МИНОБРНАУКИ РОССИИ
Федеральное государственное автономное образовательное
учреждение высшего образования
«Южный федеральный университет»

Институт математики, механики и компьютерных наук им. И. И. Воровича

На правах рукописи

Метелица Елена Анатольевна

АВТОМАТИЗАЦИЯ РАСПАРАЛЛЕЛИВАНИЯ ПРОГРАММ СО СЛОЖНЫМИ ИНФОРМАЦИОННЫМИ ЗАВИСИМОСТЯМИ

Специальность 2.3.5 – Математическое и программное обеспечение
вычислительных систем, комплексов и компьютерных сетей

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
Штейнберг Борис Яковлевич,
доктор технических наук,
старший научный сотрудник,
зав. каф. АДМ ИММиКН ЮФУ

Ростов-на-Дону – 2024

Оглавление

Список сокращений

ДВОР – диалоговый высокоуровневый оптимизирующий распараллеливатель

ОРС – оптимизирующая распараллеливающая система

OPS – Optimizing Parallelizing System

ПО – программное обеспечение

ЭВМ – электронная вычислительная машина

MIMD – Multiple Instruction, Multiple Data

SIMD – Single Instruction, Multiple Data

LLVM – Low Level Virtual Machine

GCC – GNU Compiler Collection

ICC – Intel C++ Compiler

Intel MKL – Intel Math Kernel Library

BLAS – Basic Linear Algebra Subprograms

ACML – AMD Core Math Library

AMD – Advanced Micro Devices

AST – Abstract Syntax Tree

Введение

Актуальность темы исследования

Существуют задачи, решение которых занимает много времени. К ним относятся численные методы решения задач математической физики, обработки изображений, машинного обучения, исследования космоса, создания новых лекарств, прогнозирования погоды и др. Иногда такие задачи не решаются из-за отсутствия подходящих высокопроизводительных вычислительных систем и соответствующего программного обеспечения.

Зачастую программным обеспечением используются не все возможности вычислительной системы [1]. Поэтому актуальной является проблема эффективного использования ресурсов компьютера. Есть библиотеки, ориентированные на решение часто используемых задач, например задачи линейной алгебры (библиотека BLAS). Такого рода библиотеки переписываются индивидуально для каждого нового процессора для достижения лучшей производительности. Например, в Intel MKL присутствует реализация библиотеки BLAS, оптимизированная для выполнения на процессорах с системой команд Intel x86; AMD ACML – версия библиотеки BLAS для процессоров AMD. Однако существует много задач, для которых не созданы библиотеки. Для создания высокопроизводительных программ, решающих такие задачи, могут использоваться оптимизирующие компиляторы. Компиляторы разрабатываются под конкретные архитектуры.

В то же время развивается разработка многоядерных процессоров (Colossus™ MK1 IPU и Colossus™ MK2 IPU [2], [3], Untether AI Boqueria [4], Moffett AI S4 Antoum [5], SambaNova SN30 RDU [6], Amazon Web Services Trainium1 [7], [8], Tesla Dojo D1 [9], К1879ВМ8Я (НТЦ «Модуль») [10]) под такие специфические задачи, как искусственный интеллект [11].

Для создания высокопроизводительных программ, решающих задачи, требующие больших объёмов вычислений, могут использоваться оптимизирующие компиляторы. Есть исследования 2018 года [1], которые показывают, что оптимизирующие компиляторы плохо оптимизируют программы. В основе оптимизирующих компиляторов лежит теория преобразования программ. Однако не все возможные оптимизирующие преобразования реализованы в современных компиляторах, тем более что усложнение вычислительных архитектур влечет усложнение оптимизирующих преобразований программ. В частности, в ускорении нуждаются численные алгоритмы решения дифференциальных уравнений в частных производных в задачах математического моделирования и другие алгоритмы, в основе которых лежат гнёзда циклов итерационного типа.

Создание эффективных распараллеливающих компиляторов сократит сроки разработки высокопроизводительных программ и понизит их себестоимость за счет снижения требований к квалификации разработчиков.

Степень разработанности темы

В середине XX века были написаны первые работы по автоматическому распараллеливанию. В эту область внесли вклад U. Bondhugula, M. Лэм. M. Вольф, P. Feautrier, В.Ю. Волконский [12], А.П. Ершов [13], В.А. Вальковский, Б.Я. Штейнберг, В.Н. Касьянов [14], В.А. Евстигнеев [15], А.И. Легалов, В.А. Крюков, В.В. Воеводин, В.Э. Малышкин [16], В.Г. Лебедев [17], R. Arora.

Автоматической оптимизации и распараллеливанию обычно предшествует «ручная» оптимизация программ, существенных результатов в которой достигли В.Д. Левченко, J. Dongarra, L. Lamport [18], [19], Н. Лиходед [20], [21], K. Goto [22], В.В. Корнеев и др. Эти работы представлены во многих обзорах: [23], [24], [25], [26], [27], [28].

Основы теории преобразований программ закладывались А.П. Ершовым в 70-х годах [29]. Теория преобразования программ развивалась во многих работах: [30], [31], [32], [33] и д.р.

Статья [14] посвящена алгоритмам распараллеливания циклов. В статье приводится описание основных алгоритмов распараллеливания циклов и даётся сопоставление их сильных и слабых сторон как на примерах, так и в сравнении «оптимальных» результатов, полученных при помощи этих алгоритмов.

К компилирующим инструментам, кроме компиляторов, относятся распараллеливающие системы ParaWise [34], LuNA, DVM [35], [36], [37], PLUTO, SUIF, OPC и др. Существуют разработки в области интерактивного распараллеливания, например, Interactive Parallelization Tool (IPT).

LuNA [38] представляет собой автоматизированную систему построения параллельных программ. Имеется поддержка GPU [39], однако оптимизирующие преобразования мало используются.

В работах [35], [36] идет речь о DVM-системе, в которой используются языки C-DVMH и Fortran-DVMH, являющиеся расширениями обычных языков программирования ФОРТРАН и Си за счёт дополнительных параллельных команд. Компиляторы данных языков преобразуют последовательную программу в параллельную за счёт технологий OpenMP, MPI, CUDA. В [40] говорится, что «DVMH-модель позволяет создавать эффективные параллельные программы (DVMH-программы) для гетерогенных вычислительных кластеров, в узлах которых в качестве вычислительных устройств, наряду с универсальными многоядерными процессорами, могут использоваться ускорители (графические процессоры или сопроцессоры Intel Xeon Phi). При

этом отображенные на узел вычисления могут автоматически распределяться между вычислительными устройствами узла с учетом их производительности». Данная система является полуавтоматической, поскольку программист самостоятельно редактирует программу, подставляя прагмы DVMH-компилятора, на основе которых строится параллельная программа.

Computer Aided Parallelization Tools (CAPTools) – это программный инструмент 1990-х годов, способный достаточно точно анализировать зависимости в последовательном коде и генерировать переносимый параллельный код (с директивами OpenMP или вызовами библиотеки MPI) полуавтоматическим (интерактивным) способом. Развитием данной системы является набор инструментов для автоматического распараллеливания ParaWise. Система ParaWise использует межпроцедурный и символьный анализы. Для распараллеливания кода, использующего директивы OpenMP, ParaWise анализирует последовательный код Fortran или C и генерирует код на основе директив Shared Memory. Так же, как и CAPTools, имеет графический интерфейс.

A Tool for Interactive Parallelization (IPT) [41] – интерактивный инструмент для преобразования последовательных программ в параллельные. Используются парадигмы программирования MPI, OpenMP, CUDA, гибридное программирование. В системе есть возможность выбора языка и парадигмы в командной строке или графическом интерфейсе. Используется компилятор ROSE [42] для генерации внутреннего представления – абстрактного синтаксического дерева. IPT также позиционируется как интерактивная система обучения параллельному программированию.

Приведённые выше разработки больше ориентированы на оптимизацию программ за счёт распараллеливания. Parawise и IPT анализируют информационные зависимости для автоматического распараллеливания, но в них не реализована возможность применения оптимизирующих преобразований.

Описанные выше интерактивные системы не подсказывают разработчику, в какие места его программы следует вставить прагмы распараллеливания.

Рассмотрим научно-исследовательские проекты, в которых реализуются оптимизирующие преобразования.

SUIF (Stanford University Intermediate Format) compiler [43] – система, позволяющая исследовать методы оптимизирующей компиляции для высокопроизводительных систем. Исследования, проводившиеся на основе данной системы, затрагивают такие темы, как: преобразование циклов (тайлинг) для повышения локальности и параллелизма, оптимизация скалярных выражений, планирование команд, глобальное распределение данных и вычислений для машин с общим и распределенным адресным пространством, приватизация массивов,

межпроцедурное распараллеливание, анализ указателей и оптимизация моделирования Verilog. SUIF также использовался для курсов по оптимизации компиляторов в университете Стэнфорда.

OPS (Optimizing Paralyzing System) [44] – оптимизирующая распараллеливающая система, проект, разрабатываемый в Институте математики, механики и компьютерных наук им. И.И. Воровича ЮФУ на кафедре алгебры и дискретной математики под руководством Б.Я. Штейнберга. Является основой для создания оптимизирующих компиляторов.

Внутреннее представление (ВП) OPC, как и в SUIF и ROSE, является высокоуровневым. Преимущество заключается в том, что из высокого уровня удобно генерировать граф вычислений, который является промежуточным представлением для генерации HDL-описания конвейеров (это удобнее, чем из низкоуровневого регистрового конвейера). Высокоуровневое внутреннее представление OPC, которое имеет древовидную структуру и позволяет на своей основе реализовывать различные преобразования циклов, линейных участков программ, условных операторов и др. А также графовые модели, такие как граф информационных зависимостей, граф вычислений, управляющий граф программы, граф вызова подпрограмм и др.

С одной стороны, высокий уровень внутреннего представления сужает класс входных языков, но с другой – расширяет класс целевых архитектур.

На основе OPC реализованы такие приложения, как ТПП и ДВОР.

Тренажёр параллельного программиста (ТПП) [45] – это электронное обучающее средство, ориентированное на обучение преобразованию последовательных программ в параллельные. Такая цель представляется вполне оправданной, поскольку прежде, чем писать параллельную программу, многие программисты, особенно начинающие, уже, как правило, имеют ее последовательный прототип. В ходе работы с ТПП пользователь может анализировать зависимости в своей программе с помощью графовых моделей и изменять их, используя преобразования программ, встроенные в ТПП, что позволяет новому в этой области программисту познакомиться со спецификой разработки оптимизирующих компиляторов на практике, чему также способствуют контекстные подсказки и справочные материалы.

Диалоговый высокоуровневый оптимизирующий распараллеливатель (ДВОР) основан на внутреннем представлении OPC (версия 5), выполняет те же функции, что и ТПП, так же используется для тестирования новых возможностей OPC [46], [47], [48], [49], [50].

«Тайлинг» является важным преобразованием, которое впервые представлено в работе [51]. В рамках этой работы вводится преобразование supernode partitioning, которое реализует идею группировки итераций гнезда циклов для повышения локальности данных путём разбиения пространства итераций тесного гнезда циклов на «суперузлы», которые будут атомарно выполняться процессором.

Майкл Вольф развивает исследование тайлига за счёт использования дополнительных преобразований (Loop Interchanging, Loop Skewing, Strip mining, tiling) и рассматривает нетесные гнёзда циклов. В статье [52] описывается преобразование тайлинг (tiling) как комбинация Strip mining и Loop interchanging. Приводятся задачи оптимизации: атомарность тайлов, параллелизм и локальность между тайлами, векторизация и повышение локальности внутри тайлов. Описывается алгоритм оптимизации программы с помощью тайлинга, а также приводятся результаты численных экспериментов, что подтверждает преимущества использования тайлинга. Приводится математическая формулировка реиспользуемости (повторного использования данных) и локальности данных. Теория преобразований программ, описанная а статье, использует преобразования унимодулярных матриц.

В следующей статье [53] подробно рассматривается проблема кэш-промахов. Приводится модель кэш-промахов, влияние шага доступа данных и размера тайлов на кэш-промахи. Вычисление максимально возможного размера тайла (without self interference). Представленные теоретические исследования подтверждаются приведёнными численными экспериментами. Дальнейшие исследования анализа кэш-промахов приведены в статье [54].

Гнёзда циклов – высоконагруженные части программы, для оптимизации которых применяются блочные алгоритмы. Основными целями использования блочных алгоритмов, в частности преобразования тайлинг, являются повышения локальности данных [52], [55], подготовка программы для дальнейшего параллельного исполнения полученных блоков [56], [57].

Выбор формы тайла обуславливается информационными зависимостями в гнезде циклов алгоритма. В работах [58], [59], [60], [61], [62] рассматривается непрямоугольный тайлинг, который в сочетании с распараллеливанием дает ускорение в несколько раз для сеточных методов решения различных классических уравнений математической физики. В работе [63] рассматривается алгоритм скошенного тайлинга для нетесных гнёзд циклов, данная модификация позволяет расширить класс преобразуемых программ.

В статье [58] показано ускорение (в 2,9 раза) модифицированного алгоритма Гаусса – Зейделя для решения двумерной задачи Дирихле после разбиения пространства итераций (тройное гнездо циклов, метод гиперплоскостей в сочетании с одномерным тайлингом), ускорение (в 2,4 раза) алгоритма Гаусса – Зейделя (трёхмерная задача Дирихле) после разбиения пространства итераций (4-мерное гнездо циклов, метод гиперплоскостей, двумерный прямоугольный тайлинг) в трехмерной задаче.

В статье [59] рассматривается ускорение решения волнового уравнения с помощью алгоритма разбиения DiamondTorre, который разработан с учётом особенностей иерархии памяти

и параллельности графических процессоров общего назначения (GPGPU (GTX 750Ti, GTX 970, TitanZ)) (иерархический тайлинг).

В статье [64] рассматривается ускорение (в 3 раза) задачи моделирования акустической волны (4-мерное гнездо циклов) с помощью Time-tiling (выравнивание информационных зависимостей с помощью преобразования Loop Skewing и тайлинга по внешнему циклу, отвечающему за время).

В статье [65] рассматривается ускорение трёхмерных задач, таких как 3-мерная задача Якоби (3-мерное гнездо цикла, двумерный тайлинг, ускорение 13%), Red-black SOR (3-мерное гнездо цикла, трёхмерный тайлинг, ускорение 89%), Multigrid (RESID) (3- мерное гнездо цикла, двумерный тайлинг, ускорение 16%), с помощью прямоугольного тайлинга.

Повышение сложности архитектурных решений вычислительных систем дало толчок к развитию тайлинга.

В статьях [66], [67], [68], [69], [70] рассматривается иерархический тайлинг соответствующий иерархической организации современных вычислительных систем.

Преобразование тайлинг с перекрытиями рассматривается в работах: [71], [72], [73].

Иерархический тайлинг с перекрытиями позволяет сбалансировать накладные расходы на коммуникацию и избыточные вычисления, и, таким образом, может обеспечить более высокую производительность. В статье [74] описывается иерархический тайлинг с перекрытиями и его реализация в компиляторе. Другие работы, такие как [75] и [76], описывают модели производительности для прогнозирования оптимального объема избыточных вычислений для распределённых систем или графических процессоров.

Тайлинг используют системы PLUTO, PolyMage, SUIF, Polly-llvm, MLIR и др.

Современные компиляторы (ICC, GCC, LLVM, PGI-Compiler, ROSE-Compiler и др.) хорошо оптимизируют базовые блоки и самые глубоко вложенные циклы гнезда. Эти компиляторы являются многопроходными, но часто не находят оптимальные цепочки преобразований.

GCC [77] – оптимизирующий компилятор с открытым кодом для языков C, C++, Objective-C, Fortran, Ada и др. Данный компилятор позволяет применять оптимизирующие преобразования для циклов, таких как loop interchange, unroll and jam, loop fusion, loop fission, loop reversal and loop skewing. Существует возможность применения за счёт опций компилятора и директив компилятора, которые пользователь должен самостоятельно подставить в текст программы. Например, «#pragma GCC optimize (“unroll-loops”)» или «__attribute__((optimize(“unroll-loops”)))».

LLVM [78] (Low Level Virtual Machine) – проект программной инфраструктуры для создания компиляторов и сопутствующих утилит. Представляет собой набор компиляторов из языков высокого уровня, библиотек, систем оптимизаций, интерпретаторов и компиляторов в машинный код. Clang [79] – это транслятор, предоставляющий языковой интерфейс и инфраструктуру инструментов для C-подобных языков, созданный специально для работы на базе LLVM. Комбинация Clang и LLVM представляет собой полноценный компилятор. Компилятор Clang LLVM с открытым кодом. В [80] описан перечень реализованных преобразований циклов в компиляторе LLVM, а также представлены примеры применения для лучшей иллюстрации. Преобразования Loop Unroll (-and-Jam), Loop Unswitching, Loop Interchange, Detection of memcpy, memset idioms, Delete side-effect free loops, Loop Distribution, Loop Vectorization реализованы в LLVM. Поддерживаются директивы компилятора для оптимизации циклов. Для реализации преобразований Loop Interchange, Skewing (Wavefront), Strip Mining (Vectorization), Tiling, Unroll-and-Jam, Loop Distribution, Index Set Splitting, Loop Fusion используется модель, в которой пространство итераций представлено выпуклым многогранником (The Polyhedral Model). Поддержка данных преобразований обеспечивается с помощью высокоуровневого оптимизатора Polly. Polly-LLVM [81] [82] – это высокоуровневый оптимизатор циклов для LLVM, использующий абстрактное математическое представление на основе выпуклых многогранников для анализа и оптимизации схемы доступа к памяти программы.

Компиляторы PGI включают глобальную оптимизацию, векторизацию, программную конвейерную обработку и возможности распараллеливания с общей памятью, ориентированные как на процессоры Intel, так и на AMD. PGI поддерживает следующие языки высокого уровня: Fortran 77, Fortran 90/95/2003, Fortran 2008 (partial), High Performance Fortran (HPF), ANSI C99 with K&R extensions, ANSI/ISO C++, CUDA, Fortran, OpenCL, OpenACC, OpenMP. Некоторые из представленных компиляторов были переименованы и интегрированы в Nvidia HPC SDK. Nvidia HPC SDK [83] реализует в себе оптимизации циклов, таких как Unrolling, Vectorization и Parallelization.

Фреймворк PLUTO [84] позволяет автоматически преобразовывать нетесные гнёзда циклов для параллельного выполнения и повышения локальности данных при помощи блочных преобразований на основе модели многогранника [85]. Фреймворк проводит статический анализ информационных зависимостей входной программы и подбирает комбинацию аффинных преобразований для эффективного применения тайлинга (ключевое преобразование), теоретическое описание данного подхода подробно описано в статье [86]. Для генерации преобразованного кода из внутреннего представления фреймворка используется инструмент

ClooG-ISL. Присутствует автоматическая генерация параллельного OpenMP-кода для многоядерных процессоров. Производится подготовка преобразованного кода для дальнейшей векторизации при помощи компилятора за счёт перестановки циклов внутри тайлов и добавления директив компилятора. Также описывается алгоритм генерации конвейерного параллельного кода. Дальнейшим развитием этого фреймворка является PLUTO+ [87]. Новый подход позволяет коэффициентам преобразований в рамках модели полиэдра быть отрицательными, что расширяет пространство применяемых аффинных преобразований по сравнению с PLUTO, также усовершенствован алгоритм моделирования последовательности преобразований. Эти изменения позволили реализовать ромбовидный тайлинг, представленный в статье [88], который обеспечивает лучшую балансировку нагрузки при распараллеливании. В статье [89] описывается подход к определению оптимальных размеров тайлов, учитывающий и временную, и пространственную локализации данных, есть возможность указывать размеры тайлов вручную.

В статье [90] представлен дизайн и реализация PolyMage, предметно-ориентированного языка и компилятора для конвейеров обработки изображений. Конвейер обработки изображений можно рассматривать как график взаимосвязанных этапов, которые последовательно обрабатывают изображения. А также рассматривается проблема малой пропускной способности памяти, которая препятствует эффективному использованию параллелизма.

TIRAMISU [91] – система, предназначенная для создания высокопроизводительного кода для различных платформ, включая многоядерные, графические процессоры и распределенные системы. Данная система предназначена для решения задач обработки изображений, линейной алгебры и глубокого обучения. Для оптимизации программ используется язык планирования, на основе описанных команд генерируется преобразованная программа.

В статье [92] рассматриваются алгоритмы итерационного типа, такие как дискретный оператор Лапласа (Discrete Laplace operator), оператор дивергенции, оператор градиента, алгоритм итерационного типа для имитации распределения температуры в организме человека при лечении рака гипертермией [93], [94].

Дискретный оператор Лапласа часто используется в обработке изображений, например, в задаче выделения границ [95] или в приложениях оценки движения. Дискретный лапласиан определяется как сумма вторых производных и вычисляется как сумма перепадов на соседях центрального пиксела. В обработке изображений дискретный оператор Лапласа применяется в виде фильтра как свёртка. Приложения уравнений с оператором Лапласа рассматриваются в: [96], [97], [98].

Подходы к оптимизации гнёзд итерационного типа приводятся в работах: [72], [99], [100], [101], [102]. Параллельный алгоритм на кластере из 8 узлов для решения двумерной задачи

Дирихле методом Гаусса – Зейделя представлен в [103]. В статье [104] рассматриваются подходы к построению высокоэффективных параллельных алгоритмов для численного решения краевых задач, на примере односеточного варианта метода Зейделя.

Существуют системы, ориентированные на оптимизацию гнёзд итерационного типа. В статье [105] представлена среда автоматизации Senju для ускорителей ISL на базе FPGA. В статье [106] представлена система, масштабируемая в автоматическую среду ускорения гнёзд итерационного типа на современных ПЛИС – SASA. В статье [107] представлена система AN5D, которая способна автоматически преобразовывать и оптимизировать гнёзда итерационного типа в заданном исходном коде C и генерировать соответствующий код CUDA. В статье [108] приводится система для оптимизации итерационных алгоритмов на многоядерные процессоры.

В данной работе предлагаются новые методы ускорения программ, в частности гнезд циклов итерационного типа, которые часто встречаются в задачах математического моделирования. Эти методы основаны на анализе сложных информационных зависимостей в программе. Используется и обобщается понятие вектора расстояния зависимости, которое определяется на основе решетчатых графов. В тех случаях, когда компилятору недостаточно информации для определения зависимости, предлагается использовать метод диалогового уточнения зависимостей, использующий символьный анализ текстов программ. Представленные алгоритмы автоматизации ускорения программ иллюстрируются на численных методах решения задач математической физики.

Цель диссертации состоит в разработке методов создания инструментов автоматизированной оптимизации программ, в частности, включающих гнёзда циклов итерационного типа.

Для достижения поставленной цели в рамках данной работы решаются следующие **задачи**:

1. Разработать на основе внутреннего представления оптимизирующей распараллеливающей системы (OPC) алгоритмы автоматизации выполнения преобразований тесных гнезд циклов: «скашивание» (Loop Skewing), «тайлинг» (Loop Tiling) и «метод гиперплоскостей» (Wavefront).
2. Разработать метод ускорения гнёзд циклов итерационного типа, включающий преобразования «скашивание», «тайлинг», «метод гиперплоскостей» и дополнительные оптимизации выражений и циклов.
3. Разработать метод диалогового анализа и преобразований текстов программ на основе символьного анализа.

Методы исследований

Поставленные задачи решались посредством методов теории преобразования программ, теории графов, теории языков программирования, линейной алгебры. Представленные в диссертации алгоритмы программно реализованы на основе методов объектно-ориентированного программирования на языке C++.

Научная новизна

Разработана цепочка оптимизирующих преобразований гнёзд циклов итерационного типа, основанная на универсальном высокоуровневом древовидном внутреннем представлении. Такой подход отличается от известного метода, реализованного на полиэдральном внутреннем представлении, наличием возможности использовать дополнительные оптимизирующие преобразования для ускорения программ.

Разработан и теоретически обоснован новый метод обхода точек тайла, который повышает временную локальность данных и даёт ускорение.

Предложен метод определения оптимальных размеров тайлов для получения наилучшего ускорения.

Предложен метод диалогового уточнения информационных зависимостей, позволяющий распараллеливать более широкий класс циклов, чем позволяли прежние методы.

Теоретическая и практическая значимость работы

Теоретическая значимость состоит в развитии теории преобразования программ для компилирующих систем с высокоуровневым универсальным (древовидным) внутренним представлением:

- приводится обобщение вектора расстояний, которое необходимо для обоснования целевых преобразований гнёзд циклов итерационного типа;
- теоретически обосновывается целесообразность применения перестановки циклов внутри тайла;
- обосновывается эквивалентность преобразования "метод гиперплоскостей" с предлагаемым в данной работе вектором нормали;
- доказано, что результирующая и исходная программы имеют одинаковые условия сходимости (для итерационных алгоритмов), одинаковые погрешности машинных округлений и одинаковые условия переполнения регистров (превышения максимальных размеров типов данных);

- предложены методы расчета оптимальных размеров тайлов;
- предложен новый метод диалоговой оптимизации и распараллеливания циклов основанный на символьном анализе.

Реализация преобразований программ для ускорения гнёзд циклов на основе ОРС позволяет расширить множество оптимизируемых программ и повысить их ускорение. Разработана «Программа, реализующая параллельный алгоритм Гаусса – Зейделя для задачи Дирихле», которая может служить прототипом решателя промышленного пакета прикладных программ для решения задач математического моделирования. Разработан диалоговый анализатор, который может рассматриваться как прототип блока полуавтоматического анализа программ в перспективных инструментах оптимизации. Реализованные преобразования программ «метод гиперплоскостей» и «скошенный тайлинг» в составе ОРС могут войти в состав распараллеливающих компиляторов для перспективных вычислительных систем.

Результаты работы использовались при выполнении грантов:

- Российского научного фонда № 22-21-00671,
- Правительства РФ № 075-15-2019-1928.

Основные положения, выносимые на защиту:

1. Предложена цепочка преобразований гнезд циклов итерационного типа, которая дает более высокое ускорение, чем известные методы; доказана эквивалентность данной цепочки.
2. Предложен метод диалогового анализа текстов программ, основанный на символьном анализе, который расширяет применимость оптимизирующих и распараллеливающих преобразований программ.
3. Предложен подход к реализации преобразований гнёзд циклов «скашивание циклов», «метод гиперплоскостей», «скошенный тайлинг» на основе внутреннего представления ОРС, что позволяет расширять цепочки из этих преобразований программ дополнительными преобразованиями, реализованными на основе универсального древовидного внутреннего представления.
4. Предложены методы обхода тайлов и определения их размеров, которые ускоряют выполнение программ.

Достоверность полученных результатов подтверждается корректным использованием математических формулировок и результатов теории преобразования программ, линейной

алгебры, теории графов; тестированием корректности и производительности разработанного ПО; успешным применением разработанного ПО в научной деятельности; полученными и опубликованными результатами численных экспериментов по ускорению целевых алгоритмов.

Апробация работы

Результаты научной работы были апробированы на:

1) Всероссийской XXVII научной конференции «Современные информационные технологии: тенденции и перспективы развития» в 2020 году, доклад «Использование тайлинга для оптимизации итерационных алгоритмов»; (доклад без соавторов)

2) Национальном суперкомпьютерном форуме (НСКФ 2020) с докладом «Использование блочных алгоритмов для оптимизации гнёзд циклов»; (доклад без соавторов)

3) конференции 16th International Conference on Parallel Computing Technologies (PaCT-21) с докладом «Precompiler for the ACELAN-COMPOS package solvers»; (вклад соискателя «методы оптимизации гнезд циклов итерационного типа и результаты численных экспериментов»)

4) Национальном суперкомпьютерном форуме (НСКФ 2021) с докладами «О сочетании распараллеливания и скошенного тайлинга» (вклад соискателя «методы оптимизации гнезд циклов итерационного типа, новый метод обхода точек тайла и результаты численных экспериментов для алгоритма Гаусса-Зейделя решения двумерной задачи Дирихле»), «О высокоуровневом автоматическом оптимизаторе программ» (вклад соискателя «методы оптимизации гнезд циклов итерационного типа, новый метод обхода точек тайла и результаты численных экспериментов для алгоритма Гаусса-Зейделя решения двумерной задачи Дирихле и обобщённой задачи Дирихле»

5) Национальном суперкомпьютерном форуме (НСКФ 2022) с докладом «Автоматизация распараллеливания программ с оптимизацией размещения и пересылок данных»; (вклад соискателя «анализ возможности распараллеливания алгоритма Гаусса-Зейделя для задачи Дирихле на вычислительной системе с распределенной памятью»)

6) конференции International scientific workshop ОТНА Spring 2023 с докладом «Multidimensional convolution»; (вклад соискателя «методы ускорения алгоритма Гаусса-Зейделя для двумерной задачи Дирихле»)

7) конференции International Conference on Parallel Computing Technologies (PaCT-23) с докладом «Automatic Parallelization of Iterative Loops Nests on Distributed Memory Computing Systems»; (вклад соискателя «методы ускорения алгоритмов итерационного типа»)

8) конференции 12th International Young Scientists Conference in Computational Science с докладом «Combination of parallelization and skewed tiling» (вклад соискателя «методы ускорения алгоритмов итерационного типа»).

По теме диссертации опубликовано 11 работ, из которых 6 – статьи в изданиях, индексированных в БД Scopus или WoS, 2 статьи в журналах перечня ВАК, 1 свидетельство о государственной регистрации программы для ЭВМ, а остальные индексированы в РИНЦ.

Основные публикации автора по теме работы:

Публикации, индексируемые в международных базах данных Scopus, WoS:

1. Baglij A., Mikhailuts Y., Metelitsa E., Ibragimov R., Steinberg B., Steinberg O. On the Development of the Compiler from C to the Processor with FPGA Accelerator // Frontiers in Software Engineering Education. FISEE. Lecture Notes in Computer Science. 2019. Vol. 12271. Springer, Cham. [Электронный ресурс]: URL: https://doi.org/10.1007/978-3-030-57663-9_25 (дата обращения: 01.04.2024). (Scopus, Q2, РИНЦ).

2. Vasilenko A., Veselovskiy V., Metelitsa E., Zhivykh N., Steinberg B., Steinberg O., Precompiler for the ACELAN-COMPOS Package Solvers, Parallel Computing Technologies // PaCT. Lecture Notes in Computer Science. 2021. Vol. 12942. Springer, Cham. [Электронный ресурс]: URL: https://doi.org/10.1007/978-3-030-86359-3_8 (дата обращения: 01.04.2024). (Scopus, Q2, РИНЦ).

3. Bagliy A.P., Metelitsa E.A., Steinberg B.Y. Automatic Parallelization of Iterative Loops Nests on Distributed Memory Computing Systems // Parallel Computing Technologies. PaCT. Lecture Notes in Computer Science. 2023. Vol. 14098. Springer, Cham. [Электронный ресурс]: URL: https://doi.org/10.1007/978-3-031-41673-6_2 (дата обращения: 01.04.2024). (Scopus).

4. Gervich L., Metelitsa E., Steinberg B. Combination of parallelization and skewed tiling // Procedia Computer Science. 2023. Vol. 229. P. 228–235.

5. Steinberg B., Baglij A., Petrenko V., Burkhovetskiy V., Steinberg O., Metelica E. An Analyzer for Program Parallelization and Optimization // Proceedings of the 3rd International Conference on Applications in Information Technology (ICAIT'2018). New York: Association for Computing Machinery, 2018. P. 90–95. (Scopus, WoS, РИНЦ).

6. Gervich L.R., Guda S.A., Dubrov D.V., Ibragimov R.A., Metelitsa E.A., Mikhailuts Y.M., Paterikin A.E., Petrenko V.V., Skapenko I.R., Steinberg B.Ya., Steinberg O.B., Yakovlev V.A., Yurushkin M.V. How OPS (optimizing parallelizing system) may be useful for clang // Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR 17). New York: Association for Computing Machinery, 2017. P. 1–8. (Scopus, WoS, РИНЦ).

Публикация из перечня ВАК:

7. Метелица Е.А. Обоснование методов ускорения гнёзд циклов итерационного типа // Программные системы: теория и приложения. 2024. Т. 15. № 1. С. 63–94. (К2)

8. Метелица Е.А., Штейнберг Б.Я. Об ускоряющих преобразованиях программ для решения обобщенной задачи Дирихле // Вычислительные методы и программирование. 2024.25, № 3. С. 292-301. (К1)

Свидетельство о государственной регистрации программ:

9. «Программа, реализующая алгоритм Гаусса – Зейделя для задачи Дирихле» №2021681898 от 10 января 2022 г. Василенко А.А., Метелица Е.А., Штейнберг Б.Я. (РИНЦ).

Другие публикации:

10. Метелица Е.А. Использование тайлинга для оптимизации итерационных алгоритмов // Современные информационные технологии: тенденции и перспективы развития: материалы XXVII научной конференции, г. Ростов-на-Дону – Таганрог, 24–26 сентября 2020 г. Ростов н/Д. – Таганрог: Изд-во ЮФУ, 2020. с. 185–188. (РИНЦ).

11. Метелица Е.А., Морылев Р.И., Петренко В.В., Штейнберг Б.Я. Основанная на OPC система обучения преобразованиям программ «Тренажер параллельного программиста», PLC-2017 / Всероссийская научная конференция памяти А.Л. Фуксмана. Ростов н/Д., 2017. с. 198. (РИНЦ).

Соответствие специальности

По своему научному содержанию диссертационная работа соответствует паспорту специальности 2.3.5. «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей», пунктам: №1 «Модели, методы и алгоритмы проектирования, анализа, трансформации, верификации и тестирования программ и программных систем»; №2 «Языки программирования и системы программирования, семантика программ»; №8 «Модели и методы создания программ и программных систем для параллельной и распределенной обработки данных, языки и инструментальные средства параллельного программирования».

Личный вклад автора

Автором на основе теории оптимизирующих преобразований программ разработан и реализован алгоритм автоматической оптимизации (и, в частности, распараллеливания) гнёзд циклов итерационного типа. Проведены численные эксперименты преобразованных программ. В статьях с соавторами результаты, относящиеся к оптимизации алгоритмов итерационного типа,

принадлежат автору. Сформулированы и доказаны условия эквивалентности цепочек преобразований.

Задача поставлена научным руководителем. В диссертации использована оптимизирующая распараллеливающая система, которая разрабатывалась многими студентами и аспирантами мехмата ЮФУ.

Структура и объём работы

Диссертация состоит из введения, четырёх глав, заключения, списка литературы из 148 наименований. Текст диссертации изложен на 152 (182 с приложениями) страницах и содержит 91 рисунков, 20 таблиц, 41 пример, 18 приложений.

Основное содержание работы

Во введении приводится обзор работ по теме исследования, обосновывается актуальность темы, формулируются цели и задачи исследования, описываются методы исследования, излагаются основные положения научной новизны и значение работы для теории и практики. Приведён список положений, выносимых на защиту. Изложена структура диссертации и краткое содержание работы по главам.

Первая глава носит вспомогательный характер. В ней описываются алгоритмы итерационного типа, приводятся элементы теории преобразования программ, граф информационных зависимостей, решетчатый граф, анализ информационных зависимостей с помощью векторов расстояний (distance vector), описание и анализ блочных и вспомогательных преобразований гнёзд циклов с использованием унимодулярных матриц преобразований. От определения решетчатых графов зависит определение вектора расстояний, что требуется для обоснования корректности алгоритма, описанного в главе 2, и не обязательно для понимания основных результатов работы, описанных в списке положений, выносимых на защиту, поэтому может быть опущено при первом прочтении. Приводится описание оптимизирующей распараллеливающей системы (ОРС).

Во второй главе приводится разработанный алгоритм оптимизации и распараллеливания итерационных гнёзд циклов, а также пример применения этого алгоритма к двумерному гнезду циклов. Описывается метод изменения обхода точек тайла, повышающий временную локальность данных. Обосновывается целесообразность использования этого метода. Приводятся преимущества высокоуровневого внутреннего представления для преобразования программ. Рассматриваются перспективы комбинации приведённого алгоритма и размещения с перекрытиями для распараллеливания на системы с распределённой памятью.

В третьей главе приводятся численные эксперименты выполнения программ, реализующих алгоритмы итерационного типа, до и после преобразования с помощью разработанного алгоритма. Рассмотрены алгоритм Гаусса – Зейделя для решения задачи Дирихле, обобщенный алгоритм Гаусса – Зейделя для решения задачи Дирихле, алгоритм Гаусса – Зейделя для решения задачи Дирихле уравнения Пуассона, задача теплопроводности. Проводится сравнение достигнутого ускорения с ускорением, полученным при использовании системы PLUTO. Приводится влияние дополнительных преобразований на ускорение, полученное с помощью реализованного алгоритма. Описывается теоретическая модель вычисления оптимальных размеров тайлов, подтверждённая численными экспериментами.

В четвёртой главе рассматривается метод диалогового анализа текстов программ, для уточнения информационных зависимостей при оптимизациях программ, таких так прямоугольный тайлинг, гнездование цикла, и при распараллеливании для архитектур SIMD, MIMD. Для анализа программ используется символьный анализ и линеаризация выражений. Высокоуровневое внутреннее представление ОРС позволяет анализировать код и формировать вопросы в доступном программисту виде.

В заключении подводятся итоги исследований, представленных в диссертации. Описываются основные полученные результаты, решенные задачи, обсуждается их новизна и практическая значимость. Обосновывается достижение цели диссертации.

Глава 1. Оптимизирующие преобразования программ для ускорения целевых алгоритмов

Теория оптимизирующих преобразований программ используется в компиляторах для повышения производительности программ. Данная теория описана во многих работах: [29], [30], [23], [109], [56], [110] и др.

1.1. Элементы теории преобразования программ

Эквивалентная программа – две программы считаются эквивалентными, если при одних и тех же входных данных выходные данные у этих программ тоже совпадают.

Преобразования программ – это изменения программного кода. Преобразование называется **эквивалентным**, если исходная и преобразованная программы эквивалентны.

Оптимизирующие преобразования нацелены на улучшение характеристик программы (время выполнения программы, количество операций, объем используемой памяти и т.д.).

Одни из самых первых преобразований – «протягивание констант» и «удаление мертвого кода» – оптимизируют линейные участки программы, уменьшают количество вычислений и обращений к памяти соответственно. Данные оптимизации выполняются подавляющим большинством современных компиляторов.

Наиболее трудоёмкими частями программы являются циклы. Поэтому их оптимизация зачастую приводит к повышению производительности.

Гнездо циклов (Loop nest) – это фрагмент программы, содержащий последовательность вложенных циклов.

Тесное гнездо циклов (Perfect loop nest) – это гнездо циклов, состоящее из двух и более циклов, в котором все операторы находятся внутри внутреннего цикла.

Листинг 1.1.1. Тесное гнездо циклов.

```
for (int i0 = a0; i0 < b0; i0++) {
     for (int i1 = a1; i1 < b1; i1++) {
     …
          for (int in = an; in < bn; in++) {
               LoopBody(i0,i1,...,in)
          }
     …
     }
}
```

Приведём определения из работы [110].

Вхождением переменной будем называть всякое появление переменной в тексте программы вместе с тем местом в программе, в котором эта переменная появилась. Всякому

вхождению (при конкретном значении индексного выражения для массивов) соответствует обращение к некоторой ячейке памяти.

Генератор (out, output) – вхождение, при котором происходит запись в ячейку памяти.

Использованиями (in, input) называются остальные вхождения переменных.

Граф информационных зависимостей – это ориентированный граф, вершины которого соответствуют вхождениям переменных, а дуга u-v соединяет пару вершин, если эти вхождения порождают информационную зависимость (обращаются к одной и той же ячейке памяти), причем вхождение u раньше обращается к памяти, чем v, и хотя бы одно из этих вхождений является генератором. Граф информационных зависимостей используется для анализа информационных зависимостей, что позволяет контролировать эквивалентность преобразований.

Типы дуг графа информационных зависимостей:

1) out-in – истинная информационная зависимость (true dependence),
2) in-out – антизависимость (antidependence),
3) out-out – выходная зависимость (output dependence),
4) in-in – входная зависимость (input dependence).

Дуга (u, v) информационной зависимости называется ложной, если существует такая другая дуга (u2, v), при которой к общей ячейке памяти вхождение u2 всегда обращается позже u.

Входные зависимости не влияют на эквивалентность преобразований программ и используются только при распределении данных в параллельной памяти.

Информационная зависимость между вхождениями называется **циклически независимой (loop independent dependence)**, если эти вхождения обращаются к одной и той же ячейке памяти на одной и той же итерации цикла. Иначе зависимость называется **циклически порожденной (loop carried dependence)**, а цикл называется порождающим такую зависимость.

В работе [111] приводится обзор известных методов определения информационных зависимостей по данным в программах.

Носителем информационной зависимости называют цикл, создающий эту зависимость, которая является циклически порожденной. Носители нумеруются от внешнего цикла к внутреннему, начиная с нуля. Для одной циклически порождённой зависимости может быть несколько циклов, которые её создают. Обозначим множество носителей – carriers.

Пример 1.1.1.

Пусть дано двойное гнездо циклов.

```
for(i=1; i<=3; i=i+1)
     for (j=1; j<=3; j=j+1)
        a[j] = a[j+1];
```

Построим его граф информационных зависимостей. Внешний цикл является носителем четырёх дуг информационных зависимостей, внутренний – только для дуги антизависимости (рисунок 1.1).

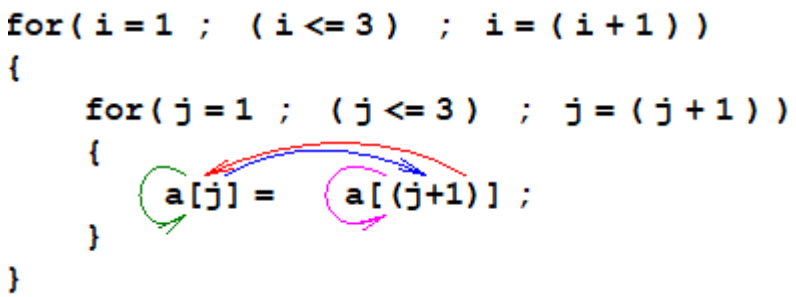

Рисунок 1.1. Граф информационных зависимостей, построенный ОРС для примера 1.1.1. Красная дуга – дуга антизависимости, синяя – истинной зависимости, зелёная – выходная зависимость, розовая – входная зависимость

Раскрутка циклов – это преобразование, которое убирает заголовок цикла и повторяет его тело для каждого значения счётчика цикла.

Преобразование, известное как «**подстановка вперёд**», заменяет все последующие вхождения левой части оператора присваивания её правой частью при условии, что это не нарушит равносильность. Если правая часть оператора присваивания является константой, то такое преобразование называется «**протягиванием констант**». Результатом протягивания констант является уменьшение кода программы, улучшение скорости выполнения, также замена переменных их значениями в индексных выражениях обеспечивает точное построение графа информационных зависимостей.

Инвариант цикла – выражение, значение которого не меняется при каждом прохождении цикла. Предварительное вычисление инвариантов до выполнения цикла может увеличить производительность.

Вынос инвариантных выражений из циклов – это преобразование, которое находит инварианты в цикле и выносит их за пределы цикла.

Расщепление цикла (Loop Splitting) – это оптимизирующее преобразование, которое разбивает цикл на несколько частей, имеющих одно и то же тело исходного цикла и различные диапазоны счётчика.

Разбиение цикла (Loop Fission / Loop Distribution) – это оптимизирующее преобразование, разбивающее цикл на несколько циклов, которые имеют заголовок исходного цикла и различные фрагменты исходного тела цикла.

1.2. Анализ информационных зависимостей в гнёздах циклов с помощью решетчатых графов

Рассмотрим гнездо из n+1 вложенных друг в друга циклов. Пронумеруем операторы циклов в этом гнезде в соответствии с порядком вложенности, начиная с самого внешнего цикла. Обозначим счетчик j-го цикла – I_j, где j от 0 до n.

Множество значений, которые может принимать вектор счетчиков циклов $I = (I_0, I_1, \ldots, I_n)$ в указанном гнезде, называется **пространством итераций** данного гнезда.

Если сделать раскрутку всех циклов гнезда, то получим код без циклов, состоящий из множества блоков, каждый из которых является копией тела исходного гнезда циклов. Каждая такая копия тела гнезда циклов соответствует набору значений счетчиков циклов гнезда.

Будем обозначать копию вхождения v в раскрутке исходного гнезда циклов при значениях счетчиков $I' = (I'_0, I'_1, \ldots, I'_n)$ следующим образом: $v(I'_0, I'_1, \ldots, I'_n)$ или $v(I')$ и называть **представителем** данного вхождения.

Для анализа и преобразований многомерных циклов используются решетчатые графы. В исследованиях В.В. Воеводина [112], [113], [114] и P. Feautrier [115], [116] описываются компактные способы хранения решетчатого графа в виде функций, а также описываются алгоритмы построения этих функций. Подробное описание теории решетчатых графов приводится в работе [117].

Приведём определение элементарного решетчатого графа из статьи [118].

Листинг 1.2.1. Цикл с вхождениями массива X: X[F(I)], X[G(I)].

```
for (int I = a; I < b; I++) {
        X(F(I)) = . . .
              ...
        . . . = X(G(I))
}
```

Пусть X – это m-мерный массив, тогда будем рассматривать вхождение X[F(I)], где F(I) – отображение пространства итераций в n-мерное целочисленное пространство индексов массива

X, а I – точка пространства итераций. Далее отображение F будем рассматривать аффинным. Например, для вхождения X[i_1-i_2, i_3+2] при размерности пространства итераций n=3 и размерности массива X m = 2, отображение будет иметь вид: F(i_1, i_2, i_3) = ($i_1 - i_2$, $i_3 + 2$).

Пусть в гнезде циклов имеется пара зависимых вхождений X[F(I)] и X[G(J)] и этой паре соответствует некоторая дуга графа информационных зависимостей (X[F(I)], X[G(J)]) (листинг 1.2.1). Для этой дуги определим элементарный решетчатый граф (F, G). Вершины решетчатого графа – точки пространства итераций гнезда циклов. Пусть (I, J) – пара точек пространства итераций и I<J (лексикографический порядок, то есть I раньше J). Дуга (I, J) принадлежит графу, если F(I)=G(J) и для любой точки пространства итераций K, для которой K<J и F(K)=G(J), выполняется K <= I. Иными словами, вершина I является лексикографическим максимумом множества всех таких точек K, для которых F(K)=G(J) и K<J. Если для любой переменной в теле гнезда циклов есть не более одного генератора, то существует взаимно однозначное соответствие между дугами графа (F, G) и не ложными дугами графа информационных зависимостей раскрутки всех циклов гнезда [118].

В каждую точку пространства итераций входит не более одной дуги элементарного решетчатого графа, соответствующей дуге (X[F(I)], X[G(J)]) графа информационных зависимостей.

Далее будем рассматривать решетчатый граф программы как объединение элементарных решетчатых графов.

Для гнезда циклов из примера 1.1.1 изображены граф информационных зависимостей (рисунок 1.1) и решетчатый граф (рисунок 1.14).

Если в гнезде n циклов, то пространство итераций, т.е. множество наборов значений векторов счетчиков циклов после раскрутки образует подмножество целочисленной решетки n-мерного пространства. Поэтому графы информационных зависимостей между копиями тела гнезда циклов естественно называть решетчатыми [113], [114], [118].

Обозначим u(i_1,i_2,…,i_d) представитель вхождения u в теле гнезда циклов с координатами (i_1,i_2,…,i_d). Также u(i_1,i_2,…,i_d) будем обозначать семейство вершин решетчатого графа, соответствующих вхождению u.

Пример 1.2.2

Для гнезда циклов из примера 1.1.1 вхождению a[j+1] соответствуют 9 представителей в раскрутке обоих циклов этого гнезда, каждой из которых соответствует точка пространства итераций решетчатого графа (рисунок 1.14). Этому вхождению соответствует семейство вхождений a(i,j). При этом, например, a(2,3) = a[4], a(3,1) = a[2].

Листинг 1.2.2. Результат полной раскрутки гнезда циклов (пример 1.1.1)

```
// i = 1
    a[1]=a[2]; // j = 1
    a[2]=a[3]; // j = 2
    a[3]=a[4]; // j = 3
// i = 2
    a[1]=a[2]; // j = 1
    a[2]=a[3]; // j = 2
    a[3]=a[4]; // j = 3
// i = 3
    a[1]=a[2]; // j = 1
    a[2]=a[3]; // j = 2
    a[3]=a[4]; // j = 3
```

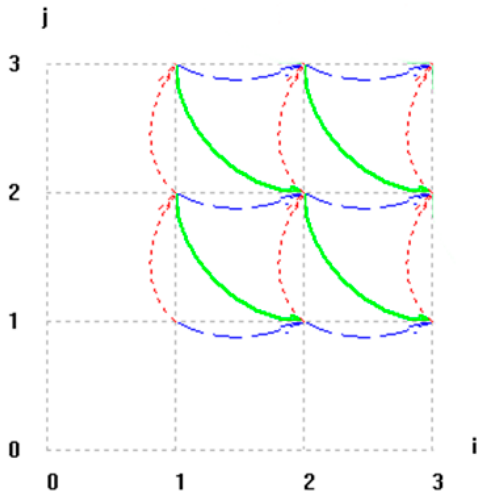

Рисунок 1.2. Решетчатый граф, построенный ОРС для примера 1.1.1. Красная дуга – дуга антизависимости, зелёная – истинной зависимости, синяя – выходной зависимости

Понятие **вектора расстояния (distance vector)** [56] [119] относится к дуге графа информационных зависимостей. Вектор расстояния зависимости используется для определения параметров скашивания гнезда циклов для последующего корректного выполнения прямоугольного тайлинга.

Приведём определение вектора расстояния дуги информационной зависимости.

Пусть в графе информационных зависимостей гнезда из n циклов есть дуга (u, v). Тогда существует такая пара представителей $u(i'_1,i'_2,\ldots,i'_n)$, $v(i''_1,i''_2,\ldots,i''_n)$, для которых есть дуга в решетчатом графе. Заметим, что таких пар представителей может быть много. Рассмотрим разность векторов счетчиков циклов $(i''_1-i'_1,i''_2-i'_2,\ldots,i''_n-i'_n)$. Если значение этой разности одинаково для всех таких дуг $(u(i'_1,i'_2,\ldots,i'_n), v(i''_1,i''_2,\ldots,i''_n))$, соответствующих (u, v), то будем называть эту разность **вектором расстояния (distance vector)** дуги информационной зависимости (u, v).

Замечание. Не для любого представителя $v(i''_1,i''_2,\ldots,i''_n)$ вхождения v существует представитель $u(i'_1,i'_2,\ldots,i'_n)$, из которого есть дуга решетчатого графа в $v(i''_1,i''_2,\ldots,i''_n)$. Но хотя бы для одного такого представителя $v(i''_1,i''_2,\ldots,i''_n)$ входящая дуга решетчатого графа имеется [118].

1.3. Анализ векторов расстояний информационных зависимостей на основе решетчатых графов

В этом параграфе приводятся примеры решетчатых графов для двойных циклов с похожими операторами присваивания в теле гнезда. Эти примеры иллюстрируют информационные зависимости, которые могут влиять на корректность применения преобразований. Такие зависимости учитываются при применении преобразований в данной работе.

Пример 1.3.1.

Между вхождениями a[i] (левое) и a[i] (правое) присутствуют дуга истинной зависимости и дуга антизависимости (рисунок 1.3). Для этих информационных зависимостей векторы расстояний равны (0,1) (рисунок 1.4).

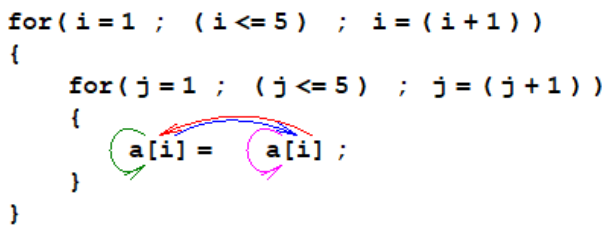

Рисунок 1.3. Граф информационных зависимостей, построенный ОРС для примера 1.3.1. Красная дуга – дуга антизависимости, синяя – истинной зависимости, зелёная – выходная зависимость, розовая – входная зависимость

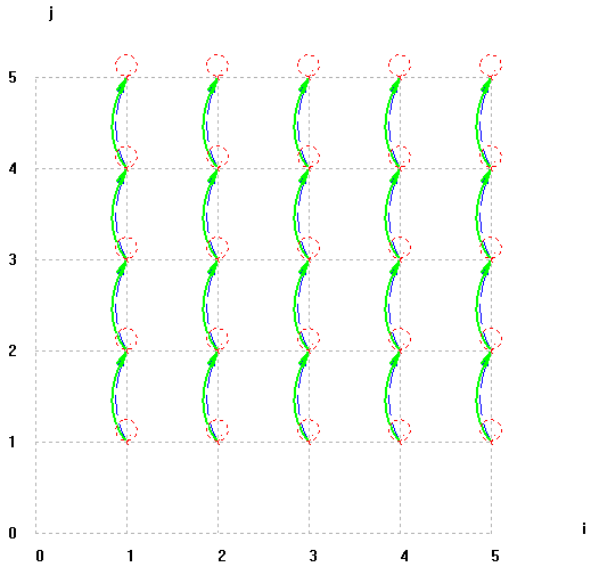

Рисунок 1.4. Решетчатый граф, построенный ОРС для примера 1.3.1

Пример 1.3.2.

На графе информационных зависимостей присутствуют дуги истинной зависимости и выходной зависимости (

рисунок 1.5). Для дуги истинной зависимости вектор расстояний не существует. Для дуги выходной зависимости вектор расстояния равен (0,1) (рисунок 1.6).

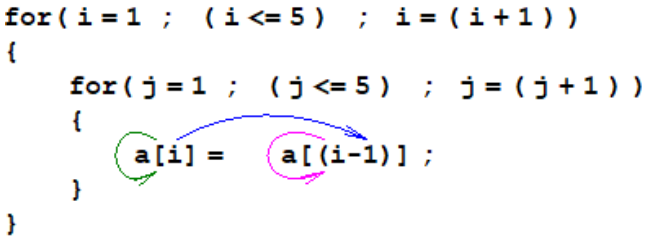

Рисунок 1.5. Граф информационных зависимостей, построенный ОРС для примера 1.3.2. Синяя дуга – дуга истинной зависимости, зелёная – выходная зависимость, розовая – входная зависимость

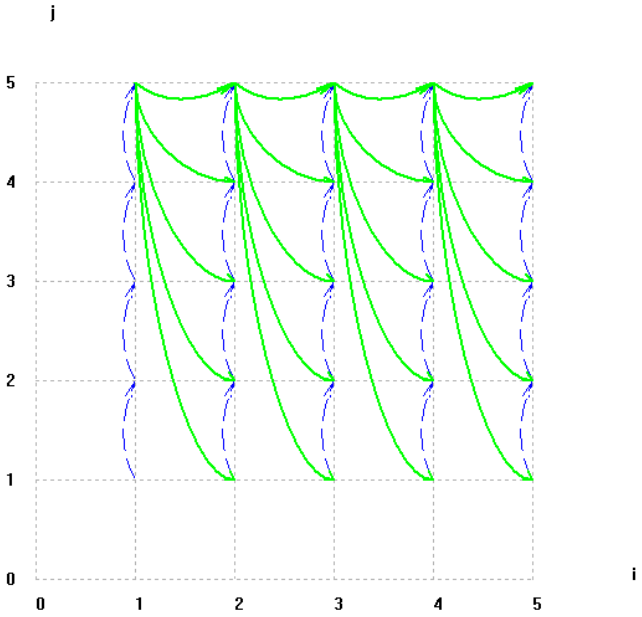

Рисунок 1.6. Решетчатый граф, построенный ОРС для примера 1.3.2

Пример 1.3.3.

На графе информационных зависимостей присутствуют дуги антизависимости и выходной зависимости (рисунок 1.7). Для дуги антизависимости вектор расстояний не существует. Для дуги выходной зависимости вектор расстояния равен (0,1) (рисунок 1.8).

