МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Синюкова Валерия Константиновича

Тема работы:

РАЗРАБОТКА И РЕАЛИЗАЦИЯ АЛГОРИТМОВ АВТОМАТИЧЕСКОГО КОНСТРУИРОВАНИЯ LUNA-ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ СИСТЕМЫ SAPFOR

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Малышкин В. Э./…………..
(ФИО) / (подпись)
«31» мая 2024 г.

Руководитель ВКР
д.т.н., профессор
зав. каф. ПВ ФИТ НГУ
Малышкин В. Э./………...
(ФИО) / (подпись)
«31» мая 2024 г.

Соруководитель ВКР
к.т.н.
ст. преп. каф. ПВ ФИТ НГУ
Перепёлкин В. А./…………..
(ФИ О) / (подпись)
«3» мая 2024 г.

Новосибирск, 2024

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий

Кафедра параллельных вычислений
(название кафедры)

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ

Зав. кафедрой Малышкин В. Э.
(фамилия, И., О.)

…………………
(подпись)

«03» ноября 2023 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА

Студенту Синюкову Валерию Константиновичу, группы 20203
(фамилия, имя, отчество, номер группы)

Тема Разработка и реализация алгоритмов автоматического конструирования LuNA-программ с использованием системы SAPFOR
(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 3 ноября 2023 г. №0413

Срок сдачи студентом готовой работы 20 мая 2024 г.

Исходные данные (или цель работы):

связывание системы SAPFOR и LuNA для автоматического распараллеливания последовательных программ для класса задач на примере итерационного метода Якоби решения систем линейных уравнений

Структурные части работы:

содержание, введение, обзор родственных работ, анализ решения задачи, разработка алгоритмов, реализация алгоритмов, тестирование

Руководитель ВКР
зав. каф. ПВ ФИТ НГУ
д.т.н., профессор
Малышкин В.Э./…………..
(ФИ О) / (подпись)

«03» ноября 2023 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ НГУ
к.т.н.
Перепёлкин В. А./…………..
(ФИ О) / (подпись)

Задание принял к исполнению
Синюков В. К./…………..
(ФИО студента) / (подпись)

«03» ноября 2023 г.

СОДЕРЖАНИЕ

Определения, обозначения и сокращения....................4
Введение....................5
1 Обзор предметной области....................9
1.1 Полиэдральная модель....................10
1.2 MapReduce, Hadoop....................10
1.3 Распараллеливание программ на основе директив....................11
1.4 LLVM IR....................11
1.5 Выводы....................12
2 Алгоритмическое и информационное обеспечение преобразователя для систем SAPFOR и LuNA....................13
2.1 Система LuNA и модель фрагментированного программирования....................13
2.2 Система SAPFOR для автоматизации распараллеливания программ....................18
2.3 Проблемы конструирования LuNA-программы....................18
2.4 Предлагаемый подход к автоматическому конструированию LuNA-программ....................19
2.5 Принципы конструирования LuNA-программы....................20
2.6 Генерация LuNA-программы....................34
2.7 Фрагментация LuNA-программы....................40
2.8 Сбор информации о параллелизме....................44
2.9 Итоги....................45
3 Генерация фрагментов кода....................46
3.1 Фрагментация по пространству....................46
3.2 Упорядоченное выполнение фрагментов вычислений....................57
3.3 Форматы представления информации о параллелизме и схемы LuNA-программе....................59
3.4 Реализация генератора фрагментов кода....................60
3.5 Тестирование генератора фрагментов кода....................62
3.6 Итоги....................63
Заключение....................64
Список использованных источников и литературы....................65
Приложение А....................68
Приложение Б....................70
Приложение В....................71
Приложение Г....................73
Приложение Д....................75
Приложение Е....................78
Приложение Ё....................80
Приложение Ж....................82
Приложение З....................84
Приложение И....................91

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Система конструирования — система, которая автоматически конструирует параллельную программу, исходя из спецификации на специально разработанном для этого языке программирования.

Системы распараллеливания — система, которая автоматически конструирует параллельную программу, исходя из последовательной программы на традиционном языке программирования, реализующей тот же алгоритм, который должен быть реализован результирующей параллельной программой.

Информация о параллелизме последовательной программы — информация о независимых частях последовательной программы и о том, как эту программу необходимо перестроить, чтобы выполнять эти части параллельно.

Связывание системы распараллеливания и системы конструирования — использование информации о параллелизме, выявленной системой распараллеливания для автоматического составления спецификации для системы конструирования.

Преобразователь — система, обеспечивающая связывание системы распараллеливания и системы конструирования.

Метод Якоби — итерационный метод Якоби решения систем линейных уравнений.

ВВЕДЕНИЕ

Параллельное программирование является важной частью всего современного программирования. Существуют программы, в которых без использования параллелизма не может быть достигнута необходимая производительность. Например, в веб-порталах за счет параллельных вычислений существенно сокращается время ожидания пользователя. Отдельной нишей параллельного программирования является научное моделирование на суперкомпьютерах. Типичной ситуацией для задач из этой области является выполнение ресурсоемких математических операций, например, умножения больших (размера $10000 \cdot 10000$ и больше) матриц. Параллельное выполнение таких операций может быть в разы, десятки или сотни раз быстрее, чем последовательное. Такое ускорение позволяет существенно сэкономить время, что дает научным специалистам больше возможностей для проведения экспериментов, разработки и отладки научных методов и т.д., снижает затраты на электроэнергию и т.д.

Но у параллельного программирования есть не только преимущества, но и недостатки. Для того, чтобы написать эффективную параллельную программу, программист должен уметь решать специфичные для параллельного программирования задачи, такие как синхронизация вычислений, декомпозиция данных, обеспечение коммуникации между потоками и процессами, балансировка нагрузки по вычислительным узлам и т.д. Под эффективной программой здесь и далее неформально понимается программа, экономно расходующая важные для конкретной предметной области ресурсы, например, оперативную память, электроэнергию, процессорное время и т.д. Помимо этого, степень недетерминизма в параллельных программах существенно больше, чем в последовательных, из-за чего их сложнее отлаживать: ошибка может проявляться не при каждом запуске программы, вследствие чего ее обнаружение осложняется. По вышеописанным причинам существуют алгоритмы, для которых есть только последовательная реализация, а параллельной нет, хотя она могла бы принести пользу.

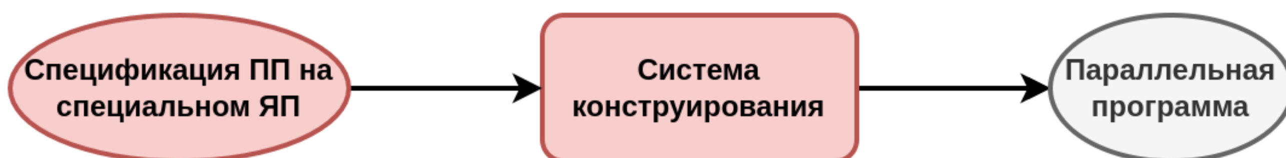
Для решения обозначенной проблемы может применяться автоматическое конструирование параллельной программы по ее спецификации, составленной программистом вручную.

Системы автоматического конструирования параллельных программ можно различать по видам спецификаций этих программ. В нижеизложенном списке описываются два класса таких систем.

1) Системы, в которых спецификации составляются на специально разработанном для этого языке программирования (рисунок 1). Такие системы будут далее называться

системами конструирования. Примерами систем конструирования могут выступать DVM [1, 2], НОРМА [3], LuNA [4-5].

2) Системы, в качестве спецификации для которых выступает последовательная программа, реализующая тот же алгоритм, который должен быть реализован сконструированной параллельной программой (рисунок 2). Такие системы будут далее называться *системами распараллеливания*. Примерами систем распараллеливания могут выступать PLUTO [6], Par4All [7], AutopaR [8]. Общий принцип работы таких систем заключается в том, что они анализируют последовательную программу, выявляют в ней независимые части и перестраивают ее так, чтобы это части можно было выполнять параллельно.



ПП — параллельная программа.

ЯП — язык программирования.

Рисунок 1 — Схема работы системы конструирования

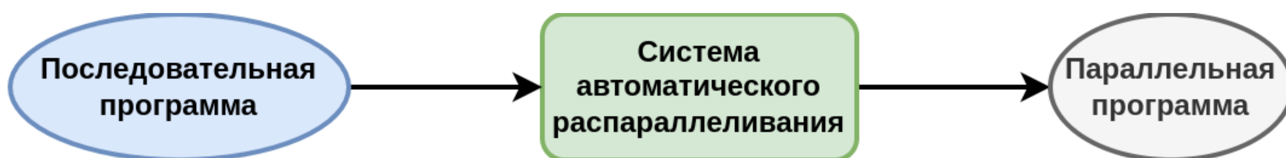


Рисунок 2 — Схема работы системы распараллеливания

Перспективным направлением исследований является использование информации о параллелизме, выявленной системой распараллеливания, для автоматического составления спецификации для системы конструирования. Это использование далее будет называться *связыванием*.

У каждой системы распараллеливания и каждой системы конструирования есть свои области применения, свои множества приложений, для которых конструируемые ими параллельные программы получаются лучше, чем конструируемые всеми остальными системами. Связывание системы распараллеливания и системы конструирования позволит получать разные параллельные программы из последовательных: ту, которую система распараллеливания вырабатывает сама, и ту, которую она вырабатывает совместно с системой конструирования. Из этих программ можно будет выбирать ту, которая в большей степени удовлетворяет условиям решаемой задачи.

При обеспечении подобного связывания возникает проблема, которая обусловлена тем, что в общем случае нельзя использовать информацию, выявленную системой распараллеливания, для составления спецификации напрямую из-за того, что системы распараллеливания и системы конструирования оперируют в разных моделях вычислений. Из-за этого возникает дополнительная задача по преобразованию и дополнению этой информации таким образом, чтобы по ней можно было составить спецификацию для системы конструирования. Для решения этой задачи требуется разработать отдельную систему, которая далее будет называться *преобразователем*. В общем случае создание преобразователя — это сложная задача, у которой на данный момент не просматривается какого-то универсального решения, поэтому важным является нахождение частных решений, что будет способствовать продвижению исследований по этой теме.

Надо отметить, что в общем случае для реализации вышеописанной идеи нужно только создать преобразователь, а серьезной переработки системы распараллеливания или системы конструирования не требуется.

Работа направлена на создание преобразователя для системы распараллеливания SAPFOR [9-14] и системы конструирования LuNA для автоматического распараллеливания последовательных программ. В такой постановке задача является слишком сложной, поэтому рассматриваться будет конкретное приложение: итерационный метод Якоби решения систем линейных уравнений (далее — *метод Якоби*). Тем не менее для многих других, похожих по способу распараллеливания, задач найденное решение также будет работать. Система SAPFOR предназначена для автоматизации распараллеливания программ в терминах моделей DVMH, OpenMP, MPI и т.д. Система LuNA предназначена для автоматического конструирования параллельных программ. Эта система строит параллельную программу, исходя из описания задачи в терминах предметной области.

Цель работы: связывание системы SAPFOR и LuNA для автоматического распараллеливания последовательных программ для класса задач на примере итерационного метода Якоби решения систем линейных уравнений.

Для того, чтобы достичь поставленной цели, были выделены следующие **задачи**.

1) Разработка алгоритмов дополнения информации о параллелизме, выявляемой SAPFOR.
2) Разработка алгоритмов перевода информации о параллелизме из модели вычислений, используемой в SAPFOR, в модель вычислений, используемую в LuNA.
3) Разработка алгоритмов доопределения этой информации, чтобы по ней можно было сгенерировать LuNA-программу.

4) Разработка алгоритмов генерации LuNA-программ.
5) Реализация алгоритмов генерации LuNA-программ и их экспериментальное исследование на примере прикладной задачи.

Решение поставленных задач должно обеспечивать автоматическое конструирование эффективной LuNA-программы из последовательной программы, реализующей метод Якоби, быть расширяемым и способным конструировать LuNA-программы для других, похожих на метод Якоби, задач.

1 Обзор предметной области

Перевод информации между моделями вычислений — это задача, которую в том или ином виде должны решать все системы конструирования программ и машинных кодов. К классам таких систем можно отнести компиляторы, системы конструирования и системы распараллеливания.

Так, принцип работы современных компиляторов (например, LLVM [15]) заключается в том, что они сначала выполняют перевод программы из модели вычислений высокоуровневого языка программирования в модель вычислений промежуточного представления, над которым работает оптимизатор. Затем выполняется перевод промежуточного представления программы в машинный код для целевой архитектуры, т.е. кодогенерация. При этом во время работы оптимизатора могут строится дополнительные представления программы, которые лучше подходят для некоторых видов оптимизаций и приведений к каноничному виду, чем промежуточное представление (в качестве примера можно привести дополнительное промежуточное представление SCEV — Scalar Evolution, которое является частью LLVM и предназначено для оптимизаций целочисленных выражений и циклов).

Системы автоматического конструирования параллельных программ должны выполнять перевод спецификации параллельной программы в модель параллельного программирования. При этом в качестве спецификации может выступать последовательная программа или программа на специально разработанном для этого языке программирования, а параллельная программа может конструироваться в терминах OpenMP [16], MPI [17], CUDA [18], Pthreads или любой другой модели параллельного программирования. Это не влияет на то, что суть этой задачи — перевод информации из одной модели вычислений в другую.

Рассмотрим существующие на данный момент средства, выполняющие перевод информации из одной модели вычислений в другую, с точки зрения их способности выполнять работу преобразователя. При этом во внимание будут приниматься следующие характеристики.

- Какие модели вычислений рассматриваемые средства используют в качестве входного и выходного представления.
- Какие у этих средств области применения.
- Какие конкретно алгоритмы пригодны для использования в преобразователе, а какие — нет.

1.1 Полиэдральная модель

В полиэдральной модели (англ. — polyhedral model) [19, 20] гнезда циклов программы представляются в виде матричных выражений. При этом существуют следующие ограничения на задачи, которые можно описывать в рамках этого представления.

- Данные, над которыми ведется работа, имеют примитивные числовые типы.
- Циклы являются арифметическими, т.е. у их индукционных переменных есть верхние и нижние границы, при этом сам цикл гарантированно завершает свое выполнение за конечное время. Границы циклов и индексы массивов — либо константы, либо представляют собой линейную комбинацию счётчиков внешних циклов и констант.
- Внутри циклов нет ветвления.

Полиэдральные модели в процессе своей работы используют такие компиляторы, как PoCC [21], Polly [22], PLUTO, которые могут распараллеливать последовательные программы. Эти системы сначала строят полиэдральную модель исходя из последовательной программы, затем выполняют над ней анализ и ряд преобразований (например, cache tilling, loop unrolling, loop padding [19, 20]), затем, исходя из преобразованной полиэдральной модели, конструируется результирующая программа (может быть, параллельная).

Исходя из вышеописанного процесса можно сделать вывод, что для выполнения части задач преобразователя (например, преобразования циклов) использование полиэдральных моделей подходит, и можно было бы использовать представление программ в этой модели, как одно из промежуточных представлений преобразователя. Недостаток такого подхода заключается в том, что многие, имеющиеся на данный момент, средства, составляющие полиэдральные модели, способны получать информацию об исходной программе, только анализируя эту программу. Использование дополнительной информации, например, выявленной системой распараллеливания, напрямую не представляется возможным, из-за чего теряется смысл использования системы распараллеливания. Помимо этого, если использовать полиэдральную модель в качестве промежуточного представления преобразователя, то нужно будет решать дополнительную задачу преобразования этого представления в представление системы конструирования.

1.2 MapReduce, Hadoop

MapReduce [23] — это фреймворк, который является частью экосистемы Hadoop [24], он предназначен для параллельного выполнения однотипных операций над данными больших объемов (1 Тб и более). При работе с этим фреймворком пользователю необходимо

только определить последовательность чередующихся процедур map и reduce. Процедура map обрабатывает пару ключ/значения и вырабатывает множество промежуточных пар ключ/значение. Процедура reduce объединяет эти промежуточные пары для получения конечного значения, ассоциированного с конкретным ключом. Определение этих процедур может приводится на традиционном языке программирования. Исходя из определения этих процедур, система строит программу, суть которой заключается в выполнении последовательности множеств параметризованных процедур map и reduce.

Использовать MapReduce в качестве преобразователя напрямую не получится, так как у них разные входные представления: у MapReduce — определения процедур map и reduce, а у преобразователя — последовательная программа и информация о ее параллелизме. Помимо этого, ввиду особенности модели вычислений MapReduce, системе, конструирующей распределенную программу, в отличие от преобразователя, не приходится учитывать информационные зависимости или упорядоченность операций, кроме той, которая явно заложена в описании изначальной последовательности.

1.3 Распараллеливание программ на основе директив

Существуют системы, в которых информация о параллелизме выражается пользователем вручную с помощью встраивания в последовательную программу специальных директив, исходя из которых система конструирует параллельную программу. В качестве примера таких систем можно выделить DVM, OpenMP, Cilk, Cilk++ [25].

Поскольку возможно распараллеливание программ на основе директив, то может быть на их основе возможно и составление спецификаций для систем конструирования. Т.е. можно рассматривать подход, при котором преобразователь будет оперировать не информацией о параллелизме, а программой с расставленными в ней директивами. Применимость такого подхода в преобразователе зависит от того, есть ли у используемой системы распараллеливания возможность автоматического встраивания в программу директив. Если такой возможности нет, то для реализации вышеописанного подхода ее придется добавить. Недостатком такого подхода является то, что в общем случае при выражении информации о параллелизме с помощью директив часть этой информации неизбежно теряется, что негативно сказывается на качестве результирующей параллельной программы.

1.4 LLVM IR

Одним из самых популярных на текущий момент промежуточных представлений программ в компиляторах является LLVM IR [26]. Оценим применимость использования

LLVM IR в качестве промежуточного представления преобразователя. В списке ниже приведены преимущества такого подхода.

1) Представление LLVM IR можно построить для программы на любом языке программирования, для которого есть конструирующий его модуль (frontend).
2) Для LLVM IR существует обширная база реализованных оптимизаций и других преобразований, которые могли бы закрыть некоторые потребности преобразователя.

В списке ниже приведены недостатки вышеописанного подхода.

1) Представление LLVM IR строится исходя только из кода программы, т.е. использовать информацию о параллелизме для улучшения его качества напрямую не получится.
2) Для составления спецификации для системы конструирования придется выполнять перевод из LLVM IR в модель вычислений этой системы.

1.5 Выводы

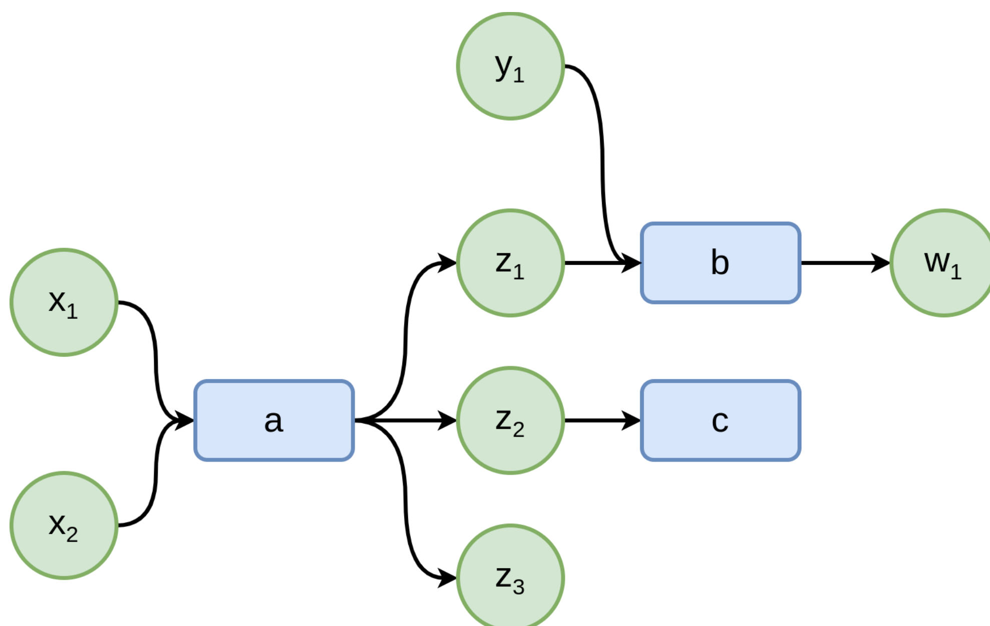
Обзор родственных работ позволяет сделать следующие выводы. Во-первых, перевод информации между моделями вычислений является актуальной задачей, которая решается в системах разных классов, таких как компиляторы, системы автоматического конструирования программ и т.д. Во-вторых, на данный момент задача перевода информации между двумя моделями вычислений не имеет какого-то универсального решения, в том числе не имеет универсального решения и задача создания преобразователя. Это означает, что системы автоматического конструирования параллельных программ еще не могут полностью удовлетворять всем потребностям пользователей ЭВМ.

