фрагмента, либо где-то в рассматриваемом фрагменте существует еще один генератор X(F1) , что противоречило бы условию 2 теоремы, либо изменило свое значение индексное выражение F1, но тогда в рассматриваемом фрагменте есть генератор какой-либо переменной, входящей в выражение F1. Наличие такого генератора означает, что в графе информационных связей из рассматриваемого фрагмента есть дуга в F1, что противоречит условию 3 теоремы.

Теперь предположим, что свое значение поменяло выражение expr(A,B,...). Тогда, где-то по пути от S1 до Sm свое значение поменяло какое-то вхождение этого выражения. Но это означает, что у операторов S2,..,S(m-1) есть генератор переменной из данного выражения, что противоречит условию 4 теоремы.

Полученные противоречия завершают доказательство теоремы.

Условия предыдущей теоремы не являются необходимыми. Действительно, если в операторе Sm вхождение X(F1) умножается на ноль, то никакие нарушения условий теоремы не препятствуют эквивалентности подстановки.

Пример 31. Фрагмент программы

```
        goto Sm
.................
S1   X(K) = A+B
          X(K) = 1
           A = T+E
Sm     B = 0*X(K) +C
```

равносилен следующему

```
        goto  Sm
.................
S1   X(K) = A+B
          X(K) = 1
          A = T+E
Sm     B = 0*(A+B) +C
```

хотя условия теоремы не выполнены.

Конец примера.

С другой стороны, если нарушается любое из условий теоремы, то существует такая программа, в которой выполняются все остальные условия, но подстановка неэквивалентна. В этом смысле условия теоремы являются необходимыми.

Пример 32. Фрагмент программы

```
      10    IF (C<0)
      15    X(K) = A(I+1)
      20    B = X(K)
```

НЕ равносилен следующему

```
      10    IF (C<0)
      15    X(K) = A(I+1)
      20    B = A(I+1)
```

(фрагмент из операторов 15-20 имеет 2 входа).

Конец примера.

Пример 33. Фрагмент программы

```
      10    X(K) = A(I+1)
      15    X(K) = 1
      20    B = X(K)
```

НЕ равносилен следующему

```
      10    X(K) = A(I+1)
      15    X(K) = 1
      20    B = A(I+1)
```

(В использование X(K) входят две дуги зависимости)

Конец примера.

Пример 34. Фрагмент программы

```
10    X(K) = A(I+1)
15    K = 1
20    B = X(K)
```

НЕ равносилен следующему

```
10    X(K) = A(I+1)
15    K = 1
20    B = A(I+1)
```

(В индексное значение использования X(K) входит дуга зависимости)

Конец примера.

Пример 35. Фрагмент программы

```
X(K) = A(I+1)
I = 1
B = X(K)
```

НЕ равносилен следующему

```
 X(K) = A(I+1)
 I = 1
B = A(I+1)
```

(Из вхождения в правую часть первого оператора есть дуга зависимости).

Конец примера.

ВЫНОС ВЫЧИСЛЕНИЯ ВЫРАЖЕНИЙ И ЗАМЕНА ОБЩИХ ПОДВЫРАЖЕНИЙ

Вынос общих подвыражений состоит в том, чтобы найти в тексте программы одинаковые подвыражения, вычислить их отдельно, присвоив значение этого подвыражения вспомогательной переменной, и заменить всюду по тексту данное подвыражение новой переменной. При выполнении преобразования следует проверять, во всех ли случаях вхождения в данное подвыражение имеют одинаковые значения.

Теорема 12. Пусть фрагмент программы

```
M  S1
   .........
   Sk       expr(A,B,...)
   .................                                    (7)
   Sm       expr(A,B,...)
```

имеет единственный вход – оператор S1. Пусть операторы Sk и Sm имеют одинаковые подвыражения expr(A,B,...). Предположим, что во вхождения переменных из любого выражения expr(A,B,...) не входят дуги информационной

зависимости из других вхождений этого фрагмента. Тогда этот фрагмент программы эквивалентен следующему, в котором вместо expr(A,B,...) подставлена переменная T, а если оператор S1 имел метку M, то эта метка должна быть переставлена оператору S0.

```
M  S0 T =  expr(A,B,...)
   S1
   .........
   Sk       T
   .................                                    (8)
   Sm       T
```

Доказательство. Заметим, что у нового фрагмента программы есть единственный вход – оператор S0. Если во фрагменте программы (8) выполнить подстановку вперед, а потом удалить оператор S0, то получим фрагмент (7). Поскольку фрагмент (8) удовлетворяет условию теоремы о подстановке вперед, а после выполнения подстановки вперед он будет удовлетворять условию теоремы об удалении оператора присваивания, фрагменты (7) и (8) эквивалентны. Конец доказательства.

Замечание. Очевидно, что можно выносить вычисление выражения перед фрагментом и из одного оператора или из нескольких (не только из двух, как приведено в формулировке теоремы).

Обсуждение целесообразности удаления общих подвыражений.

Рассматриваемое преобразование уменьшает код программы и используется не только в распараллеливающих компиляторах, но и в оптимизирующих. Следует отметить, что данное преобразование является в некотором смысле обратным к подстановке вперед.

Пример 36. Фрагмент программы

```
X = A+B+C
Y = A+B-C
```

равносилен следующему

```
T = A+B
X = T+C
Y = T-C
```

Конец примера.

Пример 37. Фрагмент программы

```
X = A+B+C
A = 1
Y = A+B-C
```

не равносилен следующему

```
T = A+B
X = T+C
```

```
      A = 1
      Y = T-C
```
Конец примера.

Пример 38. Фрагмент программы
```
      X = A+I+1
      Y = A-I-1
```
равносилен следующему
```
      T = I+1
      X = A+T
      Y = A-T
```
Конец примера.

Пример 39. Фрагмент программы (Алгоритм Флойда построения матрицы кратчайших расстояний графа)
```
      DO 77 K = 1, N
      DO 77 J = 1, N
      DO 77 I = 1, N
      IF ( A(I,J) > A(I,K)+A(K,J) )
      A(I,J) = A(I,K)+A(K,J)
   77 CONTINUE
```
равносилен следующему
```
      DO 77 K = 1, N
      DO 77 J = 1, N
      DO 77 I = 1, N
      T = A(I,K)+A(K,J)
      IF ( A(I,J) > T )
      A(I,J) = T
   77 CONTINUE
```
Конец примера.

Пример 40. Фрагмент программы
```
      IF ( X > 0 ) THEN
      A = C+D+E
      B = 2*(C+D)
```
равносилен следующему
```
      T = C+D
      IF ( X > 0 ) THEN
      A = T+E
      B = 2*T
```
Конец примера.

Пример 41. Фрагмент программы

```
IF ( X > 0 ) THEN
A = C+D+E
B = 2*(C+D)
```

НЕ равносилен следующему

```
IF ( X > 0 ) THEN
T = C+D
A = T+E
B = 2*T
```

Конец примера.

ПЕРЕИМЕНОВАНИЕ БЕЗИНДЕКСНЫХ ПЕРЕМЕННЫХ

Переименование переменной состоит в том, чтобы в какой-либо части программы имя некоторой переменной заменить новым именем. Разумеется, новую переменную, если требует синтаксис языка, предварительно следует описать.

Переименование переменных призвано устранять информационные зависимости в программе. Это преобразование также позволяет приводить фрагмент программы к виду с однократным присваиванием (т.е. каждая переменная получает свое значение только один раз).

Пример 42. Линейный участок программы

```
A = A+C
A = A+B
C = 3*A
A =  A+A+C
write(A)
```

эквивалентен следующему

```
A = A+C
AA = A+B
C = 3*AA
A =  AA+AA+C
write(A)
```

Конец примера.

Пример 43. Фрагмент программы

```
A = B+C
IF K>0
A = A+B
ENDIF
C = 3*A
```

не эквивалентен следующему фрагменту

```
      A = B+C
      IF K>0
      AA = A+B
      ENDIF
      C = 3*AA
```

Последнее вхождение переменной A исходного цикла использует значения различных генераторов в зависимости от переменной K. Для переименования переменных в этом примере необходимо предварительное преобразование условного оператора.

Конец примера.

Пример 44. Линейный участок программы

```
      A = A+C
      A = A+B
777   C = 3*A
      A =  A+A+C
      write(A)
```

вообще говоря, не эквивалентен следующему

```
      A = A+C
      AA = A+B
777   C = 3*AA
      A =  AA+AA
      write(A)
```

поскольку использование переменной A в операторе 777 может использовать значения нескольких генераторов (фрагмент имеет два входа).

Конец примера.

Будем говорить, что фрагмент программы обладаем свойством разложимости относительно оператора Operator, если этот фрагмент программы представим в виде

```
PartOfProg1
Operator
PartOfProg2
```

где каждый из фрагментов PartOfProg1 и PartOfProg2 имеет один вход и один выход.

Переименование переменных призвано устранять информационные зависимости в программе [**53**]. Это преобразование также позволяет приводить фрагмент программы к виду с однократным присваиванием (т.е. каждая переменная получает свое значение только один раз).

Теорема 13. Пусть в тексте фрагмента программы имеется генератор безиндексной переменной A (будем называть его базисным) и, предположим, что всякое использование A в данном фрагменте использует значения базисного генератора и не использует значения других генераторов. Предположим, что за пределами данного фрагмента не используются значения базисного генера-

тора переменной A. Предположим, что переменная AA не используется в данной программе. Тогда в рассматриваемом фрагменте переменную A можно заменить переменной AA (и имя базисного генератора и имена всех использований, использующих значения этого генератора). Результирующая программа будет эквивалентна исходной.

Доказательство вытекает из того, что значения операндов всех операций после преобразования остаются, очевидно, неизменными (и, как следствие, результаты тоже неизменны), а изменились только имена.

Теорема 14. Пусть в тексте программы имеется генератор безиндексной переменной A (будем называть его базисным) и, предположим, что всякое использование A использует значения базисного генератора и не использует значения других генераторов. Тогда имя базисного генератора и имена всех использований, использующих значения этого генератора, могут быть заменены именем новой переменной.

Доказательство вытекает из того, что значения операндов всех операций после преобразования остаются, очевидно, неизменными (и, как следствие, результаты тоже неизменны), а изменились только имена.

Лемма 1. Пусть оператор

L Statement(A(I),A(J),...)

с меткой L включает использования A(I), A(J),... переменной A. Тогда этот оператор можно заменить фрагментом программы

```
L           AA(I) = A(I)
              AA(J)= A(J)
. . . . . . . . . . . . . . . . . . . . . . . . . . .
L1         Statement(AA(I),AA(J),...)
```

При этом если исходный оператор был последним оператором тел каких-либо циклов, то в заголовках этих циклов следует метку L заменить меткой L1. (Метка L переносится к первому добавленному оператору присваивания на тот случай, если существуют переходы (goto) к оператору с этой меткой).

*Доказательство***.** Если в результирующем фрагменте выполнить подстановку вперед, взяв в качестве базисных добавленные операторы присваивания, после чего эти базисные операторы удалить, как неиспользуемые, то получится исходный оператор присваивания.

Лемма 2. Пусть в текст программы входит оператор присваивания

```
L        A(I) = expr
```

с меткой L. Тогда этот оператор можно заменить фрагментом программы, состоящим из двух операторов

```
L           AA(I) = expr
L1         A(I) = AA(I)
```

Если исходный оператор был последним оператором тел каких-либо циклов, то в заголовках этих циклов следует метку L заменить меткой L1.

*Доказательство***.** В результирующем фрагменте выполним подстановку вперед, взяв исходный оператор в качестве базисного. После этого добавлен-

ный оператор (с меткой L1) удалим, как неиспользуемый. Получится исходный оператор присваивания.

Теорема 15. (о переименовании переменной во фрагменте, разложимом относительно первого генератора). Пусть имеется фрагмент программы

PartOfProg(A,B,...)

и пусть в этом фрагменте имеются безиндексные переменные A,B,.... Будем предполагать, что, если для произвольной безиндексной переменной X из указанного списка этот фрагмент программы содержит хотя бы один генератор переменной X, то он представим в виде

```
PartOfProg1
X=expr1
PartOfProg2
```

где PartOfProg1 – не содержит генераторов переменной X и каждый из фрагментов PartOfProg1 и PartOfProg2 имеет один вход и один выход. Тогда эквивалентным преобразованием является замена исходного фрагмента на фрагмент программы

```
AA=A
BB=B
.............................
PartOfProg(AA,BB,...)
A=AA
B=BB
.........................
```

(здесь AA, BB,... - новые переменные).

Доказательство. Будем предполагать, что существует только одна безиндексная переменная A во фрагменте программы PartOfProg(A). Общий случай получается по индукции.

Рассмотрим случай существования генератора переменной A в исходном фрагменте. Представим фрагмент программы PartOfProg(A) в виде конкатенации фрагментов

```
PartOfProg1(A)
A=expr1
PartOfProg2(A)
```

где PartOfProg1(A) представляет собой часть PartOfProg(A) от начала до первого генератора переменной A (не включая его), а PartOfProg2(A) - оставшиеся операторы. (Заметим, что один (или оба) из этих фрагментов может быть пустым – не содержать ни одного оператора). Тогда результирующий фрагмент программы примет вид

```
AA=A
PartOfProg1(AA)
AA=expr
PartOfProg2(AA)
A=AA
```

В части

```
AA=A
```

```
PartOfProg1(AA)
```

выполним подстановку вперед и получим

```
AA=A
PartOfProg1(A)
AA=expr
PartOfProg2(AA)
A=AA
```

Теперь можно удалить первый оператор присваивания этого фрагмента, получив ему эквивалентный

```
PartOfProg1(A)
AA=expr
PartOfProg2(AA)
A=AA
```

Новый фрагмент можно получить из фрагмента

```
PartOfProg1(A)
A=expr
PartOfProg2(A)
A=A
```

переименованием переменных (и, согласно предыдущей теореме, эти фрагменты эквивалентны). В заключение осталось удалить пустой оператор присваивания A=A. Таким образом, описанное в условии теоремы преобразование является последовательностью известных эквивалентных преобразований и, значит, само оно тоже эквивалентно.

Условия переименования безиндексных переменных без труда переносятся на переменные с индексами, если у всех рассматриваемых вхождений индексные выражения одинаковы и если индексы на протяжении всего фрагмента не меняются. В частности, если индексное выражение меняется в цикле, то одновременно у всех рассматриваемых вхождений. Схема переименования переменных с разными индексами представлена в [**44**]. В работе [**55**] разработана техника анализа решетчатых графов и сообщается о возможности ее применения к переименованию индексных переменных.

В следующих двух примерах по-разному выполняется переименование, поскольку в качестве базовых выбраны разные операторы.

Пример 45**.** Цикл

```
      DO 99 I = 5, 100
      X(I) = B(I)*K(I)
      K(I) = X(I-1)+X(I-3)
99    X(I-1) = A(I+1)+X(I-1)
```

преобразуется к эквивалентному фрагменту программы

```
      XX(2)=X(2)
      XX(3)=X(3)
      DO 99 I = 5, 100
      X(I) = B(I)*K(I)
      K(I) = X(I-1)+XX(I-3)
```

```
99   XX(I-1) = A(I+1)+X(I-1)
```

После такой замены далее в тексте программы должны быть заменены многие вхождения переменной X на XX.

Конец примера.

Пример 46. Цикл

```
     DO 99 I = 5, 100
     X(I) = B(I)*K(I)
     K(I) = X(I-1)+X(I-3)
99   X(I-1) = A(I+1)+X(I-1)
```

преобразуется к эквивалентному фрагменту программы

```
     XX(4)=X(4)
     DO 99 I = 5, 100
     XX(I) = B(I)*K(I)
     K(I) = XX(I-1)+X(I-3)
99   X(I-1) = A(I+1)+XX(I-1)
     X(100) = XX(100)
```

Конец примера.

Пример 47. Цикл

```
      DO 99 I = 1, 100
      X(I) = A(I+1)+X(I)
       K(I) = X(I+1)+X(I)
       IF K(I)>0 GOTO 100
       X(I) = B(I)*X(I)
 99    CONTINUE
```

преобразуется к эквивалентному циклу

```
      DO 99 I = 1, 100
      XX(I) = A(I+1)+X(I)
      K(I) = X(I+1)+XX(I)
       IF K(I)>0 GOTO 100
       X(I) = B(I)*XX(I)
 99    CONTINUE
```

Конец примера.

Пример 48. Цикл

```
       DO 99 I = 1, 100
      X(I) = A(I+1)+X(I)
      K(I) = X(I+1)+X(I)
       X(I) = B(I)*K(I)
 99    CONTINUE
```

преобразуется к эквивалентному циклу

```
      DO 99 I = 1, 100
      XX(I) = A(I+1)+X(I)
      K(I) = X(I+1)+XX(I)
       X(I) = B(I)*K(I)
```

```
 99     CONTINUE
```
Конец примера.

Пример 49**.** Цикл

```
       DO 99 I = 1, 100
       X(I) = A(I+1)+X(I-1)
       K(I) = X(I+1)+X(I)
        X(I) = B(I)*K(I)
 99     CONTINUE
```
преобразуется к эквивалентному фрагменту программы

```
DO 99 I = 1, 100
       XX(I) = A(I+1)+X(I-1)
       K(I) = X(I+1)+XX(I)
        X(I) = B(I)*K(I)
 99     CONTINUE
```
Конец примера.

Пример 50**.** Цикл

```
        DO 99 I = 5, 100
       X(I-1) = A(I+1)+X(I-1)
       K(I) = X(I-1)+X(I-3)
        X(I) = B(I)*K(I)
 99     CONTINUE
```
преобразуется к эквивалентному фрагменту программы

```
        XX(2)=X(2)
        XX(3)=X(3)
        DO 99 I = 5, 100
       XX(I-1) = A(I+1)+X(I-1)
       K(I) = XX(I-1)+XX(I-3)
        X(I) = B(I)*K(I)
 99     CONTINUE
```
Конец примера.

Пример 51**. НЕУДОБНЫЙ ПРИМЕР ДЛЯ ПЕРЕИМЕНОВАНИЯ МАССИВОВ.**

```
       DO 77 I = 1, M do
       X(2*I) = 2*X(I)
77     continue
```

Этот цикл вычисляет члены геометрической прогрессии и записывает их в массив X. При этом отношение количества элементов массива X, которые получили новое значение к элементам, которые не получили такого значения очень мало (стремиться к нулю с увеличением количества итераций цикла).

Конец примера.

ПРЕОБРАЗОВАНИЯ ЦИКЛОВ

РАСКРУТКА ЦИКЛА

Раскрутка цикла [**18**] заменяет цикл

```
    DO 999  I=1,N
999  ТЕЛОЦИКЛА(I)
```

на следующий фрагмент программы

```
    ТЕЛОЦИКЛА(1)
    ТЕЛОЦИКЛА(2)
.....
    ТЕЛОЦИКЛА(N)
    I = N
```

Границы цикла должны быть константами (в частности, значение N в данном цикле должно быть известно на этапе трансляции).

Если ТЕЛОЦИКЛА(I) содержит помеченные операторы, на которые указывают goto, то в копиях тела цикла метки должны вводиться новые. В упрощенном варианте раскрутку можно применять лишь для циклов, в теле которых не содержатся помеченные операторы, на которые указывают goto.

Счетчику цикла следует присвоить значение последней итерации.

Результирующий фрагмент программы должен быть в операторных скобках. Это важно для того случая, в котором исходный цикл был телом некоторого другого объемлющего цикла или в зоне действия условного оператора.

Поскольку при раскрутке цикла не меняется порядок выполнения операций, это преобразование всегда эквивалентно. Более того, оператор цикла по своей семантике определяется, как краткая запись раскрутки.

Пример 52. Цикл

```
    DO 99 I = 1, 3
    X(I) = A(I+1)
    K(I) = X(I-1)
99  CONTINUE
```

эквивалентен следующему фрагменту программы

```
    X(1) = A(2)
    K(1) = X(0)
    X(2) = A(3)
    K(2) = X(1)
    X(3) = A(4)
    K(3) = X(2)
```

Конец примера.

УДАЛЕНИЕ ОПЕРАТОРА (ЗАГОЛОВКА) ЦИКЛА

Преобразование состоит в том, чтобы вместо заголовка цикла поставить оператор присваивания, в котором счетчику цикла присваивается значение последней итерации рассматриваемого цикла.

Другой вариант этого же преобразования состоит в удалении заголовка цикла и замене счетчика в теле цикла на его последнее значение.

Если заголовок цикла имел метку, то в обоих вариантах, удаляя заголовок цикла, вместо него надо написать пустой оператор **CONTINUE,** сохранив метку.

Теорема 16. Если тело цикла не содержит ни одного оператора, то удаление оператора цикла является равносильным преобразованием.

Доказательство. Это утверждение является очевидным следствием предыдущего.

Теорема 17. Если цикл имеет лишь одну итерацию, то удаление оператора цикла является равносильным преобразованием при замене счетчика цикла на его единственное значение в теле цикла.

Доказательство. Описанное преобразование является в данном случае раскруткой цикла и по этой причине - эквивалентно.

Теорема 18. Пусть в теле цикла есть только условные операторы и присваивания. Предположим, что все генераторы операторов присваивания тела цикла не зависят от счетчиков цикла или других переменных, меняющих свое значение при выполнении цикла. Пусть граф информационных связей тела цикла не содержит дуг истинной циклически порожденной зависимости. Тогда, удалив заголовок цикла, поставив в теле цикла вместо счетчика его последнее значение, и, поставив после тела цикла оператор, присваивающий счетчику его последнее значение, получим фрагмент программы, эквивалентный исходному.

Доказательство. Сделаем раскрутку цикла. Раскрутка содержит столько копий тела цикла, сколько было итераций. Докажем, что все копии, кроме последней, можно удалить.

Рассмотрим первую копию тела цикла (остальные рассматриваются аналогично). Поскольку граф информационных связей тела исходного цикла не содержит дуг истинной циклически порожденной зависимости, значения генераторов первой копии тела цикла не используются в других копиях. Поскольку генераторы операторов присваивания не зависят от счетчика цикла или других переменных, меняющих свое значения в цикле, эти генераторы на каждой итерации пишут данные в одну и ту же ячейку памяти. Поскольку значения генераторов первой копии тела цикла перезаписываются в дальнейших копиях, эти значения не используются и после раскрутки цикла (далее могут использоваться только значения последней копии тела цикла). Поэтому, все операторы присваивания первой копии тела цикла можно заменить пустыми операторами. Тогда все условные операторы первой копии тела цикла будут содержать только

пустые операторы – и, следовательно, их тоже можно заменить пустыми операторами. В конце концов, первую копию тела цикла можно заменить пустым оператором. Поскольку раскрутка цикла представляет собой конкатенацию копий тела цикла, этот пустой оператор можно удалить.

Аналогично могут быть удалены и другие копии тела цикла, кроме последней. В последней копии тела цикла вместо счетчика цикла стоит его последнее значение. Таким образом, исходный цикл эквивалентно преобразован к виду, описанному в условии теоремы.

Пример 53. Цикл

```
      DO 99 I = 5,100,10
  99 X = A+B
```

эквивалентен оператору

```
  99 X = A+B
```

Конец примера.

Пример 54. Цикл

```
      DO 99 I = 5,100,10
  99 X = A+X
```

не эквивалентен оператору

```
  99 X = A+X
```

Конец примера.

Пример 55. Цикл

```
      DO 99 I = 5,100,10
  99 X = A(I)+B
```

не эквивалентен оператору

```
  99 X = A(100)+B
```

(Последняя итерация исходного цикла происходит при I=95 и, поэтому исходный цикл должен быть заменен на оператор 99 X = A(95)+B).

Конец примера.

Пример 56. Цикл

```
      DO 99 I = 5,100,10
  99 X = A(I)+B*I
```

эквивалентен оператору

```
  99 X = A(95)+95*B
```

Конец примера.

Пример 57. Цикл

```
      DO 99 I = 5,100,10
  99 X = X+1
```

не эквивалентен оператору

```
  99 X = X+1
```

Конец примера.

КАНОНИЗАЦИЯ ЦИКЛОВ

Данное преобразование заменяет всякий цикл равносильным, с левой границей 1 и шагом тоже 1. Иногда используется термин «нормализация» (do-loop normalization) [**8**, с. 37]. Это преобразование применимо к любому оператору цикла

```
      DO 99 I=L, N, h
99    ТЕЛОЦИКЛА(I)
```

и заменяет его оператором цикла

```
      DO 99 J=1, (N-L+1)//h
99    ТЕЛОЦИКЛА(h*(J-1)+L)
```

Здесь // означает операцию целочисленного деления. К описанным требованиям канонического вида цикла можно добавить еще одно: последний оператор тела цикла должен быть CONTINUE.

Алгоритм выполнения канонизации состоит в следующем:

1) сгенерировать новую переменную (обозначим ее J) для имени нового счетчика цикла.
2) Всюду в теле цикла заменить старый счетчик цикла I на выражение (h*(J-1)+L). Для выполнения этого преобразования можно использовать подстановку вперед.
3) В заголовке цикла левая граница заменяется числом 1, а правая - (N-L+1)//h.
4) Если последний оператор тела исходного цикла не CONTINUE, то надо этот оператор (CONTINUE) приписать в конце цикла со сгенерированной новой меткой и эту же метку указать в заголовке цикла.

Замечание. Пункт 2 приведенного алгоритма может заметно увеличить время выполнения цикла, поскольку заменяет переменную I выражением с тремя операциями. Поэтому, вместо пункта 2 иногда при канонизации удобнее использовать пункт 2’

2’) В теле цикла первым оператором поставить присваивание
I = (h*(J-1)+L).

У такой формы канонизации общий объем вычислений увеличится не сильно, но появиться дополнительный оператор и в индексных выражениях вместо нового счетчика цикла J будет стоять индуктивная переменная I.

Пример 58. Для цикла

```
DO 99 I = K-1, N, 2
if (A(I)<X) goto 99
K = B+X
```

```
  99 X = I+X
```
канонический вид будет следующий
```
    DO 88 J = 1, (N-K)//2
    if (A(2*(J-1)+K-1)<X) goto 99
    K = B+X
  99 X = 2*(J-1)+K-1+X
  88 CONTINUE
```
Конец примера.

Канонический вид цикла допускает вариации, в зависимости от дальнейших преобразований. Следует отметить, что для некоторых преобразований удобно иметь цикл, у которого начальное значение счетчика (левая граница) равно 0 (например, [**23**]). Иногда перед циклом и после цикла добавляются пустые операторы, которые перехватывают передачи управления, ведущие на заголовок цикла и выходящие из цикла соответственно [**54**].

РАСЩЕПЛЕНИЕ ОДНОМЕРНОГО ЦИКЛА

Это преобразование заменяет цикл

```
    DO 99  I=1,N
99 ТЕЛОЦИКЛА(I)
```

на фрагмент программы

```
        DO 99 I=1,M
99      ТЕЛОЦИКЛА(I)
         DO 88 I=1,N-M
88       ТЕЛОЦИКЛА(I+M)
```

Это же преобразование может выглядеть и по-другому, как замена цикла

```
         DO 99 I=1,N
   99    ТЕЛОЦИКЛА(I)
```

на фрагмент программы

```
        DO 99 I=1,M
99      ТЕЛОЦИКЛА(I)
        DO 88 I=M+1,N
88      ТЕЛОЦИКЛА(I)
```

(Здесь не выполнена канонизация второго цикла в результирующем фрагменте).

Результирующий фрагмент программы должен быть в операторных скобках. Это важно для того случая, если бы исходный цикл был телом некоторого другого объемлющего цикла или под действием условного оператора.

Расщепление цикла используется как вспомогательное преобразование. Например, если цикл состоит из 100 итераций и надо сделать ему 3-кратную развертку, то этот цикл следует разделить на два: у одного 99 итераций, а у другого - 1.

Пример 59. Цикл

```
      DO 99 I=1,100
99    X(I)=Y(I-1)
```

эквивалентен фрагменту программы

```
      DO 99 I=1, 99, 3
      X(I)=Y(I-1)
      X(I+1)=Y(I)
99    X(I+2)=Y(I+1)
      DO 88 I=100,100
88    X(I)=Y(I-1)
```

Конец примера.

Теорема 19. Предположим, что в одномерном цикле

```
    DO 99  I=1,N
99 ТЕЛОЦИКЛА(I)
```

есть зависимые индексные переменные, у которых в индексных выражениях коэффициенты при счетчиках цикла имеют разные знаки: X(a*I+N1) и X(N+N2-b*I). Тогда в результате расщепления пространства итераций цикла в точке M = (N+N2-N1)/(a+b) получатся два цикла, в каждом из которых исходные вхождения не связаны зависимостью (информационная зависимость стала между циклами).

Расщепление описывается во многих работах [**17**], [**10**], [**44**].

При выполнении расщепления следует контролировать соблюдение корректности. При копировании тела цикла не должно появиться двух операторов с одинаковыми метками. Оба результирующих цикла следует взять в операторные скобки и, если у заголовка исходного цикла была метка, то эта метка должна быть перенесена к открывающей скобке. Расщепление может выполняться, если исходный цикл содержит более одной итерации, либо, если допускаются циклы с нулевым количеством операций.

При расщеплении одномерного цикла не меняется порядок выполнения операций, поэтому, если у цикла единственный выход по завершении всех итераций, указанных в заголовке, расщепление является эквивалентным преобразованием.

РАЗРЫВ ИТЕРАЦИЙ, ГНЕЗДОВАНИЕ ЦИКЛА И ПОДОБНЫЕ ИМ ПРЕОБРАЗОВАНИЯ

Разрыв итераций (strip mining) - это преобразование [**8**, с. 37], которое заменяет цикл

```
      DO 99 I=1,N
99    ТЕЛОЦИКЛА(I)
```

на следующий фрагмент программы

```
      DO 99 I=1,N//h
      DO 99 J=1,h
99    ТЕЛОЦИКЛА(h*(I-1)+J)
      DO 88 I=1,N-(N//h)*h
88    ТЕЛОЦИКЛА((N//h)*h+I)
```

Если N делится на h, то разрыв итераций выглядит более просто. Это преобразование заменяет цикл

```
      DO 99 I=1,N
99    ТЕЛОЦИКЛА(I)
```

на следующий цикл

```
      DO 99 I=1,N,h
      DO 99 J=1,h
99    ТЕЛОЦИКЛА(I+J-1)
```

Гнездование цикла похоже на разрыв итераций и заменяет цикл

```
      DO 99 I=1,N
99    ТЕЛОЦИКЛА(I)
```

на следующий фрагмент программы

```
      DO 99 J=1,h
      DO 99 I=1,N//h
99    ТЕЛОЦИКЛА(I+(J-1)*(N//h))
      DO 88 I=1,N-(N//h)*h
88    ТЕЛОЦИКЛА((N//h)*h+I)
```

Если N константа и не делится нацело на h, то ко второму циклу полученного фрагмента программы можно применить раскрутку цикла.

Метка последнего оператора тела цикла должна остаться только у оператора последней копии тела цикла. Если ТЕЛОЦИКЛА(I) содержит помеченные операторы, то в копиях тела цикла метки должны вводиться новые (вместе со ссылками на эти метки). Ввиду этого требования удобно, чтобы последним оператором тела цикла был CONTINUE (в языке FORTRAN) и, чтобы не было меток у операторов без необходимости.

Результирующий фрагмент программы должен быть в операторных скобках. Это важно для того случая, в котором исходный цикл был телом некоторого другого объемлющего цикла или под действием условного оператора.

Не должно быть операторов выхода из цикла EXIT, BREAK.

Теорема 20. Разрыв итераций и гнездование цикла являются равносильными преобразованиями.

Доказательство вытекает из того, что эти преобразования не меняют порядка выполнения операций.

Отличие этих преобразований в том, что у первого преобразования заранее задан размер внутреннего цикла, а размер внешнего зависит еще и от длины исходного, а во втором преобразовании - наоборот. Первое преобразование может быть целесообразно для компьютеров, параллельно обрабатывающих векторы фиксированной длины h. Второе преобразование может быть целесообразно при выполнении программы на компьютере типа MIMD с количеством процессоров h.

К преобразованиям «разрыв итераций» и «гнездование цикла» можно присовокупить еще два, на них похожих. Первое из них меняет цикл

```
DO 99 I=1,N
99    ТЕЛОЦИКЛА(I)
```

на следующий фрагмент программы

```
      DO 99 I=1,h
      DO 99 J=1,N//h
99    ТЕЛОЦИКЛА(h*(J-1)+I)
      DO 88 I=1,N-(N//h)*h
88    ТЕЛОЦИКЛА((N//h)*h+I)
```

а второе - на следующий

```
DO 99 J=1,N//h
DO 99 I=1,h
99    ТЕЛОЦИКЛА(J+(I-1)*(N//h))
      DO 88 I=1,N-(N//h)*h
88    ТЕЛОЦИКЛА((N//h)*h+I)
```

Эти преобразования равносильны не всегда. Если в теле цикла есть два вхождения некоторой переменной, которые обращаются к одной и той же ячейке памяти на итерациях I1 и I2, и выполняются условия I1<I2 и (I2 mod h) < (I1 mod h) то первое преобразование не равносильно. (Для второго цикла в этом рассуждении следует h заменить выражением N//h).

Пример 60. Замена цикла

```
      DO 99 I=1,128
99    X(I)= 3*X(I-1)
```

на следующий фрагмент программы равносильна.

```
      DO 99 I=1,8
      DO 99 J=1,16
99    X(16*(I-1)+J)=3*X(16*(I-1)+J-1)
```

Конец примера.

Пример 61. Замена цикла

```
      DO 99 I=1,128
99    X(I)= 3*X(I-1)
```

на следующий фрагмент программы

```
      DO 99 I=1,8
      DO 99 J=1,16
99    X(8*(J-1)+I)=3*X(8*(J-1)+I-1)
```

не равносильна. Действительно, если до выполнения цикла все элементы массива X равны 1, то после выполнения исходного цикла X(9)=3**8, а после выполнения результирующего - X(9)=3.
Конец примера.

Пример 62. Замена цикла

```
      DO 99 I=1, 129
99    X(I)= 3*X(I-8)
```

на следующий фрагмент программы равносильна

```
      DO 99 I=1, 8
      DO 99 J=1, 16
99    X(8*(J-1)+I)=3*X(8*(J-1)+I-8)
      X(129)= 3*X(129-8)
```

По поводу всех четырех преобразований следует отметить, что, поскольку в них копируется тело цикла, надо следить за тем, чтобы не возникло двух операторов с одинаковыми метками.
Конец примера.

Теорема 21. Предположим, что в теле цикла для любой переменной и любых двух ее вхождений, которые обращаются к одной и той же ячейке памяти на итерациях I1 и I2 соответственно, не выполняются одновременно условия:

- хотя бы одно из этих вхождений является генератором,
- I1<I2
- (I2 mod h) < (I1 mod h).

Тогда первое преобразование равносильно.

Доказательство. Если к исходному и результирующему фрагментам программ применить раскрутку циклов, то полученные линейные участки программ будут отличаться перестановкой операторов. Теперь остается только вспомнить условие эквивалентности перестановки операторов.

Рассмотренное преобразование может быть применено к распараллеливанию на MIMD архитектуру. Это преобразование иногда позволяет некоторые рекуррентные зависимости сосредоточить внутри отдельных процессоров и этим самым уменьшить количество межпроцессорных пересылок.