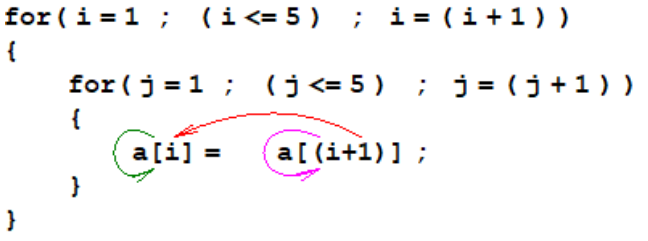

Рисунок 1.7. Граф информационных зависимостей, построенный ОРС для примера 1.3.3. Красная дуга – дуга антизависимости, зелёная – выходная зависимость, розовая – входная зависимость

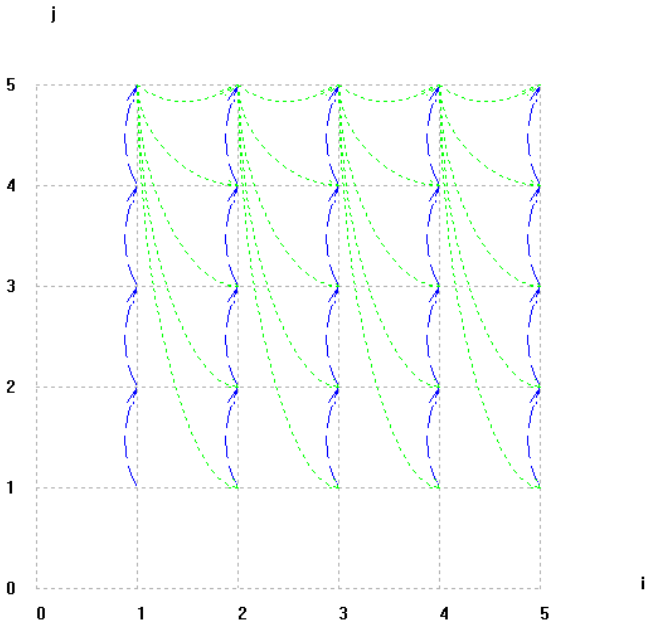

Рисунок 1.8. Решетчатый граф, построенный ОРС для примера 1.3.3

Пример 1.3.4.

Между вхождениями a[i] (левое) и a[i] (правое) присутствуют дуга истинной зависимости и дуга антизависимости (рисунок 1.9). Для этих информационных зависимостей векторы расстояний равны (1,0) (рисунок 1.10).

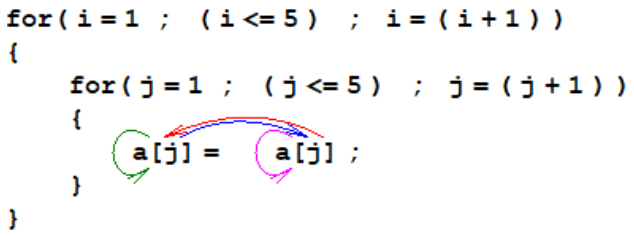

Рисунок 1.9. Граф информационных зависимостей, построенный ОРС для примера 1.3.4. Красная дуга – дуга антизависимости, синяя – истинной зависимости, зелёная – выходная зависимость, розовая – входная зависимость

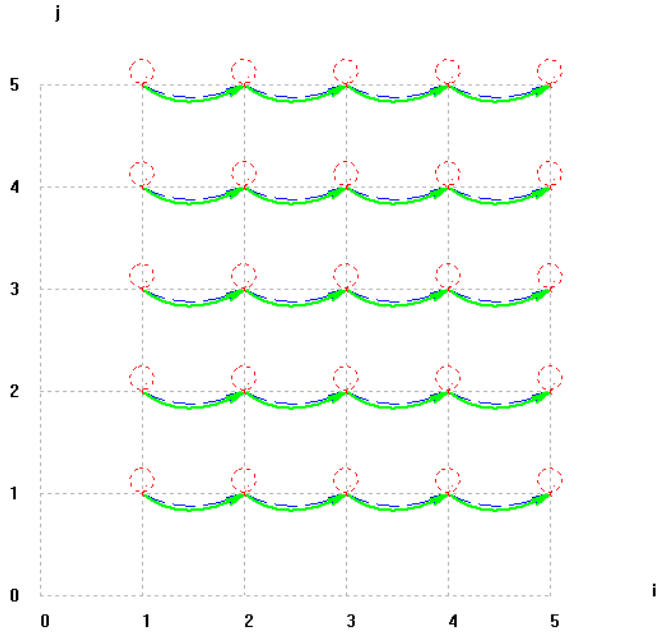

Рисунок 1.10. Решетчатый граф, построенный ОРС для примера 1.3.4

Пример 1.3.5.

На графе информационных зависимостей присутствуют дуги выходной, истинной и антизависимости (рисунок 1.11). Для дуги выходной зависимости вектор расстояния равен (0,1), истинной – (1,0), для антизависимости – (1,-1) (рисунок 1.12).

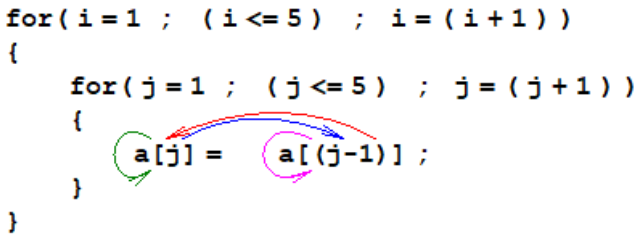

Рисунок 1.11. Граф информационных зависимостей, построенный ОРС для примера 1.3.5. Красная дуга – дуга антизависимости, синяя – истинной зависимости, зелёная – выходная зависимость, розовая – входная зависимость

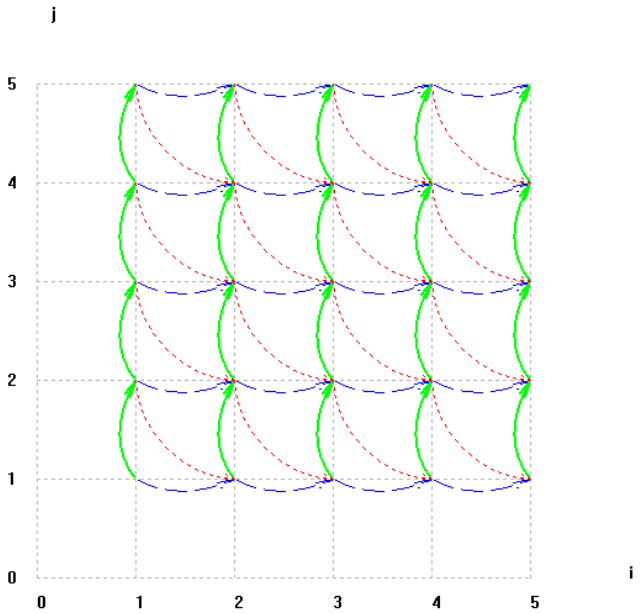

Рисунок 1.12. Решетчатый граф, построенный ОРС для примера 1.3.5

Пример 1.3.6.

На графе информационных зависимостей присутствуют дуги выходной, истинной и антизависимости (рисунок 1.13). Для дуги выходной зависимости вектор расстояния равен (1,0) истинной – (1,-1), для антизависимости – (0,1) (рисунок 1.14).

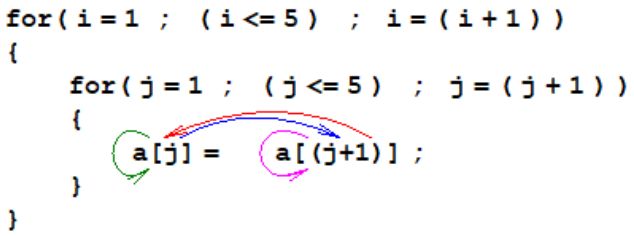

Рисунок 1.13. Граф информационных зависимостей, построенный ОРС для примера 1.3.6. Красная дуга – дуга антизависимости, синяя – истинной зависимости, зелёная – выходная зависимость, розовая – входная зависимость

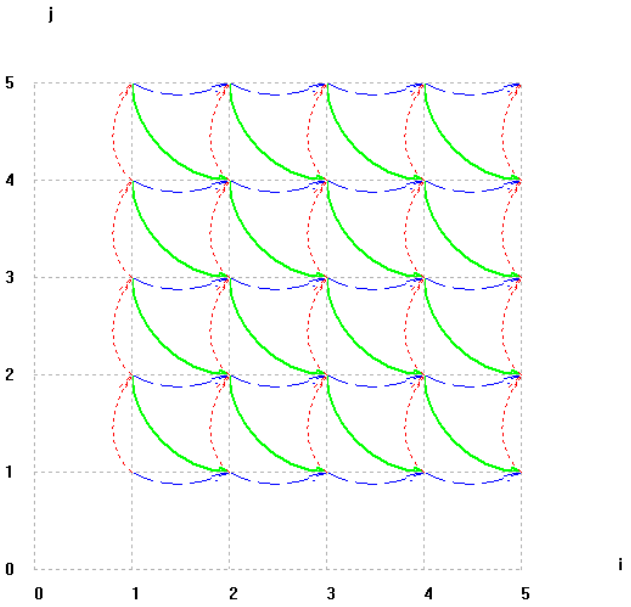

Рисунок 1.14. Решетчатый граф, построенный ОРС для примера 1.3.6

Пример 1.3.7.

На графе информационных зависимостей присутствует дуга антизависимости (рисунок 1.15). Её вектор расстояния равен (0,0) (рисунок 1.16).

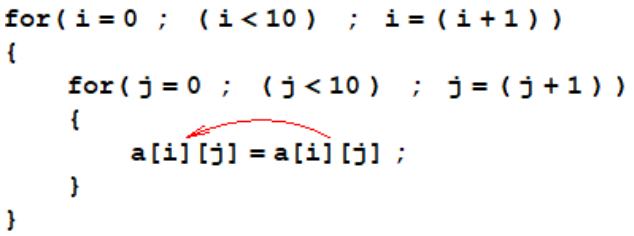

Рисунок 1.15. Граф информационных зависимостей, построенный ОРС для примера 1.3.7. Красная дуга – дуга антизависимости

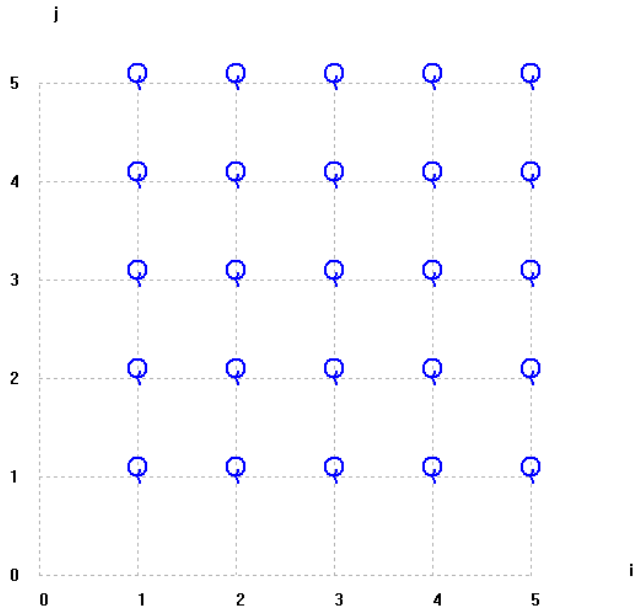

Рисунок 1.16. Решетчатый граф, построенный ОРС для примера 1.3.7

Пример 1.3.8.

На графе информационных зависимостей присутствует дуга истинной зависимости (рисунок 1.17). Её вектор расстояния равен (0,1) (рисунок 1.18).

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

Рисунок 1.17. Граф информационных зависимостей, построенный ОРС для примера 1.3.8. Синяя дуга – дуга истинной зависимости

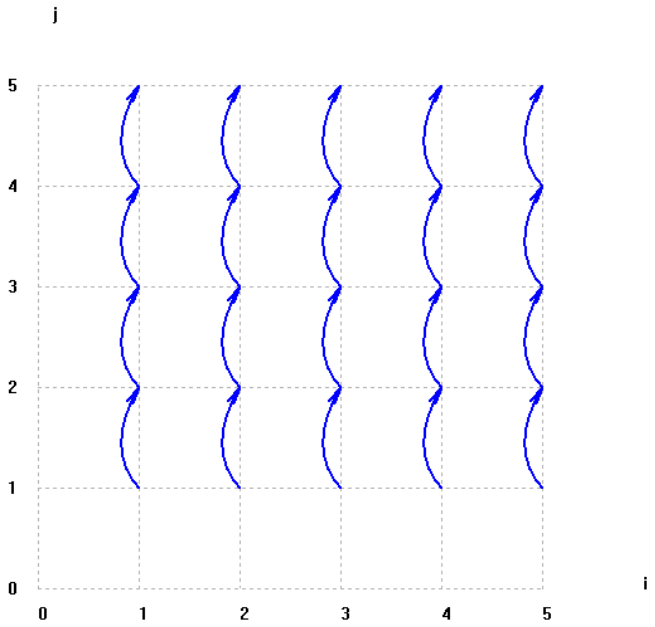

Рисунок 1.18. Решетчатый граф, построенный ОРС для примера 1.3.8

Пример 1.3.9.

На графе информационных зависимостей присутствует дуга антизависимости (рисунок 1.19). Её вектор расстояния равен (0,1) (рисунок 1.20).

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

Рисунок 1.19. Граф информационных зависимостей, построенный ОРС для примера 1.3.9. Красная дуга – дуга антизависимости

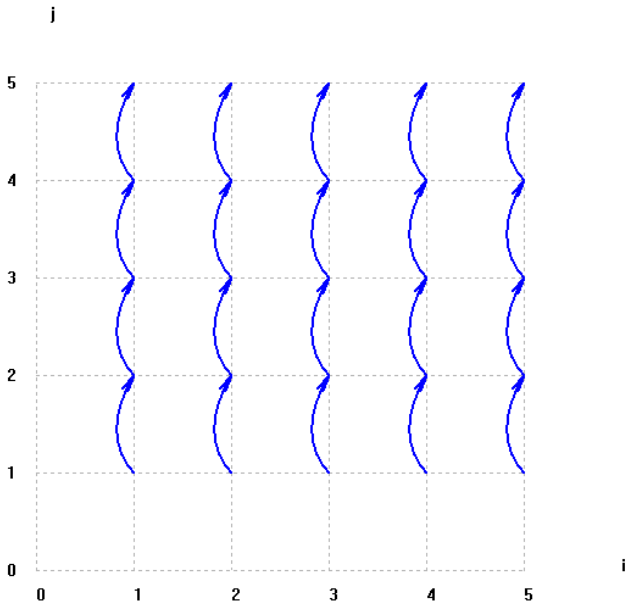

Рисунок 1.20. Решетчатый граф, построенный ОРС для примера 1.3.9

Пример 1.3.10.

На графе информационных зависимостей присутствует дуга истинной зависимости (рисунок 1.21). Её вектор расстояния равен (1,0) (рисунок 1.22).

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

Рисунок 1.21. Граф информационных зависимостей, построенный ОРС для примера 1.3.10. Синяя дуга – дуга истинной зависимости

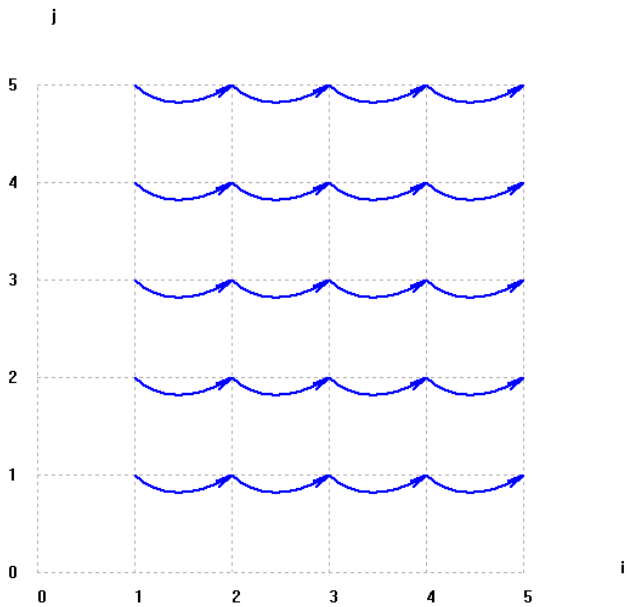

Рисунок 1.22. Решетчатый граф, построенный ОРС для примера 1.3.10

Пример 1.3.11.

На графе информационных зависимостей присутствует дуга истинной зависимости (рисунок 1.23). Её вектор расстояния равен (1,1) (рисунок 1.24).

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

Рисунок 1.23. Граф информационных зависимостей, построенный ОРС для примера 1.3.11. Синяя дуга – дуга истинной зависимости

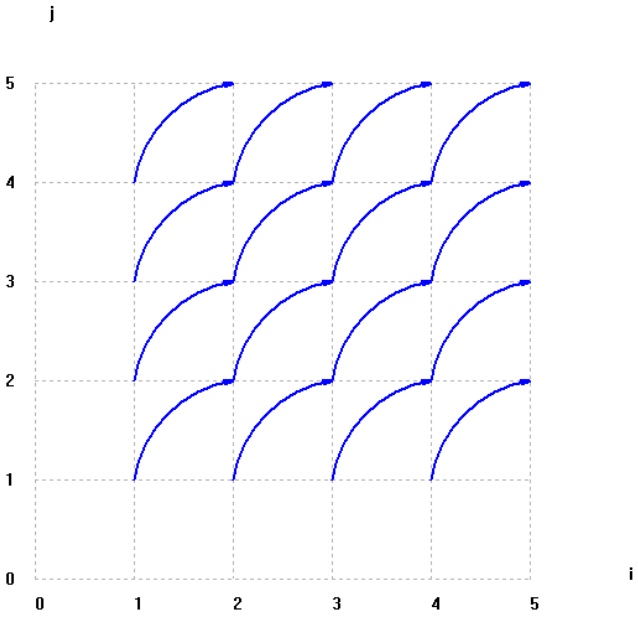

Рисунок 1.24. Решетчатый граф, построенный ОРС для примера 1.3.11

Пример 1.3.12.

На графе информационных зависимостей присутствует дуга истинной зависимости (рисунок 1.25). Её вектор расстояния равен (1,-1) (рисунок 1.26).

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

Рисунок 1.25. Граф информационных зависимостей, построенный ОРС для примера 1.3.12. Синяя дуга – дуга истинной зависимости

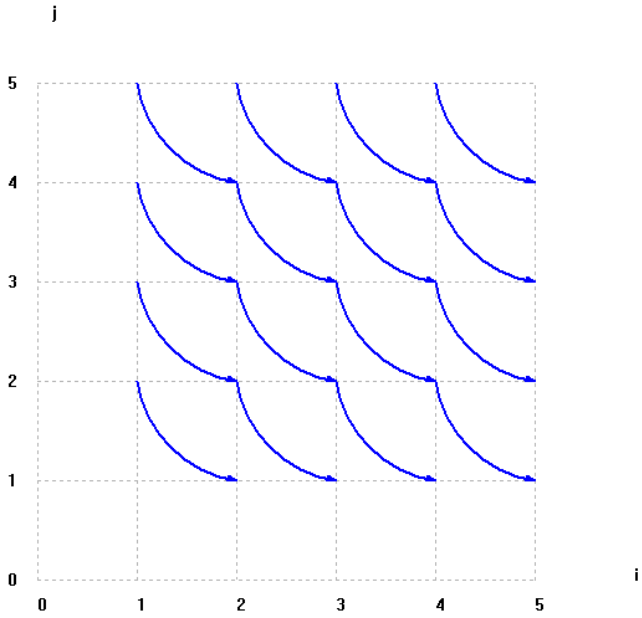

Рисунок 1.26. Решетчатый граф, построенный ОРС для примера 1.3.12

Пример 1.3.13.

На графе информационных зависимостей присутствует дуга антизависимости (рисунок 1.27). Её вектор расстояния равен (1,0) (рисунок 1.28).

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

Рисунок 1.27. Граф информационных зависимостей, построенный ОРС для примера 1.3.13. Красная дуга – дуга антизависимости

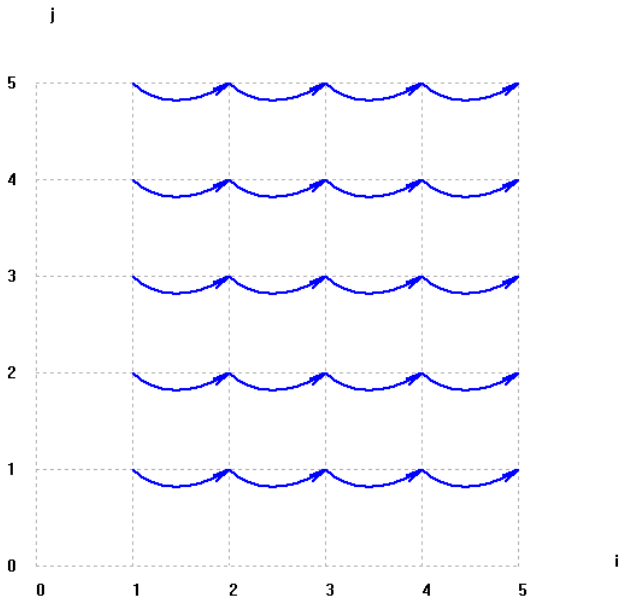

Рисунок 1.28. Решетчатый граф, построенный ОРС для примера 1.3.13

Пример 1.3.14.

На графе информационных зависимостей присутствует дуга антизависимости (рисунок 1.29). Её вектор расстояния равен (1,-1) (рисунок 1.30).

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

Рисунок 1.29. Граф информационных зависимостей, построенный ОРС для примера 1.3.14. Красная дуга – дуга антизависимости

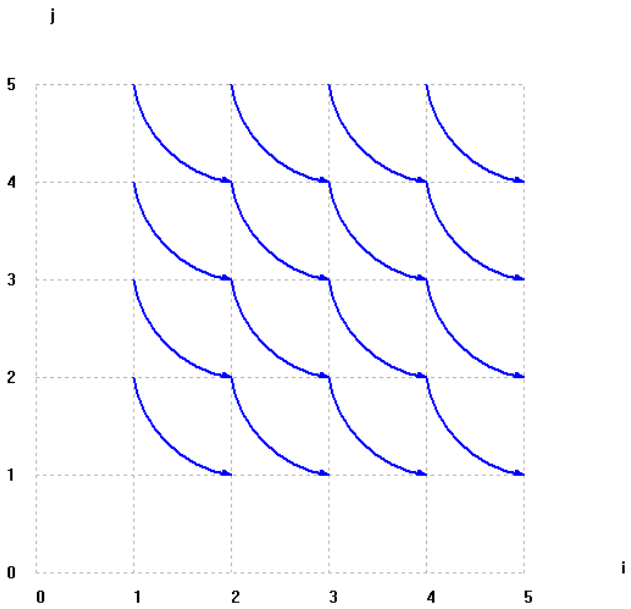

Рисунок 1.30. Решетчатый граф, построенный ОРС для примера 1.3.14

Пример 1.3.15.

На графе информационных зависимостей присутствует дуга антизависимости (рисунок 1.31). Её вектор расстояния равен (1,1) (рисунок 1.32).

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

Рисунок 1.31. Граф информационных зависимостей, построенный ОРС для примера 1.3.15. Красная дуга – дуга антизависимости

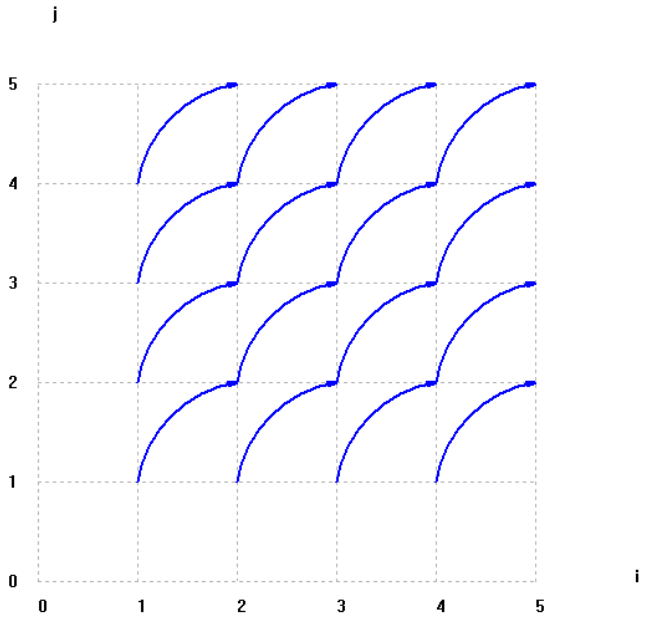

Рисунок 1.32. Решетчатый граф, построенный ОРС для примера 1.3.15

1.4. Анализ гнёзд циклов итерационного типа с использованием решетчатых графов

Гнездо циклов итерационного типа – это такое гнездо циклов [92], которое обладает следующими свойствами. Внешний цикл отвечает за обход итераций, а внутренние – за расчёт шаблона вычислений, который повторяется на итерациях внешнего гнезда. В данной работе рассматриваются такие гнёзда циклов итерационного типа, которые являются тесными, количество итераций которых не меняется на момент выполнения и в тело которых входят только операторы присваивания, безындексные переменные, константы, массивы в стиле языка С; при этом массивы имеют линейные индексные выражения вида (i + <целочисленная константа>), где i – счетчик одного из циклов. Более точно: в теле итерационного гнезда циклов (LoopBody (i_1, i_2, ... , i_n)) могут присутствовать генераторы вида u[i_1-c_1][i_2-c_2][…][i_n-c_n], (где c_1, c_2, … , c_n – целочисленные константы), в которых последовательность индексных выражений массива (i_1-c_1, i_2-c_2, … , i_n-c_n) соответствует последовательности счётчиков гнезда циклов (i_1,i_2,…,i_n). Тогда использования этой же переменной u в правой части оператора присваивания должны иметь вид: u[i_1-d_1][i_2-d_2][…][i_n-d_n], где d_1, d_2, … , d_n – целочисленные константы. Общий вид такого гнезда циклов представлен в листинге ниже. Из-за специфики информационных зависимостей ни один цикл данного гнезда не может выполняться параллельно, поэтому перед распараллеливанием надо выполнить его преобразование.

Листинг 1.4.1. Гнездо циклов итерационного типа, где LoopBody – тело гнезда циклов, зависящее от счётчиков циклов.

```
for (int i0 = 0; i0 < itera; i0++) {
     for (int i1 = a1; i1 < b1; i1++) {
          for (int i2 = a2; i2 < b2; i2++) {
               …
                    for (int in = an; in < bn; in++) {
                         LoopBody(i1,i2,...,in)
                    }
               …
          }
     }
}
```

Вычисление векторов расстояний информационных зависимостей гнезда циклов описано ранее в параграфе 1.2. В частности, если в теле гнезда циклов присутствуют вхождения массива, размерность которого равна размерности гнезда циклов, то вектор расстояний для информационных зависимостей между такими вхождениями вычисляется как разность индексных выражений вхождений этого массива: dist(a[i_0-c_0][i_1-c_1][…][i_n-c_n], a[i_0-d_0][i_1-d_1][…][i_n-d_n]) = (d_0-c_0, d_1-c_1,…,d_n-c_n), где c0, c1, … , cn; d0, d1, … , dn – целочисленные константы (см. примеры 1.3.7–1.3.15).

В алгоритме, представленном в главе 2, вектор расстояния используется только для определения матрицы скашивания (параграф 2.1).

Сформулируем понятие вектора расстояния для информационных зависимостей в гнезде циклов итерационного типа. В случае гнезда итерационного типа вхождения массивов имеют размерность на единицу меньше, чем размерность гнезда циклов (см. примеры 1.3.1–1.3.6).

Пусть размерность гнезда циклов итерационного типа равна n+1, u и v – вхождения n-мерного массива a, образующие дугу информационной зависимости (u, v) = (a[i_1-c_1][i_2-c_2][…][i_n-c_n], a[i_1-d_1][i_2-d_2][…][i_n-d_n]). Счётчик внешнего цикла, имеющего порядковый номер 0 и вычисляющего итерации алгоритма, не входит в индексные выражения массивов (согласно определению гнезда циклов итерационного типа). Тогда существует такая пара представителей, для которых есть дуга в решетчатом графе (u (i'_0, i'_1, i'_2, … , i'_n), v (i''_0, i''_1, i''_2, … , i''_n)).

Рассмотрим вектор длины n+1 разностей векторов счётчиков циклов (i''_0-i'_0, i''_1-i'_1, i''_2-i'_2, … ,i''_n-i'_n). Все координаты, начиная с первой, не зависят от выбора пары представителей u, v, а нулевая координата t всегда неотрицательна, поскольку дуга информационной зависимости направлена от u к v. Тогда определим для гнезда циклов итерационного типа вектор расстояний

дуги информационной зависимости (u, v): dist(a[i_1-c_1][i_2-c_2][…][i_n-c_n], a[i_1-d_1][i_2-d_2][…][i_n-d_n]) = (t, d_1-c_1, d_2-c_2, … , d_n-c_n), t >= 0.

Будем рассматривать далее вектор расстояния зависимости для гнёзд циклов итерационного типа, полагая, что его нулевая координата равна 1.

Заметим, что в алгоритме решения уравнения теплопроводности (листинг 3.5.1) в индексных выражениях массива встречается операция % (остаток от деления), что не соответствует определению гнезда циклов итерационного типа. Тем не менее в этом случае также может обобщаться известное понятие «вектор расстояния».

1.5. Описание некоторых используемых в работе преобразований программ

В литературе описаны [56], [120], [121] и в данной работе используются некоторые преобразования гнёзд циклов, которые описываются матрицами. Приведём необходимые понятия.

Унимодулярная матрица – квадратная матрица с целыми коэффициентами, определитель которой равен ±1.

Унимодулярные преобразования – преобразование циклов, основанное на унимодулярной матрице, изменяющее либо пространство итераций, либо порядок выполнения его итераций.

Применение цепочки матричных преобразований можно свести к перемножению матриц. Влияние преобразований на информационные зависимости можно представить в виде умножения матрицы преобразования на векторы расстояний информационных зависимостей.

1.5.1. Перестановка циклов (Loop Interchange)

Перестановка циклов (Loop Interchange) – это преобразование, осуществляющее перестановку местами операторов цикла (заголовков циклов) в гнезде циклов.

Перестановка циклов допустима, если отсутствуют дуги информационных зависимостей, имеющих отрицательные координаты вектора расстояний [122].

Унимодулярная матрица преобразования «перестановка циклов» имеет вид $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

Листинг 1.5.1. Исходное гнездо циклов:

```
for (int i = ai; i < bi; i++)
      for (int j = aj; i < bj; j++)
            LoopBody(i,j)
```

Листинг 1.5.2. Гнездо циклов после применения преобразования «перестановка циклов»:

```
for (int j = aj; i < bj; j++)
      for (int i = ai; i < bi; i++)
            LoopBody(i,j)
```

1.5.2. Гнездование цикла (Loop Nesting)

Гнездование цикла (Loop Nesting) – преобразование программ, которое разбивает одиночный цикл (листинг 1.5.3) на два вложенных цикла (листинг 1.5.4).

Листинг 1.5.3. Исходный цикл:

```
for (int i=0; i < N; i++){
        LoopBody(i)
}
```

Листинг 1.5.4. Гнездо циклов после применения преобразования «гнездование циклов»:

```
for (int i1=0; i1 < N/d; i1++){ // Обход блоков
    for (int i=i1*d; i<(i1+1)*d; i++){ // Обход внутри блока
        LoopBody(i)
    }
}
// Цикл, вычисляющий итерации, оставшиеся после разбиения
for (int i=d*(N/d); i < N; i++){
  LoopBody(i)
}
```

1.5.3. Скашивание циклов (Loop Skewing)

Скашивание циклов (Loop Skewing) [123] – преобразование, которое изменяет пространство итераций гнезда циклов: вектор счётчиков нового гнезда циклов I' равен вектору счетчиков исходного гнезда I, умноженного на нижнетреугольную матрицу с единицами на главной диагонали skew: I' = skew x I. Число f, которое находится в строке m и в столбце k (k<m), будем называть **параметром скашивания** цикла номер k относительно цикла под номером m.

Матрица skew преобразования «скашивание циклов» для двумерного гнезда циклов имеет вид:

$$\text{skew} = \begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix}.$$

Проиллюстрируем преобразование «скашивание циклов» на примере гнезда циклов ниже (листинг 1.5.5).

Листинг 1.5.5. Двумерное гнездо циклов в общем виде. LoopBody – тело цикла, зависящее от счётчиков гнезда циклов:

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

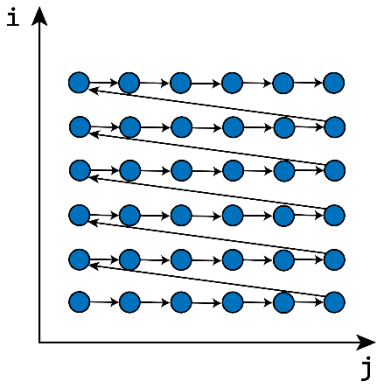

Рисунок 1.33. Пространство итераций двумерного гнезда циклов (листинг 1.5.5). Точки – точки пространства итераций, дуги – порядок выполнения этих точек

Листинг 1.5.6. Гнездо циклов после применения преобразования «скашивание циклов», f – параметр скашивания:

```
for (int i = initN; i < N; i++) {
      for (int j = initM + f * i; j < M + f * i; j++) {
            LoopBody(i, j - f * i);
      }
}
```

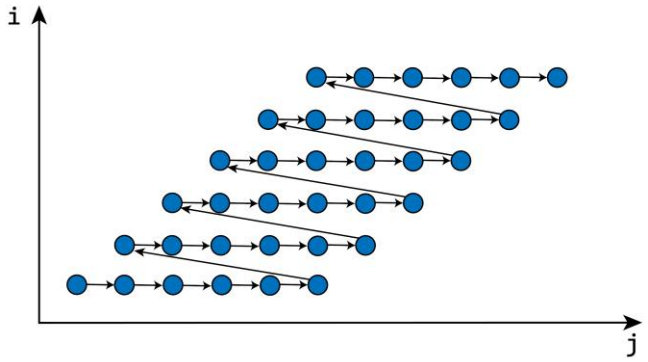

Рисунок 1.34. Пространство итераций двумерного цикла (листинг 1.5.6), изменённое с помощью преобразования Loop Skewing при f=1. Точки – точки пространства итераций, дуги – порядок выполнения этих точек; i, j – счётчики гнезда циклов

Композиция скашиваний является коммутирующей, если параметры скашивания находятся в одной строке или столбце:

$$\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ a & 0 & 1\end{pmatrix}\text{x}\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & b & 1\end{pmatrix}=\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & b & 1\end{pmatrix}\text{x}\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ a & 0 & 1\end{pmatrix}=\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ a & b & 1\end{pmatrix}.$$

Преобразование «скашивание циклов» является эквивалентным, поскольку не меняет порядок выполнения итераций.

Пусть в графе информационных зависимостей гнезда циклов присутствует дуга (u, v). Применим скашивание цикла под номером k относительно цикла под номером m (k < m) на параметр скашивания f. Тогда m-я координата вектора расстояний d_m=dist(u, v)$_m$ вычисляется по формуле d_m + = f * d_k. Остальные координаты не меняются.

Пример 1.5.1.

Проиллюстрируем влияние преобразования «скашивание циклов» на вектор расстояний информационных зависимостей в гнезде циклов на примере 1.1.1. Для дуги истинной информационной зависимости (a[j], a[j+1]) вектор расстояний = (1, -1). Матрица скашивания для примера 1.1.1 имеет вид: $\begin{pmatrix}1 & 0\\ 1 & 1\end{pmatrix}$. Рассчитаем вектор расстояний для дуги информационной зависимости (a[i+1,j-1-i], a[i,j-i]): $\begin{pmatrix}1 & 0\\ 1 & 1\end{pmatrix}\text{x}\begin{pmatrix}1\\ -1\end{pmatrix}=\begin{pmatrix}1\\ 0\end{pmatrix}$.

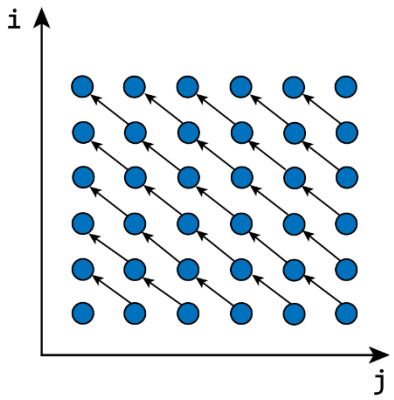

Рисунок 1.35. Решетчатый граф двумерного гнезда циклов для примера 1.1.1. Вершины графа – точки пространства итераций. Дуги – дуги антизависимости, с вектором расстояний = (1,-1)

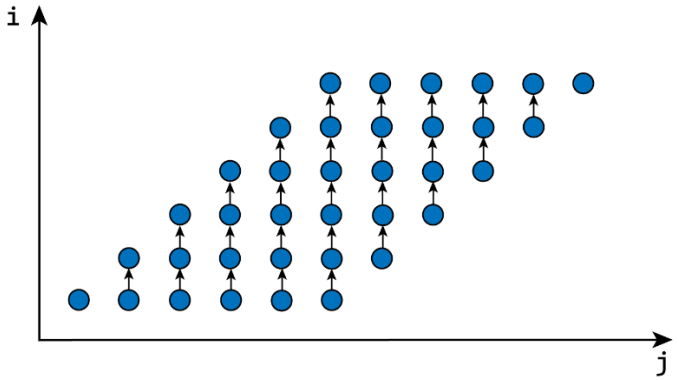

Рисунок 1.36. Изменённый решетчатый граф для двумерного гнезда циклов из примера 1.1.1 при помощи скашивания с параметром f=1. Вершины графа – точки пространства итераций. Дуги – дуги антизависимости, с вектором расстояний = (1, 0)

Дуги решетчатого графа стали параллельны оси координат i (рисунок 1.36).

Таким образом, преобразование «скашивание циклов» позволяет изменить вектор расстояний информационной зависимости. Это позволяет подготовить гнездо циклов для эквивалентного применения прямоугольного тайлинга, который описывается ниже.

1.5.4. Метод гиперплоскостей (Loop Wavefront)

Метод гиперплоскостей (Loop Wavefront) [19]**,** [56] – это преобразование гнезда циклов, которое меняет обход точек пространства итераций следующим образом. Пространство итераций разбивается на параллельные гиперплоскости (с общим вектором нормали). Гиперплоскости обходятся в том порядке, который указывает вектор нормали. Точки, находящиеся на одной гиперплоскости, обходятся в лексикографическом порядке.

В данной работе для преобразования гнёзд итерационного типа метод гиперплоскостей будет применяться поэтапно к парам соседних циклов.

Преобразование «метод гиперплоскостей» является комбинацией преобразований «скашивание» и «перестановка циклов» [52]. Он применяется к паре соседних циклов, один из которых непосредственно вложен во второй. Условие эквивалентности метода гиперплоскостей сводится к условию эквивалентности перестановки циклов после скашивания. Если все информационные зависимости в гнезде циклов имеют векторы расстояний и эти векторы не имеют отрицательные координаты, то перестановка циклов эквивалентна [122]. Матрицу преобразования «метод гиперплоскостей» можно определить как произведение матриц преобразований «скашивание» и «перестановка циклов»: $\begin{pmatrix}1 & 0\\ f & 1\end{pmatrix} \times \begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix} = \begin{pmatrix}f & 1\\ 1 & 0\end{pmatrix}$, где f – параметр скашивания.

Метод гиперплоскостей выполняется таким образом, чтобы точки, находящиеся на одной гиперплоскости, были информационно независимы, что позволяет выполнять их параллельно.

Листинг 1.5.7. Гнездо циклов (листинг 1.5.5) после применения преобразования «метод гиперплоскостей», f – параметр скашивания:

```
for (int j = initM + f * initN; j < M + f * N - f; j++){
  for (int i = max(initN, j-M+1); i < min(N, j-initN+1); i++){
      LoopBody(i, j - f * i);
  }
}
```

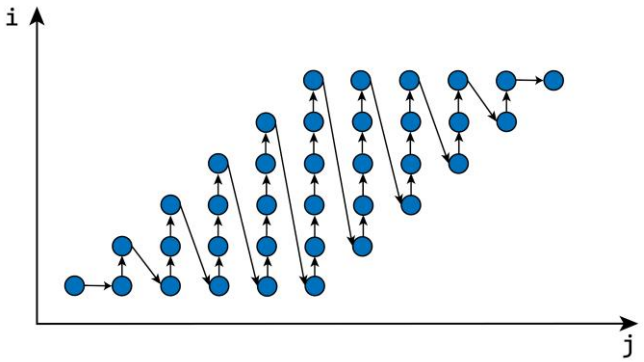

Рисунок 1.37. Пространство итераций двумерного цикла, изменённое с помощью преобразования «метод гиперплоскостей» при параметре скашивания f=1. Точки – точки пространства итераций, дуги – порядок выполнения этих точек; i, j – счётчики гнезда циклов

Иногда это преобразование позволяет параллельно выполнять итерации, находящиеся на одной гиперплоскости (рисунок 1.38).

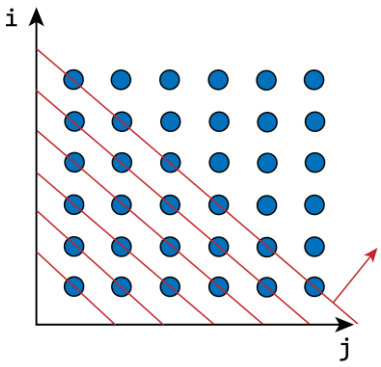

Рисунок 1.38. Пространство итераций гнезда циклов из листинга 1.5.7. Точки – точки пространства итераций; i, j – счётчики гнезда циклов. Красные линии – гиперплоскости

1.5.5. Тайлинг (Loop Tiling, Loop Blocking)

Тайлинг (Loop Tiling, Loop Blocking) – это преобразование программ, которое разбивает пространство итераций исходного гнезда цикла параллельными плоскостями (гиперплоскостями) на блоки (тайлы) меньшего размера и просматривает точки пространства итераций поблочно.

Прямоугольный тайлинг – это тайлинг у которого блоки являются прямоугольными параллелепипедами, грани которых перпендикулярны соответствующим координатным осям. Он является комбинацией преобразований «гнездование циклов» (Loop Nesting) и «перестановка циклов» (Loop Interchange).

На рисунке (рисунок 1.39) представлено пространство итераций двумерного гнезда циклов:

Листинг 1.5.8. Двумерное гнездо циклов:

```
for (int i = 0; i < N1; i++) {
  for (int j = 0; j < N2; j++) {
    LoopBody(i, j);
  }
}
```

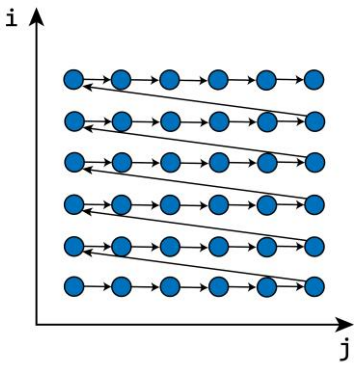

Рисунок 1.39. Пространство итераций гнезда циклов из листинга 1.5.8. Точки – точки пространства итераций, дуги – порядок выполнения этих точек

Листинг 1.5.9. Гнездо циклов после применения прямоугольного тайлинга к гнезду циклов из листинга 1.5.8. N1 и N2 кратны d1, d2 соответственно:

```
for (ii = 0; ii < N1 / d1; ii += 1) {
  for (jj = 0; jj < N2 / d2; jj += 1) {
    for (i = ii * d1; i < (ii + 1) * d1; i=i + 1) {
      for (j = jj * d2; j < (jj + 1) * d2; j=j + 1) {
        LoopBody(i, j);
      }
    }
  }
}
```

Два внешних цикла отвечают за обход тайлов, два внутренних – за обход итераций внутри тайлов. Переменные d1 и d2 отвечают за размер блоков (тайлов) разбиения (основные параметры преобразования). Ниже (рисунок 1.40) представлено разбиение на блоки точек пространства итераций гнезда циклов (листинг 1.5.9).

Размерность пространства увеличивается вдвое, но количество итераций не изменяется, и можно установить взаимно-однозначное соответствие между точками исходного пространства итераций и нового после применения тайлинга. Рисунок 1.40 иллюстрирует, как изменится порядок итераций исходного гнезда циклов после применения прямоугольного тайлинга.

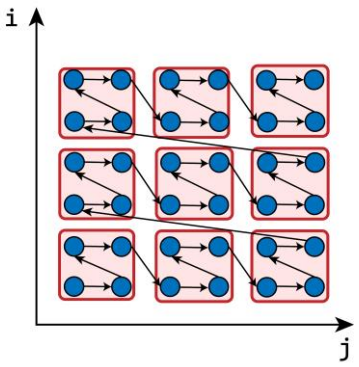

Рисунок 1.40. Пространство итераций двумерного гнезда циклов. Точки – точки пространства итераций, дуги – порядок выполнения этих точек. Красным выделены точки пространства итераций, которые будут находиться в одном блоке (тайле) после применения тайлинга

Прямоугольный тайлинг является эквивалентным не во всех случаях, т. к. тайлинг является комбинацией преобразований и на него распространяются их условия эквивалентности. Гнездование циклов является эквивалентным преобразованием, поскольку оно не меняет порядок выполнения итераций. Перестановка циклов является эквивалентным преобразованием в случае, если все координаты вектора расстояний (distance vector) являются неотрицательными [122].

Пример 1.5.2.

Рассмотрим применение прямоугольного тайлинга к гнезду циклов из листинга 1.5.10.

Листинг 1.5.10. Гнездо циклов алгоритма перемножения матриц.

```
for (i = 0; i < n; ++i) {
      for (j = 0; j < n; ++j) {
            for (k = 0; k < n; ++k) {
                  c[i][j] = c[i][j] + a[i][k] * b[k][j];
             } //    v1    v2
      }
}
```

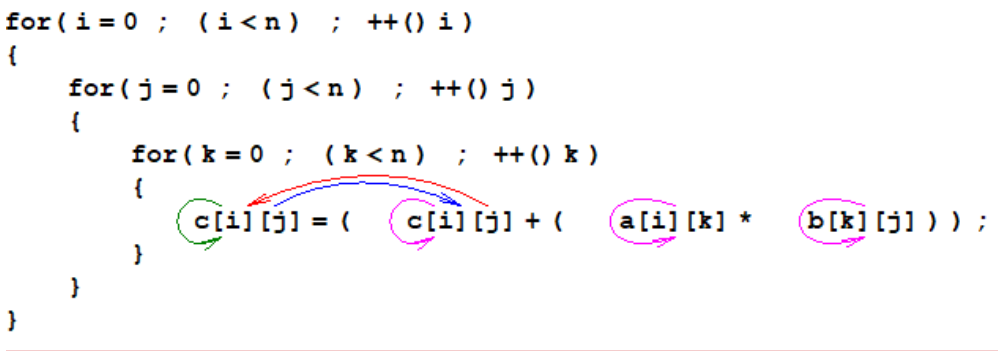

Рисунок 1.41. Граф информационных зависимостей для гнезда циклов из листинга 1.5.10

В данном фрагменте кода присутствуют шесть дуг информационной зависимости: три дуги входной зависимости (они не влияют на эквивалентность преобразования, поэтому их опускаем), одна дуга выходной зависимости (dist(v1,v1) = (0,0,1)), одна дуга потоковой зависимости (dist(v1,v2) = (0,0,0)) и одна дуга антизависимости (dist(v2,v1) = (0,0,1))). Поскольку отрицательные координаты отсутствуют, прямоугольный тайлинг допустим.

Листинг 1.5.11. Гнездо циклов алгоритма перемножения матриц (листинг 1.5.10) после применения преобразования «тайлинг» с прямоугольной формой тайлов. Программа сгенерирована с использованием OPC:

```
for (__uni3i = 0; __uni3i < n / d; __uni3i = __uni3i + 1){
  for (__uni2j = 0; __uni2j < n / d; __uni2j = __uni2j + 1){
    for (__uni1k = 0; __uni1k < n / d; __uni1k = __uni1k + 1){
      for (i = __uni3i * d; i < (__uni3i + 1) * d; i = i + 1){
        for (j = __uni2j * d; j < (__uni2j + 1) * d; j = j + 1){
          for (k = __uni1k * d; k < (__uni1k + 1) * d; k = k + 1){
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
          }
        }
      }
    }
  }
}
```

Пример 1.5.3.

Рассмотрим пример цикла, к которому прямоугольный тайлинг не применим.

Прямоугольный тайлинг не применим для примера 1.1.1, поскольку присутствует дуга истинной информационной зависимости (a[j], a[j+1]), вектор расстояний которой имеет отрицательную координату (1, -1).

1.5.6. Скошенный тайлинг (Skewed Loop Tiling)

Скошенный тайлинг (Skewed Loop Tiling) – преобразование, которое является комбинацией скашивания и прямоугольного тайлинга.

Рассмотрим двумерное гнездо циклов в общем виде.

Листинг 1.5.12. Двумерное гнездо циклов в общем виде:

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

Приведём гнездо циклов после применения скошенного тайлинга с параметром скашивания, равным 1, и соответствующее ему пространство итераций. Рисунок 1.42 иллюстрирует, как изменится порядок итераций исходного гнезда циклов после применения скошенного тайлинга.

Листинг 1.5.13. Гнездо циклов (листинг 1.5.12) после применения преобразования «скошенный тайлинг»:

```
for (int ii = 0; ii < (N - initN - 1) / d1 + 1; ii++)
for (int jj = 0; jj < ((M - initM) + (N - initN) - 1 - 1) / d2 + 1;
jj++)
for (int i = ii * d1; i < min((ii + 1) * d1, N - initN); i++)
for (int jjj = max(i, jj * d2); jjj < min((jj + 1) * d2, (M - initM) +
i); jjj++)
LoopBody(i, jjj - i);
```

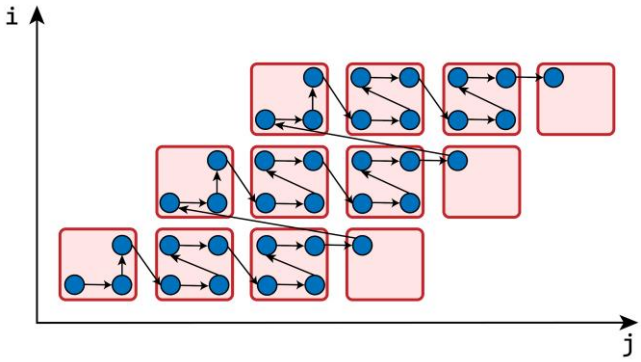

Рисунок 1.42. Пространство итераций двумерного цикла (листинг 1.5.6 – скошенное пространство итераций). Точки – точки пространства итераций, дуги – порядок выполнения этих точек. Красным выделены точки пространства итераций, которые будут находиться в одном блоке (тайле) после применения тайлинга

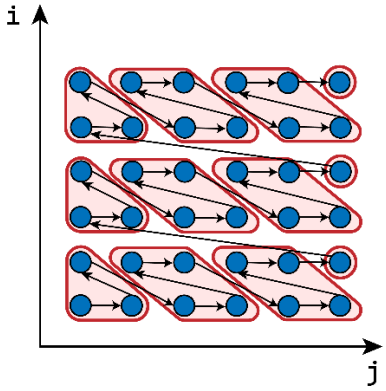

Рисунок 1.43. Пространство итераций двумерного цикла (листинг 1.5.12). Точки – точки пространства итераций, дуги – порядок выполнения этих точек. Красным выделены точки пространства итераций, которые будут находиться в одном блоке (тайле) после применения скошенного тайлинга

Замечание. Если вернуться к исходному пространству итераций, то оно будет разбито не на прямоугольные блоки, а на «скошенные». Тайлы, которые находятся на границах пространства итераций, могут быть неполными.

Следует отметить, что увеличился объём программного кода и усложнились индексные выражения.

Гнёзда циклов, для которых прямоугольный тайлинг не применим, можно преобразовывать, используя скошенный тайлинг, поскольку за счёт скашивания изменяются векторы расстояний информационных зависимостей.

Пример 1.5.4.

Рассмотрим гнездо циклов из примера 1.1.1, к которому не применим прямоугольный тайлинг. На графе информационных зависимостей присутствует дуга антизависимости, вектор расстояний которой равняется (1, -1). В примере 1.5.1 приводится анализ данного гнезда циклов и определяется параметр скашивания – такой, чтобы после применения скашивания вектор расстояния антизависимости не содержал отрицательных координат. Таким образом, к этому гнезду циклов можно применить скошенный тайлинг с параметром скашивания f=1.

1.6. Оптимизирующая распараллеливающая система (OPC)

OPS (Optimizing Paralyzing System) [44] – оптимизирующая распараллеливающая система, проект, разрабатываемый в Институте математики механики и компьютерных наук им. Воровича ЮФУ на кафедре алгебры и дискретной математики под руководством Б. Я. Штейнберга. Проект является основой для создания оптимизирующих компиляторов. Высокоуровневое внутреннее представление OPC [124] имеет древовидную структуру, типы узлов которой делятся на пять групп: узлы для представления типов данных, идентификаторов, операторов, выражений и служебные узлы. Такое внутреннее представление позволяет на своей основе реализовывать различные преобразования циклов, линейных участков программ, условных операторов и др., а также графовые модели программ, такие как граф информационных зависимостей, граф вычислений, управляющий граф программы, граф вызова подпрограмм и др.

На основе OPC реализован диалоговый высокоуровневый оптимизирующий распараллеливатель (ДВОР).

На рисунках ниже представлены скриншоты диалогового окна ДВОР, на которых изображены некоторые реализованные в OPC преобразования из данной главы.