2 Алгоритмическое и информационное обеспечение преобразователя для систем SAPFOR и LuNA

В главе прорабатываются теоретические вопросы создания преобразователя для систем LuNA и SAPFOR. Глава структурирована следующим образом. В разделе 2.1 описывается модель вычислений, на которой основана система LuNA, а также устройство LuNA-программы. В разделе 2.2 приводится описание системы SAPFOR. В разделе 2.3 описываются основные проблемы, которые должен решать преобразователь для систем SAPFOR и LuNA. В разделе 2.4 излагается предлагаемый подход, на основе которого будет работать преобразователь, в этом же разделе кратко описываются основные модули и представления информации, которые будут реализовывать этот подход. В разделе 2.5 описываются основные принципы конструирования LuNA-программы, которые используются преобразователем. В разделе 2.6 описываются алгоритмы и представление информации, предназначенные для генерации LuNA-программы. В разделе 2.7 описываются алгоритмы и представление информации, предназначенные для определения информации, исходя из которой генерируется LuNA-программа. В разделе 2.8 описывается, каким образом выполняется сбор информации о параллелизме. Завершается глава обсуждением полученных результатов и возможностей дальнейшего развития подхода (раздел 2.9).

2.1 Система LuNA и модель фрагментированного программирования

LuNA (**L**ang**u**age for **N**umerical **A**lgorithms) — это система фрагментированного программирования [27] и язык программирования. Программа на языке LuNA, как и программы на многих других языках программирования, описывает множество вычислений и данных. Данные в LuNA представлены в виде переменных единственного присваивания, которые называются *фрагментами данных*. Вычисления представлены с помощью операций — применения программного модуля к фрагментам данных. Под программным модулем здесь понимается независимая и функционально законченная часть программы, например, последовательная процедура. В LuNA операции называются *фрагментами вычислений*, а программные модули — *фрагментами кода*. Упрощенно LuNA-программа может быть представлена в виде двудольного ориентированного графа, в котором доли соответствуют множеству фрагментов данных и множеству фрагментов вычислений (см. пример на рисунке 3). Если дуга выходит из фрагмента данных и входит во фрагмент вычислений, то этот фрагмент данных является для этого фрагмента вычислений входным. Если дуга выходит из фрагмента вычислений и входит во фрагмент данных, то этот фрагмент данных является для этого фрагмента вычислений выходным.



Кругами обозначены фрагменты данных, прямоугольниками — фрагменты вычислений.

Рисунок 3 — пример LuNA-программы

LuNA-программы — это последовательность объявлений фрагментов кода.

Фрагмент кода может быть определен как на традиционном языке программирования, например, C++ (в таком случае он называется *атомарным*), так и на самом языке LuNA (в таком случае он называется *структурированным*).

Атомарный фрагмент кода обычно представляет из себя последовательную процедуру. В рамках этой процедуры параметры, которые соответствуют входным фрагментам данных, имеют тип `InputDF&`, выходным — `OutputDF&`.

В атомарных фрагментах кода можно использовать механизм *уничтожающего потребления* [28], этот механизм позволяет инициализировать фрагмент данных с помощью буфера другого фрагмента данных, при этом второй фрагмент данных перестает существовать.

В атомарных фрагментах кода N-мерные массивы представляются как одномерные. Причины возникновения такого подхода не так важны, важен сам факт того, что во всех фрагментах кода массивы, соответствующие фрагментам данных, должны быть представлены как одномерные.

Структурированный фрагмент кода состоит из опционального объявления базовых имен фрагментов данных и операторов, описывающих фрагменты вычислений. Оператор может являться:

— описанием применения фрагмента кода к конкретным аргументам;

— массовым или условным оператором, предназначенным для описания потенциально бесконечного множества фрагментов вычислений.

В качестве выражений в структурированном фрагменте кода могут выступать константы строковых, вещественных или целочисленных типов, фрагменты данных и выражения над ними. Фрагменты данных описываются с помощью базовой (символьной) и индексной частей, где

— базовая часть объявляется в начале фрагмента кода, на нее накладываются те же ограничения, что и на имена переменных в традиционных языках программирования;
— индексная часть состоит из 0 или более индексов, каждый из которых представлен с помощью выражения, результат вычисления которого — целое число.

Таким образом, имя фрагмента данных выглядит следующим образом: `<base>[<id0>]...[<idN>]`, где

— `<base>` — базовая часть имени;
— `<id0>`, …, `<idN>` — индексная часть имени.

В качестве аргументов, передаваемых во фрагменты кода, могут выступать выражения, которые имеют типы соответствующих параметров применяемых фрагментов кода.

Выполнение LuNA-программы начинается с выполнения структурированного фрагмента кода `main`, который далее будет называться *головной подпрограммой*.

2.1.1 Массовые операторы

Массовые операторы, представленные в LuNA [29], описаны в следующем списке.

1) Оператор примитивно-рекурсивного перечисления `for`, который позволяет определить множество фрагментов, и размер этого множества задается параметрически. Оператор имеет вид, представленный в листинге 1.

Листинг 1 — Массовый оператор `for`

```
for <counter> = <first_expr> .. <last_expr> <body>
```

В листинге 1:

— `<counter>` — имя параметра, например, `i` или `step` (далее будет называться *счетчиком*);
— `<first_expr>` — выражение, определяющее минимальное значение счетчика цикла `<counter>`;

— <last_expr> — выражение, определяющее максимальное значение счетчика цикла <counter>;
— <body> — тело оператора.

Оператор for предписывает вычислить значения выражений <first_expr> и <last_expr>, после чего породить все фрагменты вычислений, определенные в его теле в нескольких экземплярах, по одному экземпляру для каждого целочисленного значения от <first_expr> до <last_expr> включительно. При порождении каждого экземпляра тела оператора параметр <counter> будет определён и будет иметь уникальное целочисленное значение в указанном диапазоне.

2) Оператор частично-рекурсивного пересчиления while. Оператор имеет вид, представленный в листинге 2.

Листинг 2 — Массовый оператор while

```
while <cond_expr> , <counter> = <start_expr> .. out <out_df_id>
<body>
```

В листинге 2:
— <cond_expr> — условное выражение;
— <counter> — имя счетчика итераций;
— <start_expr> — начальное значение счётчика;
— <out_df_id> — имя фрагмента данных, он является выходным параметром оператора;
— <body> — тело оператора.

Оператор while предписывает вычислить значение выражения <start_expr>. При этом заданном значении счётчика <counter> вычисляется значение условного выражения <cond_expr>. Если значение ложно, то значение фрагмента данных <out_df_id> устанавливается равным значению <counter> и оператор считается выполненным. Если значение истинно, то порождается тело оператора, в контексте которого так же определено значение счётчика <counter>, а также порождается новый оператор while как копия старого, но со значением <start_expr> на единицу большим, чем было. Таким образом, процесс вычислений продолжается до тех пор, пока условное выражение не примет ложное значение.

3) Условный оператор if, который позволяет описывать фрагменты, существующие только в случае, если некоторое условие выполняется. Оператор имеет вид, представленный в листинге 3:

Листинг 3 — Условный оператор if

```
if <cond_expr> <body>
```

В листинге 3:

- <cond_expr> — условное выражение;
- <body> — тело оператора.

Оператор if предписывает вычислить значение выражения <cond_expr>. Если логическое значение ложно, то оператор считается исполненным. Если истинно, то тело оператора порождается.

2.1.2 Оператор >>

Помимо вышеописанных массовых операторов в LuNA есть еще оператор >>. Оператор имеет вид, представленный в листинге 4.

Листинг 4 — Оператор >>

```
<code_fragment>(...) >> (<data_fragment>);
```

В листинге 4:

- <code_fragment> — некий фрагмент кода;
- <data_fragment> — некий фрагмент данных.

Оператор предписывает инициализировать фрагмент данных <data_fragment> истинным значением после того, как применение фрагмента кода <code_fragment> будет выполнено.

2.1.3 Пример LuNA-программы

Пример структурированного фрагмента кода приведен в листинге 5.

Листинг 5 — Пример программы на языке LuNA

```
01: import init(name x);
02: import calc(int x, name y);
03: import print(int x);
04: // Объявление головной подпрограммы
05: sub main() {
06:   df x; // объявление базового имени ФД
07:   for i = 1 .. 10 {
08:     calc(x[i-1], x[i]); // определение множества ФВ
09:   }
10:   print(x[10]); // определение безымянного ФВ
```

Продолжение листинга 5

```
11:   init(x[0]); // определение одиночного ФВ
12: }
```

В листинге 5 в строках 01—03 объявлены атомарные фрагменты кода, которые должны быть определены пользователем на языке C++. Параметр типа name является выходным, остальные — входные. В строке 06 листинга объявлено базовое имя ФД — `x`. В строках 07-09 определен массовый оператор `for`, в теле которого описан вызов фрагмента кода `calc`. В строках 10 и 11 описаны вызовы фрагментов кода `print` и `init` соответственно.

2.2 Система SAPFOR для автоматизации распараллеливания программ

Система автоматизированного распараллеливания программ **SAPFOR** (**S**ystem **FOR** **A**utomate **P**arallelization) включает набор различных инструментов, разрабатываемых с целью снижения сложности и уменьшения времени создания параллельных программ на языках программирования Fortran и C++.

Основные возможности этой системы перечислены в следующем списке.

- Исследование характеристик и свойств программы (профилирование, анализ зависимостей по данным и др.).
- Автоматическое распараллеливание потенциально параллельных программ.
- Автоматизированное преобразование программ к потенциальному параллельному виду. Устранение зависимостей в программе, оптимизация доступа к памяти, изменение структуры хранения данных и структуры вычислений (подстановка процедур и переменных, преобразование циклов и др.).

2.3 Проблемы конструирования LuNA-программы

В работе будет решаться задача автоматического конструирования LuNA-программ из последовательных программ на языке C++.

Для того, чтобы сконструировать LuNA-программу, преобразователь должен решить следующие проблемы.

1) Выявленная SAPFOR информация о параллелизме исходной последовательной программы в общем случае недостаточна для того, чтобы по ней можно было сконструировать LuNA-программу, поэтому ее необходимо дополнить.
2) SAPFOR оперирует в модели вычислений потока управления (англ. — control flow), LuNA — в модели вычислений фрагментированного программирования. Поэтому

выявленную и дополненную информацию о параллелизме нужно перевести в модель вычислений, используемую в LuNA, а также доопределить эту информацию до такой степени, чтобы исходя из нее можно было сгенерировать LuNA-программу.

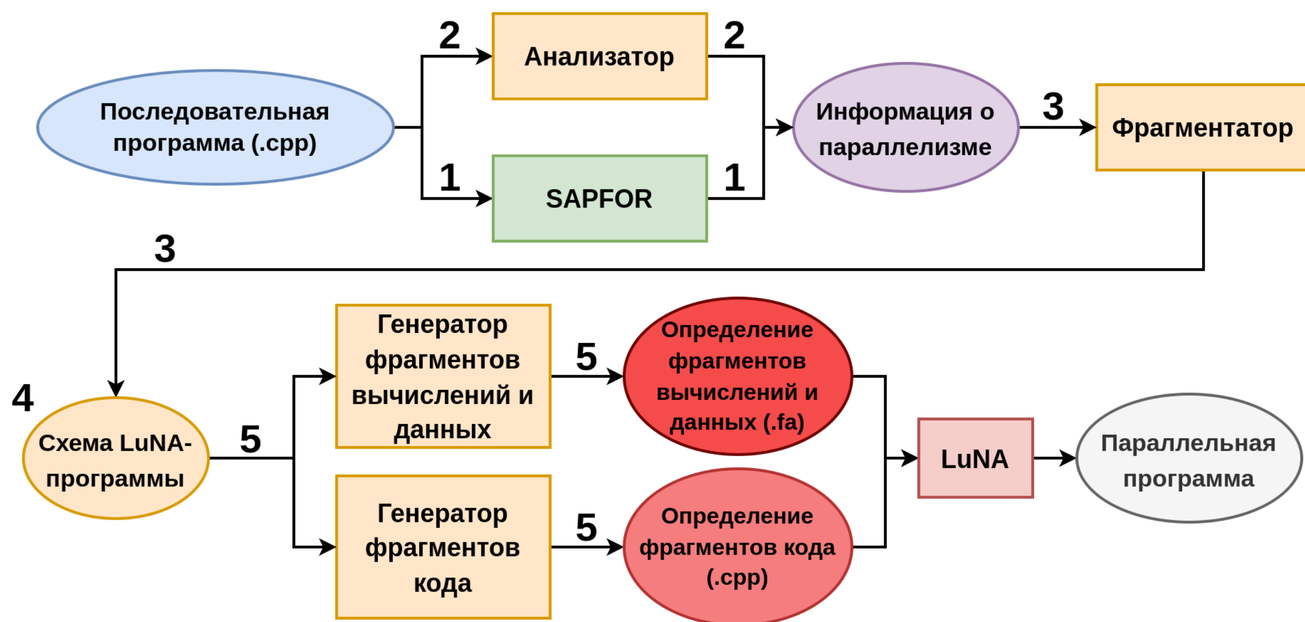
3) После этого по вышеописанной информации нужно непосредственно сгенерировать LuNA-программу.

2.4 Предлагаемый подход к автоматическому конструированию LuNA-программ

Исходя из вышеописанных проблем предлагается следующий подход к автоматическому конструированию LuNA-программ:

1) SAPFOR выявляет из исходной последовательной программы *информацию о параллелизме*.
2) Эту информацию дополняет модуль, который называется *анализатором*.
3) Информация о параллелизме обрабатывается *фрагментатором*. Его задачей является перевод информации из модели вычислений, используемой в SAPFOR, в модель вычислений, используемой в LuNA, а также принятие ряда решений по доопределению этой информации. Результатом работы фрагментатора является представление информации необходимой для генерации LuNA-программы. Это представление называется схемой LuNA-программы. Далее процесс работы фрагментатора будет называться *фрагментацией последовательной программы*.
4) *Схема LuNA-программы* содержит всю принципиальную информацию о том, как должна быть составлена LuNA-программа, без технических деталей.
5) По этой схеме *генератор фрагментов вычислений и данных* создает определения фрагментов вычислений и данных, а *генератор фрагментов кода* создает определение атомарных фрагментов кода. Т.е. задачей генераторов является непосредственно создание LuNA-программы. Далее генератор фрагментов вычислений и данных и генератор фрагментов кода вместе будут называться *генераторами*, а процесс их работы будет называться *генерацией LuNA-программы*.

Предлагаемый подход схематично представлен на рисунке 4. На этом рисунке дуга, выходящая из модуля и входящая в представление информации, означает, что этот модуль вырабатывает это представление, а дуга, выходящая из представления и входящая в модуль, означает, что этот модуль потребляет это представление. Номера действий и элементов на схеме соответствуют номерам элементов списка приведенного выше. Оранжевым цветом выделены части преобразователя.



Прямоугольниками обозначены модули, овалами — представления информации.

Рисунок 4 — схема автоматического конструирования LuNA-программ

Для реализации этого решения нужно выполнить следующие теоретические задачи:

1) Определить структуру информации о параллелизме.
2) Определить структуру схемы LuNA-программы.
3) Разработать алгоритмы фрагментатора.
4) Разработать алгоритмы генераторов.

Структура следующих разделов этой главы такая: сначала описываются некоторые общие предлагаемые принципы конструирования LuNA-программы, далее подробно описан каждый этап работы преобразователя. Это описание проще излагать в обратном порядке:

1) Генерация LuNA-программы, т.е. структура схема LuNA-программы и алгоритмы генераторов.
2) Фрагментация, т.е. структура информации о параллелизме и алгоритмы фрагментатора.
3) Сбор информации о параллелизме.

2.5 Принципы конструирования LuNA-программы

В подразделе описаны общие принципы конструирования LuNA-программы. Решение о применении тех или иных принципов принимает фрагментатор, непосредственно реализуют их в LuNA-программе — генераторы.

2.5.1 Фрагментация по пространству

Обычно при распараллеливании последовательной программы (автоматическом или ручном) преобразование к параллельному виду сначала выполняется для вычислительноемких и при этом «легко» поддающихся такому преобразованию ее частей. Используя такой подход, можно получить эффективную параллельную программу, затратив относительно немного усилий.

В программах, реализующих алгоритмы численного моделирования, такими вычислительноемкими и легко распараллеливаемыми частями обычно являются тесно-вложенные гнезда циклов, в которых вычисляются элементы некоторого массива, при этом между итерациями хотя бы одного цикла из этого гнезда нет зависимостей по управлению или зависимостей по данным, которые нельзя выразить с помощью редукционных операций, а код, с помощью которого вычисляются значения элементов массива, не имеет побочных эффектов. Пример такого гнезда приведен в листинге 6.

Листинг 6 — Пример вычислительноемкого гнезда циклов

```
1: for (int i0 = 0; i0 < N0; ++i0)
2:   for (int i1 = 0; i1 < N1; ++i1)
3:     ...
4:       for (int iM = 0; iM < NM; ++iM)
5:         // вычисление элемента массива без побочных эффектов
6:         A[i0][i1]...[iM] = calc_no_side_effects(i0, i1, …,
7:           iM);
```

Надо отметить, что здесь под циклом понимается арифметический цикл, т.е. такой цикл, выполнение которого гарантированно завершится за конечное время, а значение его индукционной переменной ограничено нижней и верхней границами.

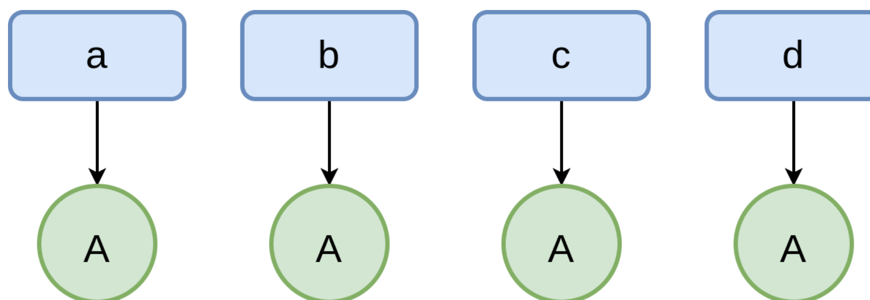
Стандартным методом для получения выгоды от параллелизма для такого гнезда циклов является составление разбиения множества его итераций и выполнение каждого множества из этого разбиения в отдельной единице исполнения (например, в отдельном потоке). Подобный способ параллельного исполнения циклов есть, например, в OpenMP (с помощью директивы `omp parallel for`). Предлагается использовать аналогичный подход для модели вычислений LuNA, который далее будет называться *фрагментацией по пространству*. При фрагментации по пространству каждое множество итераций из вышеописанного разбиения выполняется в отдельном фрагменте вычислений. Далее про вышеописанные гнезда циклов будем говорить, что их *можно фрагментировать по пространству*.

Рассмотрим пример фрагментации по пространству на примере двумерного гнезда циклов. Исходное гнездо циклов представлено в листинге 7.

Листинг 7 — Пример вычислительноемкого гнезда циклов

```
1: for (int i0 = 0; i0 < 100; ++i0)
2:   for (int i1 = 0; i1 < 100; ++i1)
3:     A[i0][i1] = calc_no_side_effects(i0, i1);
4:     // вычисление элемента массива без побочных эффектов
```

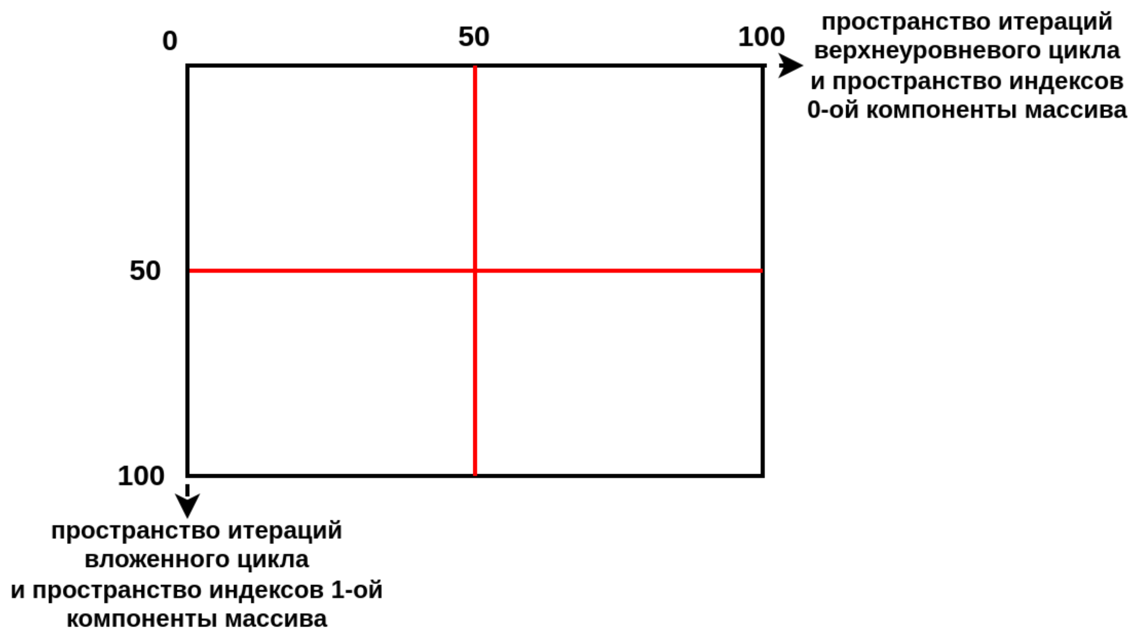
В рамках примера рассмотрим двумерную фрагментацию (т.е. фрагментироваться будут оба цикла из гнезда), в которой каждый цикл фрагментируется на две равные по количеству итераций части. Схема такой фрагментации приведена на рисунке 5. Поскольку части фрагментируемых гнезд циклов будут выполняться в разных фрагментах вычислений, то вырабатываемый в рамках этого гнезда массив A будет представлен с помощью нескольких фрагментов данных, а не одного, т.е. фрагментации по пространству подвержены не только циклы, но массивы. Так, в рассматриваемом примере в соответствии с фрагментацией цикла на 4 части выполняется аналогичная фрагментация массива на 4 части, каждая из которых содержит $(50 \cdot 50)$ элементов исходного массива.