Условие равносильности второго преобразования формулируется аналогично.

РАЗВЕРТКА ЦИКЛА

Развертка цикла [**18**] кратности h заменяет цикл

```
    DO 999  I = 1, N
999  ТЕЛОЦИКЛА(I)
```

на следующий фрагмент программы

```
    DO 888 I = 1, N, h
    ТЕЛОЦИКЛА(I)
    ТЕЛОЦИКЛА(I+1)
.....
888  ТЕЛОЦИКЛА(I+h-1)
    N2 = N mod h    //остаток от деления числа N на h.
    DO 777  I = N-N2+1, N
777  ТЕЛОЦИКЛА(I)
```

Если N – константа, которая делится нацело на h, то второй цикл и стоящий перед ним оператор присваивания можно опустить.

Если N константа и не делится нацело на h, то ко второму циклу полученного фрагмента программы можно применить раскрутку цикла.

Метка последнего оператора тела цикла должна остаться только у оператора последней копии тела цикла. Если ТЕЛОЦИКЛА(I) содержит помеченные операторы, то в копиях тела цикла метки должны вводиться новые (вместе

со ссылками на эти метки). Ввиду этого требования удобно, чтобы последним оператором тела цикла был CONTINUE и, чтобы не было без необходимости помеченных операторов.

Результирующий фрагмент программы должен быть в операторных скобках. Это важно для того случая, если бы исходный цикл был телом некоторого другого объемлющего цикла или под действием условного оператора.

Не должно быть операторов выхода из цикла EXIT, BREAK.

Теорема 22. Пусть тело цикла содержит операторы присваивания, логические или блочные условные операторы, операторы безусловного перехода GOTO (но не вычислимый GOTO языка ФОРТРАН!) или другие операторы цикла. Тогда развертка цикла, выполненная по описанным выше правилам, является эквивалентным преобразованием.

Доказательство вытекает из того, что при развертке цикла не меняется порядок выполнения операций. Следует проконтролировать лишь соблюдение корректности. Для более строго доказательства следует сделать раскрутки исходного и результирующего циклов.

Пример 63. Цикл

```
      DO 99 I = 1, 100
99    X(I) = A(I+1)+K(I)-X(I-1)
```

эквивалентен следующему

```
      DO 99 J = 1, 50
      I = 2*J
      X(I-1) = A(I)+K(I-1)-X(I-2)
99    X(I) = A(I+1)+K(I)-X(I-1)
```

Конец примера.

Пример 64. Цикл

```
      DO 99 I = 1, 100
      DO 88 J = 1, 40
88    B(J) = K(I)*X(J)
99    X(I) = A(I+1)+B(I)-X(I-1)
```

эквивалентен следующему

```
      DO 99 I = 1, 100, 2
      DO 77 J = 1, 40
77    B(J) = K(I)*X(J)
      X(I) = A(I+1)+B(I)-X(I-1)
      DO 88 J = 1, 40
88    B(J) = K(I+1)*X(J)
99    X(I+1) = A(I+2)+B(I+1)-X(I)
```

(Следует обратить внимание на изменение метки в заголовке и последнем операторе первого вложенного цикла)

Конец примера.

РАСЩЕПЛЕНИЕ ВЕРШИН (ВВЕДЕНИЕ ВРЕМЕННЫХ МАССИВОВ)

Это преобразование позволяет перенаправлять в программных циклах дуги антизависимости (дуги, направленные снизу вверх становятся направленными сверху вниз). Суть преобразования в том, чтобы какое-нибудь вхождение присвоить новой переменной и заменить этой переменной.

Пример 65. В следующем цикле есть дуга антизависимости от вхождения A(I+1) к первому вхождению A(I). Отметим, что эту дугу нельзя устранить перестановкой операторов тела цикла местами (такая перестановка нарушит равносильность программы ввиду наличия дуги потоковой зависимости между двумя вхождениями A(I)).

```
      DO 111 I = 1, N
      A(I) = B(I)+C(I)
      D(I) = (A(I)+A(I+1))/2.
111   CONTINUE
```

Данный цикл равносилен следующему

```
      DO 111 I = 1, N
      ATEMP(I) = A(I+1)
      A(I) = B(I)+C(I)
      D(I) = (A(I)+ATEMP(I))/2.
111   CONTINUE
```

После выполнения данного преобразования можно цикл разбить на части (чему мешала прежде дуга антизависимости, ведущая «снизу-вверх»).

Конец примера.

Алгоритм. Находим в графе информационных связей дугу антизависимости. Если начальная вершина этой дуги не является источником в подграфе информационных связей, соответствующем данному фрагменту программы (телу цикла), то данная дуга антизависимости неустранима.

Иначе,

1) вводим временный одномерный массив, длина которого равна длине преобразуемого цикла (вставляем в программу оператор описания массива real ATEMP(N))

2) в начале тела цикла вставляем оператор присваивания, левой частью которого является вхождение нового массива, у которого индексное выражение состоит только из счетчика цикла, а правой частью этого оператора присваивания является вхождение, соответствующее начальной вершине рассматриваемой дуги антизависимости

3) вхождение, соответствующее начальной вершине рассматриваемой дуги антизависимости, заменяем на левую часть нового оператора.

Пример 66.

```
      DO 111 I = 3, N
      A(I) = B(I-2)+C(I)
      B(I) = (A(I)+A(I+1))/2
111   CONTINUE
```

Данный цикл равносилен следующему

```
      DO 111 I = 3, N
      ATEMP(I) = A(I+1)
      BTEMP(I) = B(I)
      A(I) = B(I-2)+C(I)
      BTEMP(I) = (A(I)+ATEMP(I))/2
111   CONTINUE
```

Конец примера.

РАСТЯГИВАНИЕ СКАЛЯРОВ (ПОВЫШЕНИЕ РАЗМЕРНОСТИ МАССИВА)

Это преобразование состоит в замене скалярной переменной массивом, зависящим от счетчика цикла, в который переменная входит. Растягивание скаляров иногда позволяет избавиться от некоторых информационных зависимостей в цикле. [**8**, с. 32].

Пример 67. Данный цикл

```
        DO 111  I  =  1, 100
        A  =  B(I)+C(I)
        D(I)  =  A/2
    111 CONTINUE
```

эквивалентен следующему

```
        dimention AA[100]
        DO 111  I  =  1,  100
        AA(I)  =  B(I)+C(I)
        D(I)  =   AA(I)/2
    111 CONTINUE
```

Два вхождения переменной A в исходном цикле связаны двумя встречными дугами информационной зависимости. После замены переменной на массив AA дуга антизависимости исчезла и осталась только дуга истинной информационной зависимости.

Конец примера.

Алгоритм выполнения замены переменной на массив выглядит следующим образом.

Находим в теле цикла генератор, не зависящий от счетчика цикла. Будем считать для определенности, что имя этого генератора A, а имя счетчика цикла – I. Подбираем не использованное в данной программе имя для нового массива - предположим, что таким именем является AA. Заметим, что в теле цикла может быть несколько генераторов A. Предположим, что тело цикла состоит только из операторов присваивания. Тогда заменим все вхождения до первого генератора A на AA(I), а все вхождения A , начиная с первого генератора, – на AA(I+1). В начало программы следует вставить оператор описания массива AA. Перед заголовком цикла следует поставить оператор присваивания

AA(1)=A.

Если у заголовка цикла была метка, то ее надо передать вставленному перед заголовком оператору присваивания.

Если в теле цикла нет вхождения переменной A, как генератора, то после цикла следует вставить оператор

A=AA(I)

(в фортране после выполнения цикла значение счетчика сохраняется) или оператор

A=AA(N)

где N - последнее значение счетчика цикла.

В противоположном случае (если в теле цикла есть генератор переменной A) после цикла должен быть вставлен оператор

A=AA(I+1)

или

A=AA(N+1)

соответственно.

Теорема 23. Пусть в теле цикла есть только операторы присваивания. Тогда описанная выше процедура замены переменной массивом равносильна.

Доказательство. Заменим исходный и результирующий циклы их раскрутками (напомним, что раскрутка - всегда эквивалентное преобразование). Теперь следует доказать эквивалентность полученных линейных участков. Но эти линейные участки могут быть получены один из другого переименованием.

РАЗБИЕНИЕ ЦИКЛА

Наибольшее время работы программы отнимают циклически повторяющиеся участки. Поэтому в оптимизирующих и распараллеливающих компиляторах наиболее распространены преобразования циклов. Разбиение цикла - одно из наиболее важных преобразований, которое может быть использовано при распараллеливании программ на практически все типы параллельных компьютеров: векторные, конвейерные, MIMD, SIMD и т.д.

Суть разбиения цикла состоит в том, чтобы заменить цикл, в теле которого много операторов присваивания, на эквивалентный фрагмент программы из нескольких циклов, в телах которых меньше операторов. При распараллеливании зачастую большой цикл не может быть эффективно отображен на архи-

тектуру суперкомпьютера (например, из-за нехватки ресурсов). В этом случае есть надежда, что после разбиения хотя бы некоторые из результирующих циклов смогут быть параллельно вычислены.

Иногда для оптимизации работы с памятью рассматривается разбиение цикла по именам (fission by name) [**8**, с. 36] – здесь одна и та же переменная не может встречаться в различных результирующих циклах. Понятно, что частота применения этого варианта преобразования весьма ограничена.

Не всякий цикл можно разбить непосредственно. Иногда для этого следует применить вспомогательные преобразования. Такие преобразования тоже включены в данную работу. Добавлены также преобразования, в которых разбиение является составной частью. С помощью вспомогательных преобразований разбиение цикла позволяет получить и частичную векторизацию цикла [**8**, с. 66-77].

Разбиение кратных циклов очевидным образом сводится к разбиению одномерных. Иногда вместо термина «разбиение» используется термин «разрезание» или «распределение» [**10**].

Разбиение цикла заменяет цикл

```
    DO 333 I = 1, N
    S1
    .........
    Sk
    S(k+1)
    .........
    Sm
333  CONTINUE
```

на фрагмент программы, состоящий из последовательности двух циклов

```
    DO 111 I = 1, N
    S1
    .........
    Sk
111  CONTINUE
    DO 222 I = 1, N
    S(k+1)
    .........
    Sm
222  CONTINUE
```

Условия применения:

1) каждый из фрагментов программы S1,...,Sk и S(k+1),...,Sm имеет один вход и один выход. В частности:

1.1. ни один из операторов S(k+1),...,Sm не находится в зоне действия какого либо условного оператора из множества S1,...,Sk. (т.е. выполнение или невы-

полнение этого оператора не зависит от значения логического выражения условного оператора)
1.2. ни один из операторов S(k+1),...,Sm не находится в теле цикла, заголовок которого во множестве S1,...,Sk
1.3. всякий оператор goto из множества S1,...,Sk (или S(k+1), ..., Sm) указывает на метку оператора из этого же множества.
2) не существует такой дуги графа информационных связей (v1,v2), что v1 принадлежит Si, k+1=<i=<m, v2 принадлежит Sj, 1=<j=<k.

Результирующий фрагмент программы должен быть в операторных скобках. Это важно для того случая, если бы исходный цикл был телом некоторого другого объемлющего цикла или под действием условного оператора.

Теорема 24. Если условия применения выполнены, то разбиение циклов является эквивалентным преобразованием.

Доказательство. Обозначим для краткости LOOPBODY1 фрагмент программы из операторов S1,...,Sk и LOOPBODY2 - S(k+1), ..., Sm. Тогда разбиение цикла состоит в замене цикла

```
      DO 99 I
LOOPBODY1(I)
LOOPBODY2(I)
99    Continue
```

на фрагмент программы

```
      DO 77 I
LOOPBODY1(I)
77    Continue
      DO 99 I
LOOPBODY2(I)
99    Continue
```

Пусть счетчик цикла I пробегает последовательность итераций $I_1,I_2,...,I_n$. Раскрутка исходного цикла имеет следующий вид

$$\begin{array}{l} LOOPBODY1(I_1) \\ LOOPBODY2(I_1) \\ LOOPBODY1(I_2) \\ LOOPBODY2(I_2) \\ ..... \\ LOOPBODY1(I_n) \\ LOOPBODY2(I_n) \end{array} \qquad (9)$$

Результирующий фрагмент после раскрутки циклов примет следующий вид

$$\begin{array}{l} LOOPBODY1(I_1) \\ LOOPBODY1(I_2) \\ ..... \\ LOOPBODY1(I_n) \\ LOOPBODY2(I_1) \end{array} \qquad (10)$$

LOOPBODY2(I_2)
.....
LOOPBODY2(I_n)

Поскольку не существует дуг информационной зависимости из LOOPBODY2(I_k) в LOOPBODY1(I_m) при k>=m, с помощью эквивалентных перестановок операторов можно из (9) получить (10) (перестановки операторов описаны в [**8**, с.12]).

Пример 68**.** Цикл

```
      DO 99 I = 1, 100
      X(I) = A(I+1)
      K(I) = X(I+1)
99    CONTINUE
```

НЕ эквивалентен следующему фрагменту программы

```
      DO 99 I = 1, 100
      X(I) = A(I+1)
99    CONTINUE
      DO 88 I = 1, 100
       K(I) = X(I+1)
88    CONTINUE
```

Конец примера.

СЛИЯНИЕ ЦИКЛОВ

Данное преобразование заменяет фрагмент программы, состоящий из двух подряд написанных циклов с одинаковыми заголовками, одним циклом (loop fusion) [**8**, с. 36].

Это преобразование является обратным к разбиению циклов.

Условие выполнения: если после выполнения преобразования получается цикл, который можно, сохраняя равносильность, разрезать на две части, получив исходный фрагмент, то данное преобразование равносильно.

Кроме того, есть ограничение, связанное с синтаксической корректностью. В исходной программе не должно быть переходов на заголовок второго цикла, поскольку в этом случае в результирующей программе этот переход исчезнет или возникнет недопустимый синтаксисом переход извне во внутрь цикла. И еще, при взятии в операторные скобки обоих исходных циклов должна получаться программа эквивалентная исходной. Это условие может не выполняться, если первый из исходных циклов является телом некоторого третьего объемлющего цикла, а второй исходный цикл – не является.

Пример 69**.** Фрагмент программы

```
    DO 99 I = 1, 100
    IF Y(I)<0 goto 99
```

```
      K(I) = X(I-1)
99  CONTINUE
      DO 88 I = 1, 100
        X(I) = A(I+1)
   88 CONTINUE
```
эквивалентен следующему
```
DO 88 I = 1, 100
      IF Y(I)<0 goto 99
      K(I) = X(I-1)
99  CONTINUE
        X(I) = A(I+1)
   88 CONTINUE
```
Конец примера.

ПРИВЕДЕНИЕ ЦИКЛА К РАЗБИВАЕМОМУ ВИДУ

Иногда, условия разбиения цикла не выполняются, но, после проведения некоторых эквивалентных преобразований, этого можно добиться. Препятствиями для разбиения цикла могут быть либо управляющие связи в теле цикла либо информационные зависимости. Поэтому предварительные преобразования, ориентированные на дальнейшее разбиение цикла, состоят в устранении управляющих или информационных связей.

Если в гнезде вложенных циклов нам требуется разбить внешний цикл, то, иногда, для этого бывает необходимо предварительно разбить вложенный цикл (вложенные циклы).

Аналогичная ситуация возможна и с условным оператором в теле цикла.

Использование перестановки операторов для разбиения цикла

Иногда бывает так, что в исходном виде цикл не разбивается, но если в теле этого цикла переставить местами некоторые операторы, то разбиение станет возможно.

Пример 70**.**
```
      DO 111  I  =  1,  N
      A(I)  =  B(I)+C(I)
      D(I)  =    A(I+1)+B(I)
 111  CONTINUE
```
В этом цикле вхождение A(I+1) обращается к ячейке памяти A(2) раньше, чем вхождение A(I) и, поэтому дуга информационной зависимости направлена «снизу-вверх» от вхождения A(I+1) к A(I). После перестановки операторов будет наоборот, и цикл можно будет разрезать.

Конец примера.

Пример 71. Цикл

```
      DO 99 I = 1, 100
      X(I) = A(I+1)
      K(I) = X(I+1)
 99   CONTINUE
```

эквивалентен следующему фрагменту программы

```
      DO 99 I = 1, 100
      K(I) = X(I+1)
99    CONTINUE
      DO 88 I = 1, 100
      X(I) = A(I+1)
88    CONTINUE
```

Конец примера.

Для разбиения цикла с перестановкой операторов следует выполнить следующее. В графе информационных связей объявить эквивалентными вершины, принадлежащие одним и тем же компонентам сильной связности и вершины, соответствующие вхождениям в одни и те же операторы (комбинация двух отношений эквивалентности). По данной эквивалентности надо построить факторграф (граф, вершинами которого являются классы эквивалентности, а дуга соединяет две вершины тогда и только тогда, когда в исходном графе была дуга, соединяющая вершины, принадлежащие этим классам эквивалентности). Факторграф получится бесконтурным графом (поскольку компоненты сильной связности исходного графа склеены). Вершины факторграфа следует занумеровать правильным образом (так, чтобы не было дуг из вершин с большими номерами в вершины с меньшими номерами) [**30**]. Теперь в исходном цикле следует переставить местами операторы по порядку возрастания номеров вершин факторграфа, которым эти операторы соответствуют. Если нескольким операторам соответствует одна вершина факторграфа, то эти операторы должны сохранить порядок следования друг относительно друга. Полученный цикл можно разбивать на части, но так, чтобы операторы, которым соответствует одна и та же компонента сильной связности, оказывались в одном и том же результирующем цикле.

Рассмотренный алгоритм разбиения цикла почти совпадает с алгоритмом генерации векторного кода [**8**, с.108]

Пример 72. Цикл

```
      DO 99 I = 1, 100
        K(I) = X(I-1)
        A(I) =  B(I)
        X(I) = A(I+1)+K(I)
        Y(I) = K(I+1)
 99   CONTINUE
```

эквивалентен следующему фрагменту программы

```
       DO 66 I = 1, 100
66     Y(I) = K(I+1)
```

```
      DO 77 I = 1, 100
      K(I) = X(I-1)
77    X(I) = A(I+1)+K(I)
      DO 88 I = 1, 100
88    A(I) =  B(I)
```

Конец примера.

Описанный выше алгоритм разбиения цикла с перестановками операторов приводит к программе эквивалентной исходной.

Операторы, принадлежащие одним и тем же классам эквивалентности, не могут быть разнесены в разные циклы при разбиении цикла. Приведенный алгоритм не улучшаем без изменения операторов тела цикла. Преобразование «расщепление вершин», которое состоит во введении временных переменных (массивов), может некоторые дуги информационной зависимости от одного оператора отнести к разным («расщепить»), после чего не разбиваемый цикл может стать разбиваемым.

Пример 73**.** Фрагмент программы

```
    DO 88 I = 1, 100
     X(I) = A(I+1)+A(I-1)
     A(I)=X(I+1)*5
 88 CONTINUE
```

эквивалентен следующему

```
    DO 88 I = 1, 100
     TEMP(I) = A(I+1)
     X(I) = TEMP(I)+A(I-1)
     A(I)=X(I+1)*5
 88 CONTINUE
```

к которому можно применить перестановку операторов и получить разбиваемый цикл

```
     DO 88 I = 1, 100
     TEMP(I) = A(I+1)
     A(I)=X(I+1)*5
     X(I) = TEMP(I)+A(I-1)
 88 CONTINUE
```

Конец примера.

Использование расщепления вершин для разбиения цикла

Иногда для перестановки операторов может быть полезно предварительное введение временной переменной (расщепление вершин).

Пример 74.

```
     DO 111  I  =  1,  N
    A(I)  =  B(I)+ sin C(I)
```

```
      C(I)  =   A(I+1)+B(I)
  111 CONTINUE
```

В этом примере непосредственная перестановка операторов неэквивалентна. Для осуществления перестановки можно ввести временную переменную TEMP(I).

```
       DO 111  I  =  1,  N
       TEMP(I)=C(I)
      A(I)  =  B(I)+ sin(TEMP(I))
      C(I)  =  A(I+1)+B(I)
  111 CONTINUE
```

Теперь можно выполнить перестановку операторов и разбить цикл, хотя за это пришлось заплатить появлением еще одного оператора присваивания.

```
       DO 111  I  =  1,  N
       TEMP(I)=C(I)
111    C(I)  =    A(I+1)+B(I)
       DO 222  I  =  1,  N
      A(I)  =  B(I)+ sin(TEMP(I))
 222  CONTINUE
```

Конец примера.

Преобразование «расщепление вершин» [**8**, с.121], которое состоит во введении временных переменных (массивов) и дополнительных присваиваний, может некоторые дуги информационной зависимости от одного оператора отнести к разным («расщепить»), после чего цикл может стать разбиваемым.

Пример 75**.** Фрагмент программы

```
    DO 88 I = 1, 100
     X(I) = A(I+1)+A(I-1)
     A(I)=X(I+1)*5
  88 CONTINUE
```

эквивалентен следующему

```
    DO 88 I = 1, 100
     TEMP(I) = A(I+1)
     X(I) = TEMP(I)+A(I-1)
     A(I)=X(I+1)*5
  88 CONTINUE
```

к которому можно применить перестановку операторов и получить разбиваемый цикл

```
      DO 88 I = 1, 100
    TEMP(I) = A(I+1)
    A(I)=X(I+1)*5
    X(I) = TEMP(I)+A(I-1)
  88 CONTINUE
```

Конец примера.

Компоненту сильной связности графа информационных связей будем называть большой, если эта компонента содержит более одной вершины. Очевидно, что всякая большая компонента содержит хотя бы один генератор.

Теорема **25**. Если в цикле есть оператор, содержащий генератор G’ из одной большой компоненты M’ и использование U’ из другой большой компоненты M”

```
      DO 99 I = 1, N
99      G’ = …U’… ,
```

то введение временной переменной T(I) разводит вхождения разных больших компонент к разным операторам.

```
      DO 99 I = 1, N
      T(I) = U’
99   G’ = …T(I)…
```

Доказательство вытекает из того, что новые вхождения T(I) не принадлежат большим компонентам.

Следствие. Если граф информационных связей тела цикла содержит несколько больших компонент сильной связности, то с помощью введения временных переменных цикл можно разбить на несколько так, что у каждого результирующего цикла граф информационных связей не будет содержать более одной такой компоненты.

Теорема 26. Предположим, что в теле цикла есть операция, все аргументы которой являются вхождениями переменных, не принадлежащими большим компонентам сильной связности. Тогда вычисление этой операции можно вынести из цикла, обозначив эту операцию временной переменной.

Доказательство очевидно.

Пример 76**.** Цикл

```
      DO 99 I = 1, N
99    X(I)  =  C(I)*exp(D(I)*X(I-1)+E(I))  + A(I)*B(I)
```

можно заменить фрагментом программы

```
      DO 88 I = 1, N
88    T(I)  =  A(I)*B(I)
      DO 99 I = 1, N
99    X(I)  =  C(I)*exp(D(I)*X(I-1)+E(I)) + T(I)
```

Конец примера.

Повышение размерности массивов для разбиения циклов

Если в графе информационных связей есть контур, то ни перестановка операторов, ни введение временных переменных, как в предыдущем примере, не помогают, и какая-нибудь дуга будет вести «снизу-вверх». Некоторые контуры, образованные вхождениями переменной, не зависящей от счетчика цикла, можно разорвать заменой этой переменной новым массивом, зависящим от

счетчика цикла. Следует отметить, что такое преобразование («расщепление скаляров») предполагает дополнительный расход памяти.

Пример 77. В следующем гнезде циклов

```
      DO 111  J  =  1,  N
      DO 111  I  =  1,  N
     A(I,J)  =  B(I)+ C(J)
     C(J)  =    A(I+1,J)+B(I)
111  CONTINUE
```

после повышения размерности массива C внутренний цикл может быть разрезан с перестановкой операторов

```
      DO 111  J  =  1,  N
        CC(1,J)=C(J)
      DO 222  I  =  1,  N
        CC(I+1,J)  =    A(I+1,J)+B(I)
222     CONTINUE
      DO 333  I  =  1,  N
        A(I,J)  =  B(I)+  CC(I,J)
333     CONTINUE
        C(J) = CC(N+1,J)
111     CONTINUE
```

Конец примера.

Переименование переменных

Если в теле цикла есть два генератора одной и той же переменной, то они могут создавать в графе ложные дуги зависимости и дуги выходной зависимости [**8**, с. 29]. Чтобы от этого избавиться иногда возможно часть вхождений переменной, включая один из генераторов, переименовать. Этим самым тело цикла может быть приведено к виду с однократным присваиванием, что важно для конвейеризации. Условия переименования приводятся в [**44**] .

Пример 78 **([8**, с. 73**]).** В графе информационных связей программного цикла

```
      DO 99 I = 5, 999
     A(I) = D(I)
     E(I) = A(I-2)+D(I)*C(I)
      A(I-3) = F(I)+G(I)
      D(I) = A(I-4)
      B(I+1) = D(I)
 99   CONTINUE
```

есть контур, проходящий через все операторы тела цикла. Такой цикл непосредственно нельзя разбить на меньшие (даже после какой-либо перестановки

операторов) или векторизовать. После переименования переменной получим следующий фрагмент программы

```
      DO 99 I = 5, 999
     AA(I) = D(I)
     E(I) = AA(I-2)+D(I)*C(I)
      A(I-3) = F(I)+G(I)
      D(I) = A(I-4)
      B(I+1) = D(I)
 99   CONTINUE
A(997)=AA(997)
A(998)=AA(998)
A(999)=AA(999)
```

В полученном фрагменте граф информационных связей бесконтурный - результирующий цикл можно разбивать или векторизовать.

Конец примера.

ИНВЕРСИЯ ЦИКЛА

Суть инверсии в изменении порядка вычисления итераций цикла на противоположный.

Если все итерации цикла независимы, то инверсия цикла является эквивалентным преобразованием.

Пример 79. Данный цикл

```
      DO 99 i = 1, N
99    X(i) =A(i) * B(i)
```

равносилен следующему

```
      DO 99 i = 1, N
99    X(N+1-i) =A(N+1-i) * B(N+1-i)
```

Инверсия является равносильным преобразованием не только для независимых циклов.

Пусть # – ассоциативная операция с нейтральным элементом $ и пусть задан одномерный цикл вида

```
      DO 99 i = 1, N
99    X =A(i) # X
```

Инверсия этого цикла состоит в замене этого одномерного цикла на следующий фрагмент программы

```
      Y = $
      DO 99 i = 1, N
99    Y = Y # A(N-i+1)
      X = Y # X
```

Если операция # еще и коммутативна, то инверсия состоит в замене исходного цикла на следующий

```
      DO 99 i = 1, N
99    X =A(N-i+1) # X
```

Примерами ассоциативной и коммутативной операции # могут служить сложение, умножение, конъюнкция, дизъюнкция, максимум и минимум (двух чисел). Некоммутативной, но ассоциативной операцией является, например, произведение матриц. По поводу ассоциативности операций над числами с плавающей запятой следует отдельно удостовериться в достаточной точности при округлениях.

Инверсия цикла меняет информационный граф программы и этим самым может привести к возможности распараллеливания изначально не распараллеливаемый фрагмент программы.

Пример 80. Решетчатый граф программы следующего нетесного гнезда циклов содержит путь, проходящий через все вершины этого графа. Это не позволяет одновременно вычислять никакие две итерации этого фрагмента программы.

```
      DO 88 i = 1, N
      DO 99 j = 1, i-1
99    H(N-i+1) = H(N+i-1) - A(N-i+1,N-i+j+1)* H(N-i+j+1)
88    H(N-i+1) = H(N+i-1)/A(N-i+1,N-i+1)
```

Если сделать инверсию вложенного цикла, то решетчатый граф программы изменится и его можно будет распараллеливать.

```
      DO 88 i = 1, N
      DO 99 j = 1, i-1
99    H(N-i+1) = H(N+i-1) - A(N-i+1,N-j+1)* H(N-j+1)
88    H(N-i+1) = H(N+i-1)/A(N-i+1,N-i+1)
```

Заметим, что программа данного примера представляет собой решение системы линейных алгебраических уравнений с треугольной матрицей.

Конец примера.

Пример 81. В данном цикле есть циклически порожденная антизависимость и он не может быть инвертирован.

```
      DO 99 j = 1, N
99    H(j) =  A(j)* H(j+1)
```

Пример 82. В данном цикле все итерации независимы и он может быть инвертирован. Циклически независимая зависимость не препятствует инверсии.

```
      DO 99 j = 1, N
99    H(j) =  A(j)* H(j)
```

Конец примера.

Пример 83. В данном цикле есть истинная информационная зависимость, и он не может быть инвертирован.

```
      DO 99 j = 1, N
99    H(j) =  A(j)* H(j-1)
```

Конец примера.

Пример 84. В данном цикле выполняется ассоциативная операция дробнолинейное преобразование, и этот цикл может быть инвертирован.

```
      DO 99 j = 1, N
99    H =  A(j)* H +B(j)/ C(j)*H+D(j)
```

Конец примера.

ВЫНОС ОПЕРАТОРА ИЗ ЦИКЛА

Данное преобразование заменяет цикл

```
    DO 333 I = 1, N
    S1
    .....
    S(k-1)
    Sk
    S(k+1)
    .....
    Sm
333  CONTINUE
```

на следующий фрагмент программы

```
   Sk
    DO 333 I = 1, N
    S1
    .........
    S(k-1)
    S(k+1)
    .........
    Sm
333  CONTINUE
```

Теорема 27. Пусть тело исходного цикла состоит из трех фрагментов, каждый из которых имеет один вход и один выход.

```
    DO 333 I = 1, N
    FRAG1(i)
    FRAG2(i)
    FRAG3(i)
333  CONTINUE
```

Пусть в фрагменте FRAG2 все генераторы не зависят от счетчика цикла и предположим, что в этом фрагменте нет истинных циклически порожденных зависимостей. Тогда:

1. Если нет дуг графа информационных связей ведущих в фрагмент FRAG2 из фрагментов FRAG1 и FRAG3 , то исходный цикл равносилен следующему

```
    FRAG2(N)
    DO 333 I = 1, N
    FRAG1(i)
    FRAG3(i)
333  CONTINUE
```

2. Если нет дуг графа информационных связей ведущих из фрагмента FRAG2 в фрагменты FRAG1 и FRAG3 , то исходный цикл равносилен следующему

```
    DO 333 I = 1, N
    FRAG1(i)
    FRAG3(i)
333  CONTINUE
    FRAG2(N)
```

Доказательство. Заметим, что данное преобразование может быть сведено к двум: к разрезанию цикла, а, затем, к удалению заголовка. Рассмотрим, подробно первый случай, а второй может быть рассмотрен аналогично.

Поскольку нет дуг графа информационных связей ведущих в фрагмент FRAG2 из фрагментов FRAG1 и FRAG3 , то исходный цикл с помощью разрезания (и предварительной перестановки фрагментов) может быть преобразован к следующему равносильному фрагменту:

```
    DO 222 I = 1, N
    FRAG2(i)
222  CONTINUE
    DO 333 I = 1, N
    FRAG1(i)
    FRAG3(i)
333  CONTINUE
```

Поскольку в фрагменте FRAG2 , который является телом цикла 222, все генераторы не зависят от счетчика цикла и в этом фрагменте нет истинных циклически порожденных зависимостей, заголовок цикла можно удалить, а вместо счетчика цикла подставить его последнее значение N. В результате получиться описанный в формулировке теоремы фрагмент.

Пример 85. Цикл

```
      DO 99 I = 1, 100
      X = A+1
      K(I) = X
 99   CONTINUE
```

эквивалентен следующему фрагменту программы

```
      X = A+1
       DO 99 I = 1, 100
       K(I) = X
  99   CONTINUE
```

Конец примера.

Пример 86. Цикл

```
       DO 99 I = 1, 100
      X = A+X
      K(I) = X
    99 CONTINUE
```

не эквивалентен следующему фрагменту программы

```
      X = A+X
     DO 99 I = 1, 100
      K(I) = X
 99 CONTINUE
```

Конец примера.

Пример 87. Цикл

```
      DO 99 I = 1, 100
      K = K+1
      X(K) = B
```

```
  99 CONTINUE
```
не эквивалентен следующему фрагменту программы
```
       X(K) = B
      DO 99 I = 1, 100
      K = K+1
99   CONTINUE
```
Конец примера.

Пример 88. Цикл
```
      DO 99 I = 1, 100
      IF (A(I)<B(I)) goto 33
      X(K) = 1
33   Y(I) = 0
99   CONTINUE
```
не эквивалентен следующему фрагменту программы
```
       X(K) = 1
     DO 99 I = 1, 100
     IF (A(I)<B(I)) goto 33
33   Y(I) = 0
99   CONTINUE
```
Конец примера.

Пример 89. Цикл
```
       DO 99 I = 1, 100
       X = A+X
       K = B(I)+7
  99 CONTINUE
```
эквивалентен следующему фрагменту программы
```
       DO 99 I = 1, 100
       X = A+X
   99 CONTINUE
     K = B(100)+7
```
Конец примера.

Пример 90. Цикл
```
       DO 99 I = 1, 100
       X = A(I)+X
       K = B(I)+X
   99 CONTINUE
```
эквивалентен следующему фрагменту программы
```
        DO 99 I = 1, 100
       X = A(I)+X
   99 CONTINUE
        K = B(100)+X
```
Конец примера.

Пример 91. Цикл

```
      DO 99 I = 1, 100
      K = B(I)+X
      X = A(I)+X
   99 CONTINUE
```

не эквивалентен следующему фрагменту программы

```
      DO 99 I = 1, 100
      X = A(I)+X
   99 CONTINUE
     K = B(100)+X
```

Конец примера.

Пример 92. Цикл

```
      DO 99 I = 1,N
      K = B+X
      X = A(I)+X
   99 CONTINUE
```

не эквивалентен следующему фрагменту программы

```
      K = B+X
      DO 99 I = 1,N
      X = A(I)+X
   99 CONTINUE
```

поскольку существует дуга истинной зависимости от второго оператора к первому.

Конец примера.

Пример 93. Цикл

```
       DO 99 I = 1,N
       K = B+X
       X = A(I)
    99 CONTINUE
```

не эквивалентен следующему фрагменту программы

```
      DO 99 I = 1,N
       X = A(I)+X
    99 CONTINUE
       K = B+X
```

поскольку существует дуга антизависимости от первого оператора ко второму.

Конец примера.

Пример 94. Цикл

```
      DO 99 I = 1,N
      K = B+X
      X = C*4
      T(I)= A(I)+X
   99 CONTINUE
```

эквивалентен следующему фрагменту программы

```
      K = B+X
      X = C*4
      DO 99 I = 1,N
      T(I)= A(I)+X
   99 CONTINUE
```
Конец примера.

ИНДУКТИВНЫЕ И ОХВАТЫВАЮЩИЕ ПЕРЕМЕННЫЕ

Переменная называется индуктивной, если она при выполнении цикла принимает значения, образующие арифметическую прогрессию.

Любой из следующих двух случаев является достаточным для того, чтобы переменная была индуктивной:

1. В теле цикла лишь в одном операторе присваивания эта переменная встречается в качестве генератора, это вхождение в теле цикла встречается раньше использований и в этом операторе данная переменная линейно выражается через счетчик цикла.
2. В теле цикла данная переменная рекуррентно перевычисляется, причем рекуррентная зависимость имеет вид A=A+Const, где A – индуктивная переменная, а Const – заранее известное число.