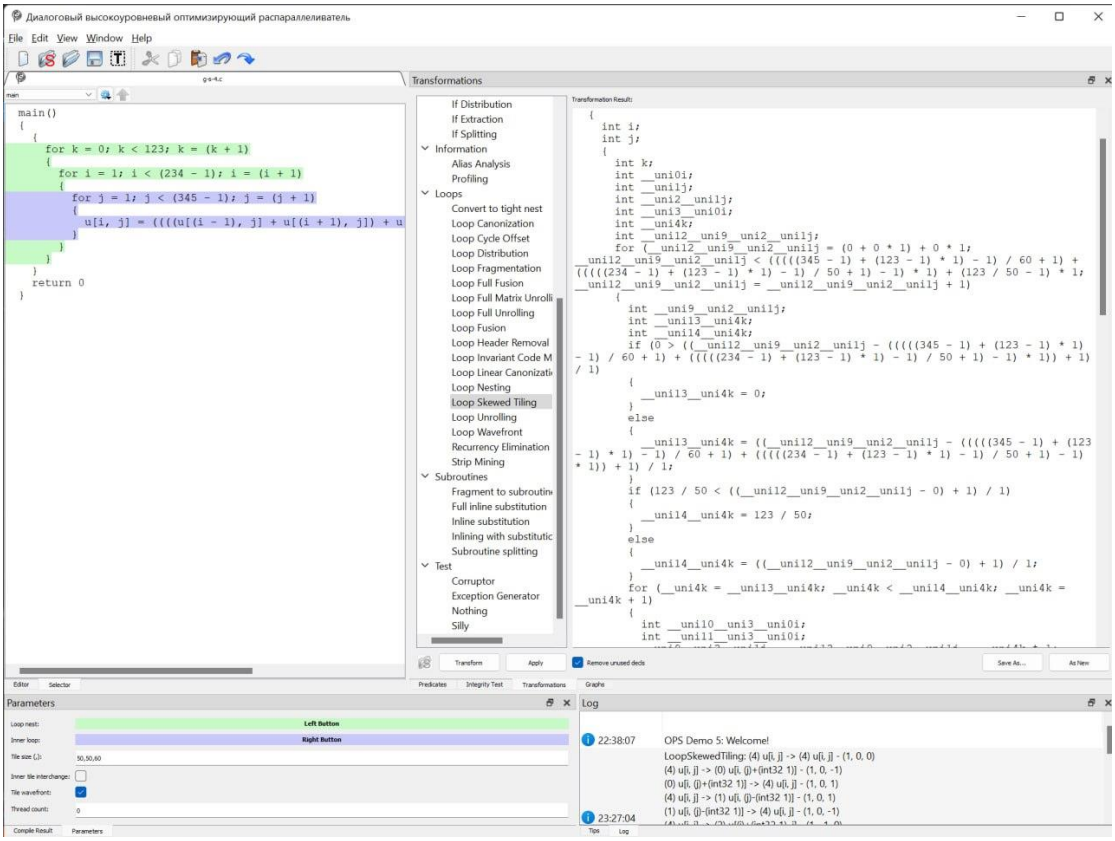

Рисунок 1.44. Диалоговое окно ДВОР. Справа – результат применения преобразования Loop Skewed Tiling к исходной программе, представленной слева

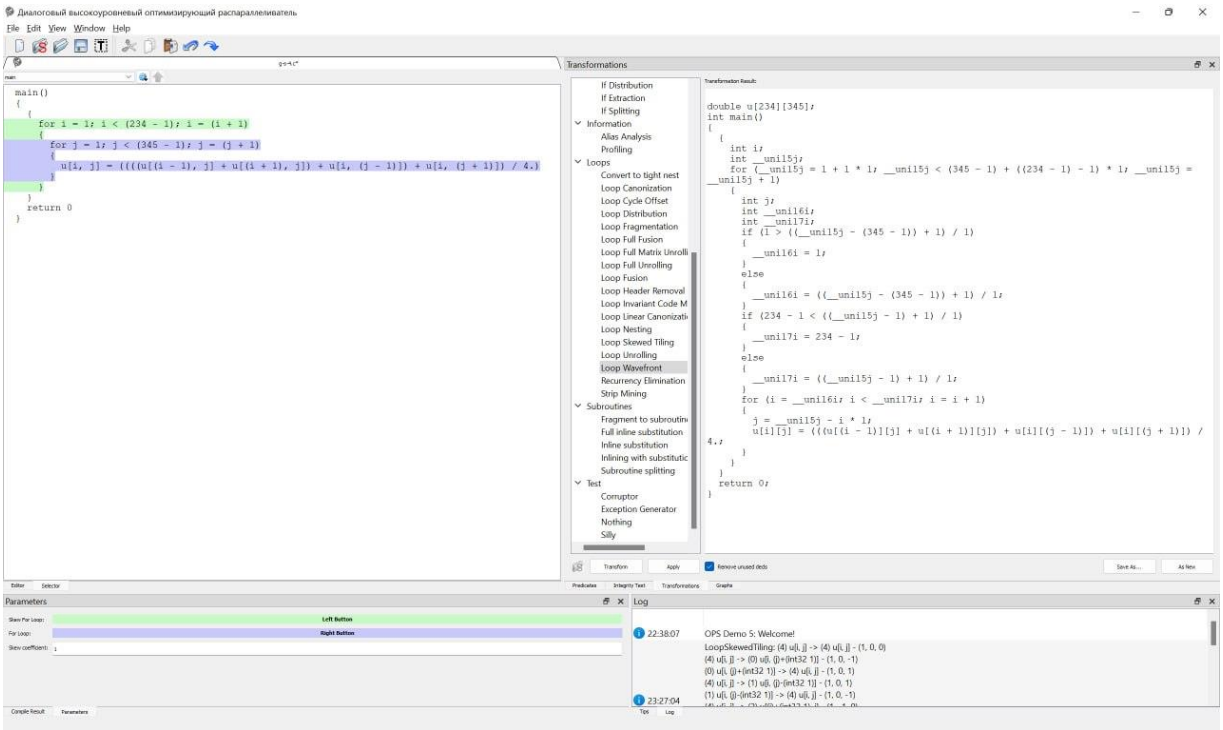

Рисунок 1.45. Диалоговое окно ДВОР. Справа – результат применения преобразования Loop Wavefront к исходной программе, представленной слева

Из внутреннего представления ОРС есть конвертор в язык С. Это позволяет после преобразований ОРС подключать любой компилятор языка С, включая компилятор с закрытым кодом ICC. При наличии отображения шаблонов языка С на CUDA, HDL или C+MPI можно получить компилятор, соответственно, на графическую карту, ПЛИС или вычислительную систему с распределённой памятью. Такой компилятор может иметь предварительные преобразования ОРС, приводящие широкий класс программ к шаблонам. В частности, такие генерации кода могут предварительно выполнять тайлинг, повышающий локальность данных, без чего распараллеливание может вызывать замедление, а не ускорение.

В главе 4 будет представлен режим диалоговой оптимизации программ, для которого существенно наличие в ОРС высокоуровневого внутреннего представления и вывода из него в язык С.

1.7. Выводы к главе 1

В первой главе описаны понятия, необходимые для изложения результатов диссертации: вхождения переменных, граф информационных зависимостей, решетчатый граф, вектор расстояния информационной зависимости. Приводятся определения преобразований программ, таких как «скашивание циклов», «перестановка циклов», «метод гиперплоскостей», «тайлинг» (прямоугольный и скошенный). Тайлинг является оптимизирующим преобразованием, поскольку повышает локальность данных. Вектор расстояния зависимостей вычисляется в представленном в диссертации алгоритме автоматической оптимизации и используется для определения параметров скошенного тайлинга. Приводится обобщение вектора расстояний, которое необходимо для обоснования целевых преобразований гнёзд циклов итерационного типа. В этой главе также описывается метод гиперплоскостей, суть которого состоит в разбиении пространства итераций гнезда циклов на гиперплоскости, точки которых могут выполняться параллельно. В рамках научного исследования приведённые преобразования реализованы в оптимизирующей распараллеливающей системе (ОРС), имеющей высокоуровневое внутреннее представление; тем самым решается задача 1. Реализация этих преобразований в ОРС подтверждается скриншотами (рисунок 1.44, рисунок 1.45). Действия этих преобразований можно увидеть в результирующем коде, представленном в работе [125]. Даётся описание ОРС и некоторых преимуществ для реализации преобразований гнёзд циклов.

Глава 2. Алгоритм оптимизации итерационных гнёзд циклов

Рассматриваемый в данной главе алгоритм оптимизации применим к гнёздам циклов итерационного типа.

Опишем процесс оптимизации гнёзд циклов с применением приведённых ранее преобразований. На вход алгоритму подаётся программа на языке C, содержащая тесное гнездо циклов итерационного типа.

Алгоритм оптимизации итерационных гнёзд циклов:

1. Вычисление параметров преобразования «скашивание» на основе анализа информационных зависимостей (параграф 2.1).
2. Применение скашивания (если необходимо) (параграф 2.2).
3. Применение тайлинга (параграф 2.2).
4. Перестановка циклов внутри тайла для повышения временно́й локальности данных (параграф 2.3).
5. Применение метода гиперплоскостей (параграф 2.4).
6. Применение дополнительных преобразований (параграф 3.8).
7. Применение прагм OpenMP для параллельного выполнения (параграф 2.4).

Результатом применения алгоритма является преобразованная программа на языке C. Полученная программа может компилироваться любым из компиляторов (GCC, ICC, LLVM, MS-Compiler и др.), включая компиляторы с закрытым кодом.

Автоматизация описанного алгоритма реализована в оптимизирующей распараллеливающей системе (ОРС).

Будем иллюстрировать действие алгоритма на примере двумерного цикла (листинг 2.1.1). С входными размерами блоков d1, d2.

Листинг 2.1.1. Тесное двумерное гнездо циклов; в теле гнезда циклов производятся вычисления над двумерным массивом:

```
for (int i = 0; i < N; i++) {          // 0
     for (int j = 0; j < M; j++) {     // 1
          u[j] = u[j] + u[j-1];
     }
}
```

2.1. Вычисление параметров преобразований на основе анализа информационных зависимостей

Для определения подходящего алгоритма тайлинга (прямоугольный или скошенный) производится анализ дуг графа информационных зависимостей. Для каждой дуги вычисляется вектор расстояний (distance vector). Если хотя бы в одном векторе расстояний присутствуют отрицательные координаты, то перед применением тайлинга выполняется скашивание.

Вычисление матрицы скашивания

Пусть на графе информационных зависимостей некоторого гнезда циклов присутствуют K дуг информационной зависимости. Анализируются векторы расстояний этих дуг.

Рассмотрим дугу информационной зависимости (u, v).

Пусть n – количество циклов в гнезде. Определим массив carriers (u, v) носителей дуги (u, v). Поскольку рассматриваются тесные гнёзда циклов, все вхождения переменных находятся в самом внутреннем цикле. Это означает, что для информационных зависимостей, которые порождены этими вхождениями, все циклы гнезда могут являться их носителями, т. е. элементы массива carriers (u, v) принимают значения от нуля до n-1, нумерация циклов производится от внешнего к внутреннему.

Пусть $dist_{m=}dist\ (u, v)_m$, где m от 0 до n-1, – отрицательная координата вектора расстояний дуги (u,v). В таком случае нужно выполнить скашивание циклов с номерами, которые являются элементами массива carriers (u, v), относительно цикла с номером m; параметр скашивания $f=|dist_m|$. В результате получим композицию скашиваний. Матрица этой композиции – s. Пусть S – множество таких матриц скашиваний, построенных для каждой дуги информационной зависимости. Тогда сформируем результирующую матрицу скашивания skew, элементы которой вычисляются по формуле: skew[i][j] = max(s1[i][j], s2[i][j], … , sK[i][j]), где s1, s2, … , sK принадлежат множеству S.

Пример 2.1.1.

Рассмотрим трёхмерное гнездо циклов итерационного типа. Найдём матрицу скашивания, после действия которой у векторов всех расстояний все координаты будут неотрицательными. В комментариях обозначены номера циклов.

```
for (int k = 0; k < N; k++) {                // 0
     for (int i = 0; i < N; i++) {           // 1
          for (int j = 0; j < M; j++) {      // 2
               u[i][j] = u[i][j-1] + u[i+1][j-1] + u[i-2][j-1];
          }
     }
}
```

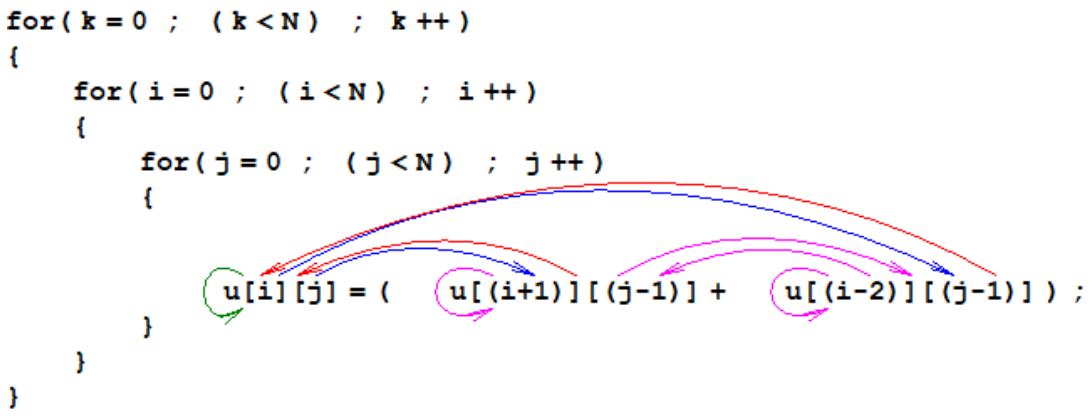

Рисунок 2.1. Граф информационных зависимостей гнезда циклов для примера 1.2.2

В данном гнезде присутствуют пять дуг, которые могут повлиять на эквивалентность выполнения тайлинга: две дуги антизависимости, две дуги истинной зависимости и одна дуга выходной зависимости. Вычислим для них векторы расстояний (dist) и массивы носителей зависимости (carriers):

Дуга информационной зависимости	Вектор расстояний информационной зависимости (dist)	Массив носителей информационной зависимости (carriers)
(u[i][j], u[i][j])	(1, 0, 0)	(0)
(u[i][j], u[i-2][j-1])	(1, 2, 1)	(0,1)
(u[i-2][j-1], u[i][j])	(1, -2, -1)	(0)
(u[i][j], u[i+1][j-1])	(1, -1, 1)	(0)
(u[i+1][j-1], u[i, j])	(1, 1, -1)	(0,1)

Векторы расстояний дуг (u[i-2][j-1], u[i][j]), (u[i][j], u[i+1][j-1]), (u[i+1][j-1], u[i, j]) имеют отрицательные координаты. Вычислим для каждой дуги матрицу скашивания.

Вектор расстояний дуги (u[i-2][j-1], u[i][j]) имеет две отрицательные координаты $dist(u[i-2][j-1], u[i][j])_1 = -2$, $dist(u[i-2][j-1], u[i][j])_2 = -1$. Носитель информационной зависимости – цикл под номером 0. Значит, нужно выполнить два скашивания: цикла под номером 0 относительно цикла под номером 1 с параметром скашивания 2 (матрица скашивания $\begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$); цикла под номером 0 относительно цикла под номером 2 с параметром скашивания 1 (матрица скашивания $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$). Тогда матрица композиции скашиваний $= \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$.

После скашивания с такой матрицей все координаты вектора расстояний рассматриваемой дуги станут неотрицательными: $\begin{pmatrix}1 & 0 & 0\\ 2 & 1 & 0\\ 1 & 0 & 1\end{pmatrix} \text{x} \begin{pmatrix}1\\ -2\\ -1\end{pmatrix} = \begin{pmatrix}1\\ 0\\ 0\end{pmatrix}$.

Замечание. Для любой матрицы скашивания, у которой элементы ниже главной диагонали не меньше, чем элементы полученной матрицы, скашивание тоже приведёт к неотрицательным координатам вектора расстояний.

Вектор расстояний дуги (u[i][j], u[i+1][j-1]) имеет одну отрицательную координату $\text{dist}(u[i][j], u[i+1][j-1])_1 = -1$. Носитель информационной зависимости – цикл под номером 0. Значит, нужно выполнить скашивание цикла под номером 0 относительно цикла под номером 1 с параметром скашивания 1. Матрица скашивания = $\begin{pmatrix}1 & 0 & 0\\ 1 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}$. После скашивания с такой матрицей все координаты вектора расстояний рассматриваемой дуги станут неотрицательными: $\begin{pmatrix}1 & 0 & 0\\ 1 & 1 & 0\\ 0 & 0 & 1\end{pmatrix} \text{x} \begin{pmatrix}1\\ -1\\ 1\end{pmatrix} = \begin{pmatrix}1\\ 0\\ 0\end{pmatrix}$.

Вектор расстояний дуги (u[i+1][j-1], u[i, j]) имеет одну отрицательную координату $\text{dist}(u[i+1][j-1], u[i, j])_2 = -1$. Носители информационной зависимости – циклы под номером 0 и 1. Значит, нужно выполнить два скашивания: цикла под номером 0 относительно цикла под номером 2 с параметром скашивания 1 (матрица скашивания $\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 1 & 0 & 1\end{pmatrix}$); цикла под номером 1 относительно цикла под номером 2 с параметром скашивания 1 (матрица скашивания $\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 1 & 1\end{pmatrix}$); тогда матрица композиции скашиваний = $\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 1 & 0 & 1\end{pmatrix} \text{x} \begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 1 & 1\end{pmatrix} = \begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 1 & 1 & 1\end{pmatrix}$.
После скашивания с такой матрицей все координаты вектора расстояний рассматриваемой дуги станут неотрицательными: $\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 1 & 1 & 1\end{pmatrix} \text{x} \begin{pmatrix}1\\ 1\\ -1\end{pmatrix} = \begin{pmatrix}1\\ 1\\ 1\end{pmatrix}$.

Замечание. В случае нескольких носителей информационной зависимости, как у дуги (u[i+1][j-1], u[i, j]), чтобы вектор расстояний удовлетворял условию применимости тайлинга, достаточно повлиять одной из матриц скашивания:

$\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 1 & 0 & 1\end{pmatrix} \text{x} \begin{pmatrix}1\\ 1\\ -1\end{pmatrix} = \begin{pmatrix}1\\ 1\\ 0\end{pmatrix}$ или $\begin{pmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 1 & 1\end{pmatrix} \text{x} \begin{pmatrix}1\\ 1\\ -1\end{pmatrix} = \begin{pmatrix}1\\ 1\\ 0\end{pmatrix}$.

Получим результирующую матрицу скашивания, выбрав максимальные значения для каждой пары координат матриц $\begin{pmatrix}1&0&0\\2&1&0\\1&0&1\end{pmatrix}, \begin{pmatrix}1&0&0\\1&1&0\\0&0&1\end{pmatrix}, \begin{pmatrix}1&0&0\\0&1&0\\1&1&1\end{pmatrix}: \begin{pmatrix}1&0&0\\2&1&0\\1&1&1\end{pmatrix}$.

Замечание. Скашивание должно применяться по строкам матрицы скашивания (по строкам, сверху-вниз). Так, например, для матрицы скашивания $\begin{pmatrix}1&0&0\\2&1&0\\1&1&1\end{pmatrix}$: (1) применяется скашивание с матрицей $\begin{pmatrix}1&0&0\\2&1&0\\0&0&1\end{pmatrix}$; (2) применяются скашивания с матрицами $\begin{pmatrix}1&0&0\\0&1&0\\1&0&1\end{pmatrix}$ и $\begin{pmatrix}1&0&0\\0&1&0\\0&1&1\end{pmatrix}$ в произвольном порядке.

Пример 2.1.2.

Проведём анализ информационных зависимостей гнезда циклов из листинга 2.1.1. В данном гнезде присутствуют две дуги антизависимости: (u[j], u[j]), distance vector = (1,0) (пример 1.3.13), (u[j-1], u[j]), distance vector = (1,-1) (пример 1.3.14). Информационная зависимость (u[j-1], u[j]) нарушает условие эквивалентности прямоугольного тайлинга, поэтому к данному гнезду циклов следует применить **скошенный тайлинг**. Используя алгоритм, описанный выше, вычислим матрицу скашивания для данного набора информационных зависимостей (1,0), (1,-1). Как уже определили, зависимость (u[j-1], u[j]), имеющая вектор расстояний dist = (1,-1), нарушает условие применимости прямоугольного тайлинга. Отрицательная координата вектора расстояний: $dist(u[j-1], u[j])_1 = -1$, носитель зависимости = 0, тогда матрица скашивания skew = $\begin{pmatrix}1&0\\1&1\end{pmatrix}$.

2.2. Применение тайлинга

Основываясь на анализе информационных зависимостей, применяют либо прямоугольный тайлинг, либо скошенный.

Параметром **прямоугольного тайлинга** является целочисленный массив размеров тайлов (tile_sizes). Размер массива tile_sizes равен количеству циклов гнезда.

Параметры для **скошенного тайлинга**: целочисленный массив размеров тайлов (tile_sizes), который задаётся пользователем; матрица скашивания (skew). Вначале к циклам гнезда применяется композиция скашиваний, определяемая матрицей skew. Затем применяется прямоугольный тайлинг с размерами тайлов (tile_sizes).

Тайлинг сводится к выполнению гнездования и перестановки циклов. Условия выполнения этих преобразований описаны в главе 1. Проиллюстрируем выполнение тайлинга на примере.

Пример 2.2.1 (продолжение примера 2.1.2).

Исходя из анализа информационных зависимостей, приведённого в примере 2.1.2, к гнезду циклов (листинг 2.1.1) применим скошенный тайлинг с матрицей скашивания skew = $\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ и размерами блоков tile_sizes = (d1, d2).

Приведём результат применения скошенного тайлинга с размерами блоков tile_sizes=(d1, d2) и обобщённой матрицей скашивания skew = $\begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix}$, где f – параметр скашивания.

Листинг 2.2.1. Результат применения скошенного тайлинга с параметром скашивания f и размерами блоков (d1, d2) к гнезду циклов (листинг 2.1.1). Первое гнездо циклов является основным (внешние два цикла гнезда отвечают за обход по блокам (тайлам), внутренние два цикла – за обход итераций внутри блока); второе и третье возникают при условии, если количество итераций циклов гнезда не делится на размеры соответствующих им блоков нацело.

// Основное гнездо алгоритма. Внешние два цикла гнезда отвечают за обход по блокам (тайлам), внутренние два цикла – за обход итераций внутри блока.

```
for (int ii = 0; ii < N/d1; ii++) {
     for (int jjj = 0; jjj < (M + f*(N-1))/d2; jjj++) {
          for (int i = ii*d1; i < (ii+1)*d1; i++) {
               for (int jj = max(f*i, jjj*d2); jj < min((jjj+1)*d2,
               M + f*i); jj++) {
                    int j = jj - f*i;
                    u[j] = u[j] + u[j-1];
               }
          }
     }
}
```

// Вычисление остаточных итераций. Возникает, если N не делится на d1 нацело.

```
for (int i=d1*(N/d1); i < N; i++) {
     for (int jjj = 0; jjj < (M + f*(N-1))/d2; jjj++) {
          for (int jj = max(f*i, jjj*d2); jj < min((jjj+1)*d2,
          M + f*i); jj++) {
               int j = jj - f*i;
               u[j] = u[j] + u[j-1];
          }
     }
}
```

// Вычисление остаточных итераций. Возникает, если M + f(N-1) не делится на d2 нацело.*

```
for (int i = 0; i < N; i++) {
      for (int jj=max(f*i,d2*((M + f*(N-1))/d2)); jj <
      min((M + f*(N-1)), M + f*i); jj++) {
            int j = jj - f*i;
            u[j] = u[j] + u[j-1];
      }
}
```

2.3. Перестановка циклов внутри тайла для повышения временно́й локальности данных

Для повышения временно́й локальности данных при вычислении тайла используется преобразование «перестановка циклов». Условия корректности перестановки циклов описаны в [122]. В данном случае после применения тайлинга условия эквивалентности выполняются (т. к. все векторы расстояний не будут иметь отрицательных координат).

Выбираются циклы, отвечающие за обход тайла. Определяется самый вложенный цикл (innermost loop) и переставляется с вышестоящими посредством преобразования «перестановка циклов», чтобы выбранный цикл стал внешним. До перестановки обход точек тайла проводился по целым точкам сечений тайла, параллельным нижней грани. После перестановки обход производится по целым точкам сечений тайла, параллельным боковой грани, ближайшей к началу координат.

Пример 2.3.1 (продолжение примера 2.2.1).

Применим описанную перестановку циклов для листинга 2.2.1.

Листинг 2.3.1. Применение перестановки для внутренних циклов основного гнезда, со счётчиками (ii, jjj, i, jj) (листинг 2.2.1), отвечающих за обход итераций внутри блока.

```
for (int ii = 0; ii < N/d1; ii++) {
      for (int jjj = 0; jjj < (M + f*(N-1))/d2; jjj++) {
            for (int jj = max(f*ii*d1, jjj*d2); jj < min((jjj+1)*d2,
            M + f*(ii+1)*d1); jj++)    {
                  for (int i = max(ii*d1, (jj - M + f)/f);
                  i < min((ii+1)*d1, (jj - 0 + f)/f); i++) {
                        int j = jj - f*i;
                        u[j] = u[j] + u[j-1];
                  }
            }
      }
}
```

```
for (int i=d1*(N/d1); i < N; i++) {
      for (int jjj = 0; jjj < (M + f*(N-1))/d2; jjj++) {
            for (int jj = max(f*i, jjj*d2); jj < min((jjj+1)*d2,
            M + f*i); jj++) {
                  int j = jj - f*i;
                  u[j] = u[j] + u[j-1];
            }
      }
}

for (int i = 0; i < N; i++) {
      for (int jj=max(f*i,d2*((M + f*(N-1))/d2)); jj <
      min((M + f*(N-1)), M + f*i); jj++) {
            int j = jj - f*i;
            u[j] = u[j] + u[j-1];
      }
}
```

На рисунках ниже (рисунок 2.2, рисунок 2.3) проиллюстрировано изменение порядка обхода итераций, которое получается после применения описанной перестановки, для программы из листинга 2.2.1.

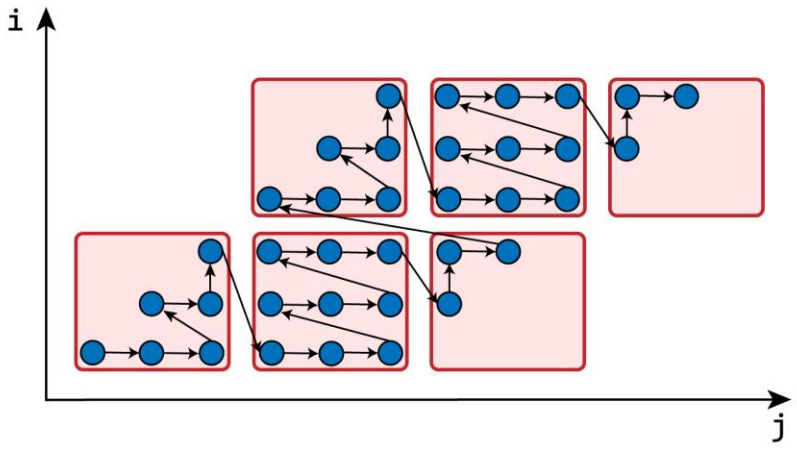

Рисунок 2.2. Порядок выполнения итераций двумерного гнезда циклов после применения преобразования «скошенный тайлинг» (листинг 2.2.1) с размером блоков 3x3, параметром скашивания f =1. Точки – точки пространства итераций, дуги – порядок выполнения точек. Красным выделены блоки (тайлы)

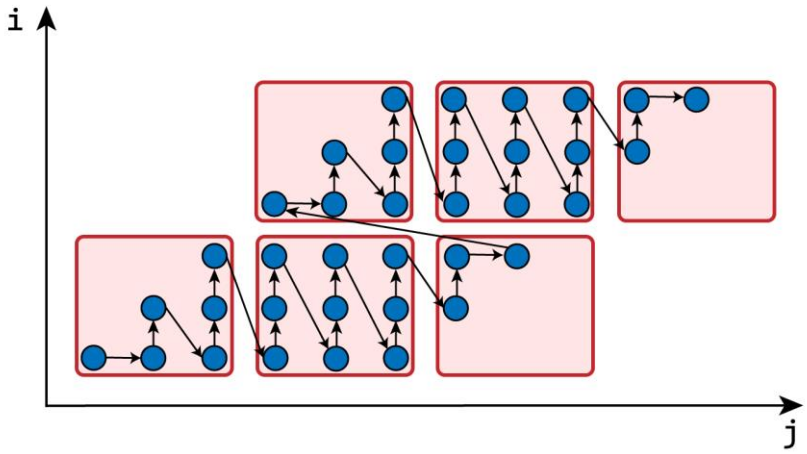

Рисунок 2.3. Порядок выполнения итераций двумерного гнезда циклов после применения перестановки циклов (листинг 2.3.1), отвечающих за обход внутри тайла. Точки – точки пространства итераций, дуги – порядок выполнения точек. Красным выделены блоки (тайлы)

Данное преобразование может повышать временную локальность за счёт того, что вычисленные на предыдущих итерациях данные используются на следующих до того, как будут вытеснены из кэш-памяти и регистров. Ускорение, которое достигается описанной перестановкой, демонстрируется в параграфе 3.6 на примере алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле уравнения Лапласа. Для данной задачи была выбрана матрица скашивания skew = $\begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$.

Для данной задачи определим, откуда считываются данные при вычислении одного тайла без перестановки и с перестановкой. Обозначим время чтения из оперативной памяти – M, из кэш-памяти – K и из регистров – R.

Для алгоритма Гаусса – Зейделя двумерной задачи Дирихле уравнения Лапласа на каждой итерации производится вычисление элемента массива на основе его соседних элементов (u[i][j]=(u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1])/4). Таким образом выполняются четыре чтения из памяти.

В первом случае, когда обход точек тайла осуществляется по горизонтальным плоскостям (сечениям), получим:

При вычислении первого сечения:

Таблица 2.1. Чтение данных первого сечения при выполнении по горизонтальным сечениям

	Чтения	Количество точек
Первая выполняемая точка тайла	4*M	1
Край сечения – нижний	R+3M	a-1
Край сечения – левый	2K+2M	b-1
Внутренние точки сечения	2K+R+M	a*b – (a+b-1)

При вычислении последующих сечений:

Таблица 2.2. Чтение данных последующих сечений при выполнении по горизонтальным сечениям

	Чтения	Количество точек
Первая выполняемая точка тайла	2K+2M	c-1
Край сечения – нижний	R+K+2M	(c-1)*(a-1)
Край сечения – левый	3K+M	(c-1)*(b-1)
Внутренние точки сечения	R+3K	(c-1)*(a-1)*(b-1)

Для второго варианта, при котором обход точек тайла осуществляется по плоскостям (сечениям), параллельным боковой грани тайла, получим:

Таблица 2.3. Чтение данных первого сечения при выполнении по боковым сечениям

	Чтения	Количество точек
Первая выполняемая точка тайла	4*M	1
Край сечения – нижний	K+3M	a-1
Край сечения – левый	2R+2M	b-1
Внутренние точки сечения	K+2R+M	a*b – (a+b-1)

При вычислении последующих сечений:

Таблица 2.4. Чтение данных последующих сечений при выполнении по боковым сечениям

	Чтения	Количество точек
Первая выполняемая точка тайла	2K+2M	c-1
Край сечения – нижний	2K+2M	(c-1)*(a-1)
Край сечения – левый	2R+K+M	(c-1)*(b-1)
Внутренние точки сечения	2R+2K	(c-1)*(a-1)*(b-1)

Во втором случае при обходе точек тайла по боковым сечениям для внутренних точек будет больше чтений данных из регистров, чем в первом случае. Это увеличивает временную локальность данных и даёт ускорение.

2.4. Применение метода гиперплоскостей и прагм OpenMP для параллельного выполнения тайлов

Основная идея алгоритма, ускоряющего вычисление гнёзд циклов итерационного типа, состоит в том, чтобы параллельно выполнять не отдельные точки пространства итераций, а блоки (тайлы) таких точек.

На начальных этапах алгоритма (глава 2) производится анализ информационных зависимостей (параграф 2.1) и, если требуется, выполняется скашивание. Получаем такое гнездо циклов, в котором для каждой информационной зависимости вектор расстояний не имеет отрицательных координат, что позволяет применять тайлинг.

После выполнения тайлинга можно концептуально построить фактор-граф решетчатого графа по тайлам, т. е. вершинами такого фактор-графа являются тайлы. Дуга между вершинами фактор-графа существует, если между точками тайлов, соответствующих этим вершинам, присутствует дуга информационной зависимости.

Покрываем вершины фактор-графа семейством параллельных гиперплоскостей. Удобно брать семейство гиперплоскостей с вектором нормали, имеющим все координаты единицы: (1,1,…,1) (рисунок 2.4), поскольку тогда получаются гиперплоскости, более насыщенные тайлами, что выгодно для распараллеливания. Приведём обоснование того, что тайлы, лежащие в таких гиперплоскостях, попарно не связаны информационными зависимостями.

Теорема 1. После применения тайлинга тайлы, лежащие на одной гиперплоскости с вектором нормали (1,1,…,1), попарно не связаны информационными зависимостями и, следовательно, могут выполняться параллельно.

Доказательство ([126])**.**

Рассмотрим двумерный случай. Предположим, что существует пара лежащих в одной гиперплоскости информационно зависимых вершин фактор-графа решетчатого графа по тайлам. Тогда в соответствующих тайлах есть точки u = (a, b), v = (c, d) пространства итераций, между которыми есть дуга решетчатого графа (u, v), соответствующая некоторой дуге графа информационных зависимостей. Эта дуга решетчатого графа имеет вектор расстояния dist (u, v) = (c-a, d-b). Покажем, что этот вектор будет иметь отрицательную координату.

Пусть d_1, d_2 – размеры тайлов, (x_1, y_1), (x_2, y_2) – координаты этих тайлов на фактор-графе, к которым принадлежат вхождения u, v; соответственно, тогда выполняются неравенства:

$$x1 * d1 \leq a < (x1 + 1) * d1$$
$$y1 * d2 \leq b < (y1 + 1) * d2$$
$$x2 * d1 \leq c < (x2 + 1) * d1$$
$$y2 * d2 \leq d < (y2 + 1) * d2$$

Поскольку (x1, y1), (x2, y2) – координаты тайлов, лежащих на одной гиперплоскости с вектором нормали (1,1), то (x2-x1) + (y2-y1) = 0. Обозначим n = x2-x1, тогда y2-y1 = -n,

$$x1 * d1 \leq a < (x1 + 1) * d1$$
$$y1 * d2 \leq b < (y1 + 1) * d2$$
$$(x1 + n) * d1 \leq c < ((x1 + n) + 1) * d1$$
$$(y1 - n) * d2 \leq d < ((y1 - n) + 1) * d2$$

Из этого следует, что при n >= 1: a < c, b > d, т. е. c – a > 0, d – b < 0. При n < 1: a > c, b < d, т. е. c – a < 0, d – b > 0. Получается, что одна из координат вектора расстояний будет отрицательной.

Конец доказательства.

Следствие 1. Применение метода гиперплоскостей с вектором нормали (1, 1, … , 1) и последующее распараллеливание являются эквивалентными.

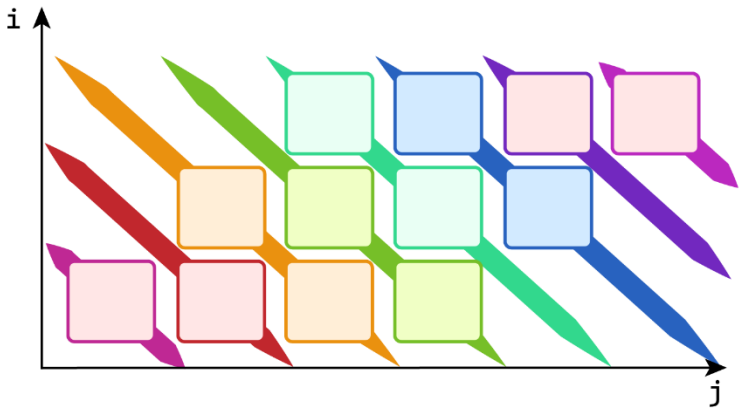

Рисунок 2.4. Фактор-граф по тайлам для гнезда циклов (листинг 1.5.13). Блоки (тайлы из листинга 1.5.13) – вершины фактор-графа. Фактор-граф покрыт семейством параллельных гиперплоскостей с вектором нормали = (1,1). Блоки, находящиеся на одной гиперплоскости, выделены одним цветом

Для того чтобы покрыть вершины фактор-графа семейством параллельных гиперплоскостей, с выбранным вектором нормали (1, 1, … , 1), выполняется следующая последовательность действий:

0. Для гнезда циклов (размерности n+1) после применения тайлинга (размерность гнезда 2*(n+1)) выбираются циклы, отвечающие за обход тайлов (первые n+1 циклов преобразованного гнезда).
1. Из выбранных циклов определяется внутренний, с порядковым номером n (нумерация циклов от 0). Переменной K присвоим номер n.
2. Преобразование Loop Wavefront, с входным параметром, равным 1 (параметр скашивания), применимо к циклам под номерами K, K-1. Порядковый номер выбранного цикла K после применения преобразования будет равняться K-1.
3. Если порядковый номер выбранного цикла K не равняется нулю, то переходим к пункту (2).
4. Циклы с номерами от 1 до n отвечают за обход тайлов, лежащих на одной гиперплоскости. К ним можно применить распараллеливание, например, за счёт добавления прагм OpenMP.

После применения описанного алгоритма может быть применено преобразование «линеаризация выражений», которое дополнительно повышает производительность.

Рисунки (рисунок 2.5 – рисунок 2.9) иллюстрируют результат применения метода гиперплоскостей, порядок выполнения блоков для трёхмерного гнезда циклов, а также то, какие блоки могут быть выполнены параллельно.

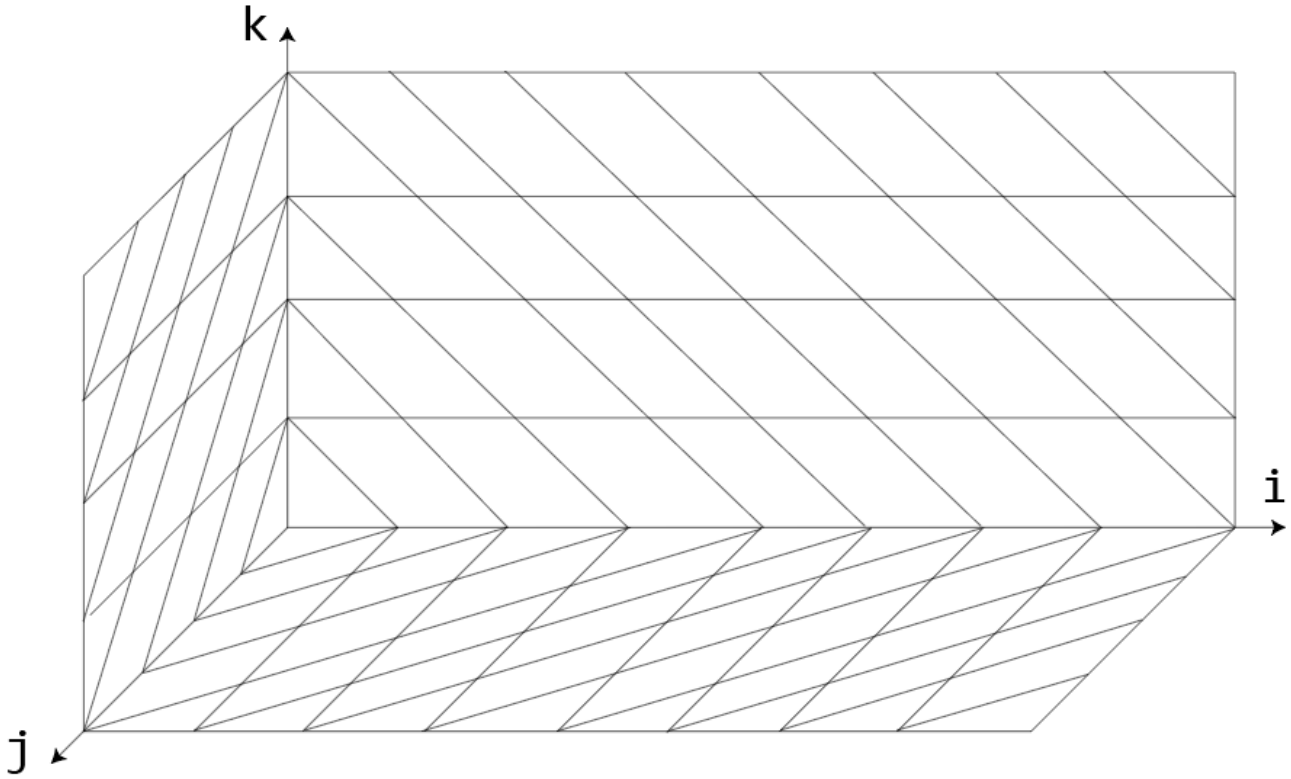

Рисунок 2.5. Трёхмерное пространство итераций, разбитое на блоки скошенным тайлингом. Линии на координатных плоскостях – границы тайлов

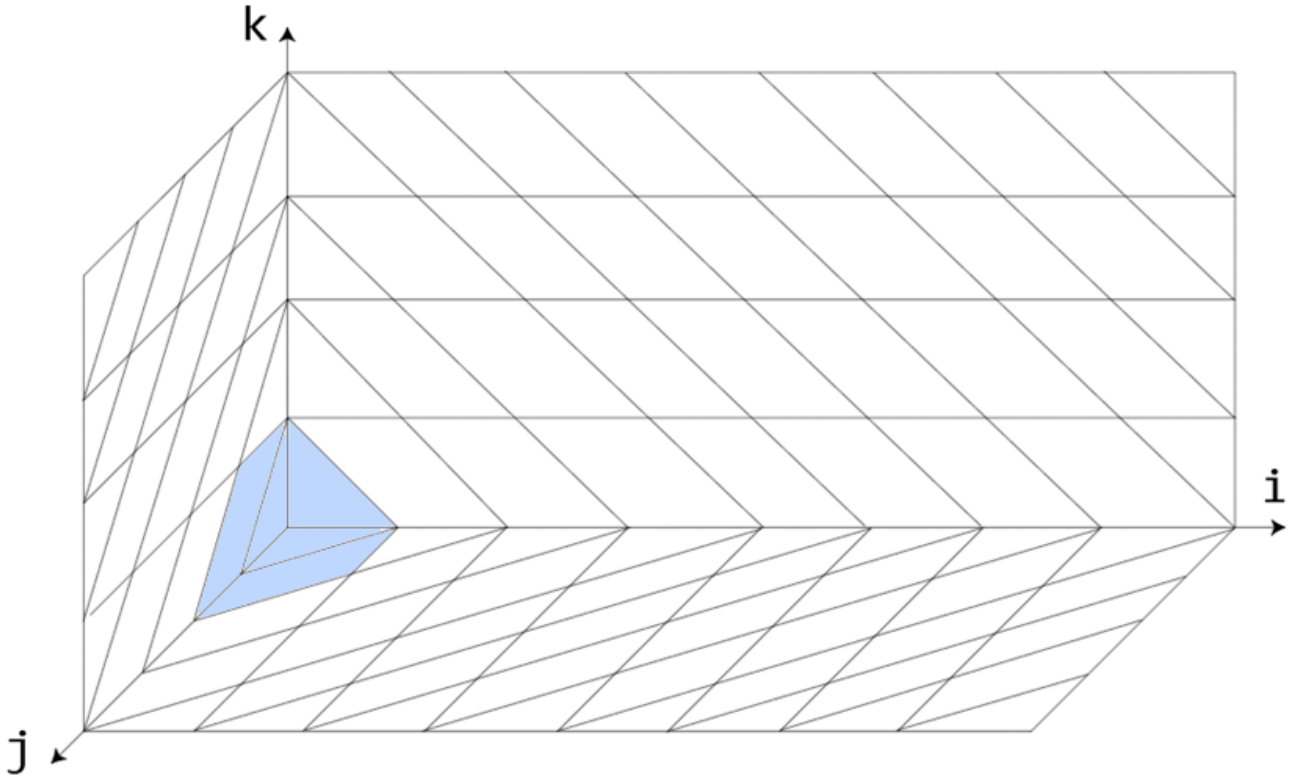

Рисунок 2.6. Синим цветом выделены тайлы, которые должны выполняться последовательно

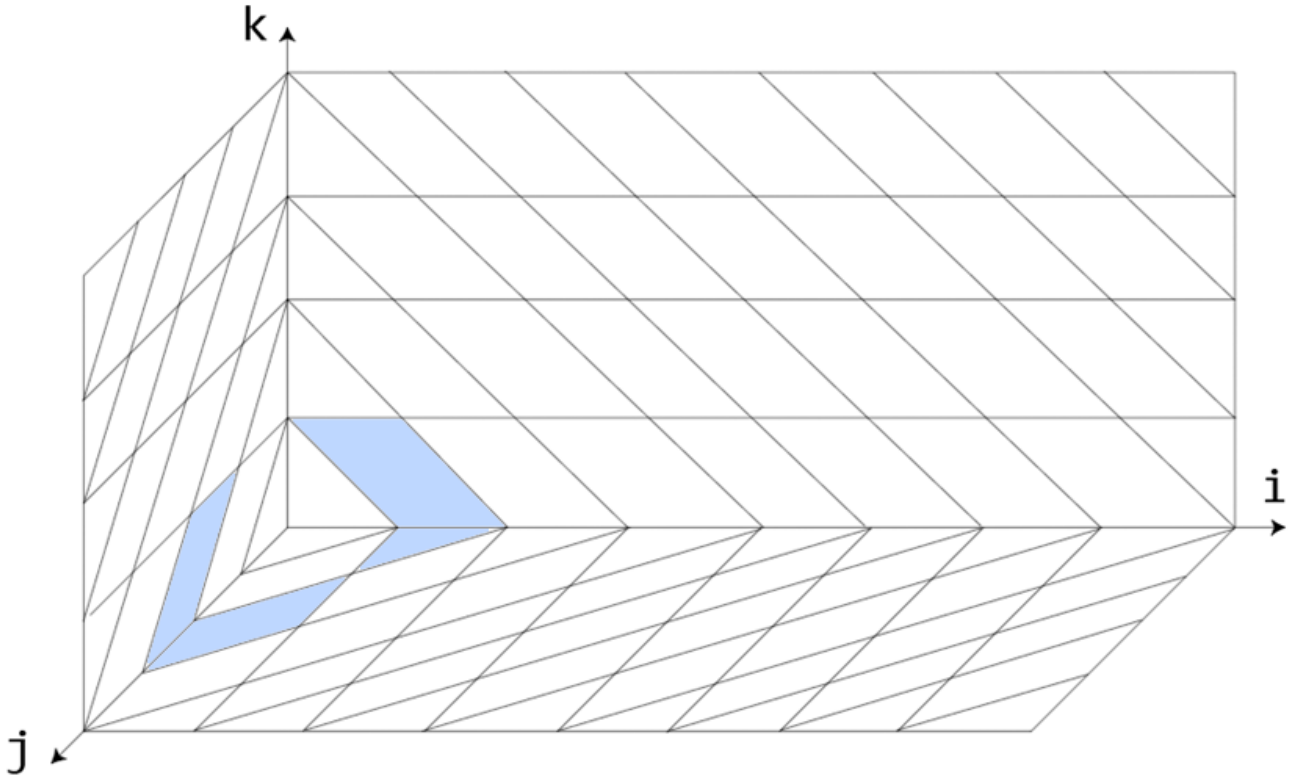

Рисунок 2.7. Синим выделены тайлы, находящиеся на одной гиперплоскости

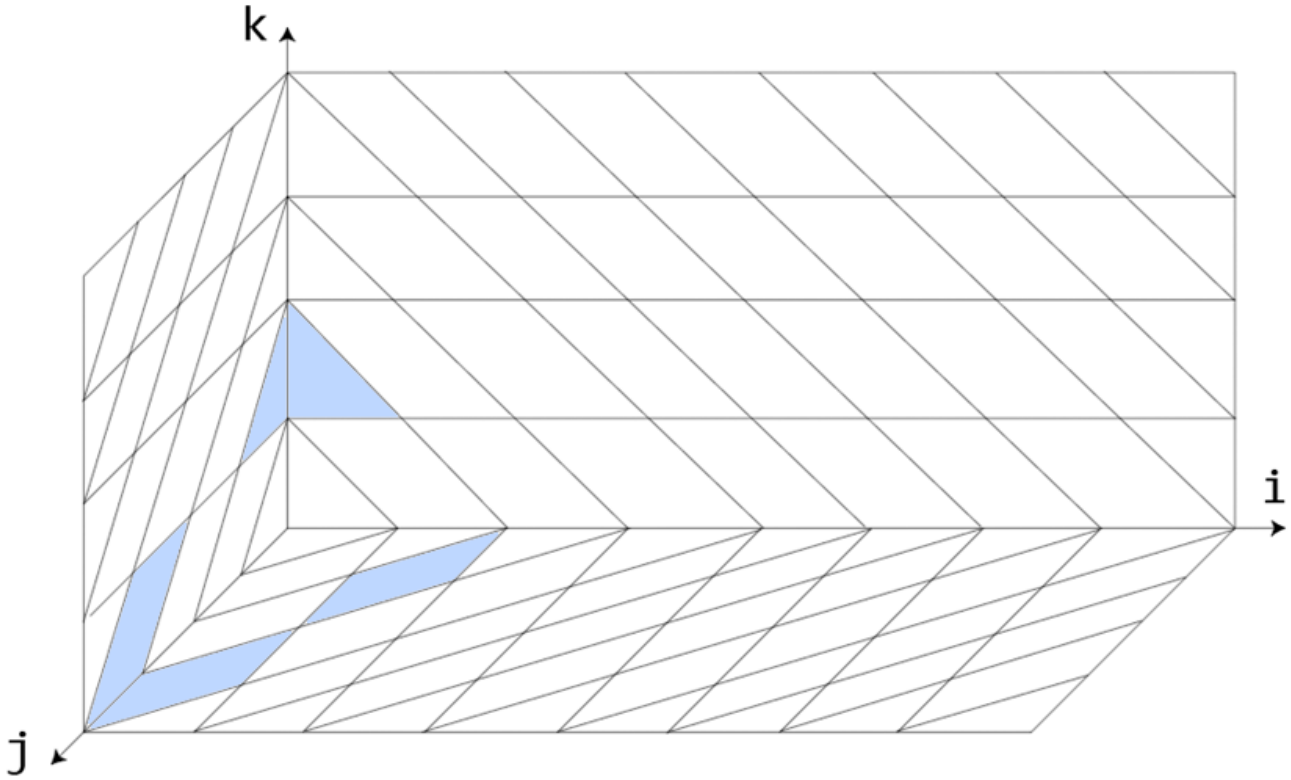

Рисунок 2.8. Синим выделены тайлы, находящиеся на одной гиперплоскости

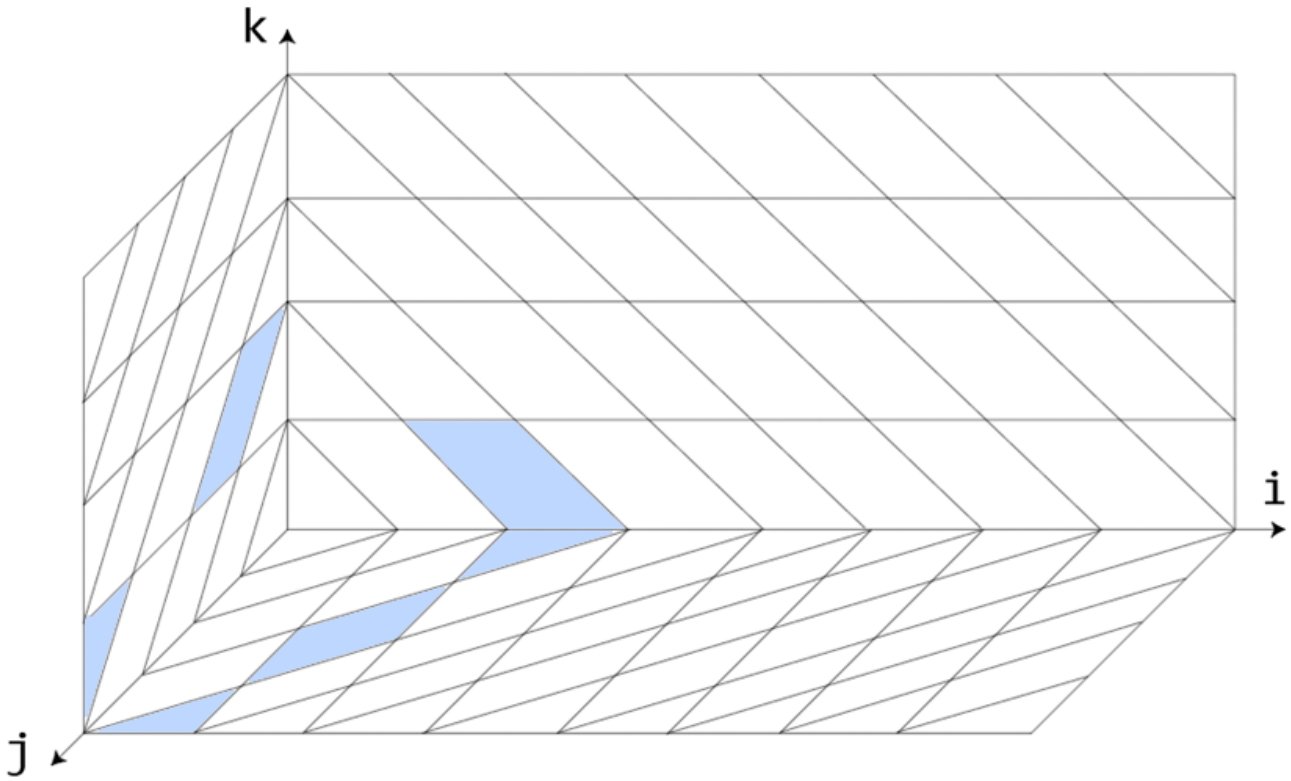

Рисунок 2.9. Синим выделены тайлы, находящиеся на одной гиперплоскости

После применения описанного алгоритма может быть применено преобразование «линеаризация выражений», которое дополнительно повышает производительность.

Пример 2.4.1 (продолжение примера 2.3.1).

Применим алгоритм, описанный выше для программы из листинга 2.3.1.

Листинг 2.4.1. Применение метода гиперплоскостей (Wavefront) и распараллеливания с помощью прагм OpenMP к основному гнезду циклов, со счётчиками (ii, jjj, jj, i) (листинг 2.3.1).

```
for (int jjjj = 0; jjjj < (M + f*(N-1))/d2 + fw*N/d1; jjjj++){
      #pragma omp parallel
      for (int ii = max(0, jjjj - (M + f*(N-1))/d2 + 1); ii <
      min(N/d1, jjjj + 1; ii++) {
            int jjj = jjjj - fw*ii
            for (int jj = max(f*ii*d1, jjj*d2); jj < min((jjj+1)*d2,
            M + f*(ii+1)*d1); jj++)    {
                  for (int i = max(ii*d1, (jj - M + f)/f);
                  i < min((ii+1)*d1, (jj - 0 + f)/f); i++) {
                        int j = jj - f*i;
                        u[j] = u[j] + u[j-1];
                  }
            }
      }
}
```

```
for (int i=d1*(N/d1); i < N; i++) {
     for (int jjj = 0; jjj < (M + f*(N-1))/d2; jjj++) {
          for (int jj = max(f*i, jjj*d2); jj < min((jjj+1)*d2,
          M + f*i); jj++) {
               int j = jj - f*i;
               u[j] = u[j] + u[j-1];
          }
     }
}
for (int i = 0; i < N; i++) {
     for (int jj=max(f*i,d2*((M + f*(N-1))/d2)); jj <
     min((M + f*(N-1)), M + f*i); jj++) {
          int j = jj - f*i;
          u[j] = u[j] + u[j-1];
     }
}
```

2.5. Эквивалентность алгоритма оптимизации итерационных гнёзд циклов

Теорема 2. Алгоритм оптимизации итерационных гнёзд циклов и его расширение являются эквивалентными.

Доказательство ([126])**.**

Алгоритм состоит из преобразований «скашивание», «тайлинг» («гнездование циклов», «перестановка циклов»), «перестановка циклов», «метод гиперплоскостей» («скашивание», «перестановка циклов»). Расширение алгоритма, представленное в параграфе 3.8, дополняет представленный алгоритм преобразованиями «линеаризация» и «вынос общих инвариантных выражений». Доказательство эквивалентности алгоритма сводится к доказательству эквивалентности каждого преобразования последовательности.

Скашивание и гнездование не меняют порядок обращения к памяти, а потому являются эквивалентными. Если все информационные зависимости в гнезде циклов имеют векторы расстояний и эти векторы не имеют отрицательных координат, то перестановка циклов эквивалентна [127], [122]. Для проверки эквивалентности тайлинга производится анализ информационных зависимостей. При наличии зависимостей, имеющих отрицательные координаты вектора расстояний, выполняется с соответствующими параметрами скашивание так, чтобы все векторы расстояний имели неотрицательные координаты (параграф 2.1). Эквивалентность метода гиперплоскостей обосновывается в параграфе 2.4. Преобразования «линеаризация» (описывается в параграфе 4.2) и «вынос общих инвариантных выражений» являются эквивалентными. Таким образом, описанный алгоритм оптимизации итерационных гнёзд циклов и его расширение являются эквивалентными.

Конец доказательства.

2.6. Перспективы развития и применения преобразований гнёзд циклов на основе ОРС

Преобразования «тайлинг», «скашивание» и «метод гиперплоскостей» используют системы PLUTO, PolyMage, Polly-LLVM, MLIR и др. Эти системы используют внутреннее представление полиэдра (polyhedral model), которое ориентировано на анализ и преобразования гнёзд циклов, в то время как внутреннее представление, основанное на AST, имеет больше возможностей для оптимизации программ.

Как отмечалось в параграфе 1.6, ОРС имеет высокоуровневое внутреннее представление, основанное на древовидной структуре [124], типы узлов которой делятся на пять групп: узлы для представления типов данных, идентификаторов, операторов, выражений и служебные узлы.

Особенности преобразований программ в ОРС отмечены в работе [50]. Опишем некоторые преимущества высокоуровневого внутреннего представления для реализации преобразований программ.