Синими прямоугольниками обозначены фрагменты вычислений, в которых выполняется гнездо циклов, зелеными кругам — части массива A.

Рисунок 5 — Схема двумерной фрагментации по пространству гнезда циклов из листинга 7

Схема разбиения пространства итераций гнезда циклов и пространства индексов массива приведена на рисунке 6.



Числа на осях соответствуют границам, по которым выполняется фрагментация по пространству. Каждый цикл и каждая компонента массива фрагментируются на две части.

Рисунок 6 — Фрагментация по пространству двумерного гнезда циклов и двумерного массива для примера из листинга 7

Далее *осью* будет называться неотрицательное целое число, которому соответствует 0 или более циклов и 0 или более компонент массивов. Если выполняется фрагментация по пространству этих циклов или компонент массива, то оси также соответствует количество фрагментов вычислений, на которое эта фрагментация выполняется, такие оси будут далее называться *фрагментируемыми*.

Если оси *N* соответствует количество фрагментов вычислений *k*, то будем говорить, что *ось N фрагментируется на k фрагментов*, а также, что *размер фрагментации оси* N равен k.

Далее, если обратное не обозначено явно, будет пониматься, что i-ой оси соответствуют только тот цикл, который в гнезде циклов находится на уровне вложенности i (считается, что они нумеруются начиная с 0) и только i-ая компонента массива (считается, что они нумеруются начиная с 0).

В случае, когда оси k соответствует только один цикл в гнезде, то под *k-ой осью гнезда циклов* будет пониматься этот цикл. В случае, когда оси k соответствует только одна компонента массива, то под *k-ой осью массива* будет пониматься эта компонента.

Далее *размерностью фрагментации по пространству гнезда циклов* будет называться количество фрагментируемых по пространству циклов в этом гнезде, а *размерностью фрагментации по пространству массива* — количество фрагментируемых по пространству компонент этого массива.

Далее *фрагментом цикла* или *гнезда циклов* будет называться часть фрагментируемого по пространству цикла или гнезда циклов, которая выполняется в рамках одного фрагмента вычислений, а *фрагментом массива* — часть массива, которая вырабатывается в рамках одного фрагмента гнезда циклов.

Далее *пространственным фрагментом вычислений* будет называться фрагмент вычислений, который выполняет один фрагмент цикла и вырабатывает один фрагмент массива.

Далее *номером фрагмента вычислений по оси k* будет называться неотрицательное число в отрезке от 0 до уменьшенного на единицу размеру фрагментации оси k, не равное номеру ни одного другого фрагмента вычислений по оси k. Далее *номером фрагмента гнезда циклов по оси k* будет называться номер фрагмента вычислений по оси k, в рамках которого выполняется этот фрагмент. Далее *номером фрагмента массива по оси k* будет называться номер вырабатывающего его фрагмента вычислений по оси k.

Далее *номером фрагмента вычислений* будет называться кортеж, длина которого равна размерности фрагментации, а i-ый элемент — номеру этого фрагмента по оси i, *номером фрагмента гнезда циклов* — номер выполняющего его фрагмента вычислений, а *номером фрагмента массива* — номер вырабатывающего его фрагмента вычислений.

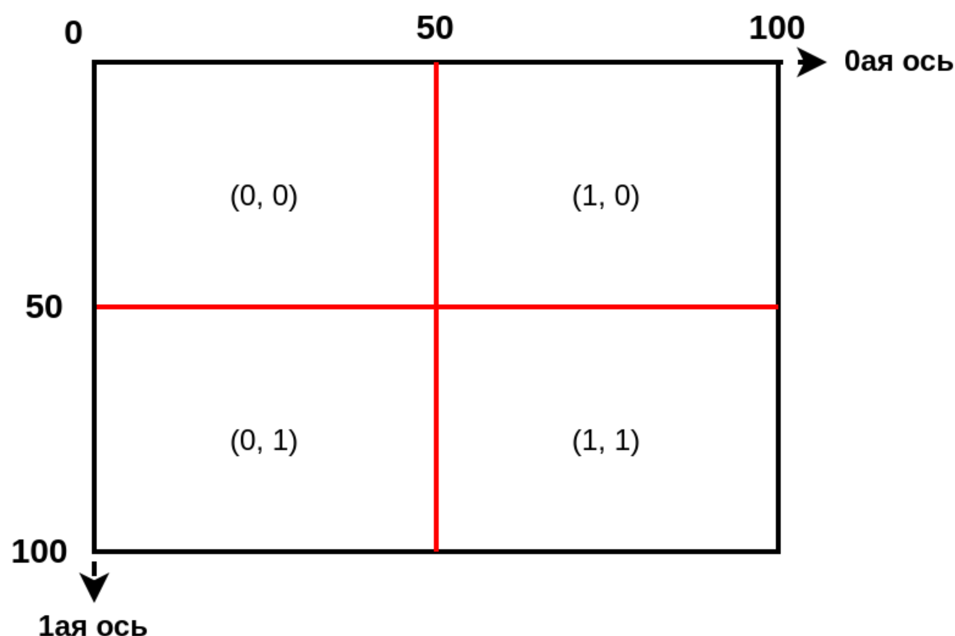
Зададим над номерами фрагментов вычислений отношение частичного порядка: номер фрагмента N1 ≤ номер фрагмента N2, если существует такое неотрицательное число k, что номер фрагмента N1 по оси k меньше, чем номер фрагмента N2 по оси k, и для любого i < k номер фрагмента N1 по оси i равен номеру фрагмента N2 по оси i.

Если номер фрагмента вычислений `a` по оси k меньше, чем номер фрагмента вычислений `b` по оси k, то в рамках фрагмента `a` выполняются итерации фрагментируемых гнезд циклов, которые соответствуют меньшему значению индукционной переменной k-ого цикла исходного гнезда, чем итерации, выполняемые в рамках фрагмента `b`.

Если номер фрагмента вычислений `a` по оси k меньше, чем номер фрагмента вычислений `b` по оси k, то в рамках фрагмента `a` вырабатываются элементы фрагментируемых массивов, которые соответствуют меньшим значениям индекса k-ой компоненты исходного массива, чем элементы, вырабатываемые в рамках фрагмента `b`.

Фрагментация по пространству может выполняться для массивов и циклов произвольных размерностей, при этом размерность фрагментации массива может варьироваться от 1 до размерности массива, а размерность фрагментации гнезд циклов — от 1 до количества циклов в этом гнезде.

На рисунке 7 представлена схема фрагментации гнезда циклов из листинга 7 с учетом введенных определений.



Четыре прямоугольника обозначают четыре фрагмента вычислений, надписи на них — номера этих фрагментов.

Рисунок 7 — Двумерная фрагментация двумерных гнезда циклов и массива для примера из листинга 7

При определении фрагментов кода, содержащих фрагментируемые по пространству циклы и/или массивы, нужно выполнять дополнительные действия, такие, как пересчет границ циклов, размеров массивов и т.д. То, каким образом это выполняется, описано в подразделе 3.1.1. Вышеописанные фрагменты кода далее будут называться *пространственными*.

Далее будем считать, что пространственный фрагмент кода фрагментируется по всем осям, по котором фрагментируется хотя бы один содержащийся в нем цикл или хотя бы один массив, который этот фрагмент кода принимает в качестве параметра. Далее *размерностью фрагментации пространственного фрагмента кода* будет называться количество фрагментируемых осей это фрагмента кода.

Множество пространственных фрагментов вычислений порождается с помощью гнезда массовых операторов `for`. То, каким образом это делается, изложено в следующем списке.

1) Количество этих массовых операторов равно размерности фрагментации применяемого фрагмента кода.
2) Каждый оператор `for` соответствует одной фрагментируемой оси. Нижняя граница счетчика оператора, соответствующего i-ой оси, равна 0, а верхняя —

уменьшенному на единицу размеру фрагментации оси i. Размер фрагментации может задаваться как статически, так и динамически.

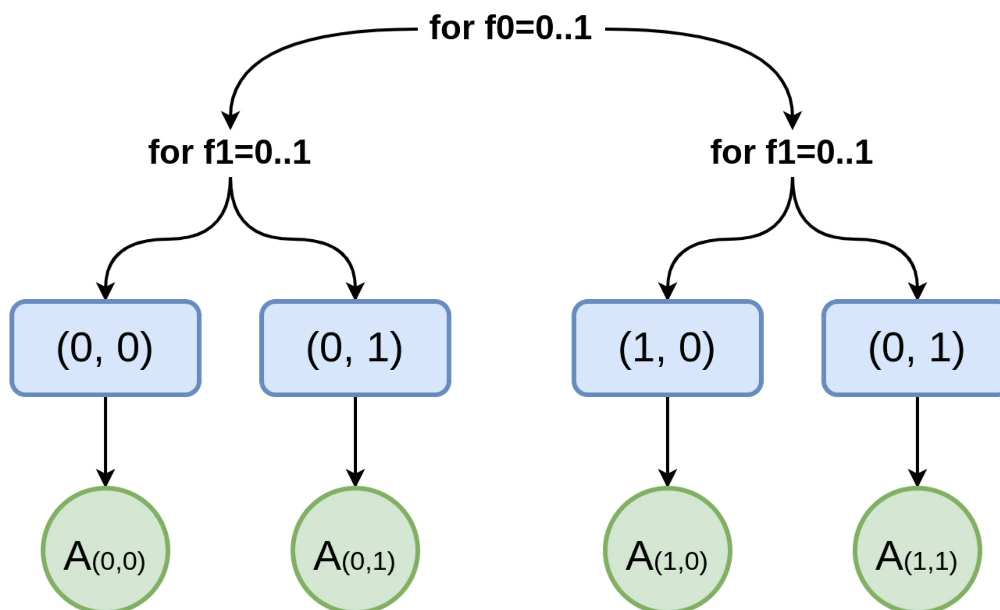
3) Гнездо массовых операторов `for` объемлет описание применения пространственного фрагмента вычислений.
4) Имена аргументов, соответствующих фрагментам массивов, строятся следующим образом. Базовые части этих имен — имена соответствующих массивов из исходной программы. В индексных частях этих имен есть N индексов, где N — размерность фрагментации этих массивов, а i-ый индекс — счетчик объемлющего массового оператора, который соответствует i-ой фрагментируемой оси в порядке возрастания. Такие индексы далее будут называться *пространственными*.

То, каким образом порождается множество пространственных фрагментов вычислений, соответствующих рассматриваемому гнезду циклов из листинга 7, показано в листинге 8.

Листинг 8 — порождение пространственных фрагментов вычислений

```
01: for f0=0..1 // массовый оператор, соответствующий 0-ой оси
02:   for f1=0..1 // массовый оператор, соответствующий 1-ой оси
03:     calc_A(f0, f1, A[f0][f1]); // применение фрагмента кода
04:                                // фрагмент массива A
05:                                // передается с
06:                                // пространственными индексами
07:                                // f0 и f1
```

В листинге 8 всего два массовых оператора (так как размер фрагментации — 2), при этом верхние границы обоих счетчиков — 1 (так как размер фрагментации по обеим осям — 2), `calc_A` — фрагмент кода, который выполняет фрагмент исходного гнезда циклов, фрагмент массива `A` передается в этот фрагмент кода с пространственными индексами `f0` и `f1`. Схема фрагментации в терминах фрагментов вычислений и данных представлена на рисунке 8.



Прямоугольники обозначают фрагменты вычислений, надписи на них — номера этих фрагментов. Круги обозначают порождаемые части массива A, постфиксы надписей на них — номера этих частей.

Рисунок 8 — Двумерная фрагментация двумерного гнезда циклов

Резюмируя, алгоритм описания фрагментации по пространству в LuNA-программе выполняется следующим образом.

1) Определяются фрагменты кода, соответствующие фрагментам циклов (подраздел 3.1.1).
2) С помощью массовых операторов описываются порождения пространственных фрагментов вычислений.

2.5.2 Теневые грани

При выполнении фрагментации по пространству может возникнуть ситуация, когда для выполнения части фрагмента гнезда циклов нужно иметь теневые грани некоторого массива. В данном подразделе описывается, каким образом в таком случае предлагается конструировать LuNA-программу. В качестве примера рассмотрим листинг 9.

Листинг 9 — Пример гнезда циклов, при фрагментации по пространству которого нужны теневые грани

```
01: for (int i = 0; i < N; ++i) // гнездо инициализации
02: {
03:   for (int j = 0; j < M; ++j)
04:   {
05:     A[i][j] = max(t * i, t * j);
06:   }
07: }
```

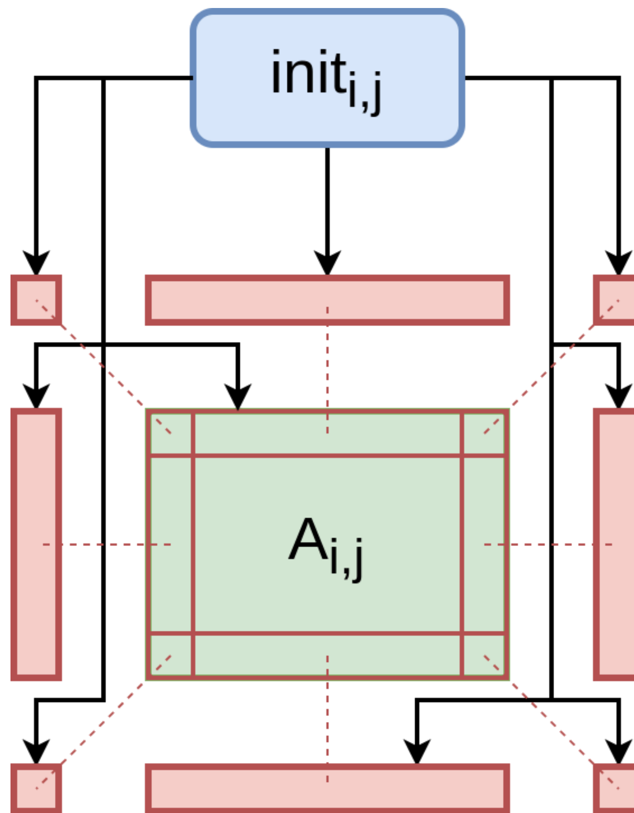
Продолжение листинга 9

```
08:
09: for (int i = 1; i < N - 1; ++i) // гнездо вычислений
10: {
11:   for (int j = 1; j < M - 1; ++j)
12:   {
13:     B[i][j] = A[i - 1][j - 1] + A[i - 1][j + 1]
14:       + A[i + 1][j - 1] + A[i + 1][j + 1];
15:   }
16: }
```

Эта часть кода содержит два гнезда циклов, в одном из которых происходит инициализация элементов некоторого массива `A` (строки 01-07), во втором — вычисление элементов массива `B` (строки 09-16), исходя из значений элементов массива `A`, далее эти гнезда будут называться *гнездом инициализации* и *гнездом вычислений* соответственно. При этом для вычисления элемента `B[i][j]` используются элементы `A[i - 1][j - 1]`, `A[i - 1][j + 1]`, `A[i + 1][j - 1]`, `A[i + 1][j + 1]`. Далее, без ограничения общности, будет считаться, что размерность фрагментации по пространству обоих гнезд циклов двумерная, т.е. фрагментируется каждый цикл.

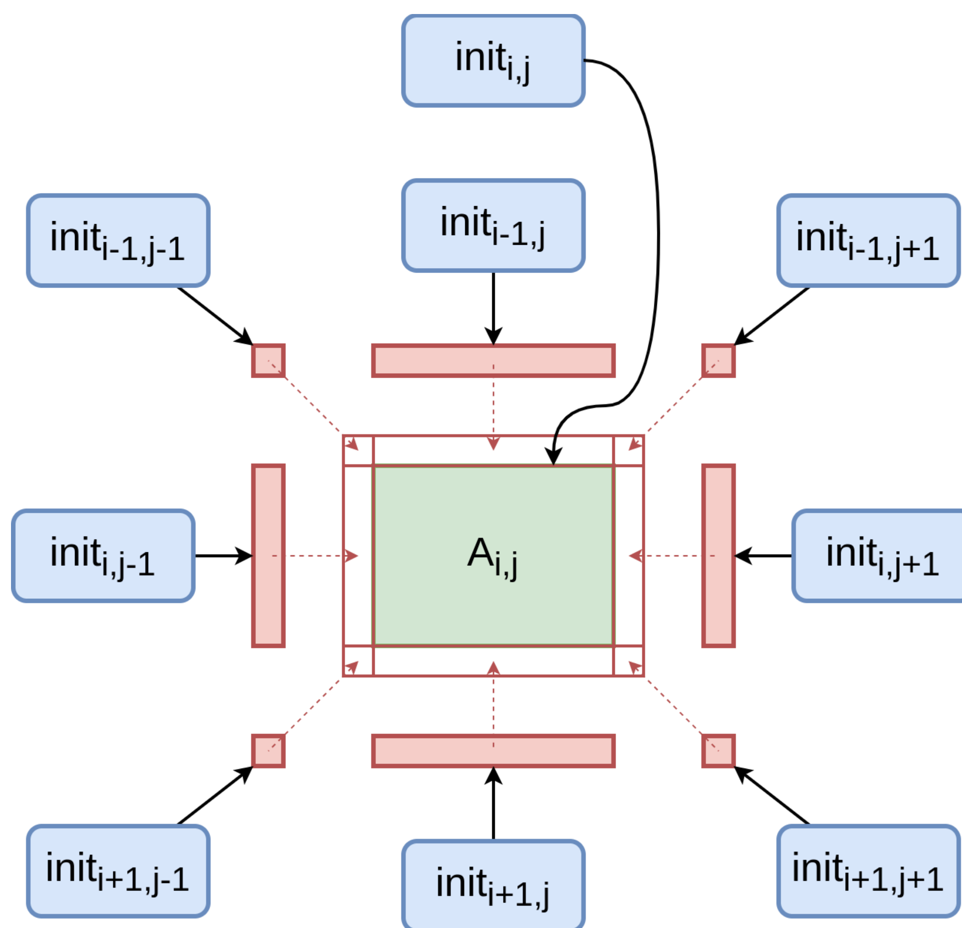
При выполнении фрагментации по пространству циклов из рассматриваемого примера для вычисления фрагментов массива `B` потребуется 8 теневых граней фрагментов массива `A`. С формальным алгоритмом определения того, какие теневые грани нужны для выполнения фрагментируемого по пространству гнезда циклов, можно ознакомиться в приложении А.

Для каждого массива, для которого нужны теневые грани, нужно выполнить их инициализацию и объединение. Под *инициализацией* (для примера из листинга 9 ее схема приведена на рисунке 9) здесь подразумевается инициализация фрагментов данных, соответствующих теневым граням. Эта инициализация должна выполняться фрагментами вычислений, которые вырабатывают соответствующие фрагменты массива, т.е. в рамках рассматриваемого примера — фрагментами вычислений, которые соответствуют гнезду инициализации. Под *объединением* (для примера из листинга 9 его схема приведена на рисунке 10) теневых граней подразумевается выработка фрагмента данных, который соответствует фрагменту массива и его соседним теневым граням, объединенным вместе. Этот объединенный фрагмент данных далее будет передаваться во фрагменты вычислений, где эта теневая грань нужна, т.е. в рамках рассматриваемого примера — во фрагменты вычислений, соответствующие гнезду вычислений.



Синим прямоугольником обозначен фрагмент вычислений с номером (i, j), зеленым — фрагмент данных, соответствующий фрагменту массива A, который вырабатывается этим фрагментом вычислений. Красными закрашенными прямоугольниками обозначены инициализируемые теневые грани, пунктирными линиями показано, каким частям фрагмента массива они соответствуют. Черные стрелки демонстрируют, что все фрагменты данных на этой схеме вырабатываются фрагментом вычислений с номером (i, j).

Рисунок 9 — Инициализация теневых граней двумерного массива для гнезда инициализации из листинга 9



Синими прямоугольниками обозначены фрагменты вычислений, вырабатывающие фрагменты данных, которые соответствуют фрагменту массива и его соседним теневым граням, черными стрелками показано, какие фрагменты вычислений вырабатывают какие фрагменты данных. Пунктирные стрелки показывают, каким образом теневые грани объединяются с фрагментом массива.

Рисунок 10 — Принцип объединения теневых граней и фрагмента массива для гнезда инициализации из листинга 9

Для описания инициализации, объединения и использования теневых граней каждого массива в LuNA-программе применяется следующий алгоритм.