Пример 95. В рассматриваемом цикле переменные INC и I2 являются индуктивными.

```
      INC=N
      DO 99 I=1,N
      I2=2*I-1
      X(INC)=Y(I)+Z(I2)
      INC=INC-1
      T(INC)=A(I)
99    CONTINUE
```

Если индуктивные переменные встречаются в индексных выражениях переменных, то они затрудняют вычисление информационных зависимостей. Для вычисления информационных зависимостей индуктивную переменную можно выразить через счетчик цикла и всюду по тексту заменить эту переменную ее значением. Приведенный выше цикл преобразится к виду

```
      DO 99 I=1,N
      X(N-I+1)=Y(I)+Z(2*I-1)
      T(N-I)=0
99    CONTINUE
      INC=1
      I2=2*N-1
```

Конец примера.

Для выражения индуктивных переменных через счетчик цикла надо использовать тот факт, что I-ый член арифметической прогрессии равен первому члену плюс разность прогрессии, умноженная на (I-1). Рассмотрим получение этих формул для обоих видов индуктивных переменных.

1. В теле цикла лишь в одном операторе присваивания эта переменная встречается в качестве генератора и линейно выражается через счетчик цикла. Тогда выполняется преобразование подстановка. Заметим, что всякое использование индуктивной переменной, которое встречается ниже по тексту генератора, заменяется просто правой частью этого оператора присваивания.

Если индуктивная переменная рекуррентно перевычисляется, то ее вхождения могут встречаться до генератора и после, и такие вхождения будут поразному выражаться через счетчик цикла (они заменяются на (I-1)-ый и I-ый члены арифметической прогрессии соответственно).

Пример 96.

```
      K=3
      J=N
      do 99 I=1,N
      B(I)=(A(J)+A(K))/2
      K=K+1
      J=I
 99   CONTINUE
```

В этом цикле переменная K является индуктивной, а переменная J - охватывающая. Этот цикл эквивалентен следующему.

```
      K=3
      J=N
      B(1)=(A(J)+A(K))/2
    K=K+1
    J=1
    DO 99 I=2,N
    B(I)=(A(J)+A(K))/2
    K=K+1
    J=I
99  CONTINUE
```

Конец примера.

Переменная называется охватывающей, если при выполнении цикла все ее значения, кроме, возможно, первого, образуют арифметическую прогрессию. Индуктивная переменная является частным случаем охватывающей. Охватывающая переменная не всегда является индуктивной. Например, если в первом

случае разрешить появление этой переменной в качестве использования раньше, чем, в качестве генератора.

Пример 97.

```
      J=N
      do 99 I=1,N
      B(I)=(A(J)+A(I))/2
      J=I
      C(I)=(A(J)+A(I))/2
99    CONTINUE
```

Если отщепить от цикла первую итерацию, то охватывающая переменная становится индуктивной.

Конец примера.

Преобразование циклов, содержащих охватывающие переменные, состоит в следующем:

1. Отделяем от цикла первую итерацию. Для остальных итераций охватывающая переменная принимает значения, образующие арифметическую прогрессию.

2. В оставшейся части цикла заменяем охватывающую переменную выражениями, содержащими счетчики цикла: при этом необходимо учитывать, что вхождения охватывающей переменной, которые встречаются до генератора и после имеют разные значения.

Рассмотренный выше цикл с охватывающей переменной может быть преобразован к следующему фрагменту программы

```
      J=N
      if (N>=1) then
      B(1)=(A(J)+A(1))/2
      C(1)=(A(1)+A(1))/2
      do 99 I=1,N
      B(I)=(A(I-1)+A(I))/2
99    C(I)=(A(I)+A(I))/2
      end if
      J=N
```

Кроме индуктивных и охватывающих переменных в теле цикла могут встречаться и другие переменные, меняющие свое значение в процессе выполнения цикла.

Пример 98. Следующая программа реализует алгоритм параллельной прогонки. Переменная L меняет свои значения в процессе выполнения внешнего цикла. Она не является индуктивной или охватывающей, поскольку значения этой переменной образуют геометрическую прогрессию.

```
      do 99 K=1,log(N)        %%N=2**M
        L=2**(K-1)
      do 99 J=L+1,N
      H(J)= H(J)-A(J)*H(J-L)
      A(J)=A(J)*A(J-L)
 99   CONTINUE
```
Конец примера.

Пример 99.

```
      do 99 I=1,N
      K=2*K
      L=2*L+I
      KK=K+1
      J=L-K
      C(I)=A(J)+B(KK).
 99   CONTINUE
```

В этом примере переменные K, L, KK, J по-разному меняют свои значения в процессе выполнения цикла, но ни одна из них не является индуктивной или охватывающей переменной.

Конец примера.

Дополнительные примеры.

Пример 100. Рассматриваемый цикл содержит охватывающую переменную

```
      DO 99 I=1,N
       Y(I) = W(I2+3)
      I2=2*I-1
      X(I)=Y(I)+Z(I2)
99    CONTINUE
```

равносилен следующему

```
       Y(1) = W(I2+3)
      I2=1
      X(1)=Y(1)+Z(1)
      DO 99 I=1,N
       Y(I) = W((2*(I-1)-1)+3)
      I2=2*I-1
      X(I)=Y(I)+Z(2*I-1)
99    CONTINUE
```

Конец примера.

РЕШЕТЧАТЫЙ ГРАФ, КАК МОДЕЛЬ ИНФОРМАЦИОННОЙ ЗАВИСИМОСТИ В ГНЕЗДАХ ЦИКЛОВ

ЛИНЕЙНЫЙ КЛАСС ПРОГРАММ

Построение решетчатых графов и многие преобразования, на них основанные, возможны в рамках линейного класса [10, с. 340], включающего в себя следующие элементы языка:

- скалярные и индексные переменные;
- операторы присваивания, правая часть которых является арифметическим выражением;
- циклы Do языка программирования Fortran (либо эквивалентные им циклы `for` языка C); структура вложенности циклов может быть произвольной; шаги изменения параметров циклов всегда равны +1; если у цикла нижняя граница больше верхней, то цикл не выполняется;
- условные и безусловные операторы перехода, передающие управление «вниз» по тексту; не допускается использование побочных выходов из циклов;
- все индексные выражения переменных, границы изменения параметров циклов и условия передачи управления задаются аффинными формами от совокупности параметров циклов и внешних переменных программы; все коэффициенты при переменных в этих формах являются целыми числами;
- внешние переменные программы всегда целочисленные.

Пример 101. Следующий фрагмент принадлежит линейному классу программ.

```
for(i=0; i<N; i=i+1)
{   a[N–i+1] = 2*b[3*i] ;
    for(j=0; j<M1; j=j+1)
      a[i+2*j] = b[i-j+2*N];
    for(j=0; j<M2; j=j+1)
      if((i+j)>M0)
         t = d[i,j+i+2];
}
```

Конец примера.

Пример 102. Рассмотрим следующий фрагмент программы.

```
for(i=0; i<N; i=i+2)
{   for(j=0; a[j]<M1; j=j+1)
      a[i+2*j] = b[i*j];
    for(j=0; j<M2; j=j+1)
      if(a[i]>0)
         b[f[i]] = b[j*N]-d[i,j];
   function_call();
}
```

Этот фрагмент не принадлежит линейному классу по следующим причинам:
1. шаг внешнего цикла не равен 1;
2. условие окончания работы второго цикла не аффинное;
3. все индексные выражения массива b содержат выражения относительно счетчиков циклов и внешней переменной N, которые не являются аффинными;
4. выражение условного оператора не является аффинной формой относительно совокупности счетчиков циклов и внешних переменных;
5. есть вызов функции.

Конец примера.

В частности, следует отметить, что программы нахождения максимального элемента массива или сортировки не принадлежат линейному классу. Многие оптимизируемые (требующие больших объемов вычислений) участки программ классических численных методов принадлежат линейному классу или сводятся к таким. Не относятся к линейному классу программы для метода конечных элементов, поскольку матрицы в этих случаях разреженные и работа с ними предполагает косвенную адресацию.

В защиту линейного класса программ заметим, что большинство оптимизирующих/распараллеливающих преобразований программ, как правило, носят локальный, а не глобальный характер. А это означает, что требования линейности предъявляются не ко всей программе, а только к оптимизируемому фрагменту.

Последний аргумент в пользу исследований линейного класса программ состоит в том, что традиционный граф информационных связей, который основан на неравенствах Банерджи или Омега-тесте, строится тоже только для линейного класса.

ПРОСТРАНСТВО ИТЕРАЦИЙ ГНЕЗДА ЦИКЛОВ

Будем рассматривать общий случай многомерных тесногнездовых циклов кратности n, содержащих вхождения многомерных массивов размерности m.

```
      DO 99 I1=L1,R1
      DO 99 I2=L2,R2
.....................................
      DO 99 In=Ln,Rn                                    (11)
      LOOPBODY (I1,I2,...,In)
99    CONTINUE
```

Пространство итераций многомерного цикла (11) – это множество всех таких целочисленных векторов I=(I1,I2,...,In), что

L1=<I1=<R1
L2=<I2=<R2
.....

Будем допускать, что границы Lj, Rj изменения счетчиков циклов Ij могут линейно зависеть от счетчиков объемлющих циклов. В этом случае пространство итераций D многомерного цикла (11) является целочисленным многогранником в n-мерном пространстве. Цикл (11) будем более компактно записывать в следующем виде

```
      DO 99 I D
      LOOPBODY (I)                                      (12)
99    CONTINUE
```

На пространстве итераций D будем рассматривать лексикографический порядок. Например, (2,4,5)>(2,3,8). Именно в лексикографическом порядке просматриваются итерации цикла при его выполнении на компьютере (не параллельном).

Пусть D0 – подмножество пространства итераций D и точка I0 принадлежит этому подмножеству. Будем говорить, что точка I0 является лексикографическим максимумом множества D0, если любая другая точка I из этого подмножества лексикографически меньше точки I0.

Теорема 28. Лексикографический максимум точек выпуклого многогранника достигается в одной из его вершин.

Доказательство. Рассмотрим функционал, сопоставляющий каждой точке выпуклого многогранника его первую координату. Этот функционал линейный и поэтому достигает своих экстремальных значений на вершинах многогранника (их может быть несколько). Пусть наибольшее значение этого функционала x_{10} . Рассмотрим сечение многогранника гиперплоскостью $x_1 = x_{10}$. Это сечение представляет собой множество всех точек, на которых рассматриваемый

функционал достигает максимального значения. Это сечение, очевидно, тоже является выпуклым многогранником. На нем рассмотрим функционал, сопоставляющий каждой точке ее вторую координату. Найдем ее максимальное значение x_{20} … и т.д. по индукции, пока не будут найдены все координаты (x_{10}, x_{20}, …, x_{n0}), определяющие единственную точку максимума.

При последовательном выполнении гнезда циклов точки пространства итераций просматриваются в лексикографическом порядке – это вытекает из семантики операторов цикла языков С, ФОРТРАН и ПАСКАЛЬ.

Пример 103.

На плоскости в треугольнике с вершинами A(1,2), B(5,4), C(4,7) лексикографическим максимумом является вершина B.

Конец примера.

Пример 104.

В трехмерном пространстве в параллелепипеде (декартово произведение трех отрезков) [1,9]x[2,7]x[1,3] точкой лексикографического максимума является вершина (9,7,3).

Конец примера.

При исследовании информационной зависимости между двумя вхождениями некоторого m-мерного массива X будем выписывать абстрактное тело цикла, состоящее только из этих вхождений. Остальные операторы, операнды или операции будут предполагаться не влияющими на рассматриваемую информационную зависимость, или их влияние будет оговариваться. Цикл с выделенными вхождениями будем выписывать следующим образом

```
      DO 99 I D
      X(F1(I),...,Fm(I)) = . . .
            ……................
            . . .    = X(G1(I),..,Gm(I))
99    CONTINUE
```

или так

```
       DO 99 I D
      X(F(I)) = . . .
      ……................
            . . .    = X(G(I))                                    (13)
99    CONTINUE
```

Пространство итераций фрагмента программы – это объединение пространств итераций всех гнезд циклов, входящих в данный фрагмент программы. Если некоторый оператор не принадлежит ни одному циклу, то будем считать, что этот оператор принадлежит гнезду кратности 0.

РЕШЕТЧАТЫЙ ГРАФ И ЕГО СООТНОШЕНИЕ С ГРАФОМ ИНФОРМАЦИОННЫХ СВЯЗЕЙ

Решетчатые графы использовались в методах распараллеливания программ с семидесятых годов в работах Л. Лампорта в методе гиперплоскостей [**60**], В. Вальковского в методе параллелепипедов [**6**] и методе пирамид [**5**] и у других исследователей. Эти графы использовались для иллюстрации идей, лежащих в основе методов распараллеливания. В явном виде эти графы не использовались, поскольку традиционные формы хранения графа в виде матрицы смежностей или в виде списка дуг не эффективны: прочтение такого графа может потребовать больше времени, чем последовательное исполнение программы, по которой этот граф построен. В конце 80-х годов был сделан прорыв в этом направлении: два математика В.В. Воеводин [**9**], [**10**], [**11**], [**69**] и P. Feautrier [**57**], [**58**], [**56**], [**55**] научились хранить этот граф в компактном виде, в виде функций, и разработали алгоритмы построения этих функций. У P. Feautrier функция строится для программ из линейного класса и содержит в своем определении не очень удобные для исследования условные операторы. У В.В. Воеводина функция является кусочно-линейной и определена на подмножестве линейного класса программ; но большинство практически значимых программ линейного класса относятся к этому подмножеству. В теории решетчатых графов можно отметить также работы [**44**], [**47**], [**34**].

Использование решетчатых графов в распараллеливающих компиляторах – одно из направлений исследований группы «Открытая распараллеливающая система» (ОРС). Эти исследования проводятся в следующих направлениях:

1. Использование решетчатых графов в преобразованиях программ, которые не могут быть основаны на традиционном графе программных зависимостей.
2. Использование решетчатых графов для отображения последовательных программ на параллельную архитектуру.
3. Расширение класса программ, к которому применимы методы теории решетчатых графов.

Пусть в гнезде циклов имеется пара зависимых вхождений X(F(I)) и X(G(J)) и этой паре соответствует некоторая дуга графа зависимостей по данным. Зависимость означает обращение этих вхождений к одной и той же ячейке памяти. Но на разных итерациях гнезда циклов эти вхождения могут обращаться к разным ячейкам памяти. Возникает задача описания всех таких пар итераций гнезда циклов (I,J), для которых оба вхождения X(F(I)) и X(G(J)) обращаются к одной и той же ячейке памяти. Для описания этих пар итераций используется элементарный решетчатый граф программы.

Можно элементарный решетчатый граф определить не только для произвольной дуги информационной зависимости, а более общо, для произвольной

упорядоченной пары вхождений X(F(I)) и X(G(J)) – если окажется, что эта пара не связана дугой графа информационных связей, то решетчатый граф будет пустым (без дуг).

Определение элементарного решетчатого графа. Пусть в гнезде циклов имеется пара зависимых вхождений X(F(I)) и X(G(J)) и этой паре соответствует некоторая дуга графа информационных связей. Через (F,G) будем обозначать элементарный решетчатый граф, построенный по двум вхождениям X(F) и X(G) переменной X в гнезде циклов. D - множество вершин этого графа – пространство итераций цикла (13). Дуга (I,J) принадлежит графу, если F(I)=G(J) , I<J (лексикографический порядок) и для любого такого K из области D, для которого K<J и F(K)=G(J), выполняется K =< I.

Иными словами, вершина I является лексикографическим максимумом множества всех таких точек K, для которых F(K)=G(J) и K<J. В [**9**] (F,G) называется элементарным решетчатым графом. Всякий раз при упоминании графа (F,G) будем предполагать, что индексные отображения F и G не содержат переменных, кроме счетчиков рассматриваемых объемлющих циклов – в этом случае граф (F,G) однозначно определен.

Описанный граф называется решетчатым, поскольку его вершины (точки пространства итераций) располагаются в узлах многомерной целочисленной решетки (размерность равна количеству циклов в гнезде).

Решетчатый граф строится по дуге графа информационных связей. Поскольку дуги графа информационных связей подразделяются на четыре типа зависимостей, решетчатые графы тоже имеют такие же типы: истинная (потоковая), анти-зависимость, выходная зависимость и входная зависимость.

Определение решетчатого графа истинной зависимости переменной Пусть задан фрагмент программы и некоторая переменная (м.б. индексная) X. Множество вершин решетчатого графа истинной зависимости переменной – это пространство итераций заданного фрагмента. Две точки пространства итераций I и J соединяются дугой, если

1. в графе информационных связей существует дуга истинной зависимости (u1, v1), решетчатый граф которой содержит дугу (I,J);
2. для любой другой дуги (u2, v1) истинной зависимости графа информационных связей, решетчатый граф которой содержит некоторую дугу (I2,J) , входящую в вершину J, точка пространства итераций I2 лексикографически меньше I, либо I2 = I, но вхождение u2 в тексте программы встречается всегда раньше вхождения u.

Аналогично определяются решетчатые графы переменной для антизависимости, выходной зависимости и входной зависимости.

Определение (полного) решетчатого графа фрагмента программы Решетчатый граф фрагмента программы – это граф, вершинами которого являются все точки пространства итераций, а множество дуг является объединением множеств дуг решетчатых графов всех переменных данного фрагмента.

Из определения решетчатого графа вытекает следующее свойство:

- в каждую вершину элементарного решетчатого графа входит не более одной дуги.

Из этого свойства вытекает возможность хранения решетчатого графа в виде отображения подмножества пространства итераций в пространство итераций: концу дуги решетчатого графа сопоставляется начало этой дуги.

Традиционные способы хранения графов в виде матрицы смежностей или списка дуг не приемлемы для решетчатых графов. Действительно, для гнезд циклов реальных долго считаемых программ количество дуг решетчатого графа может быть так велико, что прочтение всех дуг может потребовать время, сравнимое с временем последовательного выполнения исходной программы и распараллеливание, основанное на таком графе потеряет смысл.

Функция, описывающая решетчатый граф, может занимать объем, не зависящий от размерностей циклов, описывающих пространство итераций. Эта функция является кусочно-квазилинейной. «Квази-линейная» - это значит, что кроме линейных операций над счетчиками циклов эта функция допускает взятие целой части числа или условные операторы. Во многих практически важных случаях эта функция оказывается кусочно линейной.

Пример 105.

```
      DO 99 I=1,N
      DO 99 J=1,M
       Y(I) =
             = Y(J)
99    CONTINUE
```

В этом примере в точке пространства итераций с координатами (5,2) используется элемент Y(2) , который вычисляется на итерациях (2,1), (2,2), …, (2,M). Среди всех этих точек лексикографически самой старшей является точка (2,M). Таким образом, в элементарном решетчатом графе в точку с координатами (5,2) входит дуга из точки с координатами (2,M).

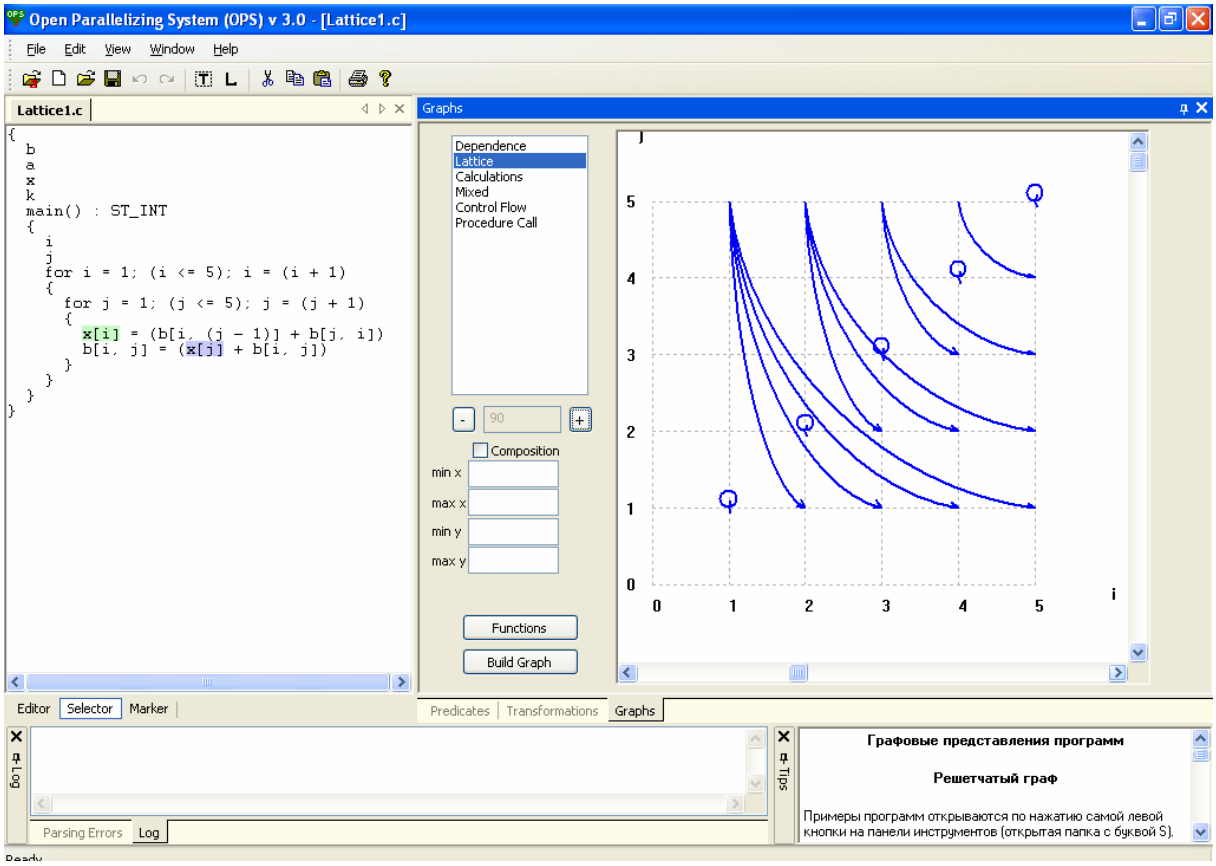


Рис. 1. Решетчатый граф построенный с помощью OPC. Петли соответствуют циклически независимой зависимости.

Кусочно-линейная функция, описывающая решетчатый граф, автоматически построенная в OPC:

Function:
i' = 5
j' = 5

Added Var(s)' Equation(s):

Definition Area:

-1*i+0*j <= -5
+1*i+0*j <= 5
+0*i-1*j <= -5
+0*i+1*j <= 5

Function:
i' = i +0
j' = j +0

Added Var(s)' Equation(s):

Definition Area:

-1*i+0*j <= -1
+1*i+0*j <= 4
+0*i-1*j <= -1
+0*i+1*j <= 4
-1*i+1*j <= 0
+1*i-1*j <= 0

Function:
i' = i -1
j' = 5

Added Var(s)' Equation(s):

Definition Area:

-1*i+0*j <= -2
+1*i+0*j <= 5
+0*i-1*j <= -1
+0*i+1*j <= 5
-1*i+1*j <= -1
+1*i-1*j <= 1

Function:
i' = j +0
j' = 5

Added Var(s)' Equation(s):

Definition Area:

-1*i+0*j <= -1
+1*i+0*j <= 5
+0*i-1*j <= -1
+0*i+1*j <= 5
-1*i+1*j <= -2

Конец примера.

При автоматическом выполнении преобразований программ необходимо контролировать эквивалентность этих преобразований. Для преобразований линейных участков и одномерных циклов эквивалентность таких преобразований выражается с помощью графа информационных связей. Для преобразований программ, содержащих многомерные циклы, обычно используют направление зависимости [**65**]. В данной работе для контроля эквивалентности будет использоваться решетчатый граф. Следующие два утверждения показывают связь между решетчатым графом и графом информационных связей.

Теорема 29. В графе информационных связей цикла (13) существует дуга от генератора переменной X к использованию этой переменной (дуга истинной информационной зависимости), причем использование X(G) использует значение, записанное в память генератором X(F), тогда и только тогда, когда в графе (F,G) существует хотя бы одна дуга.

Доказательство вытекает непосредственно из определений графа (F,G) и графа информационных связей.

Теорема 30. Пусть (X(F(I)), X(G(I))) – дуга истинной информационной зависимости в графе информационных связей и (F,G) – соответствующий решетчатый граф. Дуга (I,J) принадлежит графу (F,G) тогда и только тогда, когда в раскрутке цикла (13) вхождение X(G(J)) использует данное, записанное в память генератором X(F(I)).

Доказательство вытекает из определения графа (F,G) и определения раскрутки цикла.

Таким образом, существует взаимно однозначное соответствие между дугами графа (F,G) цикла (13) и существенными дугами графа информационных связей раскрутки. (Под существенной дугой понимается такая дуга истинной зависимости, для которой использование, соответствующее концу дуги, использует данные, записанные в память генератором, соответствующим началу дуги).

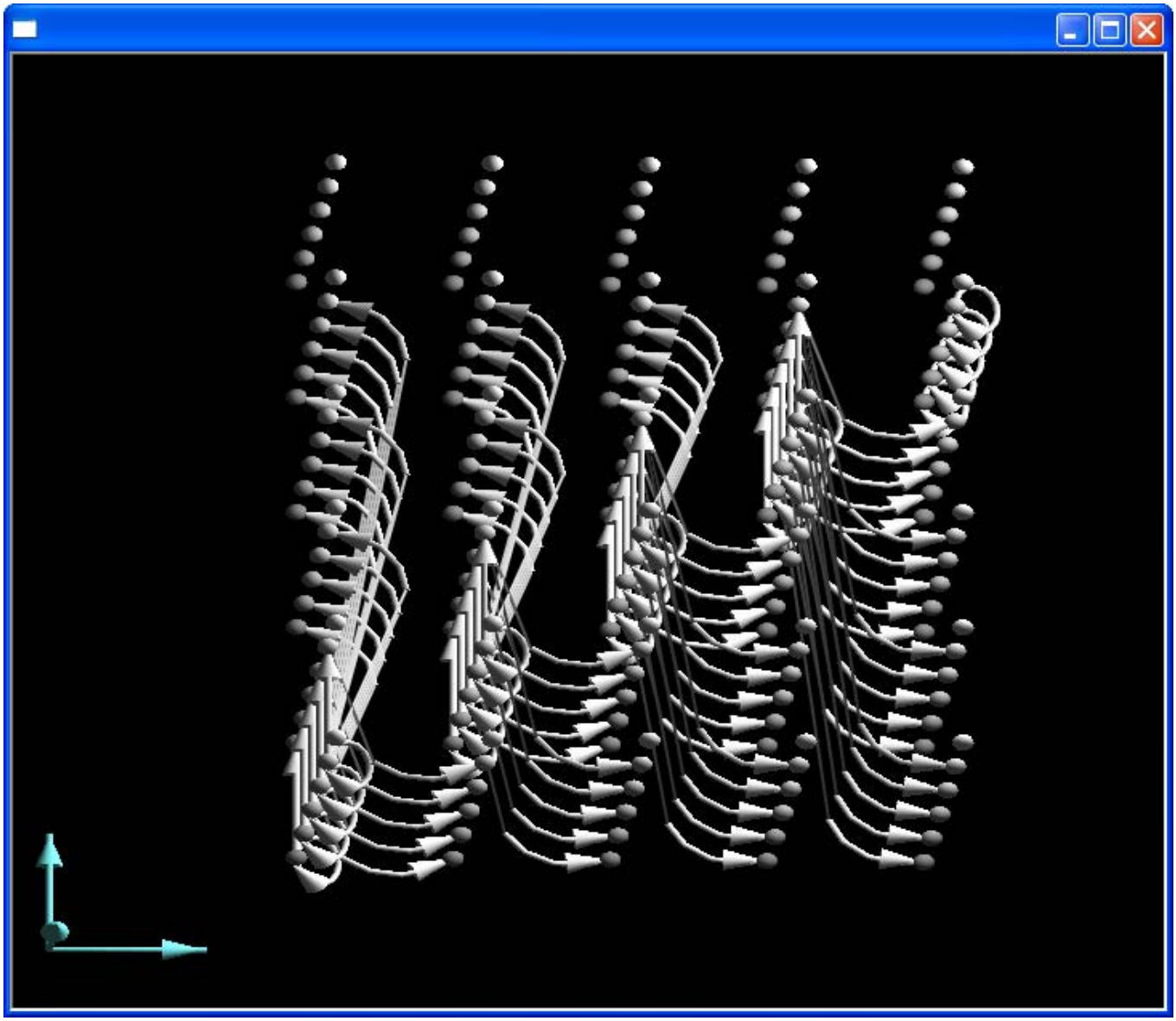


Рис. 2. Пример 3D визуализации трехмерного решетчатого графа

ПЕРЕСТАНОВКА ЦИКЛОВ В ТЕСНОМ ГНЕЗДЕ

Рассмотрим задачу перестановки местами операторов цикла (заголовков циклов) в гнезде циклов. Такая задача возникает, если какой-то из циклов гнезда удобнее сделать самым глубоко вложенным или, наоборот, самым объемлющим. Такая перестановка может повысить эффективность при использовании кэш-памяти или при отображении гнезда циклов на параллельную архитектуру (векторную, конвейерную, асинхронную …).

Сначала будем рассматривать тесное гнездо, состоящее из двух циклов.

```
      DO 99 I1=1, N
      DO 99 I2=1, M                                    (14)
      LOOPBODY (I1,I2)
99    CONTINUE
```

Результат перестановки – следующее гнездо циклов

```
      DO 99 I2=1, M
      DO 99 I1=1, N                                          (15)
      LOOPBODY (I1,I2)
99    CONTINUE
```

Корректность этого преобразования по управлению требует, чтобы у заголовков циклов не было меток, на которые были бы передачи управления (по GOTO) и, чтобы из тела цикла не было выходов, иных, чем через заголовок.

Теорема 31 (об изменении порядка обхода пространства итераций). Пусть D – пространство итераций рассматриваемого гнезда циклов. Лексикографический порядок определяет очередность выполнения итераций гнезда циклов при последовательном выполнении. Предположим, что задана некоторая новая очередность выполнения итераций гнезда циклов. Предположим, что для каждой дуги (I,J) решетчатого графа исходной программы в новой последовательности выполнения итераций итерация I тоже предшествует итерации J. Тогда новая очередность выполнения точек пространства итераций эквивалентна исходной.

Доказательство. Сделаем раскрутку циклов исходного гнезда циклов. В результате раскрутки исходное гнездо циклов превратится в последовательность копий тела циклов (итераций гнезда циклов). Дугам решетчатого графа исходного гнезда циклов будут соответствовать дуги графа информационных связей раскрутки гнезда. Копии тела циклов можно переставлять местами так, чтобы вхождения, соответствующие концу дуги всегда были после вхождения, соответствующего началу дуги – но это требование оговорено в условии теоремы.

Теорема 32. Пусть для каждой дуги графа информационной зависимости (X(G), X(F)) исходного гнезда циклов в решетчатом графе (F,G) не существует дуга (I,J), $I= (I_1 , I_2 , \ldots, I_n)$, $J=(J_1 , J_2 ,\ldots, J_n)$ такая, что для некоторых индексов $k<m$ выполняется $(I_k < J_k)$ и $(I_m > J_m)$. Тогда перестановка циклов является эквивалентным преобразованием.

Доказательство. Сделаем раскрутку циклов исходного и результирующего гнезд. В результате раскрутки исходное гнездо циклов превратится в последовательность копий гнезд циклов:

```
LOOPBODY(1,1)
LOOPBODY(1,2)
…
LOOPBODY(1,M)
LOOPBODY(2,1)
LOOPBODY(2,2)
…
```

LOOPBODY(2,M)
…
LOOPBODY(N,1)
LOOPBODY(N,2)
…
LOOPBODY(N,M)

После раскрутки второго гнезда циклов получится следующая последовательность копий тела цикла

LOOPBODY(1,1)
LOOPBODY(2,1)
…
LOOPBODY(N,1)
LOOPBODY(1,2)
LOOPBODY(2,2)
…
LOOPBODY(N,2)
…
LOOPBODY(1,M)
LOOPBODY(2,M)
…
LOOPBODY(N,M)

Теперь для доказательства теоремы следует проверить условия перестановки копий тел цикла в одной последовательности для получения другой последовательности.

РАСПАРАЛЛЕЛИВАНИЕ

РАЗНОВИДНОСТИ РАСПАРАЛЛЕЛИВАНИЯ ПРОГРАММ

Можно выделить следующие виды параллельных вычислений:

- Параллельное выполнение задач.
- Параллельное выполнение функций.
- Параллельное выполнение блоков.
- Параллельное выполнение циклов.
- Параллельное выполнение операций.
- Параллельное выполнение микрокоманд.

Параллельное выполнение задач – это функции операционной системы. Параллельное выполнение микрокоманд реализуется, как правило, аппаратно. Другие виды параллельных вычислений возлагаются либо на программиста, либо на распараллеливающий компилятор.

Под распараллеливанием понимается преобразование последовательной программы в параллельную. Таким образом, распараллеливание можно рассматривать, как частный случай оптимизации программ, и относиться к теории преобразования программ. Распараллеливание зависит от вычислительной архитектуры, на которой программы должны выполняться. Но одни и те же преобразования программ могут оказаться полезными для различных архитектур.

Параллельное выполнение функций и блоков предполагает асинхронную архитектуру MIMD (Many Instruction Multiple Data). Параллельное выполнение операций (иногда называют параллельным выполнением линейных участков) реализуется в архитектурах VLIW (Very Long Instruction Word) или суперскалярных. Циклы распараллеливаются на MIMD, SIMD (Single Instruction Multiple Data), конвейерные архитектуры.

У человека, который только приступает к параллельному программированию, возникают некоторые априорные представления об этой деятельности – гипотезы. Как правило, неверные. Приведем примеры.

Гипотеза 1. Если вычислительная система состоит из N параллельно работающих вычислительных устройств, то последовательная программа может быть выполнена в N раз быстрее.

Опровержение вытекает из понятной для школьников задачи о том, что сумму N чисел невозможно вычислить даже всем классом быстрее, чем за $\log_2(N)$ шагов. И это притом, что не учитывается расход времени на обмены данных.

То, как успех распараллеливания зависит от исходного алгоритма, иллюстрирует следующая задача.

Задача. *Предположим, что некоторая задача решается алгоритмом сложности 2^n, то есть, при количестве данных на входе n алгоритму требуется 2^n операций для решения задачи на однопроцессорном компьютере (существует много сотен таких практически значимых алгоритмов для решения NP-полных задач). Для $n=100$ однопроцессорный компьютер работает 1 час. Для какого значения n сможет решить задачу за 1 час двухпроцессорный компьютер? (Ответ: $n = 101$).*

Изучение параллельного программирования начинается с параллельных вычислительных архитектур. Здесь никак нельзя обойти SIMD и MIMD. В архитектуре SIMD все вычислительные устройства ведут одинаковые вычисления, а MIMD позволяет проводить разные вычисления. Это повод для следующей «очевидной» гипотезы.

Гипотеза 2. Всякий векторизуемый цикл может быть выполнен асинхронно.

Опровержение. Векторное выполнение следующего цикла равносильно его последовательному выполнению.

```
For I = 1 to N do
X[i] = 2*X[i+1]
```

Асинхронное параллельное выполнение этого же цикла может дать и другие результаты, например, если вторая итерация завершится до начала первой итерации.