2.6.1. Преимущества высокоуровневого внутреннего представления для реализации преобразований программ

При преобразовании циклов возникает необходимость в преобразованиях индексных выражений массивов высокоуровневого языка: «линеаризация выражений» («приведение выражений к стандартному виду для дальнейшей оптимизации»), «вынос инвариантов из цикла», «поиск и вынос одинаковых выражений». Анализ программ для таких преобразований и их применение проще выполнять на высоком уровне. Так же удобнее проводить оптимизации программ для распараллеливания на распределённую память [128], [129].

Из высокого уровня удобно генерировать граф вычислений, который является промежуточным представлением для генерации HDL-описания конвейеров (удобнее, чем из низкоуровневого регистрового конвейера).

Если программа из высокоуровневого внутреннего представления выводится на исходном языке, то она может быть более похожа на исходный код, чем после вывода из низкоуровневого внутреннего представления, поэтому результаты промежуточных преобразований легче читаются при отладке. По этой же причине, если пользователю в диалоге демонстрировать фрагменты его кода, эти фрагменты более узнаваемы. Это позволяет делать диалоговый анализ кода более удобным для разработчика.

В ОРС есть визуализация преобразований и графовых моделей программ, которые облегчают разработку преобразований. Эти визуализации используют преобразованный код программы и должны быть узнаваемы разработчиком.

2.6.2. Распараллеливание на вычислительные системы с распределённой памятью

В случае распределённой памяти узким местом производительности является пересылка данных, поэтому следует проанализировать эффективность преобразованного алгоритма для распараллеливания на такой вычислительной системе. В работе [71] рассматривается комбинация скошенного тайлинга и размещения с перекрытиями для распараллеливания программ на вычислительные системы с распределённой памятью.

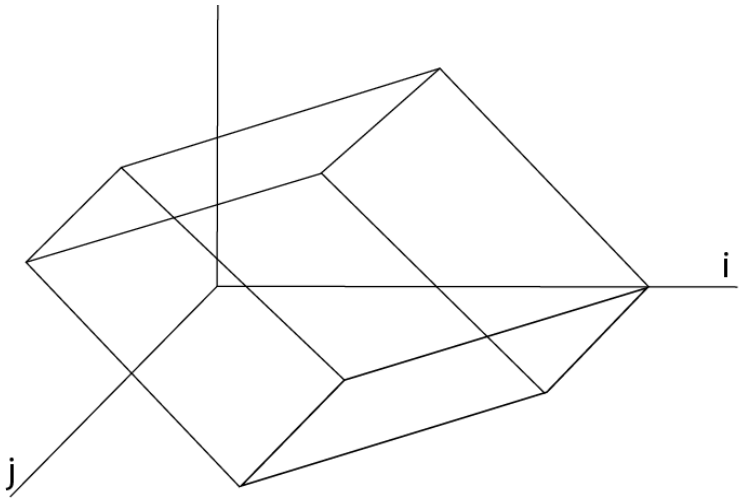

Рисунок 2.10. Вид тайла при разбиении пространства итераций двумерного алгоритма Гаусса – Зейделя для решения задачи Дирихле

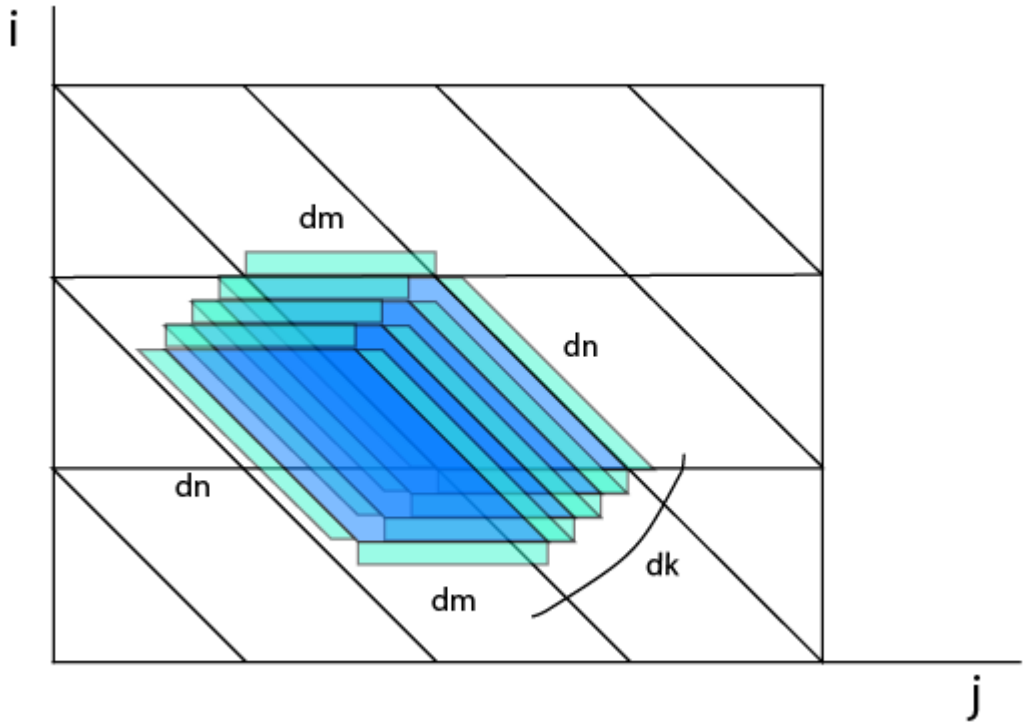

Рисунок 2.11. Проекция тайла на координатную плоскость. Выделены данные, которые нужно подгрузить для вычисления тайла

Приведём пример анализа гнезда циклов из листинга 3.3.1.

Вычислим количество данных, которые требуются для вычисления одного тайла размера [dk, dn, dm], dk – количество итераций. Множество используемых данных определяется как проекция тайла вдоль координатной оси, соответствующей итерациям, на координатную плоскость (i,j). Индексные выражения массива в правой части оператора присваивания показывают, что при вычислении u[i][j] используются соседние элементы массива u: u[i][j-1], u[i][j+1], u[i-1][j], u[i+1][j].

Для внутренних точек проекции эти соседние точки также лежат в проекции. Но для точек u[i][j] границы проекции есть соседние точки за пределами проекции. Такие точки тоже принадлежат множеству данных, которые используются при вычислении тайла.

Количество данных [dk, dn, dm]=dn*dm + dk*(dn + dm - 1) + 2*dn + 2*dm + 2*(dk-1), где данные в тайле: dn*dm + dk*(dn + dm - 1), dk – количество итераций, крайние значения из соседних тайлов: 2*dn + 2*dm + 2*(dk-1).

Количество вычислительных операций на тайл = 4*dn*dm*dk.

Поскольку количество данных, которые нужно загрузить в узел, значительно меньше количества операций с ними, то время, затраченное на загрузку данных, окупится за счёт оптимизации и параллельного выполнения тайлов при не очень маленьких размерах тайла.

Основанное на OPC (оптимизирующей распараллеливающей системе) автоматическое распараллеливание рассматриваемого алгоритма на 8-ядерном процессоре (с общей памятью) даёт 20-кратное ускорение [130]. На основе проведённого анализа следует ожидать, что и распараллеливание на ВС с распределённой памятью тоже даст заметное ускорение.

В статье [131] приводятся утверждения для оценки коммуникационных затрат (числа данных и процессов, вовлечённых в пересылки данных, а также объёма коммуникационных операций вычислительных процессов) при анализе параллельных алгоритмов для распределённой памяти.

В статье [132] описывается метод пространственного распределения данных и вычислений, основывающийся на модели полиэдра, для распараллеливания и улучшения локальности данных при компиляции аффинных программ на системах с распределённой памятью при поддержке MPI. Предлагаемый в статье метод использует идею жадного алгоритма Фотрье [133] [134], но с ослаблением ограничений за счёт сокращения коммуникационных расстояний, при этом межпроцессорные коммуникации не исключаются полностью.

2.7. Выводы к главе 2

Во второй главе приводится разработанный алгоритм оптимизации гнёзд циклов итерационного типа. Представленный алгоритм реализован в рамках оптимизирующей распараллеливающей системы (ОРС). Тем самым частично решается задача 2. Проведена государственная регистрация программы для ЭВМ [125] – программная реализация алгоритма Гаусса – Зейделя, преобразованная с использованием разработанного алгоритма оптимизации, описанного в данной главе. В приложении Е представлен фрагмент преобразованной программы.

В параграфе 2.3 теоретически обосновывается целесообразность применения перестановки циклов внутри тайла.

В параграфе 2.4 обосновывается эквивалентность метода гиперплоскостей с предлагаемым в данной работе вектором нормали (теорема 1).

В параграфе 2.5 приводится обоснование эквивалентности разработанного алгоритма (теорема 2).

В рамках научного исследования приведённые преобразования реализованы в оптимизирующей распараллеливающей системе (ОРС). Отмечены преимущества такой реализации.

Описанный алгоритм может быть расширен в последующих исследованиях за счёт ослабления требований к гнёздам итерационного типа (параграф 1.4) для оптимизации задач, например, описанных в работах [59], [60].

Алгоритм, представленный в главе 2, может использоваться для ускорения алгоритмов типа алгоритма Коши, алгоритмов, основанных на методе Ньютона, алгоритмов, основанных на вычислении свёрток, алгоритма выравнивания последовательностей [135], [136], алгоритма Флойда-Уоршелла и др.

Глава 3. Оценка ускорения алгоритмов итерационного типа

Численные эксперименты были проведены на вычислительной системе (приложение Д) с процессором Intel i7-9700 (Coffee Lake): тактовая частота – 3,00 GHz, 8 ядер, размер кэш-памяти: L1 – 256 Kb; L2 – 2 Mb; L3 – 12 Mb. В качестве компилятора использовался GCC-10.1.0 с опцией компилятора -O3. Ускорение вычислялось по формуле

Ускорение = (Время выполнения исходной программы) / (Время выполнения преобразованной программы)

В таблицах с численными экспериментами приводится среднее время выполнения программ. Замеры проводились несколько раз (5–10) и вычислялось среднее арифметическое, поскольку каждый запуск программы может выполняться на протяжении разного времени из-за многозадачного режима работы ЭВМ. Помимо запускаемых экспериментов выполняются разные системные процессы, в частности работа операционной системы, а место в кэш-памяти может быть занято другими процессами.

Вычислим пиковую производительность процессора по формуле:

$$P = H * c * T,$$

где H – частота процессора, c – количество ядер процессора, T – количество операций с плавающей точкой за такт. T = длина векторного слова / размер числа * количество вычислителей * количество сложений/умножений за инструкцию.

Процессор Intel i7-9700 с архитектурой Coffee Lake поддерживает векторные инструкции: SSE4.1/4.2, AVX2, FMA3. Инструкции FMA3 позволяют выполнять совмещённое умножение-сложение над числами с плавающей точкой. Каждое ядро процессора имеет два вычислителя, выполняющих векторные операции. Длина векторного слова равняется 256 бит. Поддерживаемые типы данных: float (32 бит), double (64 бит).

Основываясь на этих характеристиках, вычислим количество операций с плавающей точкой одинарной (float) и двойной (double) точности за такт:

Одинарная точность: T = 256 бит/32 бит * 2 ФЛОПС/такт * 2 = 32 ФЛОПС/такт.

Двойная точность: T = 256 бит/64 бит * 2 ФЛОПС/такт * 2 = 16 ФЛОПС/такт.

Таким образом, пиковая производительность:

для чисел с плавающей точкой одинарной точности:

$$P = 3 \text{ ГГц} * 8 * 32 \text{ ФЛОПС/такт} = 768 \text{ ГФЛОПС};$$

для чисел с плавающей точкой двойной точности:

$$P = 3 \text{ ГГц} * 8 * 16 \text{ ФЛОПС/такт} = 384 \text{ ГФЛОПС}.$$

Был проведён анализ ассемблерного кода исполняемых файлов программ. В нём используются регистры (XMM) и инструкции SEE3 для операций над одиночными и двойными короткими вещественными значениями (addss, mulss, addsd, mulsd). Из этого можно сделать вывод о том, что векторизация не применяется.

3.1. Про выбор оптимальных размеров тайлов

Рассмотрим выбор оптимальных размеров тайлов для задачи Гаусса – Зейделя для решения двумерной задачи Дирихле уравнения Лапласа (листинг 3.3.1). Размерность задачи – 256 x 4000 x 4000, тип данных – double. Компилятор – GCC, опция компилятора – -O3. Применялся скошенный тайлинг с матрицей скашивания $\text{skew} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$ и перестановка циклов внутри тайла.

При тайлинге ускорение достигается за счёт того, что некоторые данные при вычислении точек тайла читаются много раз, причём все чтения, кроме первого, происходят из кэш-памяти или регистров. Например, в коде листинга (листинг 3.3.1) к ячейке памяти, в которой хранится элемент массива u[3,7], программа обращается при значениях вектора счётчиков циклов (it,i,j) при вычислении не только элемента массива u[3,7], но и элементов u[2,7] u[4,7] u[3,6] u[3,8]. Идея оптимизации вычисления гнёзд циклов итерационного типа состоит в том, чтобы, не завершив вычисления одной итерации it, начинать вычисление следующей итерации (it+1), пока в кэш-памяти находятся необходимые для вычислений данные.

Пусть d1, d2, d3 – размеры тайлов двумерной задачи (трёхмерное гнездо циклов). Пронаблюдаем влияние разных размеров тайлов на ускорение. Зафиксируем d2=50, d1=32(64). Оценим влияние размера d3 на ускорение.

Таблица 3.1. Влияние величины d3 на ускорение алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа. Размерность задачи – 256 x 4000 x 4000, тип данных – double. Компилятор – GCC, опция – -O3. Время выполнения исходного алгоритма – 11,2024 сек. Последовательное выполнение

	d1=32		d1=64	
d3	Время выполнения (сек)	Ускорение	Время выполнения (сек)	Ускорение
4	3,823	2,91	3,643	3,06
8	3,359	3,32	3,257	3,42
16	3,142	3,55	3,028	3,68
20	3,088	3,61	2,990	3,73
25	3,064	3,64	2,951	3,78
40	3,007	3,71	2,880	3,87
50	2,984	3,73	2,869	3,88
80	2,959	3,77	2,815	3,96
100	2,932	3,80	2,813	3,96
125	2,920	3,82	2,825	3,94
200	2,899	3,84	2,755	4,05
250	2,895	3,85	2,754	4,05
400	2,907	3,83	2,754	4,05
500	2,887	3,86	2,731	4,08
1000	2,877	3,87	2,723	4,09
2000	2,876	3,87	2,709	4,11

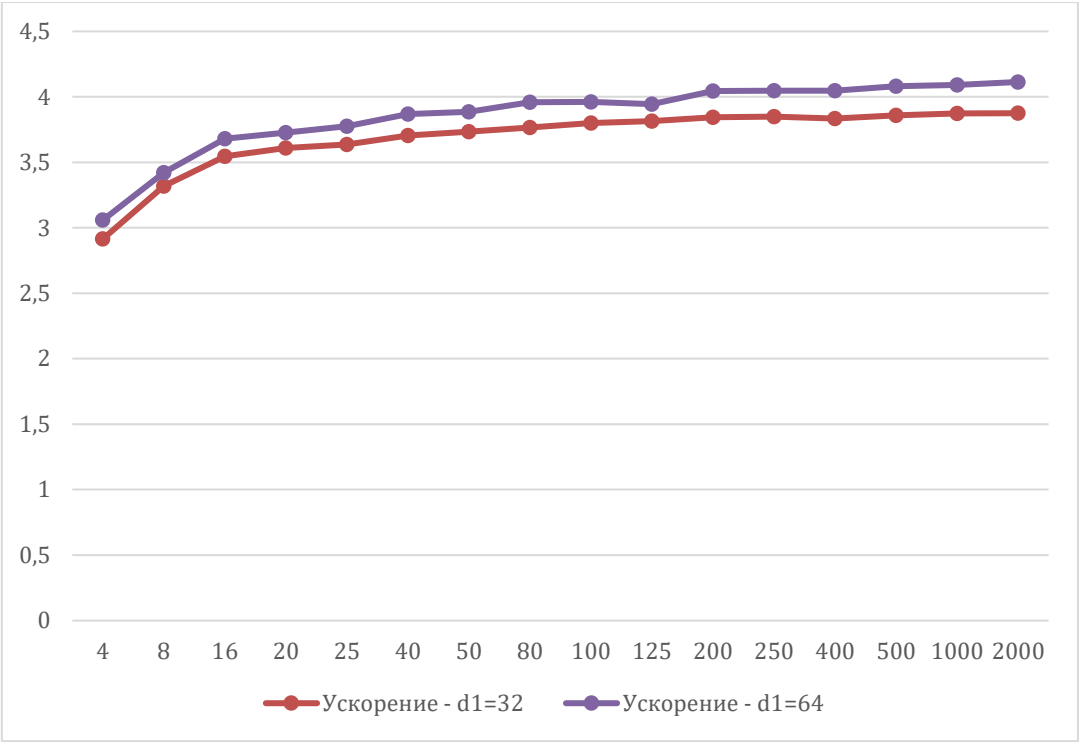

Рисунок 3.1. Сравнение влияния величины d3 на ускорение для значений d1=32 и d1=64. Последовательное выполнение

При последовательном выполнении преобразованной программы наблюдается постепенное увеличение ускорения в пределах 1.1 секунды, а при увеличении значения прирост уменьшается. Это связано с затратами на вычисление первого сечения тайла. Данные для первого сечения подгружаются из оперативной памяти, а для последующих – большая часть из кэш-памяти и регистров (см. таблица 2.3, таблица 2.4); чтобы компенсировать затраты на инициализацию, значение должно быть как можно больше.

Рассмотрим влияние размера d3 при параллельном выполнении на 8 потоках.

Таблица 3.2. Влияние величины d3 на ускорение алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа. Размерность задачи – 256 x 4000 x 4000, тип данных – double. Компилятор – GCC, опция – -O3. Время выполнения исходного алгоритма – 11,2024 сек. Распараллеливание на 8 потоков

	d1=32		d1=64	
d3	Время выполнения (сек)	Ускорение	Время выполнения (сек)	Ускорение
4	1,400	7,96	1,316	8,47
8	1,143	9,75	1,107	10,07
16	1,058	10,53	1,104	10,09
20	1,068	10,43	1,087	10,25
25	1,112	10,02	1,064	10,47
40	1,079	10,33	1,015	10,98
50	1,057	10,55	0,987	11,29
80	1,045	10,67	0,967	11,53
100	1,025	10,87	0,941	11,84
125	1,020	10,92	0,938	11,88
200	0,987	11,29	0,903	12,33
250	0,984	11,32	0,897	12,42
400	1,097	10,16	1,009	11,04
500	0,966	11,53	0,875	12,73
1000	0,940	11,85	0,856	13,01
2000	1,655	6,73	1,522	7,32

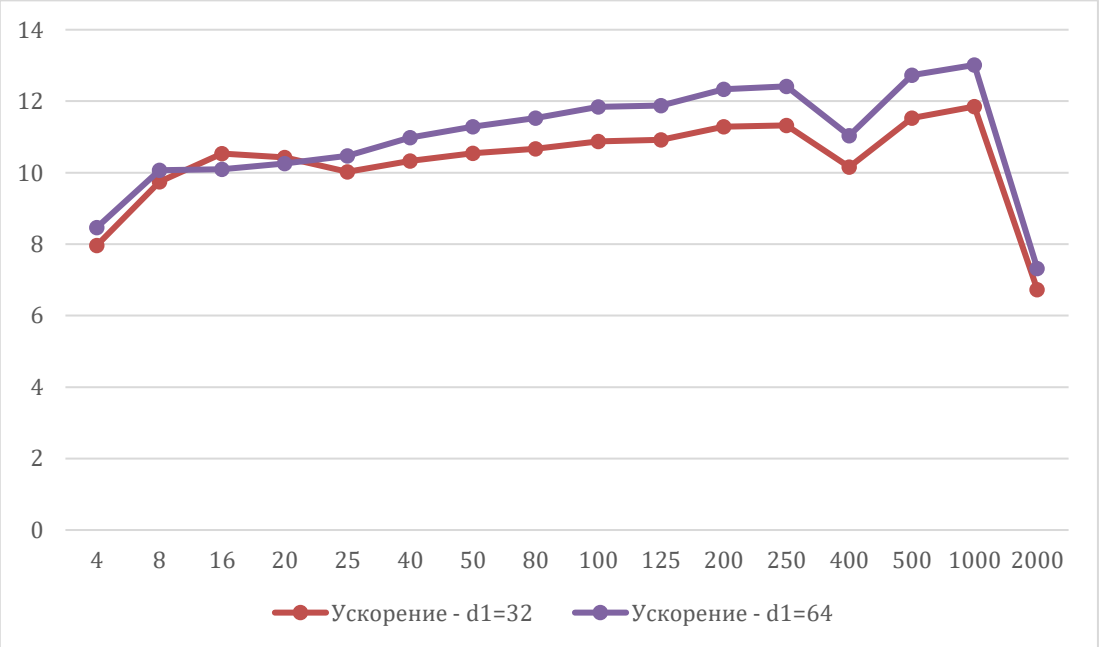

Рисунок 3.2. Сравнение влияния величины d3 на ускорение для значений d1=32 и d1=64. Распараллеливание на 8 потоков

Также рассмотрим параллельное выполнение на 16 потоках.

Таблица 3.3. Влияние величины d3 на ускорение Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа. Размерность задачи – 256 x 4000 x 4000, тип данных – double. Компилятор – GCC, опция – -O3. Время выполнения исходного алгоритма – 11,2024 сек. Распараллеливание на 16 потоков

	d1=32		d1=64	
d3	Время выполнения (сек)	Ускорение	Время выполнения (сек)	Ускорение
4	1,400	7,96	1,223	9,11
8	1,013	11,00	0,947	11,77
16	0,849	13,12	0,799	13,95
20	0,817	13,65	0,758	14,70
25	0,776	14,35	0,722	15,42
40	0,650	17,13	0,605	18,41
50	0,654	17,03	0,617	18,07
80	0,692	16,09	0,664	16,79
100	0,702	15,88	0,663	16,81
125	0,654	17,04	0,639	17,45
200	0,691	16,13	0,672	16,58
250	0,615	18,12	0,605	18,43
400	0,807	13,80	0,751	14,84
500	0,573	19,46	0,558	19,97
1000	0,970	11,48	0,891	12,51
2000	1,727	6,45	1,585	7,03

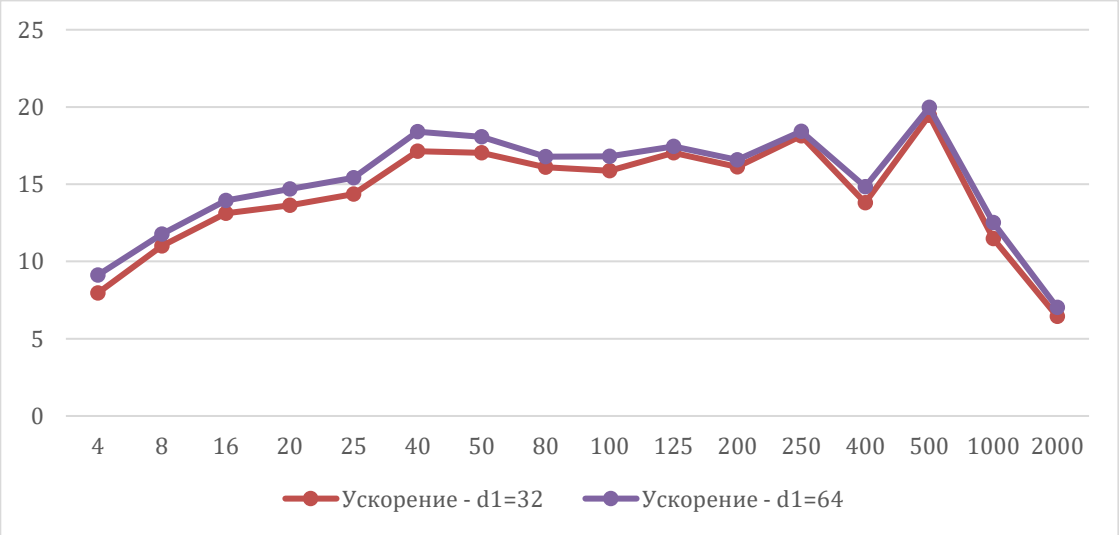

Рисунок 3.3. Сравнение влияния величины d3 на ускорение для значений d1=32 и d1=64. Распараллеливание на 16 потоков

При распараллеливании наблюдается влияние значения d3 на ускорение преобразованной программы. Наблюдается корреляция изменения ускорения для d1=32 и d1=64. Предлагается теоретически оптимальное значение d3, равное d2, что подтверждается численными экспериментами, приведёнными в этой главе.

Сформулируем гипотезу для определения размеров d1 и d2.

Для алгоритма Гаусса – Зейделя двумерной задачи Дирихле уравнения Лапласа тайл представляет собой целые точки параллелепипеда в пространстве Z^3. Рассмотрим множество данных для вычисления этого тайла. Точки тайла будут выполняться быстрее если необходимые для их вычисления данные находятся в кэш-памяти или в регистрах. Это происходит если между повторными использованиями одного данного объём использованных других данных меньше объёма кэш-памяти (не происходит вытеснения, данного из кэш-памяти).

Таким образом, объём данных между двумя использованиями одного и того же данного должен быть не более объёма кэш-памяти. Объём памяти между двумя использованиями одного и того же данного обозначим V. Если мы хотим минимизировать количество промахов к L1-кэш, то должно выполняться неравенство:

$$V \leq |L1| \qquad (1)$$

После перестановки циклов внутри тайла циклы гнезда обхода тайла расположены так, чтобы выполнение тайла проводилось по сечениям, параллельным боковой грани параллелепипеда с длинами рёбер d1 и d2.

В этом случае количество точек трёх соседних сечений (данные которых участвуют при вычислении среднего сечения в алгоритме ГЗ) равно d1*d2+2*(d1+d2), где d1*d2 – данные первого сечения, а каждое следующее сечение будет использовать d1+d2 новых данных. В соответствии с (1) должно выполняться условие

$$\mathrm{V} \leq \mathrm{d1} * \mathrm{d2} + 2 * (\mathrm{d1} + \mathrm{d2}) = (\mathrm{d1} + 2) * (\mathrm{d2} + 2) - 4 \leq |\mathrm{L1}|$$

В граничных точках тайла используются данные, которые либо вычисляются в других тайлах, либо лежат в оперативной памяти. Такие точки вычисляются дольше внутренних точек тайла и их количество желательно минимизировать. Границей сечения является прямоугольник со сторонами d1 и d2. При фиксированной площади прямоугольника длина (количество точек) границы будет минимальна, если стороны этого прямоугольника равны: d1=d2.

$$\mathrm{d1} = \mathrm{d2} \leq \sqrt{|\mathrm{L1}| + 4} - 2$$

Здесь V – объединение целых точек трёх сечений параллелепипеда (тайла).

Для процессора, на котором проводились вычисления, объём L1=32КБ на ядро процессора, тогда количество чисел двойной точности (double, 8 байт), которые могут

находиться в L1, равно 4096. Следовательно, теоретические оптимальные размеры d1 и d2: $d1 = d2 \leq 62$ числа типа $double$.

Сравним полученное значение с результатами численных экспериментов.

Численные эксперименты (таблица 3.7) показали при наиболее близких значениях размеров тайла d1=64, d2=50, d3=50 ускорение 18,57, которое незначительно отличается от лучшего ускорения 18,67 при d1=64, d2=40, d3=40.

Зафиксируем значение d3=500 и будем менять значения d1 и d2. Приведём численные эксперименты для 16 потоков.

Таблица 3.4. Влияние величин d1 и d2 на ускорение Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа. Размерность задачи – 256 x 4000 x 4000, тип данных – double. Компилятор – GCC, опция – -O3. Время выполнения исходного алгоритма – 11,2024 сек. Распараллеливание на 16 потоков. При фиксированном значении d3=500

	d1=32		d1=64		d1=128	
d2	Время выполнения (сек)	Ускорение	Время выполнения (сек)	Ускорение	Время выполнения (сек)	Ускорение
4	1,390	8,02	1,333	8,36	1,307	8,52
8	0,940	11,85	0,898	12,40	0,884	12,60
16	0,676	16,48	0,663	16,80	0,645	17,26
20	0,634	17,57	0,618	18,03	0,599	18,59
25	0,598	18,62	0,590	18,87	0,570	19,54
40	0,577	**19,31**	0,561	19,87	0,531	20,96
50	0,580	19,20	**0,555**	**20,06**	**0,526**	**21,19**
80	0,634	17,58	0,600	18,55	0,583	19,11
100	0,670	16,63	0,639	17,42	0,619	17,99
125	0,711	15,66	0,674	16,53	0,656	16,98
200	0,798	13,96	0,770	14,46	0,773	14,41
250	0,852	13,07	0,864	12,89	0,872	12,77
400	1,159	9,61	1,177	9,46	1,185	9,40
500	1,710	6,51	1,707	6,52	1,697	6,56
1000	4,238	2,63	4,187	2,66	4,155	2,68
2000	12,506	0,89	12,480	0,89	12,449	0,89

В таблице (таблица 3.4) отсутствуют тайлы с размерами d1=d2=62. Возьмём ближайшие размеры тайла, наиболее близкие к теоретическим, d1=64, d2=50. Время выполнения для тайла с такими размерами – 0,555 (ускорение – 20 раз). Полученное ускорение отличается на ~5.6 % от наибольшего достигнутого ускорения при d1=128, d2=50, время выполнения – 0,526 (ускорение – 21,19).

Таким образом подтверждается гипотеза о размерах тайлов d1, d2.

В статье [89] приводится модель выбора оптимальных размеров тайлов, которая используется в оптимизирующей системе PLUTO. Представленная модель размеров тайлов основывается на количестве повторных использований данных по каждому измерению гнезда циклов. Размеры тайлов выбираются пропорционально количеству повторных использований.

3.2. Условия сходимости алгоритмов итерационного типа

Описанный в главе 2 алгоритм оптимизации гнёзд циклов итерационного типа является комбинацией эквивалентных преобразований (параграф 2.7), а значит, если исходная программа, реализующая итерационный алгоритм, сходится, то результирующая программа после применения алгоритма оптимизации тоже будет сходиться, причём к тому же решению.

В данной главе в качестве примера применения алгоритма оптимизации приводится алгоритм Гаусса – Зейделя для разных задач. Для сходимости итерационного процесса алгоритма Гаусса – Зейделя достаточно, чтобы матрица системы уравнения обладала свойством диагонального преобладания [137]. В книге [138] формулируются достаточные условия сходимости блочного метода Гаусса – Зейделя для самосопряжённой и положительно определённой матрицы (при выполнении условия самосопряжённости и положительной определённости оператора системы). В статье [139] приводится теорема о сходимости блочных итерационных алгоритмов Гаусса – Зейделя и Якоби при выполнении блочных условий Адамара либо ослабленных блочных условий Адамара с блочной неприводимостью для входной матрицы.

3.3. Оценка ускорения алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа

Постановка задачи:

Дано уравнение Лапласа $\Delta u=\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)=0$, где Δ – оператор Лапласа. Требуется найти непрерывную функцию $u(x,y)$, удовлетворяющую уравнению Лапласа в области $\Omega=\{(x,y)|0\le x\le a,\ 0\le y\le b\}$.

Согласно алгоритму Гаусса – Зейделя, получим:

$$u_{i,j}^k=(u_{i-1,j}^k+u_{i,j-1}^k+u_{i+1,j}^k+u_{i,j+1}^k)/4$$

Листинг 3.3.1. Основное гнездо циклов программной реализации алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа.

```
for (int it = 0; it < K; it++)
  for (int i = 1; i < N - 1; i++)
```

```
for (int j = 1; j < M - 1; j++)
  u[i][j] = (u[i-1][j] + u[i][j-1] + u[i+1][j] + u[i][j+1])/4;
```

Код гнезда циклов (листинг 3.3.1) после применения алгоритма оптимизации (глава 2) приведён в приложении Е.

Согласно листингам (листинг 3.3.1, приложение Е), во внутреннем гнезде циклов выполняется четыре операции с плавающей точкой (три сложения и одно умножение). Таким образом, количество операций с плавающей точкой для задачи размерности 256 x 4000 x 4000 равняется 256 * 4000 * 4000 * 4 = $16.384*10^9$.

Численные эксперименты:

Проведены численные эксперименты для преобразованной программы, скомпилированной при помощи GCC с оптимизацией компилятора -O3.

Таблица 3.5. Результаты оптимизации алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа. Размерность задачи – 256 x 4000 x 4000, тип данных – float. Компилятор – GCC, опция компилятора – -O3. Время выполнения исходного алгоритма – 10,915 сек. Полная таблица с численными экспериментами приведена в приложении И

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=32, d2=40, d3=40	2,759	3,96	1,058	10,32	0,590	18,51
d1=32, d2=50, d3=50	**2,706**	**4,03**	0,970	11,25	0,565	19,30
d1=32, d2=80, d3=80	3,010	3,63	1,001	10,90	0,620	17,60
d1=64, d2=40, d3=40	2,896	3,77	1,024	10,66	0,578	18,87
d1=64, d2=50, d3=50	2,759	3,96	**0,919**	**11,88**	**0,539**	**20,26**
d1=64, d2=80, d3=80	2,919	3,74	0,956	11,42	0,606	18,02

Рассчитаем долю пиковой производительности для лучшего ускорения (0,5388 секунд).

$$\frac{16.384 * 10^9}{0.5388} \approx 30.41 \text{ ГФЛОПС}$$

Доля пиковой производительности = $\frac{30.41}{768} * 100\% \approx 3.96\%$

Проведены численные эксперименты для преобразованной программы, скомпилированной при помощи ICC с оптимизацией компилятора -O3.

Таблица 3.6. Результаты оптимизации алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа. Размерность задачи – 256 x 4000 x 4000, тип данных – float. Компилятор – ICC, опция компилятора – -O3. Время выполнения исходного алгоритма – 16,835 сек. Полная таблица с численными экспериментами приведена в приложении К

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=32, d2=25, d3=25	2,950	5,71	0,828	20,34	0,677	24,88
d1=32, d2=40, d3=40	2,598	6,48	0,781	21,56	0,547	30,80
d1=32, d2=50, d3=50	2,756	6,11	0,850	19,81	0,600	28,04
d1=64, d2=25, d3=25	2,920	5,76	0,876	19,21	0,625	26,94
d1=64, d2=40, d3=40	**2,537**	**6,64**	**0,777**	**21,66**	**0,535**	**31,48**
d1=64, d2=50, d3=50	2,698	6,24	0,844	19,95	0,571	29,48
d1=128, d2=25, d3=25	2,911	5,78	0,888	18,96	0,631	26,67
d1=128, d2=40, d3=40	2,640	6,38	0,787	21,39	0,544	30,92
d1=128, d2=50, d3=50	2,759	6,10	0,848	19,85	0,597	28,18

После применения предложенного алгоритма для компиляции могут использоваться различные компиляторы. Таблицы выше (таблица 3.5, таблица 3.6) демонстрируют влияние используемого компилятора на ускорение. При использовании компилятора ICC (версии 2021.01) исходная программа выполняется медленнее, чем при использовании компилятора GCC (версии 6.3.0-1).

Проведены численные эксперименты для исходного типа данных double.

Таблица 3.7. Результаты оптимизации алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа. Размерность задачи – 256 x 4000 x 4000, тип данных – double. Компилятор – GCC, опция компилятора – -O3. Время выполнения исходного алгоритма – 11,2024 сек. Полная таблица с численными экспериментами приведена в приложении Л

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=32, d2=25, d3=25	3,605	3,11	1,192	9,39	0,752	14,90
d1=32, d2=40, d3=40	3,075	3,64	1,060	10,57	0,649	17,26
d1=32, d2=50, d3=50	2,982	3,76	1,036	10,81	0,651	17,21
d1=32, d2=80, d3=80	2,954	3,79	1,015	11,03	0,677	16,55
d1=32, d2=100, d3=100	3,088	3,63	1,031	10,87	0,687	16,31
d1=64, d2=20, d3=20	3,373	3,32	1,172	9,56	0,707	15,84
d1=64, d2=25, d3=25	3,200	3,50	1,109	10,10	0,688	16,28
d1=64, d2=40, d3=40	2,852	3,93	0,996	11,25	0,600	18,67
d1=64, d2=50, d3=50	2,884	3,88	0,969	11,56	0,603	18,57
d1=64, d2=80, d3=80	2,914	3,84	0,972	11,53	0,652	17,19
d1=64, d2=100, d3=100	3,038	3,688	0,987	11,35	0,662	16,92

Рассчитаем долю пиковой производительности для лучшего ускорения (0,6 секунд).

$$\frac{16.384 * 10^9}{0.6} \approx 27.31 \text{ ГФЛОПС}$$

Доля пиковой производительности = $\frac{27.31}{384} * 100\% \approx 7.11\%$

3.4. Оценка ускорения алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Пуассона

Постановка задачи:

Дано уравнение $\Delta u = f(x, y)$, где Δ – оператор Лапласа, функция f определена на квадрате $D = \{x, y \mid 0 > x > 1, 0 > y > 1\}$, ∂D – граница квадрата D, $u(x, y) = g(x,)$ при $x, y \in \partial D$, функции f и g заданы. Требуется найти функцию $u(x, y)$.

Согласно алгоритму Гаусса – Зейделя, получим:

$$u_{i,j}^k = (u_{i-1,j}^k + u_{i,j-1}^k + u_{i+1,j}^k + u_{i,j+1}^k - h^2 f_{i,j})/4$$

Листинг 3.4.1. Основное гнездо циклов программной реализации алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Пуассона.

```
for (int it = 0; it < K; it++) {
  for (int I = 1; I < N – 1; i++) {
    for (int j = 1; j < M – 1; j++) {
     u[i][j] = 0.25 * (u[i-1][j] + u[i+1][j]  + u[i][j-1] + u[i][j+1]
– h * h * f[i][j]); }
  }
}
```

Статья [103] описывает реализацию задачи Дирихле уравнения Пуассона на языке T++ с использованием гибридного параллелизма, объединяющего функциональную модель вычислений OpenTS и традиционный подход с обменом сообщениями.

Численные эксперименты:

Таблица 3.8. Результаты оптимизации алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Пуассона. Размерность задачи – 256 x 4000 x 4000. Время выполнения исходного алгоритма – 14,7464 сек. Полная таблица с численными экспериментами приведена в приложении М

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=128, d2=20, d3=20	4,701	3,14	1,576	9,36	0,915	16,11
d1=128, d2=25, d3=25	4,513	3,27	1,486	9,92	**0,864**	**17,08**
d1=128, d2=40, d3=40	4,420	3,34	1,441	10,23	0,874	16,88
d1=128, d2=80, d3=80	4,396	3,35	1,385	10,65	0,887	16,62
d1=128, d2=100, d3=100	4,340	3,40	**1,353**	**10,90**	0,876	16,83
d1=128, d2=125, d3=125	**4,323**	**3,41**	1,383	10,66	0,896	16,46
d1=128, d2=200, d3=200	4,566	3,23	1,540	9,58	1,101	13,40

3.5. Оценка ускорения алгоритма решения задачи теплопроводности

Постановка задачи:

Рассмотрим двумерную однородную задачу теплопроводности.

$\frac{\partial u}{\partial t} = a^2\Delta = a^2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$, где a – положительная константа (a^2 – коэффициент температуропроводности), искомая функция u (x, y) задаёт температуру в точке (x, y) в момент времени t.

Листинг 3.5.1. Основное гнездо циклов программной реализации алгоритма решения задачи теплопроводности.

```
for (t = 0; t < T*2; t++) {
  for (i = 1; i < N+1; i++) {
    for (j = 1; j < N+1; j++) {
      A[(t+1)%2][i][j]= 0.125 * (A[(t)%2][i+1][j] - 2.0 *
A[(t)%2][i][j] + A[(t)%2][i-1][j]) + 0.125 * (A[(t)%2][i][j+1] - 2.0
* A[(t)%2][i][j] + A[(t)%2][i][j-1]) + A[(t)%2][i][j];
    }
  }
}
```

Численные эксперименты:

Таблица 3.9. Результаты оптимизации алгоритма решения задачи теплопроводности. Размерность задачи – 128 x 4000 x 4000. Время выполнения исходного алгоритма – 4,3616. Тип данных – float. Полная таблица с численными экспериментами приведена в приложении Н

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=64, d2=8, d3=8	9,925	0,439	3,315	1,316	1,838	2,373
d1=64, d2=20, d3=20	9,076	0,481	2,879	1,515	1,658	2,630
d1=64, d2=25, d3=25	**9,018**	**0,484**	**2,805**	**1,555**	**1,631**	**2,674**
d1=64, d2=40, d3=40	9,328	0,468	2,845	1,533	1,736	2,512
d1=64, d2=50, d3=50	9,188	0,475	2,813	1,550	1,761	2,476

Таблица 3.10. Результаты оптимизации алгоритма решения задачи теплопроводности. Размерность задачи – 128 x 4000 x 4000. Время выполнения исходного алгоритма – 5,5922. Двойная точность. Полная таблица с численными экспериментами приведена в приложении О

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=64, d2=20, d3=20	7,000	0,799	2,476	2,259	1,847	3,028
d1=64, d2=25, d3=25	6,754	0,828	2,281	2,452	1,613	3,468
d1=64, d2=40, d3=40	6,653	0,841	2,132	2,623	1,426	3,923
d1=64, d2=50, d3=50	6,508	0,859	2,084	2,683	**1,415**	**3,952**
d1=64, d2=80, d3=80	6,676	0,838	2,135	2,619	1,479	3,781

3.6. Влияние перестановки циклов внутри тайла на ускорение алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле

Рассмотрим влияние перестановки циклов внутри тайла (параграф 2.3) на примере оптимизации алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле.

Листинг 3.6.1. Основное гнездо циклов обобщённого алгоритма Гаусса – Зейделя для задачи Дирихле уравнения Лапласа.

```
for (t=0; t<=T-1; t++)  {
        for (i=1; i<=N-2; i++)  {
            for (j=1; j<=N-2; j++)  {
     u[i][j]  =  (A[i][j]*u[i  -  1][j]  +  B[i][j]*u[i  +  1][j]  +
C[i][j]*u[i][j - 1] + D[i][j]*u[i][j + 1] + y0[i][j]) / 5.0;
            }
        }
    }
```

Лучшее ускорение (в 7,87 раз) для программы (листинг 3.6.1), оптимизированной без использования перестановки внутри тайла, достигается на размерах тайла d1=64, d2=20, d3=20 (таблица 3.11). Для программы, оптимизированной с использованием перестановки, наибольшее ускорение (в 16,33 раз) достигается для тайлов размера d1=64, d2=50, d3=50 (таблица 3.12). Таким образом, перестановка циклов, описанная в параграфе 2.3, даёт ускорение в 2,1 раза больше.

Таблица 3.11. Результаты оптимизации алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле без использования перестановки циклов внутри тайла. Размерность задачи – 256 x 2000 x 2000. Время выполнения исходного алгоритма – 7,0792. Полная таблица с численными экспериментами приведена в приложении П

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время	Ускорение	Время	Ускорение	Время	Ускорение
d1=8, d2=8, d3=8	6,766	1,05	1,488	4,76	1,715	4,13
d1=8, d2=16, d3=16	5,494	1,29	1,199	5,91	1,267	5,59
d1=32, d2=8, d3=8	5,996	1,18	1,291	5,49	1,258	5,63
d1=32, d2=16, d3=16	**4,834**	**1,46**	1,029	6,88	0,978	7,24
d1=32, d2=20, d3=20	4,946	1,43	**0,994**	**7,12**	**0,949**	**7,46**
d1=32, d2=25, d3=25	5,221	1,36	1,129	6,27	1,086	6,52
d1=32, d2=40, d3=40	5,580	1,27	1,079	6,56	1,043	6,79
d1=32, d2=50, d3=50	5,622	1,26	1,092	6,48	1,051	6,74
d1=32, d2=80, d3=80	5,938	1,19	1,219	5,81	1,172	6,04
d1=64, d2=8, d3=8	5,765	1,23	1,243	5,70	1,201	5,90
d1=64, d2=16, d3=16	**4,699**	**1,51**	0,972	7,28	0,927	7,64
d1=64, d2=20, d3=20	4,940	1,43	**0,949**	**7,46**	**0,900**	**7,87**
d1=64, d2=25, d3=25	5,150	1,37	1,037	6,83	0,992	7,14
d1=64, d2=40, d3=40	5,511	1,28	1,051	6,73	1,004	7,05
d1=64, d2=50, d3=50	5,558	1,27	1,062	6,66	1,012	6,99
d1=64, d2=80, d3=80	5,869	1,21	1,189	5,96	1,120	6,32

Таблица 3.12. Результаты оптимизации алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле без использования циклов внутри тайла. Размерность задачи – 256 x 2000 x 2000. Время выполнения исходного алгоритма – 7,0792. Полная таблица с численными экспериментами приведена в приложении Р

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=8, d2=8, d3=8	3,844	1,84	1,574	4,50	1,384	5,11
d1=8, d2=16, d3=16	3,040	2,33	1,190	5,95	0,920	7,70
d1=32, d2=8, d3=8	3,257	2,17	1,109	6,38	0,718	9,87
d1=32, d2=16, d3=16	2,533	2,79	0,931	7,61	0,554	12,77
d1=32, d2=20, d3=20	2,368	2,99	0,879	8,05	0,510	13,89
d1=32, d2=25, d3=25	2,235	3,17	0,851	8,31	0,479	14,78
d1=32, d2=40, d3=40	2,126	3,33	**0,773**	**9,16**	0,457	15,48
d1=32, d2=50, d3=50	**2,101**	**3,37**	0,779	9,09	**0,448**	**15,80**
d1=32, d2=80, d3=80	2,268	3,12	0,857	8,26	0,525	13,47
d1=64, d2=8, d3=8	3,164	2,24	1,243	5,70	0,665	10,64
d1=64, d2=16, d3=16	2,463	2,87	0,961	7,37	0,502	14,11
d1=64, d2=20, d3=20	2,316	3,06	0,883	8,02	0,463	15,30
d1=64, d2=25, d3=25	2,240	3,16	0,818	8,65	0,449	15,75
d1=64, d2=40, d3=40	2,130	3,32	**0,745**	**9,50**	0,453	15,63
d1=64, d2=50, d3=50	**2,096**	**3,38**	0,754	9,39	**0,434**	**16,33**
d1=64, d2=80, d3=80	2,275	3,11	0,836	8,47	0,499	14,18

3.7. Сравнение с оптимизирующей распараллеливающей системой PLUTO

Проведено сравнение времени выполнения программ, полученных путём преобразования алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле (листинг 3.6.1) с использованием алгоритма, описанного в главе 2 и реализованного в OPS, и использования системы PLUTO. Результаты численных экспериментов представлены ниже (таблица *3.13*).

Таблица 3.13. Результаты численных экспериментов для алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле уравнения Лапласа. Размерность задачи – 256 x 2000 x 2000. Время выполнения исходного алгоритма – 7,0792

	OPC				PLUTO			
Число потоков	Последовательно		16 потоков		Последовательно		16 потоков	
Размеры блоков	Время (сек)	Ускорение	Время (сек)	Ускорение	Время (сек)	Ускорение	Время (сек)	Ускорение
d1=8, d2=8, d3=8	3,844	1,84	1,384	5,11	**4,176**	**1,70**	**0,624**	**11,34**
d1=8, d2=16, d3=16	3,040	2,33	0,920	7,70	4,212	1,68	0,648	10,93
d1=32, d2=8, d3=8	3,257	2,17	0,718	9,87	4,186	1,69	0,659	10,75
d1=32, d2=16, d3=16	2,533	2,79	0,554	12,77	4,228	1,67	0,666	10,63
d1=32, d2=20, d3=20	2,368	2,99	0,510	13,89	4,736	1,49	0,750	9,44
d1=32, d2=25, d3=25	2,235	3,17	0,479	14,78	5,194	1,36	0,836	8,47
d1=32, d2=40, d3=40	2,126	3,33	0,457	15,48	5,909	1,20	0,983	7,20
d1=32, d2=50, d3=50	**2,101**	**3,37**	**0,448**	**15,80**	6,084	1,16	1,033	6,86
d1=32, d2=80, d3=80	2,268	3,12	0,525	13,47	6,520	1,09	1,192	5,94
d1=64, d2=8, d3=8	3,164	2,24	0,665	10,64	4,262	1,66	1,252	5,65
d1=64, d2=16, d3=16	2,463	2,87	0,502	14,11	4,256	1,66	1,256	5,64
d1=64, d2=20, d3=20	2,316	3,06	0,463	15,30	4,765	1,49	1,394	5,08
d1=64, d2=25, d3=25	2,240	3,16	0,449	15,75	5,181	1,37	1,528	4,63
d1=64, d2=40, d3=40	2,130	3,32	0,453	15,63	5,889	1,20	1,721	4,11
d1=64, d2=50, d3=50	**2,096**	**3,38**	**0,434**	**16,33**	6,074	1,17	1,833	3,86
d1=64, d2=80, d3=80	2,275	3,11	0,499	14,18	6,526	1,08	2,043	3,47

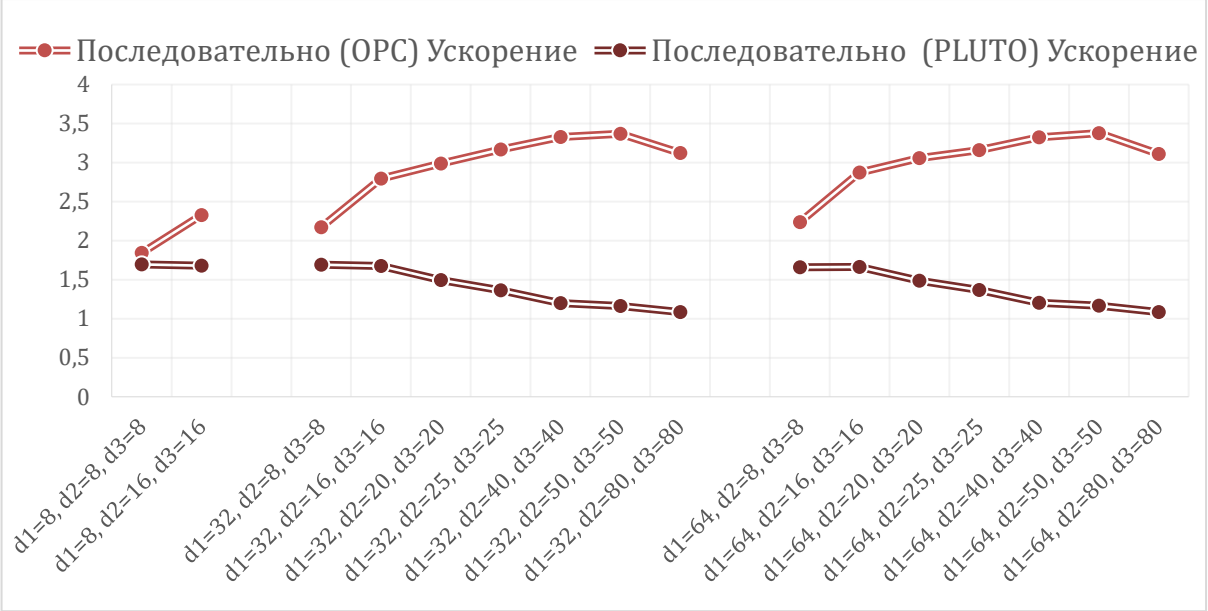

Рисунок 3.4. Сравнение ускорений, полученных при помощи PLUTO и разработанного алгоритма оптимизации итерационных гнёзд циклов (в OPC) для алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле, при последовательном выполнении

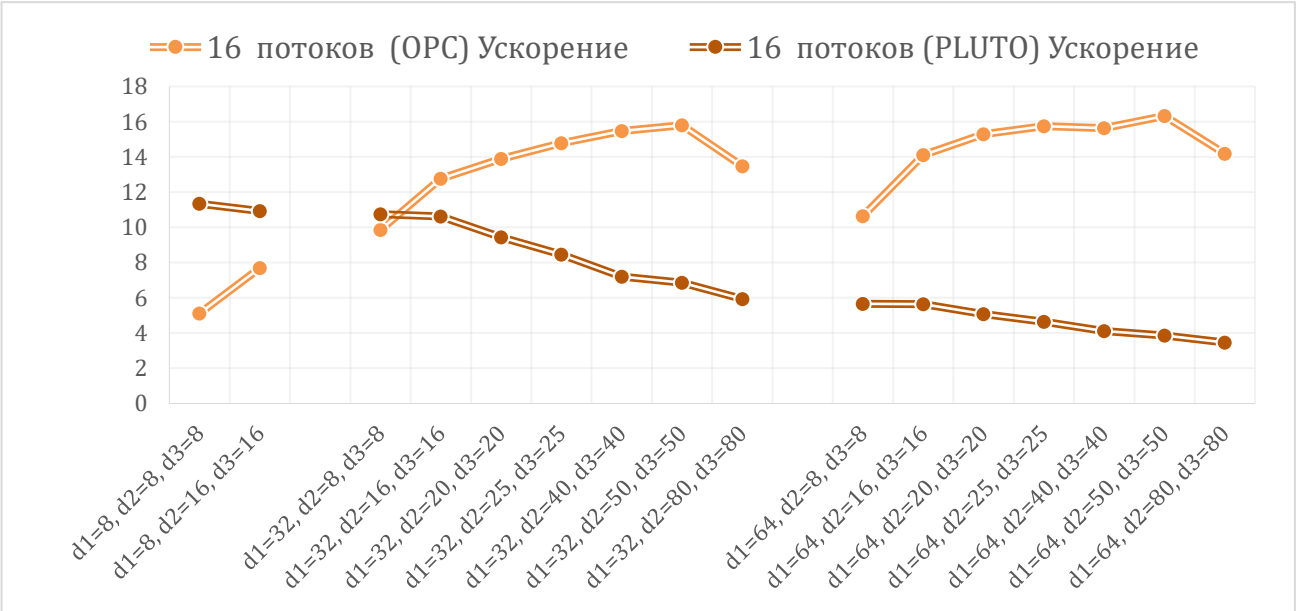

Рисунок 3.5. Сравнение ускорений, полученных при помощи PLUTO и разработанного алгоритма оптимизации итерационных гнёзд циклов (в OPC) для алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле, при выполнении на 16 потоках

Наибольшее ускорение (в 16,32 раза) гнезда циклов, описанного в листинге, достигается алгоритмом, реализованным в OPC, с размерами блоков 64 x 50 x 50, в то время как программа (приложение 3) преобразованная, используя PLUTO, показывает своё лучшее ускорение (в 11,34 раза) на блоках меньшей размерности (8 x 8 x 8). Эта разница обусловлена тем, что

разработанный в ОРС алгоритм применяет перестановку циклов внутри тайлов (параграф 2.2), что повышает временную локальность данных внутри блока (тайла). Код обхода итераций внутри тайла представлен в приложении Е.

В работе [130] приведён код (более 50 строчек) преобразованного необобщённого алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа. Также в работе [130] приведён пример применения линеаризации для упрощения выражений к рассматриваемому алгоритму.

3.8. Об оптимальной цепочке преобразований

Поиск оптимальной цепочки преобразований для минимизации времени выполнения программ является актуальной задачей, поскольку современные оптимизирующие компиляторы не гарантируют её построения [1].

В работе [140] рассматривается алгоритм выбора наилучшей комбинации для минимизации времени исполнения программы из ограниченного списка преобразований: «вынос инвариантного выражения из цикла», «подстановка вперёд», «понижение силы операций» и «удаление общих подвыражений». Алгоритм использует бинарное дерево поиска для получения оптимальной комбинации преобразований. Этот алгоритм может быть обобщён и на другие известные оптимизирующие преобразования программ – например, на преобразования, описанные в главе 1 данной диссертации. Следует отметить, что с увеличением количества рассматриваемых преобразований время поиска цепочки растёт экспоненциально.