1) Определяются два фрагмента кода, выполняющие инициализацию и объединение теневых граней (см. подраздел 3.1.3).
2) Преобразуются фрагменты кода, в рамках которых теневая грань используется (см. подраздел 3.1.3).
3) Для каждой теневой грани объявляется базовое имя фрагмента данных, также базовые имена фрагментов данных объявляются для необъединенного и объединенного с теневыми гранями фрагментов массива.
4) Во фрагмент кода, выполняющий инициализацию, передаются все фрагменты данных, соответствующие теневым граням, а также фрагмент данных, соответствующий необъединенному фрагменту массива.

5) В рамках того же гнезда массовых операторов `for`, которые порождают применение фрагмента кода, выполняющего инициализацию теневых граней, описывается применение фрагмента кода, выполняющего объединение теневых граней. В него передаются все фрагменты данных, соответствующие теневым граням, а также фрагменты данных, соответствующие объединенному и необъединенному фрагменту массива. В рамках вышеописанного фрагмента вычислений вырабатывается значение фрагмента данных, соответствующего объединенному фрагменту массива.
6) Определяются фрагменты данных, которые передаются в вышеописанные фрагменты вычислений в качестве аргументов, соответствующих теневым граням. Алгоритм, исходя из которого выполняется это определение, описан в приложении Б.
7) Фрагмент данных, соответствующий объединенному фрагменту массива, передается во все фрагменты кода, которые должны использовать фрагмент этого массива.

В приложении В продемонстрировано, каким образом описываются части LuNA-программы, исходя из п. 3-7 вышеописанного алгоритма.

2.5.3 Фрагментация по времени

Поскольку в LuNA фрагменты данных являются переменными единственного присваивания, нужно выработать способ представления множественного присваивания, которое присуще традиционным языкам программирования.

Рассмотрим программу на C++ из листинга 10.

Листинг 10 — Программа, в которой используется множественное присваивание

```
01: int main()
02: {
03:     int a;
04:     a = 3; // запись значения переменной a
05:     for (int i = 0; i < 50; ++i)
06:     {
07:         a *= 2; // чтение и запись значения переменной a
08:     }
09: }
```

В программе из листинга 10 переменной `a` значение присваивается многократно, сначала после ее объявления (строка 04), потом в теле цикла (строка 07). Для моделирования такого поведения в LuNA предлагается принцип, основанный на использовании специальных

индексов фрагментов данных, которые далее будут называться *временны́ми*. Далее этот принцип будет называться *фрагментацией по времени*.

Допустим, при создании LuNA-программы для исходной программы из листинга 10 присваивание после объявления выделено в отдельный фрагмент кода `init_a`, цикл `for` представлен с помощью LuNA-оператора `for`, а его тело вынесено в отдельный фрагмент кода `loop_body`. LuNA-программа, составленная таким образом, представлена в листинге 11.

Листинг 11 — LuNA-программа, соответствующая программе из листинга 10

```
01: sub main() {
02:     df a;
03:     init_a(a[0]);
04:     for i=0..49 {
05:         loop_body(a[1][i - 1], a[1][i]);
06:     }
07: }
```

Количество временных индексов у конкретного аргумента равно увеличенному на единицу уровню вложенности, на котором находится соответствующий оператор (при условии, что уровни вложенности нумеруются начиная с 0). Значение i-го временного индекса равно количеству применений фрагментов кода, которые принимают в качестве выходного аргумента фрагмент данных с тем же базовым именем, что и рассматриваемый, причем эти применения должны быть выполнены до того, как начнут выполняться фрагменты вычислений, описываемые рассматриваемым оператором, и описывающие эти применения операторы находятся на уровне вложенности i. Если фрагмент данных передается во фрагмент кода в качестве входного аргумента, то последний временной индекс будет на единицу меньше, чем описанный выше.

Так, в листинге 11 во фрагмент кода `init_a` передается фрагмент данных `a[0]`, так как до него не должно быть выполнено ни одного применения фрагментов кода. Самые первые временные индексы аргументов, передаваемых во фрагмент кода `loop_body`, равны 1, так как на соответствующем уровне вложенности до них должно быть выполнено применение `init_a`. Последние временные индексы этих аргументов равны `(i - 1)` и `i` для входного и выходного аргумента соответственно, так как на рассматриваемом уровне вложенности до применения фрагмента кода `loop_body` при значении счетчика `i` должно быть выполнено `i` применений фрагмента кода `loop_body`.

Формальный способ определения временных индексов приведен в приложении Г.

Но программа, предъявленная в листинге 11, является некорректной, так как фрагмент вычислений `loop_body`, порождаемый при значении счетчика `i = 0`, использует в качестве входного аргумента фрагмент данных, который никогда не будет инициализирован: `a[1][-1]`. Чтобы сделать программу корректной, нужно до и после всех массовых операторов, в которых есть хоть один оператор, описывающий фрагмент вычислений, потребляющий фрагмент данных с рассматриваемым базовым именем, описать уничтожающее потребление этого фрагмента данных. Преобразованная таким образом программа из листинга 11 приведена в листинге 12.

Листинг 12 — LuNA-программа, соответствующая программе из листинга 10

```
01: sub main() {
02:     df a;
03:     init_a(a[0]);
04:     destructive_consumption(a[0], a[1][-1]); // ун. потр.
05:     for i=0..49 {
06:         loop_body(a[1][i], a[1][i-1]);
07:      }
08:     destructive_consumption(a[1][49], a[2][-1]); // ун. потр.
09: }
```

Фрагмент кода `destructive_consumption` выполняет уничтожающее потребление первого аргумента вторым. Как можно заметить, применения `destructive_consumption` описываются до и после LuNA-оператора `for`. Вследствие этого преобразования LuNA-программа становится корректной, так как фрагмент данных `a[1][-1]` инициализируется фрагментом вычислений, описанным до LuNA-оператора `for`.

Замечение: уничтожающее потребление фрагмента данных после выполнения массового оператора необходимо, только если после порождаемых этим оператором фрагментов вычислений должен быть выполнен хотя бы один другой фрагмент вычислений, использующий этот фрагмент данных. В рассматриваемом примере таких фрагментов вычислений нет, и уничтожающее потребление не является обязательным, тем не менее в программе оно присутствует для демонстрации того, как это делается в общем случае.

Резюмируя, алгоритм описания фрагментации по времени в LuNA-программе выполняется следующим образом.

1) Исходя из порядка следования операторов в исходной программе определяется в каком порядке должны выполняться фрагменты вычислений.
2) Для каждого фрагмента данных определяются его временные индексы. Алгоритм определения временных индексов приведен в приложении Г.

2.6 Генерация LuNA-программы

В этом разделе приведены описание алгоритмов генератора фрагментов вычислений и данных и генератора фрагментов кода, а также описание структуры входных данных, которыми эти генераторы оперируют, т.е. описание схемы LuNA-программы. В процессе своей работы генераторы могут определять дополнительные процедуры, фрагменты кода и/или фрагменты данных, при этом могут возникнуть конфликты имен. Далее будет считаться, что все имена уникальные и конфликтов имен не происходит, так как их разрешение на практике является чисто техническим вопросом.

2.6.1 Схема LuNA-программы

В подразделе описаны основные структурные элементы схемы LuNA-программы, в следующих подразделах это описание будет по мере необходимости уточняться.

Для того, чтобы создать определения фрагментов кода, генератор фрагментов кода должен обладать принципиальной информацией об их организации. Т.е. в схеме LuNA-программы для каждого фрагмента кода должна быть информация о его сигнатуре, т.е. о его имени и о параметрах, которые он принимает; о его теле, а также о его фрагментации по пространству. Тело описывается в виде последовательности блоков, каждый из которых может являться:

- доступом к элементу массива;
- выделением памяти под массив;
- вызовом процедуры;
- циклом;
- последовательным участком кода, который не подпадает ни под одну вышеописанную категорию.

Также схема LuNA-программы содержит описание процедур, которые вызываются в рамках фрагментов кода. Они описываются практически так же, как и фрагменты кода.

Для того, чтобы сконструировать определения фрагментов вычислений и данных, генератор фрагментов вычислений и данных должен обладать информацией об их принципиальной организации. Т.е. в схеме LuNA-программы должна содержаться информация о применениях фрагментов кода, базовых именах фрагментов данных и массовых операторах.

Для каждого применения фрагмента кода содержится информация о том, какой фрагмент кода ему соответствует, и о передаваемых в него аргументах.

Для каждого массового оператора содержится информация о его теле (т.е. о фрагментах вычислений и данных, которые он должен породить), о верхней и нижней границе его счетчика, а также о наличии дополнительных условий порождения содержащихся в нем операторов и фрагментов вычислений, которые вытекают из наличия в циклах исходной программы выражений влияющих на граф потока управления (например, `break`).

Принципиальная организация схемы LuNA-программы приведена на рисунке 11.

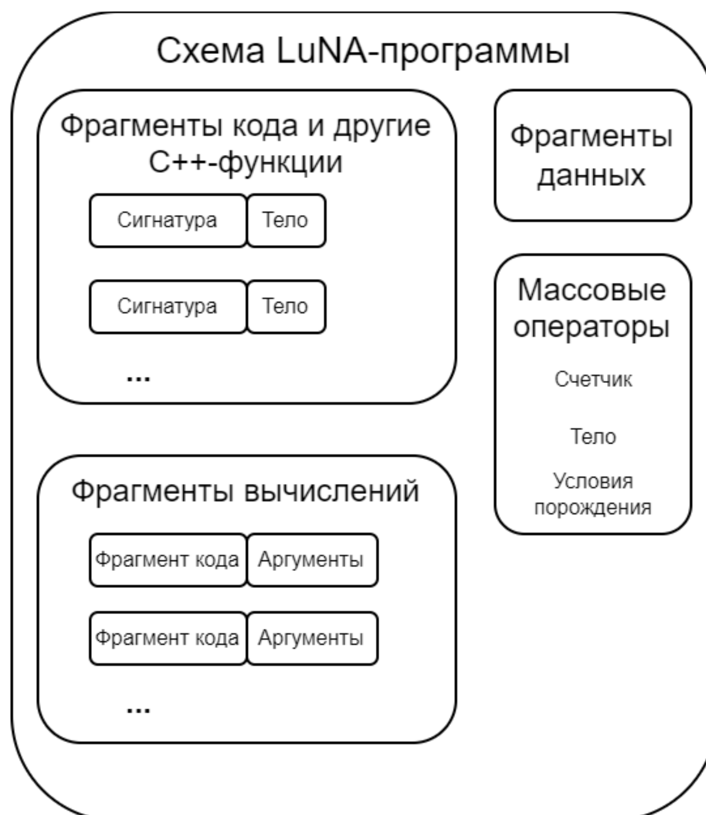


Рисунок 11 — Схема LuNA-программы

Ключевой момент, необходимый для понимания последующего описания алгоритмов генераторов и фрагментатора, заключается в том, что схема LuNA-программы содержит информацию:

- об организации вычислений, т.е. о том, какие выделены фрагменты кода, и каким образом должны порождаться соответствующие им фрагменты вычислений;
- о декомпозиции данных, т.е. о том, какие переменные исходной программы должны быть представлены с помощью локальных переменных фрагментов кода, а какие — с помощью фрагментов данных.

2.6.2 Генератор фрагментов кода

В подразделе описано, каким образом должен работать генератор фрагментов кода.

Для каждого фрагмента кода, описание которого приводится в схеме LuNA-программы, генератор должен составить его сигнатуру и его тело.

2.6.2.1 Составление сигнатуры фрагмента кода

- Имя фрагмента кода берется из схемы LuNA-программы без изменений.
- Тип его возвращаемого значения — `void`.
- Параметры фрагмента кода можно условно поделить на три класса, описанные в следующем списке.
  - Параметры, связанные непосредственно с реализуемым алгоритмом, т.е. параметры, информация о которых содержится в схеме LuNA-программы. Два параметра могут соответствовать одной и той же переменной исходной программы, при этом один из этих параметров соответствует входному фрагменту данных, второй — выходному. Такие параметры далее будут называться *парами входного и выходного параметров*. Любой другой не подпадающий под вышеописанную категорию параметр будет называться *входным*, если он соответствует входному фрагменту данных, и *выходным*, если он соответствует выходному фрагменту данных.
  - Параметры, возникающие в результате фрагментации по пространству, т.е. параметры, соответствующие номеру фрагмента вычислений и количеству фрагментов вычислений по фрагментируемым осям.
  - Параметры, соответствующие инициализируемым в этом фрагменте кода теневым граням.

2.6.2.2 Составление тела фрагмента кода

Тела фрагментов кода определяются исходя из описывающей их последовательности блоков, которая приведена в схеме LuNA-программы. Для каждого блока генератор добавляет во фрагмент кода соответствующую этому блоку синтаксическую конструкцию, при этом порядок блоков сохраняется. Правила определения синтаксических конструкций тривиально следуют из описания типов блоков, приведенного в подразделе 2.6.1.

2.6.3 Генератор фрагментов вычислений и данных

В подразделе описываются, каким образом работает генератор фрагментов вычислений и данных. Его задачей является определение головной подпрограммы `main`.

2.6.3.1 Объявления фрагментов данных

Все базовые имена фрагментов данных, которые описаны в схеме LuNA-программы, объявляются в начале головной подпрограммы. Также объявляются базовые имена фрагментов данных для каждой теневой грани.

2.6.3.2 Определение применений фрагментов кода

Применения фрагментов кода определяются исходя из их описания в схеме LuNA-программы. При этом если применяемый фрагмент кода является пространственным, то оператор, описывающий определяемое применение, обрамляется гнездом массовых операторов `for`. С тем, как это делается, можно ознакомиться в подразделе 2.5.1.

Если в рамках применяемого фрагмента кода инициализируются теневые грани, то в определяемое применение добавляются аргументы, соответствующие теневым граням. В таком случае на том же уровне вложенности, на котором находится рассматриваемый оператор, определяется применение фрагмента кода, который выполняет объединение фрагмента массива и его соседних теневых граней. С тем, как это делается, можно ознакомиться в подразделе 2.5.2.

2.6.3.3 Определение массовых операторов

Генератор фрагментов вычислений и данных должен определить массовые операторы исходя из описания в схеме LuNA-программы. При этом нужно учитывать, что циклы исходной программы, соответствующие определяемым массовым операторам, могут содержать операторы `break`. В следующем списке описаны ограничения на эти операторы `break` (пример оператора соответствующего этим ограничениям приведен в листинге 13).

- Они могут находится только в телах операторов `if`, которые непосредственно вложены в тело цикла исходной программы, соответствующего определяемому массовому оператору.
- Тело этого оператора `if` содержит только один оператор и это оператор `break`.
- Условие этого оператора `if` не имеет побочных эффектов.

Листинг 13 — LuNA-программа, соответствующая программе из листинга 10

```
01: for (int i = 0; i < 1000; ++i)
02: {
03:   if (i > 100) // условие не имеет побочных эффектов
04:   {
05:     break; // break — единственный оператор в теле if
06:   }
```

Продолжение листинга 13

```
07: }
```

Условия вышеописанных операторов `if` содержатся в схеме LuNA-программы с сохранением порядка, в котором они идут в тексте исходной программы. Далее такие условия будут называться *условиями прерывания массового оператора*.

При наличии условий прерывания генератор должен обеспечить описанное далее поведение.

- Минимальное значение счетчика, при котором выполняется любое условие прерывания, далее будет называться *значением прерывания*, а это выполненное условие прерывания — *выполненным условием прерывания*.
- Тело массового оператора должно быть порождено для всех значений счетчика, начиная от его нижней границы, заканчивая значением прерывания, уменьшенным на единицу.
- Все операторы, которые соответствуют той части исходной программы, которая находилась между оператором `if`, соответствующим выполненному условию прерывания, и началом цикла, должны быть выполнены для значения счетчика, равного значению прерывания.

С алгоритмом определения массовых операторов можно ознакомиться в приложении Д.

2.6.3.4 Упорядоченность выполнения фрагментов вычислений

Возникают ситуации, когда фрагменты вычислений нужно выполнять в определенном порядке, который при этом не следует из зависимостей по данным. Принцип обеспечения такого поведения описан в следующем списке.

1) Определим *текущий фрагмент вычислений* как фрагмент вычислений, который можно выполнять уже сейчас.
2) Определим *следующий фрагмент вычислений* как фрагмент вычислений, выполнение которого не может начаться, пока не будет выполнен текущий фрагмент вычислений.
3) Определим *индикатор выполненности* как фрагмент данных, истинное значение которого означает выполненность текущего фрагмента вычислений.
4) Индикатор выполненности нужно инициализировать истинным значением после выполнения текущего фрагмента вычислений. Этот процесс далее будет называться *индикацией выполненности*.

5) Следующий фрагмент вычислений можно начинать выполнять только при истинности индикатора выполненности. Процесс проверки истинности индикатора выполненности далее будет называться *проверкой индикатора*.

Таким образом, для обеспечения упорядоченного выполнения фрагментов вычислений нужно обеспечить индикацию выполненности после выполнения текущего фрагмента вычислений и проверку индикатора перед выполнением следующего фрагмента вычислений.

То, каким образом вышеописанный принцип реализуется в LuNA-программе, описано в разделе 3.2.

2.6.4 Квинтэссенция алгоритмов генераторов

Помимо алгоритмов, описанных в подразделах 2.6.2 и 2.6.3, генераторы также должны реализовывать алгоритмы, описанные в разделе 2.5. В этом подразделе приводится квинтэссенция алгоритмов генератора.

Входные данные: схемы LuNA-программы.

Выходные данные: LuNA-программа, а именно определения атомарных фрагментов кода, и определение головной подпрограммы `main`.

Алгоритм генераторов описан в следующем списке.

1) Генератор фрагментов кода определяет атомарные фрагменты кода, описанные в схеме LuNA-программы (пункты 2.6.2.1 и 2.6.2.2).
2) Генератор фрагментов кода определяет фрагменты кода, выполняющие инициализацию и объединение теневых граней (подраздел 3.1.3).
3) Генератор фрагментов кода описывает в определяемых им фрагментах кода фрагментацию по пространству (подразделы 3.1.1 и 3.1.2).
4) Генератор фрагментов вычислений и данных объявляет фрагменты данных (пункт 2.6.3.1).
5) Генератор фрагментов вычислений и данных описывает применения фрагментов кода (пункт 2.6.3.2).
6) Генератор фрагментов вычислений и данных определяет временные индексы фрагментов данных, исходя из описания порядка выполнения фрагментов вычислений в схеме LuNA-программы (раздел 2.5.3).
7) Генератор фрагментов вычислений и данных обеспечивает упорядоченное выполнение фрагментов вычислений, для которых это необходимо (пункт 2.6.3.4).
8) Генератор фрагментов вычислений и данных определяет массовые операторы (пункт 2.6.3.3).

9) Генератор фрагментов вычислений и данных описывает фрагментацию по пространству для применений фрагментов кода (подразделы 2.5.1 и 2.5.2).

2.7 Фрагментация LuNA-программы

В этом разделе приведены описание алгоритмов фрагментатора и описание структуры его входных данных, т.е. описание структуры информации о параллелизме.

Как было указано при описании структурных элементов схемы LuNA-программы (подраздел 2.6.1), она содержит в себе информацию:

- об организации вычислений, т.е. о том, какие в LuNA-программе должны быть фрагменты кода и фрагменты вычислений;
- о декомпозиции данных, т.е. о том, какие переменные исходной программы в LuNA-программе должны быть представлены с помощью фрагментов данных, а какие — с помощью локальных переменных фрагментов кода.

Поскольку результатом фрагментации последовательной программы должна являться схема LuNA-программы, основная задача фрагментатора — формирование вышеописанной информации и сохранение ее в схеме LuNA-программы.

Далее *формированием фрагментов кода, фрагментов данных, операторов, фрагментации по пространству* или *фрагментации по времени* будут называться составление описания фрагментов кода, фрагментов данных, операторов, фрагментации по пространству или фрагментации по времени в схеме LuNA-программы.

2.7.1 Информация о параллелизме

В подразделе описана принципиальная структура информации о параллелизме, в следующих подразделах она будет по мере необходимости уточняться.

Для того, чтобы фрагментатор мог выполнить формирование фрагментов кода и LuNA-операторов, информация о параллелизме должна включать в себя информацию о принципиальной организации процедуры `main`, которая описывается в виде последовательности блоков. Под *блоком* здесь и далее понимается некоторая структура, описывающая какой-то участок кода исходной программы. Блок является атомарным в том плане, что тот участок кода, который соответствует ему или результату его преобразований на стороне фрагментатора, гарантированно будет помещен в один фрагмент кода. Каждый блок может являться:

- циклом;

— доступом к элементу массива (при этом различаются доступы на запись и на чтение);
— выделением памяти под массив;
— вызовом процедуры;
— последовательным участком кода, который не подпадает ни под одну вышеописанную категорию.

Как можно заметить, описание принципиальной организации процедуры `main` во многом совпадает с описание принципиальной организации фрагментов кода в схеме LuNA-программы (подраздел 2.6.1).

Для процедур, вызываемых в рамках процедуры `main` информации о параллелизме включает информацию о телах (описание которых аналогично описанию тела `main`) и о сигнатурах этих процедур.