Иногда используется упрощенная формулировка о том, что асинхронное параллельное выполнение процессов – это выполнение «в любом порядке». Здесь появляется еще одна «очевидная» гипотеза.

Гипотеза 3. Пусть программа F представляет собой последовательное выполнение двух фрагментов F_1 и F_2. Предположим, что последовательное выполнение этих фрагментов в произвольном порядке не влияет на результат. Тогда эти фрагменты можно выполнять асинхронно.

Опровержение. Пусть оба фрагмента F_1 и F_2 одинаковы и имеют вид

```
{
X = 1;
Y = X;
X = 2;
}
```

Перестановка местами одинаковых фрагментов, очевидно, является эквивалентным преобразованием (просто не меняет программу). Но параллельное асинхронное выполнение этих фрагментов некорректно. Действительно, при

последовательном выполнении программы переменная Y приобретает значение 1. Но, если операторы в процессах выполняются по расписанию из следующей таблицы, то по окончании параллельного исполнения значение переменной Y будет равно 2.

Номер процесса / Время В тактах	**1**	**2**
1	X = 1	X = 1
2	Y = X	
3	X = 2	
4		Y = X
5		X = 2

Из приведенных гипотез можно сделать выводы о том, что эвристические соображения следует подкреплять математическими доказательствами, и при преобразовании последовательных программ в параллельные следует анализировать программные зависимости.

При распараллеливании программ наиболее часто распараллеливаются циклы. С одной стороны это связано с тем, что наиболее долго в программах выполняются гнезда циклов. С другой стороны, под параллельным выполнением циклов обычно понимается одновременное выполнение итераций этого цикла, а различные итерации цикла отличаются только значениями счетчика цикла. Одинаковость параллельно выполняемых процессов (итераций цикла) при синхронном распараллеливании (SIMD) просто необходима, но и при асинхронном распараллеливании тоже существенна, поскольку такие процессы имеют шанс выполняться примерно одинаковое время, обеспечивая этим самым примерно одинаковую загрузку вычислительных устройств.

Но, оказывается, на разных вычислительных архитектурах могут параллельно вычисляться разные циклы. Термин «одновременное» допускает различные толкования и требует дополнительных пояснений. Распараллеливанию циклов могут мешать программные зависимости. Если в цикле есть только операторы присваивания и нет никаких информационных зависимостей, то итерации такого цикла можно выполнять независимо на любой из описанных типов архитектур. Разные вычислительные архитектуры допускают разные типы зависимостей.

В ОРС автоматически определяются параллельно выполняемые циклы для различных вычислительных архитектур. Для определения параллельно выполняемых циклов следует щелкнуть мышью по кнопке «маркер» под левым верхним окном на экране ОРС.

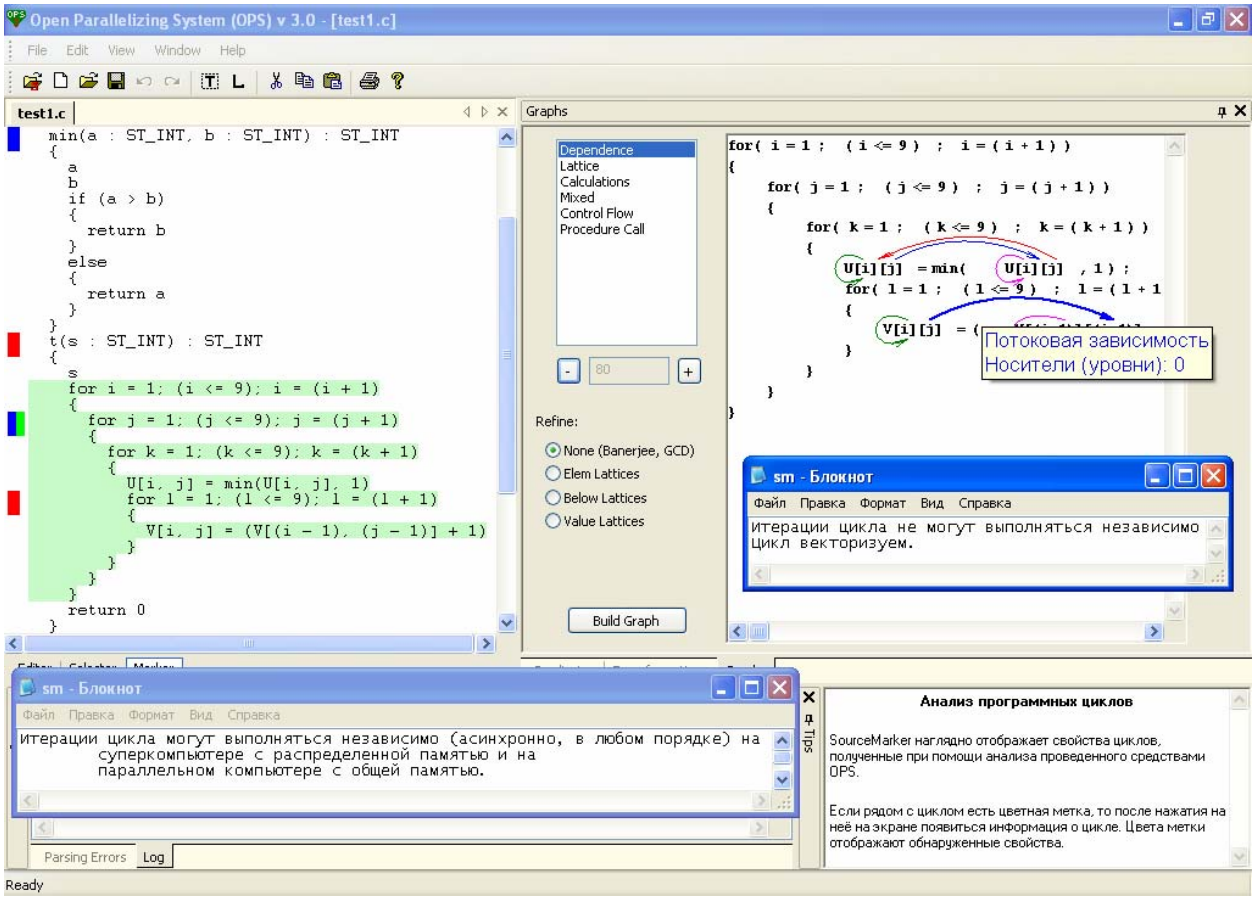


Рис. 3. Отображение свойств циклов и функций в OPC. Правое информационное окно сообщает о векторизуемости самого вложенного цикла. Нижнее информационное окно говорит о возможности независимого выполнения итераций второго цикла сверху. Синяя метка возле описания функции min() говорит о том, что эта функция не имеет побочных эффектов, которые могли бы препятствовать распараллеливанию циклов, содержащих вызов этой функции.

Одна из важных характеристик вычислительной архитектуры – общая (shared) или распределенная (distributed) память. Общая память допускает входные зависимости. Входная зависимость означает, что разные итерации цикла используют одни и те же данные. При распределенной памяти входная зависимость мешает, но может быть устранена предварительной рассылкой данных.

Распараллеливанием цикла называется замена цикла

```
      DO 99 I=L,N,h
99    ТЕЛОЦИКЛА(I)
```

циклом

```
      DO ALL  99  I=L,N,h
99    ТЕЛОЦИКЛА(I)
```

Здесь оператор параллельного цикла DO ALL означает инструкцию для параллельного компьютера, требующую одновременного выполнения ТЕЛОЦИКЛА(I) для всех значений счетчика I, указанных в заголовке цикла (предполагается, что для каждого значения счетчика цикла I ТЕЛОЦИКЛА(I) будет выполняться в своем процессорном элементе).
В общем случае ТЕЛОЦИКЛА(I) может содержать любые операторы, в том числе и другие циклы. Семантика оператора DO ALL зависит от вычислительной архитектуры. Иногда вместо DO ALL оператор параллельно выполняемого цикла обозначают DO SIM (SIMULTANIOUSLY), DO PAR, ParDO, DO Concurrently.

Распараллеливание, как преобразование, состоящее в замене заголовка последовательного цикла заголовком параллельного, тривиально. Вопрос в том, когда такое преобразование эквивалентно и в том, чтобы вспомогательными преобразованиями привести цикл к виду, в котором распараллеливание является эквивалентным преобразованием.

ВЕКТОРИЗАЦИЯ ЦИКЛОВ

Векторные компьютеры позволяют эффективно выполнять операции над векторами: найти сумму двух векторов, произведение вектора на число, и др. Наиболее известный векторный компьютер Cray-1 много лет занимал лидирующее место по производительности среди суперкомпьютеров. Этот компьютер обрабатывал векторы длины 64. В СССР тоже разрабатывались векторные суперкомпьютеры ПС-3000 и др. Для Cray-1 и для ПС-3000 разрабатывались распараллеливающие компиляторы.

Последовательные циклы, допускающие замену на векторную команду, называются векторизуемыми, а сама такая замена – векторизацией. Векторизуются циклы, которые содержат только операторы присваивания. Векторизация является эквивалентным преобразованием при дополнительных ограничениях на информационные зависимости в цикле. Для описания этих ограничений понадобятся дополнительные понятия.

Будем говорить, что вхождение V1 расположено в программе раньше вхождения V2, а вхождение V2 расположено позже вхождения V1, если при любом исполнении программы обращение к вхождению V1 происходит по времени раньше, чем обращение к вхождению V2.

Если вхождения V1 и V2 принадлежат операторам одного и того же линейного участка программы, то V1 расположено раньше V2 тогда и только тогда, когда выполняется одно из условий: 1) V1 принадлежит оператору с меньшим номером, чем V2, 2) V1 и V2 принадлежат одному и тому же оператору присваивания, причем V2 является генератором, а V1 - использованием.

Лемма 3. Пусть в теле цикла все операторы являются операторами присваивания и вхождения V1 и V2 обращаются к одной и той же ячейке памяти. Пусть V1 расположено в тексте программы раньше, чем V2. Тогда после векторизации вхождение V1 будет обращаться к памяти раньше, чем V2.

Доказательство вытекает из того, что после векторизации сначала к памяти будет обращаться вхождение V1 для всех значений счетчика цикла, а спустя некоторое время V2 (тоже для всех значений счетчика цикла).

Пример 106**.**

```
      DO 111  I  =  1,  N
      A(I)  =  B(I)+C(I)
      D(I)  =     A(I+1)/2.
 111  CONTINUE
```

В этом цикле вхождение A(I+1) обращается к ячейке памяти A(2) раньше, чем вхождение A(I), но после распараллеливания будет наоборот.

```
      DO ALL 111 I = 1, N
      A(I) = B(I)+C(I)
      D(I) = A(I+1)/2.
111   CONTINUE
```

Конец примера.

Теорема 33. Пусть в теле цикла все операторы являются операторами присваивания. Пусть в графе информационных связей тела цикла все дуги направлены от вхождений, которые встречаются раньше к вхождениям, которые в тексте расположены позже. Тогда векторизация данного цикла является эквивалентным преобразованием.

Доказательство. При векторизации изменяется только последовательность исполнения операций. Поэтому эквивалентность может быть нарушена только в том случае, когда при векторизации меняется последовательность обращения различных вхождений к одной и той же ячейке памяти. Если два вхождения обращаются к одной и той же ячейке памяти, то в графе информационных связей они соединяются дугой. Поскольку в теле цикла дуги графа информационных связей направлены от вхождений, которые расположены раньше к вхождениям, расположенным позже, порядок обращения вхождений к памяти при векторизации не меняется.

Следствие. Если цикл векторизуется, то он может быть разбит на циклы, в каждом из которых по одному оператору присваивания.

Пример 107**.** Следующий цикл имеет рекуррентную зависимость и, поэтому не векторизуется.

```
      DO 111  I  =  1,  N
      X(I)  =  A(I)*X(I-1)+B(I)
 111  CONTINUE
```

Конец примера.

Может случиться, что цикл непосредственно не векторизуется, а после некоторых вспомогательных преобразований станет векторизуемым. Это, как

правило, те же вспомогательные преобразования, которые используются при разрезании цикла. Рассмотрим такие примеры.

Пример 108**.** Следующий цикл имеет дуги зависимости, направленные снизу вверх и, поэтому не векторизуется. Но после перестановки операторов местами цикл становиться векторизуемым.

```
      DO 111  I  =  1,  N
      X(I) =  A(I)*Y(I-1)+B(I)
      Y(I) = D(I)/X(I+1)
  111 CONTINUE
```

Конец примера.

Пример 109**.** Следующий цикл содержит вхождения переменной X, которая не зависит от счетчика цикла. Эти два вхождения связаны двумя дугами, направленными в противоположные стороны.

```
      DO 111  I  =  1,  N
      X =  A(I)*Y(I)+B(I)
      Y(I) = D(I)/X
  111 CONTINUE
```

Преобразование «растягивание скаляров» может удалить одну из этих дуг (антизависимость), после чего цикл станет векторизуемым.

```
      DO 111  I  =  1,  N
      XX(I) =  A(I)*Y(I)+B(I)
      Y(I) = D(I)/XX(I)
  111 CONTINUE
      X = XX(N)
```

Конец примера.

Пример 110**.** Следующий цикл имеет две дуги антизависимости, одна из которых направлена снизу вверх, а другая, наоборот, сверху вниз. По этой причине цикл не векторизуется и перестановка операторов местами не приводит к векторизуемости.

```
      DO 111  I  =  1,  N
      X(I) =  A(I)*Y(I+1)+B(I)
      Y(I) = D(I)/X(I+1)
  111 CONTINUE
```

Введение временного массива приводит данный цикл к векторизуемому виду.

```
      DO 111  I  =  1,  N
      TEMP(I) = X(I+1)
      X(I) =  A(I)*Y(I+1)+B(I)
      Y(I) = D(I)/TEMP(I)
111   CONTINUE
```

Конец примера.

АСИНХРОННОЕ ПАРАЛЛЕЛЬНОЕ ВЫПОЛНЕНИЕ БЛОКОВ

Рассмотрим препятствия, которые могут возникнуть при преобразовании блоков в параллельные процессы.

Распараллеливание блоков является более общим преобразованием по отношению к распараллеливанию циклов. Действительно, при распараллеливании циклов параллельно выполняются итерации циклов. Но итерации циклов можно рассматривать, как блоки.

При распараллеливании блоков возникает задача выделения в программе тех блоков, которые могли бы выполняться параллельно. Для повышения параллельности можно рассматривать задачу разбиения больших программных блоков на более мелкие блоки. С другой стороны, чтобы не тратить время на частые инициализации параллельных процессов, можно некоторые мелкие блоки склеивать в более крупные. Чтобы параллельно работающие процессоры были равномерно загружены, можно решать задачу загрузочного баланса (load balancing).

Для преобразования в процесс блок должен иметь один вход. Чтобы два блока могли параллельно выполняться, они должны быть независимы по управлению. Например, один блок не может быть вложен в другой, не должно быть перехода (условного или безусловного) из одного блока в другой. Для того чтобы два блока могли выполняться параллельно, необходимо, чтобы они в программе были связаны по управлению конкатенацией.

Если два блока не связаны конкатенацией, то следует попытаться их переместить так, чтобы они оказались в конкатенации.

Пример 111.

```
for(i=1; i <= N; ++i)
{
 a = a+b[i]*c[i]
}
If eps < 0 then
for(j=1; j <= M; ++j)
{
  x = x+e[j]*d[j];
}
else
 x = a
```

В этом примере фрагмент кода представляет собой конкатенацию двух блоков: первого оператора цикла и условного оператора. Но эти блоки нельзя выполнять параллельно, поскольку они связаны информационной зависимостью по переменной x. Но второй цикл можно вынести из условного оператора и тогда он окажется в конкатенации с первым, и оба этих цикла можно будет вычислять независимо. После независимого вычисления циклов может быть вычислен и облегченный условный оператор.

```
for(i=1; i <= N; ++i)
{
 a = a+b[i]*c[i]
}
for(j=1; j <= M; ++j)
{
  x = x+e[j]*d[j];
}
If eps < 0 then
{
}
else
 x = a
```

Конец примера.

Граф информационных связей может установить зависимости по данным между вхождениями переменных в блоки.

Теорема 34. Пусть задано гнездо циклов вида

```
for(j1=1; j1 <= M1; ++j1)
for(j2=1; j2 <= M2; ++j2)
      …
for(jn=1; jn <= Mn; ++jn)
{
  B1(j1,j2,…,jn)
  B2(j1,j2,…,jn)
}
```

Предположим, что каждый из блоков B1 и B2 имеет один вход и один выход (в частности, между этими блоками нет условных или безусловных переходов) и между этими блоками нет информационных зависимостей с носителем jn. Тогда асинхронное вычисление этих блоков эквивалентно их последовательному вычислению.

Доказательство. Действительно, при асинхронном выполнении эти блоки не обращаются к одним и тем же ячейкам памяти.

Замечание. Можно допустить между блоками B1 и B2 входные циклически независимые зависимости с носителем jn. При наличии общей памяти это никак не повлияет на результат вычислений. При наличии распределенной памяти входные зависимости предполагают предварительные пересылки данных.

Теорема 35. Предположим, что в программе оператор S1 является доминатором оператора S2. Тогда замена оператора S2 блоком {Wait(S1), S2} является эквивалентным преобразованием.

Доказательство. Ясно, что в преобразованной программе S1 является доминатором блока {Wait(S1), S2}. Поэтому, всякий раз, когда при исполнении программы этому блоку будет передаваться управление, оператор S1 уже будет выполнен. Следовательно, оператор Wait(S1) можно заменить пустым оператором и удалить.

Теорема 36. Предположим, что программа F представляет собой конкатенацию фрагментов (блоков) $F = \{ F_1 , F_2 , \dots , F_r \}$ и внутри каждого фрагмента нет операторов передачи управления за пределы этого фрагмента. Пусть G – граф зависимости по данным программы F. Построим новую программу F_new, состоящую из новых фрагментов F_i_new, I = 1, 2, …, r, которая будет выполняться параллельно. Для этого, для каждой дуги (S1, S2) графа зависимостей по данным G, такой, что S1 и S2 принадлежат разным фрагментам F_i и F_j , заменим оператор S2 на блок из двух операторов {Wait(S1), S2}. Тогда параллельное асинхронное выполнение фрагментов программы F_new = { F_1_new, F_2_new, … , F_r_new } эквивалентно последовательному выполнению программы F.

Доказательство. Сначала заметим, что последовательное выполнение программы F равносильно последовательному выполнению программы F_new. Отметим, что операторы Wait() не меняют состояние памяти; эти операторы могут только привести к изменению времени исполнения программы, в частности, к тому, что программа не завершится. Теперь отметим, что всякий оператор S2 не выполнится раньше, чем любой такой оператор S1, из которого в S2 направлена дуга графа зависимостей по данным. Но тогда выполняются условия теоремы о перестановке семейства операторов, из которой и вытекает эквивалентность фрагментов F_new и F.

Пример 112. В следующем гнезде циклов между двумя блоками не имеется зависимостей с носителем – самым вложенным циклом. Эти блоки могут выполняться асинхронно.

```
for(i=1; i <= N; ++i)
for(j=1; j <= M; ++j)
{
 {
   X[j] = Y[i-1,j]*Z[i-1]
 }
```

```
 {
   Y[i,j] = Y[i,j-1]*D[j];
   Z[i] = 1
 }
}
```

Конец примера.

АСИНХРОННОЕ РАСПАРАЛЛЕЛИВАНИЕ ЦИКЛОВ С НЕЗАВИСИМЫМИ ИТЕРАЦИЯМИ

Задача асинхронного распараллеливания циклов сводится к задаче асинхронного распараллеливания блоков. Действительно, заменим цикл его раскруткой. Каждую итерацию исходного цикла будем рассматривать, как фрагмент (блок) программы, полученной в результате раскрутки. Итерации исходного цикла можно выполнять параллельно тогда и только тогда, когда параллельно могут выполняться соответствующие блоки результирующей программы.

При переходе от цикла к его раскрутке циклически независимые зависимости останутся внутри блоков, соответствующих итерациям исходного цикла, а циклически порожденные зависимости превратятся в зависимости между блоками.

Асинхронная архитектура (многоядерные процессоры, кластеры, мультитранспьютерные системы) позволяет распараллеливать циклы любой глубины вложенности. Эти циклы могут содержать условные операторы. Но в этих циклах допускаются только циклически независимые зависимости.

Если память компьютера общая, то допускаются и входные циклически порожденные зависимости. Это циклы, итерации которых могут выполняться независимо друг от друга без взаимных синхронизаций.

Всякое удаление циклически порожденных зависимостей либо приводит цикл к виду, допускающему параллельное выполнение, либо уменьшает количество необходимых синхронизаций при параллельном выполнении итераций.

Пример 113. В данном примере имеются циклически порожденные зависимости относительно внешнего цикла.

```
for(i=1; i <= N; ++i)
for(j=1; j <= M; ++j)
{
   a[j] = b[i,j]*c[i,j]
   x[i,j] = a[j]*d[i,j];
}
```

Повышение размерности массивов удаляет циклически порожденные зависимости и приводит внешний цикл к параллельно выполняемому виду.

```
for(i=1; i <= N; ++i)
for(j=1; j <= M; ++j)
{
   aa[i,j] = b[i,j]*c[i,j]
   x[i,j]  = aa[i,j]*d[i,j];
}
for(j=1; j <= M; ++j)
{
   a[j] = aa[N,j];
}
```

Конец примера.

Следует отметить, что повышение размерности массивов приводит к дополнительному расходу памяти. Эти расходы может сэкономить предварительно проведенное гнездование цикла.

Пример 114. Предположим, что вычислительная архитектура имеет p асинхронно работающих вычислительных устройств и пусть N = p*R. Рассмотрим фрагмент кода из предыдущего примера

```
for(i=1; i <= N; ++i)
for(j=1; j <= M; ++j)
{
   a[j] = a[j]+b[i,j]*c[i,j]
   x[i,j] = a[j]*d[i,j];
}
```

Выполнив гнездование внешнего цикла, получим следующий фрагмент

```
for(k=1; k <= p; ++k)
for(l=1; l <= R; ++l)
for(j=1; j <= M; ++j)
{
   i = (k-1)*R+l
   a[j] = a[j]+b[i,j]*c[i,j]
   x[i,j] = a[j]*d[i,j];
}
```

Теперь к внешнему циклу можно применить повышение размерности массивов

```
for(k=1; k <= p; ++k)
for(l=1; l <= R; ++l)
for(j=1; j <= M; ++j)
```

```
  {
     i = (k-1)*R+l
     aa[k,j] = aa[k,j]+b[i,j]*c[i,j]
     x[i,j] = aa[k,j]*d[i,j];
  }
```

В данном примере, в отличие от предыдущего, первая координата массива aa имеет длину p, а не N.

Конец примера.

АСИНХРОННОЕ РАСПАРАЛЛЕЛИВАНИЕ ЦИКЛОВ С ЗАВИСИМЫМИ ИТЕРАЦИЯМИ

Для определения условий синхронизации при параллельном выполнении гнезда циклов может быть полезен решетчатый граф программы (граф алгоритма). Решетчатый граф содержит информацию о зависимости между переменными на разных итерациях гнезда циклов. Вершинами решетчатого графа являются точки пространства итераций гнезда циклов. Каждая дуга (u,v) соединяет вершины u и v, если результат вычислений на итерации u используется на итерации v.

Пример 115.

```
{
  for(i=1; i <= N; ++i)
  for(j=1; j <= N; ++j)
  {
     a[i+j] = …
        …   = … a[i+j-1];
  }
}
```

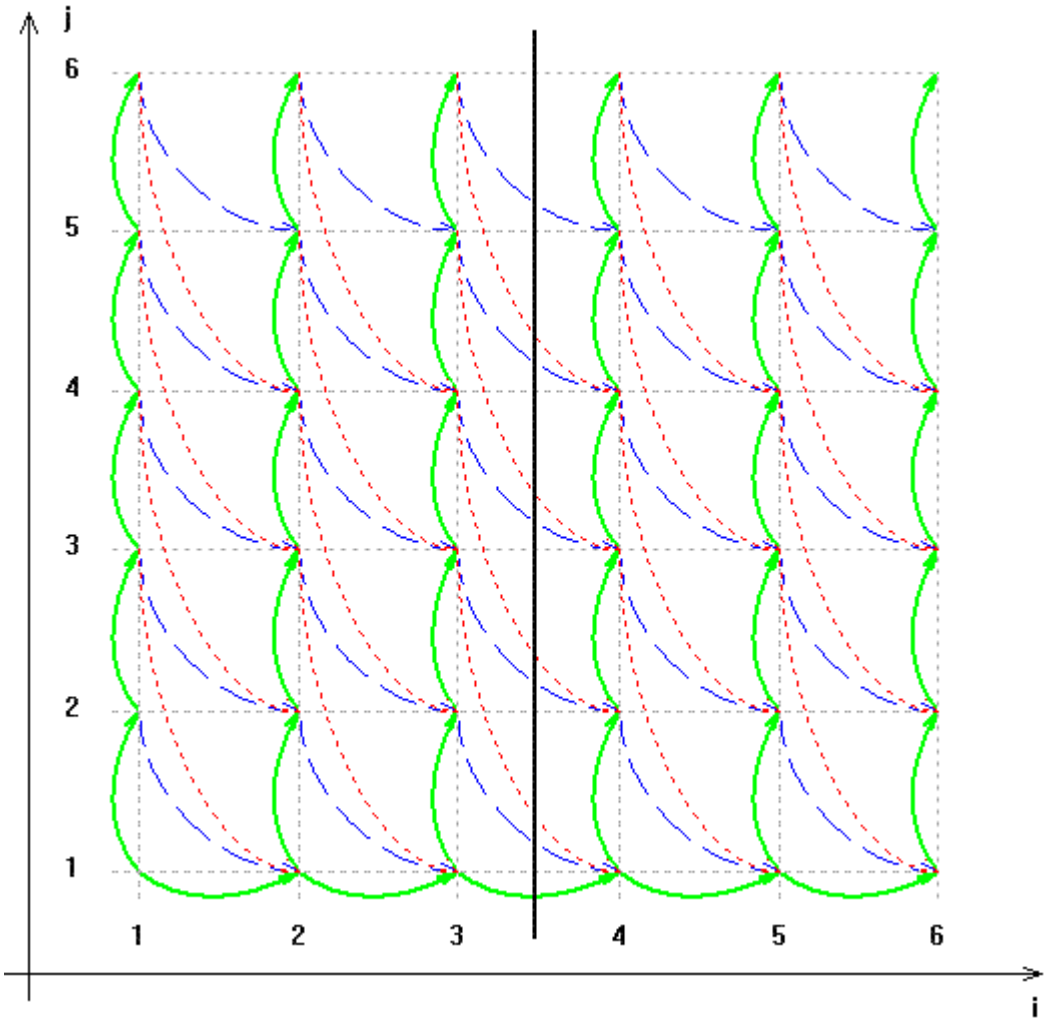


Рис. 4. Решетчатый граф, построенный с помощью OPC. Зеленые дуги – истинная зависимость, красные – антизависимость, синие – выходная самозависимость.

Конец примера.

Пример 116. В следующем гнезде циклов переменная a не зависит от счетчиков обоих циклов.

```
{
  for(i=1; i <= M; ++i)
  for(j=1; j <= N; ++j)
  {
     …        = … a;
     a = …
  }
}
```

Первое повышение размерности массивов способствует распараллеливанию с синхронизациями внутреннего цикла.

```
{
  for(i=1; i <= M; ++i)
 {
 aa[0] = a
  for(j=1; j <= N; ++j)
  {
   …        = … aa[j-1];
      aa[j] = …
  }
 a = aa[N]
 }
}
```

Второе повышение размерности массивов способствует распараллеливанию с синхронизациями внешнего цикла.

```
{
aaa[0] = a
  for(i=1; i <= M; ++i)
 {
 aa[0] = aaa[i-1]
  for(j=1; j <= N; ++j)
  {
             = … aa[j-1];
     aa[j] = …
  }
 aaa[i] = aa[N]
 }
a = aaa[M]
}
```

Конец примера.

Гнездование цикла, как и в случае с распараллеливанием циклов с независимыми итерациями, позволяет уменьшить расход памяти, связанный с повышением размерности массивов.

РАСПАРАЛЛЕЛИВАНИЕ НА АРХИТЕКТУРУ VLIW

Аббревиатура VLIW означает Very Long Instruction Word – очень длинное командное слово. Вычислительная архитектура VLIW содержит несколько одновременно работающих элементарных вычислительных устройств. Каждое из этих устройств может выполнять какую-нибудь ассемблерную операцию (выравнивание вещественных чисел с плавающей запятой, сложение выровнен-

ных вещественных чисел, умножение целых чисел и т. д.). Примером компьютера с такой архитектурой является суперкомпьютер «Эльбрус 3».

Хотя речь идет об операциях низкоуровневых (ассемблерных), для иллюстрации принципов программирования будем рассматривать примеры высокоуровневых фрагментов программ. В следующих примерах будем предполагать, что вычислительная архитектура позволяет одновременно выполнять одну операцию «+» и одну «*».

Пример 117. Рассмотрим следующий цикл

```
      DO 99   I = 1,  N
      X(I) = A(I)+B(I)
      A(I) = C(I)+D(I).
      Y(I) = X(I)*Z(I)
      V(I) = A(I)*U(I).
      W(I) = C(I)*D(I)
      B(I) = Z(I)+D(I).
 99   CONTINUE
```

Переставив пятый оператор присваивания на второе место, можно все операции тела цикла сгруппировать парами оптимальным образом.

```
      DO 99   I = 1,  N
      X(I) = A(I)+B(I) ;    W(I) = C(I)*D(I)
      A(I) = C(I)+D(I);     Y(I) = X(I)*Z(I)
      V(I) = A(I)*U(I);      B(I) = Z(I)+D(I).
 99   CONTINUE
```

Конец примера.

Пример 118. В данном примере рекуррентность не позволяет одновременно выполнять оба оператора тела цикла.

```
      DO 99   I = 1,  N
      X(I) = A(I)+B(I)
      Y(I) = X(I)*D(I).
 99   CONTINUE
```

Но развертка может привести цикл к удобному для распараллеливания на архитектуре VLIW виду

```
      DO 99   I = 1, N, 3
1       X(I) = A(I)+B(I)
2       Y(I) = X(I)*D(I).
3       X(I+1) = A(I+1)+B(I+1)
4       Y(I+1) = X(I+1)*D(I+1).
5       X(I+2) = A(I+2)+B(I+2)
6       Y(I+2) = X(I+2)*D(I+2).
 99   CONTINUE
      DO 999   I = (N//3)*3+1, N
      X(I) = A(I)+B(I)
      Y(I) = X(I)*D(I).
 999   CONTINUE
```

В полученном фрагменте программы можно одновременно выполнять операторы 2 и 3, а также, 4 и 5. Операторы 1 и 6 следует выполнять отдельно по одному. Ясно, что при большей кратности развертки эффект распараллеливания будет больше.

Конец примера.

Пример 119. В данном примере в теле каждого из циклов операции однотипные и это не позволяет их выполнять одновременно.

```
      DO 77  I = 1, N
      X(I) = A(I)+B(I)
      Y(I) = C(I)-D(I).
 77   CONTINUE
      DO 99  I = 1, N
      Z(I) = A(I)*B(I)
      T(I) = X(I)*Z(I).
 99   CONTINUE
```

Но после слияния циклов и перестановки операторов параллельность выполнения возможна

```
      DO 77  I = 1, N
      X(I) = A(I)+B(I);
       Z(I) = A(I)*B(I)
      Y(I) = C(I)-D(I);
       T(I) = X(I)*Z(I).
 99   CONTINUE
```

Конец примера.

Итак, основные оптимизирующие преобразования для программирования архитектуры VLIW, это перестановки операторов и развертка цикла. О параллельном выполнении программ для архитектуры VLIW говорят, что в этих программах параллельно выполняются операции и распараллеливаются линейные участки (а не циклы, функции, микрокоманды и пр.). Архитектура VLIW допускает эффективное автоматическое и распараллеливающие компиляторы для компьютеров такой архитектуры были разработаны.

РАСПАРАЛЛЕЛИВАНИЕ НА АРХИТЕКТУРУ SIMD

Архитектура SIMD (Single Instruction Multiple Data) одновременно выполняет одинаковые команды над разными данными. Векторные компьютеры можно считать частным случаем архитектуры SIMD. Кроме того, к этой архитектуре относятся и такие известные суперкомпьютеры с распределенной памятью, как американские ILLIAK-4 (64 процессора), Connection Machine2 (64000 процессоров) и советская ПС-2000 (64 процессора). Такие вычислительные системы синхронны и позволяют параллельно выполнять программы, состоящие из циклов и операторов присваивания, и допускают потоковые, анти- и выход-

ные зависимости, которые на графе информационных связей направлены сверху-вниз (от вхождений, которые раньше обращаются к памяти, к вхождениям, которые позже обращаются к памяти). При этом, если векторные компьютеры распараллеливают только самые глубоко вложенные циклы, то SIMD компьютеры с распределенной памятью допускают гнезда вложенных циклов.

Пример 120. В данном гнезде циклов все дуги зависимости направлены от вхождений, которые раньше встречаются к вхождениям, которые встречаются в программе позже. Поэтому данное гнездо циклов допускает параллельное выполнение на архитектуре SIMD.

```
      DO 99  I = 1, N
      X(I) = C(I)+B(I);
      DO 99  J = 1, N
      A(J) = A(J) + D(I,J)*X(I);
 99   CONTINUE
```

Конец примера.

В суперкомпьютерах типа SIMD все не отключенные процессоры в один момент времени выполняют одну и ту же операцию. Если на каком-то этапе работы алгоритма необходимо в разных процессорах выполнить разные операции, то процессоры активируются по очереди теми группами, которые должны выполнять одинаковые операции. Для таких суперкомпьютеров распараллеливаются самые вложенные циклы, причем количество итераций этих циклов не должно зависеть от вычислений в теле цикла (как бывает в итерационных процессах). Здесь очень удобны циклы DO языка ФОРТРАН или циклы FOR языка ПАСКАЛЬ. Частным случаем таких суперкомпьютеров иногда являются векторные суперкомпьютеры, обрабатывающие векторы фиксированной длины.

При распределенной памяти компьютеров типа SIMD возникает задача размещения данных с минимизацией межпроцессорных пересылок. При этом циклически порожденные истинные зависимости (направленные сверху вниз) порождают пересылку данных, которая должна быть сделана во время выполнения цикла. Входная циклически порожденная зависимость требует пересылку данных, которая должна быть сделана до выполнения цикла.

Пример 121. Предположим, что в вычислительной системе имеется P процессорных элементов, и пусть все массивы размещены так, что I-ый элемент массива находится в модуле памяти с номером (I mod P).

```
      DO 99  I = 1, N
      X(I) = A(I)+B(I);
      A(I) = C(I+1)+X(I);
      A(I-1) = X(I-1)*B(I+3);
 99   CONTINUE
```

В приведенном цикле входная зависимость между вхождениями переменной B предполагает пересылку до выполнения цикла; истинная зависимость (X(I), X(I-1)) требует выполнение пересылки после первого оператора перед третьим. Антизависимости и выходная зависимость не влияют на параллельное выполнение – новые элементы массивов окажутся сдвинутыми относительно старых элементов.

