РОСТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

На правах рукописи

Шульженко Александр Михайлович

ИССЛЕДОВАНИЕ ИНФОРМАЦИОННЫХ ЗАВИСИМОСТЕЙ ПРОГРАММ ДЛЯ АНАЛИЗА РАСПАРАЛЛЕЛИВАЮЩИХ ПРЕОБРАЗОВАНИЙ

05.13.11 - Математическое и программное обеспечение вычислительных машин, комплексов, систем и сетей

ДИССЕРТАЦИЯ
на соискание учёной степени
кандидата технических наук

НАУЧНЫЙ РУКОВОДИТЕЛЬ:

д. т. н., Б. Я. Штейнберг

Ростов-на-Дону
2006

ОГЛАВЛЕНИЕ

Введение **4**

1 Информационная зависимость в программе. Основные способы ее определения и представления **16**

1.1 Информационная зависимость и некоторые ее представления **17**

1.1.1 Модель рассматриваемых программ: линейный класс 18

1.1.2 Вхождения переменных, итерационное пространство 21

1.1.3 Информационная зависимость по памяти (*memory-based*) 22

1.1.4 Информационная зависимость по значению (*value-based*, *data-flow*) 30

1.1.5 Сравнение представлений информационной зависимости 34

1.2 Графовые модели информационной зависимости **37**

1.2.1 Граф информационных зависимостей 37

1.2.2 Решетчатый граф 39

1.2.2.1 Опорные пространства. Порядок исполнения операторов 39

1.2.2.2 Максимальные и минимальные решетчатые графы 43

1.2.2.3 Простые и элементарные решетчатые графы 51

1.2.2.4 Хранение решетчатых графов 56

1.2.3 Развертка решетчатого графа (таймирование, *schedule*) 61

1.3 Основные способы нахождения информационных зависимостей **66**

1.3.1 Неравенства Банержи и НОД тест 66

1.3.2 Методы построения решетчатых графов В. В. Воеводина и П. Фотрье (*P. Feautrier*) 68

1.3.3 Омега Тест 72

1.3.4 Сравнение методов нахождения информационных зависимостей 72

1.3.4.1 Неравенства Банержи и точные методы 73

1.3.4.2 Метод В. В. Воеводина, метод П. Фотрье и Омега тест 82

2 Анализ и преобразования программ, основанные на решетчатом графе **84**

2.1 Автоматическое распознавание ParDo циклов в программе **85**

2.1.1 Определение ParDo цикла по решетчатому графу 86

2.1.2 Распознавание ParDo циклов по решетчатому графу. Связь между минимальными решетчатыми графами и носителями зависимости по значению 87

2.1.3 Определение ParDo циклов и их связь с ParDo циклами по решетчатому графу 90

2.1.4 Алгоритм распознавания ParDo циклов в программе 92

2.1.5 Циклы ParDo и внешние переменные 93

2.1.6 Сравнение с другими методами распознавания ParDo циклов 95

2.2 Расщепление многомерных гнезд циклов **100**

2.2.1 Простые примеры рассматриваемых видов расщеплений 100
2.2.2 Построение расщепления в виде последовательности тесных гнезд циклов 105
2.2.2.1 Проверка существования дуги решетчатого графа между заданными выпуклыми множествами 109
2.2.2.2 Использование расщепления для распараллеливания 111
2.2.3 Построение расщепления в виде структуры произвольно вложенных циклов 113
2.2.3.1 Использование расщепления для распараллеливания: общий случай 121
2.2.4 Сравнение различных видов расщеплений 127
2.3 Подстановка переменных и экспансия массивов 129
2.3.1 Подстановка переменных 129
2.3.2 Экспансия массивов 136

3 Программная реализация построения графовых моделей информационных зависимостей и преобразований программ, их использующих 148

3.1 Анализ информационных зависимостей в ОРС 150
3.1.1 Построение расширенной информации о вхождениях 150
3.1.2 Реализация графа информационных зависимостей на основе неравенств Банержи 154
3.1.3 Реализация решетчатых графов. Выбор способа построения 157
3.1.3.1 Тестирование правильности построения решетчатого графа 161
3.1.3.2 Экспериментальные данные о времени построения решетчатых графов 163
3.1.4 Реализация построения графа информационных зависимостей на основе решетчатых графов 164
3.1.5 Реализация разметки ParDo циклов в программе 168
3.2 Преобразования программ, основанные на решетчатом графе, в ОРС 171
3.2.1 Реализация расщепления гнезда циклов. Выбор метода генерации границ циклов 171
3.2.2 Подстановка переменных и экспансия массивов 175

Заключение 180

Литература 183

Приложение А 195

Приложение Б 197

Приложение В 200

ВВЕДЕНИЕ

Актуальность темы.

Большую роль в прогрессе общества играют параллельные компьютеры. Создание самолетов, автомобилей, исследование семантики ДНК, прогноз погоды и изменений климата невозможно без применения параллельных ЭВМ. Такие компьютеры широко используются в оборонной промышленности при решении сложных задач криптографии, гидроакустики, радиолокации, а также в моделировании зрения и слуха, распознавании образов, математической физике и в математическом моделировании сложных систем. Для решения этих задач организации вынуждены приобретать мощные современные параллельные ЭВМ, чтобы не выпасть из мировой конкурентной борьбы. Оборона, промышленность и наука постоянно пополняют список задач, решение которых невозможно без параллельных ЭВМ.

В последнее время принципы параллельных ЭВМ все больше используются в персональных компьютерах, которые становятся все доступнее. Так, например, процессоры компании Intel Pentium-4 и Itanium используют технологию Hyper Threading. В настоящее время в продаже имеются персональные компьютеры на базе процессоров Intel Pentium D [101], Intel Pentium Extreme Edition [102], AMD Athlon 64 X2 [100], которые имеют двуядерную архитектуру. В дальнейшем, следует ожидать увеличение производительности персональных компьютеров за счет увеличения количества процессоров, расположенных на одном кристалле.

Для того чтобы программа выполнялась на параллельном компьютере быстрее, чем на последовательном, она должна быть специальным образом написана. Даже на компьютере с процессором Intel Pentium-4, можно найти и запустить одну последовательную программу, непрерывно выполняющую некоторые

вычисления, при работе которой будет задействовано чуть более чем 50% мощности процессора (см. Приложение А.).

Процесс разработки эффективных программ для параллельных компьютеров довольно трудоёмкий. Он требует применения специальных параллельных языков, библиотек или компиляторов, способных преобразовывать программы, написанные на последовательных языках программирования, в эффективный код параллельных ЭВМ. Часто такой процесс разработки дополнительно требует от пользователя знаний оптимизирующих/распараллеливающих преобразований, которые позволяют повысить эффективность распараллеливания при использовании параллельных языков или недостаточно мощных компиляторов.

В настоящее время разработано и разрабатывается большое количество средств создания параллельных программ, отметим наиболее заметные из них.

В ИПС РАН (г. Переславль-Залесский) под руководством С.М. Абрамова [1, 2] разработана Т-система, являющаяся средством автоматического динамического распараллеливания вычислений. Пользователь пишет программы на специально разработанном языке Т-системы Т++ (который является расширением языка С++), указывая потенциальные параллельные фрагменты кода с помощью, так называемых, Т-функций. Т-система освобождает программиста от решения задачи равномерного распределения нагрузки между доступными вычислительными ресурсами. Это особенно важно в случаях использования гетерогенных и/или меняющихся со временем параллельных вычислительных систем, а также в случаях, когда, анализируя тест программы трудно определить, как можно сбалансировать работу вычислительных узлов даже однородных суперЭВМ. Т-система также избавляет программиста от таких проблем как, организация явных обменов данными и синхронизации. Программы, реализуемые в Т-системе, пишутся в функциональном стиле.

Для программирования параллельных вычислений в *неоднородных* компьютерных сетях ИСП РАН специально разработал язык и систему программирова-

ния mpC [13, 75, 104]. Приложение mpC явно определяет абстрактную сеть, распределяет данные и вычисления внутри сети, обмен данными между процессами. Система программирования mpC использует эту информацию для отображения абстрактной сети на реальную вычислительную сеть так, чтобы обеспечить эффективное исполнение программы. Важной особенностью системы является то, что имеется возможность изменять параметры абстрактной сети на стадии исполнения программы (динамически), в зависимости от реальной производительности процессоров и каналов связи.

При разработке параллельной программы необходимо исследовать ее эффективность и масштабируемость. Однако анализ указанных свойств программы часто связан с необходимостью исследования ее многочисленных исполнений на целевом суперкомпьютере (кластере). Разрабатываемая в ИСП РАН среда ParJava [27], предоставляет разработчику инструменты, позволяющие построить модель программы и с помощью этой модели получить достаточно точные оценки времени ее выполнения на заданной параллельной вычислительной системе, используя только инструментальный компьютер.

В ИПМ им. М. В. Келдыша РАН разработан язык НОРМА [9, 10, 30], являющийся языком сверхвысокого уровня для параллельного программирования задач вычислительного характера (в частности, задач математической физики). Этот язык позволяет пользователю писать программы в терминах математических формул, значительно упрощая его работу. Сделаны компиляторы с языка НОРМА в Fortran 77 для последовательных компьютеров и распараллеливающие компиляторы в Fortran PVM, Fortran MPI и другие разновидности Fortran, специально ориентированные на конкретные архитектуры параллельных ЭВМ.

В работах [7, 32] говориться о DVM-системе, позволяющей разрабатывать параллельные программы для кластеров. В этой системе языки Fortran 77 и C снабжены дополнительными командами. Эти команды не воспринимаются

стандартными компиляторами для последовательных ПК, но с их помощью DVM-компилятор генерирует параллельный код.

В помощь разработчикам эффективных параллельных программ создаются различные средства [6, 15, 28, 29, 31].

Использование распараллеливающих компиляторов является наиболее удобным для пользователя т.к. в этом случае пользователь освобождается от необходимости изучения новых языков и систем программирования, архитектуры параллельного компьютера, а также от написания самой параллельной программы.

Разработка распараллеливающих компиляторов – одна из задач автоматического распараллеливания. Для её решения необходима детальная информация о структуре алгоритма, взаимодействии отдельных операций между собой и с памятью. Неотъемлемой частью получения этой информации является, так называемый, анализ информационных зависимостей программы. Такой анализ позволяет определять операции, итерации (циклов) и операторы в последовательной программе, которые могут быть исполнены независимо (одновременно, параллельно). Существует множество представлений информационных зависимостей, которые различаются по степени информативности. Одним из наиболее тонких представлений информационной зависимости является решетчатый граф (граф алгоритма [22]). В нашей стране методы распараллеливания, основанные на анализе решетчатого графа, активно исследуются академиком РАН В.В. Воеводиным и его учениками. Работать с этим графом, используя традиционные методы хранения, такие как матрица смежности или список дуг, сложно из-за его больших размеров: число вершин решетчатого графа не меньше, чем число исполняемых операций в программе. В.В. Воеводин предложил хранить этот граф в виде конечного числа аффинных отображений, описывающих дуги графа [25], а также разработал алгоритм, позволяющий вычислять эти отображения. На основе данного алгоритма, под руководством чл.-корр. РАН Вл. В. Воеводи-

на (НИВЦ МГУ), реализована исследовательская система V-Ray [105], предназначенная для выявления параллелизма в последовательных ФОРТРАН программах. Из других исследователей, внесших значительный вклад в теорию решетчатых графов, следует отметить П. Фотрье (P. Feautrier, Франция), который разработал алгоритм построения решетчатого графа в виде квазиаффинного дерева решений [70, 71]. До появления указанных алгоритмов построения решетчатого графа, применимых к широкому классу программ, этот граф использовался лишь в простых частных случаях, например, в методе гиперплоскостей Лампорта [78] и методах параллелепипедов и пирамид В.А. Вальковского [19, 20].

В целях дополнения системы V-Ray А.В. Фроловым (ИВМ РАН) разрабатывается система МАКРОГРАФ [42]. Эта система предоставляет информацию о параллелизме программы. Кроме этого, используя технологию решетчатых графов, система МАКРОГРАФ дает пользователю информацию о распределении массивов программы, при котором затраты на межпроцессорные пересылки данных будут минимальны [41]. Планировалось ([42]), что в эту систему будут встроены методы межпроцедурного анализа, также основанные на решетчатом графе, разработанные А. С. Антоновым [11] (НИВЦ МГУ). А. С. Антонов также занимается исследованием канонического описания решетчатого графа и его использования для определения таких свойств программ, как оценка вычислительной сложности фрагмента или оценка объема входных и выходных данных [12].

Технологии, основанные на развертке решетчатого графа (см. п. 1.2.3), исследуются и используются для оптимизации/распараллеливания В. Пухом (W.Pugh), М. Лэм (M.Lam), А. Дарте (A. Darte), Н.А. Лиходедом и многими другими. Несмотря на то, что теория решетчатых графов уже довольно давно разрабатывается, она не часто используется в распараллеливающих компиляторах и системах. Например, из распараллеливающих систем [3, 87, 98, 103, 105, 106,

108, 109, 111] только в четырех используются решетчатые графы (или развертки решетчатого графа): V-Ray [105], PIPS [98], SUIF [106] (начиная с версии SUIF2 в экспериментальных целях [107]), LooPo [111], которые являются *исследовательскими*. Отметим, что многие из указанных распараллеливающих систем, в том числе и коммерческие, используют Омега тест для точного построения графа информационных зависимостей, например, ParaWise [110, с. 42], Promis [103]. Возможно, редкое использование теории решетчатых графов на практике связано со сложностью методов их построения и использования. Кроме этого, еще не достаточно разработаны оптимизирующие и распараллеливающие преобразования программ, которые могут быть реализованы только с помощью решетчатых графов.

Цель работы.

Детальное сравнение решетчатых графов с наиболее известными представлениями информационной зависимости, а также различных методов построения этих графов и представлений. Разработка новых, основанных на решетчатых графах, методов оптимизации и распараллеливания программ, их последующая реализация в Открытой распараллеливающей системе (ОРС).

Из сформулированной цели вытекают следующие **задачи**:

1. Сравнить наиболее известные представления информационной зависимости, такие как вектор направления и носитель зависимости, с решетчатым графом, а также методы построения этих представлений. Найти и указать примеры, выявляющие слабые стороны анализа, основанного на традиционном подходе.

2. Проанализировать известные методы построения решетчатых графов. И либо выбрать один из существующих методов, либо разработать на их основе другой метод построения решетчатых графов, который реализовать в Открытой распараллеливающей системе (ОРС), разрабатываемой на кафедре АиДМ механико-математического факультета РГУ.

3. Используя теорию решетчатых графов, разработать метод определения циклов, итерации которых можно исполнять одновременно, который будет более удобен для реализации, и применим к большему классу программ, чем известные методы.

4. На основе известных методов и теории решетчатых графов разработать алгоритмы автоматической подстановки индексных переменных и экспансии массивов. Реализовать эти алгоритмы в ОРС.

Методы исследований.

В процессе решения рассматриваемых задач использовались математические методы теории преобразований программ, теории графов, исследования операций, линейной алгебры. При реализации программного обеспечения использовались принципы объектно-ориентированного программирования.

Научная новизна.

1. Разработаны новые методы расщепления многомерных циклов. Эти методы отличаются от известных тем, что по заданному разбиению пространства итераций, они позволяют расщеплять как тесные, так и нетесные многомерные циклы. Кроме того, полученные алгоритмы конкретизированы до возможности использования при автоматическом распараллеливании, в отличие от [21, 47]. При выполнении расщепления в тела результирующих циклов не добавляются условные операторы, в отличие от [69, 71].

2. Разработаны новые методы подстановки индексных переменных и экспансии массивов в многомерных циклах. Эти методы являются обобщением подстановки скалярных переменных и растягивания скаляров на случай переменных-массивов, расположенных в гнездах циклов. В некоторых частных случаях действие преобразования экспансия массивов совпадает с переименованием скалярных или индексных переменных [47]. Полученный в данной работе метод экспансии массивов отличается от известного метода [69, 71] тем, что в

результате его работы условные операторы в тела результирующих циклов не добавляются.

3. Разработан новый метод определения циклов, итерации которых можно исполнять одновременно. Этот метод позволяет находить более широкий класс циклов, итерации которых можно исполнять одновременно, чем известные методы [8, 21, 95]. Кроме того, если в данном методе использовать алгоритм построения решетчатых графов [21, параграф 6.5-6.6], то будет получен более быстрый, для циклов размерности больше 1, и простой в реализации способ определения параллелизма программы, чем [21, параграф 6.7].

4. Исследованы преимущества использования решетчатого графа по сравнению с традиционными методами анализа информационной зависимости. Найдены примеры, демонстрирующие эти преимущества.

5. Выявлена связь между минимальными решетчатыми графами и носителями информационной зависимости по значению. Для решетчатых графов, описывающих потоковую зависимость, эта связь указывалась еще в [71]. Однако в [71] рассматривались только минимальные *снизу* решетчатые графы, что не позволяло автору [71] рассматривать все типы зависимостей в программе. Поэтому выявленную в [71] связь можно было использовать для распараллеливания только в том случае, если в программе удалены все анти- и выходные зависимости [71, с. 28].

Практическая значимость.

Разработанные алгоритмы и методы могут быть использованы, как при автоматическом распараллеливании (в распараллеливающих компиляторах), так и при ручном и полуавтоматическом распараллеливании. Экспансия массивов позволяет в некоторых случаях устранять анти- и выходные зависимости, а подстановка переменной – позволяет устранять потоковые (!) зависимости в программе. Как известно, именно информационные зависимости (потоковая, анти- и выходная) определяют порядок исполнения операций в программе и тем са-

мым препятствуют ее эффективному распараллеливанию. Указанные преобразования, хоть и не для всего линейного класса, реализованы в ОРС. Т.к. данные преобразования часто трудно выполнимы вручную, их реализация в распараллеливающей системе может значительно облегчить труд программиста.

Предлагаемый в работе метод определения циклов, итерации которых можно исполнять одновременно, позволяет находить большее количество распараллеливаемых циклов в программах из линейного класса, чем аналогичные методы [8, 21, 95]. Этот метод может быть использован как для отслеживания эквивалентности, так и для увеличения эффективности распараллеливания. Он наиболее удобен для использования именно при *автоматическом* распараллеливании. Реализация этого метода может помочь программисту, использующему ОРС при распараллеливании.

Достоверность полученных результатов подтверждается строгими математическими формулировками и доказательствами, программно реализованными моделями и вычислительными экспериментами. Для проверки правильности построения решетчатых графов, на которых основаны все полученные автором результаты, были разработаны специальные функции тестирования (см. пункт 3.1.3.1).

Личный вклад. Постановка задачи о разработке алгоритмов подстановки и переименовании индексных переменных, а также об их реализации, принадлежит научному руководителю, Штейнбергу Б. Я. Теоретические результаты диссертации получены автором лично. Большинство научных публикаций по теме диссертации выполнены без соавторов. Программная реализация графа информационных зависимостей, решетчатых графов, расщепления многомерных циклов, подстановки индексных переменных, экспансии массивов, метода определения ParDo циклов по решетчатым графам также выполнены автором лично. Однако эти реализации были бы невозможны без труда группы разработчиков ОРС, и, особенно, Черданцева Д.Н., Петренко В.В, Макошенко Д.В, которые

реализовали внутреннее представление ОРС и богатый набор функций для работы с ним.

Гранты.

Часть исследований диссертации поддерживалась грантами:

ФЦП «Интеграция» 2002-2006 гг., гос. контракт - Б0024/2148, «Разработка комплекса программных средств для высокопроизводительных вычислений».

Грант Союзного Российско-Беларусского государства (шифр «ТРИАДА»), 2005 г.

Апробация работы.

Основные результаты диссертации докладывались и обсуждались на Всероссийских, международных и региональных научно-технических конференциях:

- Интеллектуальные и многопроцессорные системы ИМС-2003, Международная научно-техническая конференция/ Дивноморское, 22-27 сентября 2003 г.
- Научно-методическая конференция «Современные информационные технологии в образовании: Южный федеральный округ», Ростов-на-Дону, 12-15 мая 2004 г.
- Всероссийская научно-техническая конференция «Параллельные вычисления в задачах математической физики», Ростов-на-Дону, 21-25 июня 2004 г.
- XIII Международная конференция «Математика. Экономика. Образование.», г. Новороссийск, 29 мая – 5 июня 2005 г.
- Всероссийская научная конференция «Научный сервис в сети Интернет: технологии распределенных вычислений», г. Новороссийск, 19-24 сентября 2005 г.

По результатам выполненных исследований опубликовано 7 печатных работ ([50 – 56]), в том числе 3 в соавторстве. Из них 1 статья в российском журнале «Известия вузов. Северокавказский регион.», 1 статья в журнале Национальной

Академии Наук Украины «Искусственный интеллект», 3 статьи в сборниках трудов, и 2 в сборниках тезисов конференций.

Основные результаты, выносимые на защиту

1. Методы расщепления многомерных гнезд циклов.

2. Методы подстановки индексных переменных и экспансии массивов в многомерных циклах.

3. Метод определения циклов, итерации которых можно выполнять одновременно, основанный на теории решетчатых графов. Обоснование метода в виде необходимых и достаточных условий, доказанных в работе и применимых ко всему линейному классу программ.

4. Описание связи между решетчатыми графами и носителями зависимости по значению.

5. Программная реализация разработанных графовых моделей программ, основанных на этих моделях преобразований программ, а также метода определения циклов, итерации которых можно выполнять одновременно.

Структура диссертации.

В первой главе приводится описание класса программ, к которому применимы разработанные в диссертации методы. Большая часть главы посвящена обзору известных результатов, необходимых для изложения материала работы. В первом параграфе вводятся определения различных, наиболее часто используемых, представлений информационной зависимости. Приводится сравнение этих представлений. Во втором параграфе вводятся графы, описывающие информационные зависимости: граф информационных зависимостей и различные виды решетчатых графов. Описываются способы их представления. В последнем параграфе первой главы описываются наиболее часто используемые методы построения различных представлений информационной зависимости, в том числе и решетчатых графов. Приводится выполненный автором сравнительный анализ этих методов построения: указываются достоинства и недостатки.

Во второй главе приводятся основные теоретические результаты, полученные автором. Первый параграф данной главы посвящен автоматическому распознаванию циклов, итерации которых можно исполнять одновременно. Доказываются необходимые и достаточные условия ParDo цикла по решетчатому графу. Попутно доказывается связь решетчатого графа и носителей зависимости по значению. Затем указывается, какие решетчатые нужно использовать для определения циклов, итерации которых можно выполнять одновременно. Приводится сравнение полученного метода с другими методами определения ParDo циклов. Во втором параграфе приводятся методы различных видов расщеплений многомерных гнезд циклов. Указывается, как эти методы, вместе с методом определения ParDo циклов, могут быть использованы для распараллеливания гнезд циклов. В последнем параграфе второй главы описываются методы выполнения подстановки индексных переменных и экспансии массивов в многомерных циклах.

Третья глава посвящена программной реализации разработанных в диссертации методов, которая была выполненна автором в рамках проекта OPC. Здесь описываются наиболее важные аспекты, лежащие в основе этой программной реализации. Приводятся необходимые алгоритмы и структуры данных. Эта информация может быть необходима как настоящим, так и будущим разработчикам OPC. В данной главе приводятся копии экранных форм OPC, демонстрирующих некоторые реализованные автором возможности системы. Также третья глава содержит информацию об экспериментах по выявлению времени построения различных видов решетчатых графов для фрагментов реальных программ.

В заключении приводится краткое описание полученных результатов и подводятся итоги проделанной автором работы.

1 ИНФОРМАЦИОННАЯ ЗАВИСИМОСТЬ В ПРОГРАММЕ. ОСНОВНЫЕ СПОСОБЫ ЕЕ ОПРЕДЕЛЕНИЯ И ПРЕДСТАВЛЕНИЯ

Анализ информационной зависимости[1] в программе является неотъемлемой частью любого автоматического выявления параллелизма. Отношения информационной зависимости используются для определения операций, итераций (циклов) и операторов программы, которые можно исполнять одновременно (параллельно). Также, анализ информационных зависимостей является основой для проверки эквивалентности большого количества оптимизирующих и распараллеливающих преобразований программ. Детальный анализ информационных зависимостей, например, с помощью решетчатых графов, позволяет установить порядок, в котором должны исполняться операции программы, и размещение данных, необходимые для достижения максимальной производительности на определенном параллельном компьютере.

---

[1] вместо термина «информационная зависимость» иногда в литературе используется термин «зависимость по данным».

1.1 Информационная зависимость и некоторые ее представления

Возможности любого компилятора зависят от возможностей его анализаторов. Выбор правильного представления информационной зависимости для анализа программ является ключевым моментом разработки компилятора. Для того чтобы представление было целесообразным, оно должно содержать информацию достаточную для обеспечения необходимых оптимизаций и преобразований кода. Кроме этого, оно должно позволять извлекать информацию из достаточно большого класса программ. Информационная зависимость по памяти[2], вектор направления, вектор расстояния, носитель (уровень) зависимости по памяти [8, 45, 37, 84, 95] являются важными представлениями, хорошо зарекомендовавшими себя при распараллеливании и преобразованиях циклов [8, 26, 49, 66, 83, 95]. Они применимы к большому количеству программ. Однако существуют более тонкие представления информационной зависимости в программе: информационная зависимость по значению, вектор направления зависимости по значению, вектор расстояния зависимости по значению, носитель (уровень) зависимости по значению [83, 92, 91]. Хотя известны и другие виды представлений, (например, многогранник зависимости [67, 68]) указанные выше представления являются наиболее известными и широкоиспользуемыми. В этом разделе мы подробно рассмотрим и сравним эти виды представлений. Но сначала опишем класс рассматриваемых программ и введем необходимые определения.

---

[2] во многих работах вместо термина «информационная зависимость по памяти» используется термин «информационная зависимость».

1.1.1 Модель рассматриваемых программ: линейный класс

Далее в настоящей работе, кроме специально оговоренных случаев, будут рассматриваться только программы и фрагменты программ, принадлежащие линейному классу [21, с. 340]. Для полноты изложения опишем этот класс:

- в программе используется любое количество простых переменных и переменных с индексами;
- единственным типом исполнительного оператора может быть оператор присваивания, правая часть которого является арифметическим выражением;
- все повторяющиеся операции описываются только с помощью циклов Do языка программирования Fortran (либо эквивалентными циклами `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;
- допускается использование любого количества условных и безусловных операторов перехода, передающих управление «вниз» по тексту; не допускается использование побочных выходов из циклов;
- все индексные выражения переменных, границы изменения параметров циклов и условия передачи управления задаются, в общем случае, неоднородными формами, линейными по совокупности параметров циклов и внешних переменных[3] программы; все коэффициенты линейных форм являются целыми числами;
- внешние переменные программы всегда целочисленные; конкретные значения внешних переменных известны только во время работы программы и неизвестны в момент ее исследования.

Программы, принадлежащие этому классу, называются *линейными*.

---

[3] Внешние переменные фрагмента программы – переменные, значения которых в этом фрагменте не изменяются и не могут быть определены на момент компиляции программы.

Пример 1. Нижеследующее гнездо циклов принадлежит линейному классу программ.

```
for(i=0; i<N; i=i+1)

{   a[2*i] = a[i] ;

     for(j=0; j<M1; j=j+1)

       a[i+2*j] = b[i+j+n];

     for(j=0; j<M2; j=j+1)

       if((i+j)>0)

           b[i+m] = d[i+j];

}
```

Конец примера 1.

Пример 2. Рассмотрим гнездо циклов:

```
for(i=0; i<N; i=i+1)              // Stmt1
{   a[2*i] = a[i] ;               // Stmt2
     for(j=0; j<M1; j=j+2)        // Stmt3
       a[i+2*j] = b[i*j+j*n];     // Stmt4
     for(j=0; j<M2; j=j+1)        // Stmt5
       if(a[i]>0)                 // Stmt6
           b[f[i+m]] = d[i+j];    // Stmt7
    function_call();              // Stmt8
}
```

Данное гнездо циклов не принадлежит линейному классу по следующим причинам:

1. шаг цикла *Stmt3* не равен 1;

2. в правой части оператора *Stmt4* имеется массив *b*, индексное выражение которого содержит нелинейности *i***j* и *j***n*;

3. правая часть условного выражения оператора *Stmt6* – *a*[*i*] – не является линейной формой относительно совокупности счетчиков циклов и внешних переменных;

4. в левой части оператора *Stmt7* содержится массив *b*, индексное выражение которого – *f*[*i*+*m*] – не является линейной формой относительно совокупности счетчиков циклов и внешних переменных;

5. оператор *Stmt8* есть вызов функции.

Конец примера 2.

Многие программы могут быть приведены к линейному классу с помощью различных преобразований [21, параграф 8.1]. Например, если в программе встречается произведение внешних переменных, то это произведение можно заменить новой внешней переменной. Если у некоторого оператора цикла шаг не равен +1, то с помощью канонизации циклов [33, 49] можно получить программу, в которой этот цикл имеет шаг +1. Согласно статистике [21, с. 341], с помощью различных приемов, можно привести к линейному классу 90–95% текста практически используемых программ. Рассмотрение программ, не принадлежащих линейному классу, выходит за рамки данной работы.

1.1.2 Вхождения переменных, итерационное пространство

По аналогии с определениями Л. Лампорта [78, с. 86] введем следующие определения.

Пусть X – переменная в программе. *Вхождением переменной* X будем называть всякое появление этой переменной в тексте программы. Различные вхождения, в том числе одной и той же переменной, будем обозначать различными прописными латинскими буквами, иногда с индексами. Если вхождение v находится в операторе присваивания *Stmt*, то будем говорить, что вхождение v принадлежит оператору *Stmt* и писать «$v \in Stmt$».

Не умаляя общности, все переменные, кроме счетчиков циклов, можно считать индексными. Всякому вхождению переменной при конкретном значении индексного выражения соответствует обращение к некоторой ячейке памяти. Если при этом обращении меняется состояние ячейки (вхождение в левую часть оператора присваивания, которое не входит в индексное выражение другого вхождения), то такое вхождение называется *генератором*. Остальные вхождения называются *использованиями*.

Пример 3. В следующем операторе присваивания

```
A[i+1]=C[2*i-2]+A[i+2];
```

$v1$ $v2$ $v3$ $v4$ $v5$ $v6$

имеется шесть вхождений переменных. Из них только вхождение $v1$ является генератором.

Конец примера 3.

Рассмотрим гнездо из n вложенных друг в друга циклов. Занумеруем операторы циклов в этом гнезде в соответствии с порядком вложенности, начиная с самого внешнего цикла. Обозначим счетчик i-го цикла – I_i. Будем писать $v[I'_1, I'_2, \dots I'_n]$ (или $v[\boldsymbol{I'}]$, где $\boldsymbol{I}=(I'_1, I'_2, \dots I'_n)$) чтобы сослаться на вхождение v при значениях счетчиков циклов $I_1=I'_1$, $I_2=I'_2$, … $I_n=I'_n$.

<u>Определение.</u> Множество значений, которые может принимать вектор счетчиков циклов $\boldsymbol{I}=(I_1, I_2, \dots, I_n)$ в указанном гнезде, называется *пространством итераций* данного гнезда. Для программ из линейного класса пространство итераций может быть описано как все целые точки, содержащиеся в некотором выпуклом многограннике. Этот многогранник будем называть *линейным пространством итераций*.

1.1.3 Информационная зависимость по памяти (memory-based)

В этом пункте будет введено понятие информационной зависимости по памяти (*memory-based dependence* [91, 92]), а также ее представления: вектор направления, носитель, вектор расстояния.

<u>Определение.</u> ([8]) Если при допустимых индексных выражениях два вхождения u и v обращаются к одной и той же ячейке памяти, то говорят, что эти вхождения порождают *информационную зависимость по памяти*. При этом

вхождение *v зависит по памяти* от вхождения *u*, если вхождение *v* обращается к некоторой ячейке памяти позже, чем вхождение *u*.

Для упрощения изложения далее в этой работе при использовании термина «зависимость по памяти», слова «по памяти» будут опускаться (кроме либо специально оговоренных случаев, либо случаев, в которых возможно нарушение однозначности толкования). Т.е. вместо термина «информационная зависимость по памяти», будет использоваться термин «информационная зависимость». Вместо фразы «вхождение *v зависит по памяти* от вхождения *u*», будет использоваться фраза «вхождение *v зависит* от вхождения *u*», и т.д.

Пример 4. В следующем фрагменте программы

```
for(i=1;i<10;i++)

    x[i]= x[i-1]+y[i];
```

u ***v***

имеется зависимость вхождения *v* от *u* т.к., например, к ячейке памяти *x*[5] обращается сначала вхождение *u* на итерации 5, а затем вхождение *v* на итерации 6.

Конец примера 4.

<u>Определение.</u> ([95]) Пусть вхождение *v* зависит от вхождения *u*. Различают четыре типа информационной зависимости:

1. если *u* является генератором, а *v* – использованием, то зависимость такого типа называется *истинной (true dependence)* или *потоковой зависимостью (flow dependence)*;
2. если *u* является использованием, а *v* – генератором, то зависимость такого типа называется *антизависимостью (antidependence)*;

3. если *u* и *v* являются генераторами, то зависимость такого типа называется *выходной зависимостью (output dependence)* или *зависимостью по выходу*;
4. если *u* и *v* являются генераторами, то зависимость такого типа называется *входной зависимостью (input dependence)* или *зависимостью по входу*.

<u>Определение.</u> Если вхождение *v* зависит от себя же (вхождения *v*), то будем говорить, что имеет место самозависимость вхождения *v*.

Пример 5. Рассмотрим фрагмент программы умножения двух многочленов:

```
for(k=0; k<=(N+M); k++)

    c[k]=0;
     u1
for(i=0; i<=M; i++)

    for(j=0; j<=N; j++)

      c[i+j]=c[i+j]+a[i]*b[j];
       u2        v1       v2     v3
```

Перечислим все информационные зависимости в данном примере.

1. Потоковые зависимости: зависимость использования *v*1 от генератора *u*1, и зависимость использования *v*1 от генератора *u*2.

2. Выходные зависимости: зависимость генератора *u*2 от генератора *u*1, и зависимость генератора *u*2 от себя же – самозависимость (этот генератор перезаписывает ячейки памяти, которые были записаны ранее им же).

3. Антизависимость: зависимость генератора *u*2 от использования *v*1.

4. Входная зависимость: самозависимость использования *v*1, самозависимость использования *v*2, самозависимость использования *v*3.

Конец примера 5.

Пусть два зависимых вхождения u и v имеют d общих циклов. Пусть вхождение v зависит от вхождения u. Тогда существуют такие итерации $\boldsymbol{I}$ и $\boldsymbol{J}$, что $v[\boldsymbol{J}]$ зависит от $u[\boldsymbol{I}]$ (т.е. $u[\boldsymbol{I}]$ и $v[\boldsymbol{J}]$ обращаются к одной ячейке памяти, причем $u[\boldsymbol{I}]$ обращается раньше $v[\boldsymbol{J}]$). Для первых d компонент этих итераций имеют место следующие соотношения: $I_i\ \omega_i\ J_i$, где $\omega_i \in \{<, =, >\}$, $i \in [1, d]_N$. Из набора отношений ω_i (i=1..d) можно сформировать вектор-отношение $\boldsymbol{\Omega}$: $\boldsymbol{\Omega} = (\omega_1, \omega_2, \ldots, \omega_d)$. Таким образом, будем говорить, что между итерациями $\boldsymbol{I}$ и $\boldsymbol{J}$ имеет место отношение $\boldsymbol{\Omega}=(\omega_1, \omega_2, \ldots, \omega_d)$, и писать $\boldsymbol{I}\ \boldsymbol{\Omega}\ \boldsymbol{J}$, если выполняется: $I_i\ \omega_i\ J_i$, для всех $i \in [1, d]_N$. Обычно, для зависимых вхождений имеется несколько пар итераций $(\boldsymbol{I}^k, \boldsymbol{J}^k)$, где $k \in [1, m]_N$, на которых $v[\boldsymbol{J}^k]$ зависит от $u[\boldsymbol{I}^k]$. Пусть между парами итераций $(\boldsymbol{I}^k, \boldsymbol{J}^k)$ существует отношение $\boldsymbol{\Omega}^k$.

<u>Определение.</u> Объединением отношений $\cup\boldsymbol{\Omega}^k$ ($\boldsymbol{\Omega}^k = (\omega_{k,1}, \omega_{k,2}, \ldots, \omega_{k,d})$) по k называется отношение $\boldsymbol{\Psi} = (\cup\{\omega_{k,1}\}, \cup\{\omega_{k,2}\}, \ldots, \cup\{\omega_{k,d}\})$, где «$\cup$» обозначает объединение по k.

Дополним множество $\{<, =, >\}$ элементами: элемент $\{\leq\}$ – эквивалентен $\{<\}\cup\{=\}$, элемент $\{\geq\}$ – эквивалентен $\{>\}\cup\{=\}$, элемент $\{\neq\}$ – эквивалентен $\{<\}\cup\{>\}$, элемент $\{*\}$ – эквивалентен $\{<\}\cup\{=\}\cup\{>\}$.

В силу введенного определения, каждая компонента вектор-отношения $\boldsymbol{\Psi}$ состоит из множества элементов. Поэтому, к таким вектор-отношениям применимы операции, определенные для множеств: объединение, пересечение, разность и др.

Пример 6. Пусть $\boldsymbol{\Omega}^1 = (<, >, =)$, $\boldsymbol{\Omega}^2 = (=, <, <)$, $\boldsymbol{\Omega}^3 = (<, >, >)$ тогда $\cup\boldsymbol{\Omega}^i=(\{<\}\cup\{=\}\cup\{<\}, \{>\}\cup\{<\}\cup\{>\}, \{=\}\cup\{<\}\cup\{>\})=(\leq, \neq, *)$ (повторяющиеся элементы в каждом подмножестве были отброшены).

Конец примера 6.

<u>Определение.</u> ([95, 33]) Будем называть *вектором направления зависимости* v от u вектор $\boldsymbol{\Psi}=(\psi_1, \psi_2,\ldots, \psi_d)$ (где $\psi_i \in \{<, >, =, \leq, \geq, \neq, *\}$) равный объединению отношений $\boldsymbol{\Omega}^k$ по всем парам итераций $(\boldsymbol{I}^k, \boldsymbol{J}^k)$, при которых $v[\boldsymbol{J}^k]$ зависит от $u[\boldsymbol{I}^k]$.

Представление зависимости с помощью данного вектора направления используется алгоритмом распараллеливания Лэм и Вольфа [68, 96].

Пример 7. Рассмотрим фрагмент программы:

```
for(i=1; i<=M; i++)

{   for(j=1; j<=N; j++)

    { a[j]=x[i-1][j+1]+...;
       u      t
      x[i][j]=a[j]*...;
       s         v1
    }

     b[i]=a[i]*...;
           v2
}
```

В данном фрагменте программы имеется потоковая зависимость использования t от генератора s. В ячейку памяти x[2][3] сначала генератор s записывает значение на итерации (i, j)=(2, 3). Затем использование t читает это значение на итерации (i, j)=(3, 2). Между этими итерациями существует следующее отношение:

$i: \lceil 2 \rceil < \lceil 3 \rceil$

$j: \lfloor 3 \rfloor > \lfloor 2 \rfloor$

Таким образом, можно утверждать, что вектор направления (<, >) содержится в векторе направления зависимости t от s. Заметим, что отношение (<, >) имеет место между каждой парой итераций, на которых указанные вхождения обращаются к одной и той же ячейке памяти. Следовательно, вхождение t зависит от s с вектором направления (<, >).

Рассмотрим потоковую зависимость использования $v1$ от генератора u. Генератор u записывает значения в элемент массива $a[k]$ ($1 \le k \le N$) на итерациях $(i, j)=(i', k)$, где $1 \le i' \le M$. После очередной записи генератора u в $a[k]$, использование $v1$ производит чтение из элемента массива $a[k]$ на итерациях $(i, j)=(i'', k)$, где $i' \le i'' \le M$. Таким образом, между всеми парами итераций, на которых $v1$ зависит от u, существует отношение (≤, =). Следовательно, вектор направления данной зависимости имеет вид: $\boldsymbol{\Psi}_{u,\, v1} = (\le, =)$.

Рассмотрим потоковую зависимость использования $v2$ от генератора u. Т.к. у этих вхождений только один общий цикл (d=1), то вектор направления указанной зависимости будет состоять из одного элемента. Рассмотрим элемент массива $a[k]$, где $1 \le k \le min(N, M)$. Использование $v2$ читает из элемента $a[k]$ на итерации $(i)=(k)$. До этого чтения, генератор u производил запись в элемент $a[k]$ на итерациях $(i, j)=(i', k)$, где $i' \le k$. Следовательно, вектор направления зависимости $v2$ от u имеет вид: $\boldsymbol{\Psi}_{u,\, v2} = (\le)$.

Конец примера 7.

<u>Определение.</u> ([8]) Говорят, что вхождение *v зависит от вхождения u с носителем k* ($1 \le k \le d$), если существуют такие итерации $\boldsymbol{I}=(I_1, I_2, \ldots, I_{n1})$ и $\boldsymbol{J}=(J_1, J_2, \ldots, J_{n2})$, что:

1. вхождение $v[\boldsymbol{J}]$ зависит от вхождения $u[\boldsymbol{I}]$;
2. для всех $i \in [1, k\text{-}1]$ выполняется равенство $I_i = J_i$,
3. для k–тых компонент векторов итераций выполняется неравенство $I_k < J_k$.

В этом случае также говорят, что зависимость v от u является *циклически порожденной* (*loop-carried dependence* [57, с. 49]).

Замечание 1. Определение носителя можно дать в терминах вектора направления зависимости: вхождение v *зависит от вхождения* u *с носителем* k ($1 \le k \le d$), если для вектора направления данной зависимости $\boldsymbol{\Psi}=(\psi_1, \psi_2,\ldots, \psi_d)$ выполняются условия: $\{=\}\subset\psi_i$, при $1\le i\le(k\text{-}1)$ и $\{<\}\subset\psi_k$.

Определение носителя зависимости есть переформулированное определение из [8, с. 100]. Иногда вместо термина «носитель зависимости» в литературе употребляется эквивалентный термин «уровень зависимости» (*dependence level*) [83, с. 3]. Понятие носителя (уровня) зависимости используется алгоритмом распараллеливания Аллена и Кеннеди [8, 66].

Определение. ([8, с. 100]) Зависимость вхождения v от вхождения u называется *циклически независимой* (*loop independent dependence*[57, с. 49]), если существуют такие итерации $\boldsymbol{I}=(I_1, I_2, \ldots, I_{n1})$ и $\boldsymbol{J}=(J_1, J_2, \ldots, J_{n2})$, что:

1. вхождение $v[\boldsymbol{J}]$ зависит от вхождения $u[\boldsymbol{I}]$;
2. для всех $i\in[1, d]$ выполняется равенство $I_i = J_i$.

Замечание 2. Определение циклически независимой зависимости можно дать в терминах вектора направления зависимости: зависимость вхождения v от вхождения u называется циклически независимой, если для вектора направления данной зависимости $\boldsymbol{\Psi}=(\psi_1, \psi_2,\ldots, \psi_d)$ выполняются условия: $\{=\}\subset\psi_i$, при $1\le i\le d$.

Из замечаний 1 и 2 видно, что информация о носителе зависимости и о циклически независимой зависимости может быть получена из вектора направления зависимости.

Пример 8. Рассмотрим фрагмент программы примера 7.