Для того, чтобы определить, какие циклы должны быть фрагментированы по пространству и какие циклы должны в LuNA-программе быть представлены с помощью массовых операторов, информация о параллелизме включает в себя описание информационных зависимостей между итерациями цикла.

Для того, чтобы определить, каким образом должна выполняться фрагментация по времени (подраздел 2.5.3), информация о параллелизме включает в себя описание порядка выполнения блоков кода, который в общем случае тривиально следует из порядка следования операторов в коде исходной последовательной программы.

Для того, чтобы определить, какие переменные исходной программы должны соответствовать фрагментам данных, а какие — локальным переменным фрагментов кода, информация о параллелизме включает в себя описание переменных, которые используются в более чем одном блоке. Хранение информации о переменных, которые используются только в одном блоке, для достижения вышеописанной цели не имеет смысла, так как эти переменные гарантированно будут представлены с помощью локальных переменных фрагмента кода вследствие свойства атомарности блоков.

Резюмируя, можно сказать, что информация о параллелизме представляет собой совокупность принципиального описания исходной программы и некоторых дополнительных фактов, необходимых для дальнейшего распараллеливания этой программы.

2.7.2 Фрагментатор

В подразделе описано, каким образом должен работать фрагментатор.

2.7.2.1 Формирование фрагментов кода и массовых операторов

Принцип формирования фрагментов кода изложен в следующем списке.

1) Каждое гнездо циклов, которое можно фрагментировать по пространству, выделяется в отдельный фрагмент кода. Это делается с той целью, чтобы выполнить его фрагментацию по пространству.
2) В отдельные фрагменты кода выделяются последовательные участки кода, которые находятся между двумя вышеописанными гнездами, между началом процедуры `main` и гнездом или между гнездом и концом процедуры `main`.

Т.е. фрагменты кода, формируемые фрагментатором, можно условно разделить на два класса:

- фрагменты кода, соответствующие некоторым последовательным вычислениям, далее они будут называться *последовательными фрагментами кода*;
- фрагменты кода, соответствующие гнездам циклов, далее они будут называться *цикловыми фрагментами кода*.

Для формирования последовательных фрагментов кода фрагментатор должен выделить из описания блоков максимальные по включению последовательности блоков, каждый из которых соответствует некоторому последовательному участку кода (т.е. не циклу) исходной программы. Для каждой такой последовательности формируется фрагмент кода, суть которого заключается в выполнении кода (возможно, преобразованного), соответствующего блокам этой последовательности. Тело такого фрагмента кода получается с помощью тождественного отображения вышеописанной последовательности блоков. Формальный алгоритм определения параметров такого фрагмента кода описан в приложении Е.

Все те циклы, которые не стали частью циклового фрагмента кода, представляются с помощью массовых операторов, при этом фрагментатор должен отразить в схеме информацию об условиях их прерываний.

Вышеописанный принцип формирования фрагментов кода и массовых операторов проиллюстрирован на рисунке 12.

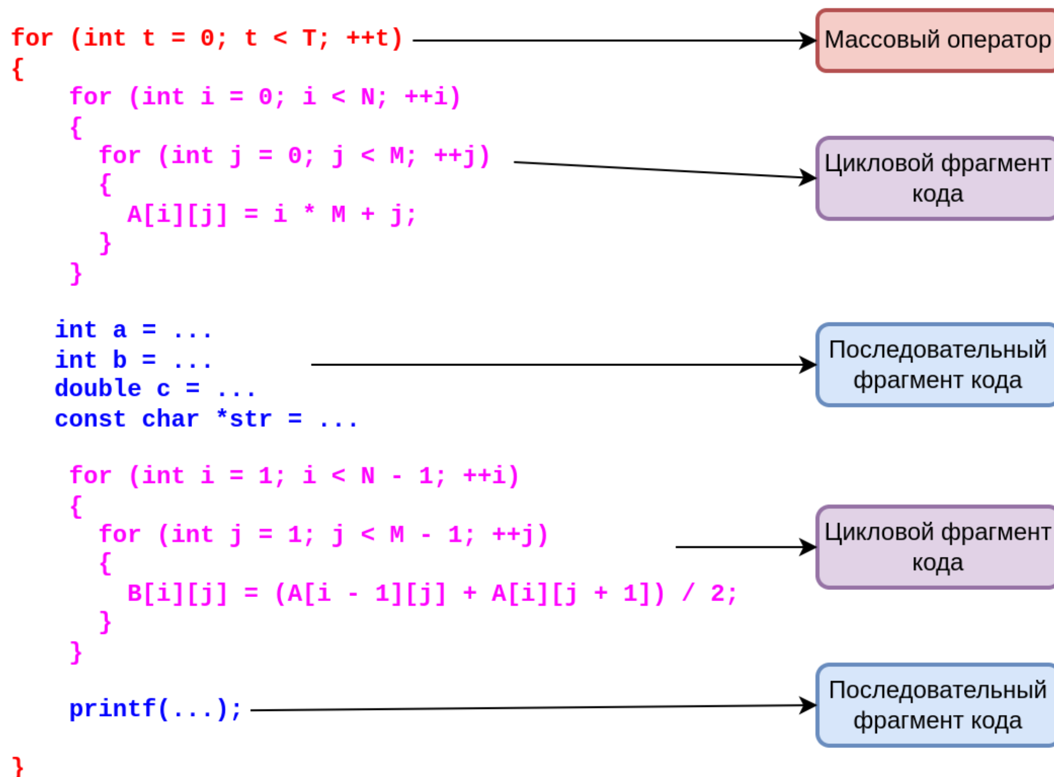


Рисунок 12 — Принцип формирования фрагментов кода и массовых операторов

Как можно увидеть в примере из рисунка 12:

— верхнеуровневый цикл был представлен с помощью массового оператора, так как его нельзя фрагментировать по пространству;
— оба гнезда циклов были выделены в цикловые фрагменты кода;
— последовательный код между двумя гнездами циклов и после второго гнезда выделяется в последовательные фрагменты кода.

2.7.2.2 Фрагментация по пространству

Фрагментатор должен принять решение о выполнении фрагментации по пространству. Последовательные фрагменты кода фрагментироваться не будут, так как в них нет гнезд циклов, которые можно было бы фрагментировать. Для цикловых фрагментов кода фрагментация по пространству формируется для тех циклов, которые не содержат информационных зависимостей, которые не могут быть выражены с помощью редукционных операций. При этом информация о параллелизме включает в себя информацию о том, могут ли конкретные информационные зависимости быть выражены с помощью редукционных операций, ее фрагментатор отражает в схеме LuNA-программы без изменений.

2.7.2.3 Декомпозиция данных

Вопрос о том, каким образом выполнить декомпозицию данных, решается исходя из описания переменных, которое содержит информация о параллелизме, а также того, какие фрагменты кода были сформированы. Те переменные, значения которых вырабатываются и используются в рамках только одного фрагмента вычислений, станут локальными в рамках этого фрагмента вычислений, остальные же станут фрагментами данных.

2.7.2.4 Формирование фрагментов вычислений

Поскольку каждый сформированный фрагмент кода соответствует некоторой части процедуры `main` исходной программы, то для каждого такого фрагмента кода должен быть сформирован оператор, описывающий его применение. Для этого в схеме LuNA-программы указывается какой фрагмент кода применяется к каким аргументам. Аргументы определяются исходя из того, как была выполнена декомпозиция данных: если в рамках применяемого фрагмента кода используется переменная, которая в LuNA-программе представлена с помощью фрагмента данных, то этот фрагмент данных должен являться аргументом формируемого фрагмента вычислений.

2.7.2.5 Фрагментация по времени

Для того, чтобы генераторы описали в LuNA-программе фрагментацию по времени (подраздел 2.5.3), фрагментатор должен зафиксировать в схеме LuNA-программы информацию о порядке выполнения фрагментов вычислений. Этот порядок тривиально следует из порядка выполнения блоков кода, зафиксированного в информации о параллелизме.

2.7.2.6 Теневые грани

Исходя из описания доступов к элементам массивов фрагментатор должен определить, какие теневые грани необходимо инициализировать, и где их необходимо использовать. Формальный алгоритм определения теневых граней описан в приложении А.

2.8 Сбор информации о параллелизме

Сбор информации о параллелизме выполняется SAPFOR'ом и анализатором. При этом SAPFOR ответственен за сбор информации:

- об индукционных переменных циклов;
- об информационных зависимостях;

— о редукционных операциях.

Анализатор ответственен за сбор информации:

— об условиях прерываний;
— о доступах к элементам массивов;
— о выделениях памяти под массив;
— об использованиях индукционных переменных в телах циклов.

2.9 Итоги

Был предложен подход к автоматическому конструированию LuNA-программ на основе системы SAPFOR. В рамках этого подхода были предложены алгоритмы и представления информации, с помощью которых этот подход реализуется. Разработанные алгоритмы и представления предназначены для конструирования LuNA-программ из последовательных программ, реализующих методы из области численного моделирования, т.е. они не являются универсальными. Тем не менее они разработаны таким образом, чтобы их в дальнейшем можно было развить для того, чтобы они подходили для более широкого класса задач.

LuNA-программа представляет фрагментированный алгоритм. Фрагментированный алгоритм — это частный случай базы активных знаний [30, 31], конструкта, предназначенного для автоматического применения некоторых знаний компьютером и для представления знаний в форме, пригодной для этого применения. Разработанные алгоритмы и представления информации можно в дальнейшем развивать таким образом, чтобы с их помощью автоматически конструировать не только фрагментированные алгоритмы, но и базы активных знаний.

3 Генерация фрагментов кода

В главе рассматривается, каким образом реализуются изложенные в главе 2 алгоритмы генерации LuNA-программы. Также описывается реализация генератора фрагментов кода и экспериментальное исследование этого генератора. Глава структурирована следующим образом. В разделе 3.1 описывается, каким образом во фрагментах кода реализуется фрагментация по пространству. В разделе 3.2 описывается, каким образом обеспечивается упорядоченное выполнение фрагментов вычислений, при условии, что этот порядок не следует из информационных зависимостей. В разделе 3.3 описываются форматы хранения информации о параллелизме и схемы LuNA-программы. В разделе 3.4 описывается реализация генератора фрагментов кода. В разделе 3.5 описывается тестирование реализованного генератора на примере программы, реализующей метод Якоби. Завершается глава подведением итогов (раздел 3.6).

3.1 Фрагментация по пространству

В разделе на примере конкретных программ представляется, каким образом в пространственных фрагментах кода реализуется фрагментация по пространству, описанная в подразделах 2.5.1 и 2.5.2.

3.1.1 Определение пространственных фрагментов кода

В подразделе на примере конкретных программ демонстрируется, каким образом определяются пространственные фрагменты кода.

В листинге 14 представлен пространственный фрагмент кода, который соответствует гнезду циклов из листинга 7 при выполнении двумерной фрагментации, при которой каждый цикл разбивается на два фрагмента.

Листинг 14 — Пространственный фрагмент кода, соответствующий фрагменту гнезда циклов из листинга 7

```
01: void loop_nest(int f0, int f1, OutputDF &A_df)
02: {
03:   int *A = A_df.create<int>(50 * 50);
04:   // пересчет границ гнезда циклов
05:   int i0_real_init = get_real_inductive_init(f0, 2, 0, 100);
06:   int i1_real_init = get_real_inductive_init(f0, 2, 0, 100);
07:
```

Продолжение листинга 14

```
08:    for (int i0 = 0; i0 < 50; ++i0)
09:      for (int i1 = 0; i1 < 50; ++i1)
10:        A[i0 * 50 + i1] = calc_no_side_effects(
11:          i0_real_init + i0,
12:          i1_real_init + i1);
13: }
```

Отметим характерные особенности пространственного фрагмента кода на примере из листинга 14.

1) У него есть параметры `f0`, `f1`, которые соответствуют номеру фрагмента вычислений по 0-ой и 1-ой оси, т.е. для этого примера соответствующие аргументы будут равны 0 или 1.
2) Границы циклов в гнезде пересчитываются (строки 05-06). В простом случае (как в листинге выше), когда нижняя граница цикла равна 0, а верхняя граница делится нацело на количество фрагментов вычислений, в преобразованном цикле нижняя граница равна 0, а верхняя — исходной верхней границе, деленной на размер фрагментации по оси, соответствующей циклу, для которого выполняется пересчет.
3) Использование индукционных переменных `i<k>` в теле цикла (строки 10-11) заменяется на `(i<k> + i<k>_real_init)`, где `i<k>_real_init` — такое значение индукционной переменной исходного цикла, которое соответствует начальному значению индукционной переменной преобразованного цикла. Далее это значение будет называться *реальным значением индукционной переменной* по оси k. Реальное значение по оси k в рассматриваемом примере считается с помощью функции `get_real_inductive`, которая имеет параметры, описанные в следующем списке.
   - Номер фрагмента вычислений по оси k. В данном примере — `f<k>`.
   - Количество фрагментов вычислений по оси k. В данном примере — 2.
   - Нижняя граница индукционной переменной исходного цикла. В данном примере — 0.
   - Верхняя граница индукционной переменной исходного цикла. В данном примере — 100.

Определение этой процедуры можно найти в приложении Ё.

Значения переменных `i<k>_real_init` для данного примера приведены в таблице 1.

Таблица 1 — Реальные значения индукционных переменных во фрагментах вычислений, соответствующих фрагментам кода, приведенным в листинге 14

Номер фрагмента вычислений	Значение (i_0_real_init, i_1_real_init)	Пояснение
(0, 0)	(0, 0)	-
(0, 1)	(0, 50)	Так как итерации исходного цикла, соответствующие значению индукционной переменной i1 от 0 до 49 выполняются в рамках фрагментов вычислений с номером по оси 1 равным 0, то реальное значение индукционной переменной по этой оси равно 50.
(1, 0)	(50, 0)	Аналогично (0, 1)
(1, 1)	(50, 50)	Аналогично (0, 1)

Нужно отметить, что в общем случае, когда количество фрагментов вычислений задается с помощью параметров фрагментов кода (т.е. нет гарантии о делимости верхней границы циклов или размеров массивов на количество фрагментов), а нижняя граница цикла может быть не равна 0, фрагмент кода несколько усложняется. В самое начало его тела добавляется пересчет границ циклов и размеров массивов с помощью специальных функций. Рассмотрим, каким образом это происходит на примере части последовательной программы из листинга 15.

Листинг 15 — Вычислительноемкое гнездо циклов с границами, значения которых неизвестны во время компиляции

```
01: for (int i0 = I0_INIT; i0 < N; ++i0)
02: {
03:   for (int i1 = I1_INIT; i1 < M; ++i1)
04:   {
05:     A[i0][i1] = calc_no_side_effects(i0, i1);
06:   }
07: }
```

Код из листинга 15 — это несколько измененный код из листинга 7: границы циклов в этом коде, вместо постоянных значений, были заменены на возможно неизвестные на момент компиляции значения, которые могут задаваться с помощью макросов или других переменных. При выполнении двумерной фрагментации по пространству такого цикла будет получатся фрагмент кода, приведенный в листинге 16.

Листинг 16 — Пространственный фрагмент кода, соответствующий фрагменту гнезда циклов из листинга 15

```
01: void loop_nest(int f0, int nf0, int f1, int nf1,
02:   OutputDF& A_df)
03: {
04:   // пересчет размеров массивов
05:   int A_size_0 = get_size(f0, nf0, N);
06:   int A_size_1 = get_size(f1, nf1, M);
07:
08:   int* A = A_df.create<int>(A_size_0  * A_size_1);
09:   // пересчет границ гнезда циклов
10:   int i0_0_init = get_inductive_init(f0, nf0, I0_INIT, N);
11:   int i1_1_init = get_inductive_init(f0, nf0, I1_INIT, M);
12:   int i0_0_end = get_inductive_end(f0, nf0, N);
13:   int i1_1_end = get_inductive_end(f0, nf0, M);
14:   int i0_real_init =
15:     get_real_inductive_init(f0, 2, I0_INIT, N);
16:   int i1_real_init =
17:     get_real_inductive_init(f0, 2, I1_INIT, M);
18:
19:   for (int i0 = i0_0_init; i0 < i0_0_end; ++i0)
20:   {
21:     for (int i1 = i1_1_init; i1 < i1_1_end; ++i1)
22:     {
23:       A[i0 * A_size_1 + i1] =
24:         calc_no_side_effects(i0_real_init + i0,
25:           i1_real_init + i1);
26:     }
27:   }
28: }
```

Как можно заметить, у фрагмента кода из листинга 16 в отличие от фрагмента кода из листинга 14 есть параметры соответствующие не только номерам фрагментов вычислений, но и их количеству (`nf0`, `nf1`). В теле фрагмента кода перед выполнением фрагментируемого по пространству цикла выполняется пересчет границ циклов (строки 09-14) и размеров массивов (строки 04-05), результаты этого пересчета далее будут называться *фрагментированными границами* и *фрагментированными размерами* соответственно. Вышеописанный пересчет выполняется с помощью специальных процедур `get_size`, `get_inductive_init`, `get_inductive_end`, `get_real_inductive_init`, определения которых приведены в приложении Ё.

3.1.2 Редукционные операции

В циклах могут выполняться редукционные операции, например, суммирование, умножение, конъюнкция и т.д. элементов некоторого массива и сохранение результата в некоторой переменной. Пример такого цикла приведен в листинге 17.

Листинг 17 — Пример цикла с редукционными операциями

```
01: int sum = 0;
02: for (int i = 0; i < N; ++i)
03: {
04:   sum += A[i];
05: }
```

В программе из листинга 17 происходит суммирование элементов некоторого массива и сохранение результата этого суммирования в переменной `sum`.

Пространственный фрагмент кода, который будет определен при фрагментации по пространству рассматриваемого цикла, приведен в листинге 18.

Листинг 18 — Фрагмент кода, соответствующий циклу из листинга 17

```
01: int sum_loop(int f, int nf, OutputDF& sum, InputDF& A_df)
02: {
03:   ...
04:
05:   sum = 0;
06:
07:   for (int i = i_init; i < i_end; ++i)
08:   {
09:     sum = sum + A[i];
10:   }
11: }
```

В начале этого фрагмента кода выполняется пересчет размера массива `A` и границ цикла (эта часть явно не приводится, так как аналогичная ей приводится в листинге 16). Затем выполняется фрагментируемый по пространству цикл.

В листинге 19 продемонстрировано, каким образом с помощью LuNA-оператора `for` будет порождаться множество пространственных фрагментов вычислений, соответствующих этому циклу. Далее пространственные фрагменты вычислений, выполняющие редукционную операцию, будем называть *редукционными*.

Листинг 19 — Порождение редукционных фрагментов вычислений, соответствующих вызовам фрагмента кода из листинга 18

```
01: for f0=0..nf-1
02: {
03:   sum_loop(f0, nf, sum[...], A[f0]);
04:   ...
05: }
```

Результаты выполнения редукционных операций, которые вырабатывают редукционные фрагменты вычислений, не содержат того же значения, что и переменная, хранящая результат редукционной операции в исходной программе, так как в редукционных фрагментах вычислений выполняются только фрагменты циклов и обрабатываются только фрагменты массивов. Результаты выполнения редукционных операций, которые вырабатывают редукционные фрагменты вычислений будут далее называться *фрагментированными результатами редукционной операции*.

Для того, чтобы выработать фрагмент данных, который будет хранить значение равное результату редукционной операции в исходной программе (значению переменной `sum` в рассматриваемом примере), предлагается в рамках оператора `for` порождать фрагменты вычислений, которые будут собирать фрагментированные результаты редукционной операции (см. листинг 20). Далее такие, собирающие фрагментированные результаты выполнения редукционных операций, фрагменты вычислений будут называться *собирающими*.

Листинг 20 — Выполнение редукционной операции суммирования в LuNA

```
01: init(defragmented_sum[0], 0);
02: for f0=0..nf-1
03: {
04:   sum_loop(f0, nf, sum[f0], A[f0]);
05:   reduction_sum(sum[f0], defragmented_sum[f0],
06:     defragmented_sum[f0 + 1]);
07: }
```

Для хранения результатов выполнения собирающих фрагментов вычислений объявляется базовое имя фрагмента данных `defragmented_sum`. Собирающие фрагменты вычислений являются применениями специального фрагмента кода. В рамках примера этим фрагментов кода является `reduction_sum`, определение которого приведено в листинге 21. После выполнения всех собирающих фрагментов вычислений значение фрагмента данных `defragmented_sum[nf]` равно той же сумме, что и значение переменной `sum` исходной

программы после выполнения цикла. При этом фрагмент данных `defragmented_sum[0]` инициализируется значением 0 (строка 01), как и переменная `sum` исходной программы.

Листинг 21 — Определение фрагмента кода, с помощью которого в LuNA реализуется редукционная операция суммирования

```
01:void reduction_sum(int fragmented_sum, int old_sum,
02:  OutputDF &new_sum_df)
03:{
04:  new_sum_df = fragmented_sum + old_sum;
05:}
```

Во фрагменте кода `reduction_sum` выполняется инициализация нового значения редукционной переменной с помощью суммы старого значения и значения, выработанного в рамках одного из пространственных фрагментов вычислений.

Резюмируя, алгоритм выполнения редукционных операций в LuNA можно описать следующим образом.