Конец примера.

Основные распараллеливающие преобразования, как и при векторизации: перестановка операторов, повышение размерности массивов, введение временных массивов.

КОНВЕЙЕРНЫЕ ВЫЧИСЛЕНИЯ

В данной главе рассматривается задача автоматического распараллеливания циклов для суперкомпьютера с архитектурой перестраиваемого конвейера. Конвейерные архитектуры описываются в [**29**]. Примеры архитектуры перестраиваемого конвейера описаны в [50], [32]. Такая архитектура оказывается удобной для эффективного выполнения широкого класса задач с регулярными зависимостями по данным, в том числе и для информационно сильно связных. Получены алгоритмы, которые по тексту распараллеливаемого цикла формируют граф соединения сегментов памяти и процессоров в конвейер для вычисления этого цикла. Отображаемые циклы могут быть рекуррентными. Некоторые ограничения часто могут быть преодолены с помощью преобразований программ. Примерная схема компьютера с архитектурой перестраиваемого конвейера (со структурно-процедурной организацией вычислений) [32] изображена на следующем рисунке.

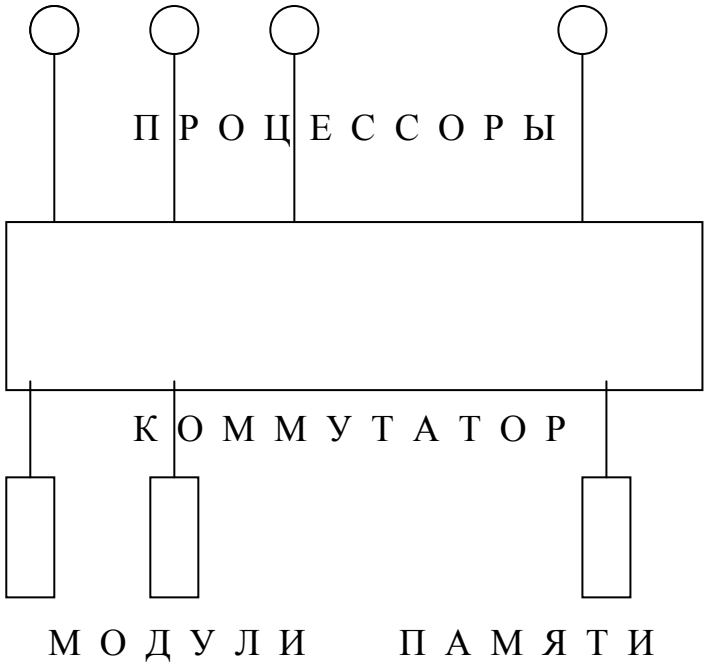


Рис. 5. Схема коммутации процессоров и модулей памяти.

Каждый процессор имеет на входе буфер. Этот буфер позволяет задерживать поступление данных в процессор. Такие действия бывают важными при синхронизации конвейерного вычисления.

Допускается в один момент времени обращаться к разным секторам памяти (для чтения или записи).

Из каждого сектора памяти и из каждого процессора есть выход, ведущий на вход коммутатора. К каждому сектору памяти и к каждому входу каждого процессора ведет какой-нибудь выход коммутатора. Будем считать, что коммутатор позволяет соединять входы и выходы вычислительной системы в любых необходимых комбинациях, допуская соединение одного входа с несколькими выходами. Рассмотренная вычислительная система позволяет соединять элементарные процессоры в конвейеры различных конфигураций и подключать к ним сегменты памяти с данными. Время перенастройки коммутатора существенно больше единицы времени, так что количество перенастроек следует минимизировать.

Отметим, что близкой к данной модели (по доступу к параллельной памяти) является архитектура суперкомпьютера RP-3 фирмы IBM [**49**].

ГРАФ ВЫЧИСЛЕНИЙ ПРОГРАММЫ

Определим граф вычислений программы следующим образом. В качестве вершин графа вычислений программы будем рассматривать операции программы. Операции будем рассматривать арифметические, булевы, а также операции считывания из памяти и записи в память. Дуга соединяет пару операций, если вторая операция использует результат вычисления первой операции. Кроме упомянутых операций следует рассматривать также k-арную операцию нахождения элемента k-мерного массива по его индексам. Но в дальнейшем эту операцию будем, как правило, опускать, чтобы излишней детализацией не скрыть основные идеи.

Ясно, что операции считывания из памяти данного, обозначенного безиндексной переменной, соответствует источник на графе (т. е. такая вершина, в которую не входит ни одна дуга), а операции записи в память соответствует сток (не выходит ни одна дуга).

Пример 122. Рассмотрим оператор присваивания:

X = (X*A(I)+B(I))+C(I)

Этому оператору соответствует дерево вычислений.

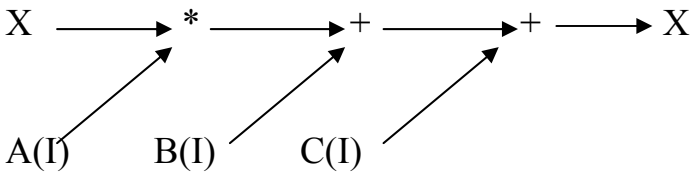


Рис. 6. Дерево вычислений

Отметим, что в общем случае граф вычислений представляет собой ориентированный лес.

Если вершинам графа вычислений сопоставить вычислительные устройства, выполняющие соответствующие операции, а дугам - связи между устройствами, то полученная вычислительная система будет вычислять ту программу, по которой был построен граф. Проблема в том, что вершин у графа вычислений, как правило, значительно больше, чем имеющихся вычислительных устройств. В этой ситуации приходится разбивать граф на такие части, для каждой из которых в системе вычислительных ресурсов достаточно. Разбиение должно быть таким, чтобы вычисление программы выполнялось как можно быстрее. Об этой задаче будет идти речь далее.

Рассмотрим теперь задачу построения графа для программного цикла.

```
      DO 99 I=1, N
99    ТЕЛОЦИКЛА(I)                                    (16)
```

Раскрутка цикла [**8**] - это замена предыдущего цикла на следующий эквивалентный фрагмент программы:

```
ТЕЛОЦИКЛА(1)
ТЕЛОЦИКЛА(2)
.....
ТЕЛОЦИКЛА(N)
```

Граф вычислений раскрутки цикла представляет собой объединение N изоморфных друг другу подграфов.

Граф вычислений раскрутки цикла слишком громоздкий для практической обработки. По этой причине далее будет строиться другой граф – называемый свернутым графом, который имеет значительно меньшие размеры (как у одной итерации тела цикла) и содержит в себе всю информацию о графе вычислений раскрутки цикла.

ГРАФ ВЫЧИСЛЕНИЙ И ИНФОРМАЦИОННЫЕ ЗАВИСИМОСТИ

Наличие информационной зависимости означает, что какие-то вхождения должны обращаться к одной и той же ячейке памяти. Это не может не сказаться на алгоритме отображения цикла на архитектуру суперкомпьютера.

Далее будут рассматриваться только такие циклы, в которых для любой пары вхождений одной и той же переменной МОЖНО ОПРЕДЕЛИТЬ (имеющимися средствами) наличие или отсутствие информационной зависимости. В частности, будем полагать, что индексные выражения переменных линейно зависят от счетчиков объемлющих циклов.

Далее будем рассматривать циклы, которые не имеют нерегулярной информационной зависимости. Это ограничение необходимо для конвейеризации рекуррентного цикла и для возможности эффективного размещения данных при зависимости in-in [**43**].

Out-out зависимости означают запись в одну и ту же ячейку памяти. Для конвейеризации цикла эта зависимость должна быть исключена какими-либо преобразованиями или операции записи в одну ячейку памяти должны быть разведены во времени.

Out-in зависимости означают, что данное сначала записывается, а затем читается. Этот факт должен учитываться при конвейеризации.

In-in зависимости означают чтение из одной и той же ячейки памяти. Некоторые коммуникационные устройства позволяют одно считанное данное одновременно разослать в несколько, разных процессоров.

In-out – это дуга антизависимости, означающая, что из какой-то ячейки памяти сначала читается одно данное, а, затем, в эту ячейку записывается другое данное. Лучше всего от таких дуг избавиться известными предварительными преобразованиями.

Далее будем предполагать, что дуг антизависимости в цикле нет и коммутатор суперкомпьютера позволяет одно данное одновременно разослать в несколько устройств (процессоров или секторов памяти).

Циклически независимое OUT-IN склеивание. Рассмотрим граф вычислений линейного участка программы. Предположим, что на этом участке есть два вхождения переменной, связанные циклически независимой out-in зависимостью. Отождествим такую пару вершин, что первая вершина вычисляет данное, а вторая читает это данное из памяти, как это сделано в следующем примере.

Пример 123.

X = Y - A

Z = X + B

Граф вычислений этой программы состоит из двух компонент связности и имеет следующий вид:

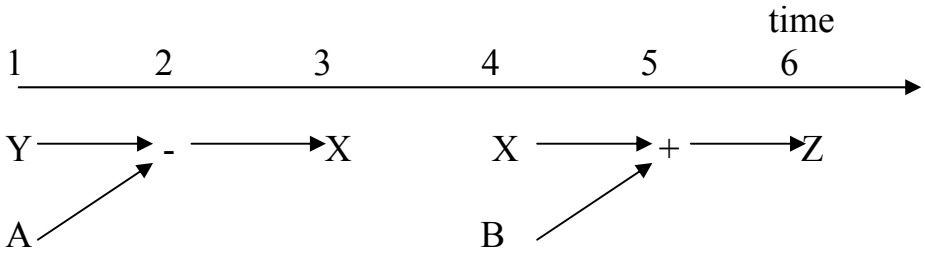


Рис. 7. Два дерева для блока из двух операторов с циклически независимой зависимостью выхода первого и входа второго.

Пока первый оператор не запишет в память X, второй оператор не может считывать X из памяти. Операция вычитания имеет такой же результат, как и чте-

ние из памяти переменной X. Поэтому можно результат вычитания сразу пересылать в суммирующий процессор:

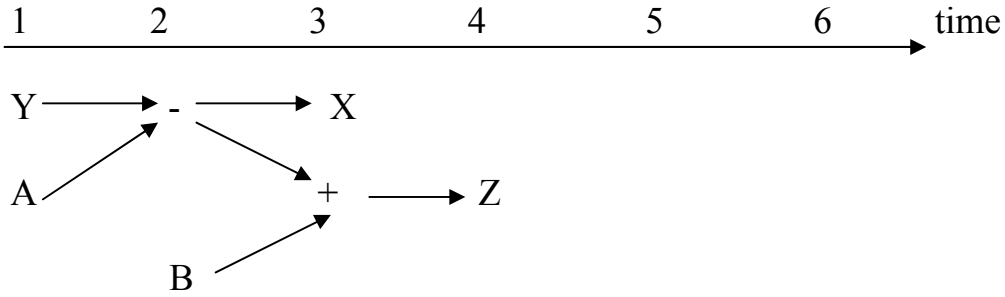


Рис. 8. Результат склейки двух деревьев.

Полученный граф вычислений требует меньше времени для выполнения, чем исходный.

Циклически порожденное OUT-IN склеивание. Рассмотрим граф вычислений линейного участка программы. Предположим, что на этом участке есть два вхождения переменной, связанные циклически порожденной out-in зависимостью. Отождествим такую пару вершин, что первая вершина вычисляет данное, а вторая читает это данное из памяти, как это сделано в следующем примере.

Пример 124.

```
      DO 99  I=3, N
      X(I) = Y(I) - A
      Z(I) = X(I-2) + B
99 CONTINUE
```

Граф вычислений этой программы состоит из двух компонент связности и имеет следующий вид:

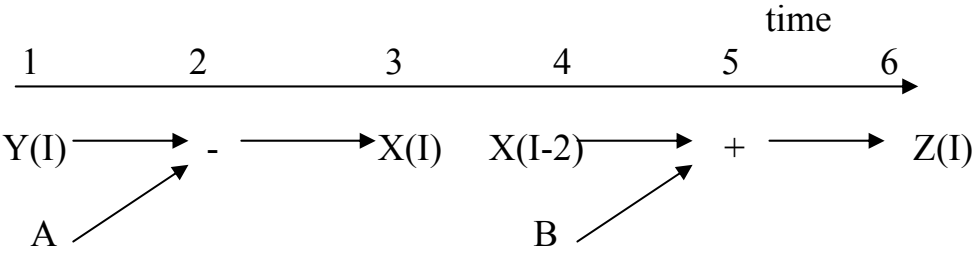


Рис. 9. Два дерева с циклически порожденной зависимостью выхода первого и входа второго.

Пока первый оператор не запишет в память X(k), второй оператор не может считывать X(k) из памяти (но через две итерации цикла). Операция вычитания в этом примере имеет такой же результат, как операция чтения из памяти переменной X. Поэтому можно результат вычитания сразу пересылать в буфер сум-

мирующего процессора. При этом получим следующий граф, в котором вес новой дуги, равен разности итераций:

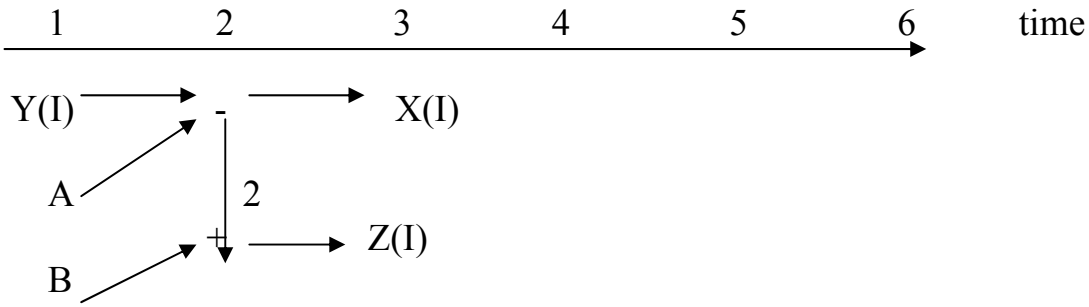


Рис. 10. Результат склейки с циклически порожденной зависимостью.

Временная функция должна доопределяется с графа вычислений на такой склеенный граф с учетом разности в итерациях.

Пример 125.

```
      DO 99 I=3, N
      X(I) = A(I)+B(I)
99    Z(I) = Y(I)+Z(I-1)*X(I-2)
```

Свернутый граф этого цикла будет иметь следующий вид:

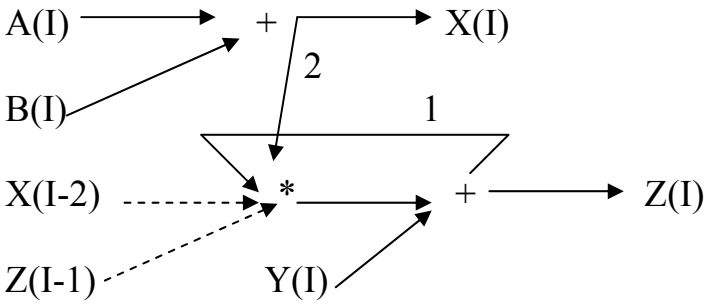


Рис. 11. Свернутый граф цикла.

Здесь пунктирные дуги обозначают начальную загрузку процессора.

Из приведенного примера видно, что, если циклически порожденная out-in зависимость связана с рекуррентными вычислениями, то ей соответствует дуга свернутого графа, создающая контур, а если эта зависимость не связана с рекуррентными вычислениями, то контура в графе может и не быть.

Циклически независимое IN-IN склеивание. В этом параграфе также будем использовать тот факт, что коммутатор суперкомпьютера позволяет взять данное из одного сектора памяти и одновременно разослать его в несколько других процессоров. В частности, если какая-либо переменная встречается несколько раз подряд в качестве использования, то будем считать, что значение этой переменной достаточно прочесть из памяти один раз. Тогда из соответствующей вершины графа будет выходить несколько дуг и, следовательно, граф вычислений уже не будет деревом, хотя останется бесконтурным графом.

Пример 126. Для оператора присваивания a=b*c+b граф со склеенными вершинами выглядит следующим образом:

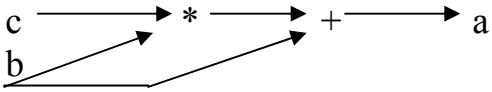


Рис. 12. Склейка in-in зависимых вершин.

Не обязательно склеивать все вершины, читающие одно и то же данное. По этой причине по данному графу можно построить не единственный свернутый граф. Далее, когда будет идти речь о свернутом графе, можно будет понимать "какой-нибудь" свернутый граф.

Свернутый граф вычислений требует меньше операций чтения данных, чем исходный.

Возможность нескольких вариантов свернутого графа допускается для случая возможных последующих обобщений, которые могут потребоваться при введении ограничений на коммутатор.

Циклически порожденное IN-IN склеивание. Склеивание вершин графа происходит для вхождений, связанных регулярной in-in зависимостью. Вхождения, соответствующие таким вершинам, обращаются к памяти за чтением одних и тех же данных (in-in-сокращение).

Алгоритм склеивания вершин, связанных регулярной in-in зависимостью состоит из трех частей:

1. Строим неориентированный граф регулярных in-in зависимостей (вершины этого графа - вхождения, а ребро соединяет две вершины тогда и только тогда, когда эти вхождения связаны регулярной информационной зависимостью типа in-in).

2. Каждую компоненту связности графа регулярных in-in зависимостей следует разбить на наименьшее количество частей так, чтобы для любых двух вершин каждой части при склеивании этих вершин не получалось дуги графа вычислений длиннее чем размер буфера.

3. Вершины каждой из полученных частей склеиваем.

В этом алгоритме пункты 1 и 3 могут быть реализованы несложным образом, а на пункте 2 остановимся подробнее.

Каждая компонента связности графа регулярных in-in зависимостей соответствует вхождению одной и той же переменной. Пусть данная компонента связности соответствует массиву X. Не умаляя общности, будем считать, что массив X одномерный. Каждой вершине v, соответствующей вхождению переменной X(I+b), поставим в соответствие число g(v) = b (ввиду регулярности зависимостей, для всех вхождений величина a одинакова). Расположим все вершины в порядке неубывания функции g. Пусть v1 – первая из них (с наименьшим значением g). Пусть h – размер буфера. Тогда к первой части отнесем все те вершины vk, для которых $g(vk) - g(v1) \le h$. Отбросив первую часть, аналогичным образом построим вторую и т.д. Очевидно, что данный алгоритм оптимален по качеству разбиения и по быстродействию.

Пример 127.

```
      DO 99 I = 1, N
99     Y(I) =  X(I)... X(I-1)... X(I-9)... X(I-5)... X(I+9)... X(I+3)... X(I+11)
                 v1       v2       v3       v4       v5       v6       v7
```

Пусть размерность буфера h равна 8. Тогда все семь вхождений можно разбить на три группы: первая – v3, v4, v2; вторая – v1, v6; третья – v5, v7.

Приведенный алгоритм склеивания вершин более актуален для упоминавшегося случая, при котором различные операции имеют задержки различной длины.

Описанное в данном параграфе склеивание не является необходимым. Если не пользоваться таким склеиванием, то в дальнейшем, при конвейеризации может понадобиться больше операций чтения данных и, возможно, копирование одного массива в различных секторах памяти.

ОТОБРАЖЕНИЕ ЦИКЛА НА АРХИТЕКТУРУ КОМПЬЮТЕРА

Свернутым графом цикла будем называть граф вычислений тела цикла, преобразованный циклически независимыми и циклически порожденными out-in и in-in склеиваниями.

Заметим, что у свернутого графа из вершины выходит более одной дуги в двух случаях. В первом случае эта вершина является результатом склейки нескольких вершин, связанных с чтением данных. В другом случае, такая вершина является результатом out-in склейки, – одна из выходящих дуг связана с записью данных, а остальные с чтением.

Для конвейерного выполнения программного цикла следует каждой операции, встречающейся в цикле, сопоставить устройство суперкомпьютера, в котором эта операция будет выполняться. Операциям чтения и записи данных сопоставляется модуль памяти, в котором это данное должно находится.

Все операции тела цикла находятся во взаимно однозначном соответствии с вершинами свернутого графа цикла. Поэтому достаточно построить отображение вершин этого графа во множество элементарных процессоров суперкомпьютера. Ясно, что это отображение должно быть инъективно.

Если результат одной операции используется, как аргумент другой операции, то с помощью коммутатора выход первого процессора должен быть соединен с входом второго. Но все пары операций, в которых результат одной является аргументом для другой, находятся во взаимно однозначном соответствии с дугами свернутого графа. Следовательно, по дугам свернутого графа можно определить задание для коммутатора – какие устройства между собой следует соединять.

СИНХРОНИЗАЦИЯ КОНВЕЙЕРА

Но для правильной работы конвейера недостаточно соединить устройства суперкомпьютера по указанной схеме. Может случиться так, что, двигаясь по такой схеме, разные аргументы какой-то операции придут в соответствующий процессор в разное время, ввиду того, что пути прохождения могут оказаться разной длины. Для решения этой проблемы конвейер следует синхронизировать. Синхронизация выполняется с помощью буферов, которые задерживают данные перед их поступлением в процессор на необходимое время. Рассмотрим алгоритм определения значений буферов для различных случаев с нарастанием их общности и сложности.

Один оператор. Индексы без сдвигов

Пример 128.

```
 DO 99 I=3, N
 Y(I) = X(I) + B(I)*X(I)
99 CONTINUE
```

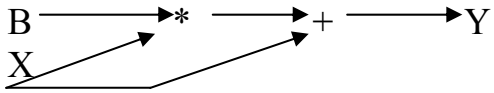


Рис. 13. Склейка с циклически порожденной in-in зависимостью

На рисунке 11 один и тот же элемент X(I) должен поступить в складывающий процессор на один такт позже, чем в умножающий.

Добавим к графу дополнительную вершину S, из которой построим дуги в источники исходного графа. Для полученного бесконтурного графа с одним источником можно построить дерево самых длинных путей с корнем S по аналогии с известным деревом кратчайших путей в бесконтурном графе [**30**].

Алгоритм синхронизации конвейера.

1. По свернутому графу цикла строим дерево самых длинных путей.
2. Каждой вершине графа x ставим в соответствие число f(x) , равное длине пути от описанной выше вершины S до x .
3. Для каждой дуги (y, x) определяем значение буфера, как величину f(x) – f(y) – 1 .
4. Для каждой вершины графа x момент начала работы соответствующего процессора определим величиной f(x).

Несложно убедиться, что в результате определения величин f(x) и значений буферов f(x) – f(y) – 1 , для любых двух вершин время движения по любым путям между этими вершинами (с учетом задержек в буферах) одинаково. Это и означает синхронизацию конвейера, т.е. что в каждый процессор нужные данные будут поступать одновременно.

Один оператор. Индексы со сдвигами. Граф без контуров

Пример 129.

```
 DO 99 I=3, N
 Y(I) = X(I-2) + B(I)*X(I)
99 CONTINUE
```

Граф для этого примера выглядит таким же образом, как и для предыдущего. В этом случае на дуге (X,+) перед изначально пустым буфером, построенным в предыдущем пункте, должно быть установлено еще две ячейки буфера с начальными данными X(1) и X(2) .

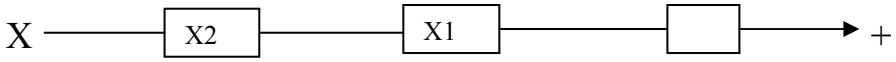


Рис. 14. Ячейки буфера с начальными данными.

Конец примера.

Много операторов. Индексы со сдвигами. Граф без контуров

В этом случае с дугами, которые связаны с out-in склейками, поступаем так же, как и с дугами, связанными с in-in склейками. Аналогичным образом на такие дуги могут быть поставлены и ячейки буфера с начальными данными.

Один оператор. Индексы со сдвигами. Одна обратная дуга

Рассмотрим случай, когда у свернутого графа цикла есть контуры, но удаление одной дуги делает этот граф бесконтурным. Будем считать, что эта дуга соответствует циклически порожденной out-in зависимости, и эту дугу будем называть обратной.

Обозначим рассматриваемую обратную дугу (y_0, x_0). Таким образом, данные на какой-то начальной итерации поступают на обработку сначала в элементарный процессор x_0, затем проходят обработку, двигаясь по некоторой цепочке процессоров, после чего, на какой-то из последующих итераций, по дуге (y_0, x_0) опять приходят в процессор x_0.

Может так случиться, что процессор x_0, выполняющий бинарную операцию, по обратной дуге (y_0, x_0) получает аргумент после того, как уже пришел другой аргумент этой операции.

Пример 130.

```
 DO 99 I = 2, N
99    X(I) = Y(I)+Z(I)*X(I-1)
```

Свернутый граф этого рекуррентного цикла будет иметь следующий вид :

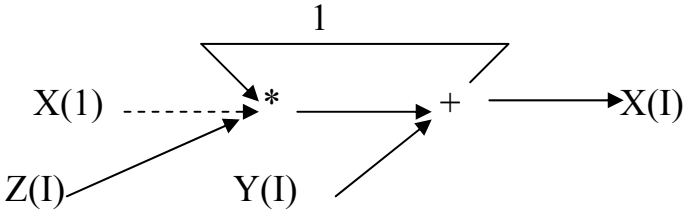


Рис. 15. Конфликт в подаче данных при обратной связи.

В этом примере при I = 3 мультипликативный процессор должен вычислять произведение X(2) и Z(3), но аддитивный процессор не успел вычислить X(2), как сумму Y(2) и (X(1)*Z(2)) .

В рассмотренной ситуации данные следует подавать на конвейер через несколько тактов. Количество тактов, через которые подаются данные, обозначим символом T. По аналогии с программной конвейеризацией будем эту величину называть интервалом инициализации итераций [**39**]. Тогда время выполнения цикла с N итерациями на конвейере равно T*N+L, где L – некоторая константа (равная длительности выполнения одной итерации цикла). Поэтому, чем меньше величина T , тем эффективнее работает конвейер. Алгоритм вычисления величины T приведем ниже для более общего случая.

Один оператор присваивания. Индексы со сдвигами
Обратные дуги

Поскольку оператор в теле цикла один, все обратные дуги выходят из одной вершины (соответствующей генератору), которую обозначим y_0. Удалим из свернутого графа рассматриваемого цикла обратные дуги (y_0,x_i). Найдем в полученном бесконтурном графе множество всех путей из x_i в y_0.

Обозначим для i-ого пути сдвиг в вершине x_i через a_i – это число итераций, через которое используются данные, передаваемые по обратной дуге. Время прохождения данных по этому пути равно количеству дуг b_i . Для каждого пути с номером i должно выполняться неравенство

$$a_i * T >= b_i .$$

Из этого неравенства вытекает, что в качестве T можно взять величину

$$T =] \max_i \{b_i / a_i\} [$$

Здесь]x[означает наименьшее целое число, которое не меньше числа x.
Для тех обратных дуг, для которых отношение b_i / a_i меньше величины T, следует установить соответствующие значения буферов $a_i * T - b_i$.

Пример 131.

```
      DO 99 I = 2, N
99    X(I) = X(I-1) + Y(I)* (A(I)+X(I-2))
```

Свернутый граф рассматриваемого цикла содержит две обратных дуги. Для одной из них отношение b_i / a_i равно 1, а для другой 3/2. Таким образом, T = 2 и на каждой обратной дуге данные должны задерживаться в буфере на один такт.

Много операторов. Индексы со сдвигами. Граф без контуров. Подача данных с интервалами

На конвейер можно подавать данные не каждый такт, а с интервалами в несколько тактов (пусть, для определенности, T тактов). В этом случае, перед каждой ячейкой буфера с начальными данными следует иметь еще (T-1) дополнительных пустых ячеек буфера. Заметим, что в данном случае, для цикла с бесконтурным свернутым графом подача данных с интервалом возможна, но не целесообразна. Рассмотрение этого случая удобно для перехода к следующему более общему и практически значимому.

Много операторов. Индексы со сдвигами. Обратные дуги

Пример 132. В этом примере приводится ситуация, в которой в качестве обратной дуги, разрывающей контур, можно выбрать любую из двух (+, /) или (-, *).

```
      DO 99 I = 2, N
      X(I) = Y(I-1)*A(I)+Z(I)
      Y(I) = X(I-2)/B(I)-W(I)
99    CONTINUE
```

Свернутый граф этого рекуррентного цикла будет иметь следующий вид :

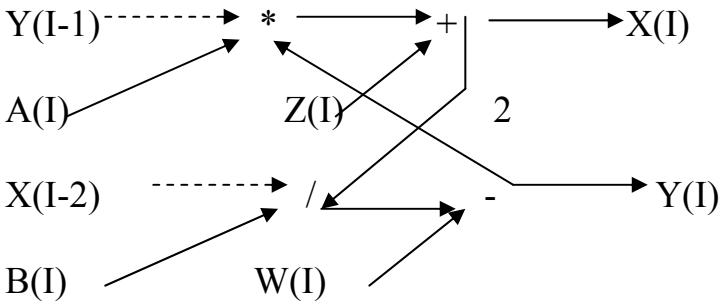


Рис. 16. Контур с двумя обратными дугами.

На вычисление T не влияет выбор разрываемой обратной дуги.

Теорема 37. Пусть в свернутом графе тела цикла есть один контур. Пусть в этот контур входят несколько обратных дуг (A,A'), (B,B'), (C,C') ... со сдвигами a, b, c,... соответственно. Предположим, что к началу каждой такой дуги A, B, C, ... от конца предыдущей ведет путь, состоящий из L(a), L(b), L(c),... дуг соответственно. Тогда минимальный интервал инициализации итераций конвейера не зависит от обратной дуги контура, которая считается разрываемой, и этот интервал равен

$$T =](L(a)+L(b)+L(c)+...) / (a+b+c+...)[$$

Доказательство. Предположим, что контур разрывается дугой (A,A'). Тогда данные, вычисленные в вершине A, потребуются для использования в вершине A' через a*T времени. С другой стороны, при вычислении исходные

данные движутся от A' к A следующее количество тактов (L(a)+L(b)+L(c)+...) – T*(b+c+...). Поскольку данные должны быть вычислены до необходимости их использования, для возможности синхронизации конвейера должно выполняться неравенство

(L(a)+L(b)+L(c)+...) – T*(b+c+...) =< a*T

Из этого неравенства вытекает формула условия теоремы.

Полученная формула периода подачи данных известна для задач программной конвейеризации для суперкомпьютеров с архитектурой VLIW [**39**, с. 24]. Эта формула используется, как нижняя оценка интервала инициализации итераций. В случае архитектуры VLIW этот интервал инициализации итераций может быть увеличен, если у компьютера не достаточно ресурсов для выполнения тела цикла в соответствии с некоторым расписанием. Для суперкомпьютера с архитектурой перестраиваемого конвейера, если ресурсов не достаточно, то цикл не конвейеризуем и увеличение периода подачи данных не поможет, а, если ресурсов достаточно, то цикл конвейеризуется и полученная в теореме формула дает оптимальный период подачи данных.

Теорема 38. Если в свернутом графе много контуров, то в качестве периода чтения данных следует брать максимальную из величин интервалов, посчитанных для каждого контура.

Доказательство. Очевидно, что величина периода подачи данных не может быть меньше такой величины, вычисленной для какого-либо одного из контуров графа. Достаточность рассматриваемой величины доказывается следующим алгоритмом синхронизации.

Алгоритм синхронизации конвейера выглядит следующим образом.

1. Вычисляем период подачи данных.
2. Разрываем все out-in дуги (выходящие из генераторов)
3. Возвращаем по одной (в любом порядке) те разорванные дуги, которые не создают новых неориентированных циклов или ориентированных контуров.
4. Синхронизируем бесконтурный подграф свернутого графа, полученный из исходного.
5. Возвращаем по одной оставшиеся out-in дуги к рассмотренному графу с синхронизацией, в зависимости от следующих 2 случаев
    a. Если возвращаемая дуга создает неориентированные циклы, ставим на этой дуге буфер, синхронизирующий один из новых циклов. Остальные новые циклы окажутся синхронизированными автоматически, т.к. до добавления обратной дуги все пути от начала этой дуги к ее концу были на текущий момент синхронизированы. Следует оговорить корректность. Не умаляя общности, можно считать, что в этом случае возвращаемая дуга может входить в вершину с двумя входами (соответствующую бинарной

операции). Тогда буфер следует поставить на одну из этих двух дуг.

b. Если создаются ориентированные контуры, то достаточно буфером на возвращаемой дуге синхронизировать один из них, а остальные синхронизируются автоматически, т.к. до возвращения out-in дуги все пути от конца этой дуги к ее началу были на текущий момент синхронизированы.

3. ПАРАЛЛЕЛЬНОЕ РЕШЕНИЕ СИСТЕМ СЛАУ С ЛЕНТОЧНЫМИ МАТРИЦАМИ И РАСПАРАЛЛЕЛИВАНИЕ РЕКУРРЕНТНЫХ ЦИКЛОВ

В данной главе рассматриваются параллельные алгоритмы решения систем линейных уравнений с ленточными матрицами. К решению таких систем уравнений сводится вычисление программных циклов с линейной рекуррентной зависимостью. Для дополнительного ознакомления с параллельными численными методами решения систем линейных алгебраических уравнений можно порекомендовать обзор [**37 - 38**], а также соответствующие главы из книг [**36**, **7**] .

Матрица $A=\{a_{ij}\}$ размера n*n называется ленточной, если существуют такие натуральные числа b и c, значительно меньшие n, что для любых i, j таких, что $i - j < -b$ или $i - j > c$, выполняется $a_{ij} = 0$. Если b и c - наименьшие такие числа, то число $d = b+c$ называется шириной ленты. В памяти компьютера такая матрица представляется в виде одномерных массивов, в которых хранятся элементы ненулевых диагоналей.

Иногда от матриц будем требовать условие диагонального преобладания. Напомним, что матрица A имеет диагональное преобладание, если в любой строке модуль диагонального элемента больше суммы модулей остальных элементов строки.