Вектор направления зависимости $v1$ от u равен (≤, =), как было показано в примере 7. В этом векторе содержится вектор (=, =), показывающий, что данная зависимость является циклически независимой. Также, *первая* компонента вектора (≤, =) содержит {<}, следовательно, указанная зависимость имеет носитель 1. Зависимость $v1$ от u не имеет носитель 2 т.к. вектор направления (≤, =) не содержит вектор (=, <).

Конец примера 8.

<u>Определение.</u> ([83, с. 3]). Пусть два вхождения u и v имеют d общих циклов. Рассмотрим множество всех итерации $\boldsymbol{I}^k=(I^k_1, I^k_2, \ldots, I^k_{n1})$ и $\boldsymbol{J}^k=(J^k_1, J^k_2, \ldots, J^k_{n2})$, где $1 \le k \le m$, $n1 \ge d$, $n2 \ge d$, таких, что $v[\boldsymbol{J}^k]$ зависит от $u[\boldsymbol{I}^k]$. Каждый вектор $(J^k_1 - I^k_1, J^k_2 - I^k_2, \ldots, J^k_d - I^k_d)$ (при фиксированном k) будем называть *вектором расстояния зависимости* v от u, а множество различных векторов $(J^k_1 - I^k_1, J^k_2 - I^k_2, \ldots, J^k_d - I^k_d)$ будем называть *множеством векторов расстояния зависимости* v от u.

Подробно ознакомиться с различными видами определений расстояния зависимости для различного вида циклов можно, например, в [90, 94].

Пример 9. Рассмотрим фрагмент программы примера 7.

Рассмотрим зависимость t от s. Разность между всеми парами итераций $\boldsymbol{I}^k$ и $\boldsymbol{J}^k$, при которых $v[\boldsymbol{J}^k]$ зависит от $u[\boldsymbol{I}^k]$, постоянна и равна (1, -1).

Рассмотрим зависимость использования $v1$ от генератора u. Из формул для пар итераций, при которых $v1$ зависит от u, приведенных в примере 7, следует, что множество векторов расстояния для данной зависимости имеет вид: (0, 0), (1, 0), (2, 0), … (M-1, 0).

Рассмотрим потоковую зависимость использования $v2$ от генератора u. Т.к. у этих вхождений только один общий цикл (d=1), то векторы расстояния указанной зависимости будут состоять из одного элемента. Из формул для пар итераций, при которых $v2$ зависит от u, приведенных в примере 7, следует, что мно-

жество векторов расстояния для данной зависимости имеет вид: (0), (1), (2), … (*min*(*N*, *M*)-1).

Конец примера 9.

1.1.4 Информационная зависимость по значению

В этом пункте будет введено понятие информационной зависимости по значению (*value-based dependence* [91, 92], *data-flow dependence* [83, 85]), а также ее представления.

<u>Определение.</u> ([92]) Говорят, что вхождение *v зависит по значению* от вхождения *u*, если существуют такие итерации ***I*** и ***J***, что сначала *u*[***I***], затем *v*[***J***] обращаются к некоторой ячейке памяти *S* и между этими обращениями в ячейку памяти *S* не производится запись. Аналогично информационной зависимости, для зависимости по значению вводятся четыре типа: потоковая, антизависимость, выходная, входная.

Пример 10. В следующем фрагменте программы

```
for(i=0;i<10;i++)

{   x[i]= z[i]+y[i];
     u1            s
    y[i]= f[i]*d[i];
     t1
    x[9-i]= x[i];
     u2        v
    y[i]= g[2*i];
     t2
}
```

имеется только одна потоковая зависимость по значению – вхождение *v* зависит по значению от вхождения *u*1. Вхождение *v* не зависит по значению от

вхождения $u2$ т.к. между любой записью генератора $u2$ в некоторую ячейку S и последующим чтением вхождения v из нее, данная ячейка S перезаписывается генератором $u1$. Например, в ячейку памяти $x[6]$ сначала генератор $u2$ записывает некоторое значение на итерации i=3, позже использование v читает содержимое ячейки $x[6]$ на итерации i=6, но между этими обращениями к $x[6]$ генератор $u1$ перезаписывает значение в $x[6]$ на итерации i=6. Таким образом, использование v читает только те значения, которые были записаны генератором $u1$.

Заметим, что в данном примере имеется две потоковых зависимости: v от $u1$ и v от $u2$. Информационную зависимость v от $u2$ часто называют *ложной зависимостью*[4] (*false data dependence*) [91].

Также в данном примере существует антизависимость по значению $t1$ от s, но не существует антизависимости по значению $t2$ от s.

Конец примера 10.

<u>Определение.</u> ([83]) *Вектором направления зависимости по значению* $\boldsymbol{\Psi}=(\psi_1, \psi_2,\ldots, \psi_d)$ (где $\psi_i \in \{<, >, =, \leq, \geq, \neq, *\}$) вхождения v от вхождения u называется объединение отношений $\boldsymbol{\Omega}^k$ по всем парам итераций $(\boldsymbol{I}^k, \boldsymbol{J}^k)$, при которых $v[\boldsymbol{J}^k]$ зависит по значению от $u[\boldsymbol{I}^k]$.

Пример 11. Рассмотрим фрагмент программы примера 7.

Рассмотрим зависимость использования $v1$ от генератора u. Пары итераций, на которых сначала генератор u, а затем использование $v1$ обращается к одной ячейке памяти $a[k]$, есть множество $\{((i', k), (i'', k)) \mid 1 \leq i' \leq M, i' \leq i'' \leq M\}$ (пример 7.). В этом множестве есть пары итераций $((i'^*, k), (i''^*, k))$, где $i'^* < i''^*$. Генератор u на итерации (i'^*, k) записывает в $a[k]$ некоторое значение. Затем этот же генера-

---

[4] формально, ложная зависимость это информационная зависимость, которая исчезает при переходе от информационной зависимости к информационной зависимости по значению.

тор перезаписывает $a[k]$ на итерациях (l, k), где $i'^*<l\le i''^*$. И только после этого, на итерации (i''^*, k) использование $v1$ читает значение $a[k]$. Использование $v1$ на итерации (i''^*, k) читает именно то значение из $a[k]$, которое было записано последним, т.е. генератором u на итерации (i''^*, k). Для построения вектора направления зависимости по значению нужно исключить из рассмотрения пары данного вида. Получим, что вектор направления зависимости по значению $v1$ от u есть объединение отношений между парами итераций во множестве $\{((i', k), (i'', k)) \mid 1\le i'\le M, 2\le i''\le M, \boldsymbol{i'=i''}\}$. Следовательно, этот вектор равен (=, =).

Рассмотрим зависимость $v1$ от u: вектор направления зависимости равен (≤, =) (пример 7) и не совпадает с вектором направления зависимости по значению (=, =). Разница между этими векторами – вектор (<, =) – есть вектор направления выходной самозависимости генератора u. Этот вектор описывает отношение между парами итераций, на которых генератор u перезаписывает одни и те же ячейки $a[k]$ и, в данном случае, совпадает с вектором направления самозависимости по значению генератора u. Таким образом, вектор направления зависимости (≤, =) содержит информацию не только о потоковой зависимости $v1$ от u, но и о выходной самозависимости u от u.

Конец примера 11.

Определение. ([83]) Говорят, что вхождение *u зависит по значению от вхождения v с носителем k* ($1 \le k \le d$), если существуют такие итерации $\boldsymbol{I}=(I_1, I_2, \ldots, I_{n1})$ и $\boldsymbol{J}=(J_1, J_2, \ldots, J_{n2})$, что:

1. сначала $u[\boldsymbol{I}]$, затем $v[\boldsymbol{J}]$ обращаются к некоторой ячейке памяти S и между этими обращениями в ячейку памяти S не производится запись;
2. для всех $i\in[1, k\text{-}1]$ выполняется равенство $I_i = J_i$;
3. для k–тых компонент векторов итераций выполняется неравенство $I_k < J_k$.

Замечание 3. Определение носителя зависимости по значению можно дать в терминах вектора направления зависимости по значению: вхождение *u зависит*

по значению от вхождения v с носителем k ($1 \le k \le d$), если для вектора направления данной зависимости по значению $\boldsymbol{\Psi}=(\psi_1, \psi_2,\ldots, \psi_d)$ выполняются условия: $\{=\}\subset\psi_i$, при $1\le i\le(k\text{-}1)$ и $\{<\}\subset\psi_k$.

Иногда вместо термина «носитель зависимости по значению» в литературе употребляется эквивалентный термин «уровень зависимости по значению» (*dependence data-flow level*).

Определение. Зависимость по значению вхождения *u* от вхождения *v* называется *циклически независимой*, если существуют такие итерации $\boldsymbol{I}=(I_1, I_2, \ldots, I_{n1})$ и $\boldsymbol{J}=(J_1, J_2, \ldots, J_{n2})$, что:

1. что сначала $u[\boldsymbol{I}]$, затем $v[\boldsymbol{J}]$ обращаются к некоторой ячейке памяти *S* и между этими обращениями в ячейку памяти *S* не производится запись;
2. для всех $i\in[1, d]$ выполняется равенство $I_i = J_i$.

Замечание 4. Определение циклически независимой зависимости по значению можно дать в терминах вектора направления зависимости по значению: зависимость по значению вхождения *u* от вхождения *v* называется циклически независимой, если для вектора направления данной зависимости по значению $\boldsymbol{\Psi}=(\psi_1, \psi_2,\ldots, \psi_d)$ выполняются условия: $\{=\}\subset\psi_i$, при $1\le i\le d$.

Из замечаний 3 и 4 видно, что информация о носителе зависимости по значению и о циклически независимой зависимости по значению может быть получена из вектора направления зависимости по значению.

Пример 12. Рассмотрим фрагмент программы примера 7.

Как было показано в примере 11, вектор направления зависимости по значению *v*1 от *u* равен (=, =). Следовательно, эта зависимость по значению является циклически независимой.

В примере 11 также показано, что вектор направления самозависимости по значению генератора *u* равен (<, =). Следовательно, эта зависимость по значению имеет носитель 1.

Конец примера 12.

<u>Определение.</u> ([83]) Пусть два вхождения u и v имеют d общих циклов. Рассмотрим множество всех итерации $\boldsymbol{I}^k=(I^k_1, I^k_2, \ldots, I^k_{n1})$ и $\boldsymbol{J}^k=(J^k_1, J^k_2, \ldots, J^k_{n2})$, где $1\le k\le m$, $n1\ge d$, $n2\ge d$, таких, что сначала $u[\boldsymbol{I}^k]$, затем $v[\boldsymbol{J}^k]$ обращаются к некоторой ячейке памяти S и между этими обращениями в ячейку памяти S не производится запись. Будем говорить, что множество различных векторов $(J^k_1 - I^k_1, J^k_2 - I^k_2, \ldots, J^k_d - I^k_d)$ образует множество *векторов расстояния зависимости по значению* вхождения v от вхождения u.

Пример 13. Рассмотрим фрагмент программы примера 7.

Все пары итераций $((i_1, j_1), (i_2, j_2))$, при которых $v[(i_2, j_2)]$ зависит по значению от $u[(i_1, j_1)]$, описываются множеством $\{((i', k), (i'', k)) \mid 1\le i'\le M,\ 2\le i''\le M,\ \boldsymbol{i'=i''}\}$ (пример 11). Следовательно, зависимость по значению v от u имеет единственный вектор расстояния (0, 0).

Заметим, что самозависимость по значению генератора u имеет вектор расстояния (1, 0).

Конец примера 13.

1.1.5 Сравнение представлений информационной зависимости

Если в определении информационной зависимости по памяти (пункт 1.1.3) два вхождения зависимы, когда они обращаются к одной и той же ячейке памяти, то в определении зависимости по значению, два вхождения зависимы, когда они гарантированно оперируют с одним и тем же значением. Поэтому, каждое представление информационной зависимости по значению тоньше соответствующего представления информационной зависимости по памяти. Часто в программах, при использовании представлений информационной зависимости по

памяти, информация о потоковых зависимостях может быть смешана с информацией о выходных зависимостях (пример 11). Выходные зависимости иногда могут быть удалены с помощью таких методов как приватизация [83, 85], переименование [65], экспансия массивов [69]. Таким образом, используя представления информационной зависимости по значению можно распараллелить большее количество циклов, чем просто используя представления информационной зависимости по памяти [60, 83].

Представления информационной зависимости по памяти имеют следующее достоинство: для этих представлений разработаны эффективные методы построения [8, 95, 59], которые, хоть и являются приближенными, применимы к большому числу программ и дают хорошие результаты на практике.

Резюмируя данный параграф, отметим следующее

Утверждение. Пусть вхождение v зависит от вхождения u. Тогда:

1. вхождение v может не зависеть по значению от вхождения u (при наличии в программе других генераторов этой же переменной);
2. вектор направления зависимости по значению v от u содержится в векторе направления зависимости v от u;
3. множество носителей зависимости по значению v от u содержится во множестве носителей зависимости v от u;
4. множество векторов расстояния зависимости по значению v от u содержится во множестве векторов расстояния зависимости v от u;

Доказательство следует из того факта, что в представлениях информационной зависимости по значению в общем случае учитывается только подмножество пар итераций, используемых в соответствующих представлениях информационной зависимости.

В таблице 1 приведены данные различных видов представлений для потоковой зависимости использования $v1$ от генератора u в программе примера 7. Эти данные были получены в примерах 7, 8, 9, 11, 12, 13.

Таблица 1.

	Информационная зависимость	Информационная зависимость по значению
Зависимость	есть	есть
Носители (уровни)	1	носители отсутствуют
Циклически независим.	да	да
Векторы направления	(≤, =)	(=, =)
Векторы расстояния	$\begin{bmatrix}0\end{bmatrix}\begin{bmatrix}1\end{bmatrix}\begin{bmatrix}2\end{bmatrix}\ldots\begin{bmatrix}M\text{-}1\end{bmatrix}$ $\begin{bmatrix}0\end{bmatrix}\begin{bmatrix}0\end{bmatrix}\begin{bmatrix}0\end{bmatrix}\ldots\begin{bmatrix}0\end{bmatrix}$	$\begin{bmatrix}0\end{bmatrix}$ $\begin{bmatrix}0\end{bmatrix}$

1.2 Графовые модели информационной зависимости

При анализе программы с целью оптимизации (распараллеливания) удобно рассматривать данные об информационных зависимостях этой программы в виде графов. Существуют как более грубые способы описания зависимостей программы, например, граф информационных зависимостей [8, 33, 36], так и более тонкие, например, решетчатый граф [21, 22, 71] и развертка решетчатого графа (таймирование, *schedule*) [2, 72, 73, 79]. В этом параграфе мы дадим определения и рассмотрим указанные графы.

1.2.1 Граф информационных зависимостей

Граф информационных зависимостей – ориентированный помеченный граф. Вершины графа – вхождения переменных. Если вхождение v зависит от вхождения u, то в графе существует дуга, направленная от u к v. Обычно, с каждой дугой графа связывается некоторая дополнительная информация о зависимых вхождениях, такая как носитель (глубина) зависимости, вектор направления зависимости, вектор расстояния зависимости и др.

Пример 14. На рисунке 1 изображен граф информационных зависимостей фрагмента программы, построенный с помощью OPC.

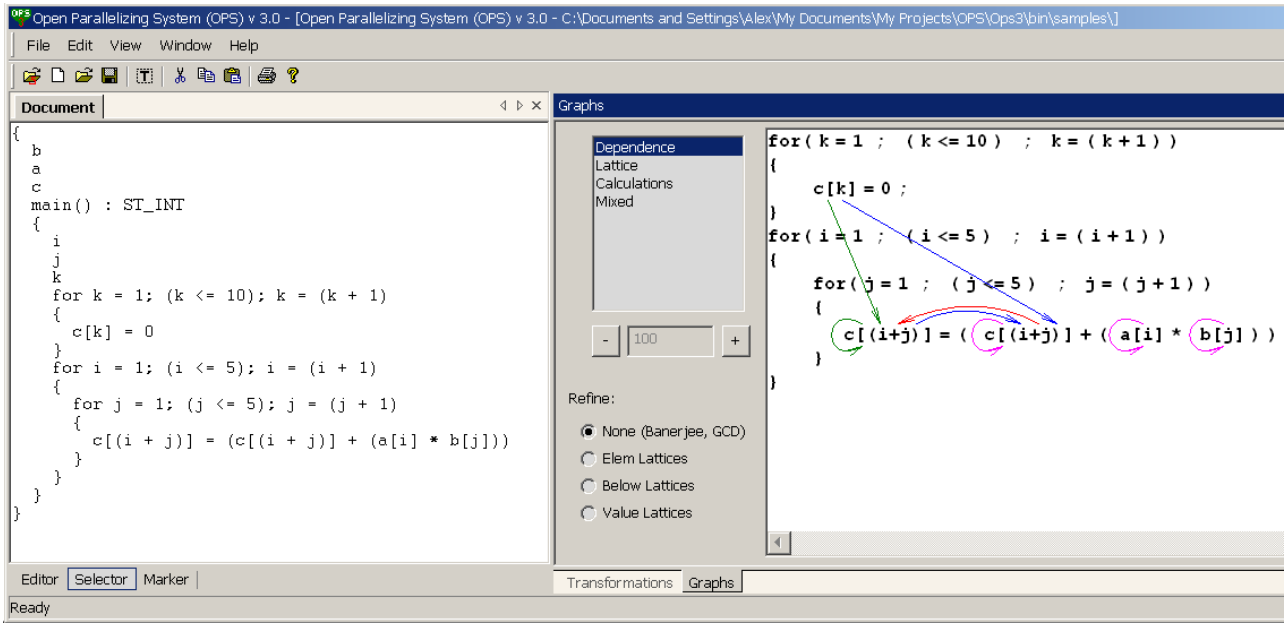


Рис. 1. Экранная форма OPC, изображающая граф информационных зависимостей фрагмента программы.

Конец примера 14.

Замечание. В определении графа используется понятие информационной зависимости (пункт 1.1.3). Если в данном определении вместо понятия информационной зависимости использовать понятие информационной зависимости по значению (пункт 1.1.4), то получим новый граф. Полученный граф назовем *графом информационных зависимостей по значению*. Очевидно, что граф информационных зависимостей по значению является остовным подграфом[5] графа информационных зависимостей.

---

[5] Подграф называется остовным, если множество его вершин совпадает с множеством вершин графа.

1.2.2 Решетчатый граф

Решетчатый граф – одна из наиболее тонких моделей информационной зависимости. Решетчатый граф в частности позволяет для любой операции в правой части оператора присваивания указать итерацию и генератор, выход которого является аргументом данной операции. Описание решетчатых графов в этом параграфе большей частью основано на [21, 22, 23, 24].

1.2.2.1 Опорные пространства. Порядок исполнения операторов

Рассмотрим некоторый фрагмент программы из линейного класса. Занумеруем операторы циклов в этом фрагменте *сверху вниз по тексту программы*. Обозначим счетчик i-го цикла – I_i.

Определение. ([21]) Множество всех вложенных друг в друга циклов, в теле каждого из которых содержится некоторый оператор *Stmt*, называют *опорным гнездом* циклов для оператора *Stmt* и для всех вхождений, которые этот оператор содержит.

Определение. Множество значений вектора счетчиков циклов опорного гнезда, при которых исполняется оператор *Stmt*, будем называть *опорным пространством* для данного оператора и всех вхождений переменных, которые он содержит.

Определение. ([21]) Рассмотрим множество всех условий, задаваемых условными операторами, при которых исполняется оператор *Stmt*. Множество этих условий вырезает из линейного пространства итераций опорного гнезда данного оператора множество выпуклых многогранников. Объединение этих многогранников называется *линейным опорным пространством* оператора *Stmt*.

Пример 15. Рассмотрим фрагмент программы:

```
for(i=1;i<=10;i++)
{   x[i]= 0; // Stmt 1
    for(j=1;j<=10;j++)
      if(i==j)
           x[i]=x[i]+u[i][j]; // Stmt 2
}
```

Рассмотрим опорное гнездо циклов для оператора *Stmt 1*, оно состоит из одного цикла по *i*. Пространство итераций этого гнезда совпадает с опорным пространством оператора *Stmt 1* и определяется как множество целых точек отрезка [1, 10]. Линейное пространство итераций рассматриваемого гнезда и линейное опорное пространство оператора *Stmt 1* определяется системой неравенств:

$-i+1\leq 0$, $i-10\leq 0$.

Рассмотрит опорное гнездо циклов для оператора *Stmt 2*, оно состоит из двух циклов: по *i* и по *j*. Итерационное пространство этого гнезда циклов есть множество целых точек квадрата, определяемого координатами противоположных углов: (1, 1) и (10, 10). Это итерационное пространство НЕ совпадает с опорным пространством оператора *Stmt 2*. Опорное пространство оператора *Stmt 2* – это все целые точки, принадлежащие диагонали соединяющей вершины (1, 1) и (10, 10) указанного квадрата.

Конец примера 15.

Если некоторый оператор присваивания не содержится в гнезде циклов, то можно считать, что этот оператор содержится в цикле, у которого верхняя граница равна нижней. Не умаляя общности, будем считать, что все операторы присваивания содержатся в гнездах циклов.

Далее будем считать, что в каждой строке программы находится только один оператор. Занумеруем все операторы присваивания фрагмента программы сверху вниз по тексту программы и обозначим их через $Stmt_1$, $Stmt_2$, …, $Stmt_m$. Обозначим опорное пространство оператора $Stmt_i$ через V_i, размерность которого равна s_i. Подчеркнем, что пространства V_i и V_j при $i \neq j$ считаются различными, даже если они образованы одними и теми же операторами циклов и условными операторами.

Определение. ([21]) *Пространством итераций фрагмента программы* назовем совокупность опорных пространств V_i для i=1, 2, …, m.

Определение. ([21]) Будем называть исполнение оператора $Stmt_i$ при конкретных значениях счетчиков циклов его опорного гнезда $I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}}$ *срабатыванием* и обозначать $Stmt_i[I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}}]$ или $Stmt_i[\boldsymbol{I}]$ считая, что вектор $\boldsymbol{I}$ равен $(I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}})$.

Каждому срабатыванию $Stmt_i[I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}}]$ поставим в соответствие точку опорного пространства V_i с теми же значениями координат $(I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}})$. Таким образом, устанавливается взаимнооднозначное соответствие между множеством точек пространства итераций фрагмента программы и множеством всех срабатываний всех операторов присваивания этого фрагмента. Заметим, что все точки одного опорного пространства V_i и только они соответствуют срабатываниям одного оператора $Stmt_i$.

Определение. Пусть имеются векторы $\boldsymbol{I}=(I_1, I_2, \ldots, I_{n1})$, и $\boldsymbol{J}=(J_1, J_2, \ldots, J_{n2})$. Пусть $m=min(n1, n2)$. Тогда:

1. если существует $k \in [1, m]$, что $I_1 = J_1$, $I_2 = J_2$, …, $I_{k-1} = J_{k-1}$, $I_k < J_k$, то будем говорить, что вектор $\boldsymbol{I}$ *лексикографически меньше* вектора $\boldsymbol{J}$ ($\boldsymbol{I} <_{lex} \boldsymbol{J}$);
2. если $n1=n2$ и для любого $k \in [1, m]$ $I_k = J_k$, то будем говорить, что вектор $\boldsymbol{I}$ *лексикографически равен* вектору $\boldsymbol{J}$ ($\boldsymbol{I} =_{lex} \boldsymbol{J}$).

Зафиксируем некоторое число d, что $1 \le d \le m$. Если условие 1 выполняется для $k \in [1, d]$, то будем говорить, что вектор ***I*** *лексикографически меньше* вектора ***J*** *по первым d координатам* ($\boldsymbol{I} <^{d}_{lex} \boldsymbol{J}$). Если условие 2 выполняется для $k \in [1, d]$, то будем говорить, что вектор ***I*** *лексикографически равен* вектору ***J*** *по первым d координатам* ($\boldsymbol{I} =^{d}_{lex} \boldsymbol{J}$).

Пример 16. Пусть ***I***=(1, 2, 5) и ***J***=(1, 2, 4, 6). Вектор ***J*** лексикографически меньше вектора ***I***, поскольку при совпадающих первых двух координатах, третья координата вектора ***J*** строго меньше, третьей координаты вектора ***I***. Значения всех последующих координат уже не важны. Для данного примера k=3. Векторы ***I*** и ***J*** лексикографически равны по первым двум координатам ($\boldsymbol{I} =^{2}_{lex} \boldsymbol{J}$).

Конец примера 16.

При выполнении линейной программы на «последовательном» компьютере все ее срабатывания $Stmt_i[\boldsymbol{I}]$ выполняются в строго определенном порядке. Для его описания введем отношение порядка в пространстве итераций фрагмента программы. Этот порядок будем называть *лексикографическим*, и обозначать символом $<_{lex}$.

Рассмотрим любую пару различных векторов $\boldsymbol{I}=(I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}}) \in V_i$ и $\boldsymbol{J}=(I_{j_1}, I_{j_2}, \ldots, I_{j_{s_j}}) \in V_j$. Так как векторы ***I*** и ***J*** могут принадлежать разным пространствам (при $i \ne j$), то при определении лексикографического порядка в пространстве итераций фрагмента программы будут использоваться не только отношения между компонентами векторов ***I*** и ***J***, но и отношение между номерами i и j опорных пространств V_i и V_j. Будем говорить, что ***I*** *лексикографически меньше* ***J*** ($\boldsymbol{I} <_{lex} \boldsymbol{J}$), если:

1. либо $i<j$ и пересечение совокупностей номеров $i_1, i_2, \ldots, i_{s_i}$ и $j_1, j_2, \ldots, j_{s_j}$ пусто; в этом случае операторы $Stmt_i$ и $Stmt_j$ не имеют общих циклов, и оператор $Stmt_i$ находится по тексту программы раньше, чем оператор $Stmt_j$;
2. либо $i<j$ и $\boldsymbol{I}=^d_{lex}\boldsymbol{J}$, где d ($d\geq 1$) – количество общих циклов для операторов $Stmt_i$ и $Stmt_j$; в этом случае $Stmt_i[I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}}]$ и $Stmt_j[I_{j_1}, I_{j_2}, \ldots, I_{j_{s_j}}]$ исполняются на одной итерации общего гнезда циклов, и $Stmt_i$ находится по тексту раньше, чем $Stmt_j$;
3. либо $\boldsymbol{I}<^d_{lex}\boldsymbol{J}$, где d ($d\geq 1$) – количество общих циклов для операторов $Stmt_i$ и $Stmt_j$; в этом случае $Stmt_i[I_{i_1}, I_{i_2}, \ldots, I_{i_{s_i}}]$ исполняется на более ранней итерации, чем $Stmt_j[I_{j_1}, I_{j_2}, \ldots, I_{j_{s_j}}]$.

Таким образом, если $\boldsymbol{I}\in V_i$ лексикографически меньше $\boldsymbol{J}\in V_j$, то при последовательном исполнении фрагмента линейной программы срабатывание $Stmt_i[\boldsymbol{I}]$ произойдет раньше, чем срабатывание $Stmt_j[\boldsymbol{J}]$.

1.2.2.2 Максимальные и минимальные решетчатые графы

Определим максимальный решетчатый граф [21, с. 346] фрагмента программы.

<u>Определение.</u> ([21]) Множество вершин *максимального решетчатого графа* – все точки пространства итераций фрагмента программы. Дуга направляется из вершины $\boldsymbol{I}\in V_i$ в вершину $\boldsymbol{J}\in V_j$, если $\boldsymbol{I}<_{lex}\boldsymbol{J}$ и существуют такие вхождения $u\in Stmt_i$ и $v\in Stmt_j$, что $u[\boldsymbol{I}]$ и $v[\boldsymbol{J}]$ обращаются к одной и той же ячейке памяти. При этом говорят, что вхождение u определяет начало дуги, а вхождение v – конец дуги.

Будем считать, что каждая пара вхождений (u, v) определяет отдельную дугу из $\boldsymbol{I}$ в $\boldsymbol{J}$. Таким образом, максимальный решетчатый граф может иметь кратные дуги.

Из определения следует, что максимальный решетчатый граф есть представление информационной зависимости по памяти.

<u>Определение.</u> ([21]) Вершины *максимального решетчатого графа потоковой зависимости* – пространство итераций фрагмента программы. Дуга направлена из вершины $\boldsymbol{I}$ в вершину $\boldsymbol{J}$ ($\boldsymbol{I} \in V_i$, $\boldsymbol{J} \in V_j$, $\boldsymbol{I} <_{lex} \boldsymbol{J}$), если существуют генератор $u \in Stmt_i$ и использование $v \in Stmt_j$, что $u[\boldsymbol{I}]$ и $v[\boldsymbol{J}]$ обращаются к одной ячейке памяти.

Аналогичным образом можно ввести максимальные решетчатые графы выходной, анти- и входной зависимости.

Пример 17. На рисунке 2 изображена часть максимального решетчатого графа потоковой зависимости фрагмента программы умножения многочленов примера 5.

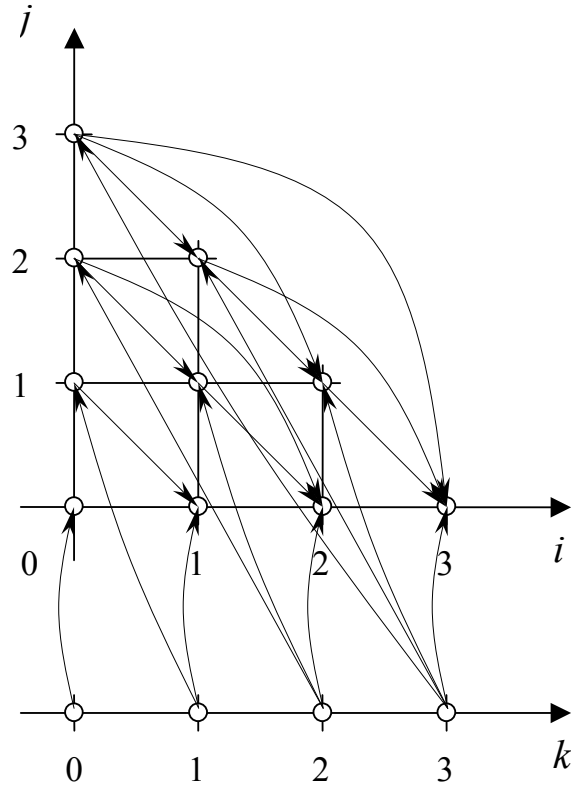


Рис. 2. Часть максимального решетчатого графа программы.

Конец примера 17.

<u>Определение</u> *минимальных решетчатых графов* дадим конструктивно ([21]).

Возьмем максимальный решетчатый граф зависимости типа *T*. Для каждой вершины *s* этого графа выполним:

1. разобьем множество дуг, *входящих в вершину s* на группы: к одной группе отнесем дуги, конец которых определялся одним и тем же вхождением;
2. в каждой группе выберем одну дугу, у которой *начальная* вершина лексикографически ближе всего к вершине *s*; остальные дуги удалим.

Полученный граф называется *минимальным снизу решетчатым графом*[6] зависимости типа *T* [21, с. 346].

Пример 18. На рисунке 3 изображена часть минимального снизу решетчатого графа потоковой зависимости для фрагмента программы умножения многочленов примера 5. Он получен из максимального графа примера 17 удалением некоторых дуг (удаленные дуги выделены пунктиром).

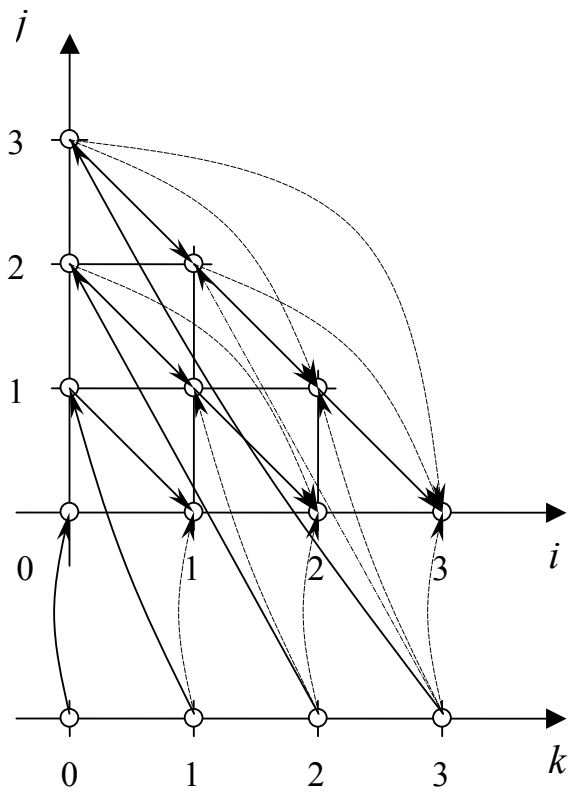


Рис. 3. Часть минимального снизу решетчатого графа потоковой зависимости.

Конец примера 18.

---

[6] в работе [21] минимальный снизу решетчатый граф потоковой зависимости называется *графом алгоритма*.

Снова возьмем максимальный решетчатый граф зависимости типа *T*. Для каждой вершины *s* этого графа выполним:

1. разобьем множество дуг, *выходящих из вершины s* на группы: к одной группе отнесем дуги, начало которых определялось одним и тем же вхождением;
2. в каждой группе выберем одну дугу, у которой *конечная* вершина лексикографически ближе всего к вершине *s*; остальные дуги удалим.

Полученный граф называется *минимальным сверху решетчатым графом* зависимости типа *T* [21, с. 347].

Пример 19. Минимальный сверху решетчатый граф потоковой зависимости для фрагмента программы примера 5 совпадает с минимальным снизу решетчатым графом потоковой зависимости этого же фрагмента.

Конец примера 19.

Для зависимостей по выходу и по входу минимальные снизу и сверху решетчатые графы совпадают [21, с. 347]. В общем случае для потоковой и антизависимости минимальные сверху и снизу решетчатые графы различаются [21, с. 347]. Проиллюстрируем это.

Потоковая зависимость. Пусть в некоторой программе к ячейке *X* некоторого массива обращаются только два вхождения. Пусть все *записи* в ячейку *X* производятся на итерациях $\boldsymbol{I}_1$, $\boldsymbol{I}_2$, $\boldsymbol{I}_3$, где $\boldsymbol{I}_1 <_{lex} \boldsymbol{I}_2 <_{lex} \boldsymbol{I}_3$. Пусть все *чтения* из ячейки *X* происходят на итерациях $\boldsymbol{J}_1$, $\boldsymbol{J}_2$, где $\boldsymbol{I}_1 <_{lex} \boldsymbol{I}_2 <_{lex} \boldsymbol{I}_3 <_{lex} \boldsymbol{J}_1 <_{lex} \boldsymbol{J}_2$. Максимальный решетчатый граф потоковой зависимости G_{Max} будет содержать следующие дуги, входящие в вершину $\boldsymbol{J}_1$: ($\boldsymbol{I}_1$, $\boldsymbol{J}_1$), ($\boldsymbol{I}_2$, $\boldsymbol{J}_1$), ($\boldsymbol{I}_3$, $\boldsymbol{J}_1$). Рассмотрим минимальный снизу решетчатый граф потоковой зависимости $\underline{G}_{Min}$ данной программы. В этом графе будет только одна дуга, входящая в вершину $\boldsymbol{J}_1$: ($\boldsymbol{I}_3$, $\boldsymbol{J}_1$) (т.к. $\boldsymbol{I}_3$ лексикографически самая близкая к $\boldsymbol{J}_1$ среди вершин $\boldsymbol{I}_1$, $\boldsymbol{I}_2$, $\boldsymbol{I}_3$). Аналогично рассуждая

для вершины $\boldsymbol{J}_2$, получим, что рассматриваемый граф $\underline{G}_{Min}$ содержит дугу ($\boldsymbol{I}_3$, $\boldsymbol{J}_2$). На рисунке 4.а изображены указанные дуги графа $\underline{G}_{Min}$, при этом пунктиром отмечены дуги максимального графа G_{Max} входящие в вершины $\boldsymbol{J}_1$, $\boldsymbol{J}_2$, и не являющиеся дугами минимального снизу графа $\underline{G}_{Min}$.

Теперь рассмотрим минимальный сверху решетчатый граф потоковой зависимости $\overline{G}_{Min}$ для этой же программы. Рассмотрим дуги максимального графа G_{Max}, выходящие из вершины $\boldsymbol{I}_1$: ($\boldsymbol{I}_1$, $\boldsymbol{J}_1$), ($\boldsymbol{I}_1$, $\boldsymbol{J}_2$). Из этих дуг граф $\overline{G}_{Min}$ будет содержать только ($\boldsymbol{I}_1$, $\boldsymbol{J}_1$) т.к. $\boldsymbol{J}_1$ лексикографически самая близкая к $\boldsymbol{I}_1$ среди $\boldsymbol{J}_1$, $\boldsymbol{J}_2$. Аналогично рассуждая для вершин $\boldsymbol{I}_2$ и $\boldsymbol{I}_3$, получим, что минимальный сверху граф $\overline{G}_{Min}$ содержит дуги ($\boldsymbol{I}_2$, $\boldsymbol{J}_1$) и ($\boldsymbol{I}_3$, $\boldsymbol{J}_1$). На рисунке 4.б изображены указанные дуги минимального сверху графа $\overline{G}_{Min}$, при этом пунктиром отмечены дуги максимального графа G_{Max} выходящие из вершин $\boldsymbol{I}_1$, $\boldsymbol{I}_2$, $\boldsymbol{I}_3$, и не являющиеся дугами минимального сверху графа G_{Min}. Видно, что минимальный снизу и минимальный сверху решетчатые графы потоковой зависимости различаются.

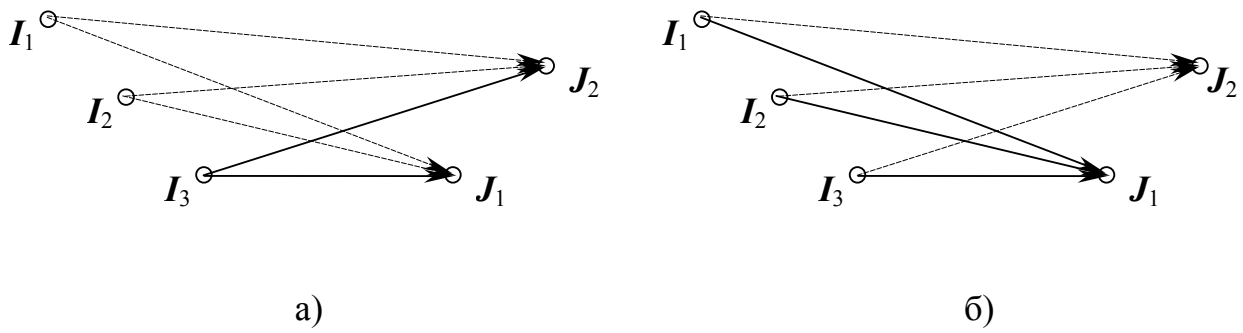


Рис. 4. Различия между минимальным снизу (а) и сверху (б) решетчатыми графами.

Антизависимость. Пусть все *чтения* из ячейки X производятся на итерациях $\boldsymbol{I}_1$, $\boldsymbol{I}_2$, $\boldsymbol{I}_3$, где $\boldsymbol{I}_1 <_{lex} \boldsymbol{I}_2 <_{lex} \boldsymbol{I}_3$. Пусть все *записи* в ячейку X происходят на итерациях $\boldsymbol{J}_1$, $\boldsymbol{J}_2$, где $\boldsymbol{I}_1 <_{lex} \boldsymbol{I}_2 <_{lex} \boldsymbol{I}_3 <_{lex} \boldsymbol{J}_1 <_{lex} \boldsymbol{J}_2$. Рассуждая также как и в случае потоковой зависимости можно увидеть, что на рисунке 4.а изображены дуги минимального

снизу решетчатого графа антизависимости, входящие в вершины $\boldsymbol{J}_1$, $\boldsymbol{J}_2$. А на рисунке 4.б изображены дуги минимального сверху решетчатого графа антизависимости, выходящие из вершин $\boldsymbol{I}_1$, $\boldsymbol{I}_2$, $\boldsymbol{I}_3$.

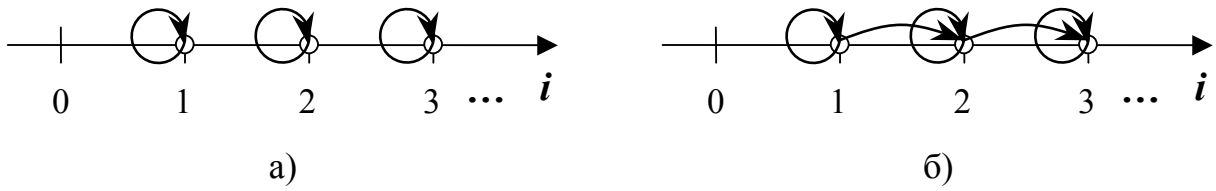
Пример 20. (Фрагмент программы, в котором минимальные снизу и сверху решетчатые графы антизависимости различаются.)

Рассмотрим следующий цикл:

```
for(i=1; i<=1000; i++)

{   a[i]=x[i+1];
          u1
    b[i]=x[i];
          u2
    x[i]=c[i]*d[i];
     v
}
```

В этом цикле имеется антизависимость вхождения v от вхождений $u1$, $u2$. Других антизависимостей нет. Часть минимального снизу решетчатого графа антизависимости $\underline{G}$ для данного цикла изображена на рисунке 5.а, минимального сверху решетчатого графа антизависимости $\overline{G}$ изображена на рисунке 5.б. Опорные пространства различных операторов на рисунках совмещены.



а) б)

Рис. 5. Решетчатые графы антизависимости: а) минимальный снизу $\underline{G}$ и б) минимальный сверху $\overline{G}$.

Из приведенного рисунка видно, что рассматриваемые графы $\underline{G}$ и $\bar{G}$ различаются.

Конец примера 20.

Из приведенной выше иллюстрации видно, что минимальный снизу решетчатый граф потоковой зависимости описывает зависимость по значению т.к. между двумя обращениями к одной ячейке памяти, которые порождают дугу, эта ячейка не перезаписывается. Это же относится и к минимальному сверху решетчатому графу антизависимости. Но, данная иллюстрация также показывает, что минимальный сверху решетчатый граф потоковой зависимости и минимальный снизу решетчатый граф антизависимости не описывают зависимость по значению. Отметим также, что ни минимальный снизу, ни минимальный сверху решетчатый граф входной зависимости не описывает зависимость по значению. Это происходит из-за того, что в определении этих графов никак не учитываются записи в память генераторами. Чтобы минимальный решетчатый граф описывал входную зависимость по значению, необходимо при его построении учесть записи генераторов в память.

Заметим, что любой минимальный решетчатый граф не является представлением информационной зависимости по памяти т.к. при его построении часть обращений к памяти вхождениями отбрасывается.

Тот факт, что с помощью решетчатых графов можно точно описывать зависимость по значению между операциями в программе дает возможность предполагать, что именно на основе этих графов можно реализовать компилятор, эффективно преобразующий последовательные программы в код потоковой машины (dataflow machine) [16, 17, 44], разрабатываемой ИПИ РАН [18, 40, 43, 45].

1.2.2.3 Простые и элементарные решетчатые графы

Для того, чтобы получить алгоритмы построения минимальных решетчатых графов, вводится понятие простого и элементарного решетчатого графа [21, с. 351]. Эти понятия более удобны, чем понятие минимального решетчатого графа, они будут широко использоваться в данной работе.

Рассмотрим программу из линейного класса. Возьмем какую-либо дугу (***I***, ***J***) минимального снизу решетчатого графа потоковой зависимости. В определении этой дуги участвует использование, читающее из памяти на итерации ***J***, и генераторы соответствующей переменной.