В работе [130] рассматривается предварительный компилятор, основанный на ОРС, ориентированный на некоторый класс итерационных алгоритмов. Результатом работы предкомпилятора является оптимизированная программа на исходном высокоуровневом языке. Далее программа передаётся любому компилятору для получения исполняемого файла. Для оптимизации программ предкомпилятор использует цепочку преобразований: «линеаризация выражений», «раскрутка циклов», «канонизация циклов», «вынос общих инвариантных выражений».

В главе 2 описан основной алгоритм, являющийся цепочкой преобразований:

1) «скашивание циклов»,
2) «тайлинг»,
3) «перестановка циклов внутри тайла»,
4) «метод гиперплоскостей».

Для дополнительной оптимизации программ после применения описанного алгоритма можно использовать предкомпилятор, в частности преобразования «линеаризация» и «вынос инвариантов».

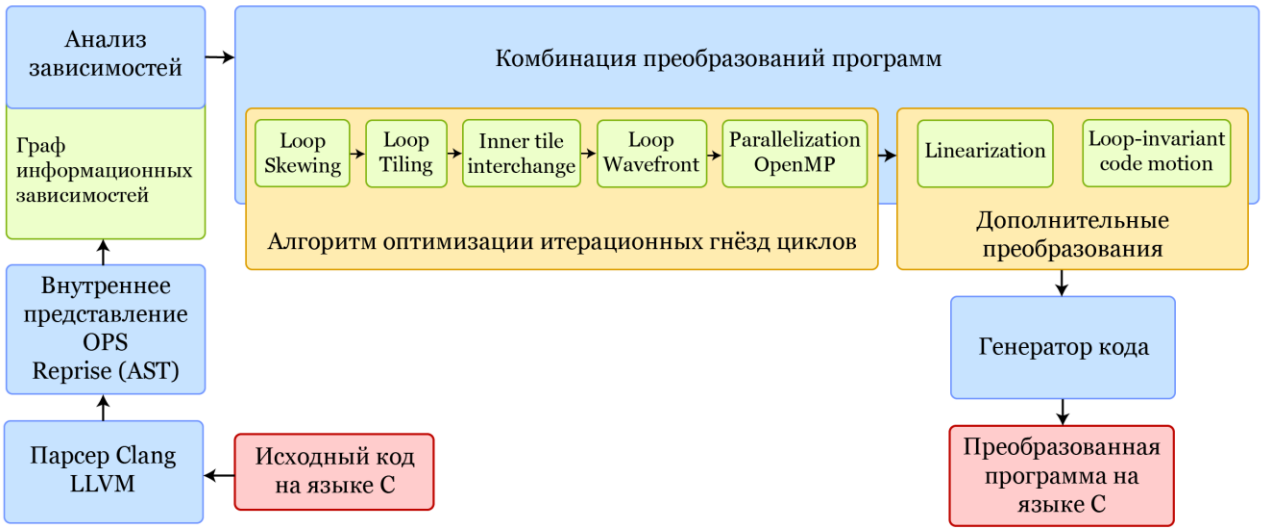

Рисунок 3.6. Схематическое представление предкомпилятора для оптимизации гнёзд циклов итерационного типа

Таблицы ниже демонстрируют влияние дополнительных преобразований «линеаризация» (Linearization) и «вынос общих инвариантных выражений» (Loop-invariant code motion) на примере гнезда циклов (листинг 3.8.1).

Листинг 3.8.1.

```
for (int k = 0; k < K; ++k )
     for (int i = 1; i < N - 1; ++i )
          for (int j = 1; j < M - 1; ++j )
               u[i][j] = (u[i-1][j] + u[i+1][j] + u[i][j-1] +
                         u[i][j+1] + u[i-1][j-1] + u[i+1][j-1] +
                         u[i+1][j-1] + u[i+1][j+1]) / 8.0;
```

Таблица 3.14. Результаты оптимизации программы (листинг 3.8.1). Размерность задачи – 256 x 4000 x 4000 при помощи алгоритма оптимизации итерационных гнёзд циклов, без применения преобразований «линеаризация» и «вынос общих инвариантных выражений». Время выполнения исходного алгоритма – 25,24. Полная таблица с численными экспериментами приведена в приложении С

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время	Ускорение	Время	Ускорение	Время	Ускорение
d1=8, d2=8, d3=8	17,716	1,42	5,714	4,42	3,841	6,57
d1=8, d2=16, d3=16	18,660	1,35	6,446	3,92	3,511	7,19
d1=8, d2=20, d3=20	20,291	1,24	7,006	3,60	3,680	6,86
d1=8, d2=25, d3=25	21,470	1,18	7,209	3,50	3,790	6,66
d1=8, d2=40, d3=40	23,061	1,09	7,401	3,41	4,039	6,25
d1=16, d2=8, d3=8	17,177	1,47	5,874	4,30	3,305	7,64
d1=16, d2=16, d3=16	18,018	1,40	5,946	4,25	3,208	7,87
d1=16, d2=20, d3=20	19,940	1,27	6,315	4,00	3,423	7,37
d1=16, d2=25, d3=25	20,879	1,21	6,580	3,84	3,566	7,08
d1=16, d2=40, d3=40	22,620	1,12	6,942	3,64	3,806	6,63
d1=32, d2=8, d3=8	**16,926**	**1,49**	5,661	4,46	3,066	8,23
d1=32, d2=16, d3=16	17,738	1,42	5,543	4,55	3,030	8,33
d1=32, d2=20, d3=20	19,356	1,30	5,944	4,25	3,235	7,80
d1=32, d2=25, d3=25	20,581	1,23	6,378	3,96	3,394	7,44
d1=32, d2=40, d3=40	22,359	1,13	6,723	3,75	3,698	6,83
d1=64, d2=8, d3=8	16,958	1,49	**5,340**	**4,73**	2,941	8,58
d1=64, d2=16, d3=16	17,731	1,42	5,342	4,73	**2,931**	**8,61**
d1=64, d2=20, d3=20	19,321	1,31	5,776	4,37	3,182	7,93
d1=64, d2=25, d3=25	20,508	1,23	6,107	4,13	3,330	7,58

Таблица 3.15. Результаты оптимизации программы (листинг 3.8.1). Размерность задачи – 256 x 4000 x 4000 при помощи алгоритма оптимизации итерационных гнёзд циклов, с использованием преобразований «линеаризация» и «вынос общих инвариантных выражений». Время выполнения исходного алгоритма – 25,24. Полная таблица с численными экспериментами приведена в приложении Т

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время	Ускорение	Время	Ускорение	Время	Ускорение
d1=8, d2=8, d3=8	12,714	1,99	4,174	6,05	3,107	8,12
d1=8, d2=16, d3=16	11,773	2,14	3,681	6,86	2,404	10,50
d1=8, d2=20, d3=20	11,798	2,14	3,893	6,48	2,307	10,94
d1=8, d2=25, d3=25	11,874	2,13	4,227	5,97	2,251	11,21
d1=8, d2=40, d3=40	13,032	1,94	4,477	5,64	2,375	10,63
d1=16, d2=8, d3=8	12,119	2,08	3,805	6,63	2,552	9,89
d1=16, d2=16, d3=16	11,123	2,27	3,871	6,52	2,122	11,89
d1=16, d2=20, d3=20	11,016	2,29	3,789	6,66	2,054	12,29
d1=16, d2=25, d3=25	11,265	2,24	3,778	6,68	2,033	12,42
d1=16, d2=40, d3=40	12,566	2,01	4,002	6,31	2,229	11,32
d1=32, d2=8, d3=8	11,923	2,12	4,137	6,10	2,280	11,07
d1=32, d2=16, d3=16	10,931	2,31	3,538	7,13	1,988	12,70
d1=32, d2=20, d3=20	10,777	2,34	3,479	7,25	1,939	13,02
d1=32, d2=25, d3=25	10,996	2,30	3,479	7,26	1,918	13,16
d1=32, d2=40, d3=40	12,295	2,05	3,772	6,69	2,103	12,00
d1=64, d2=8, d3=8	11,860	2,13	3,867	6,53	2,159	11,69
d1=64, d2=16, d3=16	10,840	2,33	3,366	7,50	1,887	13,38
d1=64, d2=20, d3=20	**10,695**	**2,36**	**3,313**	**7,62**	**1,845**	**13,68**
d1=64, d2=25, d3=25	10,872	2,32	3,328	7,58	1,851	13,63

В работе [148] рассматривается дополнение цепочки преобразований (рис 3.6) оптимизация заголовка цикла, оптимизация вычисления указателей массивов на примере программной реализации алгоритма Гаусса-Зейделя решения обобщенной двумерной задачи Дирихле уравнения Пуассона (листинг 3.8.2).

Листинг 3.8.2. Пример решения алгоритмом Гаусса-Зейделя обобщенной задачи Дирихле для уравнения Пуассона

```
for (int k = 0; k < K; ++k)
    for (int i = 1; i < N - 1; ++i)
        for (int j = 1; j < M - 1; ++j)
            u[i][j]  =  A[i][j]*u[i-1][j] + B[i][j]*u[i+1][j] +
                        C[i][j]*u[i][j-1] + D[i][j]*u[i][j+1] +
                            E[i][j];
```

Рассмотрим самый вложенный цикл листинга 3.8.2 после цепочки преобразований ОРС.

Листинг 3.8.3. Самый вложенный цикл алгоритма Гаусса–Зейделя решения обобщенной задачи Дирихле, преобразованной при помощи основных преобразований в ОРС

```
for (uni30i = __uni38__uni30i; uni30i < __uni39__uni30i; uni30i =
uni30i + 1) {
  i = uni30i - k;
  __uni28j = __uni29__uni28j - k;
  j = __uni28j - i;
  u[1+i][1+j] = A[i+1][j+1]*u[(1+i)-1][1+j] +
      B[i+1][j+1]*u[(1+i)+1][1+j] + C[i+1][j+1]*u[1+i][(1+j)-1] +
      D[i+1][j+1]*u[1+i][(1+j)+1] + E[i+1][j+1];
}
```

Заменим в листинге 3.8.3 выражения (1+i) и (1+j) новыми переменными ii и jj соответственно, сократив количество сложений. При этом i заменяется выражением (ii-1) (листинг 3.8.4).

Листинг 3.8.4. Результат замены выражений (1+i) и (1+j) новыми переменными ii и jj соответственно

```
for (uni30i = __uni38__uni30i; uni30i < __uni39__uni30i; uni30i =
uni30i + 1) {
    ii = uni30i – k + 1;
    __uni28j = __uni29__uni28j - k;
    jj = __uni28j – (ii - 1) + 1;
    u[ii][jj] =  A[ii][jj]*u[ii-1][jj] + B[ii][jj]*u[ii+1][jj] +
                 C[ii][jj]*u[ii][jj-1] + D[ii][jj]*u[ii][jj+1] +
                 E[ii][jj];
}
```

Заменим в заголовке цикла счетчик uni30i новым счетчиком ii, уменьшив тело цикла на один оператор присваивания (листинг 3.8.5).

Листинг 3.8.5. Результат подстановки переменной `ii` на место счетчика циклов `uni30i`

```
for (ii = __uni38__uni30i – k + 1; ii < __uni39__uni30i – k + 1; ii =
ii + 1) {
    __uni28j = __uni29__uni28j - k;
    jj = __uni28j – ii +1;
    u[ii][jj] =  A[ii][jj]*u[ii-1][jj] + B[ii][jj]*u[ii+1][jj] +
                 C[ii][jj]*u[ii][jj-1] + D[ii][jj]*u[ii][jj+1] +
                 E[ii][jj];
}
```

Оптимизируем заголовок цикла и вынесем вычисление переменной __uni28j за пределы цикла, поскольку эта переменная не зависит от счетчика цикла (листинг 3.8.6). Уменьшим количество операций при вычислении адресов двумерных массивов, которые являются коэффициентами при вычисляемой переменной u. Выполним оптимизацию вычисления указателей этих массивов. Так как в вычислениях (листинг 3.8.6) участвует 6 массивов одинаковой размерности, можно заранее вычислить положение их указателей. В случае если элементы массива расположены подряд, запись u[ii][jj] эквивалентна *(*u + ii * M + jj), где M— вторая размерность двумерного массива. Вычисление позиции [ii][jj] можно вынести в переменную. Листинг 3.8.7 демонстрирует результат такой оптимизации. Для того чтобы данное преобразование было корректным, требуется разместить данные массивов в памяти подряд. Размещение приводится в листинге 3.8.8.

Листинг 3.8.6. Результат выноса вычисления переменной `__uni28j` за пределы внутреннего цикла и оптимизации заголовка цикла

```
__uni39__uni30i_k1 = __uni39__uni30i – k + 1;
__uni28j = __uni29__uni28j - k;
for (ii = __uni38__uni30i – k + 1; ii < __uni39__uni30i_k1; ii = ii +
1) {
    jj = __uni28j – ii +1;
    u[ii][jj] =  A[ii][jj]*u[ii-1][jj] + B[ii][jj]*u[ii+1][jj] +
                 C[ii][jj]*u[ii][jj-1] + D[ii][jj]*u[ii][jj+1] +
                 E[ii][jj];
}
```

Листинг 3.8.7. Результат оптимизации указателей массивов внутреннего цикла

```
__uni39__uni30i_k1 = __uni39__uni30i – k + 1;
__uni28j = __uni29__uni28j - k;
for (ii = __uni38__uni30i – k + 1; ii < __uni39__uni30i_k1; ii = ii +
1) {
    jj = __uni28j – ii +1;
    int ind = ii * M + jj;
    *(*u+ind) = *(*A+ind)*(*(*u+ind-N))+*(*B+ind)*(*(*u+ind - 1)) +
*(*C+ind)*(*(*u+ind+N))+*(*D+ind)*(*(*u+ind+1))+*(*E+ind);
}
```

Листинг 3.8.8. Размещение данных в памяти для оптимизации вычисления указателей массивов

```
for (int i = 0; i < N; ++i ) {
    for (int j = 0; j < M; ++j ) {
        u[i][j] = (double)rand()/(double)(RAND_MAX);
    }
}

double* data_arr = (double*)malloc(N * M * sizeof(double));
for (size_t i = 0; i < N; i++) {
     for (size_t j = 0; j < M; j++) {
          data_arr[i * M + j] = u[i][j];
     }
}

double** U = (double**)malloc(N * sizeof(double*));
for (size_t i = 0; i < N; i++) {
     U[i] = &data_arr[i * M];
}
```

Приводятся (таблица 3.16) результаты численных экспериментов для значений k=256, N=M=4000 и демонстрируется ускорение для оптимальных размеров блоков (d1=64, d2=50, d3=50), при которых достигается наименьшее время выполнения расчетов после применения базовых преобразований, а также дополнительные преобразования дают ускорение на 6.7% больше относительно алгоритма, преобразованного базовыми преобразованиями, выполненными на 16 потоках. Размерность задачи — 4000×4000, количество итераций алгоритма равно 256, тип данных double.

Таблица 3.16. Влияние дополнительных преобразований на ускорение при разных размерах тайлов

	Основные преобразования		Дополнительные преобразования		
Размеры блоков	Время	Ускорение	Время	Ускорение	Ускорение за счёт дополнительных оптимизаций
d1=16, d2=16, d3=16	4,779	4,294	5,051	4,045	0,946
d1=16, d2=20, d3=20	4,347	4,721	4,229	4,830	1,028
d1=16, d2=25, d3=25	4,116	4,986	3,787	5,394	1,087
d1=16, d2=40, d3=40	4,576	4,485	3,723	5,487	1,229
d1=16, d2=50, d3=50	4,357	4,711	3,513	5,816	1,240
d1=16, d2=80, d3=80	3,756	5,463	3,094	6,603	1,214
d1=16, d2=100, d3=100	3,457	5,936	3,034	6,734	1,139
d1=16, d2=125, d3=125	3,602	5,697	3,721	5,490	0,968
d1=16, d2=200, d3=200	8,718	2,354	10,272	1,989	0,849
d1=32, d2=16, d3=16	4,181	4,908	4,411	4,631	0,948
d1=32, d2=20, d3=20	3,688	5,565	3,807	5,367	0,969
d1=32, d2=25, d3=25	3,350	6,127	3,274	6,240	1,023
d1=32, d2=40, d3=40	3,152	6,511	2,922	6,992	1,079
d1=32, d2=50, d3=50	3,046	6,737	2,794	7,311	1,090
d1=32, d2=80, d3=80	2,754	7,451	2,548	8,018	1,081
d1=32, d2=100, d3=100	2,798	7,334	2,614	7,816	1,071
d1=32, d2=125, d3=125	3,043	6,745	3,272	6,243	0,930
d1=32, d2=200, d3=200	8,647	2,373	10,406	1,963	0,831
d1=64, d2=16, d3=16	3,852	5,328	4,004	5,102	0,962
d1=64, d2=20, d3=20	3,210	6,393	3,380	6,044	0,950
d1=64, d2=25, d3=25	2,915	7,041	2,952	6,919	0,987
d1=64, d2=40, d3=40	2,690	7,628	2,525	8,092	1,066
d1=64, d2=50, d3=50	**2,593**	**7,915**	**2,431**	**8,405**	**1,067**
d1=64, d2=80, d3=80	2,691	7,625	2,482	8,232	1,085
d1=64, d2=100, d3=100	2,624	7,822	2,515	8,123	1,043
d1=64, d2=125, d3=125	2,970	6,910	3,233	6,319	0,919
d1=64, d2=200, d3=200	8,652	2,372	9,932	2,057	0,871

3.9. Выводы к главе 3

В данной главе приводятся результаты численных экспериментов для определения эффективности разработанной цепочки преобразований для разных размеров блоков (тайлов) и получена теоретическая модель вычисления оптимальных размеров тайлов.

В параграфе 3.1 решается задача поиска оптимальных размеров тайлов. Получена формула вычисления оптимальных размеров тайлов через объём кэш-памяти. Оказалось, что данные, используемые для вычисления тайла оптимального размера, не обязаны помещаться в кэш-память.

Эксперименты проводились для компиляторов GCC и ICC. Эксперименты проводились с программными реализациями алгоритмов: Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа (ускорение в 20,26 раз для компилятора GCC; в 31,47 раз для компилятора ICC) и Пуассона (ускорение в 17,1 раз), решения задачи теплопроводности (ускорение в 4,1 раз).

На примере обобщённого алгоритма Гаусса – Зейделя для задачи Дирихле уравнения Лапласа приведено сравнение ускорения программ, преобразованных при помощи разработанного алгоритма (ускорение в 16,32 раза) и оптимизирующей системы PLUTO (ускорение в 11,34 раза). По результатам численных экспериментов программа, преобразованная разработанным алгоритмом, показывает ускорение, превосходящее ускорение, полученное посредством использования оптимизирующей системы PLUTO, в ~1,4 раза. Такое ускорение достигается за счёт повышения временной локальности данных (перестановка циклов внутри тайла).

В параграфе 3.6 приводятся численные эксперименты, иллюстрирующие прирост производительности (в 2,1 раза) после применения перестановки циклов внутри тайла (изменение обхода итераций тайла) относительно преобразованной программы без такой перестановки.

Приводятся численные эксперименты, показывающие прирост производительности «оптимизация заголовка цикла», «оптимизация вычисления указателей массивов» на примере программной реализации алгоритма Гаусса-Зейделя решения обобщенной двумерной задачи Дирихле уравнения Пуассона. Наибольшее полученное ускорение составляет 24%.

Преобразованные программы получены при помощи цепочки автоматизированных преобразований алгоритма (глава 2) в оптимизирующей распараллеливающей системе (ОРС) и показывают ускорение.

Проведённые численные эксперименты подтверждают эффективность предлагаемых методов ускорения программ.

Глава 4. Реализация метода диалогового анализа текстов программ на базе OPC

При автоматическом применении преобразований и распараллеливании программист может столкнуться с проблемами, такими как:

- нарушение синтаксической и семантической корректности программы, связанное с преобразованиями программы;
- проблемы определения информационных зависимостей;
- изменение погрешностей округления для чисел с плавающей точкой.

Данные проблемы можно решить с помощью предварительного статического анализа программы и построенных на его основе вопросов, задаваемых пользователю. Данный подход к анализу и преобразованиям программ будем называть «диалоговый анализатор». Для анализа программ используется символьный анализ и линеаризация выражений. Высокоуровневое внутреннее представление OPC позволяет анализировать код и формировать вопросы в доступном программисту виде.

Возможные использования «диалогового анализатора»:

- при полуавтоматической оптимизации/компиляции;
- при анализе программ на информационные зависимости и возможность оптимизаций – это полезно не только для программистов, занимающихся оптимизирующими компиляторами, но и для разработчиков, которые незнакомы с данной сферой;
- для пользователей, которые вкладывают больше усилий в свои разработки, нежели в изучение парадигм параллельного программирования;
- при анализе старого кода, когда попытки разобраться в программе, а тем более в поиске фрагментов программы на оптимизацию, могут занять много времени.

4.1. Символьный анализ

Для статического анализа программного кода иногда используется построение соответствия между значениями выражений в тексте программы и абстрактными выражениями специальной математической модели. Такой анализ называют символьным [141].

Пусть компилятор в целях уточнения наличия информационной зависимости должен установить отношения между выражениями языка C x * x + y * y и (x + y) * (x - y), где x, y – целочисленные переменные. Символьный анализ ставит этим выражениям в соответствие пару абстрактных выражений xa ^ 2 + ya ^ 2 и (xb + yb) (xb - yb), где x, y принадлежат Z; a, b – контекст употребления переменной. В случае, если контексты a и b совпадают (переменные x и y в первом выражении имеют то же самое значение, что и во втором), то после перемножения

суммы в скобках анализ установит, что абстрактные выражения равны. С учётом определённых допущений это будет означать равенство исходных выражений языка С.

Символьный анализ, предлагаемый в [141], предназначен для распараллеливания и может решать следующие задачи: символьное протягивание констант, обобщённую подстановку индуктивных переменных, подстановку вперёд (в том числе и межпроцедурную), определение инвариантов цикла и статическую профилировку.

В работе [142] описывается алгоритм символьного анализа для диалогового распараллеливания программ на базе ОРС. Приводится демонстрация функционала на примере алгоритма Флойда-Уоршелла: данный алгоритм можно распараллелить в случае, если элементы матрицы смежностей неотрицательны, однако на этапе компиляции известно лишь то, что они целого типа.

Пример 4.1.1. Программная реализация алгоритма Флойда-Уоршелла.

```
int d[N][N];
for (int k = 0; k < N; k++) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (d[i, j] > d[i, k] + d[k, j]) {
                d[i, j] = d[i, k] + d[k, j];
            }
        }
    }
}
```

4.2. Линеаризация выражений

Приведём определение из работы [143].

Линеаризацией выражения относительно набора переменных a_i называется преобразование произвольного выражения к виду:

$e_1*a_1+e_2*a_2+\ldots+e_n*a_n+e_0$, где e_i – выражения, a_i – переменные, входящие в исходное выражение, при этом ни одна из них не входит ни в одно из выражений e_i. Анализом линейности выражения называется анализ возможности осуществления линеаризации. Полученное представление называется линейной формой выражения (линейным разложением). Набор переменных a_i является базой линеаризации, e_i – коэффициентами, при этом e_0 – свободным членом линейного разложения.

В ОРС реализована функция [130], выполняющая линеаризацию выражений, располагая базовые переменные a_i в алфавитном порядке. Базовые переменные могут быть индексными. Линеаризация выражений, используя свойства коммутативности, приводит выражения к виду, в котором слагаемые или сомножители расположены в алфавитном порядке. Линеаризация

выражений проводит преобразования, которые иногда называют приведением подобных. Например:

x+x -> 2*x

b+a -> a+b

Если два выражения эквивалентны как функции, т. е. на одних и тех же входных данных оба выражения вычисляют одинаковые результаты, то из этого не вытекает, что преобразование «линеаризация выражений» приведёт их к одному виду.

Преобразование «линеаризация выражений» применяется к индексным выражениям массивов для определения информационных зависимостей и используется для приведения циклов к виду, допускающему реализацию конвейерным или многоконвейерным вычислителем. Также данное преобразование позволяет подготовить программу к символьному анализу.

4.3. Примеры работы «диалогового анализатора»

Данный параграф содержит примеры использования «диалогового анализатора» для уточнения информационных зависимостей при распараллеливании на архитектуры SIMD (векторизации) и MIMD. Также демонстрируется интерактивный подход к применению преобразований, таких как «гнездование цикла» и «тайлинг».

4.3.1. Уточнение информационных зависимостей для распараллеливания программ

Для уточнения информационных зависимостей перед выполнением преобразования «диалоговый анализатор» выполняет последовательность действий:

- находит переменную, значение которой не определено на этапе компиляции, и создаёт информационную зависимость, мешающую выполнению нужного преобразования;
- берёт индексные выражения пары вхождений, порождающих такую информационную зависимость;
- формирует предикат в символьном виде, при выполнении которого преобразование будет эквивалентным;
- упрощает полученный предикат при помощи линеаризации выражений;
- формирует вопрос пользователю в интерфейсе ОРС.

Обозначим условия, при которых программа после распараллеливания будет эквивалентна исходной.

Последовательные циклы, допускающие замену на векторную команду, называются векторизуемыми, а сама такая замена – векторизацией [110].

Для векторизации требуется, чтобы в теле цикла все операторы были присваиваниями, а также чтобы отсутствовали дуги графа информационных зависимостей, направленные «снизу вверх». Тогда векторизация данного цикла является эквивалентным преобразованием (см. [110], с. 77).

Пример 4.3.1.

```
for (i=1; i < N; i++) {
      a[i] = b[i]+c[i];
      d[i]= a[i+1]+b[i];
}
```

В этом цикле вхождение a[i+1] обращается к ячейке памяти a[2] раньше, чем вхождение a[i], и поэтому дуга информационной зависимости направлена «снизу вверх» от вхождения a[i+1] к a[i].

Для асинхронного выполнения блоков программы (MIMD) требуется отсутствие информационных зависимостей между блоками, в частном случае для асинхронного выполнения итераций цикла – отсутствие циклически порождённых зависимостей.

Пример 4.3.2.

```
int k;
int x[20],A[20];
int main(){
      k = 1;
      for(int i=0; i<10; i++){
            x[i+2] = x[i+k] + A[i];
      }     |       |
}           V2      V1
```

Проанализируем граф информационных зависимостей, построенный по данному циклу с помощью ОРС.

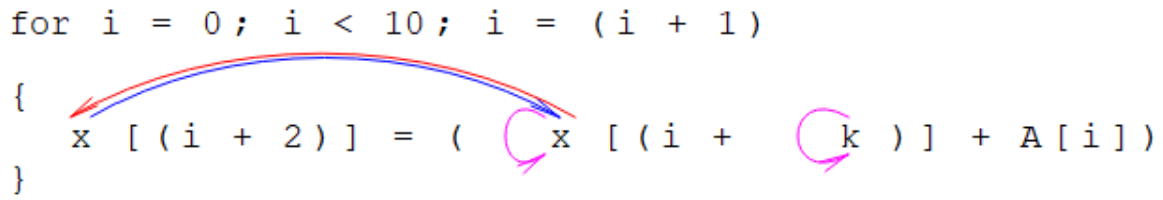

Рисунок 4.1. Граф информационных зависимостей, построенный в ОРС по примеру 4.3.2. Красная дуга – дуга антизависимости (in-out); синяя дуга – дуга потоковой зависимости (out-in); розовая дуга – дуга входной зависимости (in-in)

В данном примере присутствует один программный цикл, в нём два вхождения (V1, V2) массива x, одно вхождение массива A и одна внешняя переменная k.

На графе можно наблюдать несколько дуг между вхождениями массива x.

Рассмотрим, как значения переменной k влияют на информационные зависимости.

При k=2:

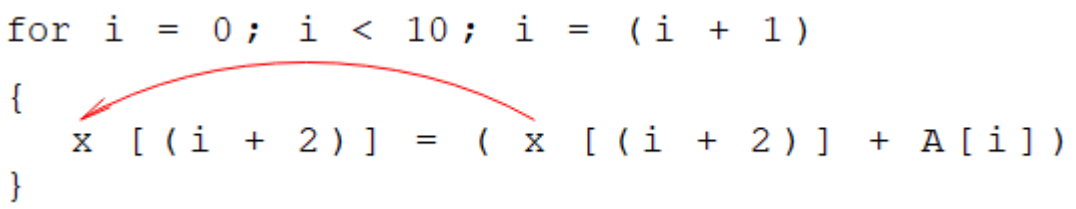

Рисунок 4.2. Граф информационных зависимостей, построенный по примеру 4.3.2, k=2. Красная дуга – дуга антизависимости (in-out)

Начало дуги – использование (in) x[i+2] – чтение из ячейки памяти. Конец дуги – генератор (out) x[i+2] – запись в ячейку памяти. Дуга антизависимости (in-out). Дуга не является циклически порождённой.

При k > 2:

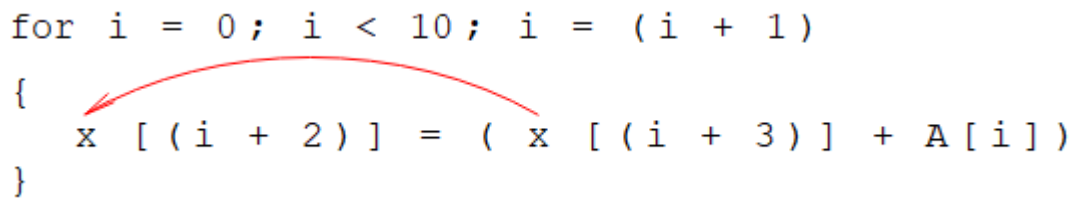

Рисунок 4.3. Граф информационных зависимостей, построенный по примеру 4.3.2, k > 2. Красная дуга – циклически порождённая дуга антизависимости (in-out)

Начало дуги – использование (in) x[i+3] – чтение из ячейки памяти. Конец дуги – генератор (out) x[i+2] – запись в ячейку памяти. Дуга антизависимости (in-out). Дуга является циклически порождённой.

При k < 2:

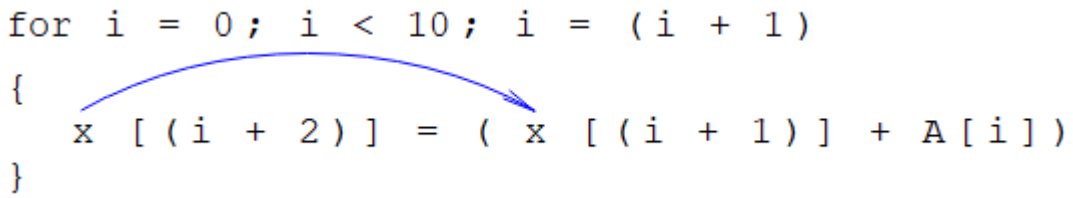

Рисунок 4.4. Граф информационных зависимостей, построенный по примеру 4.3.2, k < 2. Синяя дуга – циклически порождённая дуга потоковой зависимости (out-in)

Начало дуги – генератор (out) x[i+2] – запись в ячейку памяти. Конец дуги – использование (in) x[i+1] – чтение из ячейки памяти. Дуга потоковой зависимости (out-in). Дуга является циклически порождённой.

Таким образом, в зависимости от значения переменной k будут меняться дуги информационных зависимостей на графе.

Будем рассматривать вхождения массива x, потому что внешняя переменная k влияет только на его информационные зависимости.

Распараллеливание на архитектуру MIMD

Для того чтобы отсутствовали циклически порождённые зависимости, нужно, чтобы вхождения не обращались к одной ячейке памяти на разных итерациях, т. е. чтобы индексы разных вхождений массива x были равны.

$i + 2 = i + k$.

Упростив, получим: $k=2$.

Векторизация цикла

Для того чтобы векторизовать цикл, нужно отсутствие дуг «снизу вверх», что выполняется при условии:

$i + k \geq i + 2$.

Упростив, получим: $k \geq 2$.

Автоматическое упрощение предикатов происходит с помощью функции линеаризации выражений, реализованной в ОРС.

Основываясь на полученных предикатах, формируются вопросы, задаваемые диалогом.

Рассмотрим сценарий работы «диалогового анализатора», реализованного в ОРС.

Шаг 1. Приветствие. Информация о «диалоговом анализаторе».

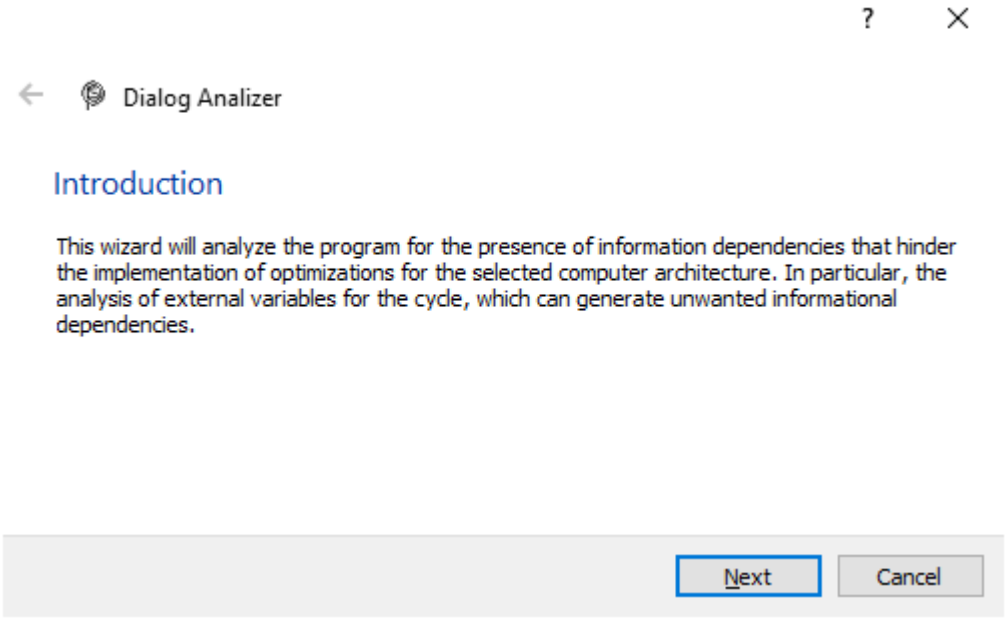

Рисунок 4.5. «Диалоговый анализатор». Шаг 1. Приветствие. Информация о «диалоговом анализаторе»

Шаг 2. Пользователю предоставляется выбор целевой архитектуры, для которой будут проверены условия применимости оптимизаций. В списке представлены архитектуры: SIMD, MIMD и гибридная SIMD+MIMD.

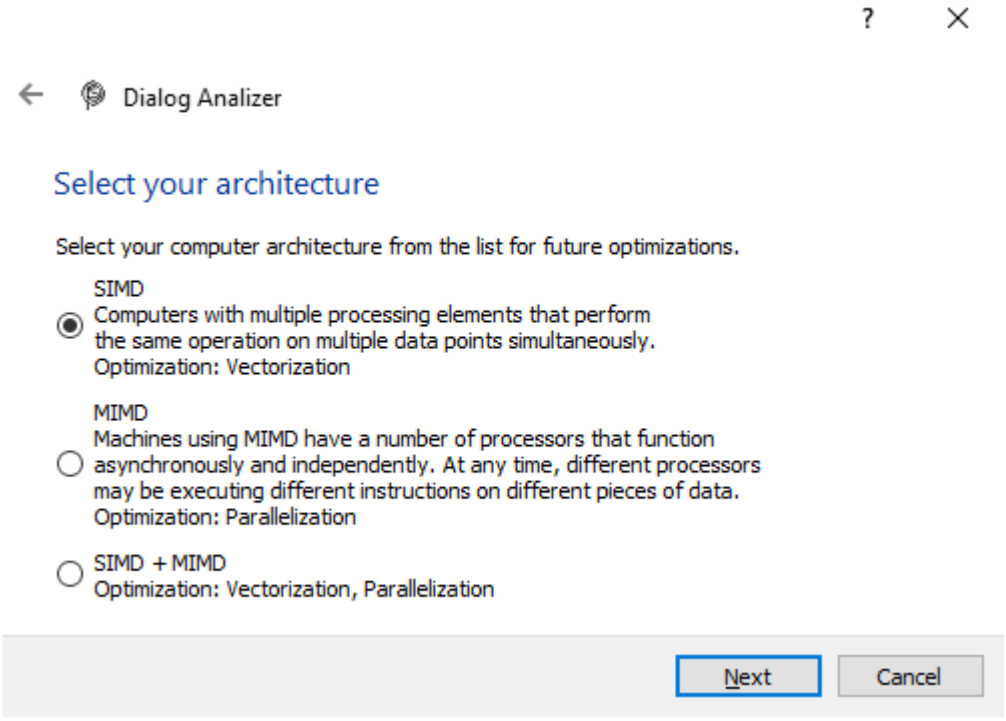

Рисунок 4.6. «Диалоговый анализатор». Шаг 2. Выбор целевой архитектуры. SIMD, MIMD, SIMD+MIMD

Шаг 3. В зависимости от выбранной архитектуры проводится проверка условий применимости оптимизации для указанного фрагмента программы. В случае наличия внешних переменных, влияющих на информационные зависимости, для уточнения их значения строится предикат, при выполнении которого возможно применение оптимизации. Пользователю задаётся вопрос о верности полученного предиката.

Окна анализа цикла из примера 4.3.2 для разных архитектур:

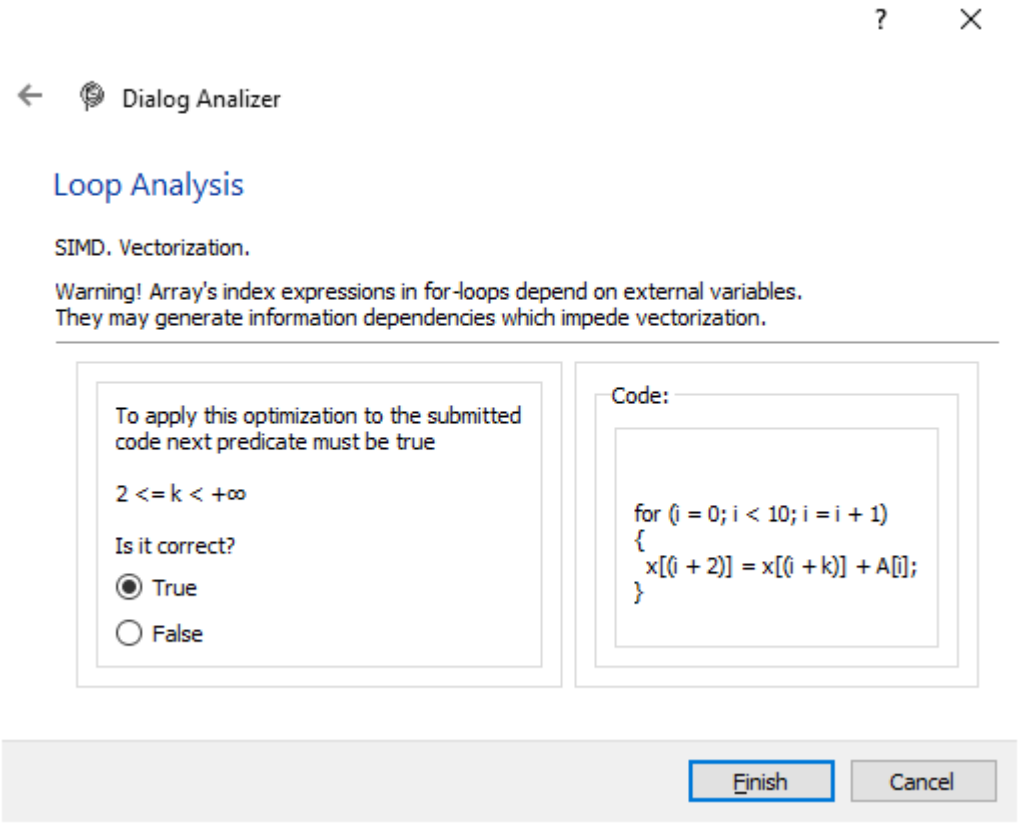

Рисунок 4.7. «Диалоговый анализатор». Шаг 3. Анализ программы для архитектуры SIMD. Пример 4.3.2

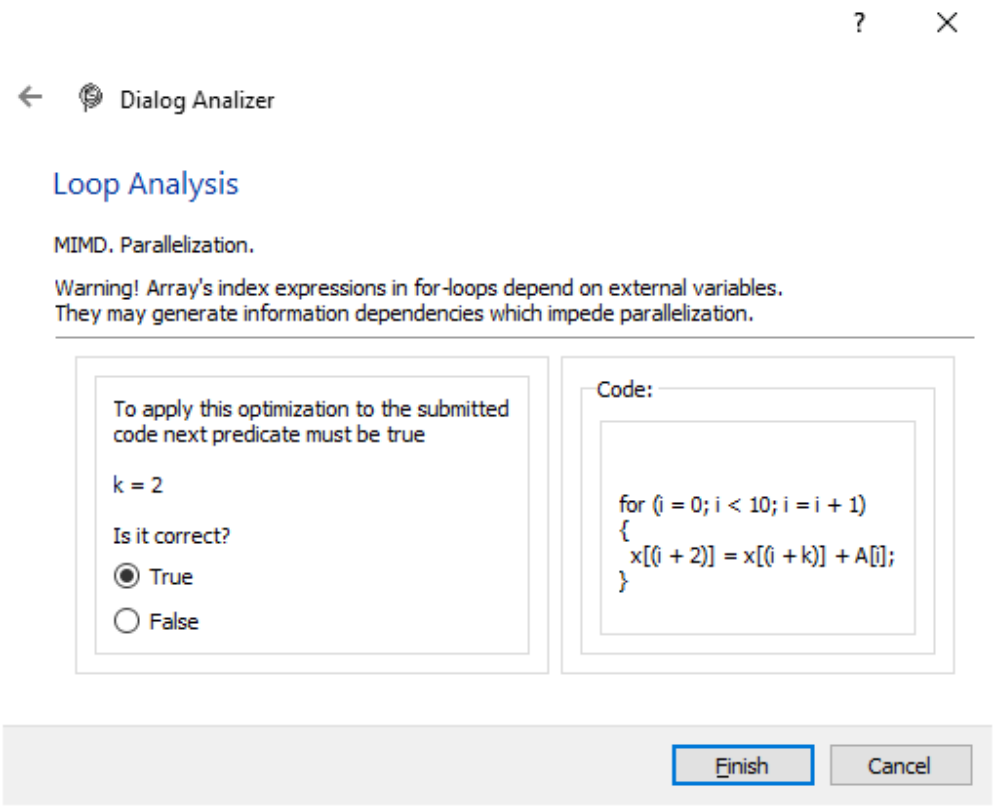

Рисунок 4.8. «Диалоговый анализатор». Шаг 3. Анализ программы для архитектуры MIMD. Пример 4.3.2

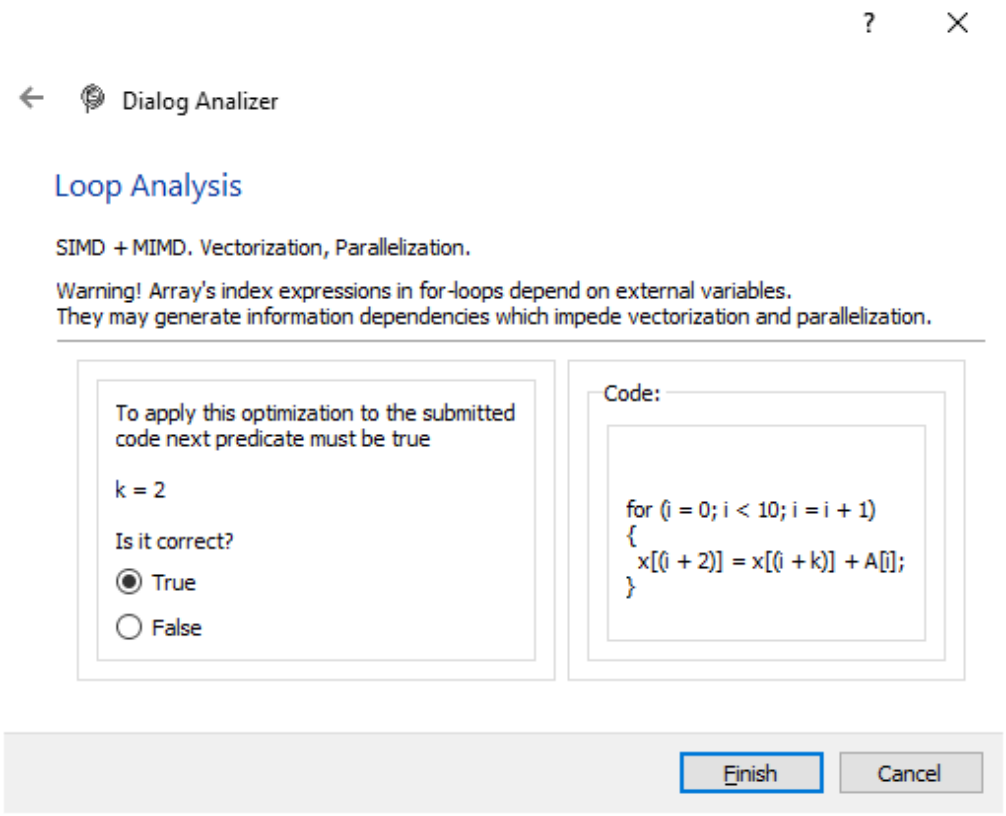

Рисунок 4.9. «Диалоговый анализатор». Шаг 3. Анализ программы для гибридной архитектуры SIMD+MIMD. Пример 4.3.2

Пример 4.3.3.

```
int k;
int x[20], A[20];
int main(){
     k = 1;
     for(int i=0; i<10; i++){
          V2       V1
          |        |
          x[i+2] = x[i+k] + A[i];
          x[i-3] = x[i+k] + A[i];
     }    |        |
          V4       V3
}
```

В данном примере присутствует один цикл, в нём четыре вхождения массива x (V1, V2, V3, V4), два вхождения массива A и одна внешняя переменная k, влияющая на информационные зависимости массива x.

Распараллеливание на архитектуру MIMD

Для распараллеливания на MIMD нужно, чтобы индексы при всех вхождениях одной переменной были равны. Составим систему:

$$\begin{cases} i + k = i + 2 \\ i + k = i - 3 \\ i + 2 = i - 3 \end{cases}$$

Полученная система не имеет решений, значит, цикл нельзя распараллелить.

Векторизация цикла

Для проверки возможности векторизации цикла строится следующая система неравенств для пар индексов вхождений ({V2, V4}, {V1, V2}, {V1, V4}, {V3, V2}, {V3, V4}):

$$\begin{cases} V2 \geq V4 \\ V1 \geq V2 \\ V1 \geq V4 \\ V2 \geq V3 \\ V3 \geq V4 \end{cases} \Leftrightarrow \begin{cases} i + 2 \geq i - 3 \\ i + k \geq i + 2 \\ i + k \geq i - 3 \\ i + 2 \geq i + k \\ i + k \geq i - 3 \end{cases}$$

Решая систему неравенств, получим, что k=2.

Окна анализа цикла из примера 4.3.3 для разных архитектур:

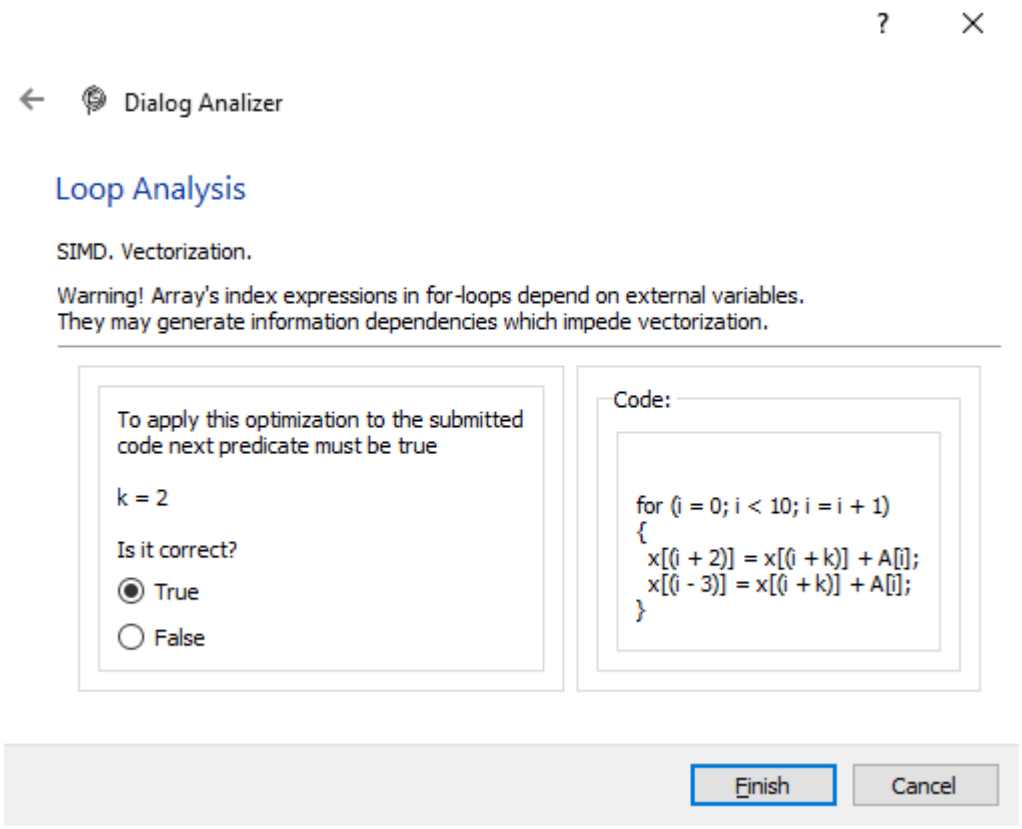

Рисунок 4.10. Пример 4.3.3. Анализ программы для архитектуры SIMD

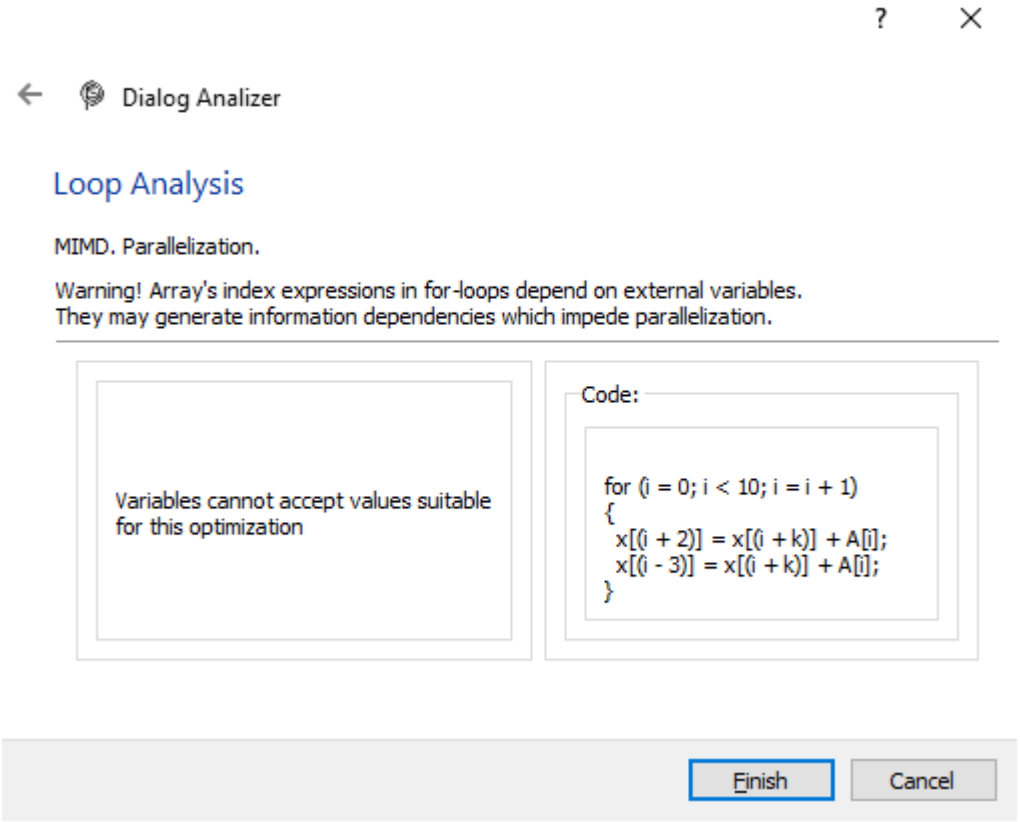

Рисунок 4.11. Пример 4.3.3. Анализ программы для архитектуры MIMD

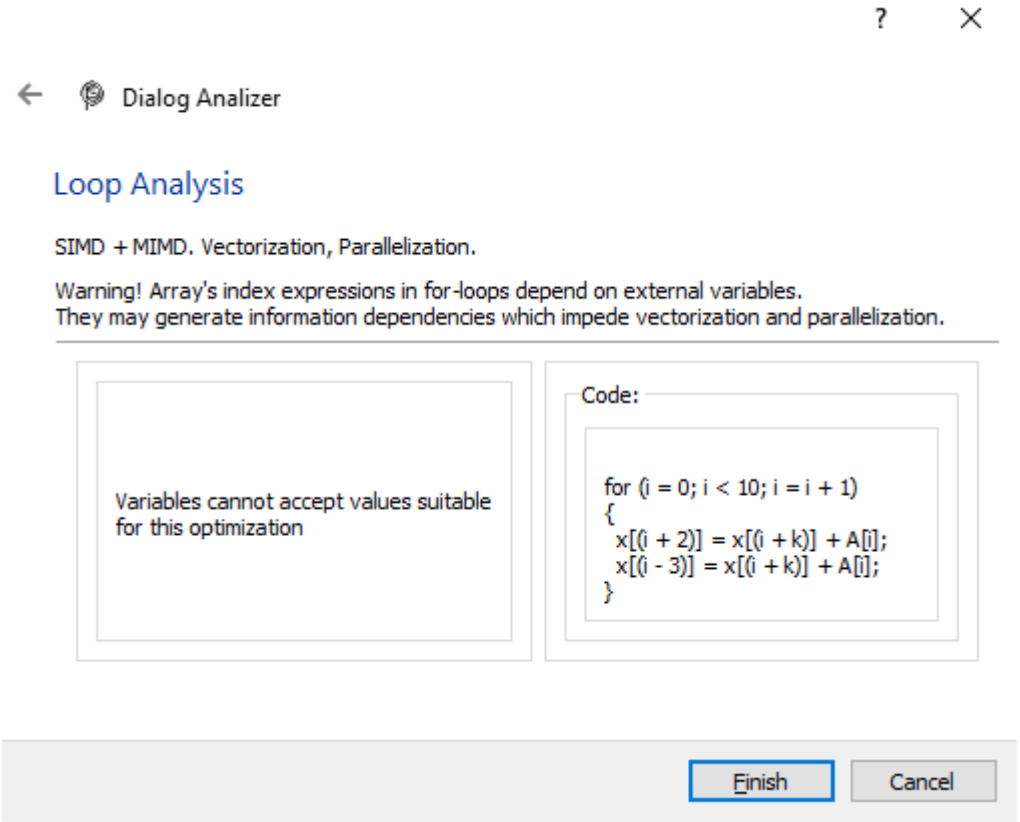

Рисунок 4.12. Пример 4.3.3. Анализ программы для гибридной архитектуры SIMD + MIMD

Пример 4.3.4.

```
int n,k;
int x[20], A[20];
int main(){
     k = 1; n = 1;
     for(int i=0; i<10; i++){
          V2       V1
          |        |
          x[i+1] = x[i+k] + A[i];
          x[i-4] = x[i+n] + A[i];
     }    |        |
          V4       V3
}
```

В данном примере присутствует один цикл, в нём четыре вхождения массива x (V1, V2, V3, V4), два вхождения массива A и две внешние переменные k и n, влияющие на информационные зависимости массива x.

Распараллеливание на архитектуру MIMD

Для распараллеливания на MIMD нужно, чтобы индексы при всех вхождениях одной переменной были равны. Составим систему:

$$\begin{cases} i+k=i+n \\ i+k=i+1 \\ i+k=i-4 \\ i+n=i+1 \\ i+n=i-4 \end{cases}$$

Полученная система не имеет решений, значит, цикл нельзя распараллелить.