1) Объявляется базовое имя фрагмента данных, значение которого соответствует результату выполнения редукционной операции.
2) Внутри гнезда операторов `for`, с помощью которого порождаются редукционные фрагменты вычислений, описываются собирающие фрагменты вычислений.
3) Определяются фрагмент кода, соответствующий выполняемой редукционной операции. Применение этого фрагмента кода — собирающий фрагмент вычислений.
4) В собирающий фрагмент вычислений с номером (f0, …, fN), передается три аргумента:
    a) результат выполнения редукционной операции, выработанный редукционным фрагментом вычислений с номером (f0, …, fN);
    b) результат выполнения собирающего фрагмента кода с номером по оси k на единицу меньшим, чем номер текущего фрагмента вычислений, где k — такой номер оси, что f<k + 1> = 0 и при этом k является максимальным, или N, если ни один f<i> не равен 0;
    c) выходной фрагмент данных, который соответствует результату выполнения собирающего фрагмента вычислений.

3.1.3 Теневые грани

В подразделе описано, каким образом определяются фрагменты кода, выполняющие инициализацию, объединение и использование теневых граней на примере программы из

листинга 9. Определения фрагмента кода, который соответствует гнезду инициализации массива A, приведено в листинге 22. Определение фрагмента кода, который выполняет объединение теневых граней и основной части фрагмента массива, приведено в листинге 25. Определение фрагмента кода, который соответствует гнезду вычислений, приведено в листинге 28.

Листинг 22 — Определение фрагмента кода, в рамках которого выполняется фрагмент гнезда инициализации из листинга 7

```
01: void init_A(int f0, int nf0, int f1, int nf1, OutputDF& A_df,
02:   OutputDF &A_shadow_1_0_df, OutputDF &A_shadow_m1_0_df,
03:   OutputDF &A_shadow_0_1_df, OutputDF &A_shadow_0_m1_df,
04:   OutputDF &A_shadow_1_1_df, OutputDF &A_shadow_m1_1_df,
05:   OutputDF &A_shadow_1_m1_df, OutputDF &A_shadow_m1_m1_df)
06: {
07:   int A_size_0 = get_size(f0, nf0, N);
08:   int A_size_1 = get_size(f1, nf1, M);
09:   int i_0_init = get_inductive_init(f0, nf0, 0, N);
10:   int j_1_init = get_inductive_init(f1, nf1, 0, M);
11:   int i_0_end = get_inductive_end(f0, nf0, N);
12:   int j_1_end = get_inductive_end(f1, nf1, M);
13:   int i_0_real_init = get_real_inductive_init(f0, nf0, 0, N);
14:   int j_1_real_init = get_real_inductive_init(f1, nf1, 0, M);
15:
16:   int *A = A_df.create<int>((A_size_0 + 2) * (A_size_1 + 2));
17:
18:   for (int i = i_0_init; i < i_0_end; ++i)
19:     for (int j = j_1_init; j < j_1_end; ++j)
20:       A[(i + 1) * (A_size_1 + 2) + (j + 1)] =
21:         max(t * (i + i_0_real_init),
22:           t * (j + j_1_real_init));
23:  }
```

У этого фрагмента кода есть параметры, соответствующие иницилизируемым теневым граням (A_shadow_...). В начале тела фрагмента кода происходит пересчет границ циклов и размеров массивов, как и в случае фрагментации по пространству без теневых граней. Затем выполняется выделение памяти под фрагмент массива A, при этом размер каждой его оси больше размера основной части фрагмента массива на сумму самых больших толщин отрицательных и положительных теневых граней. В рассматриваемом примере размер каждой оси массива был увеличен на 2, так как сумма самых больших толщин отрицательных и положительных теневых граней по обеим осям равна 2. Затем выполняется часть фрагментируемого по пространству гнезда инициализации, при этом вследствие увеличения размера массива изменяется индексация его элементов. Затем в рамках этого

фрагмента кода выполняется инициализация теневых граней, шаблон гнезда циклов, с помощью которого это будет происходить, приведен в листинге 23.

Листинг 23 — Шаблон инициализации теневой грани

```
01: for (int i0 = 0; i0 < <shadow_size_0>; ++i0)
02: {
03:   for (int i1 = 0; i1 < <shadow_size_1>; ++i1)
04:   {
05:     A_shadow_<st0>_<st1>[i0 * shadow_size_1 + i1] =
06:       A[<j0> * (A_size_1 + 2) + <j1>];
07:   }
08: }
```

Шаблона из листинга содержат следующие элементы.

- `<stk>` (shadow thickness k) — толщина теневой грани по оси k. Если теневая грань отрицательная, то записывается с префиксом `m`.
- `shadow_size_<k>`:
  - Если `st<k>` = 0, то `shadow_size_<k>` — размер k-ой оси массива.
  - Иначе `shadow_size_<k>` — толщина теневой грани (т.е. `shadow_size_<k> = st<k>`).
- `<jk>`:
  - В случае, когда по k-ой оси теневая грань отрицательная, `jk = neg_k + array_size_k - (swk - ik)`, где `neg_k` — самая большая толщина отрицательной теневой грани по k-ой оси, а `array_size_k` — размер k-ой оси фрагмента массива (без учета размера теневых граней).
  - Иначе, в случае, когда по k-ой оси теневая грань отрицательная или нулевая, `jk = ik + neg_k`.

В качестве конкретного примера такого гнезда рассмотрим инициализацию положительной по 0-ой оси и отрицательной по 1-ой оси теневой грани, которая приведена в листинге 24.

Листинг 24 — Инициализация положительной по 0-ой оси и отрицательной по 1-ой оси теневой грани

```
01: for (int i0 = 0; i0 < 1; ++i0)
02: {
03:   for (int i1 = 0; i1 < 1; ++i1)
04:   {
05:     A_shadow_1_m1[i0 + i1] = A[(1 + A_size_0 - (1 - i0))
06:       * (A_size_1 + 2) + 1 + i0];
```

Продолжение листинга 24

```
07:    }
08: }
```

Верхние границы обоих циклов равны толщинам теневых граней массива по соответствующим осям, так как теневые грани не являются нулевыми по этим осям. Индексация по 0-ой оси выполняется в соответствии с тем, что по этой оси теневая грань является положительной, а по 1-ой — в соответствии с тем, что по этой оси теневая грань является отрицательной.

Листинг 25 — Определение фрагмента кода, который выполняет объединение теневых граней из листинга 9

```
01: void merge_shadow_A(int f0, int nf0, int f1, int nf1,
02:   InputDF &A_df, InputDF &A_shadow_1_0_df,
03:   InputDF &A_shadow_m1_0_df, InputDF &A_shadow_0_1_df,
04:   InputDF &A_shadow_0_m1_df, InputDF &A_shadow_1_1_df,
05:   InputDF &A_shadow_m1_1_df, InputDF &A_shadow_1_m1_df,
06:   InputDF &A_shadow_m1_m1_df, OutputDF &A_merged_df)
07: {
08:
09:   int A_size_0 = get_size(f0, nf0, N);
10:   int A_size_1 = get_size(f1, nf1, M);
11:
12:   A_merged_df = A_df;
13:
14:   int *A = (int *)(static_cast<const size_t *>
15:     (A_merged_df.get_data()));
16:
17:   int *A_shadow_1_0 = (int *)(static_cast<const size_t *>
18:     (A_shadow_1_0_df.get_data()));
19:
20:   ...
21: }
```

У этого фрагмента кода есть параметры, соответствующие фрагменту массива (`A_df`) и теневым граням (`A_shadow_`...), а также параметр, который соответствует объединенным фрагменту массива и теневым граням (`A_merged_df`). В начале фрагмента кода вычисляется размер фрагмента массива `A` по всем осям, затем с помощью уничтожающего потребления инициализируется фрагмент данных, соответствующий объединенной части массива. В размер фрагмента массива закладывается размер теневых граней для того, чтобы объединенную часть можно было инициализировать с помощью уничтожающего потребления. Далее идет получение значений фрагментов данных, соответствующих теневым

граням, после чего идет их объединение с основной частью массива, шаблон гнезда циклов, с помощью которого это делается, приведен в листинге 26.

Листинг 26 — Шаблон объединения теневой грани и фрагмента массива

```
01: for (int i0 = 0; i0 < <shadow_size_0>; ++i0)
02: {
03:   for (int i1 = 0; i1 < <shadow_size_1>; ++i1)
04:   {
05:     A[<j0> * (A_size_1 + 2) + <j1>] =
06:       A_shadow_<st0>_<st1>[i0 * <shadow_size_1> + i1];
07:   }
08: }
```

Шаблона из листинга содержат следующие элементы.

— `<shadow_size_k>` и `<stk>` имеют тот же смысл, что и в шаблоне из листинга 23.

— `<jk>`:

  - В случае, когда по k-ой оси теневая грань положительна, `jk = neg_k + array_size_k + ik`, где `neg_k` — самая большая ширина отрицательной теневой грани по k-ой оси, а `array_size_k` — размер k-ой компоненты фрагмента массива (без учета размера теневых граней)
  - В случае, когда по k-ой оси теневая грань отрицательна, `jk = ik`.
  - Иначе, когда по k-ой оси теневая грани нулевая, `jk = ik + neg_k`.

В качестве конкретного примера рассмотрим объединение положительной по 0-ой оси и нулевой по 1-ой оси теневой грани, которое приведено в листинге 27.

Листинг 27 — Объединение положительной по 0-ой оси и нулевой по 1-ой оси теневой грани

```
01: for (int i = 0; i < 1; ++i)
02: {
03:   for (int j = 0; j < A_size_1; ++j)
04:   {
05:     A[(A_size_0 + 1 + i) * (A_size_1 + 2) + (j + 1)] =
06:       A_shadow_1_0[i * A_size_1 + j];
07:   }
08: }
```

Верхняя граница верхнеуровневого цикла равна толщине теневой грани массива по этой оси, так как по этой оси теневая грань не является нулевой. Верхняя граница вложенного цикла равна размеру массива по этой оси, так как по 1-ой оси теневая грань — нулевая. Индексация по 0-ой оси выполняется в соответствии с тем, что по этой оси теневая

грань является положительной, а по 1-ой — в соответствии с тем, что по этой оси теневая грань является отрицательной.

Листинг 28 — Определение фрагмента кода, в рамках которого выполняется фрагмент гнезда вычислений из листинга 7

```
01: void calc_B(int f0, int nf0, int f1, int nf1,
02:   InputDF &A_merged_df, OutputDF &B_df)
03: {
04:   ... // пересчет размеров массивов и границ циклов
05:
06:   int *A = (int *)(static_cast<const size_t *>
07:     (A_merged_df.get_data()));
08:   int *B = B_df.create<int>(B_size_0 * B_size_1);
09:
10:   for (int i = i_0_init; i < i_0_end; ++i)
11:   {
12:     for (int j = j_1_init; j < j_1_end; ++j)
13:     {
14:       B[i * A_size_0 + j] =
15:         A[(i + 1 - 1) * (A_size_1 + 2) + (j + 1 - 1)] +
16:         A[(i + 1 - 1) * (A_size_1 + 2) + (j + 1 + 1)] +
17:         A[(i + 1 + 1) * (A_size_1 + 2) + (j + 1 - 1)] +
18:         A[(i + 1 + 1) * (A_size_1 + 2) + (j + 1 + 1)];
19:     }
20:   }
21: }
```

Как можно видеть, в начале фрагмента кода происходит пересчет границ циклов и размеров массивов (эта часть явно в листинге не приводится, так как она аналогична соответствующей части из листинга 16), затем выполняется получение значения фрагмента данных, соответствующего массиву A, и выделение памяти под массив B. Далее выполняется часть фрагментируемого по пространству гнезда вычислений, при этом единственным артефактом наличия теневых граней является соответствующее изменение индексации массива A.

3.2 Упорядоченное выполнение фрагментов вычислений

В разделе описывается, каким образом предлагается реализовывать упорядоченность выполнения фрагментов вычислений, если она не следует из информационных зависимостей. С теоретическим обоснованием такого поведения можно ознакомиться в подразделе 2.6.3.4.

Реализовывать индикацию выполненности предлагается с помощью оператора >>, как это показано в листинге 29.

Листинг 29 — Реализация индикации выполненности

```
<a>(...) >> (<indicator>);
```

В листинге 29:

— `<a>` — описание текущего фрагмента вычислений;

— `<indicator>` — индикатор выполненности.

Реализовывать проверку индикатора предлагается с помощью условного оператора `if`, как это показано в листинге 30.

Листинг 30 — Реализация проверки индикатора

```
if (<indicator>)
     <b>(...);
```

В листинге 30:

— `<b>` — описание следующего фрагмента вычислений;

— `<indicator>` — индикатор выполненности.

Следует отметить, что в качестве индикатора может выступать индексированное имя фрагмента данных.

Далее будут рассмотрены две ситуации, в которых применяется вышеописанный принцип.

3.2.1 Упорядоченность пространственных фрагментов вычислений

Упорядоченность выполнения фрагментов вычислений требуется, когда они соответствуют пространственному фрагменту кода, в котором есть побочные эффекты (например, упорядоченный вывод элементов массива). Пример того, как обеспечивается подобное поведение, приведен в листинге 31.

Листинг 31 — Упорядоченное выполнение простраcтранственных фрагментов вычислений

```
01: init(can_execute_next[0], 1);
02: for f0=0..nf-1
03:   for f1=0..nf-1
04:     if (1 == can_execute_next[f0 * nf + f1])
05:       ordered_code_fragment(f0, f1, …) >>
06:         (can_execute_next[f0 * nf + f1 + 1]);
```

В листинге 31 `ordered_code_fragment` — фрагментированный по пространству фрагмент кода, который должен выполняться упорядоченно. В такой ситуации текущему и

следующему фрагменту вычислений соответствует один и тот же оператор. В качестве индикатора выполненности фрагмента вычислений с номером (f0, f1) выступает фрагмент данных can_execute_next[f0 * nf + f1]. Чтобы начать последовательность выполнения фрагментов вычислений, истинным значением инициализируется фрагмент данных can_execute_next[0].

3.2.2 Упорядоченность фрагментов вычислений, порождаемых массовыми операторами

Упорядоченность выполнения фрагментов вычислений требуется, когда они соответствуют фрагменту кода, в котором есть побочные эффекты (например, упорядоченный вывод элементов массива), и при этом оператор, описывающий эти фрагменты вычислений, находится в теле массового оператора. Пример того, как обеспечивается подобное поведение приведен в листинге 32.

Листинг 32 — Упорядоченное выполнение фрагментов вычислений, порождаемых массовым оператором

```
01: init(can_do_next[0], 1);
02: for t=0..100
03:   if can_do_next[t]
04:     ordered_code_fragment(...) >> (can_do_next[t + 1]);
```

В листинге 32 ordered_code_fragment — фрагмент кода, который должен выполняться упорядоченно. Так же, как и в рассмотренной выше ситуации с фрагментацией по пространству, в данном примере текущему и следующему фрагменту кода соответствует один и тот же описатель. В качестве индикатора выполненности фрагмента вычислений, порождаемого массовым оператором при значении счетчика равным i, выступает фрагмент данных can_execute_next[i]. Чтобы начать последовательность выполнения фрагментов вычислений, истинным значением инициализируется фрагмент данных can_execute_next[0].

3.3 Форматы представления информации о параллелизме и схемы LuNA-программе

В рамках работы были разработаны форматы представления информации о параллелизме (подраздел 2.7.1) и схемы LuNA-программы (подраздел 2.6.1), основанные на текстовом формате JSON (JavaScript Object Notation). Такой выбор был сделан из-за того, что JSON имеет хорошую инструментальную поддержку во многих языках программирования, в

том числе и в языке программирования C++, который используется для реализации генератора фрагментов кода. Помимо этого JSON позволяет определять форматы, которые относительно хорошо читаются человеком, что будет способствовать процессу отладки.

3.4 Реализация генератора фрагментов кода

В рамках работы был реализован генератор фрагментов кода. Решение реализовывать генератор фрагментов кода раньше остальных модулей было принято по следующим причинам. Во-первых, реализация и экспериментальное исследование генератора позволяет проверить полноту и правильность схемы LuNA-программы, что важно для реализации фрагментатора, так как эта схема должна стать результатом его работы. Во-вторых, для реализации некоторых алгоритмов генератора фрагментов кода было предложено два подхода: первый — с использованием source-to-source трансформаций, второй — без их использования. Использование второго подхода требует расширение схемы LuNA-программы, поэтому перед реализацией генератора фрагментов вычислений и данных, нужно было реализовать оба подхода и сделать выводы о том, какой из них является более подходящим. В результате было принято решение использовать подход с расширением схемы LuNA-программы и без использования source-to-source трансформаций.

Принцип работы генератора описан в следующем списке.

1) Сначала генератор должен выполнить разбор схемы LuNA-программы, которая представлена в виде файла в формате, который описан в разделе 3.3. Результат этого разбора далее будет называться *представлением LuNA-программы*. Представление LuNA-программы хранится во внутренних структурах генератора.
2) Генератор выполняет серию преобразований над представлением LuNA-программы, которые направлены на определение фрагментов кода, при этом генератор использует алгоритмы, изложенные в главе 2. В результате вышеописанных преобразований представление LuNA-программы содержит всю информацию, необходимую для определения фрагментов кода (в том числе и техническую).
3) Исходя из представления LuNA-программы генератор составляет определения фрагментов кода, т.е. текст на языке программирования C++.
4) Результирующий текст определения фрагментов кода выводится в файл, который указывается пользователем при запуске.

Было принято решение реализовывать генератор на языке программирования C++, так как он поддерживает объектно-ориентированную парадигму программирования, которая является удобной для реализации вышеописанного принципа. Также для этого языка

существует удобный инструментарий для выполнения source-to-source трансформаций программ на языке программирования C++. Для сборки исполняемого файла из исходных файлов генератора фрагментов кода используется система сборки CMake [32].

Исходный код генератора фрагментов кода хранится в репозитории в системе контроля версий git. Этот репозиторий доступен для скачивания в сети интернет[1]. Объем исходного кода составляет примерно 6000 строк. Руководство программиста и описание программы приведены в приложениях З и И соответственно.

В следующих подразделах некоторые этапы работы генератора будут описаны подробнее.

3.4.1 Разбор схемы LuNA-программы

Разбор схемы LuNA-программы выполняется с помощью библиотеки для разбора JSON nlohmann::json [33].

3.4.2 Преобразования представления LuNA-программы

В следующем списке описаны преобразования, которые генератор фрагментов кода выполняет над представлением LuNA-программы.

1) Значения выходных фрагментов данных должны инициализироваться с помощью вызова метода `create`. Генератор фрагментов кода добавляет вызовы этих методов для соответствующих этим фрагментам данных параметров.
2) Для получения значений входных фрагментов данных необходимо выполнить вызов метода `get_data`. Генератор фрагментов кода добавляет вызовы этих методов для соответствующих этим фрагментам данных параметров.
3) Генератор добавляет в представление LuNA-программы дополнительные процедуры, предназначенные для подсчета фрагментированных границ циклов и фрагментированных границ массивов. С определением этих процедур можно ознакомиться в приложении Ё.
4) Генератор добавляет в представление LuNA-программы определения фрагментов кода, которые выполняют редукционные операции (подраздел 3.1.2).
5) Генератор добавляет в пространственные фрагменты кода параметры, соответствующие размеру фрагментации и номеру фрагмента вычислений по фрагментируемым осям.
6) Генератор добавляет в пространственные фрагменты кода вычисления фрагментированных границ и размеров с помощью процедур, описанных в п. 3.

[1] https://gitlab.ssd.sscc.ru/sapfor-luna/atomiccodefragments

7) Генератор изменяет размеры всех фрагментируемых массивов на фрагментированные, описанные в п. 6.
8) Генератор изменяет границы фрагментируемых циклов на фрагментированные, описанные в п. 6.
9) Генератор изменяет индексацию массивов на одномерную.
10) Генератор добавляет во фрагменты кода, которые должны инициализировать теневые грани, параметры, соответствующие теневым граням (подраздел 3.1.3).
11) Генератор увеличивает размеры массивов, для которых инициализируются теневые грани (подраздел 3.1.3).
12) Генератор добавляет во фрагменты кода описанные в п. 10 гнезда циклов, в рамках которых выполняется инициализация теневых граней (подраздел 3.1.3).
13) Генератор добавляет определения фрагментов кода, которые выполняют объединение фрагмента массива и его соседних теневых граней (подраздел 3.1.3).

3.5 Тестирование генератора фрагментов кода

Реализованный генератор был протестирован на примере программы, реализующей метод Якоби. Эта программа была выбрана потому, что она обладает многими характерными особенностями программ из области, на которую направлен разрабатываемый подход, например, наличие вычислительноемких гнезд арифметических циклов, необходимость использования теневых граней при фрагментации по пространству этих гнезд, наличие в этих гнездах редукционных операций и т.д. Общий объем сгенерированного кода для этой программы составил примерно 1400 строк, с исходной последовательной программой можно ознакомиться в приложении Ж.