Исходя из данной программы, построим новую программу по следующим правилам. Зафиксируем какое-нибудь использование некоторой переменной *X* в операторе *Stmt*. В операторе *Stmt* вычеркнем всю правую часть кроме зафиксированного вхождения. Во всех остальных операторах присваивания полностью вычеркнем правые части. В левых частях операторов присваивания вычеркнем все вхождения переменных, имена которых отличны от *X*. Теперь вычеркнем все операторы присваивания, у которых оказались пустыми левые и правые части, все условные операторы, охватывающие только пустые операторы, и все циклы с пустыми телами.

С формальной точки зрения оставшаяся часть программы не является программой на языке С, т.к. левые и правые части некоторых операторов могут быть пустыми. Эту часть программы можно сделать программой на языке С, если пустые части заменить вхождениями отличных между собой переменных (чтобы не возникло новых зависимостей), не совпадающих с *X*.

Любая программа такого типа называется *простой*. Построенный для нее минимальный снизу решетчатый граф называется *простым снизу решетчатым графом*. Далее в работе, если специально не оговорено, под *простым решетчатым графом* будем понимать простой снизу решетчатый граф.

Пример 21. Зафиксировав вхождение *v*1 в программе примера 5, и применив алгоритм построения простой программы, получим:

```
for(k=0; k<=(N+M); k++)

    c[k]=*;
     u1
for(i=0; i<=M; i++)

    for(j=0; j<=N; j++)

      c[i+j]=c[i+j];
       u2        v1
```

Часть простого решетчатого графа потоковой зависимости изображена на рисунке 3 т.к. в данном случае этот граф совпадает со всем минимальным снизу решетчатым графом потоковой зависимости программы примера 5.

Конец примера 21.

Каждый простой решетчатый граф описывает зависимость[7] одного фиксированного вхождения от набора других вхождений. Поэтому далее в работе простой решетчатый граф будет идентифицироваться вхождениями, зависимость между которыми он описывает. Например, простой решетчатый граф, построенный с помощью программы примера 21, будет идентифицироваться как простой решетчатый граф, описывающий зависимость вхождения *v*1 от вхождений *u*1, *u*2.

---

[7] эта зависимость не является информационной зависимостью по памяти т.к. минимальный решетчатый граф не описывает зависимость по памяти. Поэтому в данном случае, и в аналогичных случаях далее, после слова «зависимость» не предполагаются слова «по памяти» (как в предложении пункта 1.1.3).

<u>Утверждение 1.</u> ([21, с. 353]) Для любой программы из линейного класса минимальный решетчатый граф любого типа есть объединение минимальных графов того же типа всех простых программ.

Определение простой программы немного меняется в зависимости от того, какой тип зависимости и какой из минимальных графов (сверху или снизу) рассматривается. Для минимального снизу решетчатого графа потоковой зависимости простая программа имеет одно фиксированное использование и все генераторы этой же переменной. Для минимального сверху решетчатого графа потоковой зависимости и минимального снизу решетчатого графа антизависимости простая программа имеет один фиксированный генератор и все использования этой же переменной. Аналогично, простая программа определяется для других видов минимальных решетчатых графов.

Процесс расщепления линейных программ и соответствующих им минимальных решетчатых графов можно продолжить. В получившейся ранее простой программе вычеркнем все операторы присваивания, кроме оператора, содержащего фиксированное использование переменной X, и какого-нибудь оператора, содержащего генератор переменной X. В частности, второй оператор может совпадать с первым. Теперь вычеркнем все условные операторы, охватывающие только пустые операторы, и все циклы с пустыми телами.

Любая программа такого типа называется *элементарной* [21]. Построенный для нее минимальный снизу решетчатый граф называется *элементарным снизу решетчатым графом* [21]. Далее в работе, если специально не оговорено, то под элементарным решетчатым графом будем понимать элементарный снизу решетчатый граф.

Пример 22. Зафиксировав вхождение $v1$ в простой программе примера 19, и применив описанный выше алгоритм, получим две элементарные программы.
Элементарная программа А:

```
for(k=0; k<=(N+M); k++)

    c[k]=*;
     u1
for(i=0; i<=M; i++)

    for(j=0; j<=N; j++)

       *=c[i+j];
              v1
```

Элементарная программа Б:

```
for(i=0; i<=M; i++)

    for(j=0; j<=N; j++)

      c[i+j]=c[i+j];
       u2        v1
```

Часть элементарного решетчатого графа потоковой зависимости, построенного по элементарной программе А, изображена на рисунке 6.а; построенного по элементарной программе Б – на рисунке 6.б.

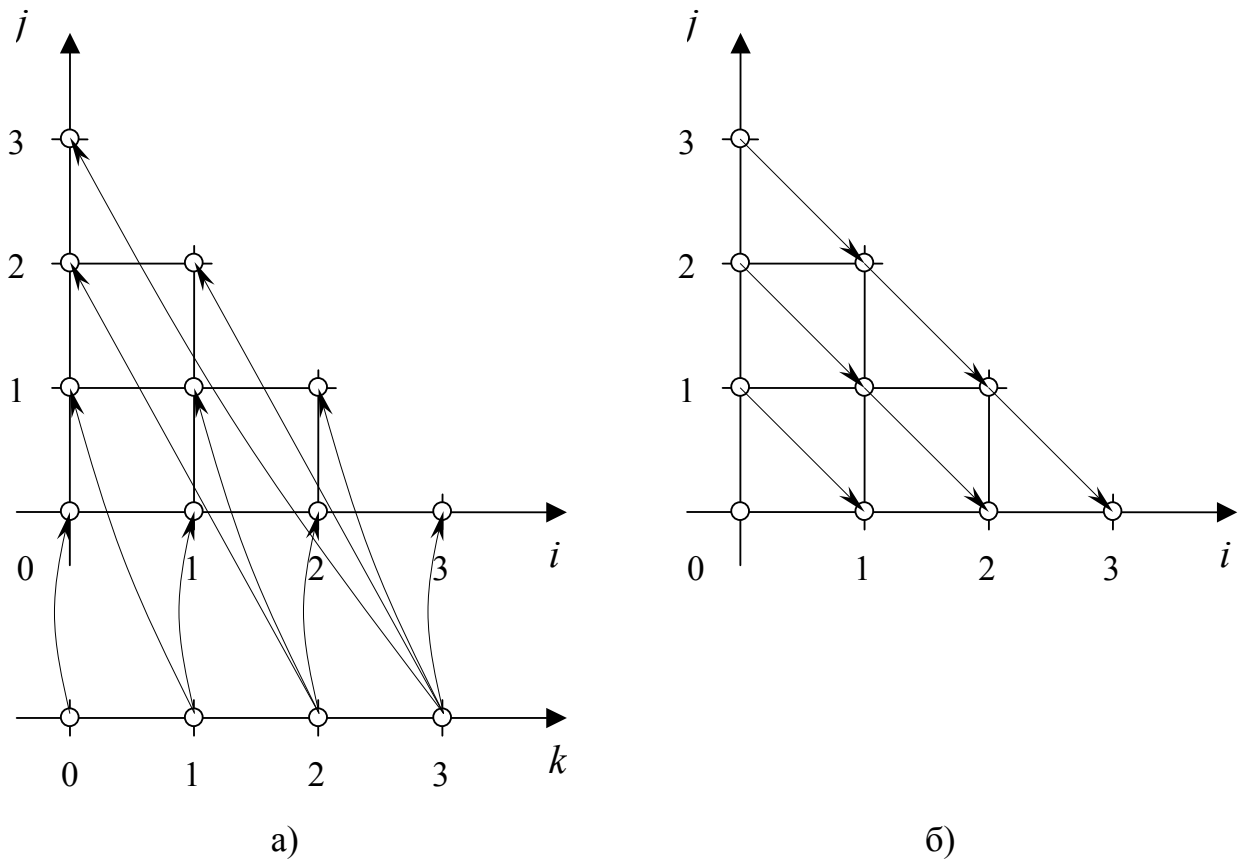


Рис. 6. Элементарный решетчатый граф, построенный по: а) элементарной программе А, б) элементарной программе Б.

Конец примера 22.

Далее в работе элементарный решетчатый граф будет идентифицироваться вхождениями, зависимость между которыми он описывает. Например, элементарный решетчатый граф, построенный с помощью элементарной программы 1 примера 22, будет идентифицироваться как элементарный решетчатый граф, описывающий зависимость вхождения $v1$ от вхождения $u1$.

<u>Утверждение 2.</u> ([21, с. 354]) Для любой программы из линейного класса минимальный решетчатый граф любого типа есть остовный подграф объединения минимальных графов того же типа всех элементарных программ.

1.2.2.4 Хранение решетчатых графов

По способам хранения нет принципиальной разницы между минимальными снизу и сверху решетчатыми графами. Поэтому далее в этой части будем рассматривать только минимальные снизу решетчатые графы, которые в отсутствии оговорок будем называть минимальными решетчатыми графами.

Для хранения минимального решетчатого графа не подходят такие традиционные методы как матрица смежности или список дуг графа. Это происходит из-за того, что множество вершин минимального решетчатого графа совпадает с итерационным пространством программы. Затраты памяти на хранение такого количества вершин или дуг для реальных программ не приемлемы. Кроме того, время просмотра такого графа будет сопоставимо со временем исполнения программы, что также недопустимо. Для представления минимальных решетчатых графов используются другие методы.

Для упрощения изложения, далее в этой части без дополнительных оговорок будем рассматривать решетчатые графы только потоковой зависимости.

Любой минимальный решетчатый граф может быть представлен как объединение всех простых решетчатых графов (утв. 1). Простой решетчатый граф можно получить из объединения элементарных решетчатых графов, удалив некоторые дуги (утв. 2). И в простом и в элементарном решетчатом графе в одну вершину может входить не более одной дуги. В [21, 70] была предложена идея хранить дуги простого и элементарного решетчатых графов в виде функции F, аргумент у которой – вершина конца дуги, а возвращаемое значение – вершина начала дуги. Эта функция определена на множестве вершин решетчатого графа H, в которые входят дуги. В общем случае функция F является кусочной[8]. По-

---

[8] кусочной называется функция, которая может быть задана на разных подмножествах области определения разными аналитическими выражениями.

этому функцию F представляют в виде набора функций F_k (k=1..n), которые определены на множествах H_k (k=1..n). Множества H_k (k=1..n) образуют разбиение множества H. Далее в работе будем говорить о наборе функций, описывающих дуги решетчатого графа. Вершины графа представляются как все целые точки некоторых областей H_k, каждая из которых задана системой неравенств.

Дуги элементарного решетчатого графа, описывающего зависимость использования $v \in Stmt_j$ от генератора $u \in Stmt_i$, храняться в виде набора функций F_k (k=1..n). Эти функции определены на подмножествах H_k (k=1..n) опорного пространства V_j, в которые входят дуги решетчатого графа. Функция F_k, определенная на H_k, ставит в соответствие вершине $\boldsymbol{J} \in H_k$ начальную вершину $\boldsymbol{I} \in V_i$ единственной дуги входящей в $\boldsymbol{J}$.

Часто в реальных программах, дуги элементарного решетчатого графа могут быть описаны функциями, для которых выполняются условия:

C.1) функции имеют вид $\boldsymbol{I}=\boldsymbol{A}*\boldsymbol{J}+\boldsymbol{b}$, где $\boldsymbol{A}$ – числовая матрица, вектор $\boldsymbol{b}$ в общем случае линейно зависит от внешних переменных;

C.2) функции заданы на всех целых точках некоторых выпуклых линейных многогранников;

C.3) количество функций и выпуклых линейных многогранников, на которых задана отдельная функция, конечно, не зависит от значений внешних переменных и не пропорционально размерам опорного пространства.

Если исходный фрагмент программы содержит внешние переменные, то функции, описывающие дуги некоторого решетчатого графа для этого фрагмента, могут быть параметризованны этими переменными.

Пример 23. Дуги элементарного решетчатого графа, описывающего зависимость $v1$ от $u1$ в фрагменте программы примера 5, определяются одной функцией $F_{u1,v1}(i, j)=i+j$. Область определения этой функции $H_{u1,v1}$ есть все целые точки многогранника, заданного системой неравенств:

$$\begin{cases} i \geq 0 \\ i \leq M \\ j \geq 0 \\ j \leq N. \end{cases}$$

Дуги элементарного решетчатого графа, описывающего зависимость $v1$ от $u2$ во фрагменте программы примера 5, определяются одной функцией $F_{u2,v1}(i, j)=(i-1, j+1)$. Область определения этой функции $H_{u2,v1}$ есть все целые точки многогранника, заданного системой неравенств:

$$\begin{cases} i \geq 1 \\ i \leq M \\ j \geq 0 \\ j \leq N\text{-}1. \end{cases}$$

Дуги каждого из рассмотренных элементарных решетчатых графов описываются только одной функцией. Для потоковой зависимости в программе примера 43 (см. пункт 2.2.3.1) дуги элементарного решетчатого графа описываются несколькими функциями.

Конец примера 23.

Однако существуют программы, принадлежащие линейному классу, в которых дуги элементарного решетчатого графа некоторой зависимости не могут быть описаны функциями, удовлетворяющими условиям С.1) – С.3).

Пример 24. Рассмотрим элементарную программу.

```
for(i=1; i<=N; i++)

   a[2*i] = a[i] ;
    u        v
```

Элементарный решетчатый граф G, описывающий зависимость v от u, изображен на рис. 7.

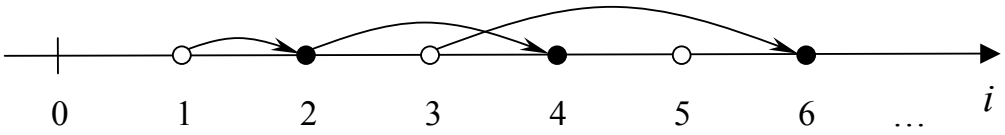


Рис. 7. Элементарный решетчатый граф, который не описывается функциями, удовлетворяющими условиям С.1) – С.3).

Область определения функции, задающей дуги графа G, состоит только из четных точек отрезка [1, N]. Эта область не может быть описана совокупностью систем линейных неравенств, количество которых не зависит от N. Следовательно, описать дуги графа G функциями, удовлетворяющими условиям С.1) – С.3), нельзя.

Дуги решетчатого графа G описываются функцией $F(i)=i\div 2$, область определения которой задается системой ограничений: $i \geq 2$, $i \leq N$, $i - 2*(i\div 2)=0$, где знаком «÷» обозначена операция целочисленного деления[9].

Конец примера 24.

<u>Определение.</u> ([71]) *Квазиаффинная форма* – это форма, полученная из параметров и целочисленных констант с помощью операций сложения, вычитания, умножения на целое, деления на целое.

Дуги элементарного решетчатого графа любой зависимости в любой линейной программе могут быть описаны функциями, заданными с помощью квазиаффинных форм [70], [71] (некоторые из форм, описывающих функции или/и их области определения, есть квазиаффинные формы).

---

[9] $a\div b = \lfloor a/b \rfloor$ (ближайшее целое, не превосходящее a/b).

<u>Определение.</u> ([21]) Говорят, что элементарный решетчатый граф, дуги которого описываются функцией F, *покрывается* системой функций F_k, если при любых значениях внешних переменных для любой вершины $\boldsymbol{J}$, в которой определена функция F, существует подсистема функций F_{k_i}, составленная из всех определенных в $\boldsymbol{J}$ функций системы F_k, что значение $F(\boldsymbol{J})$ совпадает со значениями функций $F_{k_i}(\boldsymbol{J})$.

Отметим, что система покрывающих функций может описывать дуг больше, чем имеется в элементарном решетчатом графе.

Так как простой решетчатый граф есть объединение элементарных решетчатых графов (с отбрасыванием некоторых дуг), то простые решетчатые графы хранятся аналогично элементарным решетчатым графам. Все вышесказанное относительно вида функций, описывающих дуги элементарного графа, относится и к простому решетчатому графу. Единственно, с каждой функцией F_k, описывающей дуги простого решетчатого графа, связывается генератор, записи в память которого определяют начала дуг, описываемых F_k. Аналогично описывается и минимальный решетчатый граф. С каждой функцией F_l этого графа связывается пара вхождений, обращения к памяти которых порождают дуги графа, описываемые этой функцией.

Пример 25. Пусть $H^1_{u1,v1}$ – множество всех целых точек многогранника, задаваемого системой неравенств: $i \geq 0$, $i \leq 0$, $j \geq 0$, $j \leq N$. Пусть $H^2_{u1,v1}$ – множество всех целых точек многогранника, задаваемого системой неравенств: $i \geq 1$, $i \leq M$, $j \geq N$, $j \leq N$. Тогда дуги простого решетчатого графа, описывающего зависимость вхождения $v1$ от вхождений $u1$, $u2$ в программе примера 5, задаются тройками: $(u1, H^1_{u1,v1}, F_{u1,v1})$, $(u1, H^2_{u1,v1}, F_{u1,v1})$, $(u2, H_{u2,v1}, F_{u2,v1})$ (обозначения $F_{u1,v1}$, $H_{u2,v1}$, $F_{u2,v1}$ введены в примере 21).

Конец примера 25.

1.2.3 Развертка решетчатого графа (таймирование, schedule)

Совокупность минимальных решетчатых графов дает информацию о порядке исполнения итераций программы, который должен быть сохранен при модификации программы. В частности, в совокупности этих графов можно выделить множество слоев[10] и параллельные множества[11] вершин. Все итерации в каждом слое, можно исполнять в любом порядке, в том числе и параллельно (в этом случае, возможно существование ограничений на порядок исполнения слоев относительно друг друга). Каждое множество итераций в наборе параллельных множеств можно исполнять независимо от других множеств (в этом случае, возможно существование ограничений на порядок исполнения итераций внутри самих множеств).

Имеются принципиально иные, по сравнению с указанными ранее, способы представления минимальных решетчатых графов, например, с помощью разверток[12] [21, 24, 72, 73]. С помощью разверток, в частности, можно получить описания слоев и параллельных множеств.

Определение. ([21, с. 394]). Пусть G – граф, полученный объединением минимальных решетчатых графов потоковой, анти- и выходной зависимости по значению программы. Рассмотрим вещественную функцию f, определенную на вершинах графа G. Будем говорить, что функция f возрастает (не убывает) вдоль дуг графа G, если $f(s)<f(t)$ ($f(s)\leq f(t)$) для всех вершин s, t таких, что в графе G су-

---

[10] множество вершин графа называется *слоем*, если никакие его вершины не связаны путем графа.

[11] множества вершин графа называются *параллельными*, если никакие две вершины из разных множеств не связаны путем графа [21].

[12] вместо термина «развертка» в работах [72, 73] используется термин *schedule*, а в работе [34] -- таймирующая функция.

ществует дуга из вершины s в вершину t. Функция f называется *строгой* (*обобщенной*) *разверткой* графа G, если она строго возрастает (не убывает) вдоль дуг графа G.

Пусть известна какая-нибудь строгая развертка графа G. Тогда на любой ее поверхности уровня никакие точки пространства итераций не могут быть связаны ни дугами, ни путями графа G [21, с. 394]. Таким образом, каждая поверхность уровня строгой развертки определяет слой.

Для одного графа может существовать набор разверток. Любая развертка позволяет расщепить пространство итераций на множества. Однако от развертки зависит количество этих множеств и количество точек во множествах, характер связей между множествами и между точками во множествах. Строгие развертки обеспечивают отсутствие связей внутри множеств. Развертки, которые обеспечивают отсутствие связей между множествами, называются *расщепляющими* [21, с. 395]. Расщепляющая развертка f удовлетворяет следующему условию: если в графе G существует дуга из вершины s в вершину t, то $f(s)=f(t)$.

Пример 26. Рассмотрим фрагмент программы, состоящий из двумерного гнезда циклов программы примера 5. С помощью этого двумерного гнезда осуществляются основные вычисления при умножении многочленов. Часть графа G, полученного объединением минимальных решетчатых графов потоковой, анти- и выходной зависимости по значению программы, изображена на рисунках 8.а и 8.б.

Расщепляющей разверткой для графа G будет следующая функция $f_1(i, j)=i+j$, определенная на всем пространстве итераций рассматриваемого гнезда. Каждое множество итераций данного гнезда, принадлежащее одной и той же поверхности уровня f_1, можно исполнять независимо от других таких множеств (т.е. все эти множества можно исполнять одновременно). На рисунке 8.а ука-

занные множества выделены серыми овалами, а каждая вершина подписана значением развертки f_1 в ней.

Строгой разверткой для графа G будет следующая функция $f_2(i, j)=i-j$, определенная на всем пространстве итераций рассматриваемого гнезда. Все итерации данного гнезда, принадлежащие одной и той же поверхности уровня f_2, можно исполнять в любом порядке, в том числе и одновременно. Единственно, выполнять каждое независимое множество нужно строго в определенном порядке: начинать выполнение итераций, расположенных на прямой $i–j=c$, можно только в том случае, если уже полностью завершено выполнение итераций, расположенных на прямой $i-j=c-1$. На рисунке 8.б эти множества выделены серыми овалами, а каждая вершина подписана значением развертки f_2 в ней. Пунктирными линиями изображены необходимые синхронизационные барьеры [79] (запрещающие исполнение множества итераций до тех пор, пока не закончилось исполнение всех итераций предыдущего множества).

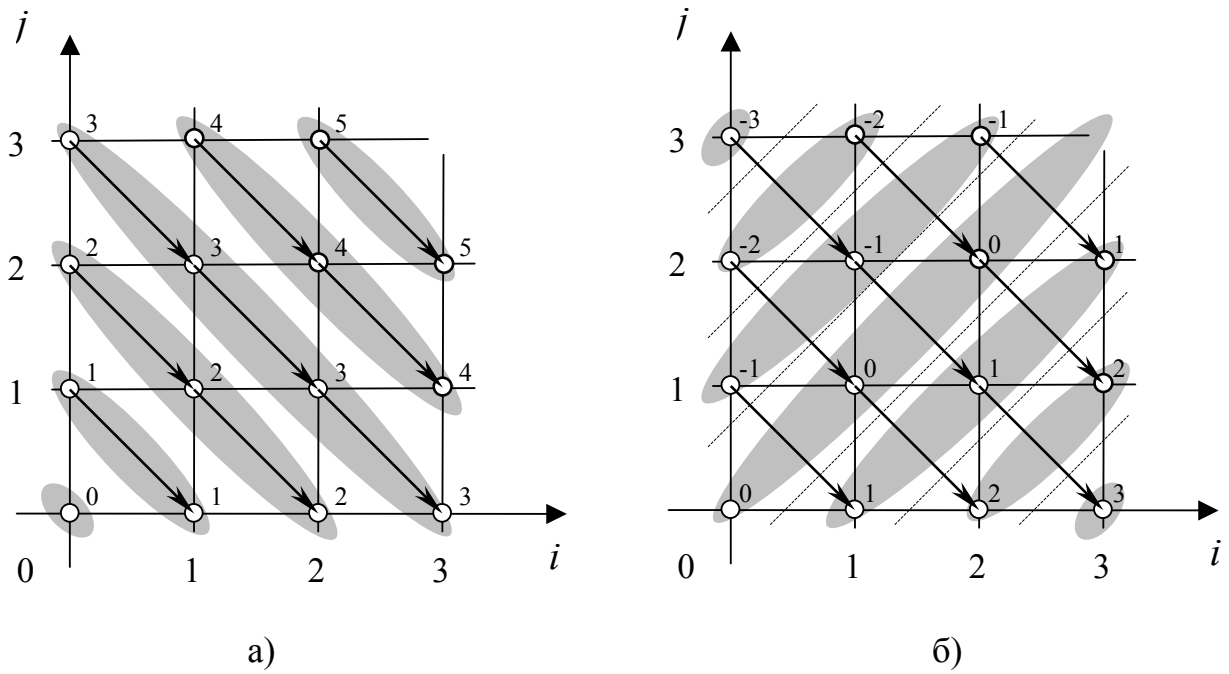


Рис. 8. Различные развертки решетчатого графа и определяемые ими множества итераций: а) расщепляющая развертка, б) строгая развертка.

Конец примера 26.

На основе разверток разработан и разрабатывается ряд оптимизирующих/распараллеливающих преобразований программ [79, 74]. Тогда как унимодулярные преобразования [96] программ, обобщающие перестановку [79], скашиание (skewing) [79] и др. преобразования циклов, применимы только к тесно вложенным циклам[13], преобразования, основанные на теории разверток, применимы к произвольно вложенным циклам. Эти преобразования обобщают унимодулярные преобразования, а также распределение циклов (loop distribution [76]), слияние циклов (fusion [76]) и многие другие [79].

Развертки могут быть построены как на основе аналитического описания решетчатого графа в виде функций [21, 72, 73], так и без этого описания [72, 73, 79]. Не смотря на упомянутые выше достоинства разверток, решетчатые графы имеют свои преимущества перед развертками:

- функции, описывающие дуги решетчатых графов, часто используются для построения разверток [4, 21, 72, 73];
- существуют оптимизирующие преобразования, такие как подстановка индексных переменных и экспансия массивов (см. 2.3.1, 2.3.2), которые удобно выполнять с помощью теории решетчатых графов (автору не известны алгоритмы выполнения этих преобразований, основанные на развертках);
- при ручной оптимизации программы более удобно исследовать зависимости по решетчатому графу, чем по разверткам, т.к., например,

[13] Гнездо тесно вложенных циклов (или тесное гнездо циклов) – гнездо циклов, в котором все операторы, не являющиеся операторами заголовков циклов, содержатся в теле самого вложенного цикла.

минимальный снизу решетчатый граф потоковой зависимости описывает алгоритм программы.

Теория разверток и методы их использования развиваются в работах [4, 5, 34, 63, 79, 80, 81, 74]. Детальное рассмотрение этих методов выходит за рамки данной работы.

1.3 Основные способы нахождения информационных зависимостей

Разработано множество методов получения данных об информационных зависимостях в программе. В этом разделе будут рассмотрены наиболее общие и часто используемые методы: НОД тест и неравенства Банержи, Омега тест, методы П. Фотрье (*P. Feautrier*) и В.В. Воеводина для построения решетчатых графов.

1.3.1 Неравенства Банержи и НОД тест

Неравенства Банержи всегда используются совместно с НОД тестом. С помощью этих методов можно для пары вхождений определить: наличие зависимости, носители (уровни) зависимости, вектор направления зависимости, циклически независимую зависимость. В данной работе будет описана только одна разновидность неравенств Банержи [8], об остальных можно узнать, например, из [57, 95].

Введем определение положительной и отрицательной части числа. Пусть r-вещественное число. Обозначим через r^+ положительную часть числа r, а через r^- - отрицательную часть числа r и приведем формулы для их вычисления:

$$r^+ = \begin{cases} 0, r < 0 \\ r, r \geq 0 \end{cases}$$

$$r^- = \begin{cases} -r, r \leq 0 \\ 0, r > 0 \end{cases}$$

Заметим, что в работах [8] и [95] даются различные определения отрицательной части числа.

Рассмотрим два вхождения, зависимость между которыми требуется установить (или указать на ее отсутствие). Будем нумеровать операторы циклов, и обозначать их счетчики так же, как в части 1.1.2 данной работы.

…

$X(P(I_1, I_2, \ldots, I_{n1}))=\ldots$

$\boldsymbol{u}$

…

$\ldots \; = X(Q(I_1, I_2, \ldots, I_{n2}))$

$\boldsymbol{v}$

Пусть нижняя граница каждого цикла, охватывающего либо вхождение u, либо вхождение v, равна 1. Пусть верхняя граница i-го цикла, содержащего вхождение u, равна M_i (некоторая константа). Пусть верхняя граница i-го цикла, содержащего вхождение v, равна N_i (некоторая константа), при этом $M_i=N_i$ для $i\in[1, d]$, где d – количество общих циклов для вхождений u и v. Пусть

$P(I_1, I_2, \ldots, I_{n1})=a_0+\sum_{i=1}^{n1} a_i I_i, \quad Q(I_1, I_2, \ldots, I_{n2})=b_0+\sum_{i=1}^{n2} b_i I_i$.

Тогда имеет место

Теорема ([8, с. 101]). Если вхождение v зависит от вхождения u с носителем k ($1\le k\le d$), то выполняются

(а) НОД-тест:

НОД($a_1-b_1, a_2-b_2, \ldots, a_{k-1}-b_{k-1}, a_k, \ldots, a_{n1}, b_k, \ldots, b_{n2}$) делит без остатка ($a_0-b_0$),

(б) неравенство Банержи:

$$-b_k-\sum_{i=1}^{k-1}(a_i-b_i)^-(N_i-1) - (a_k^-+b_k)^+ (N_k-2) -\sum_{i=k+1}^{n1} a_i^-(M_i-1) - \sum_{i=k+1}^{n2} b_i^+ (N_i-1) \le$$

$$\le \sum_{i=0}^{n2} b_i - \sum_{i=0}^{n1} a_i \le$$

$$\le -b_k+\sum_{i=1}^{k-1}(a_i-b_i)^+(N_i-1) + (a_k^+-b_k)^+ (N_k-2) +\sum_{i=k+1}^{n1} a_i^+(M_i-1) +\sum_{i=k+1}^{n2} b_i^- (N_i-1).$$

Доказательство теоремы приводится в [8].

Теорема ([8, с. 102]). Если вхождение *v* зависит от вхождения *u* причем, эта зависимость является циклически независимой, то выполняются

(а) НОД-тест:

НОД(a_1–b_1, a_2–b_2, …, a_d–b_d, a_{d+1}, …, a_{n1}, b_{d+1}, …, b_{n2}) делит без остатка (a_0–b_0),

(б) неравенство Банержи:

$$\sum_{i=1}^{d}(a_i-b_i)^-(N_i-1)-\sum_{i=d+1}^{n1}a_i^-(M_i-1)-\sum_{i=d+1}^{n2}b_i^+(N_i-1)\le$$

$$\le\sum_{i=0}^{n2}b_i-\sum_{i=0}^{n1}a_i\le$$

$$\le\sum_{i=1}^{d}(a_i-b_i)^+(N_i-1)+\sum_{i=d+1}^{n1}a_i^+(M_i-1)+\sum_{i=d+1}^{n2}b_i^-(N_i-1).$$

Доказательство теоремы приводится в [8].

Заметим, что неравенства Банержи выводятся для вхождений одномерных массивов. Для применения данных неравенств к вхождениям многомерных массивов, необходимы дополнительные аппроксимации [95, с. 174], [59] ухудшающие точность метода.

Для нахождения векторов направления зависимости данным, используются неравенства Банержи другого вида [95].

1.3.2 Методы построения решетчатых графов В. В. Воеводина и П. Фотрье (P. Feautrier)

Все минимальные решетчатые графы строятся на основе элементарных решетчатых графов. Поэтому здесь будет детально рассмотрен процесс построения элементарного решетчатого графа, тем более что знание этого процесса существенно потребуется в настоящей работе. Для упрощения изложения, далее в этом параграфе будем считать, что рассматриваемые фрагменты программ не содержат внешних переменных.

Пусть вхождение u содержится в гнезде из n циклов, пространство итераций которого V_1. Вхождение v – в гнезде из m циклов, пространство итераций которого V_2. Пусть оператор, содержащий вхождение u находится раньше по тексту программы, чем оператор, содержащий вхождение v. Обозначим через $P(I)$ вектор, координатами которого являются индексы вхождения u, через $Q(J)$ – вектор индексов вхождения v.

Рассмотрим задачу построения элементарного решетчатого графа [21, с. 364], [70], описывающего зависимость вхождения v от вхождения u. Зафиксируем некоторую вершину $\boldsymbol{J} \in V_2$. Чтобы из вершины $\boldsymbol{I} \in V_1$ выходила дуга в вершину $\boldsymbol{J} \in V_2$, необходимо выполнение условий:

$$P(\boldsymbol{I}) = Q(\boldsymbol{J}). \tag{2}$$

$$\boldsymbol{I} \in V_1,\ \boldsymbol{J} \in V_2, \tag{3}$$

$$\boldsymbol{I} \leq_{lex} \boldsymbol{J} \tag{4}$$

Система (2) при условиях (3), (4) может иметь не единственное решение. На множестве всевозможных решений $\boldsymbol{I}$ из (2), (3), (4) нужно найти то решение $\boldsymbol{I}_0$, которое является лексикографически максимальным (*lex.max*). Такое решение

$$\boldsymbol{I}_0 = lex.max\ \boldsymbol{I} \tag{5}$$

является единственным.

Задача (2) – (5) является целочисленной, т.к. целочисленными являются векторы $\boldsymbol{I}$ и $\boldsymbol{J}$. Решение $\boldsymbol{I}_0 = \boldsymbol{I}_0(\boldsymbol{J})$ задачи (2) – (5) зависит от вектора счетчиков циклов $\boldsymbol{J}$.

Для того чтобы решить указанную задачу, сначала нужно свести условия (2), (3), (4) к совокупности систем линейных неравенств. Условия $P(\boldsymbol{I}) = Q(\boldsymbol{J})$ и $\boldsymbol{I} \in V_1$, $\boldsymbol{J} \in V_2$ сводятся к системе линейных неравенств как обычно в линейном

программировании. Рассмотрим подробно, как сводится к совокупности систем линейных неравенств условие $\boldsymbol{I} \leq_{lex} \boldsymbol{J}$.

Пусть два указанных вхождения u и v имеют d общих циклов. Системы неравенств, к которым сводится условие $\boldsymbol{I} \leq_{lex} \boldsymbol{J}$, называются *альтернативными многогранниками* [21, с. 365]. Первая из этих систем имеет следующий вид:

$$\begin{aligned} I_1 &= J_1; \\ I_2 &= J_2; \\ &\ldots \\ I_d &= J_d. \end{aligned} \tag{6}$$

и рассматривается только в случае, если оператор, содержащий вхождение u находится раньше по тексту программы, чем оператор, содержащий вхождение v. Все остальные альтернативные многогранники рассматриваются во всех случаях. Второй многогранник имеет вид:

$$\begin{aligned} I_1 &= J_1; \\ I_2 &= J_2; \\ &\ldots \\ I_d &< J_d\,. \end{aligned} \tag{7}$$

В каждом следующем многограннике количество равенств уменьшается на единицу, неравенство всегда одно. Последний многогранник будет таким:

$$I_1 < J_1 \tag{8}$$

Занумеруем все альтернативные многогранники. Пусть количество равенств в системе, описывающей некоторый альтернативный многогранник, равно m.

Тогда номер этого многогранника положим равным m+1. Заметим, что номера альтернативных многогранников изменяются от 1 до d+1. По построению, любая точка $\boldsymbol{I}$ любого альтернативного многогранника удовлетворяет условию $\boldsymbol{I} \leq_{lex} \boldsymbol{J}$. Лексикографически ближе к $\boldsymbol{J}$ будет та точка, которая находится в альтернативном многограннике с большим номером [21, с. 366].

Подчеркнем, что если в решетчатом графе существует дуга из вершины $\boldsymbol{I}$ в вершину $\boldsymbol{J}$, то компоненты указанных векторов удовлетворяют соотношениям (2), (3) и одной из систем линейных неравенств, определяющей некоторый альтернативный многогранник.

Задача (2) – (5) решается по следующей схеме. Организуется цикл, со счетчиком по k, в котором выполняются нижеследующие действия, причем, если оператор, содержащий вхождение u находится по тексту программы раньше, чем оператор, содержащий вхождение v, то k изменяется от d+1 до 1, иначе k изменяется от d до 1:

1. Система (2) и условия (3) сводятся к системе линейных неравенств, к которой дописывается набор линейных неравенств, определяющих альтернативный многогранник с номером k. В полученной системе есть набор неизвестных – компоненты вектора $\boldsymbol{I}$, и набор параметров – компоненты вектора $\boldsymbol{J}$.
2. Далее во множестве решений указанной системы находится лексикографический максимум относительно неизвестных либо с помощью теории матриц с L-свойством [21, 25], либо с помощью метода параметрического целочисленного программирования[14] [70].

---

[14] в основе метода параметрического целочисленного программирования [70] лежит двойственный симплекс метод [39]. В работе [70], П. Фотрье параметризовал этот метод вместе с методом отсечений Гомори [39], который обеспечивает целочисленность получаемых решений.

3. Если лексикографический максимум найден, то выписываются функции, определяющие дуги решетчатого графа, и области их определения [21, 70]. Вычтем из V_2 объединение областей определения только что построенных функций. Если полученное множество V_2 пусто, то алгоритм заканчивается. Иначе уменьшить k на единицу и перейти на пункт 1.

Таким образом, из алгоритма построения решетчатого графа видно, что каждой функции F_k можно поставить в соответствие единственный номер s альтернативного многогранника, который использовался при ее построении. Системе неравенств, которая определяет альтернативный многогранник с номером s, будут удовлетворять компоненты каждой пары векторов ($\boldsymbol{I}$, $\boldsymbol{J}$), для которой выполняется $\boldsymbol{I}=F_k(\boldsymbol{J})$. Этот факт используется при определении ParDo циклов в программе (см. 2.1).

1.3.3 Омега Тест

Омега тест – точный метод нахождения информационных зависимостей в программе. Изначально этот метод, решая систему (2) – (4), позволял точно указывать, существует ли информационная зависимость между двумя вхождениями [89]. Для решения задачи (2) – (4), авторы Омега теста расширили метод исключений Фурье-Моцкина [39] до линейного *целочисленного* программирования [89]. Затем авторы показали, как можно использовать Омега тест для нахождения различных представлений информационной зависимости по значению [91]. После этого, Омега тест был еще усовершенствован до возможности построения решетчатых графов [92].

1.3.4 Сравнение методов нахождения информационных зависимостей

Одной из важнейших характеристик метода является сложность. Сразу отметим, что все задачи нахождения информационных зависимостей любых пред-

ставлений эквиваленты задаче определения существования *целочисленного* решения системы линейных неравенств [21, 70, 89]. Последняя задача – задача линейного целочисленного программирования – является NP-полной [39]. Поэтому все *точные* методы нахождения информационных зависимостей являются экспоненциальными (относительно максимальной из глубин вложенности гнезд циклов, содержащих зависимые вхождения).

1.3.4.1 Неравенства Банержи и точные методы

В этом разделе будет дано сравнение приближенных методов, основанных на неравенствах Банержи и НОД тесте, с точными методами П. Фотрье и Омега тестом. Будем предполагать, что эти методы используются на этапе компиляции распараллеливаемой программы. Сравнение этих методов, в зависимости от способа их использования – на этапе компиляции или на этапе выполнения программы – приводится в [88].

Методы нахождения информационных зависимостей, основанные на неравенствах Банержи и НОД тесте, используются, например, в компиляторах [3, 87, 84, 103] (наряду с другими методами анализа зависимостей). Указанные методы имеют следующие достоинства:

1. *линейная* сложность (в отличие от точных методов);
2. простота программной реализации;
3. хорошее поведение на практике: нетривиальной является задача поиска примера, в котором неравенства Банержи и НОД тест неправильно определяют вектор направления зависимости и, особенно, неправильно определяют наличие зависимости.

Теперь рассмотрим минусы данных методов, по сравнению с точными методами. Сразу можно сказать, что в отличие от указанных точных методов, неравенства Банержи и НОД тест не применимы ко всем программам из линейного

класса. С помощью этих методов можно обработать случаи с внешними переменными в индексных выражениях или границах циклов только в узком классе программ [95]. Кроме этого, неравенства Банержи и НОД тест никак не учитывают условные операторы.

Можно привести примеры программ, где анализ с помощью этих методов проигрывает по сравнению с точными методами. Эти примеры представлены в оставшейся части данного пункта.

Устранение ложных зависимостей.

С помощью неравенств Банержи и НОД теста нельзя находить представления информационной зависимости по значению. В частности, с помощью этого метода невозможно устранять ложные зависимости в программе (например, во фрагменте программы примера 10).

Точность анализа информационных зависимостей.

Методы, основанные на неравенстве Банержи и НОД тесте, не являются точными: теоремы, описанные в пункте 1.3.1, дают только необходимые условия. Таким образом, если выполняются неравенства Банержи и НОД тест, то нужно *предполагать* существование информационной зависимости или определенного вектора направления зависимости (носителя зависимости), что может быть неверно.

Случай неконстантных границ операторов циклов.

Неравенства Банержи [8, 95] разработаны для циклов с известными константными границами. Если граница цикла есть функция от счетчиков внешних циклов, то нужно использовать тесты Банержи специального вида [57, с. 104-105]. В противном случае, при использовании тестов Банержи [8, 95] нужно аппроксимировать все границы константами. После такой аппроксимации фактически будет проверяться наличие информационной зависимости в цикле, пространство итераций которого больше, чем у исходного цикла.

Пример 27. Неравенства Банержи и НОД тест показывают наличие зависимости в следующем треугольном цикле L, хотя этой зависимости не существует.

```
for(i=1; i<=5; i++)

for(j=i; j<=5; j++)

    a[i][j] = a[j][i] + b[i][j];
```

Рассмотрим рисунок 9 (стр. 81). На этом рисунке изображен решетчатый граф зависимости вхождения $a[j][i]$ от $a[i][j]$, ***но*** в гнезде циклов примера 31 (стр. 80). Рассматриваемое гнездо циклов L отличается от гнезда циклов примера 31 только пространством итераций. Пространство итераций гнезда L – множество N_1 – является подмножеством пространства итераций гнезда циклов в примере 31.

Заметим, что указанный решетчатый граф, кроме информационной зависимости по значению, описывает также и просто информационную зависимость (в данном примере нет перезаписей одних и тех же ячеек). Все дуги этого графа идут только из множества N_1 во множество N_2. Т.к итерационное пространство гнезда циклов L есть множество N_1, то решетчатый граф зависимости $a[j][i]$ от $a[i][j]$ для гнезда L не имеет дуг. Следовательно, в гнезде L нет потоковой зависимости вхождения $a[j][i]$ от $a[i][j]$.

Конец примера 27.

Далее в этом пункте все границы операторов циклов будут известными константами, чтобы устранить из рассмотрения неточность неравенств Банержи [8, 95], описанную выше.

Случай анализа зависимости между вхождениями многомерных массивов.

Неравенства Банержи и НОД тест разработаны для анализа зависимости между вхождениями одномерных массивов. Если наличие зависимости требуется

проверить между вхождениями массивов размерности больше 1, то нужно либо производить линеаризацию массивов[15] [95, с. 174], [59] либо, последовательно перебирая все размерности массива, применять неравенства Банержи и НОД тест отдельно к каждой паре индексных выражений соответствующих размерностей [95, с. 174]. Последние аппроксимации могут привести к тому, что в методе будет проверяться существование решения у системы с меньшим рангом, чем у исходной. У системы с меньшим рангом решений больше, чем у системы с большим рангом. Таким образом, может возникнуть следующая ситуация: у исходной системы не было решений, а у аппроксимированной системы с меньшим рангом решения появились. Последнее приведет к тому, что метод ошибочно определит наличие информационной зависимости.

Пример 28. НОД тест и неравенства Банержи ошибочно определяют наличие потоковой зависимости в следующем фрагменте программы.

```
int a[100][100], i, b[100], c[100];

for(i=1; i<99; i++)

    a[i+1][i] = b[i] * a[i][100-i] + c[i];
```

Неравенства Банержи и НОД тест показывают, что в указанном фрагменте программы имеется потоковая зависимость вхождения $a[i][100-i]$ от $a[i+1][i]$. Любой точный тест покажет, что потоковой зависимости $a[i][100-i]$ от $a[i+1][i]$ – нет.

Рассмотрим причину, вследствие которой неравенства Банержи и НОД тест показывают наличие потоковой зависимости. Как указывалось ранее, при про-

---

[15] линеаризация массива – аппроксимация нескольких индексных выражений массива одним выражением.