СЛАУ С ДВУХДИАГОНАЛЬНЫМИ МАТРИЦАМИ

Будем рассматривать систему уравнений с ленточной двухдиагональной матрицей следующего вида

$$\left(\begin{array}{cccccc|c} a_1 & & & & & & h_1 \\ b_2 & a_2 & & & & & h_2 \\ & b_3 & a_3 & & & & h_3 \\ & & b_4 & a_4 & & & h_4 \\ . & . & . & . & . & . & . \\ & & & & b_n & a_n & h_n \end{array}\right)$$

Ясно, что эта система имеет единственное решение тогда и только тогда, когда все элементы главной диагонали отличны от нуля. В этом случае каждое i-ое уравнение можно разделить на a_i. Все эти деления можно выполнить одновременно. В результате задача сведется к решению системы уравнений с матрицей следующего вида.

$$\left(\begin{array}{cccccc|c} 1 & & & & & & h_1 \\ b_2 & 1 & & & & & h_2 \\ & b_3 & 1 & & & & h_3 \\ & & b_4 & 1 & & & h_4 \\ . & . & . & . & . & . & . \\ & & & & b_n & 1 & h_n \end{array}\right) \qquad (17)$$

Параллельный алгоритм решения такой системы уравнений состоит в следующем. На первом шаге следует одновременно получить нули на месте элементов b(I). Для этого из каждой I-ой строки надо вычесть предыдущую, умноженную на число b(I). При этом все элементы поддиагонали станут нулевыми, но элементы второй поддиагонали окажутся ненулевыми.

$$\left(\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_1 \\ b_2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & h_2 \\ 0 & b_3 & 1 & 0 & 0 & 0 & 0 & 0 & h_3 \\ 0 & 0 & b_4 & 1 & 0 & 0 & 0 & 0 & h_4 \\ 0 & 0 & 0 & b_5 & 1 & 0 & 0 & 0 & h_5 \\ 0 & 0 & 0 & 0 & b_6 & 1 & 0 & 0 & h_6 \\ 0 & 0 & 0 & 0 & 0 & b_7 & 1 & 0 & h_7 \\ 0 & 0 & 0 & 0 & 0 & 0 & b_8 & 1 & h_8 \end{array}\right) \sim \left(\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & h_2 \\ b_3 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & h_3 \\ 0 & b_4 & 0 & 1 & 0 & 0 & 0 & 0 & h_4 \\ 0 & 0 & b_5 & 0 & 1 & 0 & 0 & 0 & h_5 \\ 0 & 0 & 0 & b_6 & 0 & 1 & 0 & 0 & h_6 \\ 0 & 0 & 0 & 0 & b_7 & 0 & 1 & 0 & h_7 \\ 0 & 0 & 0 & 0 & 0 & b_8 & 0 & 1 & h_8 \end{array}\right) \sim$$

На втором шаге из каждой I-ой строки надо вычесть строку с номером (I-2), умноженную на число b(I). При этом все элементы второй поддиагонали станут нулевыми, но элементы в четвертой поддиагонали окажутся ненулевыми.

$$\left(\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & h_2 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & h_3 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & h_4 \\ b_5 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & h_5 \\ 0 & b_6 & 0 & 0 & 0 & 1 & 0 & 0 & h_6 \\ 0 & 0 & b_7 & 0 & 0 & 0 & 1 & 0 & h_7 \\ 0 & 0 & 0 & b_8 & 0 & 0 & 0 & 1 & h_8 \end{array}\right) \sim \left(\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & h_1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & h_2 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & h_3 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & h_4 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & h_5 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & h_6 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & h_7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & h_8 \end{array}\right)$$

На k-ом шаге из каждой I-ой строки надо вычесть строку с номером I-2**(k-1), умноженную на число b(I). При этом, все элементы поддиагонали с номером 2**(k-1) станут нулевыми, но элементы поддиагонали с номером 2**k окажутся ненулевыми. Через]log(N)[шагов матрица системы станет единичной и в правой части будет получен вектор-решение (здесь и далее логарифмы берутся по основанию 2 и операция]a[означает наименьшее целое, не меньшее числа a).

Заметим, что в этом алгоритме ненулевая поддиагональ после каждого шага оказывается вдвое дальше от главной диагонали, чем перед этим шагом.

Описанный алгоритм можно записать в виде небольшой программы на фортране.

```
      do  99  k = 1, ]log(N)[
      do  99   I = 2**(k-1)+1, N
      h(I)= h(I) - b(I)*h(I-2**(k-1))                    (18)
99    b(I) = - b(I)*b(I-2**(k-1))
```

Следует обратить внимание, что операторы присваивания в этом цикле нельзя менять местами – их следует выполнять именно в том порядке, в каком они записаны.

Все итерации внутреннего цикла могут выполняться одновременно (параллельно). Это возможно на суперкомпьютерах с векторной, SIMD или MIMD архитектурами. Кроме того, внутренний цикл может быть выполнен на конвейерной архитектуре. Объемлющий цикл не распараллеливается (его итерации нельзя выполнять одновременно) ввиду выходной зависимости по данным.

Для верхнетреугольных двухдиагональных матриц алгоритм решения системы уравнений выглядит аналогичным образом.

$$\left(\begin{array}{cccccc|c} 1 & b_1 & & & & & h_1 \\ & 1 & b_2 & & & & h_2 \\ & & 1 & b_3 & & & h_3 \\ & & & 1 & & & h_4 \\ . & . & . & . & . & . & . \\ & & & & & 1 & h_n \end{array}\right)$$

```
      do  99  k = 1, ]log(N)[
      do  99   I = 1, N - 2**(k-1)
      h(I)= h(I) - b(I)*h(I+2**(k-1))
99    b(I) = - b(I)*b(I+2**(k-1))
```

О распараллеливании этого алгоритма справедливы все слова, написанные для случая нижнетреугольных матриц.

ПРИМЕНЕНИЯ АЛГОРИТМА ПАРАЛЛЕЛЬНОГО РЕШЕНИЯ СИСТЕМЫ УРАВНЕНИЯ С ЛЕНТОЧНОЙ ДВУХДИАГОНАЛЬНОЙ МАТРИЦЕЙ

Описанный выше алгоритм можно применить для параллельного вычисления многочлена

$f(x) = a_0x^n + a_1x^{n-1} + \ldots + a_{n-1}x + a_n$

Для этого достаточно решить систему, n-ый корень которой равен значению этого многочлена в точке x.

$$\left(\begin{array}{cccccc|c} 1 & & & & & & a_0 \\ -x & 1 & & & & & a_1 \\ & -x & 1 & & & & a_2 \\ & & -x & 1 & & & a_3 \\ . & . & . & . & . & . & . \\ & & & & -x & 1 & a_n \end{array}\right)$$

Для одновременного вычисления всех n сумм квадратных матриц

A_1
$A_1 + A_2$
$A_1 + A_2 + A_3$
……
$A_1 + A_2 + A_3 + \ldots + A_n$
достаточно параллельно решить блочную систему уравнений, корнями которой являются эти суммы

$$\left(\begin{array}{cccccc|c} 1 & & & & & & A_1 \\ -1 & 1 & & & & & A_2 \\ & -1 & 1 & & & & A_3 \\ & & -1 & 1 & & & A_4 \\ . & . & . & . & . & . & . \\ & & & & -1 & 1 & A_n \end{array}\right)$$

Для одновременного вычисления всех произведений квадратных матриц

A_1
$A_1 * A_2$
$A_1 * A_2 * A_3$
……
$A_1 * A_2 * A_3 * \ldots * A_n$
достаточно параллельно решить блочную систему уравнений, корнями которой являются эти произведения

$$\left(\begin{array}{cccccc|c} 1 & & & & & & I \\ -A_1 & 1 & & & & & 0 \\ & -A_2 & 1 & & & & 0 \\ & & -A_3 & 1 & & & 0 \\ . & . & . & . & . & . & . \\ & & & & -A_n & 1 & 0 \end{array}\right)$$

Последние две задачи рассматривались в [9]

СЛАУ С БЛОЧНЫМИ ДВУХДИАГОНАЛЬНЫМИ МАТРИЦАМИ

Будем рассматривать системы линейных уравнений следующего вида

$$\left(\begin{array}{cccccc|c} 1 & & & & & & H_1 \\ B_2 & 1 & & & & & H_2 \\ & B_3 & 1 & & & & H_3 \\ & & B_4 & 1 & & & H_4 \\ . & . & . & . & . & . & . \\ & & & & B_n & 1 & H_n \end{array}\right)$$

Здесь элементами n*n матрицы являются матрицы (блоки) размера m*m . Координатами вектора правой части системы являются векторы длины m. Параллельный алгоритм решения такой системы уравнений аналогичен алгоритму в скалярном случае. Этот алгоритм можно условно записать программой

```
      do  99  k = 1, ]log(N)[
      do  99   I = 2**(k-1)+1, N
      H(I)= H(I) - B(I)*H(I-2**(k-1))
99    B(I) = - B(I)*B(I-2**(k-1))
```

Эту запись следует считать условной, поскольку в фортране нет операций над векторами и матрицами. Всякую такую операцию в исполняемой программе надо заменить процедурой.

СЛАУ С ЛЕНТОЧНЫМИ ТРЕУГОЛЬНЫМИ МАТРИЦАМИ

Решение СЛАУ с ленточными треугольными матрицами сводиться к решению СЛАУ с блочными двухдиагональными матрицами. Будем рассматривать случай невырожденной матрицы системы. Не умаляя общности можно считать, что на главной диагонали матрицы стоят единицы (иначе каждую строку матрицы разделили бы на элемент главной диагонали). Будем также рассматривать случай нижнетреугольной матрицы. Для верхнетреугольной матрицы алгоритм выглядит аналогично.

$$\left(\begin{array}{cccccccccc|c}
1 & & & & & & & & & & h_1 \\
b_2 & 1 & & & & & & & & & h_2 \\
\hline
c_3 & b_3 & 1 & & & & & & & & h_3 \\
 & c_4 & b_4 & 1 & & & & & & & h_4 \\
\hline
 & & c_5 & b_5 & 1 & & & & & & h_5 \\
 & & & c_6 & b_6 & 1 & & & & & h_6 \\
\hline
. & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . \\
\hline
 & & & & & & c_{n-1} & b_{n-1} & 1 & & h_{n-1} \\
 & & & & & & & c_n & b_n & 1 & h_n
\end{array}\right)$$

Итак, пусть в нижнетреугольной матрице m-ая диагональ ненулевая, а все поддиагонали, начиная с (m+1)-ой нулевые. (В рассматриваемом примере m=2). Тогда матрицу системы разобьем на блоки размера m*m. Получится блочная двухдиагональная матрица, причем по главной диагонали стоят, очевидно, обратимые блоки. В данном примере

$$A_i = \begin{pmatrix} 1 & 0 \\ b_{2i} & 1 \end{pmatrix}, \quad B_i = \begin{pmatrix} c_{2i-1} & b_{2i-1} \\ 0 & c_{2i} \end{pmatrix}$$

Умножим каждую блочную строку слева на блок, обратный к диагональному - получим также систему с блочной двухдиагональной матрицей, причем по главной диагонали будут стоять единичные блоки.

$$\left(\begin{array}{cccccccccc|c}
1 & & & & & & & & & & t_1 \\
 & 1 & & & & & & & & & t_2 \\
\hline
y_3 & x_3 & 1 & & & & & & & & t_3 \\
z_4 & y_4 & & 1 & & & & & & & t_4 \\
\hline
 & & y_5 & x_5 & 1 & & & & & & t_5 \\
 & & z_4 & y_6 & & 1 & & & & & t_6 \\
\hline
. & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . \\
\hline
 & & & & & & y_{n-1} & x_{n-1} & 1 & & t_{n-1} \\
 & & & & & & z_n & y_n & & 1 & t_n
\end{array}\right)$$

И так далее по алгоритму решения СЛАУ с блочной двухдиагональной матрицей.

Пример 133. Рассмотрим задачу вычисления элементов последовательности Фибоначчи: 1, 1, 2, 3, 5, 8, 13, 21, 34, … Последовательность чисел Фибоначчи задается рекуррентной формулой

X(I) = X(I-1)+X(I-2), I=3,4,...N ; X(1) = 1, X(2) = 1.

Подставляя вместо I последовательно значения 3,4,..., получим систему уравнений относительно неизвестных X(I)

X(1) = 1
X(2) = 1
X(3) = X(2)+X(1)
X(4) = X(3) +X(2)
.....
X(N) = X(N-1)+X(N-2)

Перенесем все X(I) из правой части в левую и запишем матрицу системы. Заметим, что эта матрица треугольная и ленточная. Решим эту систему для случая N=8.

$$\left(\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 \end{array}\right) \sim \left(\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & -2 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -2 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -2 & 0 & 1 & 0 \end{array}\right) \sim$$

$$\left(\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 3 \\ -2 & -3 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ -3 & -5 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -2 & -3 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -3 & -5 & 0 & 0 & 0 & 1 & 0 \end{array}\right) \sim \left(\begin{array}{cccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 13 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 21 \end{array}\right)$$

СЛАУ С БЛОЧНОЙ ТРЕХДИАГОНАЛЬНОЙ НЕТРЕУГОЛЬНОЙ МАТРИЦЕЙ

Ненулевые блоки матрицы системы стоят на главной диагонали, первой поддиагонали и первой наддиагонали.

$$\left(\begin{array}{cccccc|c} A_1 & C_1 & & & & & H_1 \\ B_2 & A_2 & C_2 & & & & H_2 \\ & B_3 & A_3 & C_3 & & & H_3 \\ & & B_4 & A_4 & C_4 & & H_4 \\ . & . & . & . & . & . & . \\ & & & & B_n & A_n & H_n \end{array}\right) \qquad (19)$$

Алгоритм решения такой системы состоит из]$\log_2$ N[шагов. После каждого шага алгоритма матрица остается трехдиагональной: главная диагональ, одна поддиагональ и одна наддиагональ. При этом, на k-ом шаге алгоритма ненулевые поддиагональ и наддиагональ оказываются на расстоянии 2**(k-1) от главной диагонали (т.е. после каждого шага ненулевые поддиагональ и наддиагональ удаляются вдвое от главной). После]$\log_2$ N[шагов матрица системы становится единичной, что и завершает работу алгоритма.

На k-ом шаге алгоритма вычисляются сначала три вспомогательных блочных массива (каждая блочная строка умножается на блок, обратный к диагональному).

$$\overline{C}_i^k = A_i^{-1} * C_i^k$$
$$\overline{B}_i^k = A_i^{-1} * B_i^k$$
$$\overline{H}_i^k = A_i^{-1} * H_i^k$$

После этого вычисляются элементы k-ой матрицы.

$$C_i^k = -\overline{C}_i^{k-1} * \overline{C}_{i+2^{k-1}}^{k-1}$$
$$B_i^k = -\overline{B}_i^{k-1} * \overline{B}_{i-2^{k-1}}^{k-1}$$
$$A_i^k = I - \overline{B}_i^{k-1} * \overline{C}_{i-2^{k-1}}^{k-1} - \overline{C}_i^{k-1} * \overline{B}_{i+2^{k-1}}^{k-1}$$
$$H_i^k = \overline{H}_i^{k-1} - \overline{B}_i^{k-1} * \overline{H}_{i-2^{k-1}}^{k-1} - \overline{C}_i^{k-1} * \overline{H}_{i+2^{k-1}}^{k-1}$$

$$\left(\begin{array}{ccccccc|c} A_1 & C_1 & 0 & 0 & 0 & 0 & 0 & H_1 \\ B_2 & A_2 & C_2 & 0 & 0 & 0 & 0 & H_2 \\ 0 & B_3 & A_3 & C_3 & 0 & 0 & 0 & H_3 \\ 0 & 0 & B_4 & A_4 & C_4 & 0 & 0 & H_4 \\ 0 & 0 & 0 & B_5 & A_5 & 0 & 0 & H_5 \\ & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & A_{n-1} & C_{n-1} & H_{n-1} \\ 0 & 0 & 0 & 0 & 0 & B_n & A_n & H_n \end{array}\right) \sim$$

$$\left(\begin{array}{ccccccc|c} 1 & 0 & \overline{C}_1^2 & 0 & 0 & 0 & 0 & \overline{H}_1^2 \\ 0 & 1 & 0 & \overline{C}_2^2 & 0 & 0 & 0 & \overline{H}_2^2 \\ \overline{B}_3^2 & 0 & 1 & 0 & \overline{C}_3^2 & 0 & 0 & \overline{H}_3^2 \\ 0 & \overline{B}_4^2 & 0 & 1 & 0 & 0 & 0 & \overline{H}_4^2 \\ 0 & 0 & \overline{B}_5^2 & 0 & 1 & 0 & 0 & \overline{H}_5^2 \\ & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & \overline{H}_{n-1}^2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \overline{H}_n^2 \end{array}\right) \qquad (20)$$

$$\sim \left(\begin{array}{ccccc|cc|c} 1 & 0 & 0 & 0 & \overline{C}_1^3 & 0 & 0 & \overline{H}_1^3 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & \overline{H}_2^3 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & \overline{H}_3^3 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & \overline{H}_4^3 \\ \overline{B}_5^3 & 0 & 0 & 0 & 1 & 0 & 0 & \overline{H}_5^3 \\ & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & \overline{H}_{n-1}^3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \overline{H}_n^3 \end{array}\right) \sim \ldots \sim \left(\begin{array}{ccccc|cc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & \overline{H}_1^4 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & \overline{H}_2^4 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & \overline{H}_3^4 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & \overline{H}_4^4 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & \overline{H}_5^4 \\ & & & & & & & \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & \overline{H}_7^4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \overline{H}_8^4 \end{array}\right)$$

Если считать, что массивы A, B, C и H доопределены нулевыми значениями для индексов из промежутка [-n, 2*n], то алгоритм может быть записан в виде следующей псевдофортрановской программы («псевдо-» – поскольку в записи используются операции над матрицами, которых в фортране нет)

```
      do  11   I = 1, N-1
11    C̄_i^k = A_i^{-1} * C_i^k
      do  22   I = 2, N
22    B̄_i^k = A_i^{-1} * B_i^k
      do  33   I = 1, N
33    H̄_i^k = A_i^{-1} * H_i^k

      do  99  k = 2, ]log(N)[
      T = 2**(k-1)
      do  44   I =1, N-T
44    C_i^k = -C̄_i^{k-1} * C̄_{i+2^{k-1}}^{k-1}
      do  55   I = T, N
55    B_i^k = -B̄_i^{k-1} * B̄_{i-2^{k-1}}^{k-1}
      do  66   I = 1, N
      A_i^k = I - B̄_i^{k-1} * C̄_{i-2^{k-1}}^{k-1}  - C̄_i^{k-1} * B̄_{i+2^{k-1}}^{k-1}
66    H_i^k = H̄_i^{k-1} - B̄_i^{k-1} * H̄_{i-2^{k-1}}^{k-1}  - C̄_i^{k-1} * H̄_{i+2^{k-1}}^{k-1}
      do  11   I = 1, N-T
11    C̄_i^k = A_i^{-1} * C_i^k
      do  22   I = T+1, N
22    B̄_i^k = A_i^{-1} * B_i^k
      do  33   I = 1, N
33    H̄_i^k = A_i^{-1} * H_i^k
99    CONTINUE
```

Если не использовать доопределения значений массива, то алгоритм может быть записан в следующем виде

```
      T = 1
      do  11   I = 1, N-T
11    C(I,K) = A(I,-1)*C(I,K)
      do  22   I = T+1, N
22    B(I,K) = A(I,-1)*B(I,K)
      do  33   I = 1, N
33    H(I,K) = A(I,-1)*H(I,K)

      do  99  k = 2, ]log(N)[
      T=2*T
C     T = 2**(k-1)
      do  44   I =1, N-T
44    C(I,K) = -C(I,K-1)*C(I+T,K-1)
      do  55   I = T, N
55    B(I,K) = -B(I,K-1)*B(I-T,K-1)
      do  661   I = 1, T
      A(I,K) = I - C(I,K-1)*B(I+T,K-1)
661   H(I,K) = H(I,K-1) - C(I,K-1)*H(I+T,K-1)
      do  662   I = N-T+1, N
      A(I,K) = I - B(I,K-1)*C(I-T,K-1)
662   H(I,K) = H(I,K-1) - B(I,K-1)*H(I-T,K-1)
      do  663   I = T+1, N-T
      A(I,K) = I - B(I,K-1)*C(I-T,K-1) - C(I,K-1)*B(I+T,K-1)
663   H(I,K) = H(I,K-1) - B(I,K-1)*H(I-T,K-1) - C(I,K-1)*H(I+T,K-1)
      do  11   I = 1, N-T
11    C(I,K) = A(I,-1)*C(I,K)
      do  22   I = T+1, N
22    B(I,K) = A(I,-1)*B(I,K)
      do  33   I = 1, N
33    H(I,K) = A(I,-1)*H(I,K)

99    CONTINUE
```

Описанный алгоритм не всегда корректен, поскольку он содержит операцию обращения блоков. Даже если у исходной матрицы блоки главной диагонали обратимы, то нет гарантий, что это свойство будет сохраняться после каждого шага. Оказывается, что диагональное преобладание исходной матрицы является достаточным для корректности этого алгоритма.

Условие диагонального преобладания для блочной матрицы имеют следующий вид

$$\left\|\overline{B_i}\right\|+\left\|\overline{C_i}\right\|=\left\|A_i^{-1}*B_i\right\|+\left\|A_i^{-1}*C_i\right\|\le 1$$

Несложно заметить, что при размерности блоков 1, т.е. для скалярной матрицы, это условие равносильно обычному условию диагонального преобладания

$$\left|b_i\right|+\left|c_i\right|\le\left|a_i\right|$$

Теорема 39. Если исходная матрица системы (**19**) обладает свойством диагонального преобладания, то и после каждого шага алгоритма матрица системы также обладает условием диагонального преобладания. В частности, после каждого шага алгоритма диагональные блоки обратимы.

Доказательство. Рассмотрим вспомогательное неравенство, при получении которого используется условие диагонального преобладания

$$\left\|\overline{B}_i^{k-1}*\overline{C}_{i-2^{k-1}}^{k-1}\right\|+\left\|\overline{B}_{i+2^{k-1}}^{k-1}*\overline{C}_i^{k-1}\right\|+\left\|\overline{B}_i^{k-1}*\overline{B}_{i-2^{k-1}}^{k-1}\right\|+\left\|\overline{C}_i^{k-1}*\overline{C}_{i+2^{k-1}}^{k-1}\right\|\le$$

$$\left\|\overline{B}_i^{k-1}\right\|*\left(\left\|\overline{C}_{i-2^{k-1}}^{k-1}\right\|+\left\|\overline{B}_{i-2^{k-1}}^{k-1}\right\|\right)+\left\|\overline{C}_i^{k-1}\right\|*\left(\left\|\overline{B}_{i+2^{k-1}}^{k-1}\right\|+\left\|\overline{C}_{i+2^{k-1}}^{k-1}\right\|\right)\le$$

$$\left(\left\|\overline{C}_i^{k-1}\right\|+\left\|\overline{B}_i^{k-1}\right\|\right)\le 1$$

Из данного неравенства следуют два нужных впоследствии неравенства

$$\left\|\overline{B}_i^{k-1}*\overline{C}_{i-2^{k-1}}^{k-1}\right\|+\left\|\overline{B}_{i+2^{k-1}}^{k-1}*\overline{C}_i^{k-1}\right\|\le 1$$

$$\left\|\overline{B}_i^{k-1}*\overline{B}_{i+2^{k-1}}^{k-1}\right\|+\left\|\overline{C}_i^{k-1}*\overline{C}_{i-2^{k-1}}^{k-1}\right\|\le 1-\left(\left\|\overline{B}_i^{k-1}*\overline{C}_{i-2^{k-1}}^{k-1}\right\|+\left\|\overline{B}_{i+2^{k-1}}^{k-1}*\overline{C}_i^{k-1}\right\|\right)$$

Теперь докажем, что на k-ом шаге условие диагонального преобладания выполняется, если оно выполняется на (k-1)-ом шаге.

$$\left\|\left(A_i^k\right)^{-1}*B_i^k\right\|+\left\|\left(A_i^k\right)^{-1}*C_i^k\right\|\le\left\|\left(A_i^k\right)^{-1}\right\|*\left(\left\|B_i^k\right\|+\left\|C_i^k\right\|\right)\le$$

$$\left\|\left(1-\left(\overline{B}_i^{k-1}*\overline{C}_{i-2^{k-1}}^{k-1}+\overline{B}_{i+2^{k-1}}^{k-1}*\overline{C}_i^{k-1}\right)\right)^{-1}\right\|*\left(\left\|B_i^k\right\|+\left\|C_i^k\right\|\right)=$$

Разлагая первый множитель в ряд фон-Неймана [19, стр. 222], продолжим преобразования

$$=\left\|\sum_{m=0}^{+\infty}\left(\overline{B}_i^{k-1}*\overline{C}_{i-2^{k-1}}^{k-1}+\overline{B}_{i+2^{k-1}}^{k-1}*\overline{C}_i^{k-1}\right)^m\right\|*$$

$$\left(\left\|\overline{B}_i^{k-1}*\overline{B}_{i+2^{k-1}}^{k-1}\right\|+\left\|\overline{C}_i^{k-1}*\overline{C}_{i-2^{k-1}}^{k-1}\right\|\right)\le$$

$$\le\left\|\sum_{m=0}^{+\infty}\left(\left\|\overline{B}_i^{k-1}*\overline{C}_{i-2^{k-1}}^{k-1}\right\|+\left\|\overline{B}_{i+2^{k-1}}^{k-1}*\overline{C}_i^{k-1}\right\|\right)^m\right\|*$$

$$\left(1-\left\|\overline{B}_i^{k-1}*\overline{C}_{i-2^{k-1}}^{k-1}\right\|-\left\|\overline{B}_{i+2^{k-1}}^{k-1}*\overline{C}_i^{k-1}\right\|\right)=1$$

СЛАУ С ЛЕНТОЧНОЙ МАТРИЦЕЙ С ПРЕОБЛАДАНИЕМ ПО ГЛАВНОЙ ДИАГОНАЛИ

Такая матрица может быть разбита на блоки так, что она станет блочной трехдиагональной матрицей. Этим самым задача решения СЛАУ сводится к уже описанной.

$$\left(\begin{array}{cc|cc|cc|cc|c}
1 & d_1 & & & & & & & h_1 \\
b_2 & 1 & d_2 & & & & & & h_2 \\
\hline
c_3 & b_3 & 1 & d_3 & & & & & h_3 \\
 & c_4 & b_4 & 1 & d_4 & & & & h_4 \\
\hline
 & & c_5 & b_5 & 1 & d_5 & & & h_5 \\
 & & & c_6 & b_6 & 1 & & & h_6 \\
\hline
. & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . \\
\hline
 & & & & & & c_{n-1} & b_{n-1} & h_{n-1} \\
 & & & & & & & c_n & h_n
\end{array}\right)$$

Имеет место следующий очевидный факт.

Пусть для матрицы $A = \{a_{ij}\}_{i,j=1,\dots,n}$ m – такое наименьшее число, что $a_{ij}=0$, если $|i-j| > m$. Тогда, для того, чтобы матрица A после разбиения на бло-

ки стала блочной трехдиагональной необходимо и достаточно, чтобы размер блоков был не меньше m.

МЕТОД ОКАЙМЛЕНИЯ

Решение СЛАУ с ленточной матрицей с невырожденной крайней диагональю можно свести к решению СЛАУ с треугольной ленточной матрицей, временно отбрасывая часть уравнений и решая систему относительно нескольких свободных переменных. Рассмотрим эту идею на примере трехдиагональной матрицы.

$$\left(\begin{array}{ccccccccc|c}
a_1 & c_1 & & & . & & & & h_1 \\
b_2 & a_2 & c_2 & & . & & & & h_2 \\
 & b_3 & a_3 & c_3 & . & & & & h_3 \\
 & & b_4 & a_4 & . & & & & h_4 \\
. & . & . & . & .. & . & . & . & . \\
 & & & & . & a_{n-2} & c_{n-2} & & h_{n-2} \\
 & & & & . & b_{n-1} & a_{n-1} & c_{n-1} & h_{n-1} \\
\hline
 & & & & . & & b_n & a_n & h_n
\end{array}\right) \qquad (21)$$

Отбросим (временно) последнее уравнение системы. Оставшуюся систему будем решать относительно первого неизвестного. Это все равно, что решать систему уравнений с двумя правыми частями. Матрица новой системы будет ленточной нижнетреугольной с элементами c_i по главной диагонали.

$$\left(\begin{array}{ccccccc|cc}
c_1 & & & . & & & & h_1 & -a_1 \\
a_2 & c_2 & & . & & & & h_2 & -b_2 \\
b_3 & a_3 & c_3 & . & & & & h_3 & 0 \\
 & b_4 & a_4 & . & & & & h_4 & 0 \\
. & . & . & . & . & . & . & . & . \\
 & & & . & a_{n-2} & c_{n-2} & & h_{n-2} & 0 \\
 & & & . & b_{n-1} & a_{n-1} & c_{n-1} & h_{n-1} & 0
\end{array}\right)$$

Решение этой системы имеет вид

$$\begin{array}{l}
x_2 = d_2 * x_1 + e_2 \\
x_3 = d_3 * x_1 + e_3 \\
\ldots\ldots\ldots\ldots\ldots \\
x_{n-1} = d_{n-1} * x_1 + e_{n-1} \\
x_n = d_n * x_1 + e_n
\end{array} \qquad (22)$$

Подставляя последние два равенства в отброшенное уравнение

$b_n * x_{n-1} + c_n * x_n = h_n$,

найдем x_1 . Теперь x_1 можно одновременно подставить во все равенства (**22**) и получим значения всех неизвестных.

Отбрасывая первое уравнение в системе (**21**), можно было бы получить систему с верхнетреугольной матрицей. Эту систему можно было бы разрешить аналогичным образом относительно последнего неизвестного.

$$\left(\begin{array}{ccccccc|c}
a_1 & c_1 & & & & & & h_1 \\
\hline
b_2 & a_2 & c_2 & & & & & h_2 \\
 & b_3 & a_3 & c_3 & & & & h_3 \\
 & & b_4 & a_4 & & & & h_4 \\
 & & & & & & & \\
 & & & & a_{n-2} & c_{n-2} & & h_{n-2} \\
 & & & & b_{n-1} & a_{n-1} & c_{n-1} & h_{n-1} \\
 & & & & & b_n & a_n & h_n
\end{array}\right)$$

Пусть у ленточной матрицы $A=(a_{ij})_{i,j=1,\ldots,n}$ имеется преобладание по диагонали i-j = k (>0). Тогда окаймление следует сделать, отбросив первые k строк и разрешив оставшуюся систему относительно последних k переменных. Получатся формулы, в которых первые n-k переменных выражаются линейно через последние k переменных. Подставив полученные решения в отброшенные k первых уравнений, получим систему из k уравнений относительно k последних неизвестных. Вычислив из этой системы k последних неизвестных и подставив их в полученные формулы, найдем и значения первых n-k переменных.

$$\left(\begin{array}{ccccccccccc|c}
a_{11} & a_{12} & . & a_{1r} & & & & & & & & h_1 \\
a_{21} & a_{22} & . & a_{2r} & a_{2,r+1} & & & & & & & h_2 \\
. & . & . & . & . & . & . & . & . & . & . & . \\
a_{k1} & a_{k2} & . & & & & & & & & & h_k \\
\hline
a_{k+1,1} & a_{k+1,2} & . & & & & & & & & & \\
. & . & . & . & . & . & . & . & . & . & . & . \\
a_{s1} & a_{s2} & . & & & & & & & & & h_s \\
 & a_{s+1,2} & . & & & & & & & & & \\
 & & . & & & & & & & & & \\
 & & . & & & & & & & & & \\
 & & . & & & & & & & & & \\
 & & . & & & & & a_{n-s+1,n} & . & a_{n-k,n} & a_{n-k+1,n} & . & a_{nn} & h_{nn}
\end{array}\right)$$

Если у ленточной матрицы $A=(a_{ij})_{i,j=1,\ldots,n}$ имеется преобладание по диагонали i-j = k (<0), то окаймление следует сделать, отбросив последние -k строк и разрешив оставшуюся систему относительно первых -k переменных и т.д..

Пусть теперь у ленточной матрицы $A=(a_{ij})_{i,j=1,\ldots,n}$ сначала в первых m строках имеется преобладание по диагонали i-j = u, а затем, в остальных n-m

строках имеется преобладание по диагонали i-j = v. Тогда матрицу следует разрезать на две части, в каждой из которых окаймление следует сделать, чтобы преобладающая диагональ стала главной.

Для примера рассмотрим диагональную матрицу, у которой в первых m строках преобладает поддиагональ, а в остальных n-m строках преобладает главная диагональ.

$$\left(\begin{array}{cccccccc|c}
a_1 & & & & & & & & h_1 \\
b_2 & a_2 & & & & & & & h_2 \\
 & b_3 & a_3 & & & & & & h_3 \\
. & . & . & . & . & . & . & . & . \\
 & & & b_m & a_m & & & & h_m \\
\hline
 & & & & b_{m+1} & a_{m+1} & & & h_{m+1} \\
. & . & . & . & . & . & . & . & . \\
 & & & & & & b_n & a_n & h_n
\end{array}\right)$$

Для решения этой системы следует
1) временно отбросить первое уравнение;
2) уравнения с номерами от 2 до m следует решить (параллельно) относительно переменной x_1 ;
3) найти x_1 из временно отброшенного первого уравнения и, подставив в уравнения с номерами от 2 до m получить значения неизвестных x_2 , …, x_m;
4) в уравнение с номером m+1 подставить значение переменной x_m;
5) уравнения с номерами от m+1 до n решить параллельно.

Если матрица состоит из нескольких кусков, каждый из которых имеет преобладающую диагональ, то данную матрицу следует разрезать на эти несколько кусков. Если матрицу невозможно представить в виде нескольких подматриц, в каждой из которых преобладает некоторая диагональ, то об устойчивом численном методе для такой матрицы ничего сказать нельзя.

Упражнение. Оценить сложность рассмотренных параллельных алгоритмов.

ПАРАЛЛЕЛЬНОЕ ВЫЧИСЛЕНИЕ ДРОБНОЛИНЕЙНЫХ РЕКУРРЕНТНЫХ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

Сначала следует вспомнить, что суперпозиция двух дробнолинейных отображений опять является дробнолинейным отображением.

$$g_1(x) = \frac{a_1 * x + b_1}{c_1 * x + d_1}; \quad g_2(x) = \frac{a_2 * x + b_2}{c_2 * x + d_2}$$

$$f(x) = g_2(g_1(x)) = \frac{(a_2 * a_1 + b_2 * c_1) * x + (a_2 * b_1 + b_2 * d_1)}{(c_2 * a_1 + d_2 * c_1) * x + (c_2 * b_1 + d_2 * d_1)}$$

Если коэффициенты всякого дробнолинейного отображения записывать в виде матрицы 2x2, то коэффициенты суперпозиции отображений представляют собой произведение матриц коэффициентов исходных отображений.

Пусть задана рекуррентная последовательность

$$x_n = f_{n-1}(x_{n-1}) = \frac{a_{n-1} * x_{n-1} + b_{n-1}}{c_{n-1} * x_{n-1} + d_{n-1}}$$

Обозначим матрицу коэффициентов отображения f_n

$$A_{n-1} = \begin{pmatrix} a_{n-1} & b_{n-1} \\ c_{n-1} & d_{n-1} \end{pmatrix}$$

Тогда для вычисления членов последовательности x_n достаточно вычислить все произведения матриц

$B_1 = A_1$
$B_2 = A_1 * A_2$
$B_3 = A_1 * A_2 * A_3$
……
$B_{n-1} = A_1 * A_2 * A_3 * \ldots * A_{n-1}$

(Параллельный алгоритм вычисления всех таких произведений приводился ранее). Теперь все члены последовательности x_k могут быть одновременно вычислены по элементам матриц

$$B_k = \begin{pmatrix} a'_k & b'_k \\ c'_k & d'_k \end{pmatrix}$$

по формулам

$$x_k = \frac{a'_k * x_0 + b'_k}{c'_k * x_0 + d'_k}$$

ПАРАЛЛЕЛЬНЫЙ АЛГОРИТМ LU-РАЗЛОЖЕНИЯ ТРЕХДИАГОНАЛЬНОЙ МАТРИЦЫ

Будем рассматривать задачу разложения трехдиагональной матрицы в произведение нижне- и верхнетреугольных матриц. Нижне- и верхнетреугольая матрицы будем искать, как треугольные. Это приводит к матричному уравнению

$$\begin{pmatrix} 1 & & & & \\ u_2 & 1 & & & \\ & u_3 & 1 & & \\ & & u_4 & 1 & \\ & & & . & . \\ & & & & . \end{pmatrix} * \begin{pmatrix} v_1 & w_1 & & & \\ & v_2 & w_2 & & \\ & & v_3 & w_3 & \\ & & & v_4 & . \\ & & & & . \end{pmatrix} = \begin{pmatrix} a_1 & c_1 & & & \\ b_2 & a_2 & c_2 & & \\ & b_3 & a_3 & c_3 & \\ & & b_4 & a_4 & . \\ & & & . & . \end{pmatrix}$$

Перемножив обе неизвестные скалярные матрицы, получим следующую систему уравнений

$$\begin{aligned} & u_i * v_{i-1} = b_i \\ & u_i * w_{i-1} + v_i = a_i \\ & w_i = c_i \end{aligned} \qquad (23)$$

$i = 2,\ldots n;\quad v_1 = a_1 ;\ w_1 = c_1 .$

Из условий (23) вытекает рекуррентная формула для вычисления v_i:

$$v_i = a_i - u_i * w_{i-1} = a_i - (b_i / v_{i-1}) * c_{i-1} = (a_i * v_{i-1} - b_i * c_{i-1}) / v_{i-1}$$

Таким образом, вычисление массива v_i сводится к уже описанному выше параллельному алгоритму вычисления дробнолинейных рекуррентных последовательностей.