Для вышеописанной программы были вручную сгенерированы определения фрагментов вычислений и данных, которые в совокупности с автоматически сгенерированными определениями фрагментов кода составляют LuNA-программу. Результат работы получившейся LuNA-программы совпадал с результатом работы исходной последовательной программы, из чего можно сделать вывод, что для рассматриваемого примера генератор фрагментов кода сработал правильно.

Для оценки того, насколько LuNA-программа оказалась эффективной с точки зрения времени исполнения, была выполнена серия запусков исходной последовательной программы и вышеописанной LuNA-программы. Время работы программы при каждом запуске измерялось с помощью процедуры `gettimeofday`. Это тестирование выполнялось на ноутбуке ASUS Vivobook X1704ZA-AU123 с центральным процессором Intel Core i5-1235U. В LuNA-программе каждое гнездо циклов фрагментировалось по пространству, эта

фрагментация выполнялась по всем трем осям, размер фрагментации по каждой оси — 4. Всего было произведено 10 запусков исходной программы и 10 запусков LuNA-программы, для каждой программы выбиралось минимальное время работы. Ускорение рассчитывается как частное времен работы LuNA-программы и последовательной программы. Результаты тестирования приведены в таблице 2.

Таблица 2 — Время работы последовательной программы и LuNA-программы, реализующих метод Якоби

Время работы последовательной программы (с)	Время работы LuNA-программы (с)	Ускорение
70,4251	10,7603	6,54490

Исходя из результатов тестирования, можно сделать вывод, что сгенерированная LuNA-программа существенно производительнее исходной последовательной.

3.6 Итоги

Были предложены конкретные способы реализации алгоритмов генерации фрагментов кода, описанных в главе 2. Генератор фрагментов кода был реализован и протестирован на примере метода Якоби.

ЗАКЛЮЧЕНИЕ

Предложен подход к автоматическому конструированию LuNA-программ из последовательных программ с использованием системы SAPFOR. Разработаны алгоритмы и представления информации, необходимые для использования этого подхода. Реализация этих алгоритмов обеспечивает возможность автоматического конструирования LuNA-программы из последовательных программ для класса задач из области численного моделирования на примере метода Якоби решения систем линейных уравнений.

На защиту выносятся следующие положения.

1) Подход к автоматическому конструированию LuNA-программ на основе системы SAPFOR.
2) Представления информации о программе, необходимой для ее распараллеливания и генерации LuNA-программы. Форматы хранения этих представлений в компьютере.
3) Алгоритмы генерации LuNA-программ и алгоритмы определения информации, необходимой для этой генерации. Реализация алгоритмов генерации фрагментов кода.

В дальнейшем планируется реализовать алгоритмы генерации фрагментов вычислений и данных, а также алгоритмы определения информации, необходимой для генерации LuNA-программы. Также планируется развить разработанные алгоритмы и представления таким образом, чтобы они подходили для более широкого класса задач.

Выпускная квалификационная работа выполнена мной самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

____________________________ ____________________________

ФИО студента *Подпись студента*

« ____ »___________20 __г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Н.А. Коновалов, В.А. Крюков, С.Н. Михайлов, А.А. Погребцов. Fortran DVM - язык разработки мобильных параллельных программ // Программирование, М.: Изд-во РАН, 1995, № 1, С. 49-54.
2. Н.А. Коновалов, В.А. Крюков, Ю.Л. Сазанов. C-DVM - язык разработки мобильных параллельных программ // Программирование, М.: Изд-во РАН, 1999, № 1, С. 54-65.
3. Язык НОРМА / А.Н.Андрианов [и др.] // Препринты ИПМ им. М.В.Келдыша. 2019. № 132. 48 с. http://doi.org/10.20948/prepr-2019-132 URL: http://library.keldysh.ru/preprint.asp?id=2019-132.
4. Victor E. Malyshkin, Vladislav A. Perepelkin. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem // Parallel Computing Technologies. 11th International Conference, PaCT 2011, Proceedings. LNCS 6873. Springer, 2011. pp. 53-61.
5. Перепёлкин В.А. Система LuNA автоматического конструирования параллельных программ численного моделирования на мультикомпьютерах. [Текст] дис. ... канд. техн. наук: 05.13.11: защищена 21.02.2023: утв. 28.06.2023; [Место защиты: Институт вычислительной математики и математической геофизики]. — Н., 2023. — 135 с.
6. PLUTO - An automatic parallelizer and locality optimizer for affine loop [Электронный ресурс]. URL: nestshttps://pluto-compiler.sourceforge.net/ (дата обращения: 15.04.2024).
7. Официальная страница системы Par4All на github [Электронный ресурс]. URL: https://github.com/Par4All/par4all (дата обращения: 15.04.2024).
8. M. E. Kalender, C. Mergenci and O. Ozturk, "AutopaR: An Automatic Parallelization Tool for Recursive Calls," 2014 43rd International Conference on Parallel Processing Workshops, Minneapolis, MN, USA, 2014, pp. 159-165, doi: 10.1109/ICPPW.2014.32.
9. N.A. Kataev. Application of the LLVM Compiler Infrastructure to the Program Analysis in SAPFOR // Суперкомпьютерные дни в России: Труды международной конференции (24-25 сентября 2018 г., г. Москва), М.: Изд-во МГУ, 2018, P. 76-88.
10. Н.А. Катаев, А.С. Колганов. Дополнительное распараллеливание MPI программ с помощью системы SAPFOR // Вычислительные методы и программирование, М.: Изд-во МГУ, Том 22, Выпуск 4, С. 239-251.
11. Н.А. Катаев, С.А. Черных. Автоматизация распараллеливания программ в системе SAPFOR // Научный сервис в сети Интернет: труды XXII Всероссийской научной конференции, М.: ИПМ им. М.В.Келдыша, 2020, С. 362-376.

12. А.Д. Жуков, Н.А. Катаев, А.А. Смирнов. Динамический анализ зависимостей по данным в системе SAPFOR // Научный сервис в сети Интернет: труды XXI Всероссийской научной конференции, М.: ИПМ им. М.В.Келдыша, 2019, С. 400-412.
13. Н.А. Катаев. LLVM Based Approach to Static Analysis of C Programs in SAPFOR // 2018 Ivannikov Memorial Workshop Proccedings; IEEE Xplore Digital Library , Ереван: IEEE, 2019, С. 19-23.
14. А.А. Буланов, Н.А. Катаев. Автоматизация преобразований последовательных Фортран-программ, необходимых для их эффективного распараллеливания с помощью системы САПФОР // Сборник трудов Международной научной конференции "Параллельные вычислительные технологии 2015", Челябинск: Издательский центр ЮУрГУ, 2015, С. 172-177.
15. The LLVM Compiler Infrastructure [Электронный ресурс]. URL: https://llvm.org/ (дата обращения: 15.04.2024).
16. Barbara Chapman, Gabriele Jost, Ruud van der Pas. Using OpenMP: Portable Shared Memory Parallel Programming / Scientific and engineering computation. — The MIT Press, 2007. — 353 с. —ISBN 0262533022.
17. William Gropp, Ewing Lusk, Anthony Skjellum. Using MPI: Portable Parallel Programming with the Message-Passing Interface / MIT Press : 3 изд. — 2014. — 336 С. — ISBN-13 978-0262527392.
18. John Cheng, Max Grossman, Ty McKercher. Professional CUDA C Programming. — ISBN 978-1-118-73932-7. — 2014. — 528 с.
19. A. Cohen, S. Girbal, D. Parello, M. Sigler, O. Temam, and N. Vasilache. Facilitating the search for compositions of program transformations. In ICS, pages 151—160, June 2005.
20. Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P. (2008). Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In: Hendren, L. (eds) Compiler Construction. CC 2008. Lecture Notes in Computer Science, vol 4959. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-78791-4_9.
21. The Polyhedral Compiler Collection [Электронный ресурс]. URL: https://web.cs.ucla.edu/~pouchet/software/pocc/ (дата обращения: 15.04.2024).
22. Polly - Polyhedral optimizations for LLVM [Электронный ресурс]. URL: https://polly.llvm.org/ (дата обращения: 15.04.2024).
23. Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters / OSDI'04: Sixth Symposium on Operating System Design and Implementation. — 2004. — С. 137—150.

24. K. Shvachko, H. Kuang, S. Radia and R. Chansler, "The Hadoop Distributed File System," 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), Incline Village, NV, USA, 2010, pp. 1-10, doi: 10.1109/MSST.2010.5496972.
25. Пекунов В.В. Автоматическое распараллеливание С-программ с применением директив Cilk++ на базе распознающих объектно-событийных моделей // Программные системы и вычислительные методы. — 2018. - № 4. - С.124-133. DOI: 10.7256/2454-0714.2018.4.28086.
26. LLVM Language Reference Manual [Электронный ресурс]. URL: https://llvm.org/docs/LangRef.html (дата обращения: 15.04.2024).
27. Malyshkin, V. (2015). Active Knowledge, LuNA and Literacy for Oncoming Centuries. In: Bodei, C., Ferrari, G., Priami, C. (eds) Programming Languages with Applications to Biology and Security. Lecture Notes in Computer Science(), vol 9465. Springer, Cham. https://doi.org/10.1007/978-3-319-25527-9_19.
28. Кудрявцев А. А., Малышкин В. Э., Нуштаев Ю. Ю., Перепелкин В. А., Спирин В. А. Эффективная фрагментированная реализация краевой задачи фильтрации двухфазной жидкости // Проблемы информатики. 2023. № 2. С. 45-73. DOI: 10.24412/2073-0667-2023-2-45-73.
29. Описание массовых операторов LuNA [Электронный ресурс]. URL: https://gitlab.ssd.sscc.ru/luna/luna/-/wikis/luna_lang_v01 (дата обращения: 15.04.2024).
30. Малышкин В. Параллельные вычисления и представление активных знаний // Тр. 6-й междунар. конф. «Параллельные вычисления и задачи управления» PACO’2012., Москва, 24-26 окт. 2012 г. — М.: ИПУ РАН, 2012. — Том 1. — С. 70-77.
31. Артюхов А. А. Прототип базы активных знаний на основе вычислительных моделей // журнал "Проблемы информатики", 2021, № 4, с.5-15. DOI: 10.24412/2073-0667-2021-4-55-66.
32. CMake: A Powerful Software Build System [Электронный ресурс]. URL: https://cmake.org/ (дата обращения: 15.04.2024).
33. Официальная страница библиотеки nlohmann:json на github [Электронный ресурс]. URL: https://github.com/nlohmann/json.

ПРИЛОЖЕНИЕ А

Алгоритм определения теневых граней

В приложении описан формальный способ определения того, какие теневые грани нужны для выполнения фрагментов гнезд циклов. Для каждой пары фрагментируемых по пространству гнезда циклов и массива в общем виде определяется множество итераций фрагмента этого гнезда, и относительно этого множества определяется множество индексов, по которым в этом гнезде производится доступ на чтение к элементам фрагмента этого массива. Так, для гнезда вычислений и массива `A` из листинга 9 $\{(i, j) \mid \mathrm{INIT_I} \le i < \mathrm{END_I}, \mathrm{INIT_J} \le j < \mathrm{END_J}\}$ — множество итераций фрагмента гнезда циклов, где INIT_I, INIT_J, END_I и END_J — верхние и нижние границы циклов в рассматриваемом фрагменте гнезда циклов. Их конкретные значения при определении теневых граней не важны, важно лишь то, как соотносятся множество итераций и множество индексов. Так, для того же примера $\{(i, j) \mid \mathrm{INIT_I} - 1 \le i < \mathrm{END_I} + 1, \mathrm{INIT_J} - 1 \le j < \mathrm{END_J} + 1\}$ — множество индексов фрагмента гнезда вычислений.

Далее определяется разность множества индексов и множества итераций. Для рассматриваемого примера она равна $\{(\mathrm{INIT_I} - 1, j) \mid \mathrm{INIT_J} - 1 \le j < \mathrm{END_J} + 1\} \cup \{(\mathrm{END_I} + 1, j) \mid \mathrm{INIT_J} - 1 \le j < \mathrm{END_J} + 1\} \cup \{(i, \mathrm{INIT_J} - 1) \mid \mathrm{INIT_I} - 1 \le i < \mathrm{END_I} + 1\} \cup \{(i, \mathrm{END_J} + 1) \mid \mathrm{INIT_I} - 1 \le i < \mathrm{END_I} + 1\}$. Элементы этого множества будут называться *кортежами теневых индексов*. *Тип* кортежа теневых индексов определяется тем, какие его компоненты содержатся в множестве итераций соответствующих циклов, а какие — нет.

Далее составляется разбиение получившейся разности. Каждому подмножеству соответствует один тип кортежей теневых индексов. Для примера из листинга 9 таких подмножеств будет 8 (так как типов кортежей теневых индексов всего 8):

1) $\{(\mathrm{INIT_I} - 1, j) \mid \mathrm{INIT_J} \le j < \mathrm{END_J}\}$;
2) $\{(\mathrm{END_I} + 1, j) \mid \mathrm{INIT_J} \le j < \mathrm{END_J}\}$;
3) $\{(i, \mathrm{INIT_J} - 1) \mid \mathrm{INIT_I} \le i < \mathrm{END_I}\}$;
4) $\{(i, \mathrm{END_J} + 1) \mid \mathrm{INIT_I} \le i < \mathrm{END_I}\}$;
5) $\{(\mathrm{INIT_I} - 1, \mathrm{INIT_J} - 1)\}$;
6) $\{(\mathrm{INIT_I} - 1, \mathrm{END_J} + 1)\}$;
7) $\{(\mathrm{END_I} + 1, \mathrm{INIT_J} - 1)\}$;
8) $\{(\mathrm{END_I} + 1, \mathrm{END_J} + 1)\}$.

Для каждого подмножества из получившегося разбиения должен быть создан отдельный фрагмент данных, который будет содержать элементы массива, соответствующие

кортежам теневых индексов из этого подмножества. Эти фрагменты данных непосредственно соответствуют теневым граням.

Далее *толщиной теневой грани по оси k* будет называться модуль разности k-го индекса самых ближних к этой теневой грани элементов фрагмента массива и k-го индекса самых дальних от фрагмента массива элементов теневой грани.

Далее *толщиной теневой грани* будет называться кортеж, в котором i-ый элемент — толщина теневой грани по оси i. Так, для вышеописанных множеств толщины теневых граней следующие:

1) (1, 0);
2) (1, 0);
3) (0, 1);
4) (0, 1);
5) (1, 1);
6) (1, 1);
7) (1, 1);
8) (1, 1).

Далее *отрицательной теневой гранью по оси k* будет называться теневая грань, элементы которой соответствуют меньшим индексам, чем элементы фрагмента массива, по оси k, под *положительной* — грань, элементы которой соответствуют большим индексам, чем элементы фрагмента массива по оси k, под *нулевой* — грань, толщина которой по оси k равна 0.

Следует отметить, что существуют крайние случаи: фрагментам вычислений с номерами по оси k равным 0 или уменьшенным на единицу размером фрагментации по этой оси не требуется отрицательная теневая грань по этой оси, даже если всем остальным фрагментам вычислений она нужна. Так, для фрагмента вычислений с номером (0, 0) для примера из листинга 9 не нужны отрицательные теневые грани по осям 0 и 1. Обработка таких случаев не описывается, так как она имеет чисто технический характер.

ПРИЛОЖЕНИЕ Б

Алгоритм определения аргументов, передаваемых во фрагменты кода, выполняющие инициализацию и объединение теневых граней

Допустим, что фрагмент вычислений `a` выполняет инициализацию положительной/отрицательной по оси k теневой грани и при этом имеет по этой оси номер N. Допустим, что фрагмент вычислений `b` выполняет объединение отрицательной/положительной по оси k теневой грани и при этом имеет по этой оси номер (N + 1)/(N - 1). Тогда в вышеописанные фрагменты кода в качестве вышеописанных теневых граней передается один и тот же фрагмент данных. То есть базовое имя фрагментов кода, передаваемого в качестве вышеописанных аргументов должно быть одинаково, а пространственные индексы по k-оси должны отличаться на 1.

ПРИЛОЖЕНИЕ В

Пример LuNA-программы, в которой выполняется порождение фрагментов вычислений, выполняющих инициализацию и объединение теневых граней, для примера из листинга 9

Листинг 33 — Порождение фрагментов вычислений, выполняющих инициализацию и объединение теневых граней для примера из листинга 9

```
01: df A, A_merged,
02:   A_shadow_1_0, A_shadow_m1_0, A_shadow_0_1, A_shadow_0_m1,
03:   A_shadow_1_1, A_shadow_m1_1, A_shadow_1_m1, A_shadow_m1_m1;
04:
05: ...
06:
07: for f0=0..nf-1
08: {
09:   for f1=0..nf-1
10:   {
11:     init_A(f0, nf, f1, nf, A[f0][f1],
12:       A_shadow_1_0[f0][f1],
13:       A_shadow_m1_0[f0][f1],
14:       A_shadow_0_1[f0][f1],
15:       A_shadow_0_m1[f0][f1],
16:       A_shadow_1_1[f0][f1],
17:       A_shadow_m1_1[f0][f1],
18:       A_shadow_1_m1[f0][f1],
19:       A_shadow_m1_m1[f0][f1], ...);
20:     merge_shadow_A(f0, nf, f1, nf, A[f0][f1],
21:       A_shadow_m1_0[(nf + f0 - 1) % nf][f1],
22:       A_shadow_1_0[(f0 + 1) % nf][f1],
23:       A_shadow_0_m1[f0][(nf + f1 - 1) % nf],
24:       A_shadow_0_1[f0][(f1 + 1) % nf],
25:       A_shadow_m1_m1[(nf + f0 - 1) % nf][(nf + f1 - 1) % nf],
26:       A_shadow_1_m1[(f0 + 1) % nf][(nf + f1 - 1) % nf],
27:       A_shadow_m1_1[(nf + f0 - 1) % nf][(f1 + 1) % nf],
28:       A_shadow_1_1[(f0 + 1) % nf][(f1 + 1) % nf],
29:       A_merged[f0][f1]);
30:   }
31: }
32:
33: for f0=0..nf-1
34:   for f1=0..nf-1
35:     calc_B(f0, nf, f1, nf, A_merged[f0][f1],
36:       B[f0][f1], ...);
37:
38: ...
```

В рамках листинга 33:

- фрагменты данных `A_shadow`... соответствуют теневым граням массива `A`;
- фрагмент данных `A` соответствует необъединенному фрагменту массива `A`;

— фрагмент данных A_merged соответствует объединенному фрагменту массива A;
— фрагмент кода init_a выполняет инициализацию теневых граней массива A;
— фрагмент кода merge_shadow_A выполняет объединение теневых граней массива A;
— фрагмент кода calc_B выполняет фрагмент гнезда вычислений, в котором используются теневые грани массива A.

ПРИЛОЖЕНИЕ Г

Алгоритм определения временных индексов фрагментов данных

Для каждого фрагмента данных `df` определим:

1) Множество CF, элементами которого являются все операторы, описывающие применение фрагмента кода, в которые фрагмент данных `df` передается в качестве аргумента.
2) Множество OUT, элементами которого являются операторы, описывающие применение фрагмента кода, в которые фрагмент данных `df` передается в качестве выходного аргумента.
3) Множество MASS, элементами которого являются массовые LuNA-операторы `for` и `while`, в телах которых есть хотя бы один оператор из множества CF.
4) Множество U = CF ∪ MASS.

Для каждого элемента u множества U определяется временно́й номер. *Временной номер* (`df`, u) — это неотрицательное целое число, которое равно сумме количества операторов из множества OUT и операторов из множества MASS, которые находятся на том же уровне вложенности, что и u, и при этом описываемые ими фрагменты вычислений должны быть выполнены до того, как начнут вычисляться фрагменты вычислений, описываемые u.

Для примера из листинга 11 оператор, соответствующий `init_a`, будем называть `init_a`, соответствующий `loop_body`, — `loop_body`, а массовый оператор `for` — `luna_for`. Тогда:

1) CF = {`init_a`, `loop_body`}.
2) OUT = {`init_a`, `loop_body`}.
3) MASS = {`luna_for`}.
4) U = {`init_a`, `loop_body`, `luna_for`}.

Тогда временной номер для:

1) `init_a` равен 0, так как до порождаемых им фрагментов вычислений не должен быть выполнен ни один фрагмент вычислений или массовый оператор.
2) `luna_for` равен 1, так как до начала его выполнения должен выполниться один фрагмент вычислений — `init_a`.
3) `loop_body` равен 0, так как на уровне вложенности, на котором он находится (в теле `for`), до порождаемых им фрагментов вычислений не должен быть выполнен ни один фрагмент вычислений.

Далее для каждого оператора из CF формируется *кортеж временных номеров*. Длина этого кортежа равна увеличенному на единицу уровню вложенности, на котором находится этот фрагмент вычислений (считается, что минимальный уровень вложенности равен 0). Если описатель находится на уровне вложенности d, тогда d-ый элемент кортежа (считается, что они нумеруются с 0) равен временному номеру этого описателя, а любой другой i-ый элемент кортежа равен временному номеру массового оператора, который находится на i-ом уровне вложенности и объемлет рассматриваемый фрагмент вычислений.

Т.е. для примера выше кортеж временных номеров для:

1) `init_a` равен (0), так как `init_a` находится на 1-ом уровне вложенности и его временной номер равен 1.
2) `loop_body` равен (1, 0), так как он находится на уровне вложенности 2, имеет временной номер 0, а объемлющий массовый оператор for имеет временной номер 1.