верке зависимости между вхождениями двумерного массива *a* данным методом, необходимо воспользоваться одной из аппроксимаций:

1. индексные выражения двумерного вхождения массива *a* заменить одним индексным выражением (линеаризация массива[16] [95]);

2. в процессе анализа зависимости, проверять отдельно каждую пару индексных выражений, соответствующую одной и той же размерности вхождений массивов.

Рассмотрим сначала случай аппроксимации индексных выражений двумерного массива одним выражением. После описанной аппроксимации, тесты будут проверять, существует ли решение уравнения: $(100+1)*i_1+100=(100-1)*i_2+100$. В этом случае, неравенства Банержи покажут, что зависимость есть т.к. существует вещественное решение указанного уравнения в пространстве итераций. НОД тест покажет, что целое решение этого уравнения есть (т.к. НОД(100+1, 100-1) поделит нацело любое число). Таким образом, будет сделан вывод, что имеется целое решение указанного уравнения в пространстве итераций, что не так.

Если рассматривать отдельно каждую пару индексных выражений, то сначала будет проверяться наличие решения в пространстве итераций уравнения $i_1+1=i_2$. Затем будет проверяться существование решения уравнения $i_1=100-i_2$. У каждого из этих уравнений существуют решения в пространстве итераций (это же укажут НОД тест и неравенства Банержи). Из этого будет сделан вывод о наличии зависимости, что неверно.

Конец примера 28.

---

[16] в некоторых случаях для определения зависимости используют обратный процесс – делинеаризацию массива [82].

Пример 29. НОД тест и неравенства Банержи ошибочно определяют наличие потоковой зависимости между вхождениями, имеющими тип индексных выражений отличный от рассмотренного в примере 28.

```
int a[100][100];

for(i=1; i<50 ; i++)

for(j=1; j<50 ; j++)

{ a[i+j][i+j] = …;

   …       = a[i+j][j]…

}
```

Здесь нет потоковой зависимости $a[i+j][j]$ от $a[i+j][i+j]$, но неравенства Банержи и НОД тест определят ее.
Конец примера 29.

Пример 30. Демонстрируется случай, когда при анализе эквивалентности преобразования «перестановка циклов» [8, 26, 57] НОД тест и неравенства Банержи запрещают либо разрешают выполнять это преобразование, в зависимости от способа аппроксимации индексных выражений двумерных массивов. Если для тех же целей используется точный алгоритм, то будет показано, что преобразование эквивалентно и может быть выполнено.

```
int a[10][10], i, b[10][10];

for(i=1; i<10; i++)

for(j=1; j<10; j++)

    a[i][j] = a[i-1][j] + b[i][j];
```

Заголовки циклов можно переставлять только в том случае, если не существует пар итераций (i_1, j_1) и (i_2, j_2), для которых выполняется:

1. $i_1 < i_2$ и $j_1 > j_2$ (9)
2. $a[i_1][j_1]$ и $a[i_2 -1][j_2]$ обращаются к одной ячейке памяти, т.е. выполняются равенства: $i_1=(i_2 -1)$ и $j_1=j_2$.

Обоснование можно посмотреть в [8].

Очевидно, что таких пар итераций нет и в указанном гнезде можно переставлять заголовки циклов. Анализ с помощью любого точного метода устанавливает этот факт.

Для того чтобы можно было провести анализ возможности перестановки с помощью неравенства Банержи и НОД теста нужно применить один из вариантов аппроксимации двух индексных выражений вхождений массивов одним выражением. Если для указанных целей использовать линеаризацию массива, то Банержи и НОД запретят переставлять циклы. Дело в том, что после линеаризации, тест Банержи будет проверять наличие вещественного решения уравнения: $10* i_1 + j_1 = 10* i_2 – 10 + j_2$, для неизвестных которого выполняется условие (9). Такие решения существуют, например:

$i_1 =1.1 < 2.5 = i_2$

$j_1 = 8 > 4 = j_2$.

Вещественные решения существуют и неравенства Банержи запрещают перестановку циклов. Но запрещает перестановку наличие только целых решений.

Если вместо линеаризации массивов проверять неравенства Банержи отдельно для каждой пары индексных выражений, соответствующей одной и той же размерности вхождений массивов, то будет построен вектор направления зависимости (<, =). Последний вектор не запрещает перестановку циклов. В данном случае оказалось, что выгоднее проверять неравенства Банержи отдельно для каждой пары индексных выражений вместо линеаризации массивов.

М. Вольф и У. Банержи высказывают предположение, что при анализе зависимости между вхождениями многомерных массивов лучше всего проверять неравенства Банержи и НОД тест отдельно для каждой пары индексных выражений, соответствующих одной и той же размерности. А если имеется возможность, то вместе с этим применять НОД тест для линеаризованных индексных выражений рассматриваемых вхождений [95, с. 177]. Данный пример подкрепляет предположение М. Вольф и У. Банержи.

Конец примера 30.

Пример 31. Демонстрируется случай, когда анализ на основе НОД теста и неравенств Банержи ошибочно запрещает одновременное исполнение итераций внутреннего цикла.

Рассмотрим фрагмент программы:

```
for(i=1; i<=5; i++)

for(j=1; j<=5; j++)

    a[i][j]=a[j][i]+b[i][j];
```

Если анализировать указанный фрагмент с помощью носителей зависимости, построенных на основе неравенств Банержи с использованием линеаризации, то зависимость $a[j][i]$ от $a[i][j]$ будет иметь носители 1 и 2. Тот же результат будет получен, если применить неравенства Банержи отдельно для каждой

пары индексных выражений, соответствующих одной размерности. Из указанных результатов двух различных вариантов использования неравенств Банержи следует один вывод: ни внешний, ни внутренний цикл выполнять параллельно нельзя.

Теперь проанализируем этот фрагмент с помощью решетчатого графа, описывающего потоковую зависимость $a[j][i]$ от $a[i][j]$ (решетчатый граф оставшейся зависимости в этом цикле – антизависимости $a[i][j]$ от $a[j][i]$ – имеет точно такой же вид). Этот граф изображен на рисунке 9.

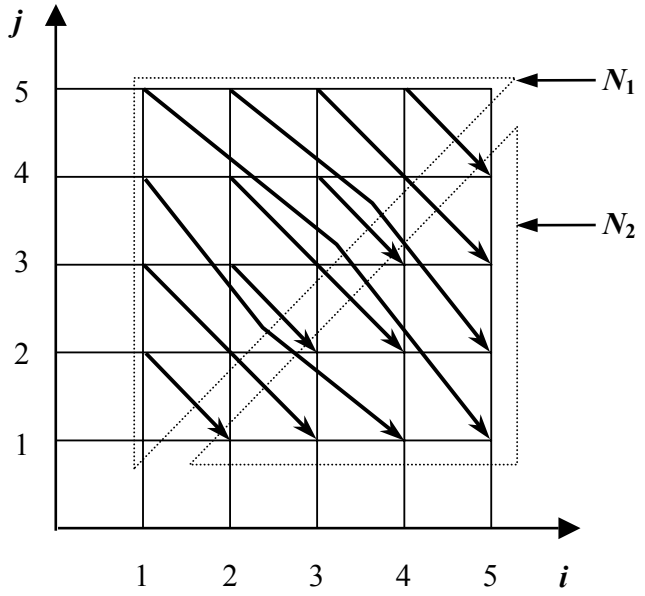


Рис. 9. Решетчатый граф, описывающий зависимость $a(j, i)$ от $a(i, j)$.

Из рисунка 9 видно, что итерации внешнего цикла выполнять одновременно нельзя т.к. существуют такие итерации цикла по «i», на которых используются значения, вычисленные на предыдущих итерациях этого же цикла. Но, на каждой конкретной итерации цикла по «i», итерации цикла по «j» могут быть выполнены в любом порядке, в том числе и одновременно. Таким образом, анализ на основе решетчатых графов показывает, что итерации внутреннего цикла выполнять одновременно можно.

Конец примера 31.

1.3.4.2 Метод В.В. Воеводина, метод П. Фотрье и Омега тест

Сравним сначала метод В.В. Воеводина и метод П. Фотрье.

1. Метод П. Фотрье позволяет точно строить решетчатые графы для всех программ из линейного класса. Метод В. В. Воеводина [21, 23, 24, 25] не позволяет точно строить решетчатые графы во всем линейном классе программ: в частности, с помощью данного метода нельзя точно построить функцию, описывающую дуги решетчатого графа фрагмента программы примера 25. С помощью этого метода можно построить систему покрывающих функций для графа, которая в общем случае, может описывать дуг больше, чем имеется в этом графе. Отметим, что метод [21, 23, 24, 25] строит функции, точно описывающие дуги решетчатого графа, если дуги этого решетчатого графа могут быть точно описаны функциями, удовлетворяющими условиям С.1) – С.3) (см. пункт 1.2.2.4).

Алгоритм В. В. Воеводина не является точным во всем линейном классе программ потому, что при нахождении лексикографического максимума в области, заданной системой линейных неравенств, в этом алгоритме отсутствует часть, которая бы отвечала за целочисленность найденного решения [21, с. 368].

2. Теоретически, оба метода являются экспоненциальными. Однако сравнивать их по сложности не совсем корректно, т.к. эти алгоритмы решают разные задачи. Отметим также, что если в методе П. Фотрье не использовать параметризированный метод отсечений, с помощью которого обеспечивается целочисленность решений, то этим методом можно получать тот же результат, что и методом В. В. Воеводина [21, 23, 24, 25].

3. Метод В. В. Воеводина выдает результат в виде набора функций и их областей определения. Метод П. Фотрье выдает результат, в общем случае, в виде

квазиаффинного дерева решений [70, 71]. Имея квазиаффинное дерево решений можно легко перейти к набору функций и их областям определения.

Сравним теперь метод П. Фотрье и Омега тест.

1. Оба метода являются точными для всех программ из линейного класса.

2. Оба метода являются экспоненциальными (относительно максимальной из глубин вложенности гнезд циклов, содержащих зависимые вхождения).

3. Если Омега тест использовать для построения решетчатого графа, то форма результата [92] будет принципиально отличаться от той, которая выдается методом П. Фотрье (или В. В. Воеводина). Результат, выдаваемый Омега тестом, не содержит явного описания функций, которые описывают дуги решетчатого графа [92]. Поэтому, по мнению автора данной работы, использование Омега теста для построения решетчатых графов не является удобным и не получило широкого распространения.

2 АНАЛИЗ И ПРЕОБРАЗОВАНИЯ ПРОГРАММ, ОСНОВАННЫЕ НА РЕШЕТЧАТОМ ГРАФЕ

2.1 Автоматическое распознавание ParDo циклов в программе

В данном параграфе предлагается новый метод распознавания циклов ParDo [21] в программе, основанный на теории и методах построения решетчатых графов. Циклы, обладающие свойством ParDo, часто приходится определять при распараллеливании программ. Если цикл обладает таким свойством, то все его итерации можно выполнять в любом порядке, в том числе и одновременно [21].

Способы распознавания ParDo циклов рассматривались в [8, 21, 95]. Предлагаемый в данной главе метод позволяет во множестве программ из линейного класса определить большее количество ParDo циклов, чем указанные методы [8, 21, 95]. Он основан на необходимых и достаточных условиях, которые доказываются в настоящей работе на основе теории решетчатых графов. Причем, этот метод удобен в использовании при автоматическом распараллеливании. Методы [8, 95] используют более грубые абстракции информациионной зависимости, чем решетчатый граф. Кроме этого, используемые в [8, 95] неравенства Банержи предоставляют только достаточные условия. В работе [21, параграф 6.7] предлагается метод, так же основанный на теории решетчатых графов. Однако не доказывается, что функции, описывающие дуги решетчатого графа, строятся алгоритмом [21, параграф 6.5-6.6] в том виде, в котором утверждение 6.11 этой книги предоставляет и необходимые условия (помимо достаточных).

Более детально отличия предлагаемого метода от методов [8, 21, 95] будут описаны в 2.1.6.

В настоящее время предлагаемый метод определения ParDo циклов используется в Открытой распараллеливающей системе.

2.1.1 Определение ParDo цикла по решетчатому графу

На основании критерия, предложенного в [21, с. 374, утверждение 6.10], можно сформулировать следующее

Определение. (Цикл ParDo по решетчатому графу G.) Пусть G – минимальный решетчатый граф фрагмента программы. Пусть цикл L находится на глубине вложенности k в некотором гнезде циклов данного фрагмента. Цикл L будем называть *циклом* ParDo *по графу* G, если в графе G не существует вершин $\boldsymbol{I}=(I_1, I_2, \ldots, I_{n1})\in V_i$ и $\boldsymbol{J}=(J_1, J_2, \ldots, J_{n2})\in V_j$, для которых одновременно выполняются условия:

1. $I_l = J_l$, для всех $l\in[1, k\text{-}1]$;
2. $I_k < J_k$;
3. решетчатый граф G содержит дугу, идущую из вершины $\boldsymbol{I}$ в вершину $\boldsymbol{J}$;
4. операторы $Stmt_i$ и $Stmt_j$, опорные пространства которых есть V_i и V_j соответственно, содержатся в цикле L.

Пример 32. Рассмотрим двумерное гнездо циклов примера 31. Минимальный решетчатый граф потоковой зависимости для данного фрагмента изображен на рисунке 9. Для данного фрагмента программы пространство V_i совпадает с V_j и совпадает с итерационным пространством двумерного гнезда циклов. Соответственно, оператор $Stmt_i$ совпадает с оператором $Stmt_j$ и содержится в обоих рассматриваемых циклах.

Рассмотрим внешний цикл. Его глубина вложенности равна 1. В графе G существует дуга из вершины $\boldsymbol{I}=(I_1, I_2)$ в вершину $\boldsymbol{J}=(J_1, J_2)$, для которой выполняется $I_1<J_1$ (рис. 9). Следовательно, цикл, находящийся на глубине вложенности 1 рассматриваемого гнезда (цикл со счетчиком «*i*»), не является циклом ParDo *по графу* G. Заметим, что данная потоковая зависимость действительно запрещает одновременное исполнение итераций цикла со счетчиком «*i*» (см. пример 31).

Рассмотрим внутренний цикл. Его глубина вложенности равна 2. В графе G НЕ существует дуги из вершины $\boldsymbol{I}=(I_1, I_2)$ в вершину $\boldsymbol{J}=(J_1, J_2)$, для которой выполняется $I_1=J_1$ и $I_2<J_2$ (рис. 9). Следовательно, цикл, находящийся на глубине вложенности 2 рассматриваемого гнезда (цикл со счетчиком «*j*»), является циклом ParDo *по графу G*. Заметим, что рассматриваемая потоковая зависимость не запрещает одновременное исполнение итераций цикла со счетчиком «*j*». Для принятия решения о том, можно ли исполнять итерации данного цикла одновременно, нужно рассмотреть и другие типы зависимостей в данном гнезде.
Конец примера 32.

Для упрощения изложения и повышения читабельности, далее в пунктах 2.1.2, 2.1.3, 2.1.4 будем считать, что рассматриваемые фрагменты программ не содержат внешних переменных. Случай, когда фрагмент программы содержит внешние переменные, будет рассмотрен отдельно в пункте 2.1.5.

2.1.2 Распознавание ParDo циклов по решетчатому графу. Связь между минимальными решетчатыми графами и носителями зависимости по значению

Утверждение 3. Пусть имеется минимальный решетчатый граф G фрагмента программы. Тогда для решетчатого графа G следующие условия **А** и **Б** эквивалентны:

А. В наборе функций, описывающих дуги графа G, существует функция F, которая удовлетворяет следующим условиям:

1. дуги графа G, определяемые функцией F, описывают зависимость некоторого вхождения $v \in Stmt_i$ от другого вхождения $u \in Stmt_j$; опорные пространства операторов $Stmt_i$ и $Stmt_j$ есть V_i и V_j соответственно;

2. при построении функции F использовался альтернативный многогранник с номером s.

Б. В решетчатом графе G существуют вершины $\boldsymbol{I}=(I_1, I_2, \ldots, I_{n1})\in V_i$ и $\boldsymbol{J}=(J_1, J_2, \ldots, J_{n2})\in V_j$, для которых выполняются условия:

1. $I_l = J_l$, для всех $l\in[1, s\text{-}1]$; (10)
2. $I_s < J_s$;
3. решетчатый граф G содержит дугу, идущую из вершины $\boldsymbol{I}$ в вершину $\boldsymbol{J}$;

Доказательство следует из алгоритма построения минимального решетчатого графа и способа нумерации альтернативных многогранников (пункт 1.3.2).

<u>Утверждение 4</u>. Пусть G – минимальный решетчатый граф фрагмента программы. Пусть цикл L находится на глубине вложенности k в некотором гнезде циклов данного фрагмента. Цикл L является циклом ParDo по графу G тогда и только тогда, когда в наборе функций, описывающих дуги графа G, не существует функции F, для которой одновременно выполняются следующие условия:

1. дуги графа G, определяемые функцией F, описывают зависимость между вхождениями, находящимися в теле цикла L;
2. при построении функции F использовался альтернативный многогранник с номером k.

Доказательство следует из утверждения 3 и определения цикла ParDo.

Пример 33. Рассмотрим двумерное гнездо циклов примера 31. Все дуги минимального решетчатого графа G потоковой зависимости описываются единственной функцией $F_1(i, j)=(j, i)$, область определения которой – множество N_2 определяется неравенствами: $i\leq 5$, $j\geq 1$, $i>j$. Нетрудно видеть, что для всех вершин $\boldsymbol{I}=(I_1, I_2)$ и $\boldsymbol{J}=(J_1, J_2)$ таких, что в графе G существует дуга из $\boldsymbol{I}$ в $\boldsymbol{J}$, выполняется $I_1<J_1$ (рис. 9). Следовательно, при построении функции F_1, использовался аль-

тернативный многогранник с номером 1. Из утверждения 4 следует, что цикл, находящийся на глубине вложенности 1, не является циклом ParDo по графу *G*.

Т.к. функция F_1 описывает все дуги графа G, то можно констатировать, что не существует функций, описывающих дуги графа G, при построении которых использовался альтернативный многогранник номер 2. Следовательно, по утверждению 4, цикл, находящийся на глубине вложенности 2, является циклом ParDo по графу *G*.

Конец примера 33.

Следствие 1. (Связь между минимальными решетчатыми графами и носителями зависимости по значению.) Пусть минимальный решетчатый граф *G* описывает зависимость *по значению* некоторого типа в программе. Пусть вхождения *u* и *v* имеют *d* общих циклов. Вхождение *u* зависит по значению от вхождения *v* с носителем *k* ($1 \le k \le d$) тогда и только тогда, когда в наборе функций, описывающих дуги графа *G*, существует функция *F*, для которой выполняется:

1. дуги графа *G*, определяемые функцией *F*, описывают зависимость вхождения *u* от вхождения *v*;
2. при построении функции *F* использовался альтернативный многогранник с номером *k*.

Зависимость по значению *u* от *v* является циклически независимой тогда и только тогда, когда в наборе функций, описывающих дуги графа *G*, существует функция *F*, для которой выполняется:

1. дуги графа *G*, определяемые функцией *F*, описывают зависимость вхождения *u* от вхождения *v*;
2. при построении функции *F* использовался альтернативный многогранник с номером $d+1$.

Доказательство следует из утверждения 3, определения носителя зависимости по значению и циклически независимой зависимости по значению.

2.1.3 Определение ParDo циклов и их связь с ParDo циклами по решетчатому графу

В предыдущих пунктах было предложено определение цикла ParDo по минимальному решетчатому графу, а также необходимые и достаточные условия, позволяющие определять такие циклы. Но, если цикл является циклом ParDo по одному какому-либо графу, то из этого еще не следует, что итерации данного цикла можно исполнять одновременно. Для того чтобы определить, можно ли итерации некоторого цикла исполнять одновременно, необходимо рассмотреть три минимальных решетчатых графа: потоковой, анти- и выходной зависимости. Кроме этого, встает дополнительный вопрос о том, какие из минимальных решетчатых графов нужно использовать: минимальные снизу или сверху. Авторы работы [21] не указывают явно нужные минимальные графы. Этот вопрос будет рассмотрен в настоящем пункте.

Для начала, рассмотрим

Пример 34.

Рассмотрим программный цикл примера 20 (стр. 49). В этом цикле имеется антизависимость вхождения v от вхождений $u1$, $u2$. Так как других зависимостей в рассматриваемом цикле нет, то только указанная антизависимость может повлиять на возможность одновременного исполнения итераций данного цикла.

Из рисунка 5.а видно, что рассматриваемый цикл является циклом ParDo по минимальному снизу решетчатому графу антизависимости $\underline{G}$. Из рисунка 5.б видно, что рассматриваемый цикл НЕ является циклом ParDo по минимальному сверху решетчатому графу антизависимости $\overline{G}$, т.к. существуют дуги из i_1 в i_2, для которых выполняется: $i_1<i_2$. Эти дуги появляются из-за антизависимости

вхождения v от вхождения $u1$, которая действительно запрещает одновременное исполнение итераций.

Конец примера 34.

Определение. Пусть G – решетчатый граф, полученный объединением минимальных решетчатых графов, описывающих потоковую, анти- и выходную зависимость *по значению*, а именно: минимального снизу решетчатого графа потоковой зависимости, минимального сверху графа антизависимости, минимального снизу или сверху графа выходной зависимости. Цикл, являющийся ParDo циклом по решетчатому графу G, будем называть *ParDo циклом*.

Утверждение 5. Все итерации ParDo цикла могут быть исполнены в любом порядке, в том числе и одновременно.

Доказательство. Итерации цикла могут исполняться независимо (одновременно), если не существует итерации внешних циклов, на которой в рассматриваемом цикле на разных итерациях:

- либо сначала записывается значение в память, которое затем используется;
- либо сначала используется значение, на место которого затем записывается новое;
- либо перезаписывается одно и тоже значение.

Пусть цикл L является ParDo циклом. Тогда в пространстве итераций гнезда циклов, содержащего цикл L, нет итераций $\boldsymbol{I}$, $\boldsymbol{J}$ для которых одновременно выполняются условия:

1. координаты векторов $\boldsymbol{I}$, $\boldsymbol{J}$, соответствующие внешним относительно L циклам, совпадают;
2. координаты векторов $\boldsymbol{I}$, $\boldsymbol{J}$, соответствующие циклу L, различны;
3. существуют вхождения u и v, которые на итерациях $\boldsymbol{I}$, $\boldsymbol{J}$ обращаются к одной и той же ячейке памяти, причем между этими обращениями общая

ячейка не перезаписывается (т.е. на этих итерациях вхождения u и v оперируют с одним и тем же значением);

4. как минимум одно из вхождений u, v является генератором.

Следовательно, все итерации цикла L могут быть исполнены в любом порядке, в том числе и одновременно. Последний вывод завершает доказательство.

Пример 35. Вернемся к рассмотрению двумерного гнезда циклов примера 31. Как было показано в примере 32, внутренний цикл является циклом ParDo по минимальному снизу решетчатому графу потоковой зависимости. Заметим, что в данном фрагменте программы минимальный снизу решетчатый граф потоковой зависимости совпадает с минимальным сверху решетчатым графом антизависимости. Минимальные решетчатые графы выходной зависимости вообще не содержат дуг. Из вышесказанного следует, что внутренний цикл является циклом ParDo по определению. Следовательно, по утверждению 5, итерации внутреннего цикла могут быть исполнены в любом порядке, в том числе и одновременно.

Конец примера 35.

Отметим, что свойство цикла ParDo по минимальному решетчатому графу, описывающему входную зависимость по значению, помогает определять циклы, удобные для эффективной реализации на многопроцессорных системах с распределенной памятью [21, с. 375].

2.1.4 Алгоритм распознавания ParDo циклов в программе

Резюмируя все вышесказанное, можно привести следующий алгоритм определения ParDo циклов.

Алгоритм определения ParDo циклов в произвольном гнезде:

1. Для рассматриваемого гнезда циклов строим минимальные решетчатые графы, описывающие потоковую, анти- и выходную зависимость по значению.
2. Обрабатываем каждую функцию F, описывающую дуги одного из построенных графов:
   - будем считать, что общие циклы для пары вхождений, которые определяют дуги описываемые функцией F, занумерованы от 1 в порядке вложенности, начиная с самого внешнего;
   - если при построении функции F использовался альтернативный многогранник с номером k, то пометим цикл с номером k как DoSeq; эта метка означает, что итерации цикла исполнять одновременно нельзя.
3. Все циклы, которые не были помечены как DoSeq, являются ParDo циклами.

2.1.5 Циклы ParDo и внешние переменные

Пусть минимальный решетчатый граф G фрагмента программы описывает, в частности, зависимость вхождения v от вхождения u. Пусть индексные выражения и описания опорных пространств этих вхождений содержат внешние переменные. Пусть эти внешние переменные упорядочены в векторе $\boldsymbol{N}$. Тогда множество вершин и множество дуг графа G зависят от вектора внешних переменных $\boldsymbol{N}$. В частности, существование множества дуг графа G, вследствие которого некоторый цикл не обладает свойством ParDo по графу G, зависит от вектора внешних переменных $\boldsymbol{N}$. Таким образом, свойство ParDo по решетчатому графу G некоторого цикла, содержащего вхождения u и v, зависит от значения вектора внешних переменных $\boldsymbol{N}$. Заметим, что если вершины и дуги решетчатого графа зависят от вектора внешних переменных $\boldsymbol{N}$, то область определения каждой

функции, описывающей дуги этого графа, также будет зависеть от $\boldsymbol{N}$ [21, 70, 71]. Резюмируя все вышесказанное, можно сформулировать

Утверждение 6. Пусть G – минимальный решетчатый граф фрагмента программы. Пусть цикл L находится на глубине вложенности k в некотором гнезде циклов данного фрагмента. Цикл L является циклом ParDo по графу G для значения вектора внешних переменных $\boldsymbol{N}'$ тогда и только тогда, когда в наборе функций, описывающих дуги графа G, не существует функции F, для которой одновременно выполняются следующие условия:

1. дуги графа G, определяемые функцией F, описывают зависимость между вхождениями, находящимися в теле цикла L;
2. при построении функции F использовался альтернативный многогранник с номером k;
3. область определения функции F не пуста при значении вектора внешних переменных $\boldsymbol{N}'$.

Доказательство следует из утверждения 4 и определения цикла ParDo по графу G.

Условия для значений вектора $\boldsymbol{N}$, при которых область определения функции F не пуста, можно получить, например, исключив из описания этой области переменные-счетчики циклов с помощью метода [89]. Указанные условия в общем случае описываются с помощью квазиаффинных форм. Однако для большинства программ встречающихся на практике, они описываются совокупностью систем линейных неравенств от $\boldsymbol{N}$ [58, 89, 92].

Пример 36. Рассмотрим фрагмент программы:

```
for(i=1;i<10;i++)

    x[i]= x[i+k]+y[i];
```

u ***v***

Дуги минимального снизу решетчатого графа G, описывающего антизависимость u от v, будут зависеть от значения параметра k. Они описываются двумя функциями: $F_1(i) = i$ с областью определения $1 \le i \le 9$, $k=0$, при построении которой использовался альтернативный многогранник с номером 2, и $F_2(i) = i-k$ с областью определения $1 \le i \le (9-k)$, $1 \le k \le 8$, при построении которой использовался альтернативный многогранник с номером 1. Из утверждения 6 следует, что рассматриваемый цикл не является циклом ParDo по графу G для $k=k'$ тогда и только тогда, когда область, описываемая системой неравенств $1 \le i \le (9-k')$ и $1 \le k' \le 8$, содержит целые точки. Если из описания области определения функции F_2 исключить переменную i с помощью метода [89], то будет получено условие на внешнюю переменную k, при котором рассматриваемый цикл не будет циклом ParDo по графу G. В данном случае, это условие имеет вид: $1 \le k \le 8$. При значении k не удовлетворяющем последнему условию (например, при $k=0$), рассматриваемый цикл является циклом ParDo по графу G.

Конец примера 36.

2.1.6 Сравнение с другими методами распознавания ParDo циклов

Распознавать ParDo циклы можно с помощью векторов направления зависимости [95] или, что эквивалентно, с помощью носителей зависимости [8]. Однако подобные способы имеют ряд недостатков:

1. один вектор направления зависимости может описывать не одну зависимость. Например, один вектор направления может описывать потоковую зависимость использования v от генератора u и, одновременно, описывать самозависимость генератора u. Поэтому, при использовании указанных представлений невозможно точно распознать тип зависимости, по которому цикл обладает или не обладает свойством ParDo.

2. вектор направления зависимости никак не учитывает ложные зависимости [91].

3. эффективные методы построения векторов направления зависимости, основанные на неравенствах Банержи [8, 95], дают лишь достаточные условия даже в случае отсутствия внешних переменных. Поэтому такие методы иногда могут не распознать ParDo цикл. Этот случай проиллюстрирован в примере 31.

Метод распознавания ParDo циклов, предложенный в данной работе, лишен перечисленных недостатков.

Другой известный метод распознавания ParDo циклов, основанный на теории решетчатых графов, описан в [21, параграф 6.7]. В этом методе на основе утверждения 6.11 [21, с. 375], производится анализ описаний функций, определяющих дуги решетчатого графа. Однако для покрывающих функций, которые точно описывают дуги решетчатого графа, удовлетворяют условиям С.1) – С.3) (пункт 1.2.2.4), и на которые не накладывается дополнительных условий, утверждение 6.11 [21, с. 375] предоставляет только достаточные условия.

Пример 37. (Показывает, что условия [21, с. 375, утв. 6.11] не являются необходимыми в случае, когда покрывающие функции точно описывают дуги решетчатого графа, удовлетворяют условиям С.1) – С.3) (пункт 1.2.2.4), а внешние переменные отсутствуют)

Условия [21, с. 375, утв. 6.11] не являются необходимыми т.к. функции, точно описывающие дуги решетчатого графа и удовлетворяющие условиям С.1) – С.3) (пункт 1.2.2.4), могут иметь различный вид. Приведем пример решетчатого графа, дуги которого могут быть заданы функциями различного вида, удовлетворяющими описанным выше условиям. Такой граф будем искать среди тех графов, у которых размерность множества вершин в которые входят дуги меньше размерности пространства, в котором заданы вершины. Рассмотрим гнездо циклов:

```
for(i=1; i≤5; i++)

    for(j=1; j≤5; j++)

      for(k=1; k≤5; k++)

         if(i==j)
             X(i, j, k) =  X(i, j, k-1)    // Stmt 1
```

Потоковая зависимость $X(i, j, k\text{-}1)$ от $X(i, j, k)$ описывается решетчатым графом G, который изображен на рисунке 10.

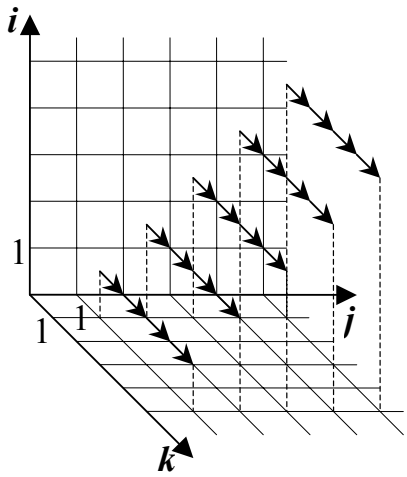


Рис. 10. Минимальный решетчатый граф G.

Дуги решетчатого графа G могут быть точно описаны функцией $F_1(i, j, k) = (0*i + j, j, k\text{-}1)$ с областью определения $1 \le i \le 5$, $1 \le j \le 5$, $2 \le k \le 5$, $i=j$. Дуги этого же графа G могут быть точно описаны функцией $F_2(i, j, k) = (i, j, k\text{-}1)$ с областью определения $1 \le i \le 5$, $1 \le j \le 5$, $2 \le k \le 5$, $i=j$.

Очевидно, самый внешний цикл (по i) является циклом ParDo по графу G. Функции F_1 и F_2 описывают дуги решетчатого графа G точно, кроме этого они удовлетворяют условию С.1) – С.3) (пункт 1.2.2.4). Если применить утвержде-

ние [2, с. 375, утв. 6.11] к функции F_1, то получим, что самый внешний цикл (по i) не является циклом ParDo по решетчатому графу G, что неверно.

Судя по виду индексных выражений вхождений переменной X в оператор *Stmt 1*, можно предположить маловероятным, что автоматически построенная функция, описывающая дуги решетчатого графа G, не будет иметь вид F_2. Однако в приведенном примере оператор *Stmt 1* может иметь другой вид (решетчатый граф G при этом останется неизменным):

1. $X(i, i, k) = X(i, j, k\text{-}1)$;
2. $X(i + j, k) = X(i + j, k\text{-}1)$.

В этом случае, алгоритм построения решетчатого графа [2, параграф 6.5-6.6] построит как минимум две функции F_1 и F_2, описывающие одни и те же дуги графа G.

Предложенный в диссертации метод успешно справляется с данной проблемой. При построении единственной функции, описывающей дуги решетчатого графа зависимости $X(i, j, k\text{-}1)$ от $X(i, j, k)$, использовался альтернативный многогранник с номером 3 (для всех вершин, соединенных дугами графа, выполняются условия 10 (утв. 3) с s=3). Следовательно, указанная зависимость лишает свойства ParDo только цикл с номером 3 (т.е. цикл по k).

Конец примера 37.

<u>*Замечание 5.*</u> Пример 37 показывает сложность выявления параллелизма и других свойств программ с помощью анализа описания функций, задающих дуги минимального решетчатого графа. В этом случае нужно гарантированно проанализировать все возможные описания имеющегося набора функций или накладывать дополнительные условия на описание функций при их построении.

Если предложенный в данной работе метод реализовать, используя алгоритм построения решетчатых графов [21, параграф 6.5-6.6], то получится более быстрый и менее сложный в реализации метод, чем [21, параграф 6.7]. Например,

пусть имеется гнездо из *n* тесно вложенных циклов, в котором имеется только одна информационная зависимость (пусть, для определенности, потоковая). Для выявления циклов, обладающих свойством ParDo методом [21, параграф 6.7] в общем случае, нужно выполнить следующие действия:

- построить *n* (!) решетчатых графов, описывающие единственную указанную зависимость (по одному графу на каждый оператор цикла; причем, при построении очередного решетчатого графа, соответствующего некоторому циклу, счетчики внешних циклов считаются внешними переменными);
- анализируя вид функций, описывающих дуги решетчатого графа, сделать вывод о наличии свойства ParDo у соответствующего оператора цикла. Анализ вида функции подразумевает проверку матриц на соответствие установленному виду.

Для выявления циклов, обладающих свойством ParDo предлагаемым в данной работе методом, нужно выполнить следующие действия:

- построить *один* решетчатый граф, описывающий указанную зависимость (при построении решетчатого графа, каждой функции, описывающей его дуги, поставить в соответствие номер альтернативного многогранника, который использовался при построении этой функции);
- анализируя номера, соответствующие построенным функциям решетчатого графа, сделать вывод о наличии свойства ParDo у всех операторов циклов рассматриваемого гнезда.

2.2 Расщепление многомерных гнезд циклов

В данном параграфе будут описаны различные методы расщепления многомерных гнезд циклов. Расщепление гнезда циклов является основой для выполнения таких преобразований как подстановка индексных переменных (см. 2.3.1) и экспансия массивов (см. 2.3.2). Кроме этого, данное преобразование само по себе может являться распараллеливающим (см. 2.2.2.2, 2.2.3.1).

Расщепление гнезда циклов описывалось в литературе для одномерных циклов [95], [21, с. 418]. В работе [47] рассматривается расщепление многомерных циклов, но в абстрактном виде: в этой работе не приводится описаний алгоритмов или методов выполнения расщепления.

Расщепление гнезда циклов заключается в замене данного гнезда циклов другим гнездом или гнездами, которые в совокупности сканируют[17] то же итерационное пространство, что и исходное гнездо циклов. При этом все операции исходного гнезда сохраняются в преобразованном гнезде.

Сразу отметим, что основная сложность при расщеплении заключается в построении нового гнезда циклов и генераций границ операторов циклов.

Некоторые виды расщепления реализованы в Открытой распараллеливающей системе.

2.2.1 Простые примеры рассматриваемых видов расщеплений

Расщепление одномерного цикла заключается в замене этого цикла несколькими подряд идущими циклами. Пространства итераций полученных циклов

---

[17] именно этот термин широко используется в зарубежной литературе, посвященной проблеме генерации гнезд циклов [58, 61, 62, 64, 93]. Говорят, что гнездо циклов сканирует полиэдр, который является линейным пространством итераций этого гнезда.

образуют разбиение пространства итераций исходного гнезда. Тела полученных циклов совпадают с телом исходного цикла.

Пример 38. (Расщепление одномерного цикла.)

Рассмотрим цикл:

```
for(i=1;i<N;i++)

    LoopBody(i);
```

Здесь и далее в работе LoopBody(i) обозначает некоторый набор операторов в теле цикла.

После расщепления данного цикла получим следующий фрагмент программы:

```
for(i=1;i<K;i++)

    LoopBody(i);

for(i=K;i<N;i++)

    LoopBody(i);
```

Здесь $1 \le K < N$.

Полученный фрагмент программы эквивалентен исходному гнезду т.к. в нем все операции и порядок их следования точно такие, как в исходном гнезде.

Конец примера 38.

Пример 39. (Расщепления двумерного цикла.)

Рассмотрим двумерный цикл:

```
for(i=1;i<=N;i++)
    for(j=1;j<=M;j++)
        LoopBody(i,j);
```

Расщепление гнезда циклов осуществляется по заданному разбиению его пространства итераций. Пространство итераций данного гнезда циклов – множество D – представляет собой множество всех целых точек, принадлежащих прямоугольнику с координатами вершин: (1, 1), (N, M) (рис. 11.а). Набор множеств $\{N_1, N_2\}$ является разбиением множества D (рис. 11.а).

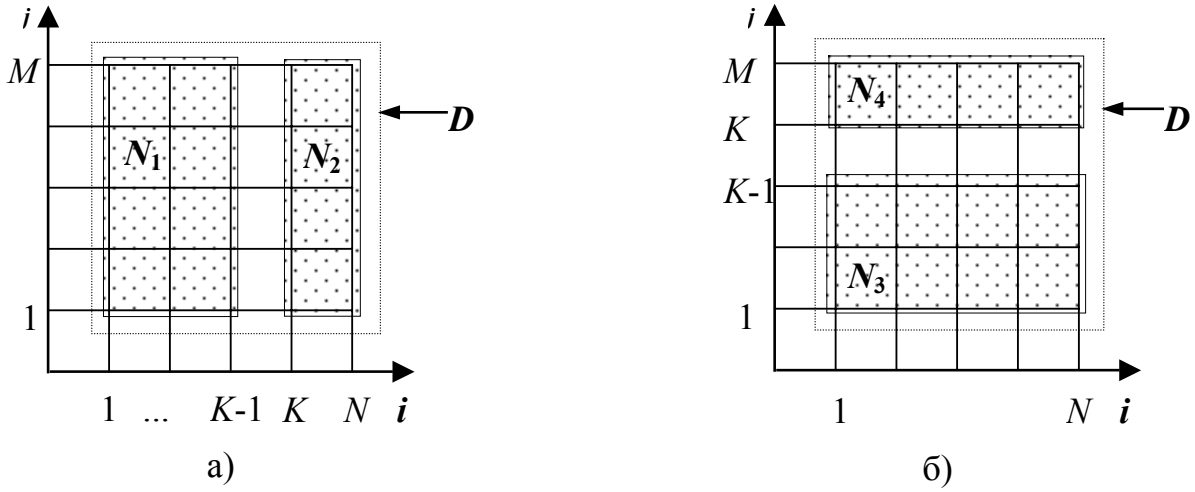


Рис. 11. Различные варианты разбиения пространства итераций D.

Используя данное разбиение, построим расщепление исходного гнезда циклов в виде последовательности тесных гнезд циклов:

```
for(i=1;i<=(K-1);i++)        // гнездо сканирует N1
   for(j=1;j<=M;j++)
      LoopBody(i,j);
for(i=K;i<=N;i++)            // гнездо сканирует N2
   for(j=1;j<=M;j++)
      LoopBody(i,j);
```

Данный фрагмент программы эквивалентен исходному гнезду циклов т.к. в нем все операции и порядок их исполнения точно такие, как в исходном гнезде. Однако порядок исполнения операций исходного гнезда можно сохранить в результирующем фрагменте только благодаря характеру разбиения $\{N_1, N_2\}$.

Пусть множества N_3 и N_4 разбивают множество D так, как показано на рисунке 11.б. Используя разбиение $\{N_3, N_4\}$, построим расщепление исходного гнезда циклов в виде последовательности тесных гнезд циклов:

```
for(i=1;i<=N;i++)            // гнездо сканирует N3
   for(j=1;j<=(K-1);j++)
      LoopBody(i,j);
for(i=1;i<=N;i++)            // гнездо сканирует N4
   for(j=K;j<=M;j++)
      LoopBody(i,j);
```

В этом фрагменте программы порядок исполнения операций отличается от соответствующего порядка исходного фрагмента. Т.к. в результате преобразования изменяется порядок исполнения операций исходного гнезда, то примене-

ние этого преобразования, без дополнительных проверок, может нарушить эквивалентность исходного и результирующего фрагментов программ.

Пусть проверка эквивалентности запрещает расщепление исходного гнезда в виде последовательности тесных гнезд циклов, используя разбиение $\{N_3, N_4\}$. Тогда для разбиения $\{N_3, N_4\}$ можно построить расщепление исходного гнезда цикла в виде структуры произвольно вложенных циклов[18]:

```
for(i=1;i<=N;i++)
{   for(j=1;j<=(K-1);j++)
      LoopBody(i,j);    // опорное гнездо сканирует N3
    for(j=K;j<=M;j++)
      LoopBody(i,j); // опорное гнездо сканирует N4
}
```

Полученный фрагмент программы эквивалентен исходному гнезду циклов т.к. в нем все операции и порядок их исполнения точно такие, как в исходном гнезде.

Для разбиения $\{N_3, N_4\}$ можно построить другое расщепление, с помощью добавления в гнездо циклов условных операторов:

---

[18] структура произвольно вложенных циклов – такой набор циклов, что некоторые циклы содержатся в телах других циклов.

```
for(i=1;i<=N;i++)
  for(j=1;j<=M;j++)
      if(j<=(K-1))
          LoopBody(i,j);// опорное гнездо сканирует N3
      else
          LoopBody(i,j);// опорное гнездо сканирует N4
```

Полученный фрагмент программы так же эквивалентен исходному гнезду циклов т.к. в нем все операции и порядок их исполнения точно такие, как в исходном гнезде.

Конец примера 39.

Расщепление с помощью добавления в гнездо циклов условных операторов (пример 39.) описано и используется П. Фотрье [69, 71]. Поэтому в данной работе метод этого расщепления рассматриваться не будет. Однако П. Фотрье не выделяет его в отдельный вид преобразования. Оно вытекает из подхода автора к хранению решетчатых графов и применения их, например, для преобразования программы к виду с однократным присваиванием [71].

Далее в работе будем предполагать, что расщепление требуется выполнять для гнезд циклов, не содержащих внешние переменные.

2.2.2 Построение расщепления в виде последовательности тесных гнезд циклов

Данный вид расщепления будем применять только к тесным гнездам циклов принадлежащих линейному классу программ. Далее в этом параграфе будем пользоваться следующей условной записью тесного гнезда циклов:

```
for I ∈ D

    LoopBody(I);
(10)
```

Вектор $\boldsymbol{I}=(I_1, I_2, \ldots, I_n)$ – вектор счетчиков циклов, образующих тесное гнездо. Множество D ($D \subset Z^{n+}$) – пространство итераций гнезда циклов. Для данного пространства итераций D будем обозначать через $\bar{D}$ – линейное пространство итераций.