РАСПАРАЛЛЕЛИВАНИЕ РЕКУРРЕНТНЫХ ЦИКЛОВ

Всякое вычисление членов рекуррентной последовательности

$$x_n = f(x_{n-1}, x_{n-2}, \ldots, x_{n-k}) \qquad (24)$$

может быть описано с помощью программного цикла

```
      DO 99 I = k+1, N
99    X(I) = f(X(I-1), X(I-2),…, X(I-k))
```

Если функция f линейна, то и рекуррентную последовательность и рекуррентный цикл называют линейными.

Итак, линейный рекуррентный цикл имеет вид

```
DO 99 I = k+1, N
X(I) = A1(I)*X(I-1)+A2(I)*X(I-2)+…+Ak(I)*X(I-k) +A0(I)
```

Выполнение этого цикла равносильно решению системы линейных уравнений с ленточной треугольной матрицей

$$\left(\begin{array}{ccccccc|c}
1 & 0 & . & 0 & 0 & 0 & . \; . & X(1) \\
0 & 1 & . & 0 & 0 & 0 & . \; . & X(2) \\
. & . & . & . & . & . & . \; . & . \\
0 & 0 & . & 1 & 0 & 0 & . \; . & X(k) \\
A_1(k+1) & A_2(k+1) & . & A_{k-1}(k+1) & -1 & 0 & . \; . & A_0(k+1) \\
0 & A_1(k+2) & . & A_{k-2}(k+2) & A_{k-1}(k+2) & -1 & . \; . & A_0(k+2) \\
. & . & . & . & . & . & . \; . & . \\
. & . & . & . & . & . & . \; . & .
\end{array}\right)$$

В формуле (**24**) если x_n является вектором, а f – линейное отображение, то в теле соответствующего программного цикла будет несколько операторов присваивания.

В следующем примере по линейной рекуррентной формуле вычисляется пара массивов.

Пример 134.

```
      DO 99 I = 2, N
      X(I) = A(I)*X(I-1)+B(I)*Y(I-1)+C(I)
      Y(I) = D(I)*X(I-1)+E(I)*Y(I-1)+F(I)                    (25)
99    CONTINUE
```

Выполнение такого цикла равносильно решению СЛАУ с блочной двухдиагональной матрицей.

$$\left(\begin{array}{cc|cc|cc|cc|c}
1 & 0 & & & & & & & X(1) \\
0 & 1 & & & & & & & Y(1) \\
\hline
A(2) & B(2) & -1 & 0 & & & & & C(2) \\
D(2) & E(2) & 0 & -1 & & & & & F(2) \\
\hline
 & & A(3) & B(3) & -1 & 0 & & & C(3) \\
 & & D(3) & E(3) & 0 & -1 & & & F(3) \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\hline
 & & & & & & A(N) & B(N) & -1 & 0 & C(N) \\
 & & & & & & C(N) & E(N) & 0 & -1 & F(N)
\end{array}\right)$$

Конец примера.

Пример 135. Выполнение следующего цикла равносильно решению СЛАУ с треугольной ленточной матрицей (вообще говоря, не блочной двухдиагональной).

```
      DO 99 I = 2, N
      X(I) = A0(I)*X(I)+A1(I)*X(I-1)+…+Ak(I)*X(I-k)+
      +B0(I)*Y(I)+B1(I)*Y(I-1)+   +Bk(I)*Y(I-k)+C(I)
      Y(I) = D0(I)*X(I)+D1(I)*X(I-1)+…+Dk(I)*X(I-k)+
      +E0(I)*Y(I)+E1(I)*Y(I-1)+   +Ek(I)*Y(I-k)+F(I)
99    CONTINUE
```

Конец примера.

Аналогичное утверждение можно сформулировать и для циклов с линейной рекуррентностью с большим количеством операторов присваивания.

Пример 136. В следующем цикле рекуррентно вычисляются не массивы, а простые переменные.

```
      DO 99 I = 1, N
      X = A(I)*X+B(I)*Y+C(I)
      Y = D(I)*X+E(I)*Y+F(I)
99    CONTINUE
```

Такие переменные могут быть заменены массивами, после чего этот цикл становиться равносильным следующему фрагменту программы

```
XX(1) = X
YY(1) = Y
      DO 99 I = 2, N+1
      X(I) = A(I)*X(I-1)+B(I)*Y(I-1)+C(I)
      Y(I) = D(I)*X(I)+E(I)*Y(I-1)+F(I)
```

```
99      CONTINUE
X = XX(N+1)
Y = YY(N+1)
```

Если в теле цикла этого фрагмента выполнить подстановку вперед: во втором операторе присваивания вместо X(I) подставить правую часть первого оператора присваивания этого цикла – то цикл примет вид (**25**).
Конец примера.

ЦИКЛЫ С НЕЛИНЕЙНОЙ РЕКУРРЕНТНОСТЬЮ

Пусть f – нелинейная обратимая (относительно суперпозиции) функция одного переменного. Тогда в следующем цикле рекуррентная зависимость не линейная.

```
        DO 99 I = k+1, N                                                    (26)
99      X(I) = f^-1(A1(I)*f(X(I-1))+A2(I)*f(X(I-2))+…+Ak(I)*f(X(I-k)) +A0(I))
```

Несложно заметить, что элементы массива Y(I) = f(X(I)), I = 1,2,…N образуют последовательность с линейной рекуррентной зависимостью. Такая замена переменных сводит распараллеливание этого цикла к предыдущему случаю.

```
        DO 88 I = 1, k
88      Y(I) = f(X(I))
        DO 99 I = k+1, N
99      Y(I) = A1(I)*Y(I-1)+A2(I)*Y(I-2)+…+Ak(I)*Y(I-k) +A0(I)
        DO 77 I = k+1, N
77      X(I) = f^-1(Y(I))
```

Пример 137. Рекуррентный цикл

```
        DO 99 I = 3, N
99      X(I) =exp( A1(I)*ln(X(I-1))+A2(I)*ln(X(I-2))+A0(I) )
```

равносилен следующему фрагменту программы

```
        Y(1) = ln(X(1))
        Y(2) = ln(X(2))
        DO 99 I = 3, N
99      Y(I) = A1(I)*Y(I-1)+A2(I)*Y(I-2)+…+Ak(I)*Y(I-k) +A0(I)
        DO 77 I = 3, N
77      X(I) = exp(Y(I))
```

Пример 138. Рекуррентный цикл

```
        DO 99 I = 3, N
```

99 $X(I) = \sqrt{A(I)*X(I-1)^2 + B(I)*X(I-2)^2 + 1/I}$

равносилен следующему фрагменту программы

```
      Y(1) = X(1)**2
      Y(2) = X(2)**2
      DO 99 I = 3, N
99    Y(I) = A(I)*Y(I-1)+B(I)*Y(I-2)+1/I
      DO 77 I = 3, N
77    X(I) = SQRT(Y(I))
```

РЕКУРРЕНТНЫЕ ЦИКЛЫ, НЕ ИМЕЮЩИЕ ЭФФЕКТИВНОГО РАСПАРАЛЛЕЛЛИВАНИЯ.

В формулах (**24**) вместо x_{n-1} одновременно для всех n можно подставить

$$x_{n-1} = f(x_{n-2} , x_{n-3} ,\ldots, x_{n-k-1}) \ . \qquad (27)$$

Получиться

$$x_n = f(f(x_{n-2} , x_{n-3} ,\ldots, x_{n-k-1}) , x_{n-2} ,\ldots, x_{n-k}) \qquad (28)$$

Теперь, используя обе формулы (28) и (27), можно одновременно вычислять x_n и x_{n-1} . Казалось бы, этим самым, процесс вычисления рекуррентной последовательности ускоряется вдвое. Но такое ускорение возможно лишь тогда, когда время вычисления функции (28) такое же, как и (27). Непосредственное вычисление (28) ровно вдвое дольше вычисления (27) и сводит на нет подстановку. Если в (28) можно провести преобразования и получить в правой части функцию вида (27), как это происходит для линейных рекуррентных зависимостей, то желаемое ускорение было бы достигнуто, исключая затраты на преобразования (28) к виду (27). Это возможно для линейных и описанных выше дробно-линейных функций, а также для циклов вида (**26**).

Следующий рекуррентный цикл нельзя эффективно распараллелить, (если функцию cos(x) не заменить более удобной интерполирующей функцией)

```
      DO 99 I = 2, N
99    X(I) = cos(X(I-1))
```

В работе [**41**] исследуется возможность распараллеливания рекуррентных циклов, содержащих условный оператор.

РАСПАРАЛЛЕЛИВАНИЕ АЛГОРИТМОВ РЕШЕНИЯ СЛАУ ДЛЯ КОМПЬЮТЕРОВ С СЕГМЕНТИРОВАННОЙ ПАМЯТЬЮ

Можно выделить два типа суперкомпьютеров по условиям доступа к сегментированной памяти: суперкомпьютеры с распределенной памятью и суперкомпьютеры с перестраиваемой архитектурой (иногда они называются суперкомпьютерами со структурной реализацией вычислений).

Особенность суперкомпьютеров с распределенной памятью состоит в том, что у каждого процессора имеется свой сегмент памяти, из которого получать данные значительно быстрее, чем из сегментов памяти других процессоров. Такие многопроцессорные вычислительные системы могут быть как MIMD, так и SIMD. Обычно в таких компьютерах выполнение арифметических операций намного быстрее межпроцессорных пересылок данных (в транспьютерных системах, в кластерах). В такой ситуации для эффективной работы суперкомпьютер должен иметь “не очень много” процессоров. Для уменьшения пересылок данных аргументы одной операции выгодно размещать в памяти одного и того же процессора.

У суперкомпьютеров с перестраиваемой архитектурой для каждого процессора все модули памяти “одинаково чужие”, в том смысле, что из всех модулей одинаково долго получать данные. Этот недостаток компенсируется возможностью пересылок данных из одного процессора непосредственно в другой, минуя память (обработка данных по программно настроенному конвейеру) и возможностью одновременного считывания сразу двух аргументов в процессор. Последняя особенность архитектуры делает выгодным размещение аргументов одной операции в разных сегментах памяти (в отличие от предыдущего случая).

Рассмотрим сначала задачу распараллеливания алгоритма решения СЛАУ с двухдиагональной матрицей.

Фрагмент программы (**18**) можно заменить эквивалентным следующим фрагментом

```
      do  99  k = 1, s
      do  99   I = 2**(k-1)+1, N
      h(I)= h(I) - b(I)*h(I-2**(k-1))
99    b(I) = - b(I)*b(I-2**(k-1))
      do  88  m = 1, 2**s
      do  88   I = 2**s+1, N
88    h(I+m-1)= h(I+m-1) - b(I+m-1)*h(I+m-1-2**s)
```

В этом фрагменте во втором гнезде циклов объемлющий цикл распараллеливается. Этот цикл может параллельно выполняться на суперкомпьютерах с архитектурой с распределенной памятью (MIMD или SIMD) и на суперкомпьютере со структурной организацией вычислений.

В случае распределенной памяти данные могут быть размещены следующим образом: элементы массивов h(j) и b(j) должны находиться в процессорном элементе с номером (j mod (2**s)). Тогда второе гнездо циклов исполняет-

ся без пересылок данных, хотя в первом гнезде циклов пересылки данных неизбежны.

В случае суперкомпьютера с архитектурой перестраиваемого конвейера данные можно разместить следующим образом: элементы массива h(j) должны находиться в модуле памяти с номером 2*(j mod (2**s)), а элементы массива b(j) должны находиться в модуле памяти с номером 2*(j mod (2**s))+1. Тогда при достаточном количестве ресурсов суперкомпьютера второе гнездо циклов эффективно отображается на архитектуру суперкомпьютера.

Теперь рассмотрим решение СЛАУ с трехдиагональной (блочной) матрицей (**19**).

Заметим, что после первого преобразования, получается система уравнений (**20**), которая распадается на две независимые системы уравнений: нечетные неизвестные входят только в уравнения с нечетными номерами, а четные неизвестные входят в уравнения с четными номерами. Если у параллельного компьютера имеется лишь два процессорных элемента, то каждому из них можно поручить решать по одной из полученных систем:

$$\left(\begin{array}{cccccc|c} 1 & \overline{C}_1^2 & & & & & \overline{H}_1^2 \\ \overline{B}_3^2 & 1 & \overline{C}_3^2 & & & & \overline{H}_3^2 \\ & \overline{B}_5^2 & 1 & \overline{C}_5^2 & & & \overline{H}_5^2 \\ & & \overline{B}_7^2 & 1 & \overline{C}_7^2 & & \overline{H}_7^2 \\ . & . & . & . & . & . & . \\ & & & & \overline{B}_{n/2-1}^2 & 1 & \overline{H}_{n/2-1}^2 \end{array}\right)$$

$$\left(\begin{array}{cccccc|c} \mathbf{1} & \overline{\mathbf{C}}_2^{\mathbf{2}} & & & & & \overline{\mathbf{H}}_2^{\mathbf{2}} \\ \overline{\mathbf{B}}_4^{\mathbf{2}} & \mathbf{1} & \overline{\mathbf{C}}_4^{\mathbf{2}} & & & & \overline{\mathbf{H}}_4^{\mathbf{2}} \\ & \overline{\mathbf{B}}_6^{\mathbf{2}} & \mathbf{1} & \overline{\mathbf{C}}_6^{\mathbf{2}} & & & \overline{\mathbf{H}}_6^{\mathbf{2}} \\ & & \overline{\mathbf{B}}_8^{\mathbf{2}} & \mathbf{1} & \overline{\mathbf{C}}_8^{\mathbf{2}} & & \overline{\mathbf{H}}_8^{\mathbf{2}} \\ \mathbf{.} & \mathbf{.} & \mathbf{.} & \mathbf{.} & \mathbf{.} & \mathbf{.} & \mathbf{.} \\ & & & & \overline{\mathbf{B}}_{\mathbf{n/2}}^{\mathbf{2}} & \mathbf{1} & \overline{\mathbf{H}}_{\mathbf{n/2}}^{\mathbf{2}} \end{array}\right)$$

После второго шага преобразований система (3) может быть представлена в виде четырех независимых систем уравнений. … После k-ого шага преобразований – в виде 2**k независимых систем уравнений.

Поскольку СЛАУ с ленточными матрицами сводятся к блочным трехдиагональным (или к частному случаю – двухдиагональным), следовательно такие задачи вполне успешно отображаются на архитектуру суперкомпьютера со структурной реализацией вычислений.

ОЦЕНКИ ПОГРЕШНОСТЕЙ РЕШЕНИЯ СЛАУ С ДВУХДИАГОНАЛЬНОЙ МАТРИЦЕЙ

Исходные данные системы с двухдиагональной матрицей (**17**) – b_i и h_i – заданы с некоторой точностью. Обозначим через $\underline{b}_i$ и $\underline{h}_i$ – точные значения этих данных. И пусть $\varepsilon_i = b_i - \underline{b}_i$ и $\delta_i = h_i - \underline{h}_i$. Обозначим $b = \max |b_i|$, $h = \max |h_i|$, $\varepsilon = \max |\varepsilon_i|$, $\delta = \max |\delta_i|$. Далее будем оценивать погрешность решения системы через эти исходные данные при условии $b<1$.

В описанном выше алгоритме матрица системы преобразуется и на каждом шаге новые данные пишутся вместо старых. Чтобы зафиксировать все промежуточные матрицы, заменим одномерные массивы двумерными. После этого алгоритм решения будет выглядеть следующим образом.

```
      do  99  k = 1, ]log(N)[
      do  99   I = 2**(k-1)+1, N
      h(k,I)= h(k-1,I) - b(k-1,I)*h(k-1, I-2**(k-1))
99    b(k,I) = - b(k-1,I)*b(k-1,I-2**(k-1))
```

Здесь каждому значению параметра k соответствует шаг алгоритма и соответствующая матрица системы.

Обозначим элементы матрицы k-ой системы - $b_i(k)$ и $h_i(k)$; $\underline{b}_i(k)$ и $\underline{h}_i(k)$ – их точные значения; $\varepsilon_i(k) = b_i(k) - \underline{b}_i(k)$ и $\delta_i(k) = h_i(k) - \underline{h}_i(k)$. Обозначим $b(k) = \max |b_i(k)|$, $h(k) = \max |h_i(k)|$, $\varepsilon(k) = \max |\varepsilon_i(k)|$, $\delta(k) = \max |\delta_i(k)|$.

Ввиду формулы b(k,I) = - b(k-1,I)*b(k-1,I-2**(k-1)) можно записать

$$b(k) \le b(k-1)^2 \le b^{2^{k-1}}$$

Из формулы h(k,I)= h(k-1,I) - b(k-1,I)*h(k-1, I-2**(k-1)) вытекает оценка

$$h(k) \le h(k-1) + b(k-1)*h(k-1) = h(k-1)*(1+b(k-1))$$
$$\le h*(1+b(k-1))*(1+b(k-2))*...*(1+b) \le$$

$$\le h*(1+b^{2^{k-2}})*(1+b^{2^{k-3}})*...*(1+b) \le$$

$$h*(1+b+b^2+b^3+...+b^{2^{k-1}-1}) = h*\frac{1-b^{2^{k-1}}}{1-b}$$

При оценках погрешностей, как обычно, будем пренебрегать слагаемыми, в которые входят погрешности исходных данных в квадрате и более высоких степенях.

Из формулы b(k,I) = - b(k-1,I)*b(k-1,I-2**(k-1)) можно записать неравенство

$$\varepsilon(\kappa) = 2*b(\kappa-1)*\varepsilon(\kappa-1) \le 2^{k-1}*b(\kappa-1)*b(\kappa-2)*...*b(1)*\varepsilon$$
$$\le 2^{k-1}*b^{2^{k-2}}*b^{2^{k-3}}*...*b*\varepsilon = 2^{k-1}*b^{2^{k-1}-1}*\varepsilon$$

Из формулы h(k,I)= h(k-1,I) - b(k-1,I)*h(k-1, I-2**(k-1)) вытекает следующая оценка

$$\delta(k) \le \delta(k-1) * (1 + b(k-1)) + \varepsilon(k-1) * h(k-1)$$
$$\le \delta(k-1) * (1 + b(k-1)) + 2^{k-1} * b^{2^{k-2}-1} * \varepsilon * \mathrm{h} * \frac{1 - b^{2^{k-2}}}{1 - b}$$

Запишем следующую лемму, которая очевидным образом доказывается методом математической индукции.

Лемма 4. Пусть имеется система неравенств

$x_k \le u_{k-1} * x_{k-1} + v_{k-1}$ $k = 2,3,\ldots$

где все числа u_i, v_i, $i = 1,2,\ldots$ и x_1 положительны. Тогда для любого $k = 2,3,\ldots$ выполняется неравенство

$x_k \le u_{k-1} * u_{k-2} * \ldots * u_1 * x_1 + u_{k-1} * u_{k-2} * \ldots * u_2 * v_1 + u_{k-1} * u_{k-2} * \ldots * u_3 * v_2 + \ldots + u_{k-1} * v_{k-2} + v_{k-1}$.

Используя эту лемму, получим оценку погрешности решения системы

$$\delta(k) \le \delta(k-1) * (1 + b(k-1)) + \varepsilon(k-1) * h(k-1)$$
$$\le (1 + b(k-1)) * (1 + b(k-2)) * \ldots * (1 + b(1)) * \delta +$$
$$+ (1 + b(k-1)) * (1 + b(k-2)) * \ldots * (1 + b(2)) * \varepsilon(1) * h(1) +$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$+ (1 + b(k-1)) * \varepsilon(k-2) * h(k-2) + \varepsilon(k-1) * h(k-1) \le$$

$$\le (1 + b^{2^{k-2}}) * (1 + b^{2^{k-3}}) * \ldots * (1 + b) * \delta +$$
$$+ (1 + b^{2^{k-2}}) * (1 + b^{2^{k-3}}) * \ldots * (1 + b^2) * \varepsilon(1) * h(1) +$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$+ (1 + b^{2^{k-2}}) * \varepsilon(k-2) * h(k-2) + \varepsilon(k-1) * h(k-1) \le$$

$$\le (1 + b + b^2 + b^3 + \ldots + b^{2^{k-1}-1}) * \delta +$$
$$+ (1 + b^2 + b^4 + b^6 \ldots + b^{2^{k-1}-2}) * \varepsilon(1) * h(1) +$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$+ (1 + b^{2^{k-2}}) * \varepsilon(k-2) * h(k-2) + \varepsilon(k-1) * h(k-1) \le$$

$$\le \frac{1-b^{2^{k-1}}}{1-b}*\delta+\frac{1-b^{2^{k-1}}}{1-b^2}*\varepsilon*h+$$

.....

$$...+\frac{1-b^{2^{k-1}}}{1-b^{2^s}}*(2^{s-1}*b^{2^{s-1}-1}*\varepsilon)*(h*\frac{1-b^{2^{s-1}}}{1-b})+...$$

.....

$$+\frac{1-b^{2^{k-1}}}{1-b^{2^{k-2}}}*(2^{k-3}*b^{2^{k-3}-1}*\varepsilon)*(h*\frac{1-b^{2^{k-3}}}{1-b})$$

$$+(2^{k-2}*b^{2^{k-2}-1}*\varepsilon)*(h*\frac{1-b^{2^{k-2}}}{1-b})$$

$$=\frac{1-b^{2^{k-1}}}{1-b}*\delta+\frac{1-b^{2^{k-1}}}{1-b}*\varepsilon*h*\sum_{s=1}^{k-1}2^{s-1}*b^{2^{s-1}-1}*\frac{1-b^{2^{s-1}}}{1-b^{2^s}}=$$

$$=\frac{1-b^{2^{k-1}}}{1-b}*\delta+\frac{1-b^{2^{k-1}}}{1-b}*\varepsilon*h*\sum_{s=1}^{k-1}2^{s-1}*b^{2^{s-1}-1}*\frac{1}{1+b^{2^{s-1}}}$$

Заметим, что при b<1 выполняются

$$\frac{1}{1+b^{2^{s-1}}}\le\frac{1}{2};$$

$$\sum_{s=1}^{k-1}2^{s-1}*b^{2^{s-1}-1}=1+\frac{2}{b}*(b^2+2*b^4+4*b^8+...+2^{k-3}*b^{2^{k-2}})=$$

$$=1+\frac{2}{b}*(b^2+b^4+b^4+b^8+b^8+b^8+b^8+...+b^{2^{k-2}})\le$$

$$\le 1+\frac{2}{b}*(b^2+b^3+b^4+b^5+b^6+b^7+b^8+...+b^{2^{k-2}})\le$$

$$\le 1+2*b*\frac{1-b^{2^{k-2}-1}}{1-b}$$

Это позволяет продолжить основную цепочку неравенств следующим образом

$$\delta(k) \le \frac{1-b^{2^{k-1}}}{1-b} * \delta + \frac{1-b^{2^{k-1}}}{1-b} * \varepsilon * h * \frac{1}{2}(1 + 2 * b * \frac{1-b^{2^{k-2}-1}}{1-b}) \le$$
$$\frac{1-b^{2^{k-1}}}{1-b} * (\delta + \varepsilon * h * \frac{1}{2}(1 + 2 * \frac{b-b^{2^{k-2}}}{1-b})) \le$$

Вспоминая, что n=2**k, получаем

$$\delta(k) \le \frac{1-b^{\frac{n}{2}}}{1-b} * (\delta + \varepsilon * h * \frac{1}{2}(1 + 2 * \frac{b-b^{\frac{n}{4}}}{1-b})) \le$$
$$\frac{1}{1-b} * (\delta + \varepsilon * h * \frac{1}{2}(1 + 2 * \frac{b}{1-b}))$$

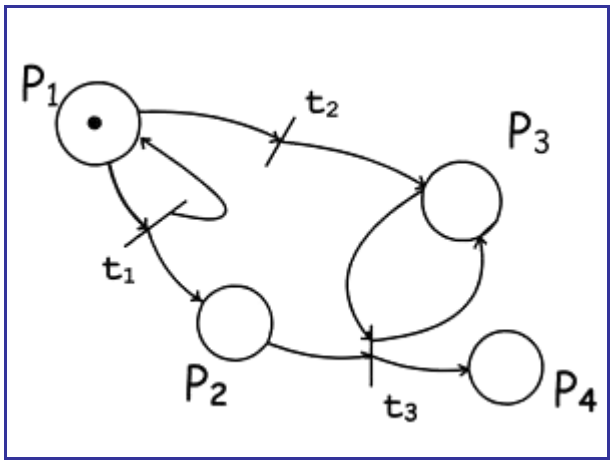
Следует отметить, что при b>1 проведенные оценки показывают, что оценка погрешности растет, как экспонента от размерности матрицы. В этом случае данным алгоритмом пользоваться нецелесообразно, т.к. при относительно небольших размерностях матриц решению системы нельзя доверять.

СЕТИ ПЕТРИ

Определение сети Петри

Сети Петри – это математический формализм, придуманный Карлом Петри для описания действий роботов. В последствии выяснилось, что сети Петри удобны для моделирования деятельности многих других сложных систем, в частности, параллельных вычислительных машин.

Сеть Петри – это двудольный ориентированный граф со специальной разметкой и правилом изменения этой разметки.



Анимация. 17. Пример сети Петри

Вершины одного типа называются местами, а другого типа – переходами. Каждое место имеет в качестве метки целое неотрицательное число. Обычно места изображаются кружочками, а переходы – отрезками. Метка места изображается внутри кружочка числом или количеством точек, соответствующим этому натуральному числу. Иногда метки называют фишками или маркерами (это синонимы).

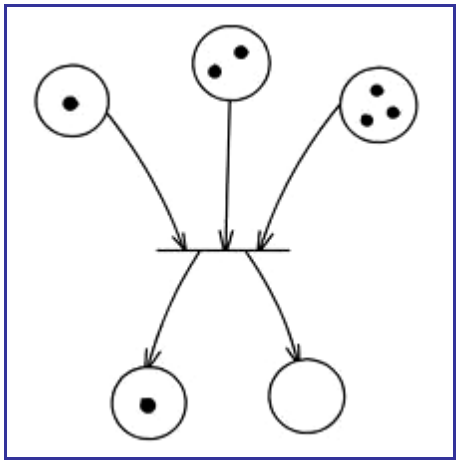
Если из места в переход ведет дуга, то это место называется входным по отношению к данному переходу. Если из перехода в место ведет дуга – то такое место называется выходным.

Переходы сети Петри могут срабатывать.

Условие срабатывания перехода – каждое его входное место должно иметь хотя бы одну фишку.

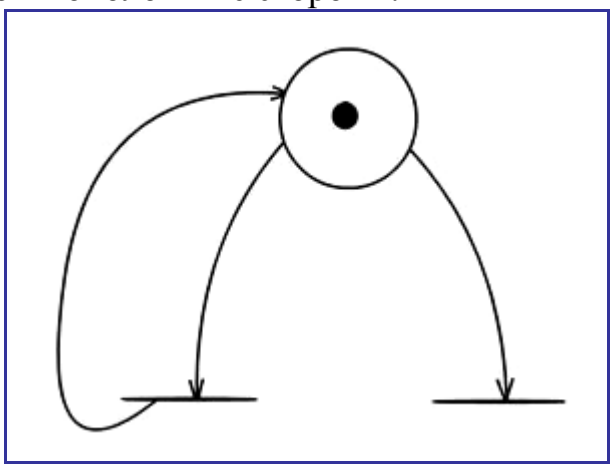
Правило срабатывания перехода – из каждого входного места удаляется по одной фишке и в каждое выходное место добавляется по одной фишке.

Для сети Петри не выполняется закон сохранения количества фишек. Фишки не являются чем-то материальным. При срабатывании переходов количество фишек может как сохраняться, так и уменьшаться или увеличиваться.



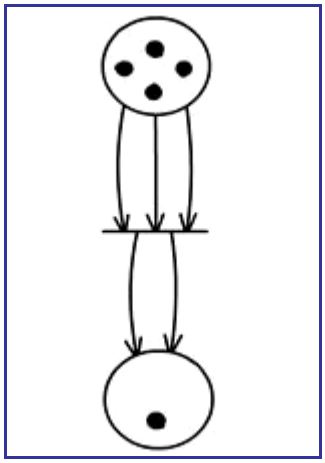
Анимация. 18. Иллюстрация к правилу срабатывания переходов

Одно и то же место может быть одновременно и входным и выходным для одного и того же перехода, если место и переход связаны двумя дугами, направленными в противоположные стороны.



Анимация. 19. Место является входным и выходным для перехода

Могут использоваться кратные дуги. Если из одного места в переход идет несколько (k) дуг, то для срабатывания перехода необходимо, чтобы в этом месте было не менее k фишек. Если из перехода в одно место ведут m дуг, по после срабатывания перехода в этом месте добавляется m фишек.



Анимация. 20. Сеть с кратными дугами

Упражнение. Сформулируйте условие срабатывания переходов для сети с кратными дугами.

Если выполнены условия срабатывания двух или нескольких переходов, то любой из них может сработать, т.е. срабатывания происходят недетерминировано.

Для упреждения многих ошибок начального использования сетей Петри отметим: в теории сетей Петри нет понятия времени! В частности, нет понятия одновременного (параллельного) срабатывания переходов. Сети Петри – это средство описания ограничений на последовательность выполнения разных процессов.

Хранить сеть Петри в памяти компьютера можно обычными способами, которыми в памяти храняться графы. Приведем два наиболее удобных таких способа.

Как и всякий двудольный граф сеть Петри можно хранить с помощью двух прямоугольных таблиц смежностей. Одна МЕСТА-ПЕРЕХОДЫ показывает наличие дуг из мест в переходы, а другая – ПЕРЕХОДЫ-МЕСТА – наличие дуг из переходов в места.

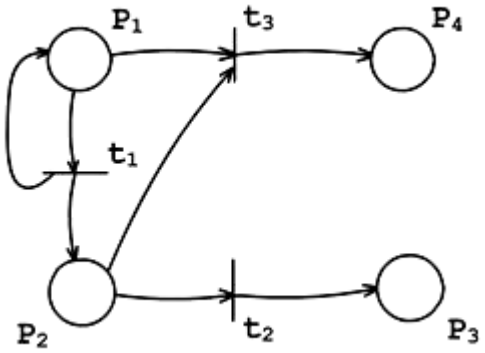


Рис. 21. Сеть Петри

	t_1	t_2	t_3
p_1	1	0	1
p_2	0	2	1
p_3	0	0	0
p_4	0	0	0

Рис. 22. таблица МЕСТА-ПЕРЕХОДЫ

	p_1	p_2	p_3	p_4
t_1	1	1	0	0
t_2	0	0	1	0
t_3	0	0	0	1

Рис. 23. таблица ПЕРЕХОДЫ-МЕСТА

Другой способ – это два списка дуг, которые тоже можно назвать соответственно также МЕСТА-ПЕРЕХОДЫ и ПЕРЕХОДЫ-МЕСТА.

(p_1 , t_1)
(p_1 , t_3)
(p_2 , t_2)
(p_2 , t_2)
(p_2 , t_3)

Рис. 24. список МЕСТА-ПЕРЕХОДЫ

(t_1 , p_1)
(t_1 , p_2)
(t_2 , p_3)
(t_3 , p_4)

Рис. 25. список ПЕРЕХОДЫ-МЕСТА

Начальную разметку сети Петри можно хранить в виде массива целых чисел, длина которого равна количеству мест.

Граф разметок

Граф разметок – это ориентированный граф, вершинами которого являются все достижимые разметки. Дуга соединяет две вершины, если из разметки, соответствующей началу дуги срабатыванием одного перехода может быть получена разметка, соответствующая концу дуги.

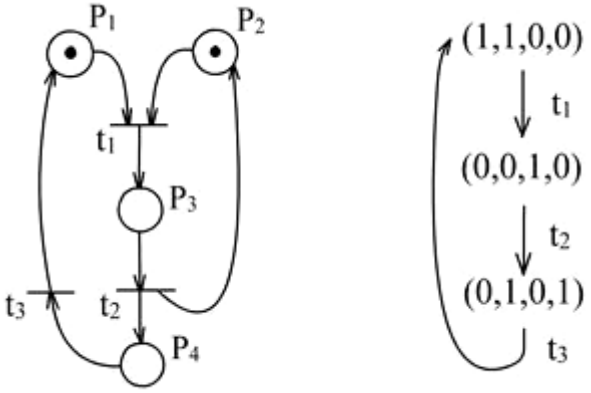


Рис. 26. Сеть Петри с конечным графом достижимых разметок

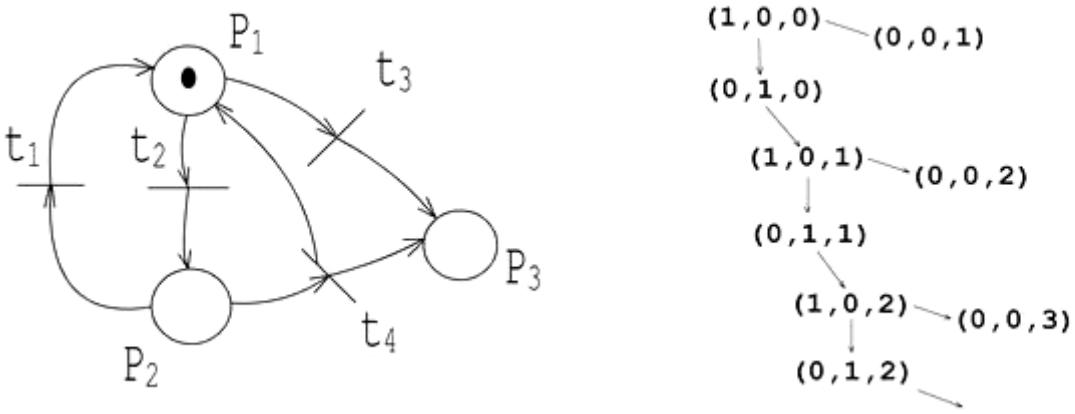


Рис. 27. Сеть Петри с бесконечным графом достижимых разметок

Свойства и виды сетей Петри

Граф разметок сети Петри – это ориентированный граф, вершинами которого являются все достижимые разметки сети. Дуга соединяет две разметки, если вторая разметка может быть получена из первой срабатыванием одного перехода.