С помощью кортежа временных номеров определяются индексы фрагментов данных, которые далее будут называться *временными*. Так, описание временных индексов фрагмента данных, который передается во фрагмент вычислений в качестве выходного аргумента с кортежем временных индексов длины d, приведено в следующем списке.

— Его 0-ый временной индекс (временные индексы нумеруются с 0) равен 0-ому элементу из кортежа временных номеров соответствующего описателя фрагментов вычислений.
— Если длина d не меньше двух, то его i-ый индекс, где i=1..d-1, равен <uses> * <index_var> + <current_use>, где
    - uses — количество элементов CF_out, находящихся на уровне вложенности i.
    - index_var — счетчик LuNA-оператора, который объемлет рассматриваемый оператор и находится на уровне вложенности i.
    - current_use — i-ый элемент кортежа временных номеров рассматриваемого описателя.

Если фрагмент данных передается в качестве входного аргумента, то его временной индекс будет таким же, как если бы он передавался в качестве выходного за исключением того, что его d-ый временной индекс уменьшается на 1.

ПРИЛОЖЕНИЕ Д

Алгоритм определения массовых операторов

В данном приложении описано, как генератор фрагментов вычислений и данных определяет массовые операторы.

Тривиальной является ситуация, когда массовый оператор не содержит ни одного условия прерывания и при этом шаг его счетчика равен 1, тогда тип определяемого массового оператора — `for`, шаблон заголовка которого приведен в листинге 34.

Листинг 34 — Шаблон массового оператора `for`

```
for <inductive_name> = <inductive_init> .. <<inductive_end>-1>
<body>
```

В шаблоне из листинга 34:

- `inductive_name` — имя счетчика;
- `inductive_init` — нижняя граница счетчика;
- `inductive_end` — верхняя граница счетчика.

Все эти параметры содержатся в схеме LuNA-программы.

Во всех остальных случаях создается оператор `while`, условием в заголовке которого является конъюнкция отрицаний всех условий прерывания оператора и условие `inductive_name[while_t] < inductive_end`, где:

- `inductive_name` — имя счетчика (из схемы LuNA-программы);
- `inductive_end` — верхняя граница счетчика (из схемы LuNA-программы);
- `while_t` — счетчик этого оператора `while`, начальное значение которого равно 0, а конечное будет помещено во фрагмент данных `while_final_t`.

Фрагменты данных `inductive_name` и `while_final_t` должны быть объявлены в начале головной подпрограммы, а значение фрагмента данных `inductive_name[0]` должно инициализироваться значением нижней границы, хранящимся в схеме LuNA-программы.

В теле оператора должна выполняться инициализация фрагмента данных `inductive_name[while_t + 1]` значением равным сумме `inductive_name[while_t]` и шага счетчика. Таким образом будет представлен шаг счетчика не равный 1.

Во всех операторах в теле этого массового оператора в качестве аргумента, соответствующего индукционной переменной исходного цикла, будет передаваться фрагмент данных `inductive_name[while_t]`.

Тело массового оператора строится практически так же, как и тело головной подпрограммы `main` за исключением особенностей, связанных с обработкой условий прерывания массового оператора.

Далее описывается, каким образом обрабатываются условия прерывания. Так как все условия прерывания, кроме самого первого, обрабатываются одинаково, дальнейшее описание приводится для одного зафиксированного условия. Определяется множество операторов, которые находятся в теле рассматриваемого массового оператора и описываемые ими фрагменты вычислений должны быть выполнены до вычисления рассматриваемого условия прерывания и после вычисления предыдущего условия прерывания. Все эти операторы копируются на уровень вложенности рассматриваемого массового оператора, при этом они обрамляются оператором `if`. Условием этого оператора является конъюнкция условия `while_final_t != (inductive.end - inductive.start) / step` и отрицаний всех условий прерывания, которые идут в теле массового оператора до рассматриваемого. Для самого первого условия прерывания обработка выполняется почти также, за исключением того, что копии операторов создаются для всех операторов, которые находятся в теле рассматриваемого массового оператора до первого условия прерывания.

Отдельно следует отметить случай, когда условие прерывания задается с помощью значения возвращаемого некоторой процедурой. В таком случае должен быть определен дополнительный фрагмент кода, который будет инициализировать значение выходного параметра значением возвращаемым этой процедурой. Также должен быть объявлен фрагмент данных, который будет использоваться в качестве рассматриваемого условия прерывания, его инициализацию будет выполнять фрагмент вычислений, соответствующий вышеописанному фрагменту кода.

Вышеописанные правила определения массового оператора `while` проиллюстрированы с помощью шаблона в листинге 35.

Листинг 35 — Шаблон массового оператора `while`

```
01: init(inductive_name[0], inductive_init);
02: while (inductive_name[while_t] < inductive_end
03:   && 0 == <break_cond_0> && ... 0 == <break_cond_N>),
04:   while_t = 0..out while_final_t
05: {
06:   init(inductive_name[while_t + 1],
07:     inductive_name[while_t] + inductive_step)
08:   ...
09: }
10:
11: ...
12:
13: if (while_final_t != (inductive_end - inductive_start) / step
14:   && 0 == <break_cond_0> && …
15:   && 0 == <break_cond_(k-1)>)
16: {
17:       <body_part_k>
18: }
19:
20: ...
```

В шаблоне из листинга 35:

- `inductive_name` — имя индукционной переменной
- `inductive_init` — нижняя граница счетчика
- `inductive_end` — верхняя граница счетчика
- `inductive_step` — шаг, с которым изменяется счетчик
- `<break_cond_k>` — k-ое условие прерывания;
- `<body_part_k>` — та часть тела массового оператора, которая находится между (k - 1)-ым и k-ым условием прерывания.

ПРИЛОЖЕНИЕ Е

Алгоритм определения параметров последовательных фрагментов кода

1) Определим список, который будет хранить все переменные последовательного фрагмента кода, причем значения этих переменных либо записываются до начала выполнения кода исходной программы, соответствующего этому фрагменту кода, либо считываются после. Хранение остальных переменных не имеет смысла, так как они станут локальными переменными фрагмента кода.
2) Формируется список `allVariables`, которому принадлежат все переменные каждого последовательного участка кода рассматриваемого фрагмента кода (их описание включено в информацию о параллелизме). Причем порядок следования переменных в данном списке соответствует порядку следования этих участков кода во фрагменте кода.
3) Для каждого уникального имени `name` переменной вышеописанного списка выполняеются следующие действия.
    a) Определим `nameVariables` — список переменных из `allVariables` с именем `name`. Порядок элементов в данном списке соответствует порядку элементов в списке `allVariables`.
    b) Если количество элементов в списке `nameVariables` — 1, то этот элемент добавляется в список всех переменных фрагмента кода.
    c) Иначе в список всех переменных фрагмента кода добавляется переменная с именем `name`, с типом, который указан в информации о параллелизме для каждой переменной `nameVariables` (он должен быть одинаков). Значение булевой переменной, указывающей на наличие записи значения переменной до рассматриваемой последовательности блоков равна значению этой булевой переменной для самой первой переменной в списке `nameVariables.` Значение булевой переменной, указывающей на наличие чтения значения переменной после рассматриваемой последовательности блоков равна значению этой булевой переменной для самой последней переменной в списке `nameVariables.`
4) Для каждого выделения памяти под массив в список всех переменных фрагментов кода добавляется переменная, соответствующая создаваемому массиву. Ее имя, тип и количество элементов массива берется из информации о параллелизме.

5) Для каждого доступа к элементу массива в список всех переменных фрагментов кода добавляется переменная, соответствующая массиву, к которому производится доступ, если такая переменная не была добавлена раньше, это определяется по наличию в списке переменной с именем массива, к элементу которого производится доступ.
6) Для каждого аргумента вызова процедуры, для которого в информации о параллелизме указано, что он является переменной, в список всех переменных фрагмента кода добавляется соответствующая переменная.
7) Для каждой переменной из списка всех переменных фрагмента кода формируется соответствующий этой переменной параметр фрагмента кода.

ПРИЛОЖЕНИЕ Ё

Определения процедур, предназначенных для выполнения фрагментации по пространству

Листинг 36 — Определения процедур, предназначенных для выполнения фрагментации по пространству

```
01: // процедура, вычисляющая размер компоненты фрагментируемого
02: // по пространству массива
03: int get_size(int f, int nf, int init_size)
04: {
05:   return (f + 1) * init_size / nf - f * init_size / nf;
06: }
07:
08: // процедура, вычисляющая верхнюю границу индукционной
09: // переменной фрагментируемого по пространству цикла
10: int get_inductive_end(int f, int nf, int real_end)
11: {
12:   return (f + 1) * real_end / nf - f * real_end / nf;
13: }
14:
15: // процедура, вычисляющая нижнюю границу индукционной
16: // переменной фрагментируемого по пространству цикла
17: int get_inductive_init(int f, int nf,
18:   int real_init, int real_end)
19: {
20:   int init = real_init - f * real_end / nf;
21:   return (init > 0) ? init : 0;
22: }
23:
24: // процедура, вычисляющая реальное значение индукционной
25: // переменной фрагментируемого по пространству цикла
26: int get_real_inductive_init(int f, int nf,
27:   int real_init, int real_end)
28: {
29:   return get_inductive_init(f, nf, real_init, real_end) +
30:     f * real_end / nf;
31: }
```

В листинге 36 представлены четыре процедуры, которые вычисляют:

- размер компоненты фрагментируемого по пространству цикла;
- верхнюю границу индукционной переменной фрагментируемого по пространству цикла;
- нижнюю границу индукционной переменной фрагментируемого по пространству цикла;
- реальное значение индукционной переменной фрагментируемого по пространству цикла.

В этих процедурах параметр:

- `f` соответствует номеру фрагмента вычислений;
- `nf` соответствует количеству фрагментов вычислений;
- `real_init` соответствует нижней границе индукционной переменной исходного цикла;
- `real_end` соответствует верхней границе индукционной переменной исходного цикла.

ПРИЛОЖЕНИЕ Ж

Последовательная программа, реализующая метод Якоби

Листинг 37 — Последовательная программа, реализующая метод Якоби

```
01: /* Jacobi-3 program */
02:
03: #include <math.h>
04: #include <stdio.h>
05:
06: #include <sys/time.h>
07:
08: #define Max(a, b) ((a) > (b) ? (a) : (b))
09:
10: #define L 384
11: #define ITMAX 100
12:
13: double B[L][L][L];
14: double A[L][L][L];
15:
16: double get_time()
17: {
18:   struct timeval tv;
19:   gettimeofday(&tv, NULL);
20:   return tv.tv_sec + ((double)tv.tv_usec) / 1000000.0;
21: }
22:
23: int main(int an, char **as)
24: {
25:   int i, j, k, it;
26:   double eps;
27:   double MAXEPS = 0.5;
28:   double start_t = get_time();
29:
30:   // гнездо циклов инициализации массивов A и B
31:   for (i = 0; i < L; i++)
32:     for (j = 0; j < L; j++)
33:       for (k = 0; k < L; k++)
34:       {
35:         A[i][j][k] = 0;
36:         if (i == 0 || j == 0 || k == 0 ||
37:           i == L - 1 || j == L - 1 || k == L - 1)
38:           B[i][j][k] = 0;
39:         else
40:           B[i][j][k] = 4 + i + j + k;
41:      }
42:
43: for (it = 1; it <= ITMAX; it++)
44: {
```

Продолжение листинга 37

```
45:     eps = 0;
46:     // гнездо циклов вычисления массива A
47:     for (i = 1; i < L - 1; i++)
48:       for (j = 1; j < L - 1; j++)
49:         for (k = 1; k < L - 1; k++)
50:         {
51:           double tmp = fabs(B[i][j][k] - A[i][j][k]);
52:           eps = Max(tmp, eps);
53:           A[i][j][k] = B[i][j][k];
54:         }
55:
56:     // гнездо циклов вычисления массива B
57:     for (i = 1; i < L - 1; i++)
58:       for (j = 1; j < L - 1; j++)
59:         for (k = 1; k < L - 1; k++)
60:           B[i][j][k] = (A[i - 1][j][k] + A[i][j - 1][k] +
61:             A[i][j][k - 1] + A[i][j][k + 1] +
62:             A[i][j + 1][k] + A[i + 1][j][k]) / 6.0;
63:
64:     printf(" IT = %4i   EPS = %14.7E\n", it, eps);
65:     if (eps < MAXEPS)
66:       break;
67:   }
68:
69:   double end_t = get_time();
70:   double t = end_t - start_t;
71:
72:   printf(" Jacobi3D Benchmark Completed.\n");
73:   printf(" Size             = %4d x %4d x %4d\n", L, L, L);
74:   printf(" Iterations       =       %12d\n", it - 1);
75:   printf(" Operation type   =     doubleing point\n");
76:   printf(" Verification     =       %12s\n",
77:     (fabs(eps - 5.058044) < 1e-4 ? "SUCCESSFUL" :
78:       "UNSUCCESSFUL"));
79:   printf(" Time             =       %.4f\n", t);
80:
81:   printf(" END OF Jacobi3D Benchmark\n");
82:   return 0;
83: }
```

Исходная версия программы взята из репозитория системы DVM в системе контроля версий git. Этот репозиторий доступ для скачивания в сети интернет[2].

[2] https://github.com/dvm-system

ПРИЛОЖЕНИЕ 3

ГЕНЕРАТОР ФРАГМЕНТОВ КОДА

РУКОВОДСТВО ПРОГРАММИСТА

Листов 7

Новосибирск 2024

АННОТАЦИЯ

В данном документе приведено руководство программиста для генератора фрагментов кода. Исходным языком программы является C++.

Основной функцией программы является создание файла с определением атомарных фрагментов кода.

Оформление программного документа «Руководство программиста» произведено по требованиям ГОСТ 19.504-79 «ЕСПД. Руководство программиста» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)».

СОДЕРЖАНИЕ

1 Назначения и условия применения программы....87
1.1 Назначение программы....87
1.2 Функции, выполняемые программой....87
1.3 Условия, необходимые для выполнения программы....87
2 Характеристика программы....88
2.1 Описание основных характеристик программы....88
2.2 Описание основных особенностей программы....88
3 Обращение к программе....89
3.1 Сборка и запуск программы....89
4 Входные и выходные данные....90
4.1 Организация используемой входной информации....90
4.2 Организация используемой выходной информации....90

1 Назначения и условия применения программы

1.1 Назначение программы

Генератор фрагментов кода предназначен для создания определений атомарных фрагментов кода, которые в совокупности с определениями фрагментов вычислений и данных представляют LuNA-программу.

1.2 Функции, выполняемые программой

Создание определений атомарных фрагментов кода исходя из их описания в схеме LuNA-программы.

1.3 Условия, необходимые для выполнения программы

Чтобы программа могла запускаться, необходимо наличие у программиста:

1) операционной системы Ubuntu;
2) компилятора gcc;
3) утилит CMake и make;
4) системы LuNA;
5) файла со схемой LuNA-программы в формате JSON.

2 Характеристика программы

2.1 Описание основных характеристик программы

Программа имеет один режим работы: считывание схемы LuNA-программы, ее разбор, создание определений фрагментов кода, формирование файла с определениями фрагментов кода.

Время работы программы зависит от:

1) количества фрагментов кода, которые нужно определить;
2) количества параметров в каждом фрагменте кода;
3) количества циклов в каждом фрагменте кода;
4) количества массивов в каждом фрагменте кода;
5) количества теневых граней массивов в каждом фрагменте кода;
6) количества доступов к элементам массивов в каждом фрагменте кода.

2.2 Описание основных особенностей программы

Программа принимает входные данные в виде файла, содержащего схему LuNA-программы в формате JSON.

Программа генерирует выходные данные в виде исходного файла на языке программирования C++. Этот файл содержит определения фрагментов кода.

3 Обращение к программе

3.1 Сборка и запуск программы

Сборка программы осуществляется с помощью утилиты CMake.

Пример bash-команд для выполнения сборки программы (исполнять эти команды нужно из корневой директории с исходными файлами генератора фрагментов кода):

```
mkdir build
cd build
cmake ..
cmake --build .
```

Результатом сборки становится исполняемый файл CppGenerator. При исполнении этого файла нужно передать два аргумента командной строки:

- Первый — позиционный, его значение — полное имя файла, содержащего схему LuNA-программы.
- Второй — передается с опцией -o, его значение — полное имя файла, в которое генератор фрагментов кода должен поместить их определение.

Пример запуска генератора: ./CppGenerator <LuNA_scheme_filename> -o <code_fragments_filename>, где:

- <LuNA_scheme_filename> — полное имя файла, содержащего схему LuNA-программы.
- <code_fragments_filename> — полное имя файла, куда генератора фрагментов кода поместит определения атомарных фрагментов кода.

4 Входные и выходные данные

4.1 Организация используемой входной информации

Входные данные — файл, содержащий схему LuNA-программы в формате JSON.

4.2 Организация используемой выходной информации

Выходные данные — файл, содержащий определения атомарных фрагментов кода на языке программирования C++.

ПРИЛОЖЕНИЕ И

ГЕНЕРАТОР ФРАГМЕНТОВ КОДА

ОПИСАНИЕ ПРОГРАММЫ

Листов 10

Новосибирск 2024

АННОТАЦИЯ

В данном документе приведено описание генератора фрагментов кода. Чтобы программа могла исполняться, необходимо наличие операционной системы Ubuntu, компилятора gcc, утилит CMake и make, системы LuNA, файла со схемой LuNA-программы в формате JSON. Оформление программного документа «Описание программы» произведено по требованиям ГОСТ 19.402-78 «ЕСПД. Описание программы» и ГОСТ 19.105-78 «Единая система программной документации (ЕСПД). Общие требования к программным документам (с Изменением N 1)».

СОДЕРЖАНИЕ

1 Общие сведения........94
1.1 Обозначение и наименование программы........94
1.2 Программное обеспечение, необходимое для функционирования программы........94
1.3 Языки программирования........94
2 Функциональное назначение........95
2.1 Назначение программы........95
3 Описание логической структуры........96
3.1 Алгоритм программы........96
3.2 Используемые методы........96
3.3 Структура программы........96
3.4 Связи между составными частями программы........96
3.5 Связи программы с другими........96
4 Используемые технические средства........97
5 Вызов и загрузка........98
6 Входные данные........99
7 Выходные данные........100

1 Общие сведения

1.1 Обозначение и наименование программы

Полное название — генератор фрагментов кода.

Обозначение — CppGenerator.

1.2 Программное обеспечение, необходимое для функционирования программы

Необходимое ПО — Ubuntu, gcc, CMake, make, LuNA.

1.3 Языки программирования

Язык программирования — C++.

2 Функциональное назначение

2.1 Назначение программы

Генератор фрагментов кода предназначен для создания определений атомарных фрагментов кода, которые в совокупности с определениями фрагментов вычислений и данных представляют LuNA-программу.

3 Описание логической структуры

3.1 Алгоритм программы

Определение фрагментов кода основывается на алгоритмах фрагментации по пространству, инициализации и объединения теневых граней, определении сигнатуры и тела фрагмента кода и определения процедур, выполняющих редукционные операции.

3.2 Используемые методы

Параллелизм выполнения вычислительноемких гнезд циклов достигается за счет использования фрагментации по пространству.

3.3 Структура программы

Программа состоит из набора классов, одни из которых хранят информацию об объектах, составляющих фрагменты кода, а другие выполняют преобразования этих объектов.

3.4 Связи между составными частями программы

Связь между модулями программы осуществляется при помощи вызова процедур и методов объектов.

3.5 Связи программы с другими

Для запуска программы необходимо наличие операционной системы Ubuntu, компилятора gcc, утилит CMake и make, системы LuNA, файла со схемой LuNA-программы в формате JSON.

4 Используемые технические средства

Режим работы — консольное приложение.

Для работы программы необходимо наличие операционной системы Ubuntu, компилятора gcc, утилит CMake и make, системы LuNA, файла со схемой LuNA-программы в формате JSON.

5 Вызов и загрузка

Сборка программы осуществляется с помощью утилиты CMake.

Пример bash-команд для выполнения сборки программы (исполнять эти команды нужно из корневой директории с исходными файлами генератора фрагментов кода):

```
mkdir build
cd build
cmake ..
cmake --build .
```

Результатом сборки становится исполняемый файл CppGenerator. При исполнении этого файла нужно передать два аргумента командной строки:

- Первый — позиционный, его значение — полное имя файла, содержащего схему LuNA-программы.
- Второй — передается с опцией -o, его значение — полное имя файла, в которое генератор фрагментов кода должен поместить их определение.

Пример запуска генератора: ./CppGenerator <LuNA_scheme_filename> -o <code_fragments_filename>, где:

- <LuNA_scheme_filename> — полное имя файла, содержащего схему LuNA-программы.
- <code_fragments_filename> — полное имя файла, куда генератора фрагментов кода поместит определения атомарных фрагментов кода.

6 Входные данные

Входные данные — файл, содержащий схему LuNA-программы в формате JSON.

7 Выходные данные

Выходные данные — файл, содержащий определения атомарных фрагментов кода на языке программирования C++.