Пусть имеется разбиение N_i (i=1..k) итерационного пространства D гнезда (10), причем каждое множество N_i (i=1..k) является выпуклым[19]. Используя данное разбиение, будем строить расщепление гнезда циклов (10) в виде последовательности тесных гнезд циклов с итерационными пространствами N_i (i=1..k):

```
for I ∈ Nσ(1)

    LoopBody(I);

for I ∈ Nσ(2)

    LoopBody(I);                                        (11)

...

for I ∈ Nσ(k)

    LoopBody(I);
```

где σ - некоторая перестановка множества k натуральных чисел $\{1, ..., k\}$.

---

[19] множество целых точек называется выпуклым, если оно может быть представлено как множество всех целых точек некоторого выпуклого линейного многогранника.

Пусть G – решетчатый граф, полученный объединением минимальных решетчатых графов, описывающих потоковую, анти- и выходную зависимость *по значению* во фрагменте программы (10). Пусть G' - фактор граф решетчатого графа G по множествам N_i. В фактор графе G' дуга направлена из N_i в N_j ($i \neq j$), если в исходном гнезде циклов (10) существует зависимость, решетчатый граф которой имеет дугу из N_i в N_j. Условие $i \neq j$ приводит к тому, что фактор граф G' не имеет петель по построению.

Утверждение 7. Пусть фактор граф G' является бесконтурным. Тогда для эквивалентности фрагментов программ (10) и (11) достаточно, чтобы выполнялось условие: если фактор граф G' содержит дугу из N_i в N_j, то во фрагменте программы (11) гнездо циклов с пространством итераций N_i находится по тексту программы раньше, чем гнездо циклов с пространством итераций N_j.

Доказательство. При выполнении условий теоремы в преобразованной программе сохраняется порядок исполнения всех пар итераций (***I***, ***J***) гнезда (10) для которых выполняется хотя бы одно из условий:

некоторое вхождение на итерации ***J*** использует значение, записанное другим вхождением на итерации ***I*** (потоковая зависимость по значению);

некоторое вхождение на итерации ***J*** записывает значение, которое было использовано другим вхождением на итерации ***I*** (антизависимость по значению);

некоторое вхождение на итерации ***J*** перезаписывает значение, которое было записано другим вхождением на итерации ***I*** (выходная зависимость по значению);

Следовательно, преобразованная программа (11) будет эквивалентна исходной программе (10).

Утверждение 8. Пусть фактор граф G' является бесконтурным. Если фактор граф G' содержит дугу из N_i в N_j, а во фрагменте программы (11) гнездо циклов с пространством итераций N_j находится по тексту программы раньше,

чем гнездо циклов с пространством итераций N_i, то существует фрагмент программы (10) не эквивалентный фрагменту программы (11).

Доказательство. Если выполняется условие утверждения, то во фрагменте программы (10) существует такая пара итераций $\boldsymbol{I} \in N_i$ и $\boldsymbol{J} \in N_j$, для которой выполняется хотя бы одно из условий утверждения 7. Пусть, для определенности, для этой пары выполняется условие: вхождение некоторой переменной X на итерации $\boldsymbol{J}$ использует значение, записанное другим вхождением X на итерации $\boldsymbol{I}$. Т.к. во фрагменте программы (11) гнездо с пространством итераций N_j находится по тексту программы раньше, чем гнездо циклов с пространством итераций N_i, то во фрагменте (11) вхождение X на итерации $\boldsymbol{J}$ использует значение, которое еще не было определено вхождением X на итерации $\boldsymbol{I}$. Следовательно, существует такой фрагмент программы (10), который не эквивалентен фрагменту программы (11). Этот вывод завершает доказательство.

Следствие. Если фактор граф G' имеет контур, то расщепление в виде (11) выполнить нельзя т.к. при любом расположении гнезд циклов в результирующем фрагменте (11) будет выполнено условие утверждения 8.

Алгоритм расщепления гнезда циклов (10) для случая, когда фактор граф G' является бесконтурным, имеет вид:

1. Строим фактор граф G' решетчатого графа G по множествам N_i (i=1..k).
2. Строим в графе G' правильную нумерацию вершин[20].
3. Преобразованную программу составляют гнезда циклов, пространства итераций которых – множества, соответствующие вершинам полученного графа. Эти гнезда циклов располагаем в порядке правильной нумерации.

Пространства итераций гнезд циклов, полученных в результате работы приведенного алгоритма расщепления, есть множества всех целочисленных то-

---

[20] Правильная нумерация вершин бесконтурного графа – такая нумерация, в которой нет дуг из вершин с большими номерами в вершины с меньшими номерами [38].

чек некоторых выпуклых линейных многогранников, каждый из которых задается набором линейных неравенств. Для получения левых и правых границ операторов циклов некоторого гнезда необходимо воспользоваться одним из алгоритмов генерации границ оператора цикла из системы линейных неравенств [58, 64].

При построении фактор графа G' решетчатого графа G по множествам N_i (i=1..k), требуется решать следующую задачу: определить, существует ли дуга в решетчатом графе G, начало которой лежит во множестве N_i, а конец – во множестве N_j. В следующем разделе будут предложены методы решения данной задачи.

2.2.2.1 Проверка существования дуги решетчатого графа между заданными выпуклыми множествами

Пусть даны выпуклые многогранники $\bar{H}_1 \subset R^n$, $\bar{H}_2 \subset R^n$, множество всех целых точек каждого есть H_1, H_2, соответственно. Пусть дан решетчатый граф G, вершины которого принадлежат R^n. Требуется определить, существует ли в графе G дуга из множества H_1 во множество H_2.

Эта задача эквивалентна следующей. Пусть имеется функция F и ее область определения – множество целых точек M, заданные с помощью квазиаффинных форм. Определить, существуют ли точки $\boldsymbol{I}$ и $\boldsymbol{J}$, где $\boldsymbol{J}=F(\boldsymbol{I})$, такие, что $\boldsymbol{I} \in H_1$ и $\boldsymbol{J} \in H_2$ (для минимального снизу графа наоборот – $\boldsymbol{I} \in H_2$ и $\boldsymbol{J} \in H_1$).

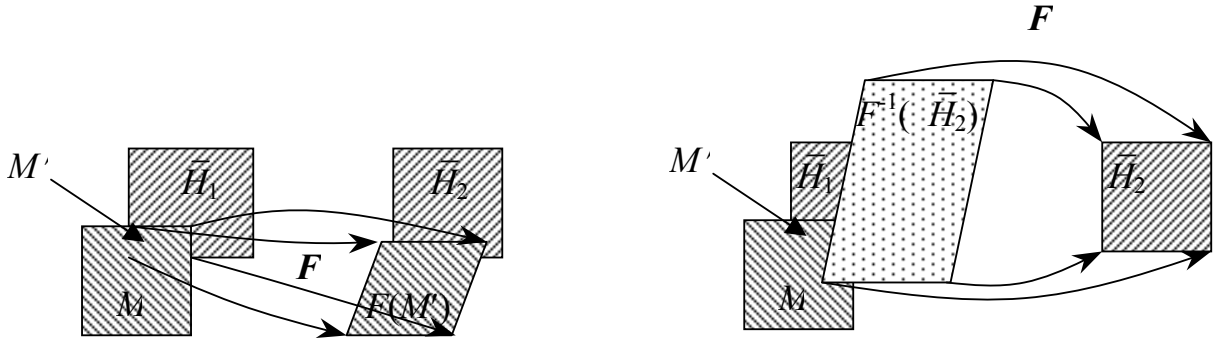
Пусть функция F задана с помощью аффинных форм, а множество M есть множество всех целых точек выпуклого многогранника $\bar{M}$. Тогда для решения задачи можно предложить очевидный алгоритм (все использующиеся в алгоритме множества изображены на рис. 11.а):

1. находим пересечение $M' = \bar{M} \cap \bar{H}_1$;

2. если выпуклый многогранник M' не содержит ни одной целой точки, то требуемых точек $\boldsymbol{I}$ и $\boldsymbol{J}$ не существует, заканчиваем алгоритм;

3. строим выпуклый многогранник $F(M')$;

4. если в $F(M') \cap \bar{H}_2$ содержится хотя бы одна целая точка, то искомые точки $\boldsymbol{I}$ и $\boldsymbol{J}$ существуют, в противном случае – нет.



а) б)

Рис. 12.

В данном алгоритме необходимо построить образ $F(M')$ выпуклого многогранника. Многогранник M' задан с помощью системы линейных неравенств: $M'=\{\boldsymbol{I} \mid \boldsymbol{a}^j\boldsymbol{I}\le b^j$, где $j=1..m$ ($\boldsymbol{a}^j\boldsymbol{I}$ – скалярное произведение векторов $\boldsymbol{a}^j\in R^n$ и $\boldsymbol{I}\in R^n$)$\}$. Переходить к описанию всех вершин M', и по их образу восстанавливать многогранник $F(M')$ в данном случае неэффективно. Если известна обратная функция F^{-1}, то можно получить описание многогранника $F(M')$ в виде системы линейных неравенств: $F(M')=\{\boldsymbol{J} \mid \boldsymbol{a}^jF^{-1}(\boldsymbol{J})\le b^j$, где $j=1..m\}$. Для реализации такого способа нужно искать обратную функцию к F, причем эта функция может оказаться необратимой в R^n.

Решить поставленную в начале пункта задачу можно без обращения функций или перехода к описанию многогранника с помощью вершин. Для этого за-

метим, что, имея функцию F и считая R^n ее областью определения, можно построить прообраз выпуклого многогранника относительно нее: если $N=\{\boldsymbol{J} \mid \boldsymbol{a}^j\boldsymbol{J}\le b^j$, где $j=1..m\}$, то прообраз N относительно F^{-1} есть $F^{-1}(N)=\{\boldsymbol{I} \mid \boldsymbol{a}^jF(\boldsymbol{I})\le b^j$, где $j=1..m\}$.

Учитывая все вышесказанное, можно предложить более эффективный и удобный для реализации алгоритм (все использующиеся в алгоритме множества изображены на рис. 11.б):

1. находим пересечение $M' = \bar{M} \cap \bar{H}_1$;

2. если выпуклый многогранник M' не содержит ни одной целой точки, то требуемых точек $\boldsymbol{I}$ и $\boldsymbol{J}$ не существует, заканчиваем алгоритм;

3. считая F определенной всюду в R^n, строим выпуклый многогранник $F^{-1}(\bar{H}_2)$, являющийся прообразом $\bar{H}_2$ относительно F;

4. если в $M' \cap F^{-1}(\bar{H}_2)$ содержится хотя бы одна целая точка, то искомые точки $\boldsymbol{I}$ и $\boldsymbol{J}$ существуют, в противном случае – нет.

Именно этот алгоритм использован в программной реализации расщепления в ОРС [54]. Заметим, что приведенный алгоритм позволяет решить указанную задачу даже в том случае, когда функция F и область M заданы квазиаффинными формами. Проверку на содержание целых точек в области, заданной квазиаффинными формами, можно выполнять в том же виде, что и работах [70, 71] (в данных работах эта задача решается при построении квазиаффинных функций, описывающих дуги решетчатого графа).

2.2.2.2 Использование расщепления для распараллеливания

Пусть в тесном гнезде циклов с итерационным пространством V имеется зависимость вхождения v от вхождения u, препятствующая эффективному распараллеливанию этого гнезда. Пусть дуги минимального снизу решетчатого графа программы, описывающие зависимость v от u, задаются функцией F, область

определения которой – множество H. Пусть для этой функции выполняется условие:

область определения H и область значения $F(H)$ не пересекаются.

Последнее условие означает, что все дуги решетчатого графа, описываемые функцией F, идут из множества $F(H)$ во множество H.

Пусть расщепление исходного гнезда циклов в соответствии с разбиением $\{H, V \backslash H\}$ возможно (если множество $V \backslash H$ не является выпуклым, то его можно разбить на набор выпуклых множеств). Тогда в результате расщепления, в каждом полученном гнезде исходной зависимости v от u не будет.

Пример 40. Рассмотрим гнездо циклов примера 31.

Минимальный решетчатый граф потоковой зависимости $a[j][i]$ от $a[i][j]$ изображен на рисунке 9. Как уже отмечалось в примере 27, все дуги указанного графа идут только из множества N_1 во множество N_2. Будем строить расщепление данного гнезда по разбиению $\{N_1, N_2\}$.

Кроме рассмотренной выше потоковой зависимости, существует антизависимость вхождения $a[i][j]$ от $a[j][i]$. Однако, решетчатый граф, описывающий данную антизависимость, полностью совпадает с решетчатым графом, описывающим потоковую зависимость $a[j][i]$ от $a[i][j]$. Т. к. других зависимостей в данном гнезде нет, то фактор граф минимального решетчатого графа программы по разбиению $\{N_1, N_2\}$ будет содержать только одну дугу, направленную из множества N_1 во множество N_2. Следовательно, в результирующем фрагменте программы гнездо циклов с итерационным пространством N_1 должно находиться по тексту программы раньше, чем гнездо циклов с итерационным пространством N_2. В результате применения описанного алгоритма расщепления получится следующий фрагмент программы:

```
for(i=1; i<=5; i++)     // гнездо сканирует N1
for(j=i; j<=5; j++)
    a[i][j] = a[j][i] + b[i][j];
for(i=2; i<=5; i++)     // гнездо сканирует N2
for(j=1; j<=(i-1); j++)
    a[i][j] = a[j][i] + b[i][j];
```

Из-за информационных зависимостей между вхождениями переменной «*a*» итерации внешнего цикла исходного гнезда нельзя исполнять одновременно (параллельно). После расщепления, все итерации внешнего цикла в каждом полученном гнезде можно исполнять одновременно, при условии, что вычисление второго гнезда циклов (по N_2) начнется строго после вычисления первого (по N_1).

Конец примера 40.

2.2.3 Построение расщепления в виде структуры произвольно вложенных циклов

Рассмотрим опорное гнездо циклов *SL* некоторого оператора. Занумеруем операторы циклов в этом гнезде в соответствии с порядком вложенности, начиная с самого внешнего цикла. Обозначим счетчик *i*-го цикла – I_i, где $i \in [1, n]$; тогда вектор счетчиков циклов опорного гнезда – $\boldsymbol{I}=(I_1, I_2, \ldots, I_n)$. Пусть *D* ($D \subset Z^{n+}$) – пространство итераций гнезда циклов *SL*. Пусть имеется разбиение N_i (i=1..k) итерационного пространства *D*, причем каждое множество N_i (i=1..k) является выпуклым. Тогда можно выполнить расщепление гнезда циклов *SL* в виде структуры произвольно вложенных циклов с помощью следующего алгоритма.

Алгоритм расщепления опорного гнезда циклов SL в виде структуры произвольно вложенных циклов:

Для произвольного опорного гнезда циклов SL, итерационное пространство которого $D \subset Z^{n+}$, будем обозначать через SL_d ($d \le n$) гнездо циклов, которое образуют первые d операторов циклов гнезда SL. Итерационное пространство гнезда SL_d будем обозначать через $D_d \subset Z^{d+}$. Будем считать, что в описании множеств $\overline{N}_i$ (i=1..k) не содержится избыточных неравенств[21].

1. Проверяем, существует ли в описании некоторого многогранника из набора $\overline{N}_i$ (i=1..k) неравенство $\boldsymbol{aI} \le b$, для которого одновременно выполняются условия:

С.4) в рассматриваемом гнезде не существует границы оператора цикла, которая бы определяла то же полупространство, что и $\boldsymbol{aI} \le b$;

С.5) в неравенстве $\boldsymbol{aI} \le b$ при I_n стоит нулевой множитель.

2. Если указанного неравенства не существует, то выполняем алгоритм замены в гнезде SL цикла по I_n на последовательность циклов по I_n, используя разбиение N_i (i=1..k) итерационного пространства гнезда SL; конец алгоритма.

3. Иначе, пусть неравенство $\boldsymbol{a'I} \le b'$ участвует в описании некоторого многогранника из $\overline{N}_i$ (i=1..k), и удовлетворяет условиям С.4), С.5). Тогда выполним следующие действия:

3.1 разрежем множества N_i (i=1..k) плоскостью $\boldsymbol{a'I}=b'$: в результате получим два набора множеств N_i^1 (i=1..k_1) и N_i^2 (i=1..k_2);

3.2 пусть в неравенстве $\boldsymbol{a'I} \le b'$ слагаемое $a_d I_d$ является ближайшим слева к $a_n I_n$ таким, что $a_d \ne 0$; разрежем пространство итераций гнезда SL_d – многогранник D_d – плоскостью $\boldsymbol{a'I}=b'$: в результате получим два многогранника D_d^1 и D_d^2;

3.3 выполняем алгоритм замены в гнезде SL_d цикла по I_d на последовательность циклов по I_d, используя разбиение $\{D_d^1, D_d^2\}$ итерационного пространства гнезда SL_d; в результате получим два гнезда SL^1 и SL^2 такие, что итерационные пространства гнезд $SL^1{}_d$ и $SL^2{}_d$ равны D_d^1 и D_d^2, соответственно;

3.4 пусть, для определенности, все множества набора N_i^1 (i=1..k_1) содержатся в итерационном пространстве гнезда SL^1, а множества N_i^2 (i=1..k_2) – в итерационном пространстве гнезда SL^2; выполним данный алгоритм расщепления для гнезда SL^1 и разбиения N_i^1 (i=1..k_1), затем для гнезда SL^2 и разбиения N_i^2 (i=1..k_2);

4. конец алгоритма.

В пунктах 2 и 3.3 описанного выше алгоритма, используется алгоритм замены в опорном гнезде SL_n цикла по I_n на последовательность циклов по I_n, используя разбиение N_i (i=1..k) итерационного пространства гнезда SL_n. Приведем этот алгоритм, считая, что не существует неравенства $\boldsymbol{aI} \le b$, участвующего в описании некоторого многогранника из $\overline{N_i}$ (i=1..k), и для которого одновременно выполняются условия С.4), С.5).

Алгоритм замены в опорном гнезде SL_n цикла по I_n на последовательность циклов по I_n, используя разбиение N_i (i=1..k) итерационного пространства D гнезда циклов SL_n:

1. копируем тело заменяемого цикла по I_n в переменную *body*;

2. генерируем k операторов циклов, со счетчиком I_n и телом *body*; пусть m–тый сгенерированный цикл соответствует N_m;

---

[21] неравенство называется избыточным, если удаление его из системы не приводит к увеличению числа решений данной системы.

3. для m–го оператора цикла ($m \in [1, k]$), с помощью алгоритма [64], используя описание $\bar{N}_m$, генерируем верхние и нижние границы изменения счетчика I_n;

4. из полностью построенных операторов циклов по I_n формируем упорядоченную последовательность циклов так, чтобы, заменив цикл по I_n в исходном гнезде, получилось новое гнездо циклов, сканирующее $N_1 \cup N_2 \cup \ldots \cup N_k$ (равное D) в лексикографическом порядке; последнее условие обеспечивает сохранение порядка исполнения всех операций исходного гнезда;

5. заменяем исходный цикл по I_n на построенную в п.4 упорядоченную последовательность циклов по I_n;

6. конец алгоритма.

В пункте 4 приведенного выше алгоритма необходимо решать следующую задачу. Пусть дано n-1 мерное опорное гнездо циклов $SL_{n\text{-}1}$, вектор счетчиков которого ($I_1, I_2, \ldots, I_{n\text{-}1}$). В тело самого внутреннего цикла гнезда $SL_{n\text{-}1}$ добавляется последовательность из двух циклов L_1 и L_2. Обозначим через $SL_{n\text{-}1}_L_1$ гнездо, составленное из циклов $SL_{n\text{-}1}$ и L_1, а через $SL_{n\text{-}1}_L_2$ – гнездо, составленное из циклов $SL_{n\text{-}1}$ и L_2. Итерационное пространство гнезда $SL_{n\text{-}1}_L_1$ есть $N_1(\subset Z^{n+})$, гнезда $SL_{n\text{-}1}_L_2$ – $N_2(\subset Z^{n+})$. Требуется определить, какой из циклов L_1, L_2 должен располагаться раньше по тексту программы, чтобы полученное гнездо циклов сканировало $N_1 \cup N_2$ в лексикографическом порядке.

Эта задача может быть решена следующим образом. С помощью одного из алгоритмов [58, 64, 89] определить условия, накладываемые на ($I_1, I_2, \ldots, I_{n\text{-}1}$), которым удовлетворяют первые n-1 координат всех целых точек принадлежащих N_1 и N_2. Эти условия имеют вид системы неравенств. Обозначим их через $P_1(I_1, I_2, \ldots, I_{n\text{-}1})$ и $P_2(I_1, I_2, \ldots, I_{n\text{-}1})$, соответственно для N_1 и N_2. Пусть $P_{12}(I_1, I_2, \ldots, I_{n\text{-}1})$ – система, полученная пересечением систем $P_1(I_1, I_2, \ldots, I_{n\text{-}1})$ и $P_2(I_1, I_2, \ldots, I_{n\text{-}1})$. Таким образом, системе $P_{12}(I_1, I_2, \ldots, I_{n\text{-}1})$ удовлетворяют все итерации гнезда $SL_{n\text{-}1}$, при которых исполняется и тело цикла L_1 и тело цикла L_2. Если

$P_{12}(I_1, I_2, \ldots, I_{n-1})$ не имеет целочисленных решений, то циклы L_1 и L_2 могут быть расположены в любом порядке. Если $P_{12}(I_1, I_2, \ldots, I_{n-1})$ имеет целочисленные решения, то пусть система $Q(I_1, I_2, \ldots, I_n)$ получена пересечением систем неравенств $P_{12}(I_1, I_2, \ldots, I_{n-1})$, $ubL_1(I_1, I_2, \ldots, I_n)$, $lbL_2(I_1, I_2, \ldots, I_n)$, где ubL_1 – условия верхней границы для счетчика цикла L_1, а lbL_2 – условия нижней границы для счетчика цикла L_2. Пусть $Q(I_1, I_2, \ldots, I_n)$ не имеет целочисленных решений. Тогда для любого фиксированного значения счетчика внешних циклов, при котором исполняется и тело цикла L_1, и тело цикла L_2, все значения счетчика цикла L_1 меньше значений счетчика цикла L_2. Поэтому, в данном случае итерации цикла L_1 должны выполняться раньше, итераций цикла L_2. Следовательно, цикл L_1 должен находиться раньше цикла L_2 по тексту программы. Если $Q(I_1, I_2, \ldots, I_n)$ имеет целочисленные решения, то цикл L_2 должен находиться раньше цикла L_1 по тексту программы.

Пример 41. Пусть имеется двумерное гнездо циклов:

```
for(i=1; i<=5; i++)     // гнездо сканирует D

for(j=1; j<=5; j++)

    LoopBody(i,j);
```

Пусть требуется расщепить данное гнездо в виде структуры произвольно вложенных циклов, используя разбиение итерационного пространства, изображенное на рисунке 13.

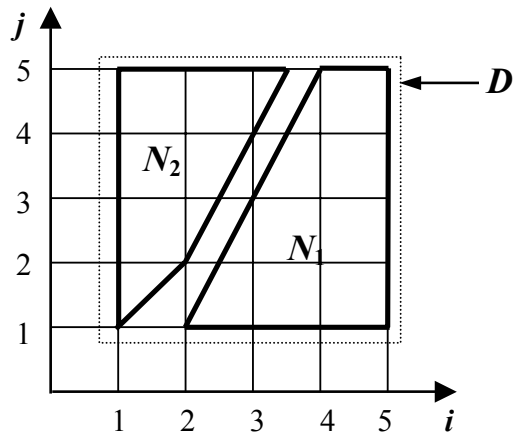


Рис. 13.

Заметим, что в описании $\bar{N}_1$ и $\bar{N}_2$ нет неравенств, удовлетворяющих условиям С.4), С.5), поэтому можно переходить к замене цикла по j последовательностью циклов по j с помощью описанного выше алгоритма.

Множества $\bar{N}_1$ и $\bar{N}_2$ описываются следующим образом:

$$\bar{N}_1 = \begin{cases} 2 \le i \le 5 \\ j \le 5 \\ j \le (2i-3) \\ 1 \le j \end{cases} \qquad \bar{N}_2 = \begin{cases} 1 \le i \le 4 \\ i \le j \\ (2i-2) \le j \\ j \le 5 \end{cases}$$

Цикл L_1 (по j), соответствующий $\bar{N}_1$, заголовок которого построен с помощью алгоритма [64], будет таким:

```
for(j=1; j<=min(2i-3, 5); j++)

    LoopBody(i,j);
```

А цикл L_2 (по j), соответствующий $\bar{N}_2$, будет таким:

```
for(j=max(2i-2, i); j<=5; j++)

    LoopBody(i,j);
```

Теперь определим, какой цикл L_1 или L_2 должен располагаться раньше по тексту программы. Система неравенств $P_1(i)$, описывающая условия для всех i, при которых пары (i, j) принадлежат N_1, состоит из неравенств: $2 \le i$, $i \le 5$. Аналогичная система $P_2(i)$ для N_2, состоит из неравенств: $1 \le i$, $i \le 3$. Система $P_{12}(i)$ – пересечение систем $P_1(i)$ и $P_2(i)$ – состоит из неравенств: $2 \le i$, $i \le 3$, и имеет целочисленные решения. Система $ubL_1(i, j)$, описывающая условия верхней границы цикла L_1, состоит из неравенств: $j \le (2i\text{-}3)$, $j \le 5$. Система $lbL_2(i, j)$, описывающая условия нижней границы цикла L_2, состоит из неравенств: $(2i\text{-}2) \le j$, $i \le j$. Тогда, система $Q(i, j)$ будет состоять из неравенств: $2 \le i$, $i \le 3$, $j \le (2i\text{-}3)$, $j \le 5$, $(2i\text{-}2) \le j$, $i \le j$. Эта система не имеет даже вещественных решений. Следовательно, в результирующем гнезде циклов, цикл L_1 должен находиться раньше по тексту программы, чем цикл L_2. После расщепления получим следующий фрагмент программы:

```
for(i=1; i<=5; i++)
{   for(j=1; j<=min(2i-3, 5); j++)
      LoopBody(i,j);
    for(j=max(2i-2, i); j<=5; j++)
      LoopBody(i,j);
}
```

Конец примера 41.

Рассмотрим теперь случай, когда существует неравенство $\boldsymbol{a}'\boldsymbol{I} \le b'$, участвующее в описании некоторого многогранника $\overline{N}_m$, для которого выполняются условия С.4), С.5). В этом случае, в результате генерации верхних и нижних границ для счетчика I_n цикла L_m, соответствующего N_m, не может быть получено

ограничение $\boldsymbol{a'I} \le b'$ т.к. в описании этого ограничения при I_n стоит 0. Обозначим через $SL_{n\text{-}1}_L_m$ гнездо циклов, составленное из циклов гнезда $SL_{n\text{-}1}$ добавлением цикла L_m в тело самого вложенного. Т. к. ни одна из границ внешних для L_m циклов гнезда $SN_{n\text{-}1}_L_m$ не определяет неравенство $\boldsymbol{a'I} \le b'$, то в описании линейного пространства итераций $\bar{D}_m$ гнезда циклов $SN_{n\text{-}1}_L_m$, отсутствует ограничение $\boldsymbol{a'I} \le b'$. Тогда итерационное пространство D_m построенного гнезда $SN_{n\text{-}1}_L_m$ может не совпадать с требуемым пространством N_m. Пункт 3 предложенного алгоритма расщепления в виде структуры произвольно вложенных циклов помогает избежать этой ситуации.

Пример 42. Рассмотрим исходное гнездо циклов в примере 39. Пусть требуется выполнить его расщепление в виде структуры произвольно вложенных циклов, в соответствии с разбиением итерационного пространства, изображенным на рисунке 11.

В описании данного разбиения существуют неравенства, удовлетворяющие условиям С.4), С.5). Например, в описании $\bar{N}_1$ это неравенство $i \le (K\text{-}1)$, а в описании $\bar{N}_2$ это неравенство $K \le i$. Если, несмотря на этот факт, применить алгоритм замены цикла по j на последовательность циклов по j, используя разбиение $\{\ \bar{N}_1,\ \ \bar{N}_2\}$, то получим фрагмент программы

```
for(i=1;i<=N;i++)
{   for(j=1;j<=M;j++)
      LoopBody(i,j);
    for(j=1;j<=M;j++)
      LoopBody(i,j);
}
```

Этот фрагмент программы не эквивалентен исходному гнезду циклов. Для выполнения требуемого расщепления необходимо сначала модифицировать цикл по i. Данная модификация выполняется с помощью пункта 3 алгоритма расщепления, используя, например, неравенство $i \leq (K-1)$.

Конец примера 42.

Замечание. Как видно из предложенного в данном пункте алгоритма расщепления в виде структуры произвольно вложенных циклов, в результате такого расщепления порядок следования операций исходного фрагмента сохранится в преобразованной программе. Следовательно, проверять эквивалентность этого преобразование не требуется.

2.2.3.1 Использование расщепления для распараллеливания: общий случай

В [21, параграф 7.2] предлагается метод распараллеливания программ на макроуровне с помощью, так называемых, макрографов зависимостей[22]. Он по-

[22] макрограф зависимости [21, с. 405] есть ни что иное, как фактор граф решетчатого графа по некоторым множествам итераций.

зволяет выявить подмножества пространств итераций, в которых все итерации могут быть выполнены одновременно.

Предложенный метод распараллеливания может быть реализован либо на основе описания минимальных решетчатых графов в виде функций, либо на основе разверток [21, параграф 7.2]. Далее в работе, будет рассматриваться только реализация на основе описания минимальных решетчатых графов в виде функций. В этом случае авторы работы [21] предлагают строить макрограф зависимости сначала для самого внешнего цикла гнезда. Если после анализа построенного графа оказывается, что некоторое множество итераций *M* не может быть выполнено одновременно, то с помощью расщепления выделяются гнезда циклов, сканирующее *M*. Затем к телу самого внешнего цикла этого гнезда применяется этот же анализ: строятся новые решетчатые графы, фактор граф и т.д. [21, с. 428]. Этот процесс продолжается до тех пор, пока либо программа не будет распараллелена, либо пока не будут проанализированы все уровни вложенности циклов. Стоит отметить, что в работе [21] не конкретизируется, как может быть выполнено расщепление на практике в автоматическом режиме. Авторы указанной работы пользуются расщеплением концептуально, а в примерах выполняют вручную.

Предложенные в настоящей работе методы расщепления, с одной стороны, могли бы послужить основой для реализации такого макроанализа на практике в автоматическом режиме. С другой стороны, используя расщепление и связь существования функций, описывающих дуги решетчатого графа, со свойством ParDo циклов (см. пункт 2.1), можно усовершенствовать метод распараллеливания [21, с. 428]. В этом случае уже не будет необходимости строить решетчатые графы для каждого уровня вложенности циклов. А по функциям, описывающим дуги решетчатого графа, можно определять глубины вложенности циклов, на которых нужно выпонять расщепление, распараллеливающее программу. Этот

метод также является обобщением метода, предложенного в пункте 2.2.2.2. Рассмотрим его в действии на следующем примере.

Пример 43. Рассмотрим фрагмент программы [21, с. 385, пример 6.4] при n=10.

```
for(i=1; i<=10; i++)
  for(j=1; j<=10; j++)
      u[i+j] = u[20+1-i-j];
```

Решетчатый граф G потоковой зависимости данного гнезда изображен на рисунке 14. Этот граф совпадает с решетчатым графом, описывающим зависимость $u[20+1-i-j]$ от $u[i+j]$, т.к. других потоковых зависимостей в данном фрагменте нет.

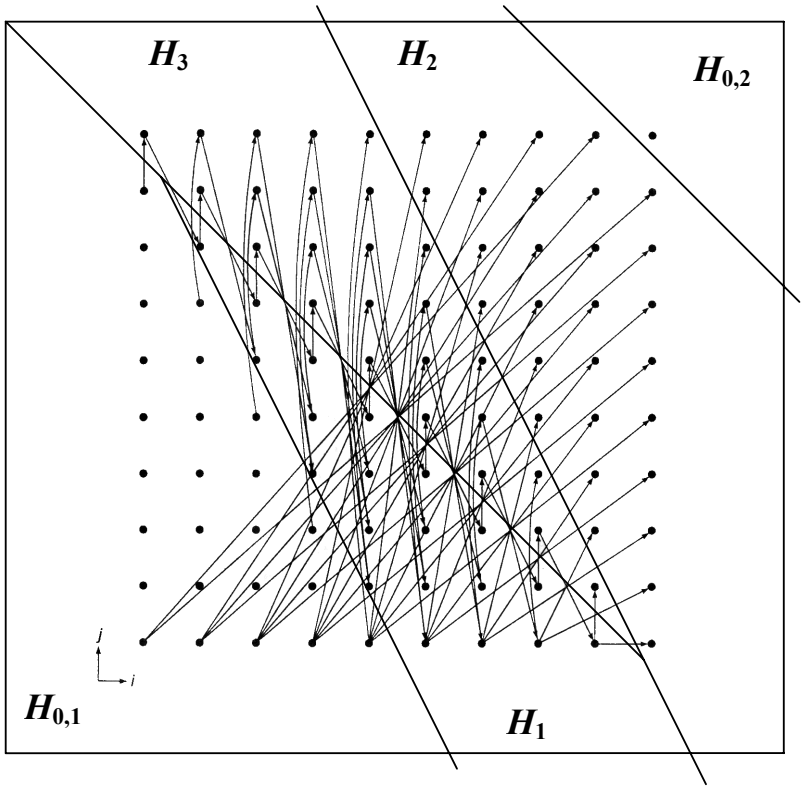


Рис. 14. Решетчатый граф потоковой зависимости $u[20+1-i-j]$ от $u[i+j]$ ([21, с. 223, 457]), с выделенными множествами вершин.

Ни цикл по i, ни цикл по j не является циклом ParDo по графу G. В тоже время, этот фрагмент программы имеет значительный параллелизм. По утверждению авторов работы [21], этот пример тестировался на многих зарубежных распараллеливающих компиляторах и автономных системах выявления параллелизма, но всюду безуспешно [21, с. 390].

Выпишем функции, описывающие дуги решетчатого графа G.

Функция	**Область определения**	**Номер альтернативного многогранника**
$F_1(i, j)=(i-1, -2i-j+22)$	$H_1=\begin{cases} -2i-j \le -12 \\ i+j \le 10 \\ -j \le -1. \end{cases}$	1
$F_2(i, j)=(-i-j+20, 1)$	$H_2=\begin{cases} -2i-j \le -21 \\ i+j \le 19 \\ i \le 10 \\ j \le 10. \end{cases}$	1
$F_3(i, j)=(i, -2i-j+21)$	$H_3=\begin{cases} -i-j \le -11 \\ 2i+j \le 20 \\ j \le 10. \end{cases}$	2
$\varnothing$	$H_{0,1}=\begin{cases} 2i+j \le 11 \\ -i \le -1 \\ -j \le -1. \end{cases}$	—
$\varnothing$	$H_{0,2}=\begin{cases} -i-j \le -20 \\ i \le 10 \\ j \le 10. \end{cases}$	—

Цикл по i не является циклом ParDo по графу G т.к. существуют функции $F_1(i, j)$ и $F_2(i, j)$, при построении которых использовался альтернативный многогранник с номером 1. Построим фактор граф G' решетчатого графа G по множествам H_1, H_2, $V/(H_1 \cup H_2)$. Множество $V/(H_1 \cup H_2)$ не является выпуклым, поэтому вместо набора $\{H_1, H_2, V/(H_1 \cup H_2)\}$ возьмем набор H_1, H_2, H_3, $H_{0,1}$, $H_{0,2}$. Фактор граф G' по данному набору будет содержать контур, связывающий множества H_1 и H_3 (см. рисунок 14). Итерации каждого множества, не принадлежащего контуру, могут быть выполнены одновременно [21, параграф 7.2].

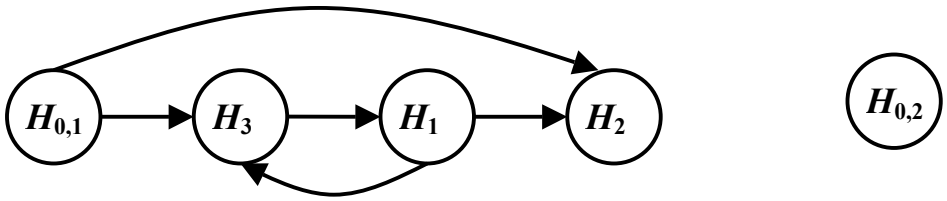


Рис. 15. Фактор граф решетчатого графа G по разбиению $\{H_1, H_2, H_3, H_{0,1}, H_{0,2}\}$.

К разбиению $\{H_1, H_2, H_3, H_{0,1}, H_{0,2}\}$ алгоритм расщепления в виде последовательности тесных гнезд циклов применять нельзя из-за наличия контура в фактор графе. В результате применения расщепления в виде структуры произвольно вложенных циклов не будет модифицирован внешний цикл (по i). Поэтому, после такого расщепления, итерации только внутренних циклов (на глубине вложенности 2) можно будет исполнять одновременно; параллелизм программы в этом случае может быть увеличен.

Заметим, что алгоритм расщепления в виде последовательности тесных гнезд циклов можно применить к разбиению: $\{H_2, H_{1,3}, H_{0,1}, H_{0,2}\}$, где $H_{1,3}$=

$H_1 \cup H_3$. В результате указанного расщепления будет получен следующий фрагмент программы:

```
for(i=1; i<=5; i++)     // сканирует H0,1
  for(j=1; j<=(11-2*i); j++)
     u[i+j] = u[20+1-i-j];
for(i=1; i<=9; i++)     // сканирует H1,3
  for(j=max(1,12-2*i); j<=min(10,20-2*i); j++)
     u[i+j] = u[20+1-i-j];
for(i=6; i<=10; i++)    // сканирует H2
  for(j=(21-2*i); j<=min(10,19-i); j++)
     u[i+j] = u[20+1-i-j];
u[20] = u[1]; // сканирует H0,2
```

Внешние и внутренние циклы в гнездах, сканирующих $H_{0,1}$ и H_2, являются циклами ParDo по решетчатому графу потоковой зависимости данного фрагмента. В данном случае, итерации всех указанных циклов могут быть выполнены одновременно. Отметим, что в совокупности подмножества $H_{0,1}$ и H_2 содержат 50% всех итераций исходного цикла.

Теперь в полученном фрагменте программы рассмотрим гнездо циклов, сканирующее $H_{1,3}$. Как было показано выше, его нельзя разбить так, чтобы итерации внешних циклов (на глубине вложенности 1) можно было выполнять одновременно. В этом гнезде перейдем к анализу циклов, находящихся на глубине вложенности 2 (т.е. к циклу по j). Но для этого мы не будем заново строить решетчатый граф, как предлагается в работе [21], а воспользуемся уже построен-

ным. Снова построим фактор граф G'' решетчатого графа G, по разбиению $\{H_3, H_{1,3}\backslash H_3\}$, но при этом построении не будем использовать функции, которым соответствует альтернативный многогранник с номером 1 (т.к. мы не планируем изменять порядок итераций цикла на глубине 1). Фактор граф G'' будет иметь единственную дугу из H_1 в H_3. Теперь будем выполнять расщепление рассматриваемого гнезда в соответствии с разбиением $\{H_3, H_{1,3}\backslash H_3\}$. Либо выполним расщепление в виде структуры произвольно вложенных циклов. Либо выполним расщепление цикла на глубине 2 в виде последовательности тесных гнезд циклов. В одном и другом случае получим фрагмент программы, в котором оба внутренних цикла имеют тип ParDo по графу G, и могут быть выполнены параллельно:

```
for(i=1; i<=9; i++)
{  for(j=max(1,12-2*i); j<i; j++)    // сканирует H1
      u[i+j] = u[20+1-i-j];
   for(j=i; j<=min(10,20-2*i); j++)  // сканирует H3
      u[i+j] = u[20+1-i-j];
}
```

Конец примера 43.

2.2.4 Сравнение различных видов расщеплений

В данном параграфе было рассмотрено три различных вида расщепления. Расщепление в виде последовательности тесных гнезд циклов имеет существенное ограничение, по сравнению с двумя другими видами расщеплений: оно может быть выполнено не для всякого гнезда циклов. Кроме этого, после исполь-

зования расщепления в виде последовательности тесных гнезд циклов, порядок следования операций в исходном и преобразованном фрагментах может различаться. Однако расщепление в виде последовательности тесных гнезд позволяет расщеплять гнезда, модифицируя циклы на более внешнем уровне вложенности, что может способствовать более эффективному распараллеливанию программы. Также, в результате этого расщепления получается код, значительно более удобный для чтения, чем код, полученный в результате других видов расщеплений. Расщепление, выполняемое с помощью добавления в тело цикла условных операторов, приводит к увеличению количества проверок, выполняемых на каждой итерации цикла, а также значительно ухудшает читабельность получаемого кода [71, с. 25]. Кроме того, такое расщепление не позволяет выделять циклы, итерации которых можно выполнять одновременно (т.к. сами операторы циклов не модифицируются).

2.3 Подстановка переменных и экспансия массивов

В данном параграфе будут описаны методы выполнения подстановки переменных и экспансии массивов, которые применимы к любым переменным в циклах: как к индексным, так и к безындексным. Эти преобразования иногда позволяют устранять выходные, анти- и даже потоковые зависимости, в результате чего параллелизм программы может увеличиться. Методы указанных преобразований основаны на расщеплении гнезд циклов и представлении решетчатых графов в виде набора функций.

Подстановка переменных и экспансия массивов неоднократно описывались в литературе [8, 14, 57, 65, 77] и использовались на практике для безындексных переменных или близких простых случаев. Экспансия массивов, позволяющая устранять выходные и антизависимости, для любых переменных из всего линейного класса программ было впервые разработано П. Фотрье [69]. В этой и других работах П. Фотрье расщепляет итерационные пространства с помощью добавления условных операторов в тело цикла [71, с. 25]. Подстановка индексных переменных, позволяющая устранять анти- и потоковые зависимости, предлагается в [47]. В этой работе ([47]) приводятся абстрактные формулы, и не указываются алгоритмы или методы выполнения данного преобразования.

В настоящее время подстановка индексных переменных и экспансия массивов реализованы в Открытой распараллеливающей системе [55] на основе расщепления в виде последовательности тесных гнезд циклов.

2.3.1 Подстановка переменных

Подстановка переменных является обобщением подстановки вперед [45, с. 20], [57, с. 156], которая выполняется либо для вхождений безындексных переменных, либо для вхождений с одинаковыми индексными выражениями. Последнее преобразование позволяет подставить правую часть оператора присваи-

вания вместо последующих вхождений левой части этого же оператора в другие операторы (если это не приводит к неэквивалентной программе). В частном случае, когда правая часть оператора присваивания является константой, это преобразование называется протягиванием констант [57, с. 146]. Условия, при которых подстановка вперед приводит к эквивалентной программе, доказываются в [45, с. 20].

Пример 44. (Подстановка вперед.)

Фрагмент программы

```
X[k]=A[i+2]+B;

B=X[k]+D[i];
```

эквивалентен следующему фрагменту:

```
X[k]=A[i+2]+B;

B=A[i+2]+B+D[i];
```

В последнем фрагменте уже отсутствуют потоковые зависимости.

Конец примера 44.

Определение. Оператор, правая часть которого подставляется при подстановке индексной переменной, будем называть *базисным*. Использование, вместо которого подставляется правая часть базисного оператора, будем называть *целевым*.

Пример 45. (Простой пример подстановки переменной.)