Векторизация цикла

Для проверки возможности векторизации цикла строится следующая система неравенств для пар индексов вхождений ({V2, V4}, {V1, V2}, {V1, V4}, {V3, V2}, {V3, V4}):

$$\begin{cases} V2 \geq \mathrm{V4} \\ V1 \geq \mathrm{V2} \\ V1 \geq \mathrm{V4} \\ V2 \geq \mathrm{V3} \\ V3 \geq \mathrm{V4} \end{cases} \Leftrightarrow \begin{cases} \mathrm{i}+1 \geq \mathrm{i}-4 \\ \mathrm{i}+\mathrm{k} \geq \mathrm{i}+1 \\ \mathrm{i}+\mathrm{k} \geq \mathrm{i}-4 \\ \mathrm{i}+1 \geq \mathrm{i}+\mathrm{k} \\ \mathrm{i}+\mathrm{k} \geq \mathrm{i}-4 \end{cases}$$

Решая систему неравенств, получим, что $k \geq 1$, $-4 \leq n \leq 1$.

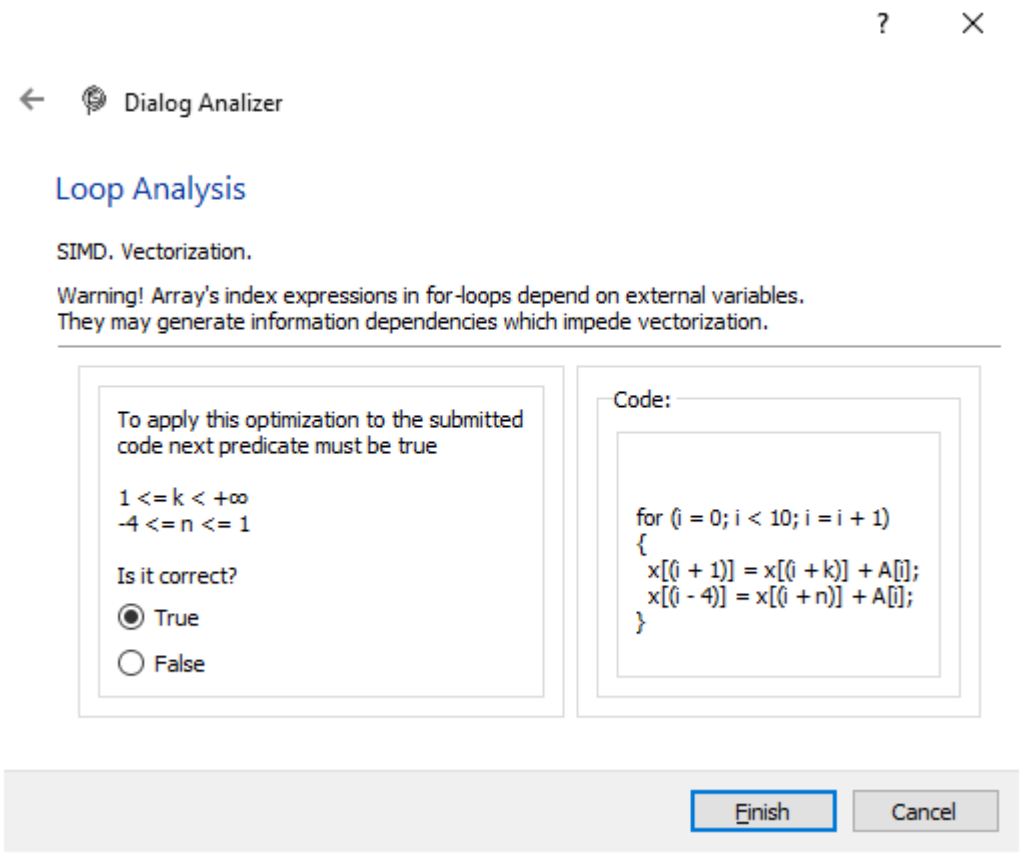

Рисунок 4.13. Пример 4.3.4. Анализ программы для архитектуры SIMD

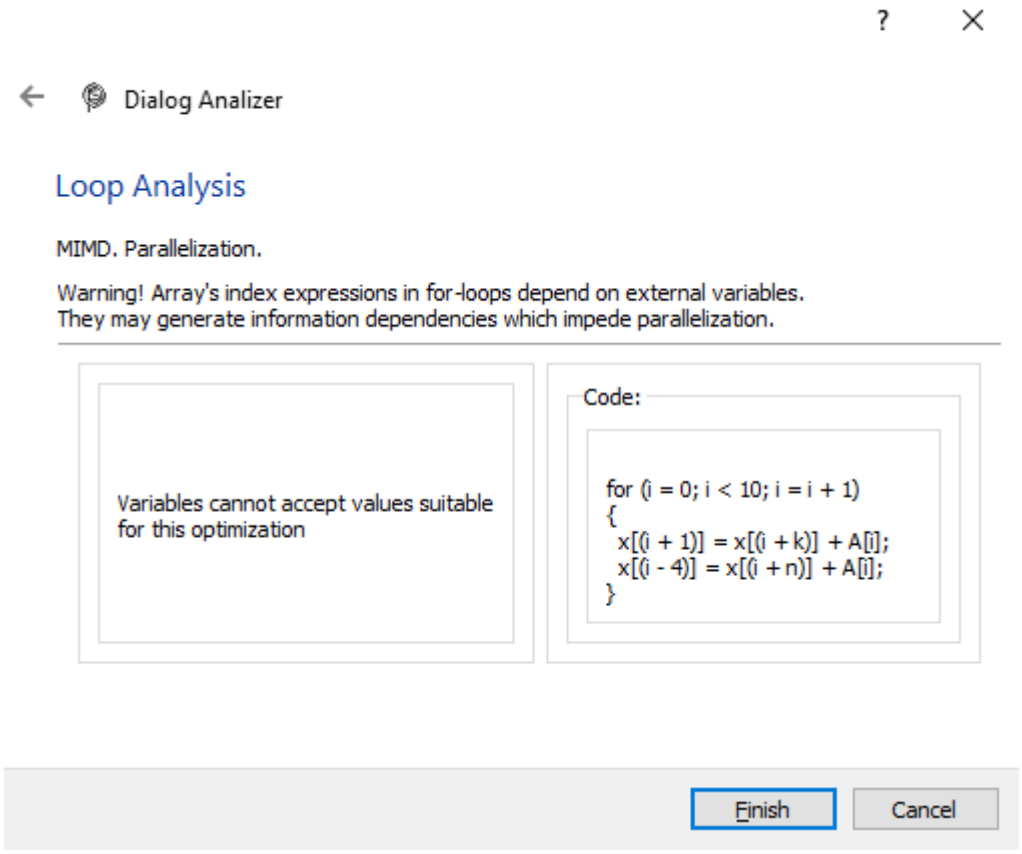

Рисунок 4.14. Пример 4.3.4. Анализ программы для архитектуры MIMD

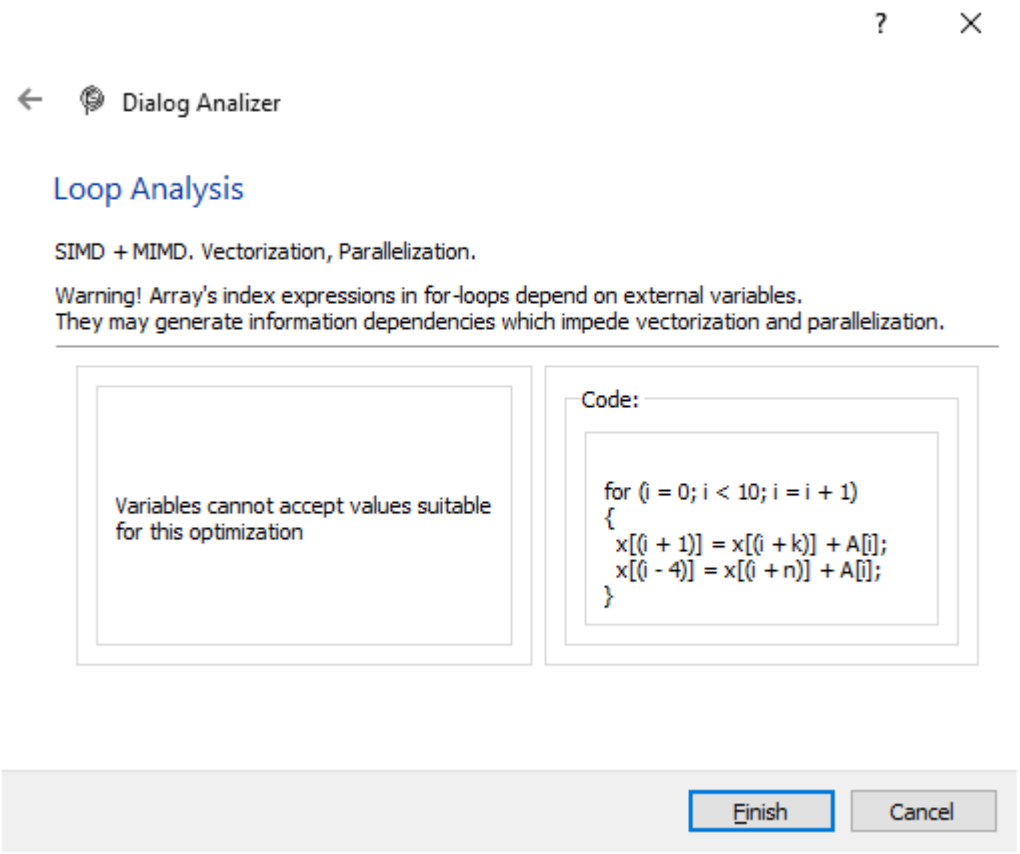

Рисунок 4.15. Пример 4.3.4. Анализ программы для гибридной архитектуры SIMD+MIMD

Пример 4.3.5.

```
int k;
int x[20], A[20];
int main(){
      k = 1;
      for(int i=0; i<10; i++){
            x[i+2] =x [i+k] + A[i];
      }     |         |
            V2        V1
      for(int i=0;i<10;i++){
            x[i+k] = x[i+4] + A[i];
      }     |         |
            V4        V3
}
```

В данном примере присутствуют два цикла, в каждом два вхождения массива x (V1, V2, V3, V4), одно вхождение массива A и одна внешняя переменная k, влияющая на информационные зависимости массива x.

Распараллеливание на архитектуру MIMD

Для распараллеливания на MIMD нужно, чтобы индексы при всех вхождениях одной переменной были равны. Составим систему:

$$\begin{cases} i + k = i + 2 \\ i + k = i + 4 \\ i + 2 = i + 4 \end{cases}$$

Полученная система не имеет решений. Одновременно рассматриваемые циклы не распараллелить. Но один из них можно: k=2 – распараллеливание первого цикла, k=4 – распараллеливание второго цикла.

Векторизация цикла

Для проверки возможности векторизации цикла строится следующая система неравенств для пар индексов вхождений ({V1, V2}, {V3, V4}):

$$\begin{cases} V1 \geq \mathrm{V2} \\ V3 \geq \mathrm{V4} \end{cases} \Leftrightarrow \begin{cases} i + k \geq \mathrm{i} + 2 \\ i + 4 \geq \mathrm{i} + \mathrm{k} \end{cases}$$

Решая систему неравенств, получим, что $2 \leq \mathrm{k} \leq 4$.

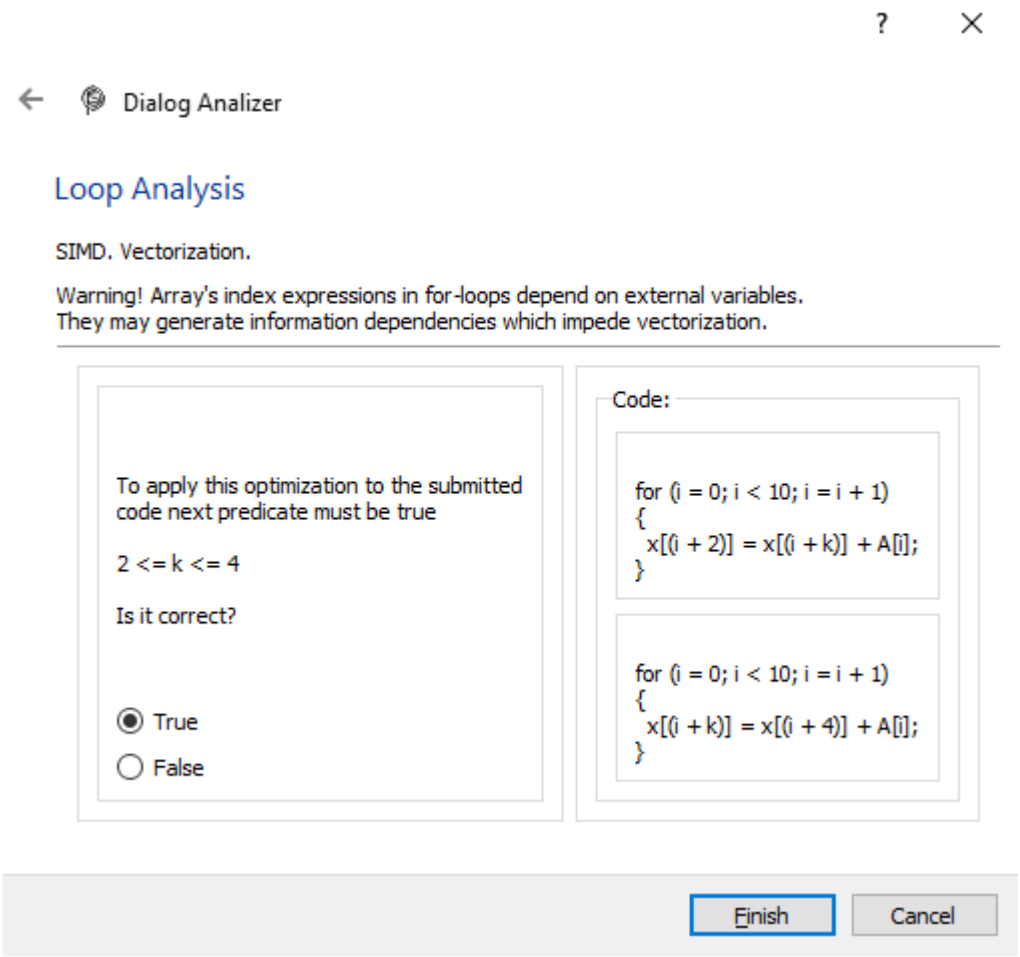

Рисунок 4.16. Пример 4.3.5. Анализ программы для архитектуры SIMD

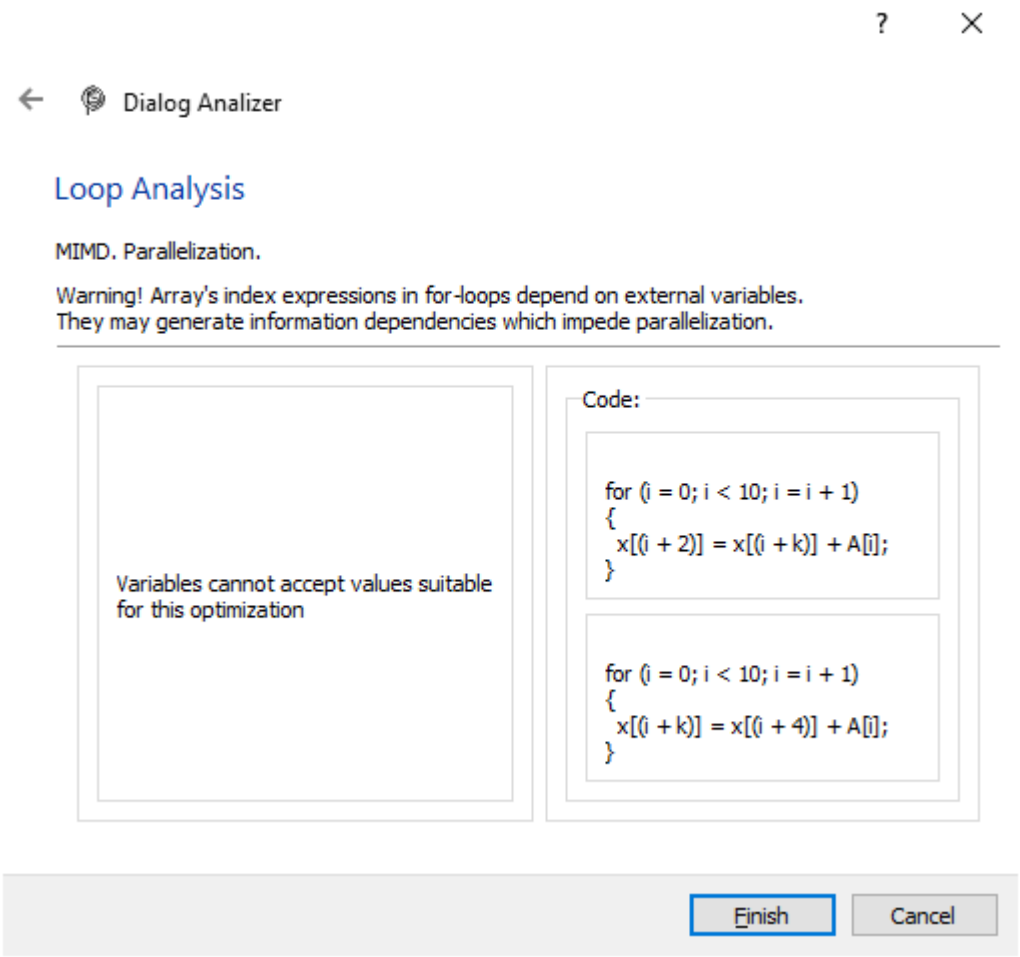

Рисунок 4.17. Пример 4.3.5. Анализ программы для архитектуры MIMD

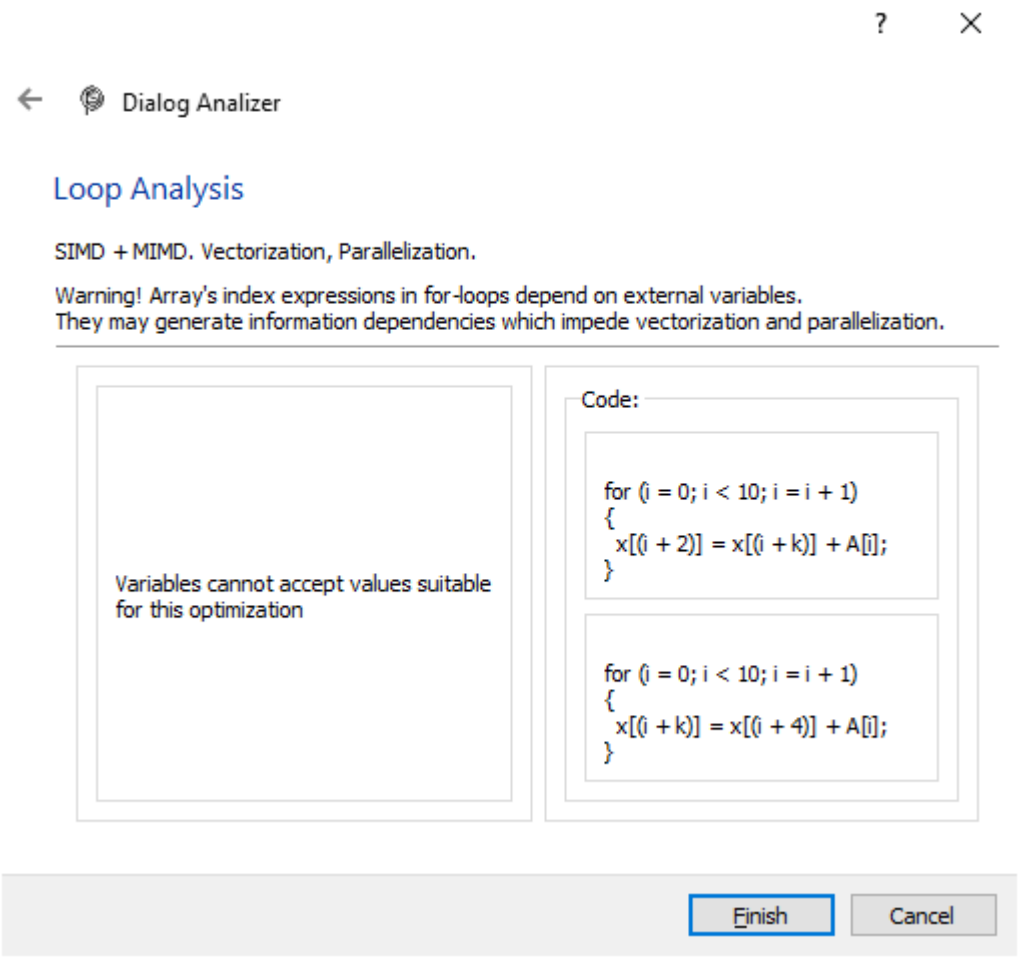

Рисунок 4.18. Пример 4.3.5. Анализ программы для гибридной архитектуры SIMD + MIMD

Пример 4.3.6.

```
int n, k;
int x[20], A[20];
int main(){
     k = 1; n = 1;
     for(int i=0; i<10; i++){
          V2        V1
          |         |
          x[i+5] = x[i+k] + A[i];
          x[i-4] = x[i+n] + A[i];
     }    |         |
          V4        V3

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

```
            V6        V5
             |         |
            x[i+1] = x[i+k] + A[i];
            x[i-2] = x[i+n] + A[i];
        }   |         |
            V8        V7
}
```

В данном примере присутствуют два цикла, в каждом четыре вхождения массива x (V1, V2, V3, V4, V5, V6, V7, V8), два вхождения массива A и две внешние переменные k и n, влияющие на информационные зависимости массива x.

Распараллеливание на архитектуру MIMD

Для распараллеливания на MIMD нужно, чтобы индексы при всех вхождениях одной переменной были равны. Составим систему:

$$\begin{cases} i+k=i+5 \\ i+k=i-4 \\ i+n=i+5 \\ i+n=i-4 \\ i+5=i-4 \\ i+k=i+1 \\ i+k=i-2 \\ i+n=i+1 \\ i+n=i-2 \\ i+1=i-2 \end{cases}$$

Полученная система не имеет решений, значит, цикл нельзя распараллелить.

Векторизация цикла

Для проверки возможности векторизации цикла строится следующая система неравенств для пар индексов вхождений. Разобьём на 2 системы:

1. По k: {V2, V4}, {V6, V8}, {V1, V2}, {V1, V4}, {V5, V6}, {V5, V8}.
2. По n: {V2, V4}, {V6, V8}, {V2, V3}, {V3, V4}, {V6, V7}, {V7, V8}.

$$\begin{cases} V2 \geq \mathrm{V4} \\ V6 \geq \mathrm{V8} \\ V1 \geq \mathrm{V2} \\ V1 \geq \mathrm{V4} \\ V5 \geq \mathrm{V6} \\ V5 \geq \mathrm{V8} \end{cases} \Leftrightarrow \begin{cases} i+5 \geq \mathrm{i}-4 \\ i+1 \geq \mathrm{i}-2 \\ i+k \geq \mathrm{i}+5 \\ i+k \geq \mathrm{i}-4 \\ i+k \geq \mathrm{i}+1 \\ i+k \geq \mathrm{i}-2 \end{cases} \qquad \begin{cases} V2 \geq \mathrm{V4} \\ V6 \geq \mathrm{V8} \\ V2 \geq \mathrm{V3} \\ V3 \geq \mathrm{V4} \\ V6 \geq \mathrm{V7} \\ V7 \geq \mathrm{V8} \end{cases} \Leftrightarrow \begin{cases} i+5 \geq \mathrm{i}-4 \\ i+1 \geq \mathrm{i}-2 \\ i+n \leq \mathrm{i}+5 \\ i+n \geq \mathrm{i}-4 \\ i+n \leq \mathrm{i}+1 \\ i+n \geq \mathrm{i}-2 \end{cases}$$

Полученные неравенства для каждой переменной: $k \geq 5$, $-2 \leq n \leq 1$.

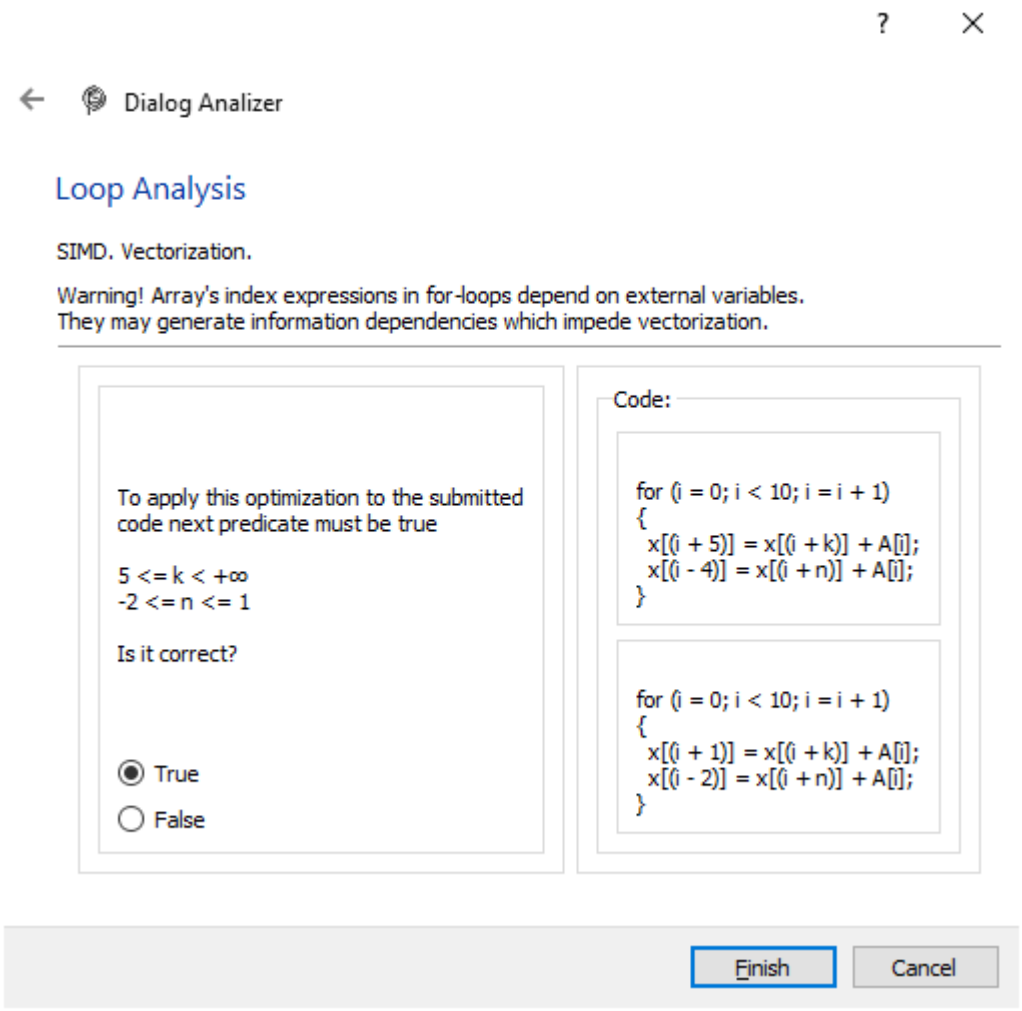

Рисунок 4.19. Пример 4.3.6. Анализ программы для архитектуры SIMD

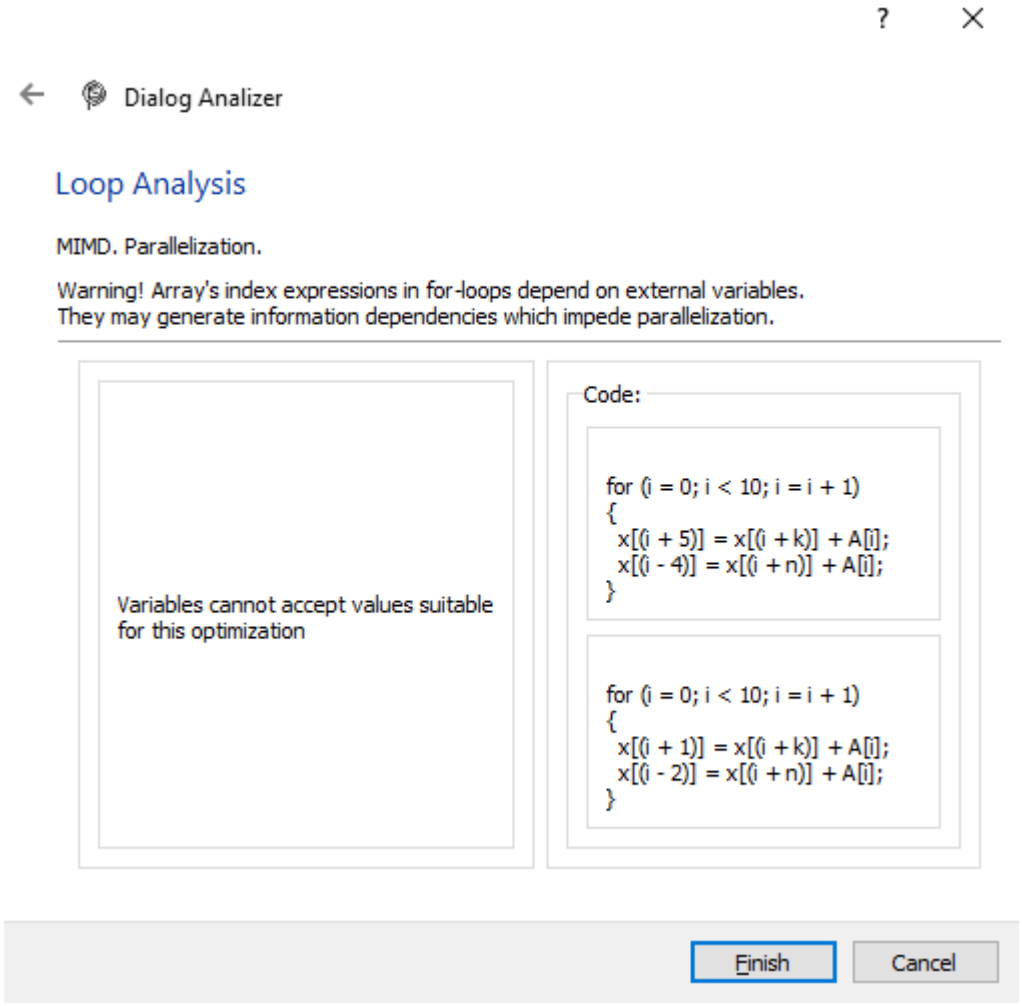

Рисунок 4.20. Пример 4.3.6. Анализ программы для архитектуры MIMD

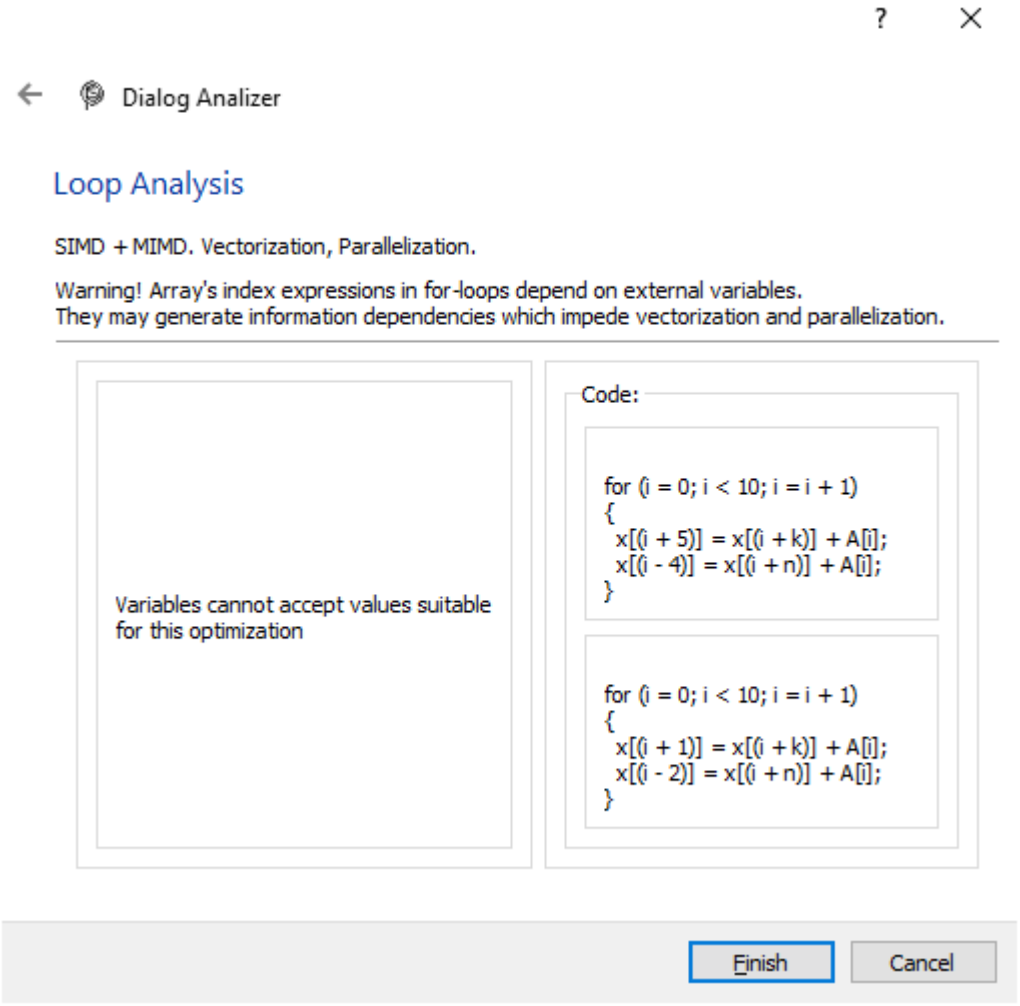

Рисунок 4.21. Пример 4.3.6. Анализ программы для гибридной архитектуры SIMD + MIMD

Пример 4.3.7.

```
int k;
int x[20], y[20], A[20];
int main(){
     k = 1;
     for(int i=0; i<10; i++){
          Vx2      Vx1
          |        |
          x[i+k] = x[i+8] + A[i];
          y[i-4] = y[i+k] + A[i];
     }    |        |
}         Vy2      Vy1
```

В данном примере присутствует один цикл, в нём два вхождения массива x (Vx1, Vx2), два вхождения массива y (Vy1, Vy2), два вхождения массива A и одна внешняя переменная k, влияющая на информационные зависимости массивов x, y.

Распараллеливание на архитектуру MIMD

Для распараллеливания на MIMD нужно, чтобы индексы при всех вхождениях одной переменной были равны. Составим систему:

$$\begin{cases} i + k = i + 8 \\ i + k = i - 4 \end{cases}$$

Полученная система не имеет решений, значит, цикл нельзя распараллелить.

Векторизация цикла

Для проверки возможности векторизации цикла строится следующая система неравенств для пар индексов вхождений ({Vx1, Vx2}, {Vy1, Vy2}):

$$\begin{cases} Vx1 \geq \mathrm{Vx2} \\ Vy1 \geq \mathrm{Vy2} \end{cases} \Leftrightarrow \begin{cases} i + 8 \geq \mathrm{i} + \mathrm{k} \\ i + k \geq \mathrm{i} - 4 \end{cases}$$

Решая систему неравенств, получим: $-4 \leq \mathrm{k} \leq 8$.

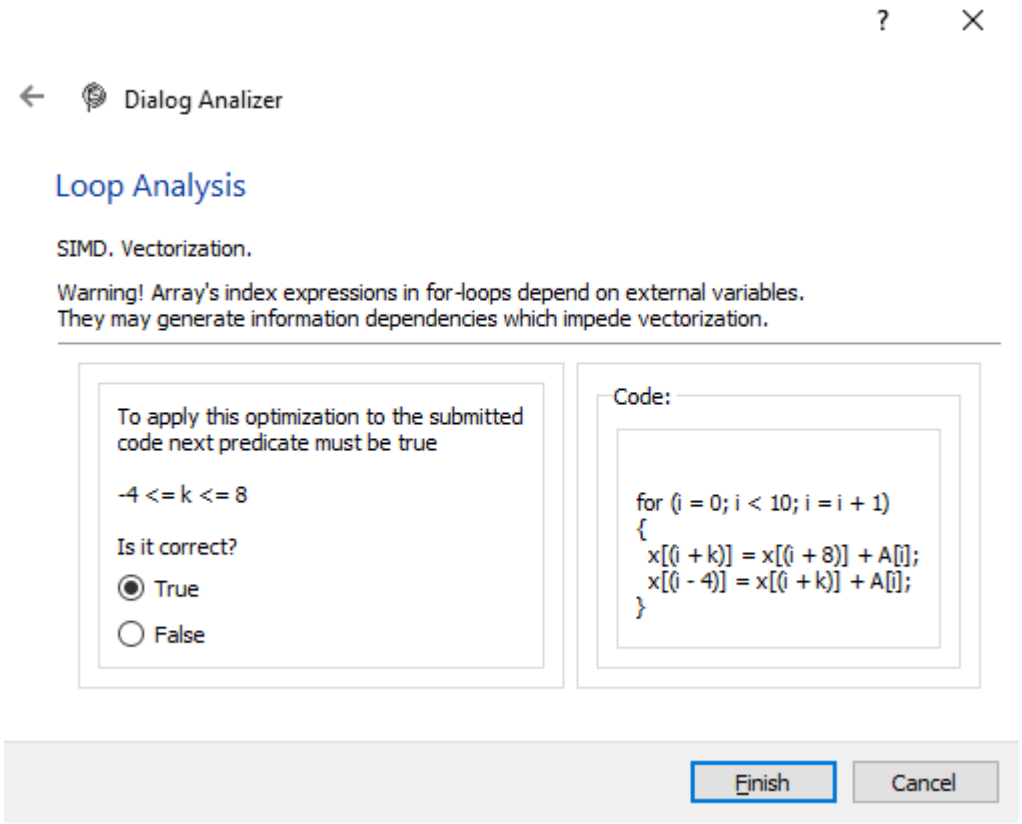

Рисунок 4.22. Пример 4.3.7. Анализ программы для архитектуры SIMD

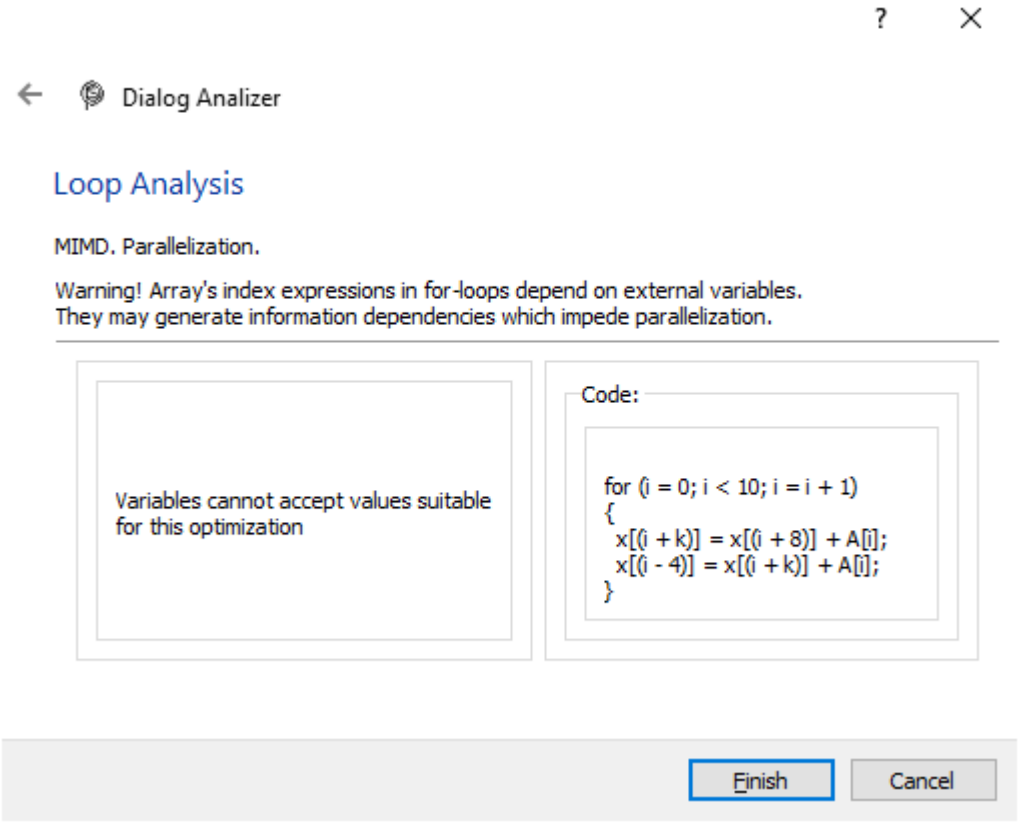

Рисунок 4.23. Пример 4.3.7. Анализ программы для архитектуры MIMD

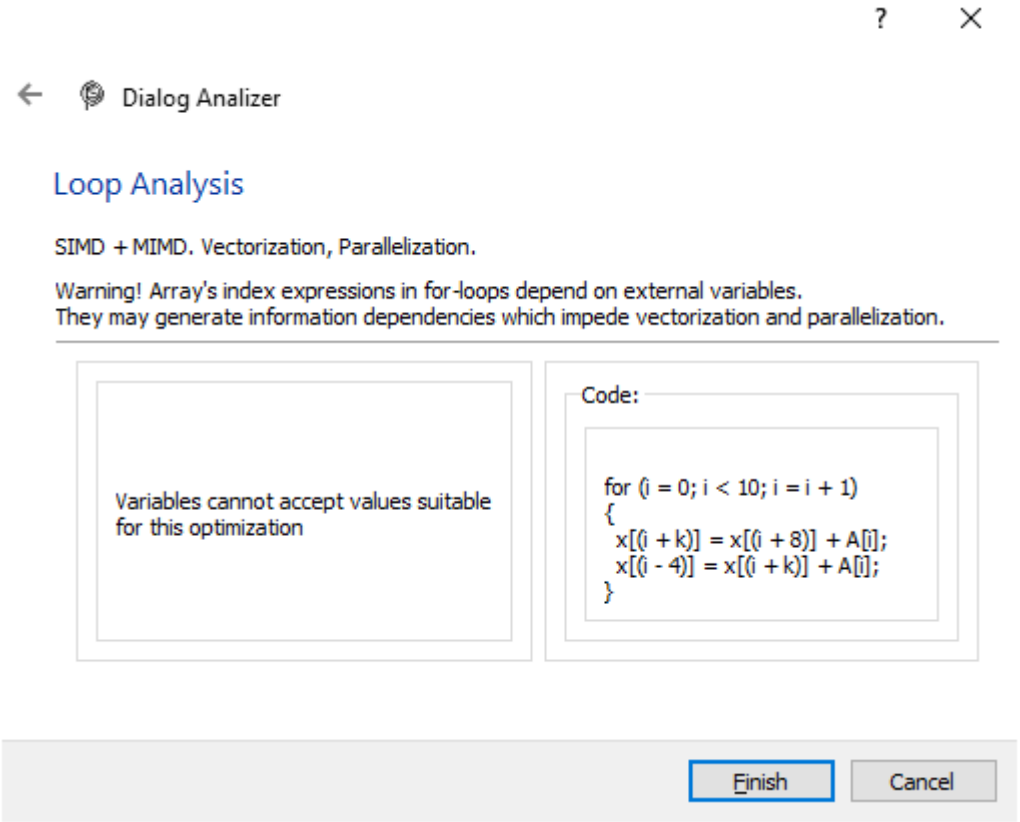

Рисунок 4.24. Пример 4.3.7. Анализ программы для гибридной архитектуры SIMD+MIMD

Пример 4.3.8.

```
int k;
int x[20], y[20], A[20];
int main(){
     k = 1;
     for(int i=0; i<10; i++){
          x[i+k] = y[i+k] + A[i+k+2];
     }    |         |      |
          Vx1       Vy1       VA1
}
```

В данном примере присутствует один цикл, в котором одно вхождение массивов x (Vx1), y (Vy1), A (VA1) и одна внешняя переменная k. В данном примере внешняя переменная не влияет на информационные зависимости в цикле.

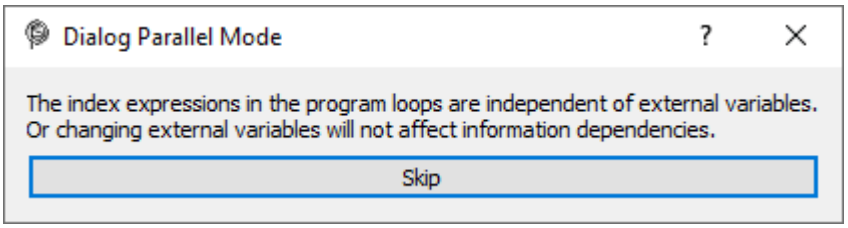

Рисунок 4.25. Диалог, построенный по примеру 4.3.8

Пример 4.3.9.

```
int k;
int x[20], A[20];
int main(){
k = 1;
     for(int i=0; i<10; i++){
          x[i+1] = x[i-1] + A[i];
     }    |         |
}         Vx1       Vx2
```

В данном примере присутствует один цикл, в котором два вхождения массива x (Vx1, Vx2) и одно вхождение массива A. Внешние переменные в цикле отсутствуют.

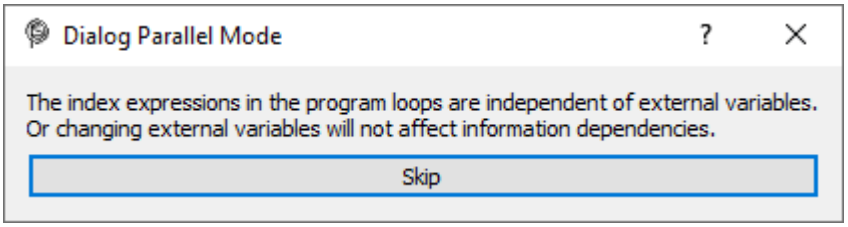

Рисунок 4.26. Диалог, построенный по примеру 4.3.9

Пример 4.3.10.

```
int k;
int x[10], a[10], b[10];
int main(){
      k = 1;
      for(int i=0; i<10; i++){
            x[i] = x[a[i]] + b[i];
      }
}
```

Для распараллеливания на MIMD требуется выполнение условия: `i=a[i]`.

Для того чтобы векторизовать цикл, нужно, чтобы выполнялось условие: `i ≥ a[i]`.

4.3.2. Уточнение информационных зависимостей для применения алгоритма оптимизации итерационных гнёзд циклов

В алгоритме, представленном в данной работе (Глава 2), накладываются ограничения на входную программу. Одним из ограничений является то, что линейные индексные выражения массивов должны иметь вид (i + <целочисленная константа>), где i – один из счётчиков гнезда циклов.

Возможна ситуация, когда в индексном выражении присутствует константа, значение которой неизвестно на этапе компиляции.

Пример 4.3.11.

Пример гнезда циклов: в индексных выражениях массива присутствует переменная c, значение которой неизвестно на этапе компиляции.

```
for (int k = 0; k < K; ++k )
      for (int i = 1; i < N - 1; ++i )
            for (int j = 1; j < M - 1; ++j )
                  u[i][j] = (u[i-c][j]+u[i][j-c]+u[i+c][j]+u[i][j+c]);
```

Для определения значения данной константы можно использовать полуавтоматический диалоговый подход, который попросит пользователя уточнить значение данной переменной или указать диапазон. На основе ответа программа может быть преобразована посредством либо подстановки итогового значения, либо генерации кода, разделённого условными операторами. После этого возможно применение оптимизирующего алгоритма, описанного в главе 2.

4.3.3. Уточнение условий выполнения тайлинга

Определение тайлинга приведено в главе 1. Среди факторов, влияющих на оптимизацию, отмечаются вычислительные характеристики целевой машины, например размер процессорного кэша. Тайлинг хорошо локализует данные для попадания в кэш-память и ускоряет выполнение программы, но является не всегда эквивалентным преобразованием, часто нарушая информационные зависимости. Иногда, несмотря на информационные зависимости, такое преобразование может быть корректным. С помощью дополнительных вопросов пользователю можно подстроиться под кэш-память целевой машины.

Рассмотрим пример работы диалога для данного преобразования в OPC.

Шаг 1. Выбираем программу и гнездо цикла в ней.

Пример 4.3.12.

```
int main(){
      double y, a[10], b[10];
      for(int i=0;i<10;i++) {
            for(int j=0;j<10;j++) {
                  y+=a[i]*b[j];
      }
}
```

В этом примере применение тайлинга может привести к изменению погрешностей округления или переполнению регистров, но если разработчик программы (пользователь компилятора) знает, что диапазоны используемых переменных не приведут к переполнению и погрешность округлений не выйдет за пределы допустимых значений, то он может принять решение о применении тайлинга с возможным последующим распараллеливанием.

Шаг 2. Применяем диалоговое преобразование.

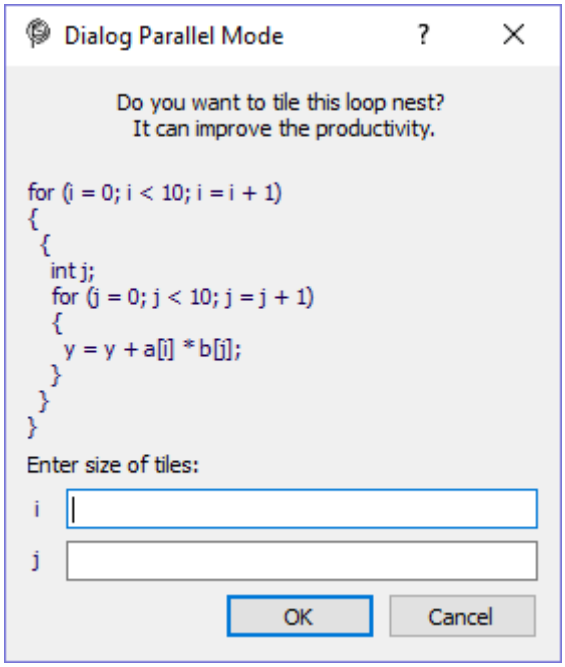

Рисунок 4.27. Вызов диалога для преобразования «тайлинг»

Шаг 3. Выбираем размеры блоков, на которые разбивается пространство итераций исходного гнезда циклов. Например, 2 x 5.

Пример, обосновывающий применение данного преобразования, но нарушающий информационные зависимости. Решение задачи Дирихле уравнения Пуассона.

После применения тайлинга к данному гнезду циклов получаем эквивалентную программу, но с точки зрения теории информационных зависимостей происходит нарушение информационных зависимостей. Компилятор самостоятельно не применит тайлинг для данного примера, поэтому пользователю можно предложить его применить.

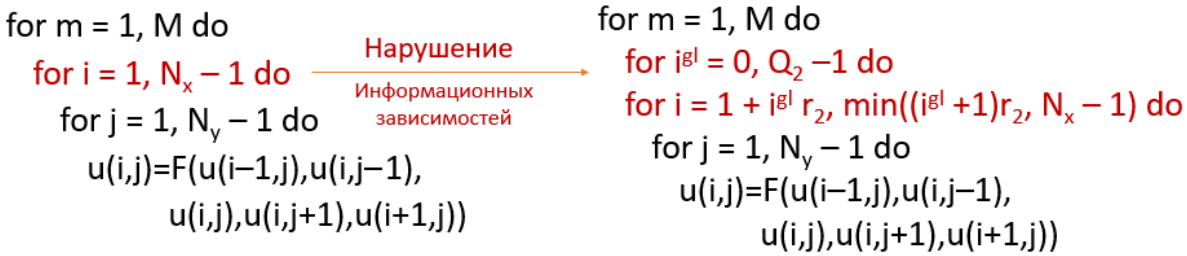

Рисунок 4.28. Пример псевдокода: применение тайлинга к решению задачи Дирихле уравнения Пуассона

4.3.4. Диалоговое выполнение гнездования цикла

Оптимизация «гнездование» пытается расщепить цикл на несколько частей, имеющих одно и то же тело исходного цикла и различные диапазоны счётчика. За счёт данного

преобразования каждую итерацию внешнего цикла можно выполнять в разных потоках, что ускоряет вычисления.

Это может, однако, вызвать проблемы, такие как: переполнение (для целых чисел) или изменение погрешности (для чисел с плавающей точкой), поэтому для применения данной оптимизации нужно согласие пользователя.

Рассмотрим пример работы диалога для данного преобразования в OPC.

Пример 4.3.13.

Шаг 1. Выберем программу с циклом.

```
int main(){
      int y, a[10], b[10];
      for(int i=0;i<10;i++){
            y+=a[i]*b[i];
      }
}
```

Шаг 2. Применяем диалоговое преобразование.

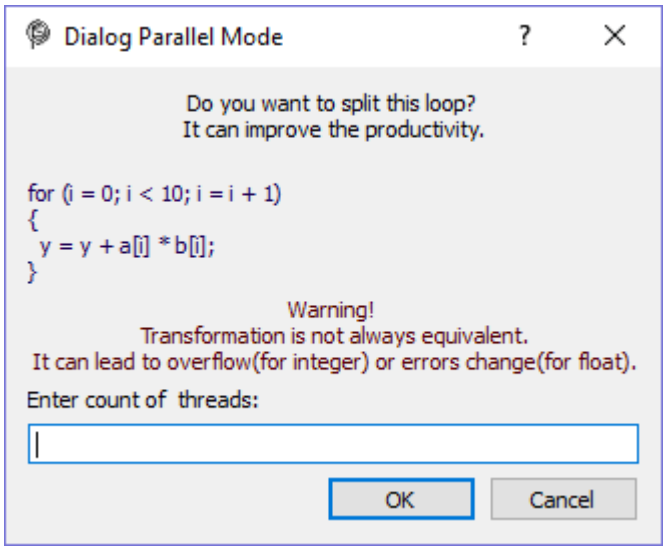

Рисунок 4.29. Вызов диалога для преобразования «гнездование цикла»

Шаг 3. Выбираем количество частей (потоков), на которое будем расщеплять цикл (в нашем случае на два).

Получаем три цикла:

- 1-й – гнездо циклов, которое можно выполнять параллельно; результат записывается во временный массив (__uni3[2]);

- 2-й – вычисляется остаток итераций в случае, если количество итераций цикла не делится нацело на параметр разбиения;
- 3-й – суммирование вычисленных значений, записанных в массив.

Листинг 4.3.1.

```
int main()
{
  int y;
  int a[10];
  int b[10];
  {
    int i;
    {
      int __uni0i;
      int __unili;
      int __uni2i;
      int __uni3[2];
      int __uni4i;

      for (__uni0i = 0; __uni0i < 2; __uni0i = __uni0i + 1)
      {
        for (__unili = 0; __unili < 10 / 2; __unili = __unili + 1)
        {
          __uni3[__uni0i] = __uni3[__uni0i] + a[(__uni0i * (10 / 2) + __unili)] *
b[(__uni0i * (10 / 2) + __unili)];
        }
      }

      for (__uni2i = 0; __uni2i < 10 % 2; __uni2i = __uni2i + 1)
      {
        y = y + a[(2 * (10 / 2) + __uni2i)] * b[(2 * (10 / 2) + __uni2i)];
      }
      i = 10;

      for (__uni4i = 0; __uni4i < 2; __uni4i = __uni4i + 1)
      {
        y = y + __uni3[__uni4i];
      }
    }
  }
}
```

Автоматически сгенерированные переменные: счётчики циклов (__uni0i, __uni1i, __uni2i, __uni4i) и временный массив (__uni3).

Пример 4.3.14. Суммирование значений матрицы.

```
int a[100][100], y, k, i;
for(k=0; k<100; k++)
    for(i=0; i<100; i++)
        y+=a[i][k];
```

В результате можно в нескольких потоках (например, в пяти) выполнять суммирование значений в столбцах матрицы, а потом в последнем цикле просуммировать результаты, записанные во временный массив (__uni3[5]), и получить эквивалентный результат.