Граф разметок может быть бесконечным.

Разметка называется тупиковой, если при ней ни один переход не может сработать.

Следует отметить, что тупиковой разметке на графе разметок соответствует сток (вершина, из которой не выходит ни одна дуга).

Место в сети Петри называется ограниченным, если существует такое число d, что для любой достижимой разметки количество фишек данного места не может превысить d. Если это свойство не выполняется, то место называется неограниченным.

Сеть называется ограниченной, если любое место ограничено. В противоположном случае – сеть неограниченна.

Теорема 40. Множество достижимых разметок конечно тогда и только тогда, когда сеть ограничена.

Доказательство очевидно.

Место называется безопасным, если количество фишек на этом месте не более 1. Сеть безопасна, если все ее места безопасны.

Любая достижимая в безопасной сети разметка представляет собой вектор из 0 и 1.

Сеть называется консервативной, если сумма всех фишек в сети постоянна при любой разметке.

Ясно, что консервативная сеть ограничена.

Переход называется потенциально живым, если существует достижимая разметка, при которой этот переход может сработать.

Переход мертвый, если он не может сработать ни при одной достижимой разметке. То есть, переход мертвый, если он не является потенциально живым.

Переход в сети называется живым, если он является потенциально живым для любой достижимой разметки.

Переход называется потенциально мертвым, если существует достижимая разметка, в которой этот переход является мертвым.

Сеть называется живой, если все ее переходы живы.

Проблема ограниченности состоит в возможности проверки, правда ли, что все переходы сети ограничены.

Бесконечные метки

Расширим множество целых неотрицательных чисел N символом w, означающим бесконечность. Отметим свойства этого символа:

1. для любого числа $r \in N$ выполняется $r < w$.
2. для любого числа $r \in N$ выполняется $r + w = w$.
3. для любого числа $r \in N$ выполняется $w - r = w$.

Множество натуральных чисел, расширенное символом w будем обозначать $N_w = N \cup \{w\}$. Отмеченные свойства N_w позволяют помечать места сети Петри символами из этого множества.

На множестве векторов с координатами из множества N_w введем покоординатные отношения $\{<, \leq, =\}$.

Пусть $a = (a_1, a_2, \dots, a_n)$, $b = (b_1, b_2, \dots, b_n)$.

$a = b$ тогда и только тогда, когда для любого $i=1,\dots,n$ выполняется $a_i = b_i$.

$a \leq b$ тогда и только тогда, когда для любого $i=1,\dots,n$ выполняется $a_i \leq b_i$.

$a < b$ тогда и только тогда, когда для любого $i=1,\dots,n$ выполняется $a_i \leq b_i$ и существует такое j, что $a_j < b_j$.

Покрывающее дерево сети Петри

Граф разметок сети Петри не удобен для автоматического анализа, поскольку он может оказаться бесконечным. Удобным для анализа является покрывающее дерево, которое определяется следующим алгоритмом построения.

Покрывающее дерево – это ориентированное исходящее дерево, вершины и дуги которого строятся рекурсивно следующим образом.

Вершины дерева – это некоторые достижимые разметки, включая бесконечные. Начальная разметка принадлежит покрывающему дереву и является его корнем. Вначале алгоритма корень считается непросмотренной вершиной.

Пусть M – текущая (непросмотренная) вершина дерева. Рассмотрим четыре возможных случая.

А) Ни один из переходов в сети не может сработать при разметке M. В этом случае вершина M объявляется листом.

Б) На пути из корня дерева в вершину M существует вершина M’ такая, что M = M’ . В этом случае вершина M объявляется листом.

В) На пути из корня дерева в вершину M существует вершина M’ такая, что M’ < M. Для любого места p такого, что M’(p) < M(p), значение соответствующей координаты M заменяется на w и вершина M объявляется w-листом.

Г) Иначе M внутренняя вершина дерева. Для каждого перехода t, который может сработать при разметке M добавляется новая вершина M’ – разметка, полученная из M срабатыванием t и добавляется дуга (M, M’).

Вершина M объявляется просмотренной.

Построенный граф является деревом. Действительно, дерево, это связный (в неориентированном смысле) граф без циклов. Связность вытекает из того, что в каждую новую вершину строится дуга из старой, в которую есть путь от корня. Поскольку в каждую новую вершину строится только одна дуга, количество вершин (включая корень) на единицу больше количества дуг. Из этого свойства и связности вытекает отсутствие циклов в построенном графе.

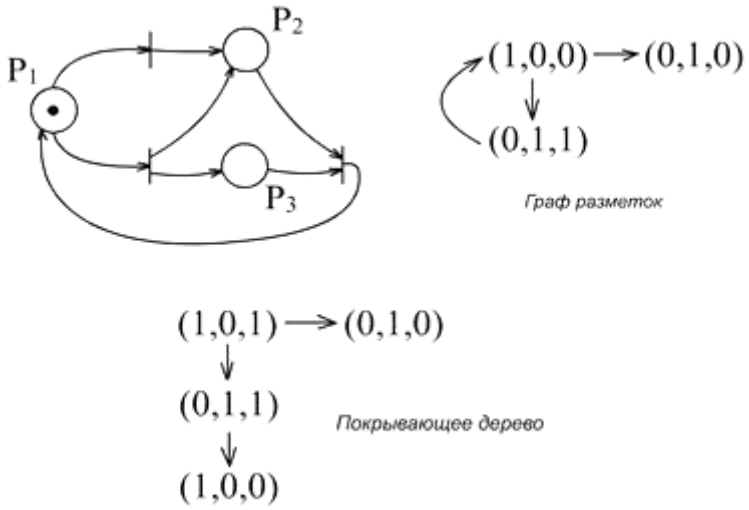


Рис. 28. Сеть Петри с графом достижимых разметок и покрывающим деревом

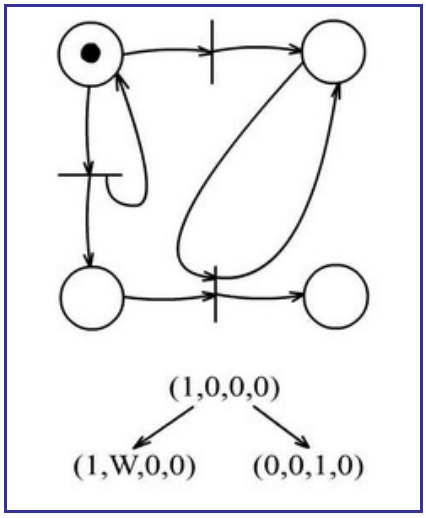


Рис. 29. Сеть Петри с графом достижимых разметок и покрывающим деревом, содержащим w-лист

Теперь следует доказать конечность покрывающего дерева. Для этого понадобятся вспомогательные утверждения.

Лемма. Множество векторов, которые по каждой координате не превосходят вектора $(b_1, b_2, \ldots, b_n)$ конечно. Более точно, это множество содержит $(b_1+1)*(b_2+1)* \ldots *(b_n+1)$ элементов.

Доказательство очевидно.

Лемма. Любая бесконечная последовательность векторов из множества $(N_w)^n = N_w*N_w*\ldots*N_w$ (n сомножителей) содержит бесконечную неубывающую подпоследовательность.

Доказательство. Пусть $a(1)$, $a(2)$,…, $a(m)$,… - произвольная последовательность векторов из множества $(N_w)^n$. Рассмотрим последовательность первых координат этой последовательности $a(1)_1$, $a(2)_1$,…, $a(m)_1$,… Если эта последовательность ограничена, то можно найти ее постоянную подпоследовательность, т.е. подпоследовательность $a(i_1)_1 = a(i_2)_1 = \ldots = a(i_s)_1 = \ldots$, состоящую из одинаковых чисел. Если рассматриваемая последовательность первых координат неограниченна, то из этой последовательности можно выделить строго возрастающую подпоследовательность $a(i_1)_1 < a(i_2)_1 < \ldots < a(i_s)_1 < \ldots$. В любом случае, из исходной последовательности можно выделить подпоследовательность, у которой первые координаты неубывают.

Для последовательности $a(i_1)$, $a(i_2)$, … $a(i_s)$, … рассмотрим последовательность вторых координат и выберем из нее неубывающую (по второй координате) подпоследовательность. Продолжая этот процесс получим подпоследовательность исходной последовательности, которая неубывает по всем координатам, т. е. неубывает в смысле введенного выше неравенства векторов. Это и завершает доказательство леммы.

Теорема 41. Покрывающее дерево любой сети Петри конечно.

Доказательство. Предположим, что это неверно. Тогда, в силу того, что степени всех вершин дерева ограничены (количеством переходов), на дереве существует бесконечный путь, начинающийся от корня. Рассмотрим последовательность векторов-разметок, соответствующих вершинам этого пути. Согласно лемме из этой последовательности векторов можно выделить бесконечную неубывающую подпоследовательность. Возьмем два соседних элемента этой подпоследовательности

$$a = (a_1, a_2, \dots, a_n) \le b = (b_1, b_2, \dots, b_n).$$

По построению существует путь на дереве от корня до b через вершину a. Тогда, если a = b, то согласно 2 случаю алгоритма вершина b должна быть листом, а если a < b, то согласно 3 случаю алгоритма вершина b должна быть w-листом. Оба случая противоречат тому, что b принадлежит бесконечному пути на графе и листом быть не может.

Полное покрывающее дерево

Полное покрывающее дерево также определяется динамически.

- Сначала строится покрывающее дерево.
- Затем, для каждого w-листа строится покрывающее дерево той же сети, начальной разметкой которой является данный w-лист.
- Отождествляем рассматриваемый w-лист исходного дерева и корень построенного дерева.
- Полученный граф опять будет деревом. К нему применяем ту же процедуру, пока не будут исчерпаны все w-листы.

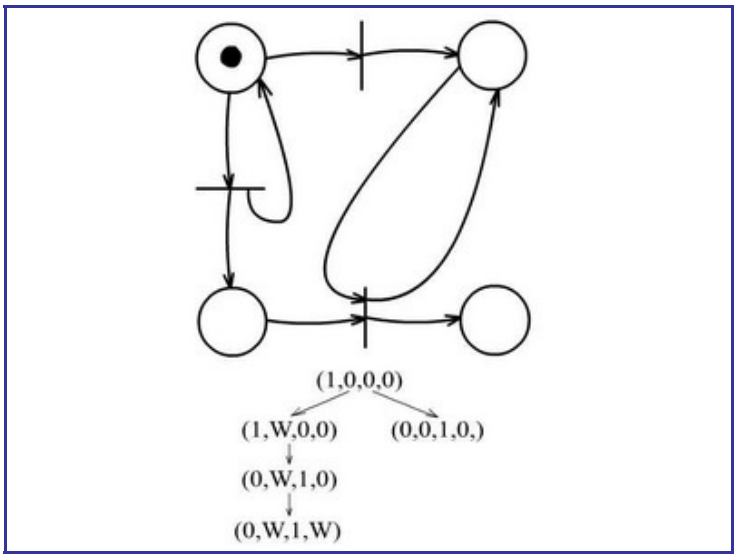


Рис. 30. Сеть Петри и ее полное покрывающее дерево

Теорема 42. Полное покрывающее дерево конечно.

Доказательство аналогично доказательству конечности покрывающего дерева. Действительно, все рассуждения доказательства конечности покрывающего дерева переносятся с разметок из множества натуральных чисел N на разметки из множества $(N_w)^n$.

Замечание. Следует подчеркнуть сохранение формулировки случая 2 алгоритма построения покрывающего дерева на разметки, содержащие символ w. Хотя текущая разметка и содержат символ w , она объявляется не w-листом, а листом. Иначе алгоритм построения полного покрывающего дерева мог бы зациклиться.

Проблемы ограниченности и безопасности

Проблема ограниченности сетей Петри состоит в вопросе о существовании алгоритма, проверяющего, нет ли в сети такого места, в котором может накапливаться бесконечное число фишек.

Аналогично выглядит проблема безопасности сетей Петри – существует ли алгоритм, проверяющий, нет ли в сети такого места, в котором может накопиться более одной фишки.

Проблемы ограниченности и безопасности алгоритмически разрешимы и в основе алгоритмов их решения лежит алгоритм построения покрывающего дерева.

Определение. Достижимую (конечную) разметку M сети Петри будем называть максимальной, если в сети не существует такой достижимой разметки M', что выполняется неравенство M < M'.

Лемма. Пусть M – какая либо максимальная достижимая разметка сети Петри. Тогда эта разметка принадлежит покрывающему дереву сети.

Доказательство. Рассмотрим произвольный путь на графе достижимых разметок (последовательность срабатывания переходов), приводящий от начальной разметки M_0 к разметке M. Если все разметки на этом пути несравнимы, то, из алгоритма построения покрывающего дерева вытекает, что разметка M принадлежит покрывающему дереву. Покажем, что другие случаи невозможны.

Действительно. Покажем сначала, что на этом пути не существует разметок, сравнимых с M. На этом пути не может быть разметки строго большей M, поскольку тогда M не была бы максимальной. Пусть на этом пути есть разметка M' строго меньшая M. Тогда рассмотрим последовательность срабатывания переходов, приводящую от M' к M. Эта последовательность может быть выполнена и начиная с разметки M, что приведет к появлению разметки, строго большей M, что противоречит максимальности M. Пусть теперь на пути от

M_0 к M встречается разметка, равная M. Тогда рассмотрим первое появление этой разметки на данном пути. Часть рассматриваемого пути, ведущая от M_0 к этой первой разметке такая, что на этом пути разметки не повторяются.

Предположим, что на пути от начальной разметки M_0 к разметке M не существует пары сравнимых между собой разметок. Предположим, что это неверно. Пусть на пути от начальной разметки M_0 к разметке M встречаются сначала разметка M_1, а затем и разметка M_2, причем $M_1 < M_2$. Тогда рассмотрим последовательность срабатывания переходов S_1, которая приводит от M_1 к M_2 и последовательность срабатывания переходов S_2, которая приводит от M_2 к M. Применив последовательность S_1 к M_2, получим новую разметку M_3, которая строго больше M_2. Тогда применив к разметке M_3 последовательность срабатывания переходов S_2, получим разметку, которая строго больше M, что противоречит максимальности M.

Пусть на пути от начальной разметки M_0 к разметке M встречаются сначала разметка M_1, а затем и разметка M_2, причем $M_1 = M_2$. Тогда на пути от начальной разметки M_0 к разметке M можно выбросить часть пути от разметки M_1 к разметке M_2 и получить путь без пары сравнимых разметок M_1 и M_2.

Теорема 43. Сеть Петри ограничена тогда и только тогда, когда покрывающее дерево сети не содержит w-листов.

Доказательство.

Предположим, что полное покрывающее дерево сети Петри содержит w-лист. Пусть разметка этого w-листа содержит символ w в координате, соответствующей месту p. Тогда в покрывающем дереве существует путь от корня M_0 к вершине с разметкой M' и далее к вершине с разметкой M, причем выполняется $M' \le M$ и $M'(p) < M(p)$. Тогда существует такая последовательность срабатывания переходов $t_1, t_2, \ldots, t_s$, которая от разметки M' приводит к разметке M. Тогда n – кратное срабатывание этой последовательности переходов приведет к увеличению количества фишек в месте p не менее, чем на n. Это и означает неограниченность места p.

Пусть теперь покрывающее дерево сети Петри не содержит w-листов. Пусть M – какая либо достижимая разметка данной сети. Покажем, что эта разметка принадлежит покрывающему дереву. Действительно, Пусть $t_1, t_2, \ldots, t_s$ такая последовательность срабатывания переходов, которая приводит от начальной разметки к разметке M и пусть на этом пути ни одна разметка не встречается дважды. Поскольку покрывающее дерево не содержит w-листов, на этом пути не встречаются разметки $M_i < M_j$, $1 \le i < j \le s$. Тогда, согласно алгоритму построения покрывающего дерева, на покрывающем дереве есть вершина с разметкой M. Поскольку покрывающее дерево конечно, то количество фишек в каждом месте произвольной достижимой разметки ограничено (наибольшим количеством фишек всех мест всех разметок покрывающего дерева).

Следствие. Проблема ограниченности сети Петри разрешима.

Доказательство. Алгоритм проверки ограниченности сводится к построению покрывающего дерева и проверке существования w-листов в нем.

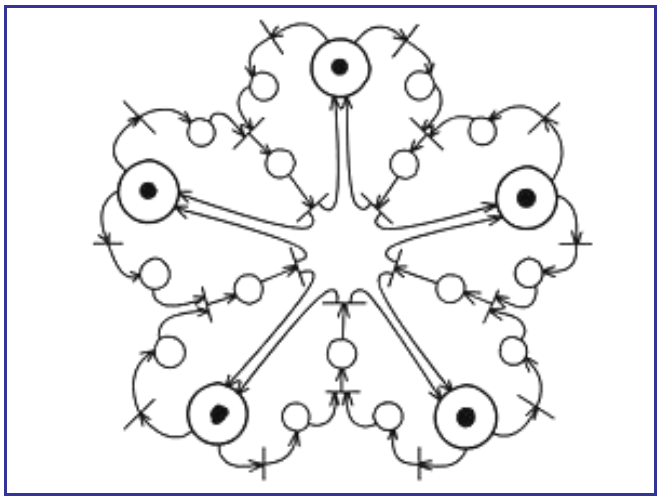
Анализ сетей Петри

При неограниченности всей сети Петри может быть интересен вопрос об ограниченности какого-нибудь конкретного места в этой сети. Для ответа на этот вопрос следует строить полное покрывающее дерево сети. Если в полном покрывающем дереве сети это место не помечено символом w, то оно ограничено.

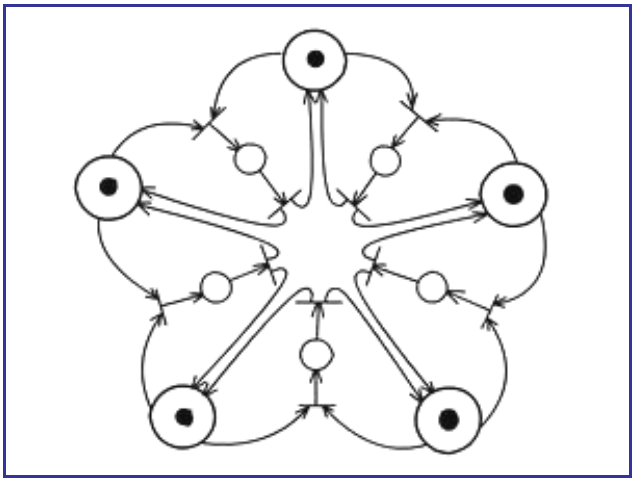
Тупиковые разметки соответствуют дедлокам (deadlock) в операционных системах.

Задача о пяти китайских обедающих философах

Пять китайских философов бродят в саду и рассуждают о вечных проблемах бытия. Каждого из них иногда отвлекает желание поесть – и он идет в беседку в центре сада, в которой накрыт круглый стол. На столе равномерно по кругу стоят 5 чашек риса. Между каждой парой соседних чашек лежит палочка. Философ, подходя к столу, выбирает чашку риса, берет палочки, лежащие по обе стороны этой чашки и ест. Насытившись, философ кладет палочки на место и опять бродит по саду, рассуждая о проблемах бытия.



Анимация 31**.** Пять обедающих философов. Модель содержит дедлок (deadlock). Если все философы одновременно подойдут к столу и каждый возьмет левой рукой палочку, то правой рукой брать будет нечего. В таком состоянии ожидания философы умрут от голода, поскольку правая палочка ни для кого не появится.



Анимация 32**.** Пять обедающих философов. Модель не содержит дедлоки. Каждый философ не может брать палочки по одной. Он может взять их только одновременно. Поэтому, он сможет насытиться и положить палочки на место. После этого другие философы смогут этими палочками воспользоваться.

Не обязательно взаимная блокировка процессов выражается тупиковой разметкой. Несколько процессов могут оказаться заблокированными при том, что другие процессы будут оставаться живыми. Поэтому в сети интересен анализ потенциально мертвых переходов. Вопрос о проверке, является ли переход потенциально мертвым решается с помощью полного покрывающего дерева. Для этого следует в сеть добавить еще одно место и дугу от исследуемого перехода к данному месту. Если в полном покрывающем дереве новой сети у всех листов в данном месте стоит символ w, то данный переход потенциально живой, а иначе – потенциально мертвый.

Описание взаимодействия параллельно выполняемых процессов с помощью сетей Петри

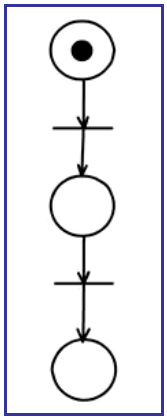


Рис. 33. Сеть Петри, в которой переходы могут срабатывать только в определенной последовательности

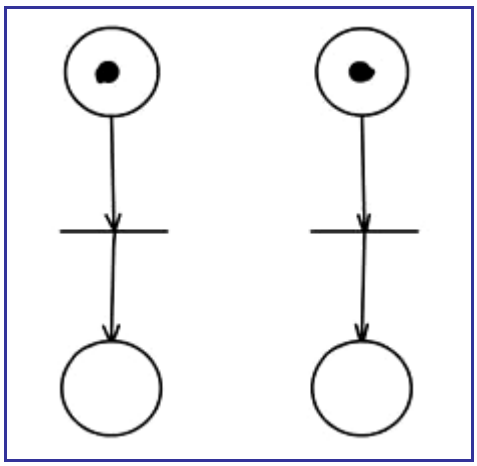


Рис. 34. Сеть Петри, в которой переходы могут срабатывать независимо друг от друга

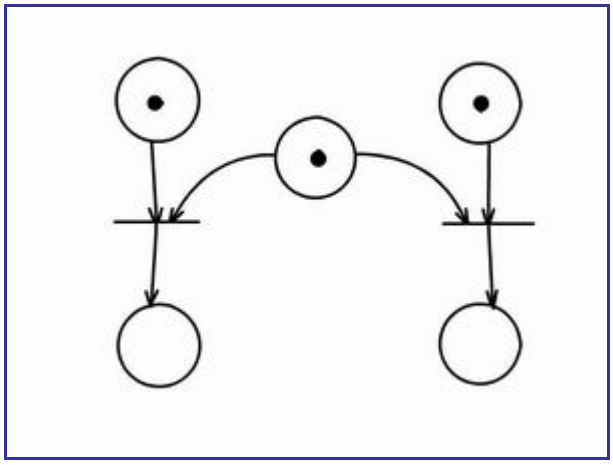


Рис. 35. Может сработать любой переход, но при этом другой переход уже сработать не сможет – будет заблокирован

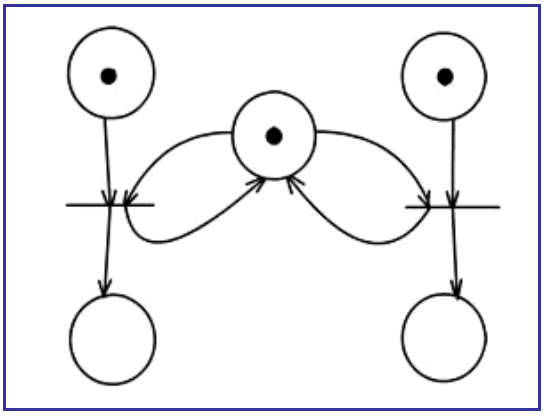


Рис. 36. Могут сработать оба перехода в любом порядке, но не одновременно

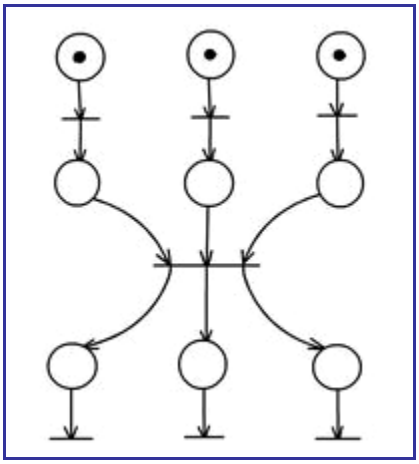


Рис. 37. Моделирование барьера с помощью Сети Петри

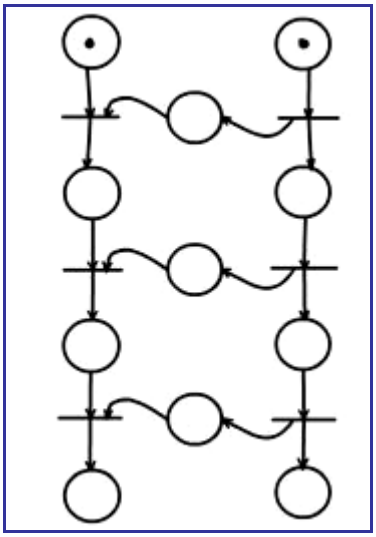


Рис. 38. Два взаимодействующих конвейера, один из которых запаздывает по отношению к другому (например, для получения от второго конвейера ресурсов)

СПИСОК ЛИТЕРАТУРЫ

1. Антонов А.С. Параллельное программирование с использованием технологии MPI.// М.: Изд-во МГУ, 2004 г., 71 с.
2. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов// М., «Мир», 1979, 536 с.
3. Бабичев А.В., Лебедев В.Г. Распараллеливание программных циклов./ Программирование// 1983, N 5, с. 52-63.
4. Вальковский В.А., Распараллеливание алгоритмов и программ. Структурный подход/ М., «Радио и связь», 1989 г., 176 с.
5. Вальковский В.А. Параллельное выполнение циклов. Метод пирамид. - Кибернетика. 1983, N 5. с. 51-55.
6. Вальковский В.А. Параллельное выполнение циклов. Метод параллелепипедов. - Кибернетика. 1982, N 2. с. 51-62.
7. Валях Е. Последовательно-параллельные вычисления// М., Мир, 1985, 456 с.
8. Векторизация программ. // Векторизация программ: теория, методы, реализация. / Сборник переводов статей М.: Мир, 1991. С. 246 - 267.
9. Воеводин В.В. Математические модели и методы в параллельных процессах// М.: Наука, гл. ред. физ.-мат. лит., 1986, 296 с.
10. Воеводин В.В. Воеводин Вл.В. Параллельные вычисления, С-Петербург «БХВ-Петербург», 2002, 599 с.
11. Воеводин В.В. Пакулев В.В. Определение дуг графа алгоритма. - М., Отдел вычислительной математики АН СССР, 1989, 22 с. (препринт).
12. Воеводин Вл. В. Статистические оценки возможности выявления параллельной структуры последовательных программ. // Программирование, № 4, 1990, с. 44-54.
13. Воробьев Н.Н. Числа Фибоначчи// М., Наука, Гл. ред. физ. мат. лит., 1984, 144 с.
14. Гохберг И.Ц., Фельдман И.А. Уравнения в свертках и проекционные методы их решений. - М.: Наука, 1971.
15. Евстигнеев В.А. Некоторые особенности программного обеспечения ЭВМ с длинным командным словом (Обзор) //Программирование, 1991, N 2, с. 69-80.
16. Евстигнеев В.А. Применение теории графов в программировании// М. «Наука», Главная редакция физико-математической литературы, 1985 г., 352 с.
17. Евстигнеев В.А., Касьянов В.Н. Оптимизирующие преобразования в распараллеливающих компиляторах// Программирование, 1996, № 6, с. 12-26.
18. Евстигнеев В.А., Спрогис С.В. Векторизация программ. - В сборнике: "Векторизация программ: теория, методы, реализация." - М.: Мир, 1991, с. 246 - 267.
19. Канторович Л.В., Акилов Г.П., Функциональный анализ// М.: «Наука», 1977, 741 с.
20. Касперский К. Техника оптимизации программ. Эффективное использование памяти. - СПб.: БХВ-Петербург, 2003 г. – 456 с.
21. Касьянов В.Н., Оптимизирующие преобразования программ/ М., «Наука», 1988 г., 336 с.

22. Касьянов В.Н., Евстигнеев В.А. Графы в программировании: обработка, визуализация и применение. - СПб.: БХВ-Петербург, 2003 г. – 1104 с.
23. Ки-Чанг Ким Мелкозернистое распараллеливание неполных гнезд циклов// Программирование, 1997, №2, с. 52-66.
24. Ковалев М.М. Дискретная оптимизация./ Мн. Изд-во БГУ,
1977, 192 с.
25. Корнеев В.В. Параллельные вычислительные системы// М., «Нолидж», 1999, 311 с.
26. Корнеев В.В., Киселев А.В., Современные микропроцессоры. 2 издание. М., «НОЛИДЖ», 2000 г., с. 315.
27. Котов В.Е., Сети Петри/ М. «Наука», Гл. редакция ф.-м. лит., 1984, 159 с.
28. Котов В.Е., Черкасова Л.А. Сетевой подход к описанию семантики параллельных систем и процессов. //В сб. «Кибернетика и вычислительная техника», М., «Наука», 1986, №2, с. 75-94.
29. Коуги П.М. Архитектура конвейерных ЭВМ//М., «Радио и связь», 1985 г. 358с.
30. Кристофидес Н. Алгоритмы на графах. - М.: Мир, 1974.
31. Лацис А.О. Как собрать и использовать суперкомпьютер.// М. «Бестселлер», 2003, 238 с.
32. Левин И.И. Модульно наращиваемая многопроцессорная вычислительная система со структурно-процедурной организацией вычислений на основе ПЛИС-технологий.// Искусственный интеллект. Научно-теоретический журнал. Институт проблем искусственного интеллекта НАНУ. Украина, Донецк, ДонДИШИ, “Наука и Освита”, 2003, № 4, с. 446-453.
33. Левин И.И., Штейнберг Б.Я. Сравнительный анализ эффективности параллельных программ для различных архитектур параллельных ЭВМ. // Искусственный интеллект. Научно-теоретический журнал. Институт проблем искусственного интеллекта НАНУ. Украина, Донецк, ДонДИШИ, “Наука и Освита”, 2001, № 3, с. 234-242.
34. Лиходед Н.А. Распределение операций и массивов данных между процессорами.// Программирование, 2003, №3, с. 73-80.
35. Матиясевич Ю.В. Вещественные числа и ЭВМ. //В сб. «Кибернетика и вычислительная техника», М., «Наука», 1986, №2, с. 104-133.
36. Прангишвили И.В., Виленкин С.Я., Медведев И.Л., Параллельные вычислительные системы с общим управлением// М.: Энергоатомиздат, 1983. – 312 с.
37. Фаддеева В.Н., Фаддеев Д.К. Параллельные вычисления в линейной алгебре. 1.// Кибернетика.- 1977.- N 6, с. 28-40.
38. Фаддеев Д.К., Фаддеева В.Н. Параллельные вычисления в линейной алгебре. 2. – Кибернетика, 1982, № 3, с. 18-31, 44.
39. Французов Ю.А. Обзор методов распараллеливания кода и программной конвейеризации// Программирование, 1992, № 3, с. 16-37.
40. Хожайнова С.А. Статистические данные о распараллеливаемости программ.- Смешанные вычисления и преобразования программ. Тезисы докладов. Новосибирск, институт систем информатики и вычислительный центр АН СССР. 27-29 ноября 1990 .

41. Штейнберг Б.Я. Распараллеливание рекуррентных циклов с условными операторами// Автоматика и телемеханика/ 1995, N6, с. 176-184.
42. Штейнберг Б.Я. Оптимальные параллельные переразмещения двумерных массивов.//Программирование, N 6, 1993 г. с.81-88.
43. Штейнберг Б.Я. Бесконфликтные размещения массивов при параллельных вычислениях// Кибернетика и системный анализ/ 1999, N 1, с. 166-178.
44. Штейнберг Б.Я. Подстановка и переименование индексных переменных в многомерных циклах.// Известия вузов. Северокавказский регион. Юбилейный выпуск. 2002 г., с. 94-99.
45. Штейнберг Б.Я. Распараллеливание рекуррентных программных циклов.// Информационные технологии, №4, 2004 г., с. 16 – 23.
46. Штейнберг Б.Я. Математические методы распараллеливания рекуррентных программных циклов на суперкомпьютеры с параллельной памятью.// Ростов-на-Дону, Издательство Ростовского университета, 2004 г., 192 с.
47. Шульженко А.М. Автоматическое определение циклов ParDo в программе // Известия вузов. Северокавказский регион. Естественные науки. Приложение 11’05. с. 77-88.
48. R. Allen, K. Kennedy Optimizing compilers for Mordern Architetures// Morgan Kaufmann Publisher, Academic Press, USA, 2002, 790 p.
49. Almasi G.S., Gottlib A. Highly Parallel Computing.- 1989, The Benjamin/Cummings Publishing Company, inc.
50. Bondalapati K.K. Modeling and Mapping for Dynamically Reconfigurable Hybrid Architectures, Thesis of Philosophy Doctor Dissertation, University of Southern Caroline, 2001.
51. Cooper K.D., Torcson L. Engineering a Compiler.// Morgan Kaufmann Publishers, Elsvier Science, San Francisko, USA, 2003.
52. Collard Jean-Francois, Feautrier Paul, Risset Tanguy Construction of DO Loops from Systems of Affine Constraints//Laboratoire de I’Informatique du Parallelisme, Lyon, Institut IMAG, Research Report № 93-15, May 1993, p. 1-26.
53. Cytron R., Ferrante J. What’s in a name? Or The value of renaming for parallelism detection and storage allocation. IBM Res. Rep. RC 12785 (#55984), 1987.
54. Fahringer T., Scholz B. A Unified Symbolic Evaluation Framework for Parallelizing Compilers// IEEE Transactions on Parallel and Distributed Systems, vol. 11, № 11, November 2000, с. 1105-1125.
55. Feautrier Paul Parametric Integer Programming// Laboratoire MASI, Institut Blaise Pascal, Universite de Versailles St-Quentin, 1988, p. 25.
56. Feautrier Paul Data Flow Analysis for Array and Scalar References// International Journal of Parallel Programming/V. 20, N 1, Feb. 1991, p. 23-53.
57. Feautrier Paul. Some efficient solutions no the affine scheduling problem. Part 1. One-dimentional Time. //”International journal of Parallel Programming”, V. 21, № 5, Octouber, 1992.
58. Feautrier Paul. Some efficient solutions no the affine scheduling problem. Part 2. Multidimentional Time. // Technical Report, IBP/MASI, Number 92.78, Octouber, 1992, 28 p.

59. Kulkurni D., ‚Stumm M. Loop and Data Transformations: A Tutorial. Technical Report CSRI 337, Computer Systems Research Institute, Univercity of Toronto, June 1993, 53 p.
60. Lamport L. The parallel execution of DO loops// Commun.
ACM.- 1974.- v.17, N 2, p. 83-93.
61. Pugh W. The Omega Test: a fast and practical integer programming algorithm for dependence analysis.// Comm. Of the ACM, August, 1992.
62. polaris.cs.uiuc.edu
63. Treleaven P.C. Parallel architecture overview.// Parallel Computing, 8 (1988), p. 59-70. North-Holland.
64. Wolfe M. Beyond Induction Variables //SIGPLAN 1992, v. 27, № 7, p. 162-174.
65. Wolf M., Banerjee U. Data Dependence and its Application to Parallel Processing/ International Journal of Parallel Programming// Vol.16, N 2, 1987, p. 137-178.
66. www.top500.org
67. www.parallel.ru
68. www.parallel.ru/v-ray
69. www.parallel.guru.ru
70. www-suif.stanford.edu/research/