Фрагмент программы

```
for(i=1;i<N;i++)
{   X[i]=A[i+2]+B;
    Y[i]=X[i-1]+D[i];
}
```

эквивалентен следующему фрагменту:

```
    X[1]=A[3]+B;
    Y[1]=X[0]+D[1];
for(i=2;i<N;i++)
{   X[i]=A[i+2]+B;
    Y[i]=A[i+1]+B+D[i];
}
```

В результирующем фрагменте, в отличие от исходного, отсутствуют потоковые зависимости.

В данном примере, в отличие от примера 44, пришлось модифицировать правую часть базисного оператора, расщеплять цикл и проверять дополнительные условия эквивалентности (помимо условий принадлежности программы линейному классу). Несмотря на это, выполнить подстановку в данном примере просто. В общем случае, (например, когда размерность гнезда циклов больше 1, когда имеется несколько генераторов целевого вхождения с нетривиальными индексными выражениями) выполнение этого преобразования вручную может потребовать значительных усилий.

Конец примера 45.

Пусть требуется выполнить подстановку правой части базисного оператора *Stmt* в целевое вхождение *v* индексной переменной. Будем рассматривать фрагмент программы, образованный самым внешним циклом, содержащим и оператор *Stmt* и целевое вхождение *v*. Если такого цикла нет, то будем рассматривать фрагмент программы, содержащий и оператор *Stmt* и целевое вхождение *v*. Для упрощения изложения будем считать, что рассматриваемый фрагмент программы не содержит условных операторов. Пусть в графе информационных зависимостей нет дуг антизависимости, выходящих из вхождений, находящихся в правой части базисного оператора. Тогда для автоматического выполнения подстановки правой части оператора *Stmt* вместо вхождения *v* можно использовать следующий алгоритм.

Алгоритм выполнения подстановки переменной.

1. построим простой решетчатый граф G, описывающий потоковую зависимость по значению целевого вхождения v от всех генераторов, содержащихся в рассматриваемом фрагменте программы;

2. из всех функций, описывающих дуги графа G, выберем только те, которые соответствуют зависимости целевого вхождения от генератора базисного оператора; пусть это аффинные функции F_k (k=1..n), заданные на выпуклых подмножествах H_k (k=1.. n) опорного пространства V целевого вхождения;

3. производим расщепление опорного гнезда циклов целевого вхождения в соответствии с разбиением $\{H_1, \ldots, H_n, V \setminus (H_1 \cup \ldots \cup H_n)\}$;

4. в каждом полученном гнезде, сканирующем H_k (k=1.. n), заменяем целевое вхождение правой частью базисного оператора *Stmt*; при этом каждую компоненту вектора счетчиков циклов $\boldsymbol{I}$, входящую в правую часть базисного оператора *Stmt*, заменяем на соответствующую компоненту $F_k(\boldsymbol{I})$.

Пример 46. (Демонстрируется расщепление итерационного пространства опорного гнезда циклов и подстановку правой части, без необходимости ее изменения.)

Рассмотрим фрагмент программы умножения многочленов (см. пример 5). Пусть требуется подставить правую часть оператора $c[k]=0$ в использование $v1$. Простой решетчатый граф, описывающий потоковую зависимость по значению $v1$ от $u1$, $u2$, задается тройками $(u1, H^1_{u1,v1}, F_{u1,v1})$, $(u1, H^2_{u1,v1}, F_{u1,v1})$, $(u2, H_{u2,v1}, F_{u2,v1})$ (см. пример 25). Выполним расщепление опорного гнезда для вхождения $v1$ в соответствии с разбиением $\{H^1_{u1,v1}, H^2_{u1,v1}, H_{u2,v1}\}$. В гнездах циклов, сканирующих $H^1_{u1,v1}$ и $H^2_{u1,v1}$, заменим $c[i+j]$ на 0 (правую часть базисного оператора). В результате получим следующий фрагмент программы:

```
for(k=0; k<=(N+M); k++)
    c[k]=0;
for(i=0; i<=0; i++)          // цикл по H^1_{u1,v1}
    for(j=0; j<=N; j++)
      c[i+j]=0+a[i]*b[j];
for(i=1; i<=M; i++)          // цикл по H^2_{u1,v1}
    for(j=N; j<=N; j++)
      c[i+j]=0+a[i]*b[j];
for(i=1; i<=M; i++)          // цикл по H_{u2,v1}
    for(j=0; j<=(N-1); j++)
      c[i+j]=c[i+j]+a[i]*b[j];
```

После упрощения данного фрагмента (раскрутка циклов [45], удаление неиспользуемых операторов) получим программу, эквивалентную исходной:

```
for(j=0; j<=N; j++)

    c[j]=a[0]*b[j];

for(i=1; i<=M; i++)

    c[i+N]=a[i]*b[N];

for(i=1; i<=M; i++)

    for(j=0; j<=(N-1); j++)

      c[i+j]=c[i+j]+a[i]*b[j];
```

Конец примера 46.

Пример 47. (Демонстрируется преобразование правой части базисного оператора при подстановке.)

Рассмотрим фрагмент программы:

```
x=b[N];

for(i=1;i<=N;i++)

{   a[i]=(b[i]+x)*0.5; // оператор Stmt1
                 v

    x=b[i]+c[2*i]; // оператор Stmt2
    u

}
```

Итерации данного цикла исполнять одновременно нельзя в виду наличия потоковой зависимости *v* от *u* и выходной самозависимости *u* от *u*. Выполнить раз-

биение [49, с. 54] этого цикла также нельзя. Выполним подстановку правой части оператора *Stmt2* вместо использования *v*.

Дуги решетчатого графа, описывающего потоковую зависимость по значению *v* от *u*, определяются функцией $F(i)=i-1$, заданной на целых точках отрезка [2, *N*]. Для выполнения подстановки расщепим цикл по i, в соответствии с разбиением {[1,1], [2, *N*]}. После этого, в цикле, сканирующем [2, *N*], заменим вхождение *v* модифицированной правой частью оператора *Stmt2*. Причем правая часть оператора *Stmt2* модифицируется следующим образом: всякое вхождение переменной *i* заменяется на выражение *i*-1. После подстановки получим следующий фрагмент программы:

```
x=b[N];

for(i=1;i<=1;i++)        // Loop1

{   a[i]=(b[i]+x)*0.5;

    x=b[i]+c[2*i];

}

for(i=2;i<=N;i++)        // Loop2

{   a[i]=(b[i]+b[i-1]+c[2*(i-1)])*0.5;

    x=b[i]+c[2*i];

}
```

Полученный фрагмент программы более пригоден для оптимизации, чем исходный: к нему можно применить большее количество оптимизирующих преобразований. Выполним раскрутку цикла ***Loop*1** и разбиение цикла ***Loop*2**. Получим следующий фрагмент программы:

```
x=b[N];

a[1]=(b[1]+x)*0.5;

for(i=2;i<=N;i++)        // Loop2.1

    a[i]=(b[i]+b[i-1]+c[2*(i-1)])*0.5;

for(i=2;i<=N;i++)        // Loop2.2

    x=b[i]+c[2*i];
```

В полученном фрагменте итерации цикла ***Loop*2.1** можно исполнять одновременно. Гнездо циклов ***Loop*2.2** может быть заменено на один оператор – «x=b[N]+c[2*N];», а если переменная x далее в программе не используется, то это гнездо может быть удалено.

Конец примера 47.

2.3.2 Экспансия массивов

Экспансия массивов используется для устранения анти- и выходных зависимостей [57, 65]. Это преобразование также позволяет приводить фрагмент программы к виду с однократным присваиванием (когда каждая переменная или элемент массива получает свое значение только один раз).

Экспансия массива заключается в замене некоторого генератора другим генератором новой переменной так, чтобы генератор новой переменной на каждой итерации опорного гнезда циклов записывал разные элементы массива. Часто для этого необходимо, чтобы размерность новой переменной была больше, чем размерность переменной заменяемого генератора. Иногда размерность новой переменной может совпадать с размерностью переменной заменяемого генератора. В этом случае, такое преобразование полностью совпадает с извест-

ным преобразованием «переименование переменной», которое описано для безиндексных переменных или близких простых случаев в [8, 57, 65, 77]. В работе [47] предлагается переименование массивов, однако в этой работе приводятся только абстрактные формулы и не указываются алгоритмы или методы выполнения этого преобразования. В некоторых случаях при замене генератора, размерность новой переменной может быть меньше, размерности переменной заменяемого генератора (то, как определяется размерность новой переменной, указывается в алгоритме экспансии массива данного пункта). После замены генератора производится модификация некоторых использований для сохранения эквивалентности исходной и преобразованной программ.

Пример 48. (Переименование скалярной переменной.)

Рассмотрим фрагмент программы:

```
 A = B + C; //Stmt1
u1
 B = A + 2; //Stmt2
     v1
 A = A * 3; //Stmt3
u2   v2
 F = A * A + D; //Stmt4
     v3   v4
```

В данном фрагменте имеются потоковые зависимости $v1$ от $u1$ и $v2$ от $u1$, указывающие, что операторы *Stmt2*, *Stmt3* должны исполняться только после выполнения оператора *Stmt1*. Антизависимость $u2$ от $v1$ указывает, что оператор *Stmt3* должен исполняться только после исполнения *Stmt2*. Также в данном фрагменте имеется выходная зависимость $u2$ от $u1$, показывающая, что перезаписывается одна и та же ячейка памяти A.

Анти- и выходные зависимости препятствуют конвейеризации программы. В данном примере от этих зависимостей можно избавиться, если заменить гене-

ратор *u*1 переменной *A* на генератор новой переменной *AA* (естественно, новая переменная *AA* должна быть описана). Те вхождения, которые использовали значения, записанные генератором *u*1, должны быть заменены на использования переменной *AA*. После описанной процедуры получим следующий фрагмент программы, эквивалентный исходному:

```
 AA = B + C; //Stmt1
u1
 B = AA + 2; //Stmt2
     v1
 A = AA * 3; //Stmt3
u2   v2
 F = A * A + D; //Stmt4
     v3  v4
```

В полученном фрагменте каждая переменная получает свое значение только один раз, а также отсутствуют антизависимости. Кроме этого, оператор *Stmt3* может быть исполнен сразу после оператора *Stmt1*, в отличие от исходной программы.

Конец примера 48.

Пример 49. (Переименование скалярной переменной в цикле или экспансия скалярной переменной.)

Рассмотрим цикл:

```
 for(i=0;i<N;i++)

     x = x + y[i];   //Stmt
     u    v
```

Здесь на каждой итерации цикла происходит перезапись одной и той же ячейки *x*. Чтобы избавиться от выходной и антизависимости нужно сделать так, чтобы на каждой итерации цикла оператор *Stmt* производил запись в разные переменные. Заменим генератор переменной *x* на генератор одномерного массива

xx[*i*]. Тогда, на каждой итерации цикла *i′* оператор *Stmt* будет производить запись в отдельную ячейку памяти *xx*[*i′*]. Такую замену можно рассматривать как переименование каждого генератора *x* на конкретной итерации *i′* в генератор *xx*[*i′*]. После необходимой модификации использований переменной *x*, получим следующий фрагмент программы, эквивалентный исходному:

```
xx[0] = x + y[0];

for(i=1;i<N;i++)

    xx[i] = xx[i-1] + y[i];
```

В данном примере генератор скалярной переменной был заменен генератором массива. В литературе такое преобразование имеет специальное название – растягивание скаляров [45, 57, 65, 69].

Конец примера 49.

Пусть во фрагменте программы требуется выполнить экспансию массива *x* для генератора *u*. Для упрощения изложения будем считать, что рассматриваемый фрагмент содержит все использования переменной *x* в программе и не содержит условных операторов. Для выполнения требуемого преобразования можно воспользоваться следующим алгоритмом.

Алгоритм экспансии массива.

1. Для каждого использования *v* переменной *x* строим соответствующий простой решетчатый граф потоковой зависимости по значению *G*.

2. Для каждого использования *v* удалим в соответствующем решетчатом графе *G* все функции, не описывающие зависимость *v* от *u*. Если после этого в описании графа *G* не останется ни одной функции, то вхождение *v* не использует значений, записанных генератором *u*; удалим его из списка рассматриваемых использований.

3. Заменяем генератор u переменной x на генератор $xx(\boldsymbol{I})$ новой переменной, где $\boldsymbol{I}=(I_1, I_2, \ldots, I_n)$ – вектор счетчиков опорного гнезда циклов генератора u; размерность переменной xx равна n; величина массива xx по j-ой (j=1..n) размерности равна максимальному значению, которое принимает счетчик j-го цикла – I_j.

4. Для каждого набора использований v_i (i=1..k) переменной x, имеющих одинаковое опорное гнездо циклов LN, выполним:

а) пусть дуги решетчатого графа G_i, соответствующего использованию v_i, описываются функциями $F^{v_i}{}_t$ (t=1..s_i), области определения которых есть выпуклые множества $H^{v_i}{}_t$;

б) с помощью разбиения множеств $H^{v_i}{}_t$ (i=1..k, t=1..s_i) построим новый набор множеств N_j (j=1..p) такой, что для любых $j1$ и $j2$, $j1 \neq j2$, множества N_{j1} и N_{j2} либо не пересекаются, либо совпадают;

в) производим расщепление опорного гнезда циклов LN в соответствии с разбиением $\{N_1, \ldots, N_p, V \setminus (N_1 \cup \ldots \cup N_p)\}$, где V – пространство итераций гнезда LN;

г) в каждом полученном гнезде, сканирующем N_j (j=1..p), выполняем: для всех i=1..k, если при некотором t (t=1..s_i) $N_j \subset H^{v_i}{}_t$, то заменяем вхождение v_i на вхождение $xx(F^{v_i}{}_t(\boldsymbol{I}))$;

Пример 50. (Демонстрируется экспансия массива, без необходимости дополнительного разбиения областей определения функций, описывающих дуги решетчатых графов (пункт 4.б алгоритма экспансии массива).)

Рассмотрим фрагмент программы умножения многочленов (см. пример 5). Как отмечалось в примере 5, генератор $u2$ на разных итерациях перезаписывает одни и те же ячейки памяти (выходная самозависимость). Также, в двойном цикле данного примера имеется антизависимость $u2$ от $v1$.

Выполним экспансию массива c в генераторе $u2$. В данной программе имеется только одно использование переменной c – использование $v1$. Простой решетчатый граф потоковой зависимости по значению $v1$ от $u1$, $u2$ описывается тройками ($u1$, $H^1_{u1,v1}$, $F_{u1,v1}$), ($u1$, $H^2_{u1,v1}$, $F_{u1,v1}$), ($u2$, $H_{u2,v1}$, $F_{u2,v1}$) (см. пример 25). В этом описании оставим только тройку ($u2$, $H_{u2,v1}$, $F_{u2,v1}$): остальные тройки ($u1$, $H^1_{u1,v1}$, $F_{u1,v1}$) и ($u1$, $H^2_{u1,v1}$, $F_{u1,v1}$) удалим т.к. они не описывают зависимость $v1$ от $u2$. Заменим генератор $c[i+j]$ на $cc[i][j]$. Выполним расщепление опорного гнезда для вхождения $v1$, пространство итераций которого V, в соответствии с разбиением $\{H_{u2,v1}, V\backslash H_{u2,v1}\}$. Т.к. множество $V\backslash H_{u2,v1}$ не является выпуклым, то расщеплять будем в соответствии с разбиением $\{H_{u2,v1}, H^1_{u1,v1}, H^2_{u1,v1}\}$, где все множества являются выпуклыми. В гнезде циклов, которое после расщепления сканирует $H_{u2,v1}$, заменим использование $c[i+j]$ на использование $cc[i-1][j+1]$. После выполнения указанных действий получим следующую программу, эквивалентную исходной:

```
for(k=0; k<=(N+M); k++)
    c[k]=0;
for(i=0; i<=0; i++)          // цикл по H^1_{u1,v1}
    for(j=0; j<=N; j++)
      cc[i][j]=c[i+j]+a[i]*b[j];
for(i=1; i<=M; i++)          // цикл по H^2_{u1,v1}
    for(j=N; j<=N; j++)
      cc[i][j]=c[i+j]+a[i]*b[j];
for(i=1; i<=M; i++)          // цикл по H_{u2,v1}
    for(j=0; j<=(N-1); j++)
      cc[i][j]=cc[i-1][j+1]+a[i]*b[j];
```

В последней программе отсутствуют анти- и выходные зависимости, в отличие от исходной. Заметим, что полученный фрагмент программы может быть упрощен после раскрутки циклов [45], пространство итераций которых состоит из одной точки.

Конец примера 50.

Пример 51. (Демонстрируется экспансия массива на основе расщепления, выполняемого с помощью добавления в тело цикла условных операторов [71].)

С целью сравнения метода экспансии массивов, предлагаемого в данной работе, с методом экспансии массивов [69, 71], приведем результат экспансии массива *c* для генератора *u*2 во фрагменте программы примера 5, выполненного с помощью метода [69, 71].

```
for(k=0; k<=(N+M); k++)
    c[k]=0;
for(i=0; i<=M; i++)
    for(j=0; j<=N; j++)
    if(i<=0)
      cc[i][j]=c[i+j]+a[i]*b[j];
    else
      if(j>=N)
           cc[i][j]=c[i+j]+a[i]*b[j];
      else
           cc[i][j]=cc[i-1][j+1]+a[i]*b[j];
```

Конец примера 51.

Пример 52. (Демонстрируется экспансия массива, при которой необходимо дополнительно разбивать области определения функций, описывающих дуги решетчатых графов (пункт 4.б алгоритма экспансии массива).)

Рассмотрим фрагмент программы:

```
for(i=0; i<=N; i++)

    for(j=0; j<=N; j++)

      a[i][j]=a[i-1][j]+a[i][j-1]+a[i+1][j]+a[i][j+1];
      u          v1          v2          v3          v4
```

В данном примере имеются две антизависимости u от $v3$ и u от $v4$, препятствующие конвейеризации. Дуги решетчатого графа, описывающего потоковую зависимость по значению $v1$ от u, определяются функцией $F_{u,v1}(i,j)=(i-1,j)$. Область определения функции $F_{u,v1}(i,j)$ есть множество целых точек $H_{u,v1}$ области, заданной неравенствами: $1\le i\le N$, $0\le j\le N$. Дуги решетчатого графа, описывающего потоковую зависимость по значению $v2$ от u, определяются функцией $F_{u,v2}(i,j)=(i,j\text{-}1)$. Область определения функции $F_{u,v2}(i,j)$ есть множество целых точек $H_{u,v2}$ области, заданной неравенствами: $0\le i\le N$, $1\le j\le N$. Решетчатые графы потоковой зависимости по значению $v3$ от u и $v4$ от u не имеют дуг (таких зависимостей нет).

Выполним экспансию массива a для генератора u. Вхождения $v3$, $v4$ не используют значений, записанных генератором u. Поэтому вхождения $v3$, $v4$ исключим из рассмотрения. Заменим генератор $a[i][j]$ на $aa[i][j]$. Множества $H_{u,v1}$ и $H_{u,v2}$ пересекаются, поэтому построим набор множеств N_j (j=1..3) такой, что для любых $j1$ и $j2$, $j1\ne j2$, множества N_{j1} и N_{j2} либо не пересекаются, либо совпадают. Разрезая множества $H_{u,v1}$ и $H_{u,v2}$ плоскостями, описывающими их границы, можно получить следующие множества N_j (j=1..3):

- N_1 описывается неравенствами $1\le i\le N$, $0\le j\le 0$;
- N_2 описывается неравенствами $0\le i\le 0$, $1\le j\le N$;
- N_3 описывается неравенствами $1\le i\le N$, $1\le j\le N$;

Теперь выполним расщепление опорного гнезда для вхождений $v1$, $v2$ (т.е. исходного гнезда), в соответствии с разбиением $\{N_1, N_2, N_3, V\backslash(N_1\cup N_2\cup N_3)\}$, где

V – пространство итераций расщепляемого гнезда. Если в полученных гнездах выполнить замену использований согласно пункту 4.г алгоритма экспансии массива, то получим следующую программу, эквивалентную исходной:

```
for(i=0; i<=0; i++)     // цикл по V\(N1∪N2∪N3), в нем
    for(j=0; j<=0; j++) // использования не изменяются
    aa[i][j]=a[i-1][j]+a[i][j-1]+a[i+1][j]+a[i][j+1];
for(i=1; i<=N; i++)     // цикл по N1
    for(j=0; j<=0; j++)
    aa[i][j]=aa[i-1][j]+a[i][j-1]+a[i+1][j]+a[i][j+1];
for(i=0; i<=0; i++)     // цикл по N2
    for(j=1; j<=N; j++)
    aa[i][j]=a[i-1][j]+aa[i][j-1]+a[i+1][j]+a[i][j+1];
for(i=1; i<=N; i++)     // цикл по N3
    for(j=1; j<=N; j++)
    aa[i][j]=aa[i-1][j]+aa[i][j-1]+a[i+1][j]+a[i][j+1];
```

В последней программе отсутствуют антизависимости, препятствовавшие конвейеризации.

Заметим, что полученный фрагмент программы может быть упрощен после раскрутки циклов [45], пространство итераций которых состоит из одной точки.

Конец примера 52.

Пример 53. (Демонстрируется экспансия массива, в результате которой размерность новой переменной меньше, чем размерность переменной заменяемого генератора.)

Рассмотрим фрагмент программы:

```
for (i=0; i<=10; i=i+1)
    a[i][i] = a[i][i-1];


 for (i=0; i<=10; i=i+1)
     for (j=0; j<=10; j=j+1)
         b[i][i] = a[i][j];
```

После выполнения экспансии массива *a* в генераторе *a*[*i*][*i*] получим следующий фрагмент программы:

```
for (i=0; i<=10; i=i+1)
    aa[i] = a[i][i-1];


 for (i=0; i<=10; i=i+1)
     for (j=0; j<=(i-1); j=j+1)
         b[i][i] = a[i][j];


 for (i=0; i<=10; i=i+1)
     for (j=i; j<=i; j=j+1)
         b[i][i] = aa[i];


 for (i=0; i<=10; i=i+1)
     for (j=(i+1); j<=10; j=j+1)
         b[i][i] = a[i][j];
```

Конец примера 53.

3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ ПОСТРОЕНИЯ ГРАФОВЫХ МОДЕЛЕЙ ИНФОРМАЦИОННЫХ ЗАВИСИМОСТЕЙ И ПРЕОБРАЗОВАНИЙ ПРОГРАММ, ИХ ИСПОЛЬЗУЮЩИХ

Автор работы является одним из разработчиков Открытой распараллеливающей системы. Разработка данной системы ведется с использованием принцыпов объектно-ориентированного программирования на языке C++.

Предполагается, что ОРС будет использоваться:

- как основа для разработки автоматических распараллеливающих компиляторов для параллельных компьютеров различных архитектур;
- как основа для разработки системы полуавтоматического и диалогового распараллеливания;
- как система для исследования параллелизма в последовательных программах;
- как обучающая система по параллельным вычислениям.

О концепциях и структуре ОРС можно прочитать в [48, 49, 97].

К моменту написания данной работы, автором реализована следующая функциональность ОРС:

1. построение графа информационных зависимостей для фрагмента программы на основе неравенств Банержи и НОД теста;
2. построение элементарных и простых снизу решетчатых графов;
3. построение графа информационных зависимостей на основе решетчатых графов;
4. разметка циклов ParDo по решетчатым графам;
5. расщепление многомерных гнезд циклов в виде последовательности тесных гнезд;

6. подстановка индексных переменных, на основе расщепления в виде последовательности тесных гнезд циклов;
7. экспансия массивов, на основе расщепления в виде последовательности тесных гнезд циклов.

Данная функциональность обеспечивается кодом, размер которого составляет около 13 тысяч строк (или 370 кБайт исходных текстов).

В этой главе будут описаны основные алгоритмы, структуры данных и решения, лежащие в основе программной реализации указанной функциональности ОРС. Все созданные автором программные модули используют или модифицируют внутреннее представление программы в ОРС. Диаграмма основных классов, реализующих графовые модели информационных зависимостей в ОРС, приведена в приложении В.

3.1 Анализ информационных зависимостей в OPC

3.1.1 Построение расширенной информации о вхождениях

При построении графа информационных зависимостей нужно находить коэффициенты при счетчиках циклов в индексных выражениях вхождений переменных, в границах циклов, в условных операторах. Причем, в процессе построения, приходится неоднократно вызывать процедуру определения зависимости для одних и тех же вхождений, которая использует указанные коэффициенты. Между тем, внутреннее представление программы в OPC имеет сложную древовидную структуру [34, 52]. Поэтому несколько раз получать одни и те же данные из внутреннего представления неэффективно, тем более что иногда в выражениях не приведены подобные относительно счетчиков и т.п. В связи с этим, автором были использованы идеи работы [33] для реализации построения расширенной информации о вхождениях. Эта информация о вхождениях оказалась удобна для использования и при построении решетчатых графов.

Расширенная информация о вхождениях представляет собой контейнер (класс `IndOccurContainer` (Indexed Occurrence Container)), хранящий информацию о вхождениях переменных фрагмента программы в удобном виде. В контейнере `IndOccurContainer` отдельно хранится список использований и список генераторов. Такой способ хранения вхождений удобен на практике: часто приходится отдельно работать с генераторами, отдельно с использованиями, а в этом случае все вхождения уже разобраны по типу и дополнительных вычислений не требуется.

Расширенная информация о вхождениях строиться при вызове функции `BuildVO`, которая заполняет список генераторов и список использований контейнера `IndOccurContainer`. Одним из параметров функции `BuildVO` является ссылка на объект класса `id`. Этот объект определяет фрагмент программы, для

которого нужно строить расширенную информацию о вхождениях. В процессе построения последовательно по тексту программы перебираются все операторы требуемого фрагмента. При рассмотрении очередного оператора присваивания из него выделяются вхождения переменных и заносятся в соответствующие списки. При этом вхождения переменных в индексные выражения массивов на данном этапе не выделяются и не помещаются в указанные контейнеры[23].

При построении расширенной информации о вхождениях для фрагмента программы, все вхождения занумеровываются. Они нумеруются слева направо и сверху вниз, начиная с нуля.

Пример 54. Фрагмент программы, с занумерованными вхождениями.

```
for(i=1; i<=M; i++)

{   for(j=1; j<=N; j++)

    { a[j]=x[i-1][j+1]+y;
      0     1             2
      x[i][j]=a[j]*b[i];
      3        4     5
    }

     b[i]=a[i];
     6     7
}
```

Конец примера 54.

Такая нумерация вхождений позволяет эффективно решать следующие задачи:

---

[23] на момент реализации разработчикам ОРС не требовался разбор операторов на вхождения в строгом соответствии с определением вхождения.

- Для двух вхождений определить, какой из операторов, содержащих эти вхождения, расположен раньше по тексту программы (функции `PrecedesStrictly`, `PrecedesNonStrictly`).
- Последовательно перебрать все вхождения в блоке операторов, содержащемся в исходном фрагменте. Для этого можно получить номер первого и последнего вхождения в блоке – функция `IndOccurContainer::GetFirstAndLastBlockOccurNumb`. Искомый набор вхождений определяется всеми номерами между двумя найденными.

Все вхождения в контейнере `IndOccurContainer` индексированы. По номеру вхождения можно получить его описание из контейнера.

Теперь рассмотрим основную информацию о вхождении, которая содержится в элементах контейнеров. Каждый элемент контейнера описывается структурой `OccurDesc` (Occurrence Description):

```
struct OccurDesc:public Status
{
	int occurNumb;//номер вхождения (Occurence's Number).
	int loopNumb;// количество циклов в опорном гнезде для данного вхождения
	int dim; //размерность переменной
	int** data;// индексное выражение
	int *ub,*lb;		//ub верхние границы размерностей переменной,lb - нижние
	NameIter nameIter;//ссылка на имя переменной во внутреннем представлении
	Statement* pStmt;//Указатель на оператор во внутреннем представлении, ко-
торому принадлежит данное вхождение
	ExprData* pOccur;//Указатель на данное вхождение во внутреннем представле-
нии
	LoopDesc* loops; //Массив, описывающий операторы циклов опорного гнезда.
Объект loops[i] описывает i-й по вложенности оператор цикла.

	Context* suppPolyhedron;// Опорный многогранник для вхождения
	...
};
```

Рис. 16. Данные члены структуры OccurDesc.

Динамический массив `OccurDesc::data` содержит коэффициенты в индексном выражении при счетчиках циклов опорного гнезда. Этот массив имеет `dim` строк и `loopNumb+1` столбцов. Число `data[i][j]` это коэффициент при счетчике `j`–го цикла в индексном выражении, соответствующем `i`–ой размерности вхож-

дения-массива. При этом число `data[i][loopNumb+1]` это свободный член в индексном выражении, соответствующем `i`–ой размерности вхождения-массива.

Пример 55. Для вхождения номер 1 программы примера 54, массив будет иметь следующий вид.

При счетчике «*i*»	При счетчике «*j*»	Свободный член
1	0	-1
0	1	1

Конец примера 55.

Каждый объект типа `OccurDesc` имеет набор флагов, которые устанавливаются при заполнении объекта данными. В данный момент реализованы следующие флаги:

- `NONLINEAR` – указывает, что индексное выражение не линейно относительно счетчиков циклов;
- `FREE_IS_PARAMETER` – указывает, что свободный член в индексном выражении содержит параметры;
- `INDEX_COEF_IS_PARAMETER` – указывает, что индексное выражение содержит параметр при счетчике цикла;
- `IS_GENERATOR` – указывает, что вхождение является генератором (записывает в память);
- `EXAMINED` – указывает, что вхождение «просмотрено» (используется различными алгоритмами для пометки вхождения).

Работу с флагами обеспечивает класс `Status`. Благодаря механизму наследования, объекты типа `OccurDesc` имеют функцию-член `GetStatus` для проверки установленных флагов.

3.1.2 Реализация графа информационных зависимостей на основе неравенств Банержи

Для проверки существования зависимости вхождения *v* от вхождения *u*, был реализован НОД тест и неравенства Банержи, описанные в пункте 1.3.1. Эти тесты реализованы в функции `GetDepSupp`. С помощью нескольких последовательных вызовов `GetDepSupp`(*u*, *v*, ...), можно найти все носители зависимости *v* от *u*. Отметим, что для проверки существования зависимости вхождения *u* от вхождения *v* необходимо вызывать `GetDepSupp`(*v*, *u*, ...).

Граф информационных зависимостей, реализованный в ОРС, храниться в виде списка дуг. Поэтому, в процессе построения графа находятся его дуги (зависимости), и их описания добавляются в список.

Алгоритм построения графа информационных зависимостей на основе неравенств Банержи[24].

В данном алгоритме носители зависимости определяются с помощью функции `GetDepSupp`. Если у одного из тестируемых вхождений установлен хотя бы один из флагов `NONLINEAR`, `INDEX_COEF_IS_PARAMETER`, `FREE_IS_PARAMETER`, то считается, что между этими вхождениями существует зависимость со всеми возможными носителями.

1. В списке генераторов находим первое непросмотренное вхождение *u*. Если таких вхождений нет, то переход на пункт 8. Иначе пусть это вхождение переменной *NAME*.

2. Проверяем наличие самозависимости *u* от *u*. Если существует циклически порожденная зависимость, то добавляем дугу, характеризующую эту выходную самозависимость.

---

[24] основная идея этого алгоритма заимствована из работы [33].

3. Находим в списке генераторов очередное непросмотренное вхождение переменной *NAME*. Если таких нет, то перейти на п. 5, иначе пусть это вхождение *v*.

4. Если существует зависимость вхождения *v* от вхождения *u*, то добавляем дугу, характеризующую эту выходную зависимость. Если существует *циклически порожденная* зависимость вхождения *u* от вхождения *v*, то добавляем дугу, характеризующую эту выходную зависимость. Переход на пункт 3.

5. Находим в списке использований очередное непросмотренное вхождение переменной *NAME*. Если таких нет, то перейти на п. 7, иначе пусть это вхождение *v*.

6. Если существует зависимость *v* от *u*, то добавляем дугу, характеризующую потоковую зависимость. Если существует зависимость *u* от *v*, то добавляем дугу, характеризующую антизависимость. (В этом случае, при определении зависимостей, необходимо учитывать, какое из вхождений расположено раньше по тексту программы.) Пометим вхождение *v* как просмотренное. Перейти на п. 5.

7. Пометим вхождение *u* как просмотренное, а все вхождения в списке как непросмотренные. Переход на п. 1.

8. Если в параметрах построения графа указано, что нужно строить и дуги входной зависимости, то выполним действия, описанные в пунктах 1, 2, 3 данного алгоритма, но не для списка генераторов, а для списка использований.

9. Граф информационных зависимостей построен.

Рассмотрим структуры данных, которые используются для описания графа информационных зависимостей. Как уже указывалось, этот граф представляется в виде списка дуг, описывающих зависимости между вхождениями. Отдельная дуга графа представляется структурой `LampArrow`. Приведем ее описание.

```
struct LampArrow
{
private:
      std::list<int> suppList; // список носителей данной зависимости, носители
нумеруются с 0 (а не с 1, как в определении)
public:

      EN_DepType  type; // тип зависимости: FLOWDEP, ANTIDEP, OUTPUTDEP, INPUTDEP

      int srcOccurNumb, depOccurNumb; // номера зависимых вхождений такие, что
вхождение № depOccurNumb зависит от вхождения № srcOccurNumb

      int commonLoopNumb;      // количество циклов, которые одновременно охваты-
ваю два зависимых вхождения. Носитель всегда строго меньше этого числа.

      Statement* pSrcStmt,*pDepStmt; // указатели на операторы во внутреннем
представлении. Оператор pSrcStmt содержит вхождение № srcOccurNumb, оператор
pDepStmt содержит вхождение № depOccurNumb.

      ExprData* pSrcOccur,*pDepOccur; // указатели на соотв. вхождения во внут-
реннем представлении. Вхождение pSrcOccur имеет номер № srcOccurNumb, вхождение
pDepOccur имеет номер № depOccurNumb.

      int TestSupp(int _supp); // Функция проверки носителя зависимости данной
дуги. Возвращает не 0, если среди носителей имеется носитель равный _supp.

      int IsLoopIndependent(); // Возвращает не 0, если зависимость - циклически
независимая.

      int IsLoopIndependentOnly();   // Возвращает не 0, если зависимость - цик-
лически независимая и других носителей у данной зависимости НЕТ.

      ...

   };
```

Рис. 17. Структура LampArrow.

Граф зависимостей представляется классом `LamportGraph`. Единственное данное-член этого класса – список дуг. Класс содержит набор функций, позволяющих решать следующие задачи:

- определить, существует ли дуга графа из вхождения v во вхождение u; если существует, то найти все носители данной зависимости;

- определить, зависит ли оператор *Stmt1* от оператора *Stmt2*[25];
- получить все дуги графа, идущие из генераторов в данное вхождение;
- получить все дуги графа, идущие из использований в данное вхождение;
- проверить, существуют ли дуги, идущие во вхождение v, отличные от некоторой дуги (u, v);
- проверить, существуют ли в графе дуги, описывающие циклически порожденные зависимости.

3.1.3 Реализация решетчатых графов. Выбор способа построения

В настоящий момент в ОРС реализовано построение элементарных и простых снизу решетчатых графов для линейных программ, не содержащих внешних переменных и условных операторов.

Построение элементарного снизу решетчатого графа реализовано на основе метода параметрического целочисленного программирования, разработанного П. Фотрье [70]. Таким образом, указанные графы строятся в соответствии с алгоритмом, описанным в пункте 1.3.2, а для нахождения целочисленного лексикографического максимума среди решений систем используется параметризованный двойственный симплекс метод [70] и параметризованный метод отсечений Гомори [70]. Однако построенное решение – функции описывающие дуги решетчатого графа – выписываются не в виде квазиаффинного дерева решений [70, 71], а в виде квазиаффинных функций и их областей определения.

---

[25] говорят, что оператор *Stmt1* зависит от оператора *Stmt2*, если существуют вхождения $u \in Stmt1$, $v \in Stmt2$ такие, что u зависит от v; для некоторых преобразований программ, таких как разбиение цикла [49], требуются именно такие проверки.

Выбор метода параметрического целочисленного программирования [70], по сравнению с методом использования матриц с L-свойством [21, 25], в качестве основы потроения решетчатых графов, обусловлен следующими причинами:

1. с помощью метода [70] можно строить решетчатые графы для более широкого класса программ, чем с помощью метода [21];

2. при необходимости, в методе [70], можно не использовать параметризированный метод отсечений, с помощью которого обеспечивается целочисленность решений; это позволит получать покрывающие функции, аналогично методу [21], и уменьшить время построения решетчатого графа;

3. на основе метода [70] разработаны алгоритмы генерации границ многомерных гнезд циклов [64], в отличие от [21].

Рассмотрим структуры данных, которые используются для описания решетчатых графов. Функция, описывающая дуги решетчатого графа, представляется структурой `ArrowFunc`.

```
struct ArrowFunc
{
      int** data;  // матрица, описывающая функцию.
      int dimi;    // количество строк матрицы data
      int dimj;    // количество столбцов матрицы data
      int altPolyNumb;   //номер альтернативного многогранника, который использо-
вался для построения функции
      GenArea areas; // область определения функции (многосвязная)
      NewParamList* newParamList;//ур-ния для добавленных параметров

      void GetSource(LoopIndex& source,LoopIndex& sink);//по вершине конца дуги
sink возвращает начало дуги source. Предполагается, что sink принадлежит О.О.Ф.
этой функции

      ...
};
```

Рис. 18. Структура, представляющая функцию, которая описывает дуги решетчатого графа.

Пример 56. Рассмотрим функцию $F_{u2,v1}(i, j)=(i-1, j+1)$, определенную на $H_{u2,v1}$, которая описывает дуги решетчатого графа (см. пример 21). Массив `data`

объекта типа `ArrowFunc`, описывающего эту функцию, будет иметь следующий вид:

	коэф. при «i»	*коэф. при «j»*	*своб. член*
строка/столбец	*0*	*1*	*2*
0	1	0	-1
1	0	1	1

Рассмотрим функцию, описывающую дуги решетчатого графа, изображенного на рисунке 7. В общем виде она записывается следующим образом: $F(i)=0*i+1*q+0$, где $q=(i+0)\div 2$. Таким образом, массив `data` объекта типа `ArrowFunc`, описывающего данную функцию, будет иметь вид:

	коэф. при «i»	*коэф. при «q»*	*своб. член*
строка/столбец	*0*	*1*	*2*
0	0	1	0

Равенство, определяющее q ($q=(i+0)\div 2$), будет описываться первым элементом списка `newParamList`.

Конец примера 56.

Элементарный снизу решетчатый граф представляется классом `ElemLatticeGraph`. Среди данных-членов этого класса есть ссылки на объекты класса `OccurDesc`, описывающие пару вхождений, зависимость между которыми описывает данный граф. Также, среди данных членов имеется список `std::list<ArrowFunc*>`, представляющий набор функций, описывающих дуги графа. Перечислим наиболее важные методы этого класса:

- `Build(OccurDesc& srcEntry,OccurDesc& depEntry)` - построить решетчатый граф, описывающий зависимость вхождения `depEntry` от вхождения `srcEntry`.

- `int SelfTest()` – функция проверки правильности построения графа. Функция полным перебором находит множество всех дуг решетчатого графа и проверяет совпадение этого множества с множеством дуг, описываемых построеннми функциями.
- `int TestLatticeBasedSupp(int supp)` – проверяет, существует ли в описании графа функция, при построении которой использовался альтернативный многогранник с номером `supp`.
- `int GetSource(LoopIndex& source,LoopIndex& sink)` – данная функция по вершине конца дуги `sink` возвращает вершину начала дуги `source`. Если в вершину `sink` не входит дуга графа, то функция возвращает 0.
- `Print(const char* fileName)` – вывод функций, описывающих дуги графа, в текстовый файл.

Простой снизу решетчатый граф представляется классом `LatticeGraph`. Среди данных-членов этого класса есть ссылка на объект класса `OccurDesc`, описывающий зависимое вхождение *v*. Также, среди данных членов имеется список `std::list<LatticeDataElem*>`. Каждый элемент этого списка содержит ссылку на некоторое вхождение *u* и набор функций, которые описывают зависимость *v* от *u*. Основные методы данного класса аналогичны рассмотренным методам класса `ElemLatticeGraph`.

Элементарный решетчатый граф в настоящий момент визуализирован в ОРС, для гнезд циклов глубиной 1 и 2. Простой решетчатый граф строиться только в преобразованиях, и в настоящий момент не визуализирован. Кроме того, в настоящий момент не реализовано построение простого снизу решетчатого графа, который не может быть описан функциями, удовлетворяющими условиям С.1) – С.3) (пункт 1.2.2.4).

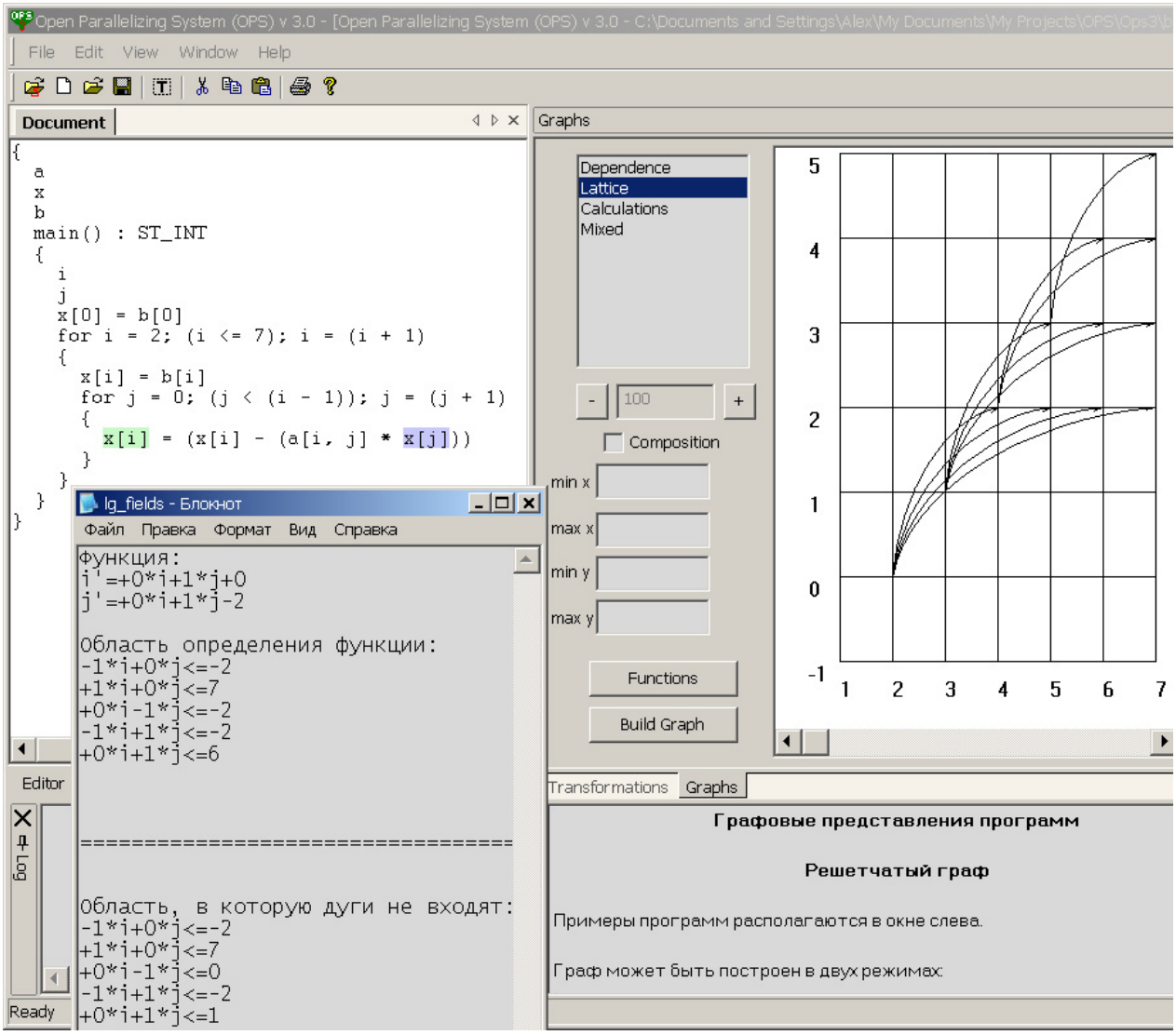


Рис. 19. Визуализация элементарного снизу решетчатого графа зависимости использования $x[j]$ от генератора $x[i]$ в ОРС.