```
int main()
{
  int y;
  int a[100][100];
  {
    int k;
    {
      int __uni0k;
      int __uni1k;
      int __uni2k;
      int __uni3[5];
      int __uni4k;
      for (__uni0k = 0; __uni0k < 5; __uni0k = __uni0k + 1)
      {
        for (__uni1k = 0; __uni1k < 100 / 5; __uni1k = __uni1k + 1)
        {
          {
            int i;
            for (i = 0; i < 100; i = i + 1)
            {
              __uni3[__uni0k] = __uni3[__uni0k] + a[i][(__uni0k * (100 / 5) + __uni1k)];
            }
          }
        }
      }
      for (__uni2k = 0; __uni2k < 100 % 5; __uni2k = __uni2k + 1)
      {
        {
          int i;
          for (i = 0; i < 100; i = i + 1)
          {
            y = y + a[i][(5 * (100 / 5) + __uni2k)];
          }
        }
      }
      k = 100;
      for (__uni4k = 0; __uni4k < 5; __uni4k = __uni4k + 1)
      {
        y = y + __uni3[__uni4k];
      }
    }
  }
}
```

Автоматически сгенерированные переменные: счётчики циклов (__uni0k, __uni1k, __uni2k, __uni4k) и временный массив (__uni3[5]).

4.4. Сравнение динамического и диалогового подходов к оптимизации программ

Динамическая оптимизация состоит в оптимизации программы в процессе её выполнения [144].

4.4.1. Пример динамического распараллеливания

Анализ значений переменных при выполнении динамической оптимизации может сильно замедлять выполнение программы. В случае алгоритма Флойда-Уоршелла (пример 4.1.1) для распараллеливания потребуется проанализировать значения входной матрицы, которая может иметь большие размерности.

Рассмотрим пример 4.3.2. В зависимости от разных значений переменной k возможно распараллеливание на различные архитектуры. После применения динамического распараллеливания:

```
if (k == 2) {
      // оптимизации для архитектур simd и mimd
} else if (k > 2) {
      // оптимизации для архитектуры simd
} else {
      // исходная программа
      for(int i=0; i<10; i++){
            x[i+2] = x[i+k] + A[i];
      }
}
```

В результате применения динамического распараллеливания увеличивается объём кода. Из-за условных операторов может увеличиться время выполнения программы. Если проверяемые переменные редко принимают значение, которое удовлетворяет случаю, при котором будет выполнен распараллеленный код, то исходная программа будет в среднем работать медленнее.

4.4.2. «Диалоговый анализатор» учитывает возможность переполнения

Рассмотрим другой пример. Требуется распараллелить вычисление количества сочетаний C(n, k)=n!/(k!*(n-k)!)=1/k! * $\prod_{i=0}^{k-1} n - i$, где n – число элементов в множестве, k – число элементов в подмножестве.

Листинг 4.4.1. Исходная программа для вычисления количества сочетаний.

```
int n, k;
float c;
for (int i = 1; i <= k; i++) {
     c *= n - i + 1;
     c /= i;
}
```

Если мы хотим преобразовать программу, используя расщепление цикла, таким образом, чтобы параллельно выполнить вычисление числителя и знаменателя (листинг 4.4.2), то существует вероятность переполнения, когда вычисляемое значение выходит за границы, установленные размером типа переменной.

Листинг 4.4.2. Преобразованная программа.

```
int n, k;
float c, c1, c2;
for (int i = 1; i <= k; i++) { // поток 1
     c1 *= n - i + 1;
}
for (int i = 1; i <= k; i++) { // поток 2
     c2 /= i;
}
c=round(c1*c2)
```

В этом случае для применения динамического распараллеливания потребуется сформировать условие, оценивающее возможность переполнения, что приведёт к последовательному вычислению значения c1, что по сути является половиной вычислений. В таком случае прирост производительности будет мизерным.

4.4.3. «Диалоговый анализатор» учитывает возможность изменения погрешности

Рассмотрим пример 4.3.14. Если после применения гнездования мы захотим распараллелить выполнение цикла, то порядок выполнения операций сложения чисел с плавающей точкой будет изменён, что может привести к изменению погрешности вычислений, и в результате будет получен неэквивалентный результат. Такого рода проблему динамическая оптимизация не сможет разрешить.

4.5. Выводы к главе 4

Реализован, основанный на символьном анализе, диалоговый режим общения компилятора с пользователем на основе оптимизирующей распараллеливающей системы (ОРС). Сформированный символьным анализом предикат при помощи линеаризации выражений для лучшего понимания пользователем. На основе предиката и фрагмента программы на исходном языке формулируется вопрос. Пользователь вводит ответ на вопрос системы. Система на основе ответов пользователя уточняет информационные зависимости. На уточнённых зависимостях может приниматься решение о применении оптимизирующих распараллеливающих преобразований.

Существуют тексты программ, из которых невозможно получить информацию о зависимостях без участия пользователя. В главе приводятся тексты таких программ. Таким образом «диалоговый анализатор» расширяет класс оптимизируемых задач.

Приводятся сравнения динамических оптимизаций и диалоговых. Исходя из представленных в данной главе примеров, можно сделать вывод о том, что динамический анализ программ не заменяет диалоговый подход; когда программисту известны значения переменных, может быть применён диалоговый подход, иначе – динамический.

Если программа из высокоуровневого внутреннего представления выводится на исходном языке, то она может быть более похожа на исходный код, чем после вывода из низкоуровневого внутреннего представления. Именно поэтому результаты промежуточных преобразований легче читаются при отладке. По этой же причине, если пользователю в диалоге демонстрировать фрагменты его кода, эти фрагменты более узнаваемы. Это позволяет делать диалоговый анализ кода более удобным для разработчика. Используется вывод из внутреннего представления ОРС в язык С. Следует отметить, что такого вывода нет в компиляторах GCC и LLVM.

Таким образом, в главе 4 разработан метод диалогового анализа и преобразований текстов программ на основе символьного анализа. Тем самым решается задача 3.

Заключение

В ходе выполнения диссертационного исследования были получены следующие результаты:

1. Доказана эквивалентность разработанной в оптимизирующей распараллеливающей системе (OPC) цепочки преобразований: «скошенный тайлинг», «метод гиперплоскостей», «распараллеливание с использованием OpenMP», «перестановка циклов внутри тайла», «линеаризация», «вынос общих инвариантных выражений». Тем самым решается выносимое на защиту положение 1.

2. В OPC реализованы преобразования «скашивание циклов», «метод гиперплоскостей», «скошенный тайлинг», тем самым решается положение 3, выносимое на защиту.

3. Теоретически и экспериментально обоснован новый метод обхода точек тайла, который повышает временную локальность данных и даёт ускорение. Тем самым частично решается положение 4, выносимое на защиту.

4. Предложен новый метод расчета оптимальных размеров тайла для получения лучшего ускорения. Тем самым завершается решение выносимого на защиту положения 4.

5. Проведены численные эксперименты, демонстрирующие ускорение программ, преобразованных с использованием разработанного алгоритма. Получены ускорения относительно исходных программ для: алгоритма решения задачи теплопроводности (в 3,95 раза); алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнений Лапласа (в 20,26 раза) и Пуассона (в 17,08 раз). Для алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле достигнуто ускорение в 16,33 раза, что на 40% эффективнее чем при использовании известной системы PLUTO. Тем самым подтверждается правильность полученных теоретических моделей и практическая значимость выполненной работы.

6. Предложен основанный на символьном анализе подход к диалоговому ускорению программ, который позволяет расширить класс задач, оптимизируемых полуавтоматически с помощью компилятора. Приводятся примеры программ, которые нельзя оптимизировать автоматически, но можно с помощью в предложенного метода. Тем самым решается положение 2, выносимое на защиту.

Задача 1 решена в главе 1; задача 2 решена в главах 2, 3; задача 3 решена в главе 4. Таким образом, все поставленные задачи решены и цель диссертации достигнута.

В главе 1 реализованы оптимизирующие преобразования гнёзд циклов «скашивание», «тайлинг», «метод гиперплоскостей» для древовидного внутреннего представления OPC, в отличие от известных оптимизирующих систем, где преобразования гнёзд циклов основаны на внутреннем представлении полиэдра. В главе 2 получен алгоритм автоматизированного

ускорения гнёзд циклов итерационного типа – более быстрый, чем известный, что показано в главе 3. В главе 4 предложен метод диалогового уточнения информационных зависимостей, который позволяет распараллеливать более широкий класс циклов, чем прежние методы.

Теоретическая значимость состоит в развитии теории преобразования программ для компилирующих систем с высокоуровневым универсальным (древовидным) внутренним представлением. В рамках работы предложен новый метод обхода точек тайла; предложены методы расчета оптимальных размеров тайлов; предложен новый метод диалоговой оптимизации и распараллеливания циклов. Приводятся преимущества высокоуровневого древовидного внутреннего представления для формирования цепочек преобразований программ. Приводится анализ возможного использования описанной цепочки преобразований для распараллеливания на распределённую память.

Следует отметить, что и метод преобразования гнёзд циклов итерационного типа, и метод диалоговой оптимизации при преобразовании программ сохраняют последовательность обращений к памяти. Это доказывает, что результирующая и исходная программы имеют одинаковые условия сходимости (для итерационных алгоритмов), одинаковые погрешности машинных округлений и одинаковые условия переполнения регистров (превышения максимальных размеров типов данных).

Предложенные в данной работе методы могут быть реализованы при создании инструментов (компиляторов, конверторов) автоматического (и полуавтоматического) распараллеливания и оптимизации программ. Это позволит сократить время разработки высокопроизводительных программ, тем самым повышая производительность труда программистов.

Предложенные в работе методы ориентированы на оптимизацию алгоритмов итерационного типа и могут найти применение в следующих приложениях. В статье [92] рассматриваются алгоритмы итерационного типа, такие как дискретный оператор Лапласа (Discrete Laplace operator), алгоритм итерационного типа для имитации распределения температуры в организме человека при лечении рака гипертермией [93], [94]. Уравнения Пуассона и Лапласа являются основными дифференциальными уравнениями электростатики [145]. В работе [146] рассматривается моделирование влияния факельной установки на деградацию вечной мерзлоты. В этой работе предлагается использовать численные методы Самарского А.А. и Моисеенко Б.Д. [147], которые могут решаться итерационными методами, допуская ускорение предложенными в данной диссертации методами.

Библиография

1. Gong Z., Chen Z., Szaday Z., Wong D., Sura Z., Watkinson N., Maleki S., Padua D., Veidenbaum A., Nicolau A. An empirical study of the effect of source-level loop transformations on compiler stability // Proceedings of the ACM on Programming Languages. 2018. № 2. P. 1–29.

2. Intelligence Processing Unit. [Электронный ресурс]: URL: https://www.graphcore.ai/products/ipu (дата обращения: 01.04.2024).

3. Jia Zh., Tillman B., Maggioni M., Scarpazza D.P. Dissecting the Graphcore IPU Architecture via Microbenchmarking // Technical Report. December 7. 2019. P. 91.

4. Robinson C. Untether.AI Boqueria 1458 RISC-V Core AI Accelerator. [Электронный ресурс]: URL: https://www.servethehome.com/untether-ai-boqueria-1458-risc-v-core-aiaccelerator-hc34/ (дата обращения: 01.04.2024).

5. Yen I.E., Xiao Zh., Xu D. S4: a High-sparsity, High-performance AI Accelerator. [Электронный ресурс]: URL: https://arxiv.org/abs/2207.08006 (дата обращения: 01.04.2024).

6. Peckham O. SambaNova Launches Second-Gen DataScale System. [Электронный ресурс]: URL: https://www.hpcwire.com/2022/09/14/sambanova-launches-second-gen-datascalesystem/ (дата обращения: 01.04.2024).

7. Barth A. Amazon EC2 TRN1 instances for high-performance model training are now available. [Электронный ресурс]: URL: https://aws.amazon.com/blogs/aws/amazon-ec2- trn1-instances-for-high-performance-model-training-are-now-available/ (дата обращения: 01.04.2024).

8. AWS Neuron Documentation. [Электронный ресурс]: URL: https://awsdocs-neuron.readthedocshosted.com (дата обращения: 01.04.2024).

9. Hot Chips 34 – Tesla’s DojoMicroarchitecture. [Электронный ресурс]: URL: https://chipsandcheese.com/2022/09/01/hot-chips-34-teslas-dojomicroarchitecture/ (дата обращения: 01.04.2024).

10. СБИС К1879ВМ8Я. [Электронный ресурс]: URL: https://www.module.ru/products/1/26-18798 (дата обращения: 07.10.2023).

11. Reuther A., Michaleas P., Jones M., Gadepally V., Samsi S., Kepner J. AI Accelerator Survey and Trends // 2021 IEEE High Performance Extreme Computing Conference (HPEC). Waltham, MA, USA, 2021. P. 1–9.

12. Волконский В.Ю. Оптимизирующие компиляторы для архитектуры с явным параллелизмом команд и аппаратной поддержкой двоичной совместимости // Информационные технологии и вычислительные системы. 2004. № 3. с. 4–26.

13. Евстигнеев В.А., Касьянов В.Н. Оптимизирующие преобразования в распараллеливающих компиляторах // Программирование. 1996. Т. 22, № 6. с. 12–26.

14. Касьянов В.Н., Мирзуитова И.Л. Реструктурирующие преобразования: алгоритмы распараллеливания циклов // Программные средства и математические основы информатики. Новосибирск: Институт систем информатики им. А.П. Ершова СО РАН, 2004. с. 142–188.

15. Евстигнеев В.А., Мирзуитова И.А. Анализ циклов: выбор кандидатов на распараллеливание // Новосибирск: Институт систем информатики им. А.П. Ершова СО РАН, 1999. 48 с.

16. Арыков С.Б., Малышкин В.Э. Система асинхронного параллельного программирования «Аспект» // Вычислительные методы и программирование. 2008. Т. 9, № 1. с. 48–52.

17. Бабичев А.В., Лебедев В.Г. Распараллеливание программных циклов // Программирование. 1983. № 5. с. 52–63.

18. Lamport L. The Coordinate Method for the parallel execution of DO loops // Sagamore Computer Conference on Parallel Processing. 1973. P. 1–12.

19. The parallel execution of DO loops // Commun.ACM. 1974. Vol. 17, № 2. P. 83–93.

20. Баханович С.В., Лиходед Н.А. Построение параллельных вычислительных процессов на основе гексагонального тайлинга // Информационные системы и технологии = Information Systems and Technologies: материалы междунар. науч. конгресса по информатике. В 3 ч. Ч. 2. Респ. Беларусь, Минск, 27–28 окт. 2022 г. / Белорус. гос. ун-т ; редкол.: С.В. Абламейко (гл. ред.) [и др.]. Минск: БГУ, 2022. с. 317–322.

21. Лиходед Н.А., Толстиков А.А. Формализованное получение коммуникационных операций параллельных зернистых алгоритмов // Вес. Нац. акад. навук Беларусі. Сер. фіз.-мат. навук. 2018. Т. 54, № 1. с. 50–61.

22. Goto K., Geijn van de R. A. Anatomy of High-Performance Matrix Multiplication // ACM Transactions on Mathematical Software. 2008. Vol. 34, № 3. P. 1–25.

23. Евстигнеев В.А., Касьянов В.Н. Оптимизирующие преобразования в распараллеливающих компиляторах // Программирование. 1996. № 6. с. 12–26.

24. Вальковский В.А. Распараллеливание алгоритмов и программ. Структурный подход. М.: Радио и связь, 1989. 176 с.

25. Евстигнеев В.А., Мирзуитова И.А. Анализ циклов: выбор кандидатов на распараллеливание. Новосибирск: Препринт ИСИ РАН, 1999. № 58. 41 с.

26. Французов Ю.А. Обзор методов распараллеливания кода и программной конвейеризации // Программирование. 1992. № 3. с. 16–37.

27. Французов Ю.А. Обзор методов распараллеливания кода и программной конвейеризации // Программирование. 1992. № 3. с. 16–37.

28. Штейнберг Б.Я., Штейнберг О.Б. Преобразования программ – фундаментальная основа создания оптимизирующих распараллеливающих компиляторов // Программные системы: теория и приложения. 2021. Т. 12, № 1. с. 21–113.

29. Ершов А.П. Введение в теоретическое программирование (Беседы о методе): учебное пособие. М.: Наука, 1977. 288 с.

30. Касьянов В.Н. Оптимизирующие преобразования программ. М.: Наука, 1988. 336 с.

31. Muchnick S. Advanced Compiler Design Implementation. San Diego, 1997. 888 p.

32. Ахо А.В., Лам М.С., Сети Р., Ульман Дж.Д. Компиляторы: принципы, технологии и инструментарий = Compilers: Principles, Techniques, and Tools. 2 изд. М.: Вильямс, 2008. 1184 с.

33. Меткалф М. Оптимизация в фортране / Под ред. Ю.М. Баяковского. М.: Мир, 1985. 264 с.

34. ParaWise Parallelizing System. [Электронный ресурс]: URL: http://www.parallelsp.com/ (дата обращения: 01.04.2024).

35. Алексахин В.Ф., Ильяков В.Н., Коновалов Н.А., Ковалева Н.В., Крюков В.А., Поддерюгина Н.В., Сазанов Ю.Л. Система автоматизации разработки параллельных программ для вычислительных кластеров и сетей (DVM-система) // Труды Всероссийской научной конференции «Научный сервис в сети Интернет» (г. Новороссийск, 23–28 сентября 2002 г.). М.: изд-во МГУ, 2002. с. 272–273.

36. Ильяков В.Н., Коваленко И.В., Крюков В.А. Анализ и предсказание эффективности выполнения DVM-программ на неоднородных сетях ЭВМ // Труды Всероссийской научной конференции «Научный сервис в сети Интернет» (22–27 сентября 2003 г.). М.: изд-во МГУ, 2003. с. 181–182.

37. Катаев Н., Колганов А. Дополнительное распараллеливание существующих программ MPI с использованием SAPFOR // PaCT 2021: Параллельные вычислительные технологии. Конспекты лекций по информатике. Т. 12942. Springer, Cham.

38. Malyshkin V.E., Perepelkin V.A. LuNA Fragmented Programming System, Main Functions and Peculiarities of Run-Time Subsystem // PaCT 2011: Parallel Computing Technologies. 2011. P. 53–61.

39. Belyaev N.A., Perepelkin V. Automated GPU support in LuNA fragmented programming system // Lecture Notes in Computer Science. 2017. Vol. 10421: Parallel Computing Technologies: 14 intern. conf.: PaCT 2017, Nizhny Novgorod. P. 272–277.

40. DVM-Система. [Электронный ресурс]: URL: http://dvm-system.org/ru/ (дата обращения 11.12.2019).

41. Arora R., Olaya J., Gupta M. A Tool for Interactive Parallelization // Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment. Atlanta, GA, USA, 2014. P. 1–8.

42. ROSE project. [Электронный ресурс]: URL: http://rosecompiler.org/ (дата обращения: 01.04.2024).

43. SUIF (Stanford University Intermediate Format). [Электронный ресурс]: URL: https://suif.stanford.edu/ (дата обращения: 01.04.2024).

44. Оптимизирующая распараллеливающая система. [Электронный ресурс]: URL: http://www.ops.rsu.ru/ (дата обращения: 01.04.2024).

45. Метелица Е.А., Морылев Р.И., Петренко В.В., Штейнберг Б.Я. Основанная на ОРС система обучения преобразованиям программ «Тренажер параллельного программиста» // Языки программирования и компиляторы – 2017. Всероссийская научная конференция памяти А.Л. Фуксмана. Ростов н/Д., 2017. 198 с.

46. Штейнберг Б.Я., Арутюнян О.Э., Бутов А.Э., Гуфан К.Ю., Морылев Р., Науменко С.А., Петренко В.В., Тузаев А., Черданцев Д.Н., Шилов М.В., Штейнберг Р.Б., Шульженко А.М. Обучающая распараллеливанию программа на основе ОРС // Материалы научно-методической конференции «Современные информационные технологии в образовании: Южный федеральный округ» (г. Ростов-на-Дону, 12–15 мая 2004 г.). Ростов н/Д., 2004. с. 248–250.

47. Штейнберг Б.Я., Нис З.Я., Петренко В.В., Черданцев Д.Н., Штейнберг Р.Б., Шульженко А.М. Открытая распараллеливающая система // Труды III международной конференции «Параллельные вычисления и задачи управления» (г. Москва, 2–4 октября 2006 г.). М.: ИПУ РАН, 2006. с. 526–541.

48. Состояние и возможности открытой распараллеливающей системы // Труды семинара «Наукоемкое программное обеспечение» в рамках шестой международной конференции памяти академика А.П. Ершова «Перспективы систем информатики» (28–29 июня 2006 г.). Новосибирск, 2006. с. 122–125.

49. Штейнберг Б.Я. Открытая распараллеливающая система // Открытые системы. 2007. № 9. с. 36–41.

50. Gervich L.R., Guda S.A., Dubrov D.V., Ibragimov R.A., Metelitsa E.A., Steinberg B.Ya. How OPS (Optimizing Parallelizing System) May be Useful for Clang // CEE-SECR '17 Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia. 2017. P. 1–8.

51. Irigoin F., Triolet R. Supernode Partitioning // POPL'88. 1988. P. 319–329.

52. Wolfe M.E., Lam M. A Data Locality Optimizing Algorithm // PLDI'91. 1991. P. 30–44.

53. Lam M.S., Rothberg E.E., Wolf M.E. The cache performance and optimizations of blocked algorithms // Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems. 1991. P. 63–74.

54. Lim A.W., Lam M.S. Cache Optimizing with Affine Partitioning // Proceedings of the Tenth SIAM Conference on Parallel Processing for Scientific Computing. Portsmouth, Virginia, 2001. P. 14.

55. Wolfe M. Iteration space tiling for memory hierarchies // Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing. 1987. P. 357–361.

56. Wolf M.E., Lam M.S. A Loop Transformation Theory and an Algorithm to Maximize Parallelism // IEEE Transactions on Parallel and Distributed Systems. 1991. Vol. 2, № 4. P. 452–471.

57. Wolfe M. More Iteration Space Tiling // Supercomputing'89. 1989. P. 655–664.

58. Ammaev S.G., Gervich L.R., Steinberg B.Y. Combining Parallelization with Overlaps and Optimization of Cache Memory Usage // Parallel Computing Technologies: 14th International Conference: PaCT 2017. Nizhny Novgorod, 2017. P. 257–264.

59. Perepelkina A.Yu., Levchenko V.D. DiamondTorre algorithm for high-performance wave modeling. Keldysh Institute Preprints, 2015. 20 p.

60. Perepelkina A.Yu., Levchenko V.D. TheDiamondCandy Algorithm for Maximum Performance Vectorized Cross-Stencil Computation. Keldysh Institute Preprints, 2018. 24 p.

61. Zakirov A.V., Levchenko V.D., Perepelkina A.Yu., Yasunari Z. High performance FDTD code implementation for GPGPU supercomputers. Keldysh Institute Preprints, 2016. 22 p.

62. Корнеев Б.А., Левченко В.Д. Эффективное решение трёхмерных задач газовой динамики Рунге – Кутты разрывным методом Галеркина // Журнал вычислительной математики и математической физики. 2016. Т. 56, № 3. с. 465–475.

63. Ahmed N., Mateev N., Pingali K. Tiling Imperfectly-nested Loop Nests. Department of Computer Science, Cornell University, Ithaca, NY, 2000. 31 p.

64. McCormick D. Applying the Polyhedral Model to Tile Time Loops in Devito. 2017. [Электронный ресурс]: URL: https://arxiv.org/abs/1707.02347 (дата обращения: 01.04.2024).

65. Rivera G., Tseng C.-W. Tiling optimizations for 3D scientific computations // Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (SC’00). Dallas, TX, USA, 2000. P. 32–32.

66. Liu J., Zhang Y., Ding W., Kandemir M. On-chip cache hierarchy-aware tile scheduling for multicore machines // Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO’11. Washington, DC, USA: IEEE Computer Society, 2011. P. 161–170.

67. Leung A., Vasilache N., Meister B., Baskaran M., Wohlford D., Bastoul C., Lethin R. A mapping path for multi-gpgpu accelerated computers from a portable high level programming

abstraction // Proceedings of the 3rd Workshop on GeneralPurpose Computation on Graphics Processing Units, GPGPU'10. New York, NY, USA: ACM, 2010. P. 51–61.

68. Carter L., Ferrante J., Hummel S.F. Hierarchical tiling for improved superscalar performance // Proceedings of the 9th International Symposium on Parallel Processing, IPPS'95. Washington, DC, USA: IEEE Computer Society, 1995. P. 239–245.

69. Carter L., Ferrante J., Hummel S.F., Alpern B., Gatlin K.-S. Hierarchical tiling: A methodology for high performance // Proceedings of 9th International Parallel Processing Symposium. Santa Barbara, CA, USA, 1995. P. 239–245.

70. Hartono A., Baskaran M.M., Bastoul C., Cohen A., Krishnamoorthy S., Norris B., Ramanujam J., Sadayappan P. Parametric multi-level tiling of imperfectly nested loops // Proceedings of the 23rd International Conference on Supercomputing, ICS'09. New York, NY, USA: ACM, 2009. P. 147–157.

71. Metelitsa E., Steinberg B., Gervich L. Combination of parallelization and skewed tiling // [Принято в печать].

72. Krishnamoorthy S., Baskaran M., Bondhugula U., Ramanujam J., Rountev A., Sadayappan P. Effective Automatic Parallelization of Stencil Computations // PLDI '07 Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. San Diego, California, USA, 2007. P. 235–244.

73. Zhao J., Cohen A. Flextended tiles: A flexible extension of overlapped tiles // ACM Transactions on Architecture and Code Optimization. 2019. Vol. 16, № 4. P. 1–25.

74. Zhou X., Giacalone J.P., Garzarán M.J., Kuhn R.H., Ni Y., Padua D. Hierarchical Overlapped Tiling // Proceedings of the International Symposium on Code Generation and Optimization, CGO. 2012. P. 207–218.

75. Ripeanu M., Iamnitchi A., Foster, I.T. Cactus application: Performance predictions in grid environments // Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing, Euro-Par '01. London, UK: Springer-Verlag, 2001. P. 807–816.

76. Meng J., Skadron K. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs // Proceedings of the 23rd International Conference on Supercomputing, ICS '09. New York, NY, USA: ACM, 2009. P. 256–265.

77. GCC, the GNU Compiler Collection. [Электронный ресурс]: URL: https://gcc.gnu.org/ (дата обращения: 11.12.2019).

78. The LLVM Compiler Infrastructure. [Электронный ресурс]: URL: https://llvm.org/ (дата обращения: 11.12.2019).

79. Clang: a C language family frontend for LLVM. [Электронный ресурс]: URL: http://clang.llvm.org/ (дата обращения: 11.12.2019).

80. Kruse M., Finkel H. Loop Optimizations in LLVM: The Good, The Bad, and The Ugly. [Электронный ресурс]: URL: https://llvm.org/devmtg/2018-10/slides/Kruse-LoopTransforms.pdf (дата обращения: 11.12.2019).

81. Polly. LLVM Framework for High-Level Loop and Data-Locality Optimizations. [Электронный ресурс]: URL: https://polly.llvm.org/ (дата обращения: 11.12.2019).

82. Grosser T., Groslinger A., Lengauer C. Polly – performing polyhedral optimizations on a low-level intermediate representation // Parallel Processing Letters. 2012. Vol. 22, № 4. P. 1–27.

83. Nvidia. [Электронный ресурс]: URL: https://docs.nvidia.com/hpc-sdk/pgi-compilers/19.10/x86/pgi-user-guide/index.htm#opt-overview (дата обращения: 01.04.2024).

84. Bondhugula U., Hartono A., Ramanujam J., Sadayappan P. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer // ACM SIGPLAN Notices. 2008. Vol. 43, № 6. P. 101–113.

85. Bondhugula U. Effective Automatic Parallelization and Locality Optimization Using The Polyhedral Model. Ph.D. Dissertation. Ohio State University, USA, 2008. 190 p.

86. Bondhugula U., Baskaran M., Krishnamoorthy S., Ramanujam J., Rountev A., Sadayappan P. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model // Proceedings of the International Conference on Compiler Construction (ETAPS CC). Budapest, Hungary, 2008. P. 132–146.

87. Bondhugula U., Acharya A., Cohen A. The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests // ACM Trans. Program. Lang. Syst. 2016. Vol. 38, № 3. P. 1–32.

88. Bondhugula U., Bandishti V., Pananilath I. Diamond Tiling: Tiling Techniques to Maximize Parallelism for Stencil Computations // IEEE Transactions on Parallel and Distributed Systems (TPDS). 2017. Vol. 28, № 5. P. 1285–1298.

89. Narasimhan K., Acharya A., Baid A., Bondhugula U. A practical tile size selection model for affine loop nests // Proceedings of the ACM International Conference on Supercomputing. Association for Computing Machinery, 2021. P. 27–39.

90. Mullapudi R.T., Vasista V., Bondhugula U. Polymage: Automatic optimization for image processing pipelines // SIGARCH Comput. Archit. News. 2015. Vol. 43, № 1. P. 429–443.

91. Baghdadi R., Ray J., Romdhane M., Del Sozzo E., Akkas A., Zhang Y., Suriana P., Kamil S., Amarasinghe S. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code // Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization.

2018. [Электронный ресурс]: URL: https://arxiv.org/pdf/1804.10694.pdf (дата обращения: 10.12.2019).

92. Christen M., Schenk O., Burkhart H. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures // 2011 IEEE International Parallel & Distributed Processing Symposium. Anchorage, AK, USA, 2011. P. 676–687.

93. Christen M., Schenk O., Neufeld E., Paulides M., Burkhart H. Manycore Stencil Computations in Hyperthermia Applications // Scientific Computing with Multicore and Accelerators. CRC Press, 2010. P. 255–277.

94. Christen M., Schenk O., Neufeld E., Messmer P., Burkhart H. Parallel Data-Locality Aware Stencil Computations on Modern Micro-Architectures // IEEE International Parallel & Distributed Processing Symposium (IPDPS). Rome, Italy, 2009. P. 1–10.

95. Reuter M., Biasotti S., Giorgi D., Patanè G., Spagnuolo M. Discrete Laplace–Beltrami operators for shape analysis and segmentation. Computers & Graphics. 2009. Vol. 33, № 3. P. 381–390.

96. Коротяев Е., Сабурова Н. Спектральные оценки для оператора Шрёдингера на периодических дискретных графах // Алгебра и анализ. 2018. Т. 30, № 4. с. 61–106.

97. Karam R. Schrödinger's original struggles with a complex wave function // American Journal of Physics. 2020. Vol. 88, № 6. P. 433–438.

98. Рабинович В.С. Существенный спектр операторов Шрёдингера на периодических графах // Функциональный анализ и его приложения. 2018. Т. 52, № 1. с. 66–69.

99. Cattaneo R., Natale G., Sicignano C., Sciuto D., Santambrogio M.D. On How to Accelerate Iterative Stencil Loops // ACM Transactions on Architecture and Code Optimization. 2015. Vol. 12, № 4. P. 1–26.

100. Zhiyuan L., Yonghong S. Automatic tiling of iterative stencil loops // ACM Trans. Program. Lang. Syst. 2004. Vol. 26, № 6. P. 975–1028.

101. Lim A.W., Lam M.S. Maximizing Parallelism and Minimizing Synchronization with Affine Partitions // Parallel Computing. 1998. Vol. 24. P. 445–475.

102. Климов А.В. К автоматическому порождению программ трафаретных вычислений с улучшенной временной локальностью // Программные системы: теория и приложения. 2018. Т. 9, № 4(39). с. 493–508.

103. Роганов В.А., Осипов В.И., Матвеев Г.А. Решение задачи Дирихле для уравнения Пуассона методом Гаусса – Зейделя на языке параллельного программирования T++ // Программные системы: теория и приложения. 2016. Т. 7, № 3(30). с. 99–107.

104. Волохов В.М., Мартыненко С.И., Токталиев П.Д., Яновский Л.С., Волохов А.В. Новые подходы к построению высокоэффективных параллельных алгоритмов для численного

решения краевых задач на структурированных сетках // Вычислительные методы и программирование. 2016. Т. 17, № 47. с. 72–80.

105. Sozzo E., Conficconi D., Santambrogio M.D., Sano K. Senju: A Framework for the Design of Highly Parallel FPGA-based Iterative Stencil Loop Accelerators // Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '23). New York, NY, USA: Association for Computing Machinery, 2023. 233 p.

106. Tian X., Ye Z., Lu A., Guo L., Chi Y., Fang Z. SASA: A Scalable and Automatic Stencil Acceleration Framework for Optimized Hybrid Spatial and Temporal Parallelism on HBM-based FPGAs // ACM Transactions on Reconfigurable Technology and Systems. 2023. Vol. 16, № 2. P. 1–33.

107. Matsumura K., Zohouri H.R., Wahib M., Endo T., Matsuoka S. AN5D: automated stencil framework for high-degree temporal blocking on GPUs // Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2020). New York, NY, USA: Association for Computing Machinery, 2020. P. 199–211.

108. Li M., Liu Y., Yang H., Hu Y., Sun Q., Chen B., You X., Liu X., Luan Z., Qian D. Automatic Code Generation and Optimization of Large-scale Stencil Computation on Many-core Processors // Proceedings of the 50th International Conference on Parallel Processing (ICPP '21). New York, NY, USA: Association for Computing Machinery, 2021. P. 1–12.

109. Allen J.R., Kennedy K. Automatic Loop Interchange // Proceedings of the ACM SIGPLAN ‘84 Symposium on Compiler Construction. Montreal, Canada, 1984. Vol. 19, № 6. P. 233–246.

110. Савельев В.А., Штейнберг Б.Я. Распараллеливание программ: учебник. Ростов н/Д.: Изд-во ЮФУ, 2008. 192 с.

111. Арапбаев Р.Н. Анализ зависимостей по данным: тесты на зависимость и стратегии тестирования: дис. … кандидата физико-математических наук: 05.13.11. Новосибирск: ИСИ СО РАН, 2008. 116 с.

112. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. 599 с.

113. Воеводин В.В. Математические модели и методы в параллельных процессах. М.: Наука, 1986. 296 с.

114. Воеводин В.В. Математические основы параллельных вычислений. М.: Изд-во МГУ, 1991. 345 с.

115. Feautrier P. Some efficient solutions to the affine scheduling problem. Part 1. One-dimensional time // International journal of Parallel Programming. 1992. Vol. 21, № 5. P. 313–347.

116. Feautrier P. Some efficient solutions to the affine scheduling problem. Part 2. Multidimensional time // Int J Parallel Prog. 1992. Vol. 21. P. 389–420.

117. Шульженко А.М. Исследование информационных зависимостей программ для анализа распараллеливающих преобразований: дис. … кандидата технических наук: 05.13.11. Ростов н/Д.: Изд-во ЮФУ, 2006. 202 с.

118. Штейнберг Б.Я. О взаимосвязи между решетчатым графом программы и графом информационных связей // Известия высших учебных заведений. Северо-Кавказский регион. Серия: Естественные науки. 2011. № 5(165). с. 28–30.

119. Maydan D.E., Hennessy J.L., Lam M.S. Efficient and exact data dependence // ACM SIGPLAN Notices. 1991. Vol. 26, № 6. P. 1–14.

120. Ramanujam, J. Beyond unimodular transformations // The Journal of Supercomputing. 1995. Vol. 9, № 4. P. 365–389.

121. Gottlieb A., Banerjee U., Bilardi G., Pucci G., Carlson W., Merkey P. Unimodular Transformations // Encyclopedia of Parallel Computing. Springer, Boston, MA, 2011. P. 2103–2112.

122. Wolfe M., Banerjee U. Data Dependence and its Application to Parallel Processing // International Journal of Parallel Programming. 1987. Vol. 16, № 2. P. 137–178.

123. Wolfe M. Loop skewing: the wavefront method revisited // Int J Parallel Program. 1986. Vol. 15, № 4. P. 279–293.

124. Петренко В.В. Внутреннее представление REPRISE распаралелливающей системы // Труды 4-й Международной конференции «Параллельные вычисления и задачи управления» (PACO’2008, Москва). [Электронный ресурс]: URL: https://ops.rsu.ru/download/works/PACO2008-petrenko.pdf (дата обращения: 01.04.2024).

125. Василенко А.А., Метелица Е.А., Штейнберг Б.Я. Свидетельство о государственной регистрации программ «Программа, реализующая алгоритм Гаусса – Зейделя для задачи Дирихле» № 2021681898 от 10 января 2022 г.

126. Allen J.R., Kennedy K. Automatic Translation of Fortran Programs to Vector Form // ACM Transactions on Programming Languages and Systems. 1987. Vol. 9, № 4. P. 491–542.

127. Баглий А.П., Кривошеев Н.М., Штейнберг Б.Я., Штейнберг О.Б. Преобразования программ в Оптимизирующей распараллеливающей системе для распараллеливания на распределённую память // Инженерный вестник Дона. 2023. № 12(96). с. 266–285.

128. Баглий А.П., Кривошеев Н.М., Штейнберг Б.Я. Автоматизация распараллеливания программ с оптимизацией пересылок данных // Научный сервис в сети Интернет: труды XXIV Всероссийской научной конференции (19–22 сентября 2022 г., онлайн). М.: ИПМ им. М.В. Келдыша, 2022. с. 81–92.

129. Vasilenko A., Veselovskiy V., Metelitsa E., Zhivykh N., Steinberg B., Steinberg O. Precompiler for the ACELAN-COMPOS Package Solvers // PaCT 2021: Parallel Computing Technologies. 2021. P. 103–116.

130. Лиходед Н.А., Толстиков А.А. Метод оценки локальности параллельных алгоритмов, ориентированных на компьютеры с распределённой памятью // Доклады НАН Беларуси. 2020. Т. 64, № 6. с. 647–656.

131. Lebedev A.S., Magomedov S.G. Automatic Parallelization of Affine Programs for Distributed Memory Systems // FTNCT 2020: Futuristic Trends in Network and Communication Technologies. 2021. P. 91–101.

132. Feautrier P. Toward automatic distribution // Parallel Processing Letters. 1994. Vol. 4, № 3. P. 233–244.

133. Feautrier P. Automatic distribution of data and computations. 2000. [Электронный ресурс]: URL: https://www.researchgate.net/publication/324171325_Automatic_Distribution_of_Data_and_Computations (дата обращения: 01.04.2024).

134. Метелица Е.А. Обоснование методов ускорения гнёзд циклов итерационного типа // Программные системы: теория и приложения. 2024. Т. 15, № 1. с. 63–94.

135. Абу-Халил Ж.М., Морылев Р.И., Штейнберг Б.Я. Параллельный алгоритм глобального выравнивания с оптимальным использованием памяти // Современные проблемы науки и образования. 2013. № 1. [Электронный ресурс]: URL: https://science-education.ru/ru/article/view?id=8139 (дата обращения: 01.04.2024).

136. Абу-Халил Ж.М., Гуда С.А., Штейнберг Б.Я. Перенос параллельных программ с сохранением эффективности // Открытые системы. СУБД. 2015. № 4. с. 18–19.

137. Денисова Э.В., Кучер А.В. Основы вычислительной математики. СПб.: СПбГУ ИТМО, 2010. 164 с.

138. Самарский А.А., Николаев Е.С. Методы решения сеточных уравнений. М.: Наука, 1978. 592 с.

139. Котина Е.Д. О сходимости блочных итерационных методов // Известия Иркутского государственного университета. Серия: Математика. 2012. Т. 5, № 3. с. 41–55.

140. Макошенко Д.В. Аналитическое предсказание времени исполнения программы и основанные на нём методы оптимизации: дис. … кандидата физико-математических наук: 05.13.11. Новосибирск, 2011. 122 с.

141. Haghighat M.R. Symbolic analysis for parallelizing compilers. Springer New York, NY, 1995. 138 p.

142. Морылев Р.И., Шаповалов В.Н., Штейнберг Б.Я. Символьный анализ в диалоговом распараллеливании программ // Информационные технологии. 2013. № 2. с. 33–36.

143. Баглий А.П., Дубров Д.В., Штейнберг Б.Я., Штейнберг Р.Б. Повторное использование ресурсов при конвейерных вычислениях // Научный сервис в сети Интернет: труды XIX Всероссийской научной конференции (18–23 сентября 2017 г., г. Новороссийск). М.: ИПМ им. М.В. Келдыша, 2017. с. 43-47.

144. Дроздов А.Ю. Компонентный подход к построению оптимизирующих компиляторов: автореф. дис. … доктора технических наук: 05.13.11. М., 2010. 50 с.

145. Макаров А.М., Лунева Л.А. Уравнение Пуассона для потенциала электростатического поля. Понятие о краевых задачах в теории потенциала и методах их решения // Физика в техническом университете. Т. 3. [Электронный ресурс]: URL: http://fn.bmstu.ru/data-physics/library/physbook/tom3/ch1/texthtml/ch1_5.htm (дата обращения: 12.06.2024).

146. Филимонов М.Ю., Ваганова Н.А. Моделирование тепловых полей в вечномерзлых грунтах у устья скважин при различных режимах их эксплуатации // XII Всероссийский съезд по фундаментальным проблемам теоретической и прикладной механики: сборник трудов в 4-х томах, Уфа, 19–24 августа 2019 года. Т. 4. – Уфа: Башкирский государственный университет, 2019. – С. 413-415.

147. Самарский А.А., Моисеенко Б.Д. Экономичная схема сквозного счета для многомерной задачи Стефана // Журнал вычисл. матем. и матем. физ., т. 5, №5, 1965, с. 816–827.

148. Метелица Е.А., Штейнберг Б.Я. Об ускоряющих преобразованиях программ для решения обобщенной задачи Дирихле // Вычислительные методы и программирование. 2024.25, № 3. С. 292-301.

Приложения

Приложение А. Свидетельство о государственной регистрации программы для ЭВМ

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2022610227

Программа реализующая параллельный алгоритм Гаусса-Зейделя для задачи Дирихле

Правообладатель: ***федеральное государственное автономное образовательное учреждение высшего образования «Южный федеральный университет» (RU)***

Авторы: ***Василенко Александр Александрович (RU), Метелица Елена Анатольевна (RU), Штейнберг Борис Яковлевич (RU)***

Заявка № **2021681898**

Дата поступления **28 декабря 2021 г.**

Дата государственной регистрации в Реестре программ для ЭВМ ***10 января 2022 г.***

Руководитель Федеральной службы по интеллектуальной собственности

ДОКУМЕНТ ПОДПИСАН ЭЛЕКТРОННОЙ ПОДПИСЬЮ
Сертификат 0x75699C0D08AEB78543BD83DF819A6CD1
Владелец **Ивлиев Григорий Петрович**
Действителен с 24.12.2021 по 24.12.2022

Г.П. Ивлиев

Приложение Б. Сертификат участника Молодёжной конференции студентов и аспирантов в рамках «НСКФ-2018»

НАЦИОНАЛЬНЫЙ СУПЕРКОМПЬЮТЕРНЫЙ ФОРУМ
НСКФ
МОЛОДЁЖНАЯ КОНФЕРЕНЦИЯ ДЛЯ СТУДЕНТОВ И АСПИРАНТОВ

ФОНД ПРЕЗИДЕНТСКИХ ГРАНТОВ

Молодёжная конференция студентов и аспирантов
в рамках «НСКФ-2018»
Проводится при поддержке Фонда президентских грантов

СЕРТИФИКАТ УЧАСТНИКА

Настоящий Сертификат выдан

Метелице Елене Анатольевне,

принявше**й** активное участие в Первой молодёжной конференции для студентов и аспирантов в рамках «Национального суперкомпьютерного форума – 2018», проводимой при поддержке Фонда президентских грантов.

Россия, г. Переславль-Залесский
ИПС имени А.К. Айламазяна РАН
26 ноября 2018 г.

Председатель оргкомитета, профессор, д.ф.-м.н. **Н. Н. Непейвода**

Ответственный секретарь, к.т.н. **И. Н. Григоревский**

АНО «НСКФ» АНО «Национальный суперкомпьютерный форум»

Институт программных систем имени А.К. Айламазяна РАН

Национальная Суперкомпьютерная Технологическая Платформа (НСТП)

Приложение В. Сертификат участника летней школы Intel “Intel Summer Internship”

Приложение Г. Диплом победителя полуфинала Студенческой лиги Международного инженерного чемпионата “CASE-IN”

Приложение Д. Конфигурация ЭВМ

CPU: i7-9700; 3,00 GHz.

Кэш-память: L1 – 256 Kb; L2 – 2 Mb; L3 – 12 Mb.

Частота системной шины: 8 GT/s.

Макс. пропускная способность: 41.6 GB/s.

RAM: DDR4 16Gb.

Мин. частота: 1600 МГц.

Макс. частота: 2666 МГц.

Источник: https://ark.intel.com/content/www/ru/ru/ark/products/191792/intel-core-i7-9700-processor-12m-cache-up-to-4-70-ghz.html

Приложение Е. Гнездо циклов преобразованного с помощью OPC алгоритма Гаусса – Зейделя для решения задачи Дирихле (листинг 3.3.1); d1, d2, d3 – размеры тайлов

```
{
    int i;
    int j;
    int __uni28j;
    {
      int k;
      int __uni29__uni28j;
      int __uni30i;
      int __uni31__uni29__uni28j;
      int __uni32__uni30i;
      int __uni33k;
      int __uni45__uni42__uni31__uni29__uni28j;
      for (__uni45__uni42__uni31__uni29__uni28j=(0 + 0 * 1) + 0 * 1;
__uni45__uni42__uni31__uni29__uni28j < (((((((((M - 1) - 1) + ((((N - 1) - 1) + K
* 1) - K * 1) * 1) + K * 1) - 1) - 1) - 1) / d3 + 1) + (((((((N - 1) - 1) + K * 1)
- 1) - 1) / d2 + 1) - 1) * 1) + (K / d1 - 1) * 1;
__uni45__uni42__uni31__uni29__uni28j=__uni45__uni42__uni31__uni29__uni28j + 1)
      {
        int __uni42__uni31__uni29__uni28j;
        int __uni46__uni33k;
        int __uni47__uni33k;
        if (0 > ((__uni45__uni42__uni31__uni29__uni28j - (((((((((M - 1) - 1) +
((((N - 1) - 1) + K * 1) - K * 1) * 1) + K * 1) - 1) - 1) - 1) / d3 + 1) +
(((((((N - 1) - 1) + K * 1) - 1) - 1) / d2 + 1) - 1) * 1)) + 1) / 1)
        {
          __uni46__uni33k=0;
        }
        else
        {
          __uni46__uni33k=((__uni45__uni42__uni31__uni29__uni28j - (((((((((M - 1)
- 1) + ((((N - 1) - 1) + K * 1) - K * 1) * 1) + K * 1) - 1) - 1) - 1) / d3 + 1) +
(((((((N - 1) - 1) + K * 1) - 1) - 1) / d2 + 1) - 1) * 1)) + 1) / 1;
        }
        if (K / d1 < ((__uni45__uni42__uni31__uni29__uni28j - 0) + 1) / 1)
        {
          __uni47__uni33k=K / d1;
        }
        else
        {
          __uni47__uni33k=((__uni45__uni42__uni31__uni29__uni28j - 0) + 1) / 1;
        }
        #pragma omp parallel num_threads(thread_count) private (i, j, k, __uni28j,
__uni29__uni28j, __uni31__uni29__uni28j, __uni30i, __uni32__uni30i,__uni33k)
firstprivate( __uni45__uni42__uni31__uni29__uni28j)
        for (__uni33k=__uni46__uni33k; __uni33k < __uni47__uni33k;
__uni33k=__uni33k + 1)
        {
          int __uni43__uni32__uni30i;
          int __uni44__uni32__uni30i;
```

```
          __uni42__uni31__uni29__uni28j=__uni45__uni42__uni31__uni29__uni28j -
__uni33k * 1;
          if (0 > ((__uni42__uni31__uni29__uni28j - (((((((M - 1) - 1) + ((((N -
1) - 1) + K * 1) - K * 1) * 1) + K * 1) - 1) - 1) - 1) / d3 + 1)) + 1) / 1)
          {
            __uni43__uni32__uni30i=0;
          }
          else
          {
            __uni43__uni32__uni30i=((__uni42__uni31__uni29__uni28j - (((((((M -
1) - 1) + ((((N - 1) - 1) + K * 1) - K * 1) * 1) + K * 1) - 1) - 1) - 1) / d3 +
1)) + 1) / 1;
          }
          if ((((((N - 1) - 1) + K * 1) - 1) - 1) / d2 + 1 <
((__uni42__uni31__uni29__uni28j - 0) + 1) / 1)
          {
            __uni44__uni32__uni30i=(((((N - 1) - 1) + K * 1) - 1) - 1) / d2 + 1;
          }
          else
          {
            __uni44__uni32__uni30i=((__uni42__uni31__uni29__uni28j - 0) + 1) / 1;
          }
          #pragma omp for
          for (__uni32__uni30i=__uni43__uni32__uni30i; __uni32__uni30i <
__uni44__uni32__uni30i; __uni32__uni30i=__uni32__uni30i + 1)
          {
            int __uni36__uni29__uni28j;
            int __uni37__uni29__uni28j;
            __uni31__uni29__uni28j=__uni42__uni31__uni29__uni28j - __uni32__uni30i
* 1;
            if ((0 + (__uni32__uni30i * d2 - (__uni33k * d1) * 1) * 1) + (__uni33k
* d1) * 1 > __uni31__uni29__uni28j * d3)
            {
              __uni36__uni29__uni28j=(0 + (__uni32__uni30i * d2 - (__uni33k * d1)
* 1) * 1) + (__uni33k * d1) * 1;
            }
            else
            {
              __uni36__uni29__uni28j=__uni31__uni29__uni28j * d3;
            }
            if ((((M - 1) - 1) + ((__uni32__uni30i + 1) * d2 - ((__uni33k + 1) *
d1) * 1) * 1) + ((__uni33k + 1) * d1) * 1 < (__uni31__uni29__uni28j + 1) * d3)
            {
              __uni37__uni29__uni28j=(((M - 1) - 1) + ((__uni32__uni30i + 1) * d2
- ((__uni33k + 1) * d1) * 1) * 1) + ((__uni33k + 1) * d1) * 1;
            }
            else
            {
              __uni37__uni29__uni28j=(__uni31__uni29__uni28j + 1) * d3;
            }
            for (__uni29__uni28j=__uni36__uni29__uni28j; __uni29__uni28j <
__uni37__uni29__uni28j; __uni29__uni28j=__uni29__uni28j + 1)
            {
              int __uni34__uni30i;
```

```
              int __uni35__uni30i;
              int __uni40k;
              int __uni41k;
              if ((__uni29__uni28j + 1) / 1 < (__uni33k + 1) * d1)
              {
                __uni41k=(__uni29__uni28j + 1) / 1;
              }
              else
              {
                __uni41k=(__uni33k + 1) * d1;
              }
              if ((((__uni29__uni28j - ((M - 1) - 1)) - (__uni32__uni30i + 1) *
d2) + 1) / 1 > __uni33k * d1)
              {
                __uni40k=(((__uni29__uni28j - ((M - 1) - 1)) - (__uni32__uni30i +
1) * d2) + 1) / 1;
              }
              else
              {
                __uni40k=__uni33k * d1;
              }
              for (k=__uni40k; k < __uni41k; k=k + 1)
              {
                int __uni38__uni30i;
                int __uni39__uni30i;
                if (0 + k * 1 > __uni32__uni30i * d2)
                {
                  __uni34__uni30i=0 + k * 1;
                }
                else
                {
                  __uni34__uni30i=__uni32__uni30i * d2;
                }
                if (((N - 1) - 1) + k * 1 < (__uni32__uni30i + 1) * d2)
                {
                  __uni35__uni30i=((N - 1) - 1) + k * 1;
                }
                else
                {
                  __uni35__uni30i=(__uni32__uni30i + 1) * d2;
                }
                if ((__uni29__uni28j + 1) / 1 < __uni35__uni30i)
                {
                  __uni39__uni30i=(__uni29__uni28j + 1) / 1;
                }
                else
                {
                  __uni39__uni30i=__uni35__uni30i;
                }
                if ((((__uni29__uni28j - ((M - 1) - 1)) - 0) + 1) / 1 >
__uni34__uni30i)
                {
                  __uni38__uni30i=(((__uni29__uni28j - ((M - 1) - 1)) - 0) + 1) /
1;
```

```
                }
                else
                {
                  __uni38__uni30i=__uni34__uni30i;
                }
                for (__uni30i=__uni38__uni30i; __uni30i < __uni39__uni30i;
__uni30i=__uni30i + 1)
                {
                  i=__uni30i - k * 1;
                  __uni28j=__uni29__uni28j - k * 1;
                  j=__uni28j - i * 1;
                  u[(1 + i)][(1 + j)]=(((u[((1 + i) - 1)][(1 + j)] + u[((1 + i) +
1)][(1 + j)]) + u[(1 + i)][((1 + j) - 1)]) + u[(1 + i)][((1 + j) + 1)]) / 4.;
                }
              }
            }
          }
        }
      }
    }
  }
```

Приложение Ж. Внутреннее гнездо преобразованного с помощью OPC алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле (листинг 3.6.1); d1, d2, d3 – размеры тайлов

```
int __uni36__uni29__uni28j;
int __uni37__uni29__uni28j;
__uni31__uni29__uni28j=__uni42__uni31__uni29__uni28j - __uni32__uni30i * 1;
if ((0 + (__uni32__uni30i * d2 - (__uni33k * d1) * 1) * 1) + (__uni33k * d1) * 1 >
__uni31__uni29__uni28j * d3) {
      __uni36__uni29__uni28j=(0 + (__uni32__uni30i * d2 - (__uni33k * d1) * 1) * 1) +
(__uni33k * d1) * 1;
} else {
      __uni36__uni29__uni28j=__uni31__uni29__uni28j * d3;
}
if ((((M - 1) - 1) + ((__uni32__uni30i + 1) * d2 - ((__uni33k + 1) * d1) * 1) * 1) +
((__uni33k + 1) * d1) * 1 < (__uni31__uni29__uni28j + 1) * d3) {
      __uni37__uni29__uni28j=(((M - 1) - 1) + ((__uni32__uni30i + 1) * d2 - ((__uni33k +
1) * d1) * 1) * 1) + ((__uni33k + 1) * d1) * 1;
} else {
__uni37__uni29__uni28j=(__uni31__uni29__uni28j + 1) * d3;
}
for (__uni29__uni28j=__uni36__uni29__uni28j; __uni29__uni28j < __uni37__uni29__uni28j;
__uni29__uni28j=__uni29__uni28j + 1) {
      int __uni34__uni30i;
      int __uni35__uni30i;
      int __uni40k;
      int __uni41k;
      if ((__uni29__uni28j + 1) / 1 < (__uni33k + 1) * d1) {
        __uni41k=(__uni29__uni28j + 1) / 1;
      } else {
        __uni41k=(__uni33k + 1) * d1;
      }
      if ((((__uni29__uni28j - ((M - 1) - 1)) - (__uni32__uni30i + 1) * d2) + 1) / 1 >
__uni33k * d1) {
            __uni40k=(((__uni29__uni28j - ((M - 1) - 1)) - (__uni32__uni30i + 1) * d2) +
      1) / 1;
      } else {
            __uni40k=__uni33k * d1;
      }
      for (k=__uni40k; k < __uni41k; k=k + 1) {
            int __uni38__uni30i;
            int __uni39__uni30i;
            if (0 + k * 1 > __uni32__uni30i * d2) {
                  __uni34__uni30i=0 + k * 1;
            } else {
                  __uni34__uni30i=__uni32__uni30i * d2;
            }
            if (((N - 1) - 1) + k * 1 < (__uni32__uni30i + 1) * d2) {
            __uni35__uni30i=((N - 1) - 1) + k * 1;
            } else {
            __uni35__uni30i=(__uni32__uni30i + 1) * d2;
           }
           if ((__uni29__uni28j + 1) / 1 < __uni35__uni30i) {
            __uni39__uni30i=(__uni29__uni28j + 1) / 1;
           } else {
            __uni39__uni30i=__uni35__uni30i;
            }
           if ((((__uni29__uni28j - ((M - 1) - 1)) - 0) + 1) / 1 > __uni34__uni30i){
```