3.1.3.1 Тестирование правильности построения решетчатого графа

Решетчатые графы лежат в основе всех методов, разработанных автором. Сложность реализации некоторых из них, таких как, например, подстановка индексных переменных и экспансия массивов, довольно велика. Кроме этого, на программные реализации данных методов ложится функция апробации теоретических результатов. Таким образом, не имея программных модулей, которые

гарантированно правильно строят решетчатые графы, нельзя приступать к программным реализациям полученных в работе методов (в противном случае, отладка займет недопустимое количество времени, из-за потенциально большого количества ошибок и мест их появления).

Ручная проверка правильности построения элементарных решетчатых графов во многих случаях является нетривиальной задачей, отнимающей значительные силы и время: нужно проверить правильность построения каждой дуги. Для гнезд циклов, размерности больше чем 2, и не совсем тривиальных индексных выражений зависимых вхождений, ручная проверка правильности построения элементарного решетчатого графа часто бывает очень сложной. Сложность проверки еще увеличивается, когда проверяются элементарные решетчатые графы, дуги которых описываются квазиаффинными функциями, и простые решетчатые графы.

Для решения обозначенных выше проблем, в ОРС были реализованы функции автоматического тестирования правильности построения решетчатых графов. Они реализованы как функции-члены (`SelfTest`) классов, описывающих решетчатые графы, и могут быть вызваны после того, как соответствующий граф построен (построены функции, описывающие дуги графа). Эти функции полным перебором вершин графа находят множество всех его дуг, и проверяют совпадение этого множества с множеством дуг, описываемых построенными функциями. Функции `SelfTest` способны тестировать элементарные решетчатые графы, дуги которых описываются квазиаффинными функциями, и простые решетчатые графы.

3.1.3.2 Экспериментальные данные о времени построения решетчатых графов

Для того чтобы оценить, насколько использование решетчатых графов может замедлить процесс компиляции программы, был проведен эксперимент. Большинство программ, подобранных для осуществления этого эксперимента, реализуют практически используемые алгоритмы, такие как метод Жордана решения системы линейных алгебраических уравнений [21, с. 390], метод исключения Гаусса (упрощенный), некоторый вариант LU разложения, умножение многочленов и др. В ходе эксперимента для каждой программы выполнялся замер времени построения минимальных снизу решетчатых графов всех типов зависимости (т.е. всевозможных простых снизу решетчатых графов). Также, выполнялся замер времени построения всевозможных элементарных снизу решетчатых графов. И в том и в другом случаях, отдельно измерено время построениях всевозможных решетчатых графов, без учета тех, которые описывают входную зависимость. Измерение времени производилось на персональном компьютере с процессором Intel Pentium-4 530. Стоит отметить, что при реализации методов построения решетчатых графов автор работы не занимался оптимизацией по времени. Поэтому, время построения этих графов в ОРС может быть уменьшено.

Полученные в ходе эксперимента измерения, а также тексты рассматриваемых программ, приведены в приложении Б. Отметим, что время построения графа информационных зависимостей с помощью неравенств Банержи и НОД теста во всех рассматриваемых программах неразличимо для стандартных функций С измерения времени и считается равным нулю.

По мнению автора работы, несмотря на то, что время построения всех простых снизу решетчатых графов значительно превышает время построения графа информационных зависимостей на основе неравенств Банержи и НОД теста, оно является приемлемым для использования в компиляторах.

3.1.4 Реализация построения графа информационных зависимостей на основе решетчатых графов

Важной частью построения графа информационных зависимостей является метод, с помощью которого определяются зависимые вхождения. В пункте 3.1.2 описывался метод построения графа на основе неравенств Банержи и НОД теста. Такое построение графа информационных зависимостей было реализовано в ОРС первым. Однако, несмотря на все достоинства тестов Банержи и НОД, они не точны (см. 1.3.4.1). Поэтому, в настоящий момент в ОРС реализовано построение графа зависимостей на основе элементарных снизу решетчатых графов и простых снизу решетчатых графов. Реализация выполнена с помощью, так называемого, уточнения графа информационных зависимостей. Сначала этот граф строится на основе тестов Банержи и НОД. Затем дуги построенного графа перепроверяются с помощью других методов анализа зависимостей. В результате, некоторые дуги или носители у некоторых дуг могут быть удалены. Такой подход представляется удобным т.к. тесты Банержи и НОД есть эффективный способ доказательства либо информационной независимости, либо отсутствия некоторого носителя.

Алгоритм построения графа информационных зависимостей на основе элементарных снизу решетчатых графов.

1. Строим граф информационных зависимостей на основе неравенств Банержи и НОД теста.

2. Производим уточнение построенного графа:

для каждой дуги графа (u, v) выполним:

а) строим элементарный решетчатый граф G, описывающий зависимость v от u;

б) если граф G не имеет дуг, то удалим дугу (u, v) из графа зависимостей;

в) иначе заменяем список носителей дуги (u, v) на список номеров альтернативных многогранников, которые использовались при построении графа G.

Пример 57. На нижеследующих рисунках приведен пример графа информационных зависимостей, построенный с помощью тестов Банержи и НОД и с помощью элементарных снизу решетчатых графов.

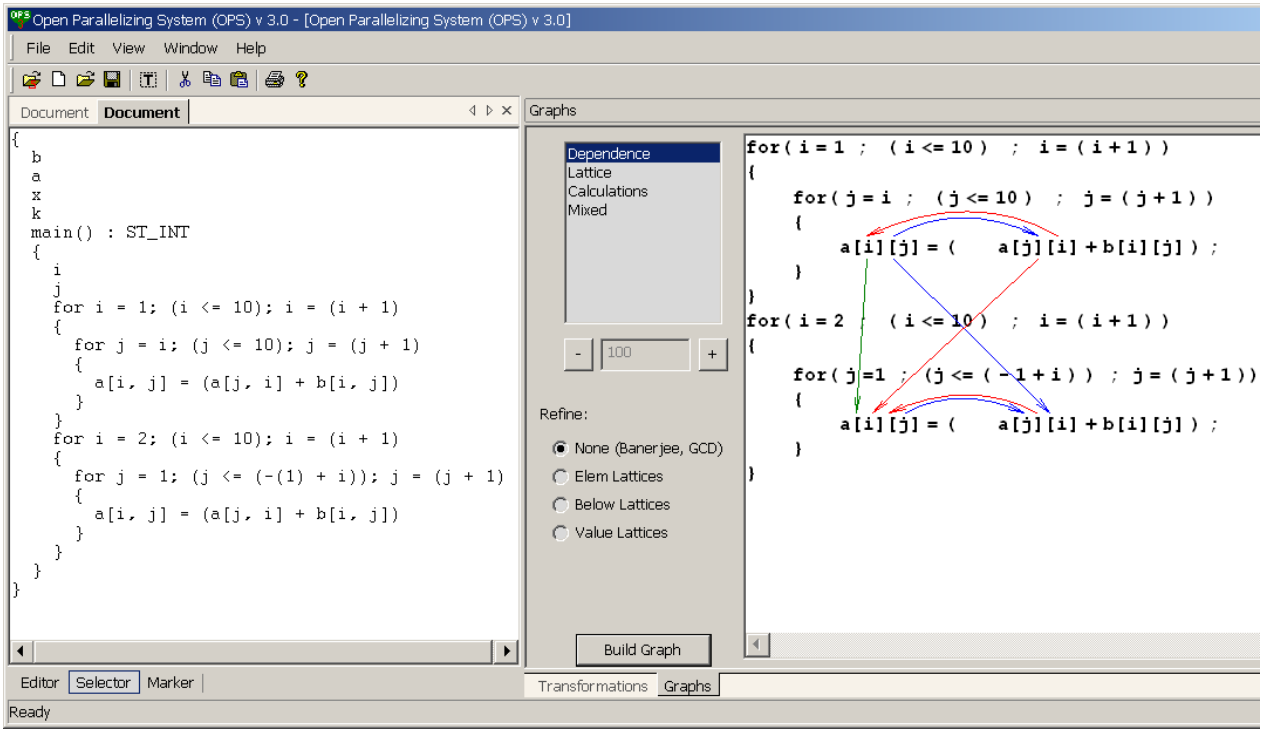


Рис. 20. Граф информационных зависимостей, построенный с помощью неравенств Банержи и НОД теста.

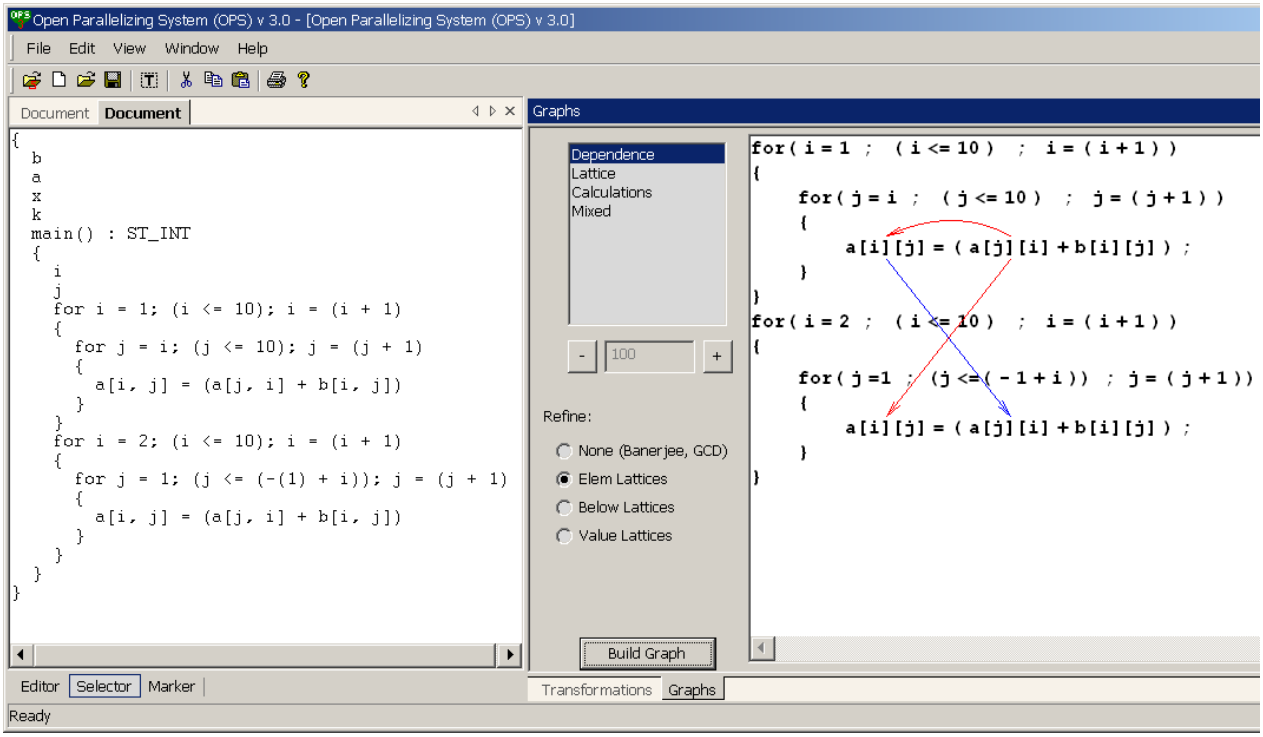


Рис. 21. Граф информационных зависимостей, построенный с помощью элементарных снизу решетчатых графов.

Конец примера 57.

Алгоритм построения графа информационных зависимостей на основе простых снизу решетчатых графов.

1. Строим граф информационных зависимостей на основе неравенств Банержи и НОД теста.

2. Производим уточнение построенного графа:

для каждого вхождения v, в которое входят дуги, выполним:

а) найдем множество генераторов u_i (i=1..n), из которых идут дуги в вхождение v;

б) построим простой снизу решетчатый граф G, описывающий зависимость v от u_i (i=1..n);

в) для каждой пары (u_i, v) в графе G находим набор функций, описывающих зависимость v от u_i, с помощью которых модифицируем носители дуги (u_i, v) и принимаем решение о ее дальнейшем существовании в графе;

г) выполняем пункты а), б), в) данного алгоритма для всех использований, из которых идут дуги в вхождение v.

Пример 58. На нижеследующих рисунках приведен пример графа информационных зависимостей, построенный с помощью тестов Банержи и НОД и с помощью простых снизу решетчатых графов.

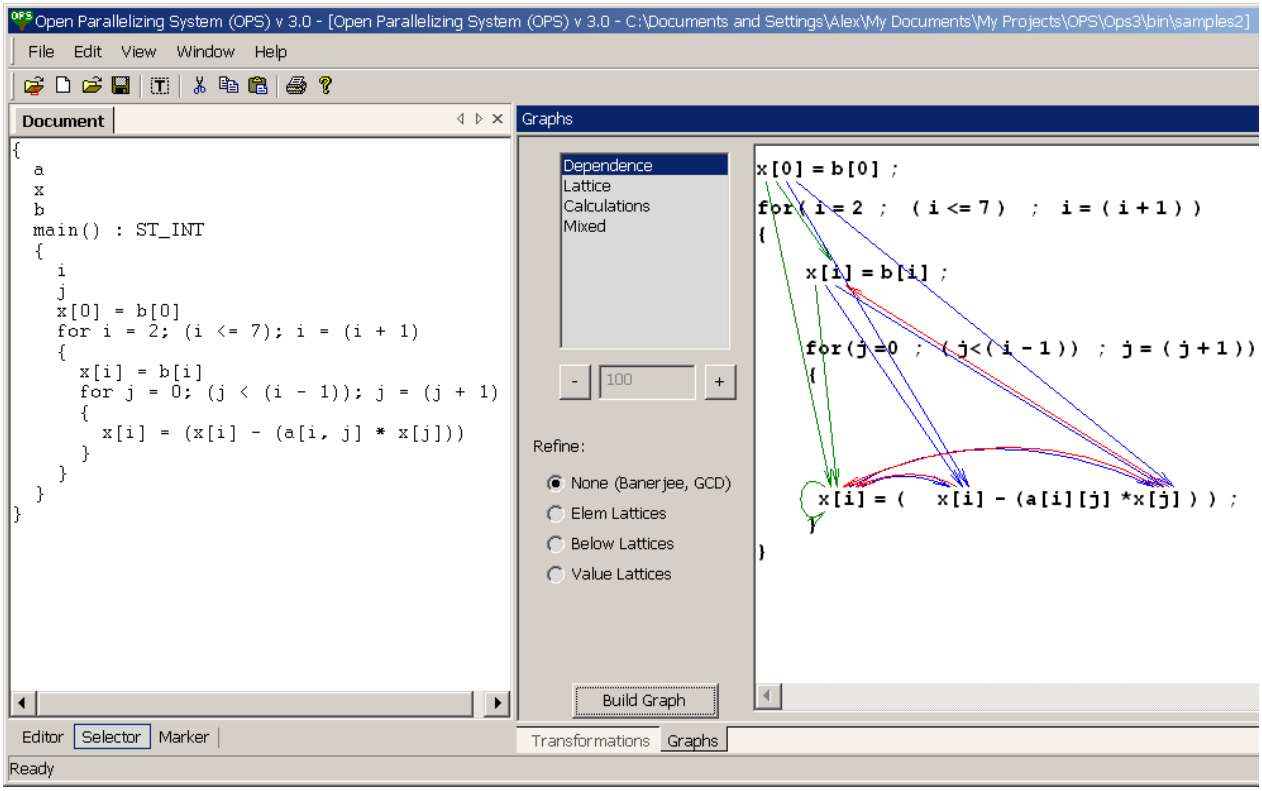


Рис. 22. Граф информационных зависимостей, построенный с помощью неравенств Банержи и НОД теста. Граф содержит дугу ложной потоковой зависимости использования $x[j]$ от генератора $x[i]$ оператора $x[i]=b[i]$.

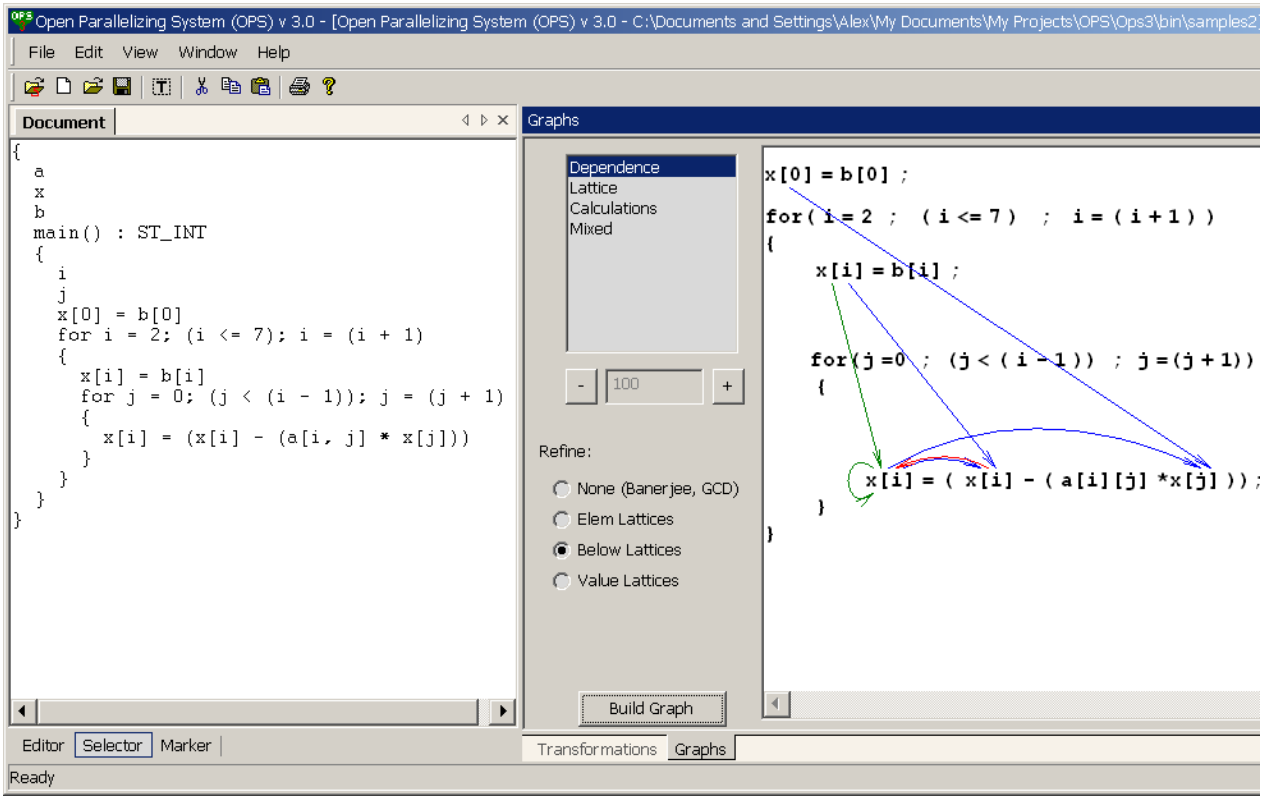


Рис. 23. Граф информационных зависимостей, построенный с помощью простых снизу решетчатых графов. Все дуги потоковой зависимости в графе есть дуги зависимости *по значению*. (Дуга потоковой зависимости использования $x[j]$ от генератора $x[i]$ оператора $x[i]=b[i]$ удалена.)

Конец примера 58.

3.1.5 Реализация разметки ParDo циклов в программе

В настоящий момент в OPC реализована разметка ParDo циклов на основе неравенств Банержи и НОД теста. Кроме этого, реализована разметка ParDo циклов по элементарным и простым снизу решетчатым графам.

Все эти различные способы разметки реализованы единым образом: одной функцией. Это возможно благодаря тому, что носитель зависимости влияет на свойство ParDo цикла, и совпадает с номером некоторого альтернативного многогранника (см. 2.1.2).

Алгоритм разметки ParDo циклов в программе.

1. Строим граф информационных зависимостей без учета входных зависимостей. Для построения используем один из следующих методов (в зависимости от того, как нужно разметить циклы):

- неравенства Банержи и НОД тест;
- элементарные снизу решетчатые графы;
- простые снизу решетчатые графы.

2. Для каждой дуги (u, v) построенного графа, описывающего зависимость типа T, выполним:

- если у данной зависимости существует носитель s, то в гнезде циклов, содержащих и вхождение u и вхождение v, пометим цикл на глубине вложенности s+1 как НЕ ParDo цикл по зависимости типа T.

3. Конец алгоритма.

Пусть в данном алгоритме граф информационных зависимостей построен с помощью неравенств Банержи и НОД теста. Тогда циклы, итерации которых исполнять одновременно нельзя, будут помечены как НЕ ParDo циклы по некоторым зависимостям. Этот классический метод имеет все недостатки, присущие неравенствам Банержи и НОД тесту.

Пусть в данном алгоритме граф информационных зависимостей построен либо с помощью элементарных, либо с помощью простых снизу решетчатых графов. Тогда в результате работы алгоритма будут помечены циклы, не являющиеся циклами ParDo *по элементарным* либо *простым снизу решетчатым графам* некоторой зависимости. Такой метод позволяет определять циклы, итерации которых можно выполнять одновременно, только в случае совпадения минимального снизу и сверху графов антизависимости.

В ОРС необходимо реализовать минимальные сверху решетчатые графы. Тогда можно будет строить граф информационных зависимостей на основе решетчатых графов, описывающих зависимость *по значению*. После добавления

соответствующего варианта построения графа информационных зависимостей в функцию разметки ParDo циклов, будет реализован метод определения циклов, итерации которых можно исполнять одновременно, основанный на решетчатых графах (без дополнительных ограничений).

3.2 Преобразования программ, основанные на решетчатом графе, в ОРС

Автором работы были реализованы три преобразования программ, основанные на решетчатых графах. Однако эти преобразования реализованы только в экспериментальном варианте: в некоторых из случаев, когда минимальный снизу и сверху графы антизависимости не совпадают, они могут привести к неэквивалентной программе. Далее будут описаны эти реализации.

3.2.1 Реализация расщепления гнезда циклов. Выбор метода генерации границ циклов

В настоящий момент в ОРС реализовано расщепление тесных гнезд циклов в виде последовательности тесных циклов.

Алгоритм расщепления тесного гнезда, реализованный в ОРС.

На входе алгоритма тесное гнездо, которое нужно расщепить, и список выпуклых областей, которые образуют разбиение итерационного пространства исходного гнезда.

1. Области в списке сортируются так, чтобы для любых двух областей N_1, N_2 и любого элементарного снизу решетчатого графа G выполнялось: если граф содержит дугу, идущую из N_1 в N_2, то N_2 находится ближе к концу списка, чем N_1.

2. Если такая сортировка невозможна, то алгоритм завершается без выполнения расщепления.

3. В соответствии с построенным списком генерируем описание требуемой последовательности тесных гнезд. Эта структура описывается деревом, узлы которого имеют тип `LoopBoundsNode`. Для генерации границ циклов используется алгоритм [63].

4. Имея описание последовательности гнезд циклов, модифицируем внутреннее представление.

Для генерации границ циклов использовался алгоритм [63] вместо алгоритма [58], по следующим причинам:

1. как показано в работе [61], алгоритм [63] требует меньше памяти, чем алгоритм [58];

2. как показано в работе [61], при больших размерностях гнезд циклов алгоритм [63] выполняется быстрее, чем алгоритм [58];

3. алгоритмом [58] можно сгенерировать только сразу все границы всех циклов в гнезде; алгоритм [63] более удобен, им можно генерировать границы циклов отдельно для каждой глубины вложенности;

4. алгоритм [63] использует параметризованный двойственный симплекс метод, который был реализован в ОРС для построения решетчатых графов.

Также, в ОРС реализована функция, которая строит расщепление по разбиению, полученному из областей определения функций, описывающих дуги некоторого решетчатого графа. Это позволяет выполнять оптимизацию, описанную в 2.2.2.2.

Пример 59. На рисунке 24 приведен фрагмент программы в ОРС, непосредственно перед расщеплением. Расщепление будет выполнено в соответствии с разбиением пространства итераций, состоящем из области определения функции, описывающей дуги решетчатого графа зависимости $a[j][i]$ от $a[i][j]$ (эти вхождения выделены), и дополнения этой области до пространства итераций.

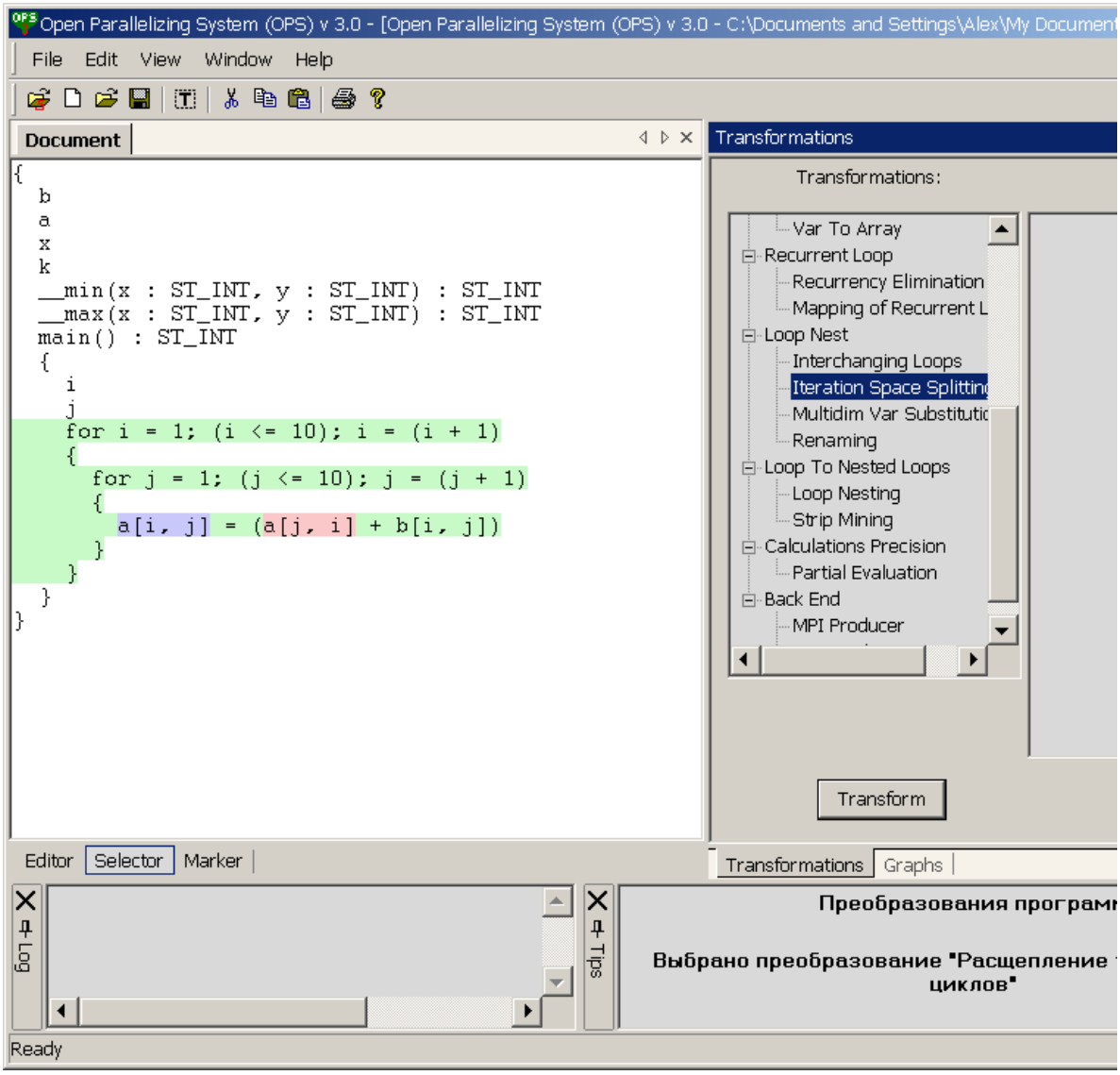


Рис. 24. ОРС непосредственно перед расщеплением.

После выполнения расщепление, будет получен фрагмент программы, изображенный на рисунке 25.

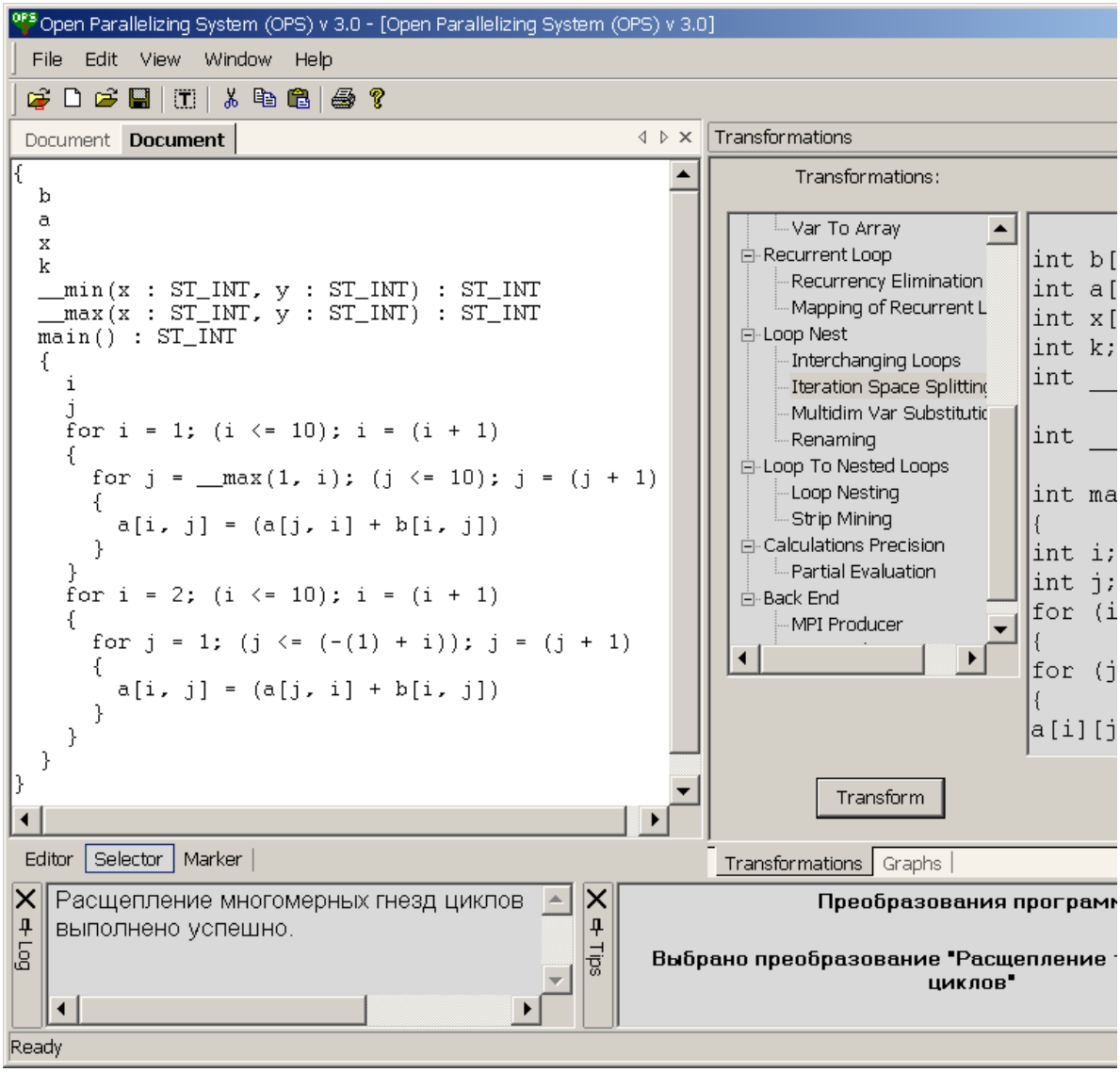


Рис. 25. OPC после выполнения расщепления и разбора полученного кода.

Отметим, что в OPC не было проделано никакой работы по оптимизации полученной в ходе расщепления структуры циклов (в полученной программе можно вызов «*__max*(1, *i*)» заменить на «*i*»). Методы нетривиальной оптимизации сгенерированных циклов даются в работе [62].

Конец примера 59.

3.2.2 Подстановка переменных и экспансия массивов

Программная реализация подстановки переменных и экспансии массивов выполнена в соответствии с алгоритмами, описанными в пунктах 2.3.1, 2.3.2. Стоит отметить, что сложность реализации экспансии массивов значительно превышает сложность реализации подстановки и расщепления. Экспансия массивов, которая применима для всех программ из линейного класса, была впервые разработана П. Фотрье более 15 лет назад. Несмотря на это, автору не известна распараллеливающая система, где это преобразование было бы запрограммировано. Даже в такой «богатой» оптимизирующими и распараллеливающими преобразованиями французской системе PIPS [98], где реализованы многие методы П. Фотрье, это преобразование еще не реализовано, о чем разработчики сообщают на странице [99].

Ниже приведены формы ОРС, демонстрирующие эти преобразования.

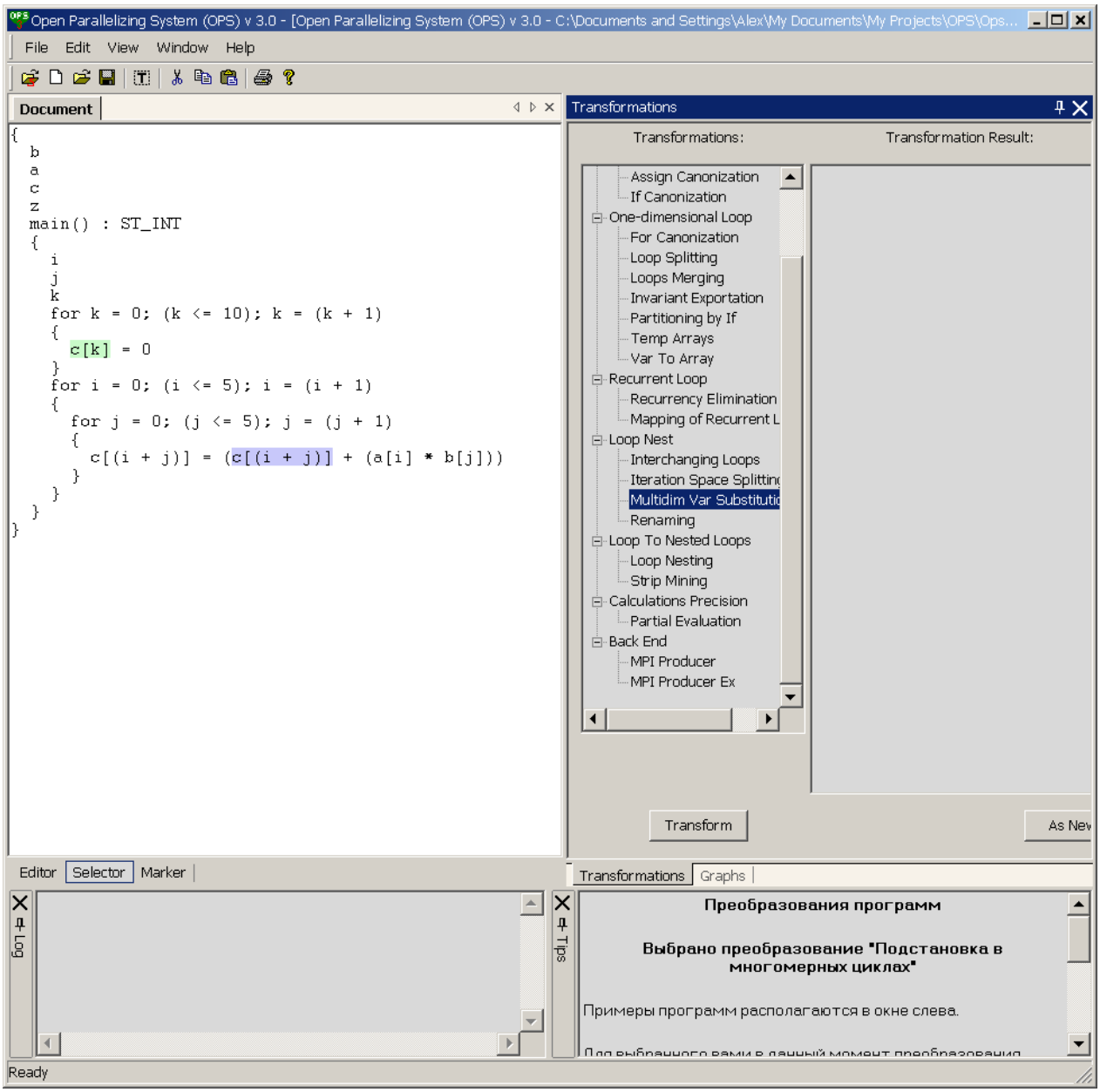


Рис. 26. ОРС перед подстановкой генератора $c[k]$ вместо использования $c[i+j]$ в программе умножения многочленов.

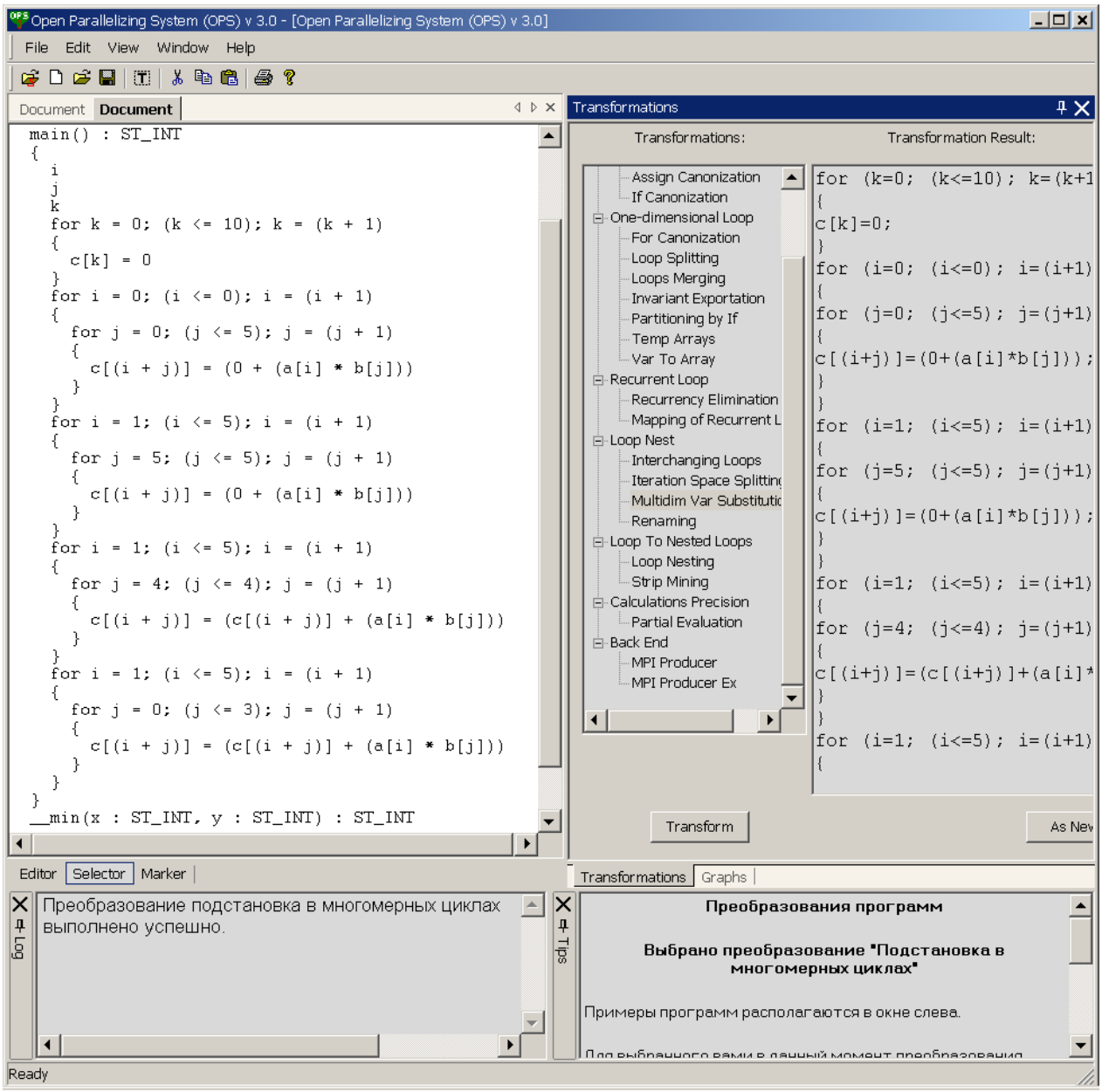


Рис. 27. ОРС после подстановки генератора $c[k]$ вместо использования $c[i+j]$ в программе умножения многочленов.

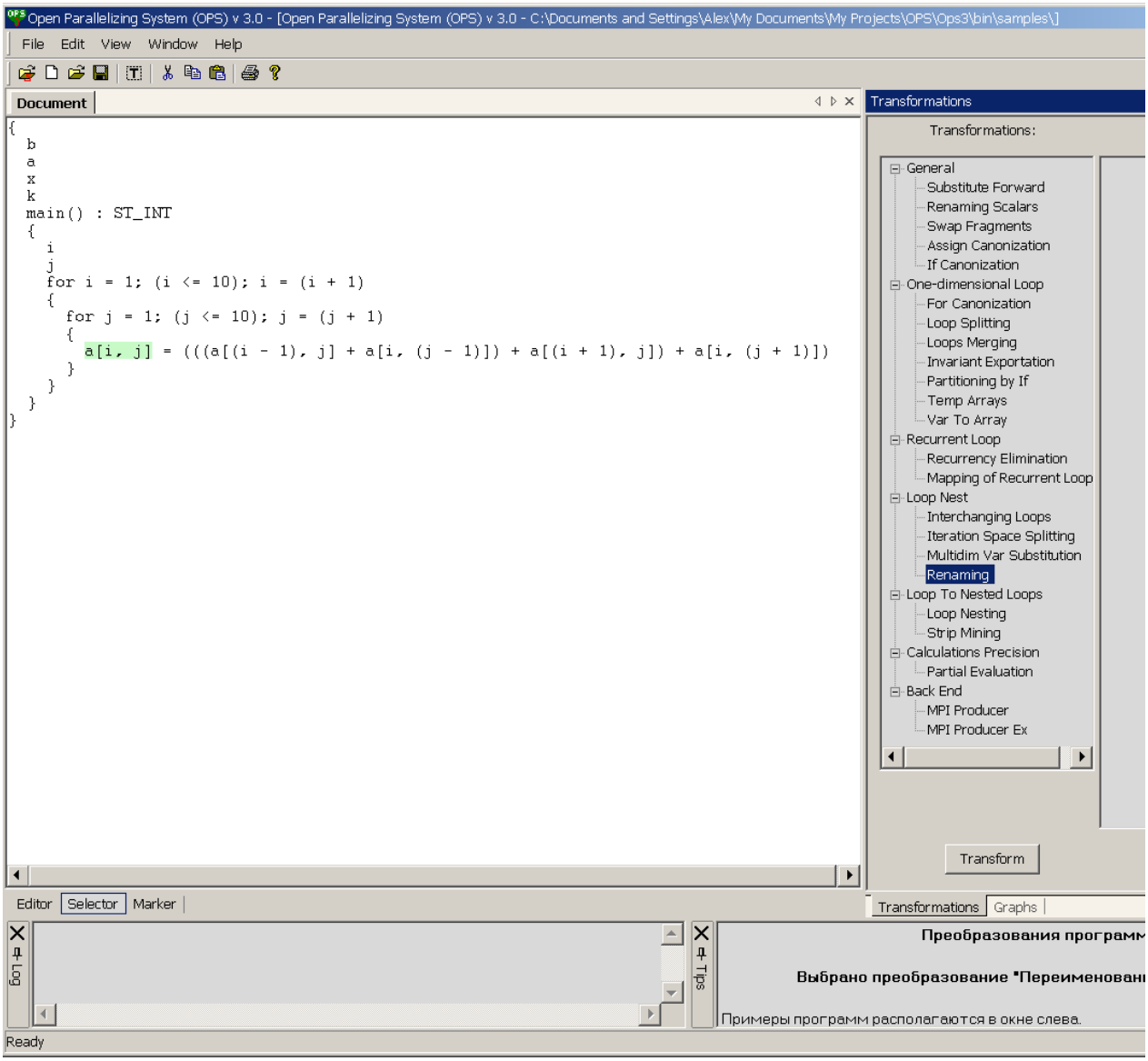


Рис. 28. ОРС перед экспансией массива *a* для генератора *a*[*i*][*j*].

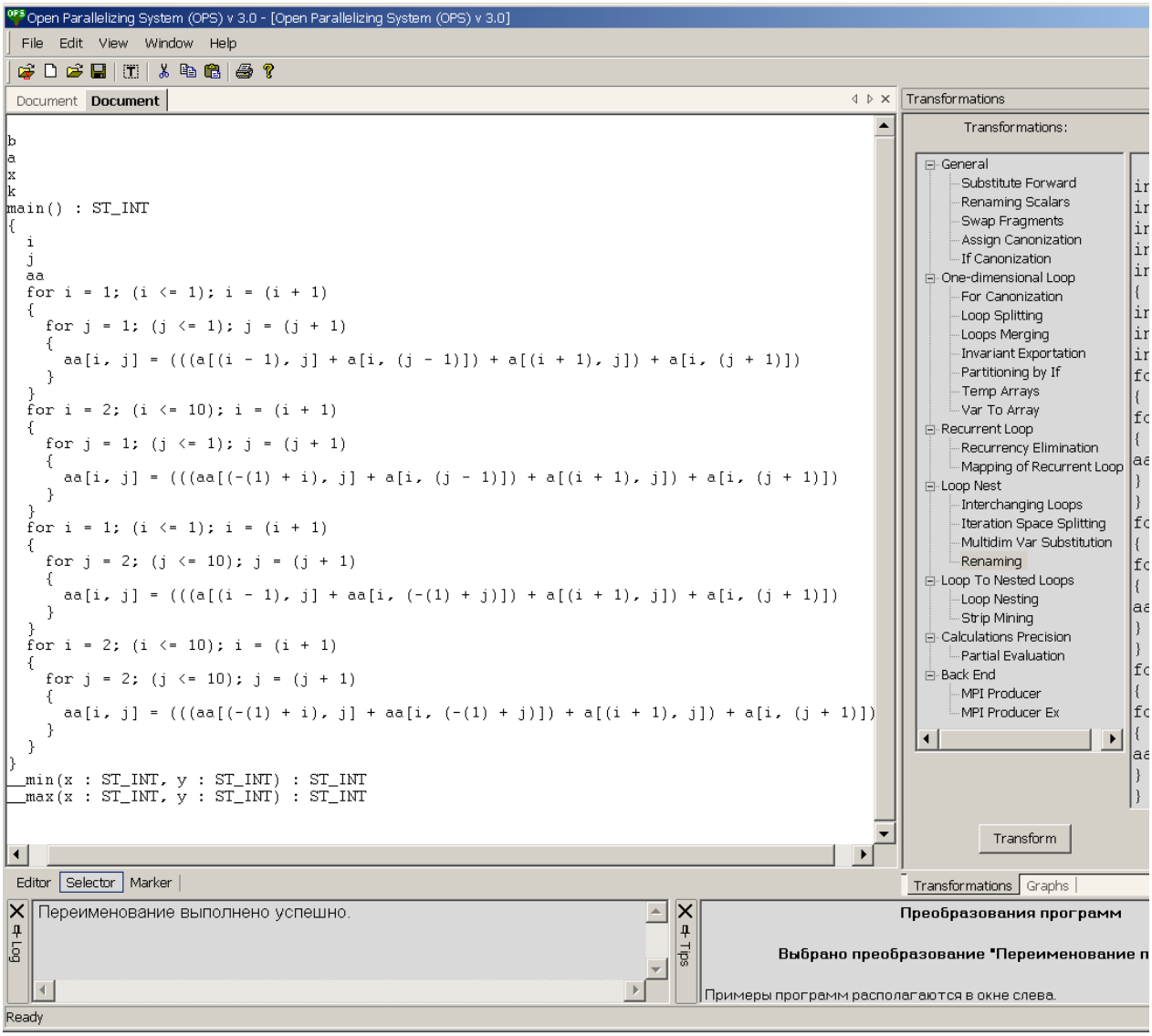


Рис. 29. ОРС после экспансии массива *a* для генератора *a*[*i*][*j*].

ЗАКЛЮЧЕНИЕ

В диссертационной работе выполнен сравнительный анализ известных методов, с помощью которых возможно построение решетчатых графов. В силу причин, изложенных в пункте 3.1.3, в качестве основы построения решетчатых графов в ОРС выбран метод П. Фотрье, который программно реализован автором работы.

Также в работе выполнено детальное сравнение решетчатого графа с наиболее часто используемыми представлениями информационной зависимости. Доказана связь существования функций, описывающих дуги решетчатого графа, с носителями информационной зависимости по значению. Указаны преимущества использования решетчатых графов перед традиционными представлениями информационной зависимости по памяти. Особое внимание уделено выявлению слабых мест методов построения представлений информационной зависимости по памяти – неравенств Банержи и НОД теста. Приведены примеры, в которых, пользуясь данными методами можно обнаружить не весь параллелизм программы, неправильно определить наличие зависимости, ошибочно запретить выполнение преобразования. Методы использования решетчатых графов не обладают указанными недостатками.

При изучении решетчатых графов были получены необходимые и достаточные условия цикла ParDo по решетчатому графу, применимые ко всем программам из линейного класса. Метод определения циклов, итерации которых можно выполнять одновременно, основанный на этих условиях, позволяет находить большее количество ParDo циклов, чем известные методы [8, 21, 95]. Кроме того, этот метод удобен для использования при автоматическом распараллеливании.

В диссертационной работе получены новые методы расщепления многомерных циклов. Они позволяют по заданному разбиению пространства итераций

строить расщепление как тесных, так и нетесных многомерных циклов. Показано, как полученные методы, вместе с необходимыми и достаточными условиями ParDo циклов по решетчатым графам, могут быть использованы для усовершенствования и последующей программной реализации метода распараллеливания на макроуровне [21, параграф 7.2]. Таким способом можно распараллеливать программы, которые невозможно распараллелить с помощью методов, основанных на традиционных представлениях информационной зависимости (вектор направления, носитель, расстояние зависимости). Полученные алгоритмы конкретизированы до возможности использования при автоматическом распараллеливании.

На основе разработанных алгоритмов расщепления многомерных циклов, в работе получены новые методы подстановки индексных переменных и экспансии массивов. Эти методы позволяют повысить эффективность распараллеливания высокоуровневых последовательных программ. Данные преобразования также невозможно реализовать, используя традиционные представления информационной зависимости.

Все полученные в диссертации методы, с некоторыми ограничениями, реализованы автором в OPC.

В ходе выполнения работы, был проделан численный эксперимент с целью изучения времени построения решетчатых графов для фрагментов программ. Результаты эксперимента показали, что время построения решетчатых графов является приемлемым для использования в распараллеливающих компиляторах.

Проведенная автором работа показывает, что современный компилятор может содержать методы анализа, основанные на теории решетчатых графов.

Из всего вышесказанного следует, что цель диссертационной работы автором достигнута.

Полученные в работе результаты могут быть использованы в автоматическом, полуавтоматическом и ручном режимах создания параллельных программ.

ЛИТЕРАТУРА

1. Абрамов С.М., Адамович А.И., Инюхин А.В., Московский А.А., Роганов В.А., Шевчук Ю.В., Шевчук Е.В. Т-Система с открытой архитектурой. // Тр. междунар. науч. конф. «Суперкомпьютерные системы и их применение» (SSA`2004). – Минск, 2004, с. 18-23.
2. Абрамов С.М., Московский А.А., Роганов В.А., Шевчук Ю.В., Шевчук Е.В., Парамонов Н.Н., Чиж О.П. 2005. Open TS: архитектура и реализация среды для динамического распараллеливания вычислений. Proc. Научный сервис в сети Интернет: технологии распределенных вычислений : Труды Всероссийской научной конференции, 19-24 сентября 2005 г. Новороссийск, Изд-во МГУ, М., с. 79-81.
3. Адигеев М.Г., Дубров Д.В., Лазарева С.А., Штейнберг Б.Я. Экспериментальный распараллеливающий компилятор на Супер-ЭВМ со структурной организацией вычислений// Фундаментальные и прикладные аспекты разработки больших распределенных программных комплексов. Тезисы докладов всероссийской научной конференции. Новороссийск. 21-26.09.98. Москва: МГУ, 1998. с.101-108.
4. Адуцкевич Е.В., Баханович С.В., Лиходед Н.А. Условия получения согласованного таймирования и распределения операций и данных между процессорами. // Тр. междунар. науч. конф. «Суперкомпьютерные системы и их применение» (SSA`2004). – Минск, 2004, с. 160-165.
5. Адуцкевич Е.В., Баханович С.В. Адаптация алгоритмов к реализации на системах с распределенной памятью. Пространственно-временная локализация и распределение данных. // Тр. междунар. науч. конф. «Суперкомпьютерные системы и их применение» (SSA`2004). – Минск, 2004, с. 165-170.
6. Алексахин В.Ф., Ефимкин К.Н., Ильяков В.Н., Крюков В.А., Кулешова М.И., Сазанов Ю.Л. Средства отладки MPI-программ в DVM-системе.//

Труды Всероссийской научной конференции «Научный сервис в сети Интернет: технологии распределенных вычислений», 19-24 сентября 2005 г., г. Новороссийск, с. 113-115.

7. Алексахин В.Ф., Ильяков В.Н., Коновалов Н.А., Ковалева Н.В., Крюков В.А., Поддерюгина Н.В., Сазанов Ю.Л. Система автоматизации разработки параллельных программ для вычислительных кластеров и сетей (DVM-система). Научный сервис в сети Интернет. Труды Всероссийской научной конференции, 23-28 сентября 2002, г. Новороссийск. – М.: изд-во МГУ, с. 272-273.
8. Аллен Р., Кеннеди К. Автоматическая трансляция Фортран-программ в векторную форму. Векторизация программ: теория, методы, реализация. Сб. статей. Москва: Мир. 1991, с.77 - 140.
9. Андрианов А.Н., Ефимкин К.Н., Задыхайло И.Б. Непроцедурный язык для решения задач математической физики.// Программирование, №2, 1991, с. 80-94.
10. Андрианов А.Н. Использование языка НОРМА для решения вычислительных задач на нерегулярных сетках. // Фундаментальные и прикладные аспекты разработки больших распределенных программных комплексов. Тезисы докладов всероссийской научной конференции. Новороссийск. 21-26.09.98. Москва: МГУ, 1998. с.120-122.
11. Антонов А.С., Воеводин Вл.В. Новый подход к построению методов межпроцедурного анализа программ // Труды первой международной научно-практической конференции по программированию УкрПРОГ-98, Киев, 1998 г.
12. Антонов А.С. Использование канонического описания графа алгоритма для определения свойств программ.// Труды Всероссийской научной конференции «Научный сервис в сети Интернет: технологии распределенных вычислений», 19-24 сентября 2005 г., г. Новороссийск, с. 100-102.

13. Арапов Д. М., Калинов А. Я., Ластовецкий А. Л., Ледовских И. Н., Посыпкин Н. А. Язык и система программирования для высокопроизводительных параллельных вычислений на неоднородных сетях. // Программирование. – 2000. - № 4. с. 55-80.
14. Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструменты. : Пер. с англ. – М. Издательский дом «Вильямс», 2001. – 768с.
15. Букатов А.А., Жегуло О.А., Коваль В.В. Создание системы поддержки распараллеливания программ, основанной на непроцедурном описании распараллеливающих программ.// Труды Всероссийской научной конференции «Научный сервис в сети Интернет: технологии распределенных вычислений», 19-24 сентября 2005 г., г. Новороссийск, с. 69-71.
16. Бурцев В.С. Выбор новой системы организации выполнения высокопараллельных вычислительных процессов, примеры возможных архитектурных решений построения суперЭВМ. В сб. трудов академика В.С. Бурцева "Параллелизм вычислительных процессов и развитие архитектуры суперЭВМ". // ИВВС РАН, М., 1997, с. 41-78.
17. Бурцев В.С. Новые принципы организации вычислительных процессов высокого параллелизма. // Труды Первой Всероссийской научной конференции «Методы и средства обработки информации», 1-3 октября 2003, Москва, с. 17-32.
18. Бурцев В.С. Новые подходы к оценке качества вычислительных средств.// Интеллектуальные и многопроцессорные системы – 1999/ Тезисы докладов Международной научной конференции. Таганрог: Изд-во ТРТУ, 1999, с. 9-21.
19. Вальковский В.А. Параллельное выполнение циклов. Метод параллелепипедов. - Кибернетика. 1982, N 2. с. 51-62.
20. Вальковский В.А. Параллельное выполнение циклов. Метод пирамид. - Кибернетика. 1983, N 5. с. 51-55.

21. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. Санкт-Петербург «БХВ-Петербург» 2002. – 608 с.
22. Воеводин В. В. Математические модели и методы в параллельных процессах. Москва: Наука. 1986. – 296 с.
23. Воеводин В. В. Информационная структура алгоритмов. М.: МГУ, 1997. – 139 с.
24. Воеводин В. В. Математические основы параллельных вычислений. М.: МГУ, 1991. – 345 с.
25. Воеводин В.В., Пакулев В.В. Определение дуг графа алгоритма. М.: Препринт № 228, 1989. 22 с.
26. Вольф М. Перестановка циклов. Векторизация программ: теория, методы, реализация. Сб. статей. Москва: Мир. 1991. С. 48 – 65.
27. Гайсарян С.С., Аветисян А.И., Падарян В.А., Леонтьев Г. Применение среды ParJava для разработки параллельных программ. // Тр. междунар. науч. конф. «Суперкомпьютерные системы и их применение» (SSA`2004). – Минск, 2004. с. 99-104.
28. Гергель В.П., Сибирякова А. Программная система для изучения и исследования параллельных методов решения сложных вычислительных задач. // Материалы второго Международного научно-практического семинара «Высокопроизводительные параллельные вычисления на кластерных системах», 26-29 ноября, 2002 г, с. 82-88.
29. Громова В.В., Прохоров Д.А., Самофалов В.В. Комбинированный метод оптимизации параллельных программ для систем с общей памятью.// Труды Всероссийской научной конференции «Научный сервис в сети Интернет: технологии распределенных вычислений», 19-24 сентября 2005 г., г. Новороссийск, с. 104-106.
30. Ефимкин К.Н. Язык НОРМА и методы его трансляции для параллельных вычислительных систем. // Фундаментальные и прикладные аспекты разра-

ботки больших распределенных программных комплексов. Тезисы докладов всероссийской научной конференции. Новороссийск. 21-26.09.98. Москва: МГУ, 1998. с.117-119.

31. Игумнов А.С., Открытая платформа отладки параллельных программ// Научный сервис в сети Интернет. Труды Всероссийской научной конференции, г. Новороссийск, 22-27 сентября 2003. – М.: изд-во МГУ, с. 92-94.
32. Ильяков В.Н.. Коваленко И.В.. Крюков В.А. Анализ и предсказание эффективности выполнения DVM-программ на неоднородных сетях ЭВМ// Научный сервис в сети Интернет. Труды Всероссийской научной конференции, г. Новороссийск, 22-27 сентября 2003. – М.: изд-во МГУ, с. 181-182.
33. Лазарева С. А. Использование многоуровневого внутреннего представления в автоматическом распараллеливании программ для многопроцессорных ЭВМ. Диссертация на соискание ученой степени кандидата технических наук. РГУ, 2000.
34. Лиходед Н.А. Распределение операций и массивов данных между процессорами.// Программирование, 2003, №3, с. 73-80.
35. Петренко В. В. Внутреннее представление ОРС 3.// Труды Всероссийской научной конференции «Научный сервис в сети Интернет: технологии распределенных вычислений», 19-24 сентября 2005 г., г. Новороссийск, с. 106-108.
36. Падуа Д., Вольф М. Оптимизация в компиляторах для суперкомпьюте-ров. Векторизация программ: теория, методы, реализация. Сб. статей. Москва: Мир. 1991. С. 7-47.
37. Сб. статей. Векторизация программ: теория, методы, реализация. Москва: "Мир". 1991. 272 с.
38. Свами М., Тхуласираман К.. Графы, сети и алгоритмы. Москва: Мир. 1984. 455 с.

39. Схрейвер А.. Теория линейного и целочисленного программирования. Москва: Мир. 1991. Т. 1, 2.
40. Торчигин С.В. Особенности отладки прикладных и системных задач на вычислительной системе с автоматическим распределением ресурсов// Интеллектуальные и многопроцессорные системы – 2003/ Тезисы докладов Международной научной конференции. Таганрог: Изд-во ТРТУ, 2003, с. 55-57.
41. Фролов А.В. Оптимизация размещения массивов в Фортран-программах на многопроцессорных вычислительных системах // Программирование, 1998, №3, с. 70-80.
42. Фролов А.В. Система МАКРОГРАФ и другие технологии ускорения исполнения ФОРТРАН-программ. // Труды Первой Всероссийской научной конференции «Методы и средства обработки информации», 1-3 октября 2003, Москва, с.
43. Чумаченко Г. О. Технология отложенных токенов для преодоления ситуаций блокировки по переполнению буферов в суперпроцессоре нетрадиционной архитектуры. // Труды Первой Всероссийской научной конференции «Методы и средства обработки информации», 1-3 октября 2003, Москва, с. 151-158.
44. Ширай А.Е. Виртуальная потоковая машина // Труды Всероссийской научной конференции «Методы и средства обработки информации», 5-7 октября 2005, Москва.
45. Ширай А.Е. Системная поддержка вычислений в комплексе с автоматическим распределением ресурсов. // Интеллектуальные и многопроцессорные системы – 2003/ Тезисы докладов Международной научной конференции. Таганрог: Изд-во ТРТУ, 2003, с.54-55.
46. Штейнберг Б. Я. Математические методы распараллеливания рекуррентных циклов для суперкомпьютеров с параллельной памятью.// Ростов-на-Дону, Издательство Ростовского университета, 2004 г., 192 с.

47. Штейнберг Б. Я. Подстановка и переименование в многомерных циклах при распараллеливании. // Материалы международной научно-технической конференции «СуперЭВМ и многопроцессорные вычислительные системы (МВС-2002)», 26-30 июня 2002, г. Таганрог, Россия, с. 161-164.
48. Штейнберг Б. Я. Открытая распараллеливающая система // РАСО'2001/ Труды международной конференции «Параллельные вычисления и задачи управления». М.: ИПУ РАН, 2001. С. 214-220.
49. Штейнберг Б. Я. Распараллеливание программ для суперкомпьютеров с параллельной памятью и Открытая распараллеливающая система. Диссертация на соискание ученой степени доктора технических наук. РГУ, 2004.
50. Штейнберг Б.Я., Арутюнян О.Э., Бутов А.Э., Гуфан К.Ю., Морылев Р., Науменко С.А., Петренко В.В., Тузаев А., Черданцев Д.Н., Шилов М.В., Штейнберг Р.Б., Шульженко А.М. Обучающая распараллеливанию программа на основе ОРС.// Научно-методическая конференция «Современные информационные технологии в образовании: Южный федеральный округ», Ростов-на-Дону, 12-15 мая 2004 г., с. 248-250.
51. Штейнберг Б.Я., Бутов А.Э., Науменко С.А., Петренко В.В., Черданцев Д.Н., Штейнберг Р.Б., Шульженко А.М. Полуавтоматическое распараллеливание на основе ОРС.// Труды всероссийской научно-технической конференции «Параллельные вычисления в задачах математической физики», 21-25 июня 2004, г. Ростов-на-Дону, с. 186-194.
52. Штейнберг Б. Я., Макошенко Д. В., Черданцев Д. Н., Шульженко А. М. Внутреннее представление в Открытой распараллеливающей системе. //Искусственный интеллект. Научно-теоретический журнал. Институт проблем искусственного интеллекта НАНУ. Украина, Донецк, ДонДИШИ, «Наука и Освита», 2003, №4, с. 89-97.
53. Шульженко А. М. Преимущества определения информационных зависимостей в программе с помощью решетчатых графов // XIII Международная

конференция «Математика. Экономика. Образование.». Тезисы докладов. Ростов н/Д, 2005 - 195 с. - ISBN 5-94153-097-8. с. 137

54. Шульженко А. М. Расщепление многомерных циклов для эффективного распараллеливания.// Труды всероссийской научно-технической конференции «Параллельные вычисления в задачах математической физики», 21-25 июня 2004, г. Ростов-на-Дону, с. 186-194.
55. Шульженко А. М. Решетчатый граф и использующие его преобразования программ в Открытой распараллеливающей системе.// Труды Всероссийской научной конференции «Научный сервис в сети Интернет: технологии распределенных вычислений», 19-24 сентября 2005 г., г. Новороссийск, с. 82-85.
56. Шульженко А. М. Автоматическое определение циклов ParDo в программе. // Известия вузов. Северокавказский регион. Естественные науки. Приложение 11’05. с. 77-88.
57. Allen R., Kennedy K. Optimizing compilers for modern architectures. Morgan Kaufmann Publishers, An Imprint of Elsevier. 2002. – 790 p.
58. Ancourt C., Irigoin F. Scanning polyhedra with DO loops. // In ACM/SIGPLAN Symp. on Principles and Practice of Parallel Programming, pages 39--50, ACM, June 1991.
59. Banerjee U., Chen S. C., Kuck D., Towle R. Time and Parallel Processor Bounds for Fortran-like Loops// IEEE Trans. on Computers. 1979. C-28. No.9. P.660-670.
60. Calland P., Darte A., Robert Y., Vivien F. Plugging anti and output dependence removal techniques into loop parallelization algorithms.// Parallel Computing 23(1-2): 251-266 (1997).
61. Chamski Z. Fast and efficient generation of loop bounds. // Proceedings of ParCo '93, Elsevier Science Publishers (North Holland).

62. Chamski Z. Nested Loop Sequences: Towards Efficient Loop Structures in Automatic Parallelization. // HICSS 1994: Maui, Hawaii, USA - Volume 2. P 14-22.

63. Collard J.-F., Feautrier P. Automatic Generation of Data Parallel Code.//Proceedings of the Fourth Interantional Workshop on Compilers for Parallel Computers. Delft, The Netherlands, Dec. 1993, pages 321-332.
64. Collard J.-F., Feautrier P., Risset T. Construction of DO Loops from Systems of Affine Constraints. Research Report 93-15, Ecole Normale Superieure de Lyon, Lyon, France, May 1993.
65. Cytron R., Ferrante J. What's in a name? Or the value of renaming for parallelism detection and storege allocation. IBM Res. Rep. RC 12785 (#55984), 1987.
66. Darte A., Vivien F. On the optimality of Allen and Kennedy's algorithm for parallelism extraction in nested loops. // Euro-Par '96 Parallel Processing, Second Int. Euro-Par Conf., Lyon, France, August 26-29, 1996, Proceedings, Volume I. P: 379-388.
67. Darte A., Vivien F. Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs. // International Journal of Parallel Programming, 25(6):447-496, December 1997.
68. Darte A., Robert Y., Vivien F. Loop Parallelization Algorithms. // Chapter in volume 1808 of LNCS on Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques, and Run Time Systems, Dharma P. Agrawal and Santosh Pande editors, pages 141-172. Springer Verlag, 2001.
69. Feautrier P. Array Expansion. //In ACM Int. Conf. on Supercomputing, St Malo, 1988.
70. Feautrier P. Parametric integer programming// Operationnelle/Operations Research, 22(3):243--268, September 1988.
71. Feautrier P. Dataflow analysis of scalar and array references// International Journal of Parallel Programming, 20(1):23--52, February 1991.

72. Feautrier P. Some efficient solution to the affine scheduling problem, part I, One Dimensional Time. //Int. J. of Parallel Programming, 21(5), Oct. 1992.
73. Feautrier P. Some efficient solution to the affine scheduling problem, part II, Multidimensional time. //Int. J. of Parallel Programming, 21(6), Oct. 1992.
74. Feautrier P. Fine-grain scheduling under Resource Constraints// Proceedings of Languages and Compilers for Parallel Computing, 7th International Workshop, LCPC'94, Ithaca, NY, USA, August 8-10, 1994.
75. Kalinov A., and Klimov S., Optimal mapping of a parallel application processes onto heterogeneous platform // Proceedings of 19th International Parallel and Distributed Processing Symposium, Denver, USA, April 2005, IEEE CS, CD-ROM.
76. Kelly W., Pugh W. A framework for unifying reordering transformations.// Technical Report: CS-TR-2995. University of Maryland at College Park, MD, USA. 1993.
77. Kuck D.J., Kuhn R.H., Leasure B., Wolfe M. Depedance graph and compiler optimizations. Proc. 8-th ACM Symp. On Principles of Progr. Lang. (Williamsburg, Va., Jan. 26-28), 1981, p. 207-218.
78. Lamport L. The parallel execution of DO loops// Commun. ACM. 1974. Vol.17. No.2. P.83-93.
79. Lim A., Lam M. Maximizing parallelism and minimizing synchronization with affine partitions.// Parallel Computing, 24:445-475, 1998.
80. Lim A.W., Lam M.S. Cache optimizations with affine partitioning// Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing, Portsmouth, Virginia, 2001.
81. Lim A.W., Liao S.-W., Lam M.S. Blocking and array contraction across arbitrary nested loops using affine partitioning// Proceedings of the ACM SIGPLAN Simposium on Principles and Practice of Programming Languages, 2001.
82. Maslov V. Delinearization: An Efficient Way to Break Multiloop Dependence Equations. // Proceedings of the ACM SIGPLAN'92 Conference on Programming

Language Design and Implementation (PLDI), San Francisco, California, June 17-19, 1992. SIGPLAN Notices 27(7) (July 1992). 152-161.

83. Maydan D. E., Amarasinghe S. P., Lam M. S.. Data dependence and data-flow analysis of arrays// In 5th Workshop on Languages and Compilers for Parallel Computing, New Haven, CT, August 1992.
84. Maydan D. E., Hennessy J. L., and Lam M. S.. Efficient and exact data dependence analysis. //In SIGPLAN 1991 Conference on Programming Language Design and Implementation, pages 1--14, ACM, Toronto, Canada, June 1991.
85. Maydan D. E., Hennessy J. L., and Lam M. S.. Effectiveness of data dependence analysis.// International Journal of Parallel Programming. Volume 23, Issue 1 (February 1995). Pages: 63 - 81
86. Padua D., Wolfe M. Advanced Compiler Optimizations for Supercomputers// Commun. ACM. 1986. Vol. 29. No. 12. P. 1184-1201.
87. Paek Y., Petersen P. A Data Dependence Graph in Polaris// Technical Report 1495, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1996.
88. Petersen P., Padua D. Static and Dynamic Evaluation of Data Dependence Analysis Techniques. // IEEE Trans. on Computers. 1996. Vol 7, No 11. P. 1121-1132.
89. Pugh W. The Omega test: a fast and practical integer programming algorithm for dependence analysis// Communications of ACM, 8:102-114, August 1992.
90. Pugh W. Definitions of Dependence Distance// Letters on Programming Languages and Systems, 1(3):261-265, September 1993.
91. Pugh W., Wonnacott D. Going beyond integer programming with Omega test to eliminate false data dependences.// Technical Report CS-TR-2993, Dept. of Computer Science, University of Maryland, Collage Park, December 1992. An earlier version of this paper appeared at the SIGPLAN PLDI'92 conference.

92. Pugh W., Wonnacott D. An Exact Method for Analysis of Value-based Array Data Dependences.// Technical Report CS-TR-3196, Dept. of Computer Science, University of Maryland, Collage Park, December 1993.
93. Verge H., Dongen V., Wilde D. Loop Nest Synthesis using Polyhedral Library.// IRISA Technical Report #830 (English) Presented at RENPAR`6 (French).
94. Wolfe M. The Definition of Dependence Distance. // ACM Trans. Program. Lang. Syst. 16(4): 1114-1116 (1994).
95. Wolfe M., Banerjee U. Data Dependence and Its Application to Parallel Processing// International Journal of Parallel Programming. 1987. Vol.16. No.2. P.137-178.
96. Wolfe M. E., Lam M. S. A loop transformation theory and an algorithm to maximize parallelism// IEEE Transactions on Parallel and Distributed Systems, July 1991.
97. http://ops.rsu.ru
98. http://www.cri.ensmp.fr/~pips/
99. http://www.cri.ensmp.fr/~pips/pipsmake-rc/node39.html#SECTION00079300000000000000
100. http://www.amd.com/ru-ru/Processors/ProductInformation/0,,30_118_9485_13041,00.html
101. http://www.intel.com/cd/products/services/emea/rus/desktop/processors/pentium_d/216831.htm
102. http://www.intel.com/cd/products/services/emea/rus/processors/211601.htm
103. http://www.csrd.uiuc.edu/promis/
104. http://www.ispras.ru/~mpc
105. www.parallel.ru/v-ray
106. http://suif.stanford.edu/
107. http://suif.stanford.edu/research/affine.html
108. http://www.crescentbaysoftware.com/

109. ParaWise 2.4. // Parallel Software Products Inc. (http://www.parallelsp.com/index.htm)

110. ParaWise User Guide.// World Wide Web Document. http://www.parallelsp.com/downloads/parawise-2.4/parawise-user-guide-2.4-001.pdf

111. http://www.infosun.fmi.uni-passau.de/cl/loopo/

ПРИЛОЖЕНИЕ А

На нижеследующем рисунке показана степень загрузки процессора Intel Pentium-4 530 (с технологией Hyper Threading) при исполнении одной, непрерывно вычисляющей программы test. Текст этой программы можно видеть в окне «test – Microsoft Visual C++ …» (рис. 30): основная часть работы этой программы состоит в исполнении трехмерного гнезда циклов, внутри которого исполняется единственный оператор. При исполнении программы в окне «Диспетчер задач Windows» (рис. 30) можно наблюдать, что мощность процессора используется чуть более чем на 50%.

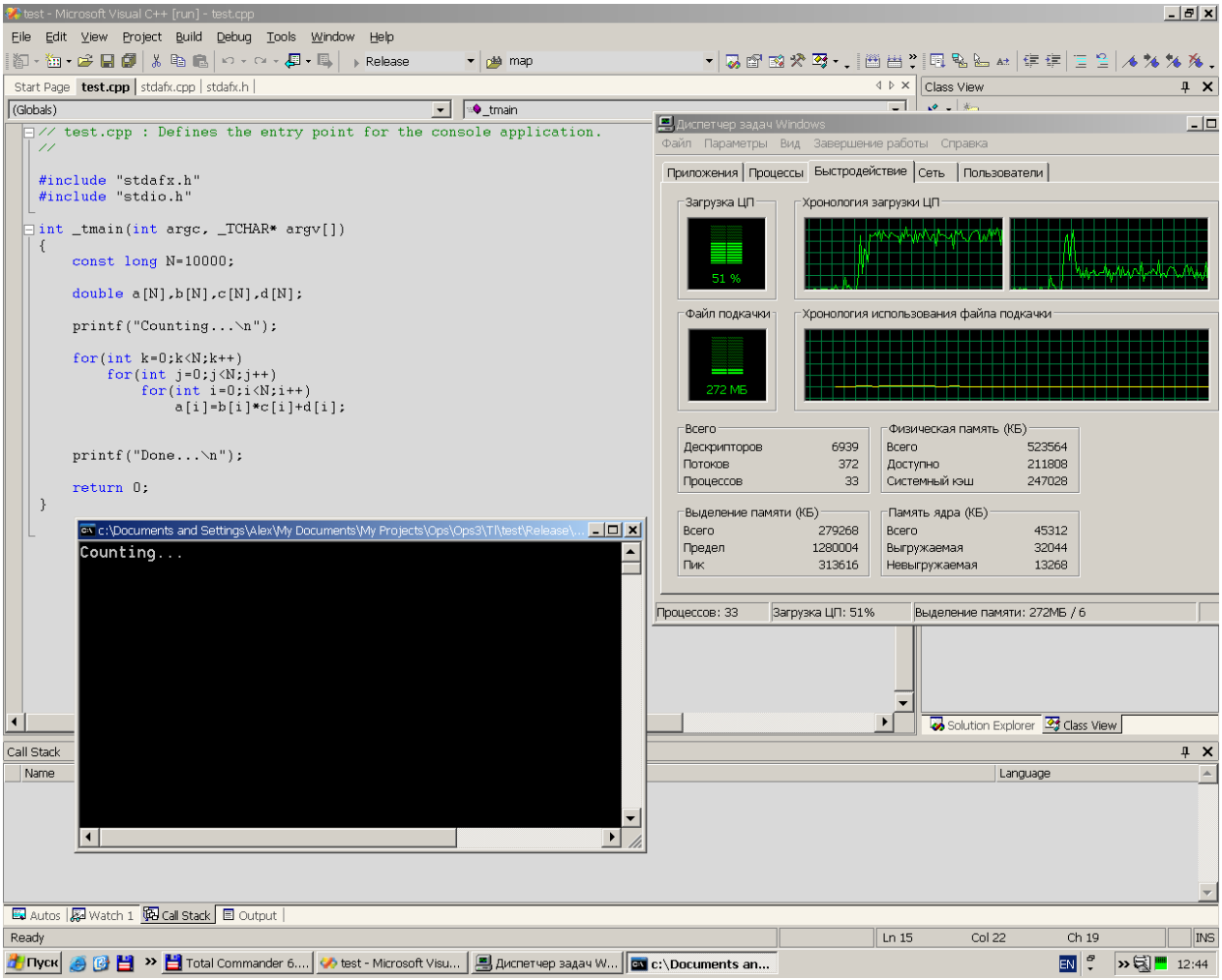


Рис. 30. Исполнение программы test на персональном компьютере с процессором Intel Pentium-4 530.

ПРИЛОЖЕНИЕ Б

В таблице 2 приведены данные о времени построения решетчатых графов для указанных фрагментов программ. Тексты программ приведены в таблице 3.

Таблица 2.

	Время построения всевозможных элементарных снизу решетчатых графов для программы, сек.		Время построения всевозможных простых снизу решетчатых графов для программы, сек.		Количество дуг в графе инф. зависимостей программы, построенном с помощью неравенств Банержи и НОД теста	
Программа	Без учета входной зависимости	С учетом входной завис-ти	Без учета входной зависимости	С учетом входной завис-ти	Без учета входной зависимости	С учетом входной завис-ти
Scalar_exp_tst.c	0.031	0.031	0.062	0.094	8	11
Slae_revsubst.c	0.031	0.047	0.125	0.297	11	16
Poly_mult.c	0.015	0.031	0.070	0.094	5	8
Matrix_mult.c	0.063	0.109	0.120	0.188	3	6
Dirih.c	0.016	0.040	0.016	0.079	4	10
Jordan.c	0.135	0.219	0.344	0.625	21	31
Gauss_elim.c	0.125	0.266	0.313	0.954	10	27
LU_decomp2.c	0.235	0.375	0.625	1.515	38	69

Таблица 3.

Scalar_exp_tst.c	Slae_revsubst.c

`for(i=1;i<=10;i=i+1)`
`{`
`x=b[i][10];`
`for(j=1;j<=10;j=j+1)`
`{`
`a[i][j] = (b[i][j]+x);`
`x = b[i][j];`
`}`
`}`
	`x[0]=b[0];`
	`for(i=1;i<=100;i=i+1)`
	`{`
	`x[i]=b[i];`
	`for ( j=0 ;j<=(i-1); j=j+1)`
	`{`
	`x[i]=x[i]-a[i][i-j]*x[i-j];`
	`}`
	`}`

Poly_mult.c

```
for(k=1;k<=100;k=k+1)
 {
  c[k]=0;
 }

 for(i=1;i<=50;i=i+1)
  for(j=1;j<=50;j=j+1)
  {
   c[i+j]=c[i+j]+a[i]*b[j];
  }
```

Dirih.c

```
for(i=1;i<=100;i=i+1)
  for(j=1;j<=100;j=j+1)
  {
   a[i][j]=a[i-1][j]+a[i][j-1] +
a[i+1][j]+a[i][j+1];
  }
```

Jordan.c

```
for(k=0;k<=99;k=k+1)
 {
    l[0]=1/a[0][k];

    for(p=1;p<=99;p=p+1)
    {
      l[p]=-a[p][k];
    }

    for(j=k+1;j<=100;j=j+1)
    {
      u[j]=a[0][j]*l[0];
      for(i=1;i<=99;i=i+1)
      {
        a[i-1][j]=a[i][j]+l[i]*u[j];
      }
      a[99][j]=u[j];
    }
 }
```

Gauss_elim.c

```
for(i=0;i<=99;i=i+1)
 {
  for(j=(i+1);j<=99;j=j+1)
  {
   tmp[i][j]=a[j][i]/a[i][i];
   for (k=(i+1);k<=100;k=k+1)
   {
    a[j][k]=a[j][k]-
tmp[i][j]*a[i][k];
   }
  }
 }
```

LU_decomp2.c

```
for(i1=0;i1<=99;i1=i1+1)
 {
   L[i1][0]=A[i1][0];
 }

 for(j1=0;j1<=99;j1=j1+1)
 {
   U[0][j1]=A[0][j1]/A[0][0];
 }

  for(j=1;j<=99;j=j+1)
  {
   for(i=j;i<=99;i=i+1)
   {
     SUM[j][i]=0;
     for (k1=0;k1<=(j-1);k1=k1+1)
     {
```

Matrix_mult.c

```
for(i=0;i<100;i=i+1)
  for(j=0;j<100;j=j+1)
   for(k=0;k<100;k=k+1)
   {
    c[i][j]=c[i][j]+a[i][k]*b[k][j];
   }
```

```
SUM[j][i]=SUM[j][i]+L[i][k1]*U[k1][j];
      }
      L[i][j]=A[i][j]-SUM[j][i];
    }
    U[j][j]=1;
    for(i2=(j+1);i2<=99;i2=i2+1)
    {
      SUMM[j][i2]=0;
      for (k=0;k<=(j-1);k=k+1)
      {

SUMM[j][i2]=SUMM[j][i2]+L[j][k]*U[k][i2];
      }
      U[j][i2]=(A[j][i2]-
SUMM[j][i2])/L[j][j];
    }
   }
```

ПРИЛОЖЕНИЕ В

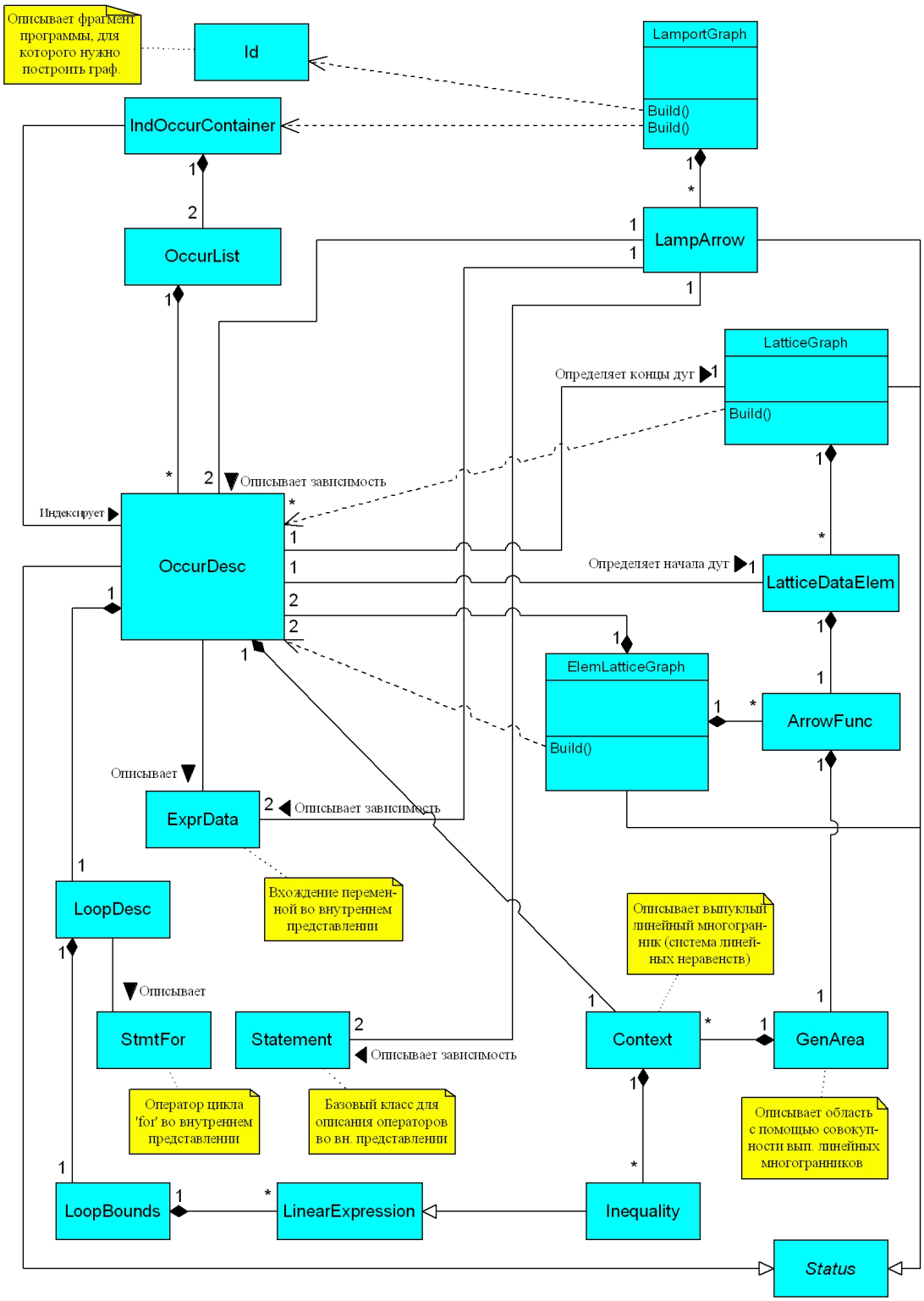


Рис. 31. Диаграмма основных классов, реализующих анализ информационных зависимостей в ОРС.