```
            __uni38__uni30i=(((__uni29__uni28j - ((M - 1) - 1)) - 0) + 1) / 1;
           } else {
            __uni38__uni30i=__uni34__uni30i;
           }
           for (__uni30i=__uni38__uni30i; __uni30i < __uni39__uni30i; __uni30i=__uni30i +
1) {
                  i=__uni30i - k * 1;
                  __uni28j=__uni29__uni28j - k * 1;
                  j=__uni28j - i * 1;
                 u[(1 + i)][(1 + j)]=(((A[i+1][j+1]*u[((1 + i) - 1)][(1 + j)] +
           B[i+1][j+1]*u[((1 + i) + 1)][(1 + j)]) + C[i+1][j+1]*u[(1 + i)][((1 + j) - 1)])
           + D[i+1][j+1]*u[(1 + i)][((1 + j) + 1)] + y0[i+1][j+1]) / 5.0;
                  }
           }
      }
```

Приложение 3. Гнездо циклов, полученное с помощью PLUTO, для алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле

```
#define ceild(n,d)  (((n)<0) ? -((-(n))/(d)) : ((n)+(d)-1)/(d))
#define floord(n,d) (((n)<0) ? -((-(n)+(d)-1)/(d)) : (n)/(d))
#define max(x,y)    ((x) > (y)? (x) : (y))
#define min(x,y)    ((x) < (y)? (x) : (y))

 int t1, t2, t3, t4, t5, t6;
 int lb, ub, lbp, ubp, lb2, ub2;
 register int lbv, ubv;
if ((N >= 3) && (T >= 1)) {
  for (t1=0;t1<=floord(2*T+N-4,8);t1++) {
    lbp=max(ceild(t1,2),ceild(8*t1-T+1,8));
    ubp=min(min(floord(T+N-3,8),floord(8*t1+N+5,16)),t1);
#pragma omp parallel for private(lbv,ubv,t3,t4,t5,t6) num_threads(16)
    for (t2=lbp;t2<=ubp;t2++) {
      for (t3=max(ceild(8*t2-N-4,8),t1-t2);t3<=min(min(floord(T+N-3,8),floord(8*t2+N+4,8)),floord(8*t1-8*t2+N+5,8));t3++) {
        for (t4=max(max(8*t1-8*t2,8*t2-N+2),8*t3-N+2);t4<=min(min(min(T-1,8*t2+6),8*t3+6),8*t1-8*t2+7);t4++) {
          for (t5=max(8*t2,t4+1);t5<=min(8*t2+7,t4+N-2);t5++) {
            for (t6=max(8*t3,t4+1);t6<=min(8*t3+7,t4+N-2);t6++) {
              a[(-t4+t5)][(-t4+t6)]=(   A[(-t4+t5)][(-t4+t6)]*a[(-t4+t5)   -   1][(-t4+t6)] + B[(-t4+t5)][(-t4+t6)]* a[(-t4+t5)][(-t4+t6) - 1] + C[(-t4+t5)][(-t4+t6)]*a[(-t4+t5) + 1][(-t4+t6)] + D[(-t4+t5)][(-t4+t6)]*a[(-t4+t5)][(-t4+t6) + 1] + y[(-t4+t5)][(-t4+t6)]) / 5.0;;
            }
          }
        }
      }
    }
  }
}
```

Приложение И. Результаты оптимизации алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа. Размерность задачи – 256 x 4000 x 4000. Компилятор – GCC, опция компилятора – -O3. Время выполнения исходного алгоритма – 10,915 сек.

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=16, d2=16, d3=16	3,4474	3,1662	1,1746	9,2925	0,8266	13,2047
d1=16, d2=20, d3=20	3,2280	3,3814	1,0928	9,9881	0,7910	13,7990
d1=16, d2=25, d3=25	3,0174	3,6174	0,9994	10,9216	0,7090	15,3949
d1=16, d2=40, d3=40	2,8182	3,8730	1,0008	10,9063	0,6470	16,8702
d1=16, d2=50, d3=50	**2,7834**	**3,9215**	1,0132	10,7728	0,6092	17,9169
d1=16, d2=80, d3=80	3,0246	3,6087	1,0868	10,0433	0,6564	16,6286
d1=16, d2=100, d3=100	3,1858	3,4261	1,0936	9,9808	0,6678	16,3447
d1=16, d2=125, d3=125	3,2528	3,3556	1,1402	9,5729	0,7030	15,5263
d1=16, d2=200, d3=200	3,2982	3,3094	1,1946	9,1370	0,8344	13,0813
d1=16, d2=250, d3=250	3,1598	3,4543	1,1558	9,4437	0,7542	14,4723
d1=16, d2=400, d3=400	3,4860	3,1311	1,4580	7,4863	1,1326	9,6371
d1=32, d2=16, d3=16	3,3266	3,2811	1,0828	10,0804	0,7224	15,1094
d1=32, d2=20, d3=20	3,1062	3,5139	1,0172	10,7304	0,6782	16,0941
d1=32, d2=25, d3=25	2,9098	3,7511	1,0248	10,6509	0,6160	17,7192
d1=32, d2=40, d3=40	2,7586	3,9567	1,0580	10,3166	0,5896	18,5126
d1=32, d2=50, d3=50	**2,7064**	**4,0330**	**0,9702**	**11,2503**	**0,5654**	**19,3049**
d1=32, d2=80, d3=80	3,0096	3,6267	1,0010	10,9041	0,6202	17,5992
d1=32, d2=100, d3=100	3,0260	3,6071	1,0188	10,7136	0,6362	17,1566
d1=32, d2=125, d3=125	3,0470	3,5822	1,0258	10,6405	0,6430	16,9751
d1=32, d2=200, d3=200	3,0994	3,5216	1,0754	10,1497	0,7732	14,1167
d1=32, d2=250, d3=250	3,0708	3,5544	1,0736	10,1667	0,7260	15,0344
d1=32, d2=400, d3=400	3,4690	3,1464	1,4048	7,7698	1,0778	10,1271
d1=64, d2=16, d3=16	3,2512	3,3572	1,1240	9,7109	0,6650	16,4135
d1=64, d2=20, d3=20	3,0872	3,5356	1,0664	10,2354	0,6328	17,2487
d1=64, d2=25, d3=25	2,9752	3,6687	0,9894	11,0319	0,5800	18,8190
d1=64, d2=40, d3=40	2,8964	3,7685	1,0244	10,6550	0,5784	18,8710
d1=64, d2=50, d3=50	**2,7594**	**3,9556**	**0,9186**	**11,8822**	**0,5388**	**20,2580**
d1=64, d2=80, d3=80	2,9192	3,7390	0,9558	11,4198	0,6056	18,0235
d1=64, d2=100, d3=100	2,9752	3,6687	0,9580	11,3935	0,6108	17,8700
d1=64, d2=125, d3=125	3,0288	3,6037	0,9872	11,0565	0,6304	17,3144
d1=64, d2=200, d3=200	3,0598	3,5672	1,0480	10,4151	0,7514	14,5262
d1=64, d2=250, d3=250	3,1088	3,5110	1,0514	10,3814	0,7032	15,5219
d1=64, d2=400, d3=400	3,5062	3,1131	1,3998	7,7975	1,0686	10,2143

Приложение К. Результаты оптимизации алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа. Размерность задачи – 256 x 4000 x 4000. Компилятор – ICC, опция компилятора – -O3. Время выполнения исходного алгоритма – 16,835 сек.

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=32, d2=20, d3=20	2,9678	5,6724	0,8142	20,6762	0,6642	25,3457
d1=32, d2=25, d3=25	2,9496	5,7074	0,8276	20,3415	0,6766	24,8812
d1=32, d2=40, d3=40	2,5978	6,4803	0,7810	21,5552	0,5466	30,7988
d1=32, d2=50, d3=50	2,7562	6,1079	0,8496	19,8147	0,6004	28,0390
d1=32, d2=80, d3=80	3,0840	5,4587	0,9892	17,0184	0,6846	24,5904
d1=32, d2=100, d3=100	3,2308	5,2107	1,0324	16,3063	0,7338	22,9417
d1=32, d2=125, d3=125	3,2346	5,2045	1,0596	15,8877	0,7838	21,4782
d1=32, d2=200, d3=200	3,1516	5,3416	1,0998	15,3070	0,8216	20,4900
d1=32, d2=250, d3=250	3,1648	5,3193	1,1222	15,0014	0,8622	19,5252
d1=32, d2=400, d3=400	3,6180	4,6530	1,5412	10,9230	1,2664	13,2933
d1=32, d2=500, d3=500	5,2088	3,2320	2,1870	7,6976	2,2522	7,4747
d1=64, d2=20, d3=20	2,9184	5,7684	0,8738	19,2660	0,6888	24,4405
d1=64, d2=25, d3=25	2,9204	5,7645	0,8762	19,2132	0,6248	26,9440
d1=64, d2=40, d3=40	2,5368	6,6362	0,7772	21,6606	0,5348	31,4783
d1=64, d2=50, d3=50	2,6976	6,2406	0,8438	19,9509	0,5710	29,4827
d1=64, d2=80, d3=80	3,0372	5,5428	0,9840	17,1083	0,7006	24,0288
d1=64, d2=100, d3=100	3,1916	5,2747	1,0138	16,6054	0,7278	23,1308
d1=64, d2=125, d3=125	3,4060	4,9426	1,0512	16,0146	0,7524	22,3745
d1=64, d2=200, d3=200	3,1396	5,3620	1,0934	15,3966	0,8372	20,1082
d1=64, d2=250, d3=250	3,1690	5,3123	1,1136	15,1173	0,8190	20,5551
d1=64, d2=400, d3=400	3,6400	4,6249	1,5432	10,9089	1,2698	13,2577
d1=64, d2=500, d3=500	5,2898	3,1825	2,1990	7,6556	2,2658	7,4299
d1=128, d2=20, d3=20	2,8922	5,8207	0,8746	19,2483	0,6580	25,5845
d1=128, d2=25, d3=25	2,9106	5,7839	0,8880	18,9579	0,6312	26,6708
d1=128, d2=40, d3=40	2,6402	6,3763	0,7872	21,3854	0,5444	30,9232
d1=128, d2=50, d3=50	2,7586	6,1026	0,8482	19,8474	0,5974	28,1798
d1=128, d2=80, d3=80	3,2856	5,1238	0,9846	17,0979	0,7016	23,9946
d1=128, d2=100, d3=100	3,2106	5,2434	1,0130	16,6186	0,7248	23,2265
d1=128, d2=125, d3=125	3,2358	5,2026	1,0474	16,0728	0,7134	23,5977
d1=128, d2=200, d3=200	3,1542	5,3372	1,0902	15,4418	0,8298	20,2875
d1=128, d2=250, d3=250	3,1698	5,3109	1,1148	15,1010	0,8552	19,6850
d1=128, d2=400, d3=400	3,7302	4,5131	1,5792	10,6602	1,2752	13,2015
d1=128, d2=500, d3=500	5,3122	3,1690	2,2002	7,6514	2,2706	7,4142
d1=256, d2=20, d3=20	3,0240	5,5670	0,9146	18,4065	0,6268	26,8580

d1=256, d2=25, d3=25	3,0396	5,5384	0,9262	18,1760	0,6224	27,0479
d1=256, d2=40, d3=40	2,7650	6,0885	0,8472	19,8709	0,5616	29,9761
d1=256, d2=50, d3=50	2,8424	5,9227	0,8964	18,7802	0,6428	26,1895
d1=256, d2=80, d3=80	3,0868	5,4537	1,0036	16,7742	0,6780	24,8298
d1=256, d2=100, d3=100	3,2258	5,2187	1,0114	16,6448	0,7000	24,0494
d1=256, d2=125, d3=125	3,1932	5,2720	1,0484	16,0574	0,7372	22,8359
d1=256, d2=200, d3=200	3,1936	5,2714	1,1184	15,0524	0,8102	20,7783
d1=256, d2=250, d3=250	3,2650	5,1561	1,1594	14,5201	0,8432	19,9651
d1=256, d2=400, d3=400	3,7186	4,5271	1,5864	10,6118	1,2730	13,2244
d1=256, d2=500, d3=500	5,2646	3,1977	2,1862	7,7004	2,2570	7,4588

Приложение Л. Результаты оптимизации алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Лапласа. Размерность задачи – 256 x 4002 x 4002, тип данных – double. Компилятор – GCC, опция компилятора – -O3. Время выполнения исходного алгоритма – 11,2024 сек.

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=16, d2=16, d3=16	4,0824	2,7441	1,3646	8,2093	1,0682	10,4872
d1=16, d2=20, d3=20	3,8338	2,9220	1,2886	8,6935	1,0008	11,1934
d1=16, d2=25, d3=25	3,6778	3,0460	1,2500	8,9619	0,9466	11,8344
d1=16, d2=40, d3=40	3,2532	3,4435	1,1338	9,8804	0,8028	13,9542
d1=16, d2=50, d3=50	3,2082	3,4918	1,1574	9,6789	0,7810	14,3437
d1=16, d2=80, d3=80	3,2086	3,4914	1,1370	9,8526	0,7750	14,4547
d1=16, d2=100, d3=100	3,3654	3,3287	1,1670	9,5993	0,7616	14,7090
d1=16, d2=125, d3=125	3,4852	3,2143	1,2038	9,3059	0,7996	14,0100
d1=16, d2=200, d3=200	3,3556	3,3384	1,1858	9,4471	0,8962	12,4999
d1=16, d2=250, d3=250	3,2672	3,4287	1,1812	9,4839	0,8380	13,3680
d1=16, d2=400, d3=400	3,4904	3,2095	1,4406	7,7762	1,1154	10,0434
d1=32, d2=16, d3=16	3,7718	2,9700	1,2254	9,1418	0,8474	13,2197
d1=32, d2=20, d3=20	3,5692	3,1386	1,2316	9,0958	0,8042	13,9299
d1=32, d2=25, d3=25	3,6048	3,1076	1,1924	9,3948	0,7518	14,9008
d1=32, d2=40, d3=40	3,0754	3,6426	1,0596	10,5723	0,6490	**17,2610**
d1=32, d2=50, d3=50	2,9824	3,7562	1,0362	10,8110	0,6508	17,2133
d1=32, d2=80, d3=80	2,9540	**3,7923**	1,0152	**11,0347**	0,6770	16,5471
d1=32, d2=100, d3=100	3,0880	3,6277	1,0308	10,8677	0,6868	16,3110
d1=32, d2=125, d3=125	3,1786	3,5243	1,0612	10,5564	0,7166	15,6327
d1=32, d2=200, d3=200	3,1908	3,5108	1,1110	10,0832	0,8270	13,5458
d1=32, d2=250, d3=250	3,2028	3,4977	1,1176	10,0236	0,7650	14,6437
d1=32, d2=400, d3=400	3,5074	3,1939	1,4186	7,8968	1,0792	10,3803
d1=64, d2=16, d3=16	3,5796	3,1295	1,2208	9,1763	0,7484	14,9685
d1=64, d2=20, d3=20	3,3730	3,3212	1,1724	9,5551	0,7072	15,8405
d1=64, d2=25, d3=25	3,1998	3,5010	1,1090	10,1014	0,6880	16,2826
d1=64, d2=40, d3=40	**2,8518**	**3,9282**	**0,9956**	**11,2519**	**0,6000**	**18,6707**
d1=64, d2=50, d3=50	2,8840	3,8843	0,9690	11,5608	0,6032	18,5716
d1=64, d2=80, d3=80	2,9138	3,8446	0,9718	11,5275	0,6518	17,1869
d1=64, d2=100, d3=100	3,0376	3,6879	0,9874	11,3454	0,6622	16,9169
d1=64, d2=125, d3=125	3,1128	3,5988	1,0226	10,9548	0,6770	16,5471
d1=64, d2=200, d3=200	3,1602	3,5448	1,1412	9,8163	0,7870	14,2343
d1=64, d2=250, d3=250	3,2776	3,4179	1,1390	9,8353	0,7762	14,4324
d1=64, d2=400, d3=400	3,6332	3,0833	1,4546	7,7014	1,0954	10,2268

Приложение М. Результаты оптимизации алгоритма Гаусса – Зейделя для решения задачи Дирихле уравнения Пуассона. Размерность задачи – 256 x 4000 x 4000. Время выполнения исходного алгоритма – 14,7464 сек.

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=8, d2=10, d3=10	6,6036	2,2331	2,4754	5,9572	2,1984	6,7078
d1=8, d2=16, d3=16	5,6432	2,6131	1,9900	7,4103	1,6332	9,0291
d1=8, d2=20, d3=20	5,4744	2,6937	1,8882	7,8098	1,5018	9,8192
d1=8, d2=25, d3=25	5,2984	2,7832	1,8036	8,1761	1,3874	10,6288
d1=8, d2=40, d3=40	5,2500	2,8088	1,7304	8,5220	1,2622	11,6831
d1=8, d2=50, d3=50	5,6920	2,5907	1,8718	7,8782	1,2606	11,6979
d1=8, d2=80, d3=80	5,7978	2,5434	1,9856	7,4267	1,2784	11,5350
d1=8, d2=100, d3=100	5,5558	2,6542	1,9322	7,6319	1,2156	12,1310
d1=8, d2=125, d3=125	5,5320	2,6657	1,9400	7,6012	1,2372	11,9192
d1=8, d2=200, d3=200	5,2584	2,8044	1,9364	7,6154	1,3740	10,7325
d1=8, d2=250, d3=250	5,8998	2,4995	2,2196	6,6437	1,5872	9,2908
d1=8, d2=400, d3=400	8,8298	1,6701	3,7856	3,8954	3,0806	4,7869
d1=8, d2=500, d3=500	10,9966	1,3410	4,5566	3,2363	4,5132	3,2674
d1=16, d2=10, d3=10	6,0710	2,4290	2,0548	7,1766	1,4888	9,9049
d1=16, d2=16, d3=16	5,3210	2,7714	1,7554	8,4006	1,2224	12,0635
d1=16, d2=20, d3=20	5,1196	2,8804	1,6864	8,7443	1,1416	12,9173
d1=16, d2=25, d3=25	4,9074	3,0049	1,6818	8,7682	1,0718	13,7585
d1=16, d2=40, d3=40	4,7818	3,0839	1,7286	8,5308	1,0104	14,5946
d1=16, d2=50, d3=50	4,8840	3,0193	1,7122	8,6125	1,0208	14,4459
d1=16, d2=80, d3=80	4,9918	2,9541	1,6940	8,7051	1,0464	14,0925
d1=16, d2=100, d3=100	4,8778	3,0232	1,6334	9,0280	1,0048	14,6760
d1=16, d2=125, d3=125	4,7492	3,1050	1,6280	9,0580	1,0152	14,5256
d1=16, d2=200, d3=200	4,8064	3,0681	1,6968	8,6907	1,2212	12,0753
d1=16, d2=250, d3=250	5,5062	2,6781	2,0058	7,3519	1,4014	10,5226
d1=16, d2=400, d3=400	8,4910	1,7367	3,6288	4,0637	2,9424	5,0117
d1=16, d2=500, d3=500	10,7186	1,3758	4,4416	3,3201	4,3848	3,3631
d1=32, d2=10, d3=10	5,8726	2,5111	1,8922	7,7933	1,1980	12,3092
d1=32, d2=16, d3=16	5,0912	2,8964	1,7478	8,4371	1,0608	13,9012
d1=32, d2=20, d3=20	4,8844	3,0191	1,7084	8,6317	0,9954	14,8145
d1=32, d2=25, d3=25	4,6906	3,1438	1,6300	9,0469	0,9358	15,7581
d1=32, d2=40, d3=40	4,5432	3,2458	1,5646	9,4250	**0,9048**	**16,2980**
d1=32, d2=50, d3=50	4,6364	3,1806	1,5578	9,4662	0,9138	16,1374
d1=32, d2=80, d3=80	4,5228	3,2605	1,4932	9,8757	0,9390	15,7044
d1=32, d2=100, d3=100	4,4520	3,3123	**1,4474**	**10,1882**	0,9258	15,9283

d1=32, d2=125, d3=125	**4,4396**	**3,3216**	1,4722	10,0166	0,9514	15,4997
d1=32, d2=200, d3=200	4,5980	3,2071	1,5816	9,3237	1,1592	12,7212
d1=32, d2=250, d3=250	5,3598	2,7513	1,9200	7,6804	1,3286	11,0992
d1=32, d2=400, d3=400	8,3652	1,7628	3,5510	4,1527	2,8920	5,0990
d1=32, d2=500, d3=500	10,5986	1,3914	4,3782	3,3681	4,3382	3,3992
d1=64, d2=10, d3=10	5,7636	2,5585	1,8998	7,7621	1,1132	13,2469
d1=64, d2=16, d3=16	5,0412	2,9252	1,6942	8,7040	0,9878	14,9285
d1=64, d2=20, d3=20	4,8296	3,0533	1,6378	9,0038	0,9416	15,6610
d1=64, d2=25, d3=25	4,6166	3,1942	1,5466	9,5347	0,8886	16,5951
d1=64, d2=40, d3=40	4,4606	3,3059	1,4792	9,9692	**0,8746**	**16,8607**
d1=64, d2=50, d3=50	4,5084	3,2709	1,4648	10,0672	0,8800	16,7573
d1=64, d2=80, d3=80	4,4298	3,3289	1,4222	10,3687	0,9098	16,2084
d1=64, d2=100, d3=100	4,3744	3,3711	**1,3832**	**10,6611**	0,8940	16,4949
d1=64, d2=125, d3=125	**4,3518**	**3,3886**	1,4126	10,4392	0,9240	15,9593
d1=64, d2=200, d3=200	4,5162	3,2652	1,5300	9,6382	1,1144	13,2326
d1=64, d2=250, d3=250	5,3552	2,7537	1,8884	7,8089	1,2954	11,3837
d1=64, d2=400, d3=400	8,3196	1,7725	3,5146	4,1958	2,8616	5,1532
d1=64, d2=500, d3=500	10,5514	1,3976	4,3442	3,3945	4,3030	3,4270
d1=128, d2=10, d3=10	5,7084	2,5833	1,9000	7,7613	1,0660	13,8334
d1=128, d2=16, d3=16	4,9804	2,9609	1,6392	8,9961	0,9586	15,3833
d1=128, d2=20, d3=20	4,7012	3,1367	1,5758	9,3580	0,9152	16,1128
d1=128, d2=25, d3=25	4,5128	3,2677	1,4864	9,9209	0,8636	17,0755
d1=128, d2=40, d3=40	4,4200	3,3363	1,4410	10,2334	**0,8736**	**16,8800**
d1=128, d2=50, d3=50	4,4638	3,3036	1,4152	10,4200	0,8744	16,8646
d1=128, d2=80, d3=80	4,3958	3,3547	1,3848	10,6488	0,8872	16,6213
d1=128, d2=100, d3=100	4,3396	3,3981	**1,3526**	**10,9023**	0,8764	16,8261
d1=128, d2=125, d3=125	**4,3230**	**3,4111**	1,3832	10,6611	0,8960	16,4580
d1=128, d2=200, d3=200	4,5656	3,2299	1,5396	9,5781	1,1008	13,3961
d1=128, d2=250, d3=250	5,3912	2,7353	1,8882	7,8098	1,2812	11,5098
d1=128, d2=400, d3=400	8,3318	1,7699	3,5000	4,2133	2,8520	5,1705
d1=128, d2=500, d3=500	10,5398	1,3991	4,3094	3,4219	4,3076	3,4233

Приложение Н. Результаты оптимизации алгоритма решения задачи теплопроводности. Размерность задачи – 128 x 4000 x 4000. Время выполнения исходного алгоритма – 4,3616 сек. Одинарная точность

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=8, d2=8, d3=8	11,0290	0,3955	3,7586	1,1604	2,9220	1,4927
d1=8, d2=16, d3=16	10,0932	0,4321	3,2642	1,3362	2,2604	1,9296
d1=8, d2=20, d3=20	9,9180	0,4398	3,2014	1,3624	2,1364	2,0416
d1=8, d2=25, d3=25	9,6334	0,4528	3,2200	1,3545	2,0576	2,1198
d1=8, d2=40, d3=40	9,7592	0,4469	3,3916	1,2860	2,0582	2,1191
d1=8, d2=50, d3=50	9,5846	0,4551	3,3140	1,3161	2,0090	2,1710
d1=8, d2=80, d3=80	9,6564	0,4517	3,2586	1,3385	2,0532	2,1243
d1=8, d2=100, d3=100	9,4622	0,4609	3,1472	1,3859	1,9946	2,1867
d1=8, d2=125, d3=125	9,3112	0,4684	3,1624	1,3792	2,0212	2,1579
d1=8, d2=200, d3=200	9,5252	0,4579	3,3358	1,3075	2,4834	1,7563
d1=8, d2=250, d3=250	10,5256	0,4144	3,7582	1,1606	2,6886	1,6223
d1=8, d2=400, d3=400	12,0472	0,3620	4,9656	0,8784	3,9226	1,1119
d1=16, d2=8, d3=8	10,4524	0,4173	3,3368	1,3071	2,2614	1,9287
d1=16, d2=16, d3=16	9,4436	0,4619	3,1962	1,3646	1,9466	2,2406
d1=16, d2=20, d3=20	9,3180	0,4681	3,1866	1,3687	1,8444	2,3648
d1=16, d2=25, d3=25	9,2750	0,4703	3,1338	1,3918	1,7972	2,4269
d1=16, d2=40, d3=40	9,5216	0,4581	3,1276	1,3946	1,8838	2,3153
d1=16, d2=50, d3=50	9,3256	0,4677	3,0698	1,4208	1,8480	2,3602
d1=16, d2=80, d3=80	9,3984	0,4641	3,0352	1,4370	1,9430	2,2448
d1=16, d2=100, d3=100	9,2562	0,4712	2,9460	1,4805	1,8940	2,3029
d1=16, d2=125, d3=125	9,2012	0,4740	3,0038	1,4520	2,0058	2,1745
d1=16, d2=200, d3=200	9,5042	0,4589	3,2334	1,3489	2,3856	1,8283
d1=16, d2=250, d3=250	10,4984	0,4155	3,6472	1,1959	2,5440	1,7145
d1=16, d2=400, d3=400	12,0922	0,3607	4,9288	0,8849	3,8544	1,1316
d1=32, d2=8, d3=8	10,0114	0,4357	3,3978	1,2837	2,0032	2,1773
d1=32, d2=16, d3=16	9,2376	0,4722	3,0604	1,4252	1,7322	2,5180
d1=32, d2=20, d3=20	9,1496	0,4767	2,9888	1,4593	1,7192	2,5370
d1=32, d2=25, d3=25	9,0912	0,4798	2,9796	1,4638	1,7120	2,5477
d1=32, d2=40, d3=40	9,3712	0,4654	2,9552	1,4759	1,7546	2,4858
d1=32, d2=50, d3=50	9,1958	0,4743	2,8910	1,5087	1,7750	2,4572
d1=32, d2=80, d3=80	9,3120	0,4684	2,9172	1,4951	1,8930	2,3041
d1=32, d2=100, d3=100	9,2534	0,4714	2,8660	1,5218	1,8764	2,3245
d1=32, d2=125, d3=125	9,1932	0,4744	2,9314	1,4879	1,9162	2,2762
d1=32, d2=200, d3=200	9,4746	0,4603	3,1762	1,3732	2,3286	1,8731
d1=32, d2=250, d3=250	10,5432	0,4137	3,6226	1,2040	2,5170	1,7329
d1=32, d2=400, d3=400	12,2364	0,3564	4,9600	0,8794	3,8764	1,1252

d1=64, d2=8, d3=8	9,9252	0,4394	3,3146	1,3159	1,8384	2,3725
d1=64, d2=16, d3=16	9,1564	0,4763	2,9296	1,4888	1,6682	2,6146
d1=64, d2=20, d3=20	9,0756	0,4806	2,8790	1,5150	1,6582	2,6303
d1=64, d2=25, d3=25	**9,0182**	**0,4836**	**2,8048**	**1,5550**	**1,6314**	**2,6735**
d1=64, d2=40, d3=40	9,3282	0,4676	2,8446	1,5333	1,7362	2,5122
d1=64, d2=50, d3=50	9,1878	0,4747	2,8134	1,5503	1,7614	2,4762
d1=64, d2=80, d3=80	9,3080	0,4686	2,8726	1,5183	2,0182	2,1611
d1=64, d2=100, d3=100	9,2454	0,4718	2,8728	1,5182	1,9604	2,2249
d1=64, d2=125, d3=125	9,1856	0,4748	2,9148	1,4964	1,9764	2,2068
d1=64, d2=200, d3=200	9,5240	0,4580	3,1748	1,3738	2,3042	1,8929
d1=64, d2=250, d3=250	10,6622	0,4091	3,6398	1,1983	2,4858	1,7546
d1=64, d2=400, d3=400	12,3684	0,3526	4,9820	0,8755	3,8562	1,1311

Приложение О. Результаты оптимизации алгоритма решения задачи теплопроводности. Размерность задачи – 128 x 4000 x 4000. Время выполнения исходного алгоритма – 5,5922 сек. Двойная точность

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=8, d2=8, d3=8	9,4570	0,5913	4,6166	1,2113	4,8124	1,1620
d1=8, d2=16, d3=16	7,9448	0,7039	3,3366	1,6760	3,1714	1,7633
d1=8, d2=20, d3=20	7,7142	0,7249	3,0852	1,8126	2,9332	1,9065
d1=8, d2=25, d3=25	7,2960	0,7665	2,8858	1,9378	2,6636	2,0995
d1=8, d2=40, d3=40	7,1430	0,7829	2,7134	2,0610	2,2954	2,4363
d1=8, d2=50, d3=50	7,0756	0,7903	2,6068	2,1452	2,2028	2,5387
d1=8, d2=80, d3=80	6,9786	0,8013	2,5350	2,2060	2,0796	2,6891
d1=8, d2=100, d3=100	6,8424	0,8173	2,4900	2,2459	2,0414	2,7394
d1=8, d2=125, d3=125	6,7670	0,8264	2,5326	2,2081	2,2388	2,4979
d1=8, d2=200, d3=200	7,0548	0,7927	2,7272	2,0505	2,4860	2,2495
d1=8, d2=250, d3=250	8,0200	0,6973	3,0168	1,8537	2,6640	2,0992
d1=8, d2=400, d3=400	9,4750	0,5902	4,0326	1,3867	3,3220	1,6834
d1=16, d2=8, d3=8	9,2696	0,6033	3,9126	1,4293	3,7848	1,4775
d1=16, d2=16, d3=16	7,5840	0,7374	2,9344	1,9057	2,4286	2,3026
d1=16, d2=20, d3=20	7,1834	0,7785	2,7448	2,0374	2,1608	2,5880
d1=16, d2=25, d3=25	6,9224	0,8078	2,5312	2,2093	1,9452	2,8749
d1=16, d2=40, d3=40	6,7964	0,8228	2,4016	2,3285	1,6950	3,2992
d1=16, d2=50, d3=50	6,6370	0,8426	2,3044	2,4267	1,6116	3,4700
d1=16, d2=80, d3=80	6,6466	0,8414	2,2686	2,4650	1,6040	3,4864
d1=16, d2=100, d3=100	6,6310	0,8433	2,2380	2,4987	1,5788	3,5421
d1=16, d2=125, d3=125	6,6218	0,8445	2,2776	2,4553	1,6300	3,4308
d1=16, d2=200, d3=200	6,9498	0,8047	2,4674	2,2664	1,9374	2,8864
d1=16, d2=250, d3=250	8,0202	0,6973	2,8440	1,9663	2,0664	2,7063
d1=16, d2=400, d3=400	9,6804	0,5777	4,0288	1,3881	3,1086	1,7989
d1=32, d2=8, d3=8	9,0538	0,6177	4,1020	1,3633	3,2462	1,7227
d1=32, d2=16, d3=16	7,3952	0,7562	2,7982	1,9985	2,0774	2,6919
d1=32, d2=20, d3=20	7,0292	0,7956	2,5640	2,1810	1,8908	2,9576
d1=32, d2=25, d3=25	6,7724	0,8257	2,3824	2,3473	1,7582	3,1806
d1=32, d2=40, d3=40	6,6822	0,8369	2,2404	2,4961	1,5648	3,5737
d1=32, d2=50, d3=50	6,5532	0,8534	2,1752	2,5709	1,4870	3,7607
d1=32, d2=80, d3=80	6,6550	0,8403	2,1716	2,5752	1,5318	3,6507
d1=32, d2=100, d3=100	6,5800	0,8499	2,1264	2,6299	1,4890	3,7557
d1=32, d2=125, d3=125	6,5608	0,8524	2,1802	2,5650	1,5246	3,6680
d1=32, d2=200, d3=200	7,0974	0,7879	2,4656	2,2681	1,9014	2,9411
d1=32, d2=250, d3=250	8,2492	0,6779	2,8718	1,9473	2,0290	2,7561
d1=32, d2=400, d3=400	9,8076	0,5702	4,0432	1,3831	3,1278	1,7879

d1=64, d2=8, d3=8	8,9676	0,6236	3,6890	1,5159	3,0994	1,8043
d1=64, d2=16, d3=16	7,2560	0,7707	2,6268	2,1289	2,0538	2,7229
d1=64, d2=20, d3=20	6,9996	0,7989	2,4756	2,2589	1,8468	3,0280
d1=64, d2=25, d3=25	6,7538	0,8280	2,2806	2,4521	1,6126	3,4678
d1=64, d2=40, d3=40	6,6532	0,8405	2,1318	2,6232	1,4256	3,9227
d1=64, d2=50, d3=50	6,5076	0,8593	2,0842	2,6831	**1,4152**	**3,9515**
d1=64, d2=80, d3=80	6,6764	0,8376	2,1352	2,6191	1,4792	3,7806
d1=64, d2=100, d3=100	6,6412	0,8420	2,1182	2,6401	1,4686	3,8078
d1=64, d2=125, d3=125	6,7530	0,8281	2,2074	2,5334	1,5300	3,6550
d1=64, d2=200, d3=200	7,3848	0,7573	2,5238	2,2158	1,9132	2,9230
d1=64, d2=250, d3=250	8,3466	0,6700	2,8824	1,9401	2,0300	2,7548
d1=64, d2=400, d3=400	9,7976	0,5708	4,0122	1,3938	3,1028	1,8023

Приложение П. Результаты оптимизации алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле без использования перестановки циклов внутри тайла. Размерность задачи – 256 x 2000 x 2000. Время выполнения исходного алгоритма – 7,0792 сек.

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=8, d2=8, d3=8	6,7660	1,0463	1,4876	4,7588	1,7146	4,1288
d1=8, d2=16, d3=16	5,4942	1,2885	1,1986	5,9062	1,2674	5,5856
d1=8, d2=20, d3=20	5,4692	1,2944	1,2152	5,8255	1,2380	5,7183
d1=8, d2=25, d3=25	5,6540	1,2521	1,3388	5,2877	1,3498	5,2446
d1=8, d2=40, d3=40	5,9012	1,1996	1,2368	5,7238	1,2228	5,7893
d1=8, d2=50, d3=50	5,9146	1,1969	1,2574	5,6300	1,2292	5,7592
d1=8, d2=80, d3=80	6,1352	1,1539	1,3926	5,0834	1,3186	5,3687
d1=8, d2=100, d3=100	6,1596	1,1493	1,4420	4,9093	1,3656	5,1839
d1=8, d2=125, d3=125	6,1834	1,1449	1,5558	4,5502	1,4378	4,9236
d1=8, d2=200, d3=200	6,2262	1,1370	1,8784	3,7687	1,8088	3,9138
d1=8, d2=250, d3=250	6,2556	1,1317	2,0308	3,4859	2,0408	3,4688
d1=8, d2=400, d3=400	6,2546	1,1318	2,6396	2,6819	2,7262	2,5967
d1=16, d2=8, d3=8	6,4054	1,1052	1,3502	5,2431	1,4040	5,0422
d1=16, d2=16, d3=16	5,0974	1,3888	1,1036	6,4146	1,0880	6,5066
d1=16, d2=20, d3=20	5,1142	1,3842	1,0674	6,6322	1,0708	6,6111
d1=16, d2=25, d3=25	5,4202	1,3061	1,2524	5,6525	1,1710	6,0454
d1=16, d2=40, d3=40	5,7480	1,2316	1,1378	6,2218	1,1112	6,3708
d1=16, d2=50, d3=50	5,7022	1,2415	1,1440	6,1881	1,1276	6,2781
d1=16, d2=80, d3=80	5,9950	1,1809	1,2754	5,5506	1,2246	5,7808
d1=16, d2=100, d3=100	6,0324	1,1735	1,3090	5,4081	1,2634	5,6033
d1=16, d2=125, d3=125	6,0552	1,1691	1,4116	5,0150	1,3148	5,3842
d1=16, d2=200, d3=200	6,1036	1,1598	1,7398	4,0690	1,6622	4,2589
d1=16, d2=250, d3=250	6,1344	1,1540	1,9026	3,7208	1,8846	3,7563
d1=16, d2=400, d3=400	6,1680	1,1477	2,5790	2,7449	2,6252	2,6966
d1=32, d2=8, d3=8	5,9962	1,1806	1,2906	5,4852	1,2582	5,6265
d1=32, d2=16, d3=16	4,8336	1,4646	1,0294	6,8770	0,9776	7,2414
d1=32, d2=20, d3=20	4,9458	1,4314	**0,9938**	**7,1234**	**0,9494**	**7,4565**
d1=32, d2=25, d3=25	5,2208	1,3560	1,1286	6,2726	1,0856	6,5210
d1=32, d2=40, d3=40	5,5802	1,2686	1,0792	6,5597	1,0428	6,7886
d1=32, d2=50, d3=50	5,6218	1,2592	1,0918	6,4840	1,0508	6,7370
d1=32, d2=80, d3=80	5,9376	1,1923	1,2192	5,8064	1,1724	6,0382
d1=32, d2=100, d3=100	5,9722	1,1854	1,2520	5,6543	1,1812	5,9932

d1=32, d2=125, d3=125	5,9982	1,1802	1,3862	5,1069	1,2318	5,7470
d1=32, d2=200, d3=200	6,0694	1,1664	1,7994	3,9342	1,5736	4,4987
d1=32, d2=250, d3=250	6,0946	1,1616	2,0248	3,4962	1,8130	3,9047
d1=32, d2=400, d3=400	6,1156	1,1576	2,5976	2,7253	2,5676	2,7571
d1=64, d2=8, d3=8	5,7648	1,2280	1,2428	5,6962	1,2008	5,8954
d1=64, d2=16, d3=16	**4,6990**	**1,5065**	0,9724	7,2801	0,9270	7,6367
d1=64, d2=20, d3=20	4,9396	1,4332	**0,9486**	**7,4628**	**0,9000**	**7,8658**
d1=64, d2=25, d3=25	5,1504	1,3745	1,0372	6,8253	0,9918	7,1377
d1=64, d2=40, d3=40	5,5112	1,2845	1,0514	6,7331	1,0040	7,0510
d1=64, d2=50, d3=50	5,5576	1,2738	1,0622	6,6647	1,0122	6,9939
d1=64, d2=80, d3=80	5,8690	1,2062	1,1886	5,9559	1,1204	6,3185
d1=64, d2=100, d3=100	5,9184	1,1961	1,2186	5,8093	1,1528	6,1409
d1=64, d2=125, d3=125	5,9660	1,1866	1,3108	5,4007	1,1926	5,9359
d1=64, d2=200, d3=200	6,0332	1,1734	1,6358	4,3277	1,5094	4,6901
d1=64, d2=250, d3=250	6,0618	1,1678	1,8242	3,8807	1,7738	3,9910
d1=64, d2=400, d3=400	6,0904	1,1624	2,6122	2,7101	2,5304	2,7977

Приложение Р. Результаты оптимизации алгоритма Гаусса – Зейделя для решения обобщённой задачи Дирихле без использования циклов внутри тайла. Размерность задачи – 256 x 2000 x 2000. Время выполнения исходного алгоритма – 7,0792 сек.

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=8, d2=8, d3=8	3,8438	1,8417	1,5742	4,4970	1,3842	5,1143
d1=8, d2=16, d3=16	3,0396	2,3290	1,1900	5,9489	0,9198	7,6965
d1=8, d2=20, d3=20	2,9488	2,4007	1,1382	6,2196	0,8644	8,1897
d1=8, d2=25, d3=25	2,8794	2,4586	1,1120	6,3662	0,8232	8,5996
d1=8, d2=40, d3=40	2,7186	2,6040	1,0380	6,8200	0,7430	9,5279
d1=8, d2=50, d3=50	2,5858	2,7377	1,0378	6,8214	0,6832	10,3618
d1=8, d2=80, d3=80	2,5668	2,7580	1,0506	6,7382	0,7034	10,0643
d1=8, d2=100, d3=100	2,9888	2,3686	1,2056	5,8719	0,9218	7,6798
d1=8, d2=125, d3=125	3,0824	2,2967	1,3262	5,3380	0,9420	7,5151
d1=8, d2=200, d3=200	3,6452	1,9421	1,7326	4,0859	1,3060	5,4205
d1=8, d2=250, d3=250	7,5008	0,9438	3,5832	1,9757	3,2012	2,2114
d1=8, d2=400, d3=400	17,5180	0,4041	9,9462	0,7117	9,9688	0,7101
d1=16, d2=8, d3=8	3,4228	2,0682	1,2738	5,5575	0,9126	7,7572
d1=16, d2=16, d3=16	2,7528	2,5716	1,0234	6,9173	0,6754	10,4815
d1=16, d2=20, d3=20	2,6374	2,6842	0,9596	7,3772	0,6432	11,0062
d1=16, d2=25, d3=25	2,5314	2,7966	0,9518	7,4377	0,6272	11,2870
d1=16, d2=40, d3=40	2,2586	3,1343	0,8636	8,1973	0,5634	12,5651
d1=16, d2=50, d3=50	2,1962	3,2234	0,8702	8,1351	0,5170	13,6928
d1=16, d2=80, d3=80	2,3792	2,9755	0,9228	7,6714	0,5750	12,3117
d1=16, d2=100, d3=100	2,8310	2,5006	1,0856	6,5210	0,8320	8,5087
d1=16, d2=125, d3=125	2,9000	2,4411	1,1652	6,0755	0,7886	8,9769
d1=16, d2=200, d3=200	3,5166	2,0131	1,6200	4,3699	1,1822	5,9882
d1=16, d2=250, d3=250	7,4560	0,9495	3,5158	2,0135	3,1352	2,2580
d1=16, d2=400, d3=400	17,5300	0,4038	9,8600	0,7180	9,8730	0,7170
d1=32, d2=8, d3=8	3,2572	2,1734	1,1092	6,3823	0,7176	9,8651
d1=32, d2=16, d3=16	2,5330	2,7948	0,9308	7,6055	0,5542	12,7737
d1=32, d2=20, d3=20	2,3682	2,9893	0,8794	8,0500	0,5098	13,8862
d1=32, d2=25, d3=25	2,2352	3,1671	0,8514	8,3148	0,4790	14,7791
d1=32, d2=40, d3=40	2,1264	3,3292	**0,7726**	**9,1628**	0,4574	15,4770
d1=32, d2=50, d3=50	**2,1008**	**3,3698**	0,7788	9,0899	**0,4480**	**15,8018**
d1=32, d2=80, d3=80	2,2680	3,1213	0,8568	8,2624	0,5254	13,4739
d1=32, d2=100, d3=100	2,7542	2,5703	1,0056	7,0398	0,7120	9,9427
d1=32, d2=125, d3=125	2,8288	2,5025	1,0926	6,4792	0,7044	10,0500
d1=32, d2=200, d3=200	3,5012	2,0219	1,5872	4,4602	1,1486	6,1633

d1=32, d2=250, d3=250	7,4778	0,9467	3,4938	2,0262	3,1056	2,2795
d1=32, d2=400, d3=400	17,5038	0,4044	9,8414	0,7193	9,8196	0,7209
d1=64, d2=8, d3=8	3,1640	2,2374	1,2426	5,6971	0,6654	10,6390
d1=64, d2=16, d3=16	2,4628	2,8745	0,9606	7,3696	0,5016	14,1132
d1=64, d2=20, d3=20	2,3162	3,0564	0,8826	8,0208	0,4628	15,2965
d1=64, d2=25, d3=25	2,2396	3,1609	0,8180	8,6543	0,4494	15,7526
d1=64, d2=40, d3=40	2,1300	3,3236	**0,7454**	**9,4972**	0,4528	15,6343
d1=64, d2=50, d3=50	**2,0958**	**3,3778**	0,7538	9,3914	**0,4336**	**16,3266**
d1=64, d2=80, d3=80	2,2746	3,1123	0,8356	8,4720	0,4992	14,1811
d1=64, d2=100, d3=100	2,7286	2,5944	0,9762	7,2518	0,6790	10,4259
d1=64, d2=125, d3=125	2,8570	2,4778	1,0876	6,5090	0,6944	10,1947
d1=64, d2=200, d3=200	3,5264	2,0075	1,5882	4,4574	1,1418	6,2000
d1=64, d2=250, d3=250	7,5166	0,9418	3,4826	2,0327	3,0920	2,2895
d1=64, d2=400, d3=400	17,4582	0,4055	9,7852	0,7235	9,7784	0,7240

Приложение С. Результаты оптимизации программы (листинг 3.8.1). Размерность задачи – 256 x 4000 x 4000 при помощи алгоритма оптимизации итерационных гнёзд циклов, без применения преобразований «линеаризация» и «вынос общих инвариантных выражений». Время выполнения исходного алгоритма – 25,24 сек.

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=8, d2=8, d3=8	17,7162	1,4247	5,7136	4,4175	3,8412	6,5709
d1=8, d2=16, d3=16	18,6602	1,3526	6,4462	3,9155	3,5108	7,1892
d1=8, d2=20, d3=20	20,2906	1,2439	7,0064	3,6024	3,6804	6,8580
d1=8, d2=25, d3=25	21,4704	1,1756	7,2090	3,5012	3,7904	6,6589
d1=8, d2=40, d3=40	23,0608	1,0945	7,4008	3,4104	4,0392	6,2488
d1=8, d2=50, d3=50	23,4802	1,0749	7,5126	3,3597	4,2036	6,0044
d1=8, d2=80, d3=80	24,0730	1,0485	7,7154	3,2714	4,4302	5,6973
d1=8, d2=100, d3=100	24,3018	1,0386	7,7402	3,2609	4,4000	5,7364
d1=8, d2=125, d3=125	24,4550	1,0321	7,9918	3,1582	4,6306	5,4507
d1=8, d2=200, d3=200	24,7810	1,0185	8,4264	2,9953	5,7162	4,4155
d1=8, d2=250, d3=250	24,8714	1,0148	8,5824	2,9409	5,2732	4,7865
d1=8, d2=400, d3=400	24,9610	1,0112	9,8882	2,5525	7,2002	3,5055
d1=16, d2=8, d3=8	17,1766	1,4694	5,8740	4,2969	3,3052	7,6365
d1=16, d2=16, d3=16	18,0176	1,4009	5,9456	4,2452	3,2080	7,8678
d1=16, d2=20, d3=20	19,9402	1,2658	6,3154	3,9966	3,4228	7,3741
d1=16, d2=25, d3=25	20,8788	1,2089	6,5800	3,8359	3,5656	7,0788
d1=16, d2=40, d3=40	22,6202	1,1158	6,9416	3,6360	3,8056	6,6323
d1=16, d2=50, d3=50	23,1138	1,0920	7,1584	3,5259	3,9878	6,3293
d1=16, d2=80, d3=80	23,8652	1,0576	7,4818	3,3735	4,3352	5,8221
d1=16, d2=100, d3=100	24,1130	1,0467	7,5380	3,3484	4,3266	5,8337
d1=16, d2=125, d3=125	24,2944	1,0389	7,8056	3,2336	4,5574	5,5382
d1=16, d2=200, d3=200	24,6672	1,0232	8,3018	3,0403	5,6358	4,4785
d1=16, d2=250, d3=250	24,7920	1,0181	8,4796	2,9766	5,2374	4,8192
d1=16, d2=400, d3=400	24,8914	1,0140	9,8088	2,5732	7,1364	3,5368
d1=32, d2=8, d3=8	**16,9264**	**1,4912**	5,6614	4,4583	3,0662	8,2317
d1=32, d2=16, d3=16	17,7380	1,4229	5,5432	4,5533	3,0300	8,3300
d1=32, d2=20, d3=20	19,3558	1,3040	5,9440	4,2463	3,2354	7,8012
d1=32, d2=25, d3=25	20,5810	1,2264	6,3780	3,9574	3,3938	7,4371
d1=32, d2=40, d3=40	22,3592	1,1288	6,7232	3,7542	3,6978	6,8257
d1=32, d2=50, d3=50	22,9080	1,1018	6,9844	3,6138	3,9048	6,4638
d1=32, d2=80, d3=80	23,7416	1,0631	7,3586	3,4300	4,2770	5,9013
d1=32, d2=100, d3=100	24,0060	1,0514	7,4320	3,3961	4,3002	5,8695

d1=32, d2=125, d3=125	24,2132	1,0424	7,7108	3,2733	4,5432	5,5556
d1=32, d2=200, d3=200	24,6218	1,0251	8,2442	3,0615	5,5544	4,5441
d1=32, d2=250, d3=250	24,7282	1,0207	8,4310	2,9937	5,1932	4,8602
d1=32, d2=400, d3=400	24,8532	1,0156	9,7634	2,5852	7,1012	3,5543
d1=64, d2=8, d3=8	16,9576	1,4884	**5,3404**	**4,7262**	2,9414	8,5809
d1=64, d2=16, d3=16	17,7314	1,4235	5,3416	4,7252	**2,9314**	**8,6102**
d1=64, d2=20, d3=20	19,3206	1,3064	5,7756	4,3701	3,1822	7,9316
d1=64, d2=25, d3=25	20,5080	1,2307	6,1070	4,1330	3,3298	7,5800
d1=64, d2=40, d3=40	22,2508	1,1343	6,6168	3,8145	3,6580	6,8999
d1=64, d2=50, d3=50	22,8500	1,1046	6,8954	3,6604	3,8596	6,5395
d1=64, d2=80, d3=80	23,6908	1,0654	7,2906	3,4620	4,2282	5,9694
d1=64, d2=100, d3=100	23,9470	1,0540	7,3744	3,4227	4,2462	5,9441
d1=64, d2=125, d3=125	24,1666	1,0444	7,6644	3,2931	4,4584	5,6612
d1=64, d2=200, d3=200	24,5934	1,0263	8,2166	3,0718	5,5464	4,5507
d1=64, d2=250, d3=250	24,7836	1,0184	8,4040	3,0033	5,1762	4,8762
d1=64, d2=400, d3=400	24,8362	1,0163	9,7334	2,5931	7,0776	3,5662

Приложение Т. Результаты оптимизации программы (листинг 3.8.1). Размерность задачи – 256 x 4000 x 4000 при помощи алгоритма оптимизации итерационных гнёзд циклов, с использованием преобразований «линеаризация» и «вынос общих инвариантных выражений». Время выполнения исходного алгоритма – 25,24 сек.

Число потоков	Последовательно		8 потоков		16 потоков	
Размеры блоков	Время выполнения	Ускорение	Время выполнения	Ускорение	Время выполнения	Ускорение
d1=8, d2=8, d3=8	12,7142	1,9852	4,1744	6,0464	3,1066	8,1246
d1=8, d2=16, d3=16	11,7732	2,1439	3,6812	6,8565	2,4042	10,4983
d1=8, d2=20, d3=20	11,7976	2,1394	3,8926	6,4841	2,3074	10,9387
d1=8, d2=25, d3=25	11,8738	2,1257	4,2266	5,9717	2,2512	11,2118
d1=8, d2=40, d3=40	13,0316	1,9368	4,4774	5,6372	2,3746	10,6292
d1=8, d2=50, d3=50	13,3272	1,8939	4,4722	5,6438	2,4812	10,1725
d1=8, d2=80, d3=80	13,7334	1,8379	4,5160	5,5890	2,6430	9,5498
d1=8, d2=100, d3=100	13,8546	1,8218	4,5274	5,5749	2,6286	9,6021
d1=8, d2=125, d3=125	13,9444	1,8100	4,6604	5,4158	2,6950	9,3655
d1=8, d2=200, d3=200	14,1864	1,7792	4,9080	5,1426	3,3442	7,5474
d1=8, d2=250, d3=250	14,2302	1,7737	4,9884	5,0597	3,0684	8,2258
d1=8, d2=400, d3=400	14,2732	1,7683	5,7162	4,4155	4,1732	6,0481
d1=16, d2=8, d3=8	12,1186	2,0827	3,8052	6,6330	2,5524	9,8887
d1=16, d2=16, d3=16	11,1228	2,2692	3,8708	6,5206	2,1220	11,8944
d1=16, d2=20, d3=20	11,0162	2,2912	3,7892	6,6610	2,0544	12,2858
d1=16, d2=25, d3=25	11,2652	2,2405	3,7784	6,6801	2,0328	12,4164
d1=16, d2=40, d3=40	12,5660	2,0086	4,0016	6,3075	2,2290	11,3235
d1=16, d2=50, d3=50	13,0192	1,9387	4,1190	6,1277	2,3230	10,8653
d1=16, d2=80, d3=80	13,5454	1,8634	4,3120	5,8534	2,5074	10,0662
d1=16, d2=100, d3=100	13,6658	1,8469	4,3428	5,8119	2,5062	10,0710
d1=16, d2=125, d3=125	13,7836	1,8312	4,4940	5,6164	2,6334	9,5846
d1=16, d2=200, d3=200	14,0812	1,7925	4,7874	5,2722	3,2578	7,7476
d1=16, d2=250, d3=250	14,1498	1,7838	4,8864	5,1654	3,0250	8,3438
d1=16, d2=400, d3=400	14,2200	1,7750	5,6348	4,4793	4,1092	6,1423
d1=32, d2=8, d3=8	11,9234	2,1168	4,1366	6,1016	2,2804	11,0682
d1=32, d2=16, d3=16	10,9308	2,3091	3,5384	7,1332	1,9876	12,6987
d1=32, d2=20, d3=20	10,7774	2,3419	3,4792	7,2545	1,9392	13,0157
d1=32, d2=25, d3=25	10,9964	2,2953	3,4788	7,2554	1,9178	13,1609
d1=32, d2=40, d3=40	12,2950	2,0529	3,7724	6,6907	2,1034	11,9996
d1=32, d2=50, d3=50	12,7348	1,9820	3,9308	6,4211	2,2228	11,3551
d1=32, d2=80, d3=80	13,3828	1,8860	4,1892	6,0250	2,4510	10,2978
d1=32, d2=100, d3=100	13,5592	1,8615	4,2440	5,9472	2,4510	10,2978

d1=32, d2=125, d3=125	13,7094	1,8411	4,4064	5,7280	2,5884	9,7512
d1=32, d2=200, d3=200	14,0304	1,7990	4,7300	5,3362	3,2130	7,8556
d1=32, d2=250, d3=250	14,0952	1,7907	4,8350	5,2203	2,9932	8,4324
d1=32, d2=400, d3=400	14,1778	1,7802	5,5952	4,5110	4,0740	6,1954
d1=64, d2=8, d3=8	11,8596	2,1282	3,8672	6,5267	2,1594	11,6884
d1=64, d2=16, d3=16	10,8396	2,3285	3,3656	7,4994	1,8868	13,3772
d1=64, d2=20, d3=20	**10,6954**	**2,3599**	**3,3132**	**7,6180**	**1,8454**	**13,6773**
d1=64, d2=25, d3=25	10,8718	2,3216	3,3280	7,5841	1,8514	13,6329
d1=64, d2=40, d3=40	12,2040	2,0682	3,6712	6,8751	2,0600	12,2524
d1=64, d2=50, d3=50	12,6360	1,9975	3,8518	6,5528	2,1858	11,5473
d1=64, d2=80, d3=80	13,3058	1,8969	4,1252	6,1185	2,6564	9,5016
d1=64, d2=100, d3=100	13,5048	1,8690	4,1860	6,0296	2,6354	9,5773
d1=64, d2=125, d3=125	13,6578	1,8480	4,3558	5,7946	2,7588	9,1489
d1=64, d2=200, d3=200	13,9954	1,8035	4,6994	5,3709	3,1564	7,9965
d1=64, d2=250, d3=250	14,0700	1,7939	4,8092	5,2483	2,9744	8,4857
d1=64, d2=400, d3=400	14,1546	1,7832	5,5660	4,5347	4,0538	6